Projeto e Seminário - Ano Letivo 2019/2020

Relatório de Progresso

Daniel dos Santos Cabral – 40569 – LI61N

Orientador: Eng. Carlos Martins

cmartins@cc.isel.ipl.pt

Projeto IQueue

1 - Requisitos

Este projeto consiste na implementação de um sistema de gestão de senhas virtuais para atendimento em diversos serviços aderentes.

Envolve o desenvolvimento de três 3 componentes fundamentais:

- API REST Servidora
- Aplicação *Android*
- Aplicação *Web*

1.1 - REST API e Aplicação WEB

- Deve suportar a configuração de vários operadores, cada um com filas de espera personalizadas.
- Deve suportar três perfis de utilizadores:
    - Um perfil de administrador *master*, que pode configurar os vários operadores, definir utilizadores com perfil de gestão para cada operador e aceder ao *log* de erros do sistema. Quando é configurado um utilizador de gestão para um operador, este recebe um *email* com a respectiva confirmação e credenciais de acesso.
    - Um perfil para utilizadores de atendimento, que só pode "chamar" as senhas, marcar o início e fim do atendimento assim como informar da desistência.
    - Um perfil para gestão do sistema, que pode:
        - Definir várias filas de espera para vários serviços, podendo definir um limite de senhas por dia para o serviço.
        - Adicionar/remover utilizadores com perfil de atendimento. Quando é configurado um utilizador de atendimento, recebe um *email* com a respectiva confirmação e credenciais de acesso.
        - Obter relatórios de: número de pessoas atendidas; tempo médio de espera; tempo médio de atendimento; número de pessoas que desistiram; classificações dos serviços, e; sugestões feitas pelos utilizadores.
        - Eliminar informação de clientes, visando cumprir o Regulamento Geral de Proteção de Dados Pessoais (RGPD).
- O sistema de *log* terá um automatismo para envio de *email* para o administrador *master* quando ocorrerem erros severos no sistema (e.g. *status code* na gama 500).

1.2 - Aplicação *Android*

- Terá um sistema de registo e de autenticação. Após registo, o utilizador poderá usar a aplicação para tirar senhas virtuais para os serviços dos operadores aderentes.
- Após *login*, quando entra no raio de alcance de um transmissor *beacon* pertencente a um operador aderente, abre *pop-up* perguntando se pretende "tirar senha".
- Ao confirmar, é desencadeado um pedido HTTPS ao servidor.
- Se responder "Não", o *pop-up* só volta a abrir quando sair e voltar a entrar no raio de alcance do *beacon*.
- Após tirar a senha:
    - Fica disponível a informação atualizada da senha que está a ser atendida no momento;
    - Tem como opção desistir da espera.
- Quando sai do alcance do transmissor *beacon*, é aberto *pop-up* a questionar se pretende desistir ou continuar à espera.
- Quando faltarem N senhas para ser atendido, recebe uma notificação.
- Quando chegar a sua vez, o utilizador recebe uma notificação (com o balcão onde será atendido, caso tenha sido configurado algum). A notificação pode ser repetida até ser considerado desistência e, nesse caso, recebe a notificação de desistência.
- Depois de ser atendido, o utilizador recebe um pedido para classificar o atendimento e, opcionalmente, para dar sugestões sobre o mesmo.
- Para tirar senha fora do raio de alcance do *beacon*, o utilizador:
    - Após *login,* tem uma opção para visualizar os operadores aderentes assim como os serviços que disponibilizam com a opção de tirar senha antecipadamente.
    - Quando entra no raio de alcance de um *beacon* pertencente a esse operador, já não será despoletado o *pop-up* para tirar senha.
    - Pode ver quantas senhas faltam para ser atendido mesmo antes de entrar no raio de alcance do *beacon*.
    - Recebe notificações quando faltarem N senhas para ser atendido.
    - Se ainda não estiver no raio de alcance do *beacon* quando chegar a sua vez, é despoletado um *pop-up* a questionar se pretende tirar nova senha ou desistir.

1.3 - Funcionalidades Gerais

- Ambas as *user interfaces* (*Android* e *Web*) devem suportar internacionalização, estando disponíveis pelo menos em dois idiomas (Português e Inglês).
- Usar *Transport Layer Security* (TLS) nas comunicações com o servidor.
- *Deployment* do servidor na *cloud* e da aplicação Web com suporte para balanceamento de carga.

2 - API REST

Foi utilizada uma abordagem "*database first"* definindo-se inicialmente as entidades e associações a criar na base de dados. Foi criado um *script* de criação da base de dados, através do qual são definidas as tabelas correspondentes às entidades e associações, para além de serem definidas e preenchidas as tabelas auxiliares de parametrização e criado o utilizador "*Master*" inicial.
Foi criada uma aplicação em *Java Spring*, onde foram definidos todos os pedidos HTTP que a API suporta e foi utilizado a aplicação *Postman* para testar esses pedidos.

Nas subsecções seguintes estes componentes são analisados em pormenor.

2.1 – Base de dados

2.1.1 - Entidades

Foram definidas as seguintes entidades:

Operator:

Esta entidade corresponde aos dados referentes às organizações que utilizam a aplicação nos seus serviços. Na Tabela 1 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
operatorId	int identity(1,1)	X	X	X
operatorDescription	varchar(100)		X	
email	varchar(100)			
telephoneNumber	varchar(15)			
address	varchar(200)			

Tabela 1 – Definição da entidade Operator.

Beacon:

Esta entidade corresponde à informação dos transmissores *beacon* disponíveis. Na Tabela 2 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
beaconId	int identity(1,1)	X	X	X
beaconMacAddress	varchar(12)		X	X
uidNamespaceId	varchar(10)		X	
uidInstanceId	varchar(6)		X	X
manufacturer	varchar(50)		X	
model	varchar(50)		X	

Tabela 2 – Definição da entidade Beacon.

User:

A entidade User é utilizada para representar a informação dos utilizadores da aplicação, ou seja, os colaboradores dos operadores e os clientes. Na Tabela 3 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
userId	int identity(1,1)	X	X	X
userName	varchar(100)		X	
email	varchar(100)		X	
telephoneNumber	varchar(15)			
address	varchar(200)			
userProfileId	int		X	

Tabela 3 – Definição da entidade User.

UserCredentials:
A entidade UserCredentials é uma entidade fraca da entidade User e tem como propósito armazenar a *password* encriptada utilizada para a autenticação dos utilizadores. Na Tabela 4 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
userId	int identity(1,1)	X	X	X
password	varchar(128)		X	

Tabela 4 – Definição da entidade UserCredentials.

ServiceQueue:
Esta entidade é utilizada para representar as filas de espera dos serviços dos operadores. Na Tabela 5 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
serviceQueueId	int identity(1,1)	X	X	X
operatorId	int		X	
serviceQueueDescription	varchar(100)		X	
serviceQueueTypeId	int		X	
dailyLimit	int			

Tabela 5 – Definição da entidade ServiceQueue.

Desk:

Esta entidade representa os balcões disponíveis para o atendimento dos serviços dos operadores. Na Tabela 6 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
deskId	int identity(1,1)	X	X	X
serviceQueueId	int		X	
deskDescription	varchar(50)			

Tabela 6 – Definição da entidade ServiceQueueDesk.

Attendance:

A entidade Attendance é utilizada para representar o atendimento efetuado pelo utilizador do operador ao cliente do serviço do operador. Na Tabela 7 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
attendanceId	int identity(1,1)	X	X	X
serviceQueueId	int		X	
deskId	int		X	
clientId	int		X	
startWaitingdateTime	datetime		X	
startAttendancedateTime	datetime			
endAttendancedateTime	datetime			
attendanceStatusId	int		X	

Tabela 7 – Definição da entidade Attendance.

AttendanceClassification:

Esta é uma entidade fraca da entidade Attendance e representa a classificação que o cliente do serviço do operador atribui ao atendimento que lhe foi efetuado. Na Tabela 8 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
attendanceId	int identity(1,1)	X	X	X
classificationCreationDateTime	datetime		X	
rate	int		X	
observations	varchar(200)			

Tabela 8 – Definição da entidade AttendanceClassification.

Log:

A entidade Log é utilizada para armazenar os detalhes dos pedidos e respostas efetuados à API. Na Tabela 9 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
logId	int identity(1,1)	X	X	X
logCreationDateTime	datetime		X	
requestMethod	varchar(6)		X	
requestUri	varchar(256)		X	
requestHeaders	varchar(1024)		X	
requestBody	varchar(1024)		X	
responseStatus	int		X	
responseHeaders	varchar(1024)		X	
responseBody	varchar(1024)		X	

Tabela 9 – Definição da entidade Log.

2.1.2 – Associações

Foram definidas as associações OperatorBeacon, OperatorUser e DeskUser que consistem na definição de tabelas com duas colunas, sendo cada coluna o identificador de cada entidade que faz parte da associação, operatorId e beaconId, operatorId e userId e deskId e userId, respectivamente.

2.1.3 – Tabelas auxiliares de parametrização

Foram definidas as tabelas auxiliares UserProfile, ServiceQueueType e AttendanceStatus assim como a tabela Language para permitir uma parametrização que suporte internacionalização. As suas definições técnicas encontram-se nas Tabelas 10, 11, 12 e 13.

Language:

Coluna	Tipo	Chave	Obrigatório	Único
languageId	int	X	X	X
languageDescription	varchar(20)		X	

Tabela 10 – Definição da tabela auxiliar Language.

UserProfile:

Coluna	Tipo	Chave	Obrigatório	Único
userProfileId	int	X	X	X
languageId	int		X	
userProfileDescription	varchar(50)		X	

Tabela 11 – Definição da tabela auxiliar UserProfile.

ServiceQueueType:

Coluna	Tipo	Chave	Obrigatório	Único
serviceQueueTypeId	int	X	X	X
languageId	int		X	
serviceQueueTypeDescription	varchar(50)		X	

Tabela 12 – Definição da tabela auxiliar ServiceQueueType.

AttendanceStatus:

Coluna	Tipo	Chave	Obrigatório	Único
attendanceStatusId	int	X	X	X
languageId	int		X	
attendanceStatusDescription	varchar(50)		X	

Tabela 13 – Definição da tabela auxiliar AttendanceStatus.

2.2 – Pedidos e Respostas HTTP

A API desenvolvida disponibiliza pedidos com quatro métodos HTTP distintos: GET, POST, PUT e DELETE.

Os pedidos GET, em caso de sucesso, obtêm uma resposta com *status* 200 cujo corpo contém a representação em formato JSON (*JavaScript Object Notation*) do estado do recurso identificado pelo respectivo URI alvo. Caso o recurso não exista, é retornada uma resposta com *status* 404 e caso existam outros problemas no processamento do pedido, é retornada uma resposta com *status* 500.
Os pedidos POST, em caso de sucesso, obtêm uma resposta com *status* 201 e criam um novo recurso cujo estado é definido pela representação presente no corpo do pedido. A representação JSON do recurso criado faz parte do corpo da resposta e é-lhe adicionado o *header Location* com o URI alvo do recurso criado. Caso o recurso já exista, é retornada uma resposta com *status* 409 e caso existam outros problemas no processamento do pedido, é retornada uma resposta com *status* 500.

Os pedidos PUT, em caso de sucesso, obtêm uma resposta com *status* 200 e alteram o estado do recurso identificado no pedido para reflectir a representação presente no pedido. A representação JSON do recurso alterado faz parte do corpo da resposta. Caso o recurso não exista, é retornada uma resposta com *status* 404 e caso existam outros problemas no processamento do pedido, é retornada uma resposta com *status* 500.

Os pedidos DELETE pedem ao servidor para eliminar o recurso apontado pelo URI alvo. Em caso de sucesso obtêm uma resposta com *status* 200. Caso o recurso não exista, é retornada uma resposta com *status* 404 e caso existam outros problemas no processamento do pedido, é retornada uma resposta com *status* 500.

Os pedidos GET, POST e PUT possuem o *header* Accept com o valor application/json, uma vez que os corpos das suas respostas têm esse formato. Nesse sentido, as respostas a estes três tipos de pedidos possuem o *header* Content-Type com o valor application/json. Os pedidos POST e PUT possuem também o *header* Content-Type com o valor application/json uma vez que o corpo desses pedidos tem esse formato.

De forma a compatibilizar a utilização da API com o mecanismo CORS (*Cross Origin Resource Sharing*, para mais informação acerca deste mecanismo consultar https://developer.mozilla.org/pt-PT/docs/Web/HTTP/CORS) dos *browsers* foram adicionados os *headers* Access-Control-Allow-Origin, Access-Control-Allow-Headers e Access-Control-Allow-Methods. Para tal foi definida a classe AccessFilter, que extende a classe abstrata org.springframework.web.filter.OncePerRequestFilter, cuja redefinição do seu método doFilterInternal permite a predefinição de *headers* comuns a todas as respostas da API.

Nas subsecções seguintes encontram-se os métodos HTTP e os URIs definidos para todos os pedidos suportados pela API.

2.2.1 – Language

GET /api/iqueue/language

GET /api/iqueue/language/{languageid}

POST /api/iqueue/language

PUT /api/iqueue/language/{languageid}

DELETE /api/iqueue/language/{languageid}

2.2.2 – UserProfile

GET /api/iqueue/userprofile?languageid={languageId}

GET /api/iqueue/userprofile/{userProfileId}?languageid={languageId}

POST /api/iqueue/userprofile

PUT /api/iqueue/userprofile/{userProfileId}

DELETE /api/iqueue/userprofile/{userProfileId}?languageid={languageId}

2.2.3 – User

GET /api/iqueue/user

GET /api/iqueue/user/{userId}

POST /api/iqueue/user

PUT /api/iqueue/user/{userId}

DELETE /api/iqueue/user/{userId}

2.2.4 – Operator

GET /api/iqueue/operator

GET /api/iqueue/operator/{operatorId}

POST /api/iqueue/operator

PUT /api/iqueue/operator/{operatorId}

DELETE /api/iqueue/operator/{operatorId}

2.2.5 – OperatorUser

GET /api/iqueue/operator/{operatorId}/user

GET /api/iqueue/operator/user/{userId}

GET /api/iqueue/operator/user/

POST /api/iqueue/operator/user/

DELETE /api/iqueue/operator/{operatorId}/user/{userId}

2.2.6 – Beacon

GET /api/iqueue/beacon

GET /api/iqueue/beacon/{beaconId}

POST /api/iqueue/beacon

PUT /api/iqueue/beacon/{beaconId }

DELETE /api/iqueue/beacon/{beaconId}

2.2.7 – OperatorBeacon

GET /api/iqueue/operator/{operatorId}/beacon

GET /api/iqueue/operator/{operatorId}/beacon/{beaconId}

GET /api/iqueue/operator/beacon

POST /api/iqueue/operator/beacon

DELETE /api/iqueue/operator/{operatorId}/beacon/{beaconId}

2.2.8 – ServiceQueueType

GET /api/iqueue/servicequeuetype?languageid={languageId}

GET /api/iqueue/servicequeuetype/{serviceQueueTypeId}?languageid={languageId}

POST /api/iqueue/servicequeuetype

PUT /api/iqueue/servicequeuetype/{serviceQueueTypeId}

DELETE /api/iqueue/servicequeuetype/{serviceQueueTypeId}?languageid={languageId}

2.2.9 – ServiceQueue

GET /api/iqueue/servicequeue

GET /api/iqueue/servicequeue/{serviceQueueId}

GET /api/iqueue/servicequeue?operatorId={operatorId}

POST /api/iqueue/servicequeue

PUT /api/iqueue/servicequeue/{serviceQueueId}

DELETE /api/iqueue/servicequeue/{serviceQueueId}

2.2.11 – AttendanceStatus

GET /api/iqueue/attendancestatus?languageid={languageId}

GET /api/iqueue/attendancestatus/{attendanceStatusId}?languageid={languageId}

POST /api/iqueue/attendancestatus

PUT /api/iqueue/attendancestatus/{attendanceStatusId}?languageid={languageId}

DELETE /api/iqueue/attendancestatus/{attendanceStatusId}?languageid={languageId}

2.2.12 – Desk

GET /api/iqueue/desk

GET /api/iqueue/desk?serviceQueueId={serviceQueueId}

POST /api/iqueue/desk

PUT /api/iqueue/{deskid}

DELETE /api/iqueue/{deskid}

2.2.13 – DeskUser

GET /api/iqueue/desk/{deskid}/user

GET /api/iqueue/desk/user/{userid}

POST /api/iqueue/operator/servicequeue/desk/user

DELETE /api/iqueue/desk/{deskid}/user/{userid}

2.2.14 – Attendance

GET /api/iqueue/attendance

GET /api/iqueue/attendance/{attendanceId}

POST /api/iqueue/attendance

PUT /api/iqueue/attendance/{attendanceId}

DELETE /api/iqueue/attendance/{attendanceId}

2.2.15 – AttendanceClassification

GET /api/iqueue/attendance/{attendanceId}/classification

GET /api/iqueue/attendance/classification

POST /api/iqueue/attendance/{attendanceId}/classification

DELETE /api/iqueue/attendance/{attendanceId}/classification

2.2.16 – UserCredentials

PUT /api/iqueue/user/{userId}/credentials

2.3 - Implementação

Foi implementado um modelo de objetos paralelo ao modelo da base de dados, isto é, um modelo de objetos de acesso a dados (DAO) e um conjunto de classes utilizadas para efetuar o mapeamento entre os dois modelos cuja implementação tem por base a interface DaoModelMapper. Para implementar os objetos DAO cuja chave na definição da tabela em SQL correspondente é uma chave composta, foi adicionado o pacote "*embeddable*" do qual fazem parte as classes utilizadas para representar as chaves compostas desses objetos.

A camada de acesso a dados foi implementada através da utilização da biblioteca *Spring Data JPA* (*Java Persistance API*), tendo sido criadas interfaces que estendem a interface org.springframework.data.jpa.repository.JpaRepository para cada um dos objetos DAO criados anteriormente.

Foi definida a classe abstrata e genérica Controller<M, K, D> sendo que o tipo M representa o tipo do objeto de domínio, o tipo K representa o tipo da chave do objeto DAO, e o tipo D representa o tipo do objeto DAO. Esta classe tem a definição genérica dos métodos utilizados para implementar os diferentes pedidos da API e possui como estado uma instância para uma implementação de org.springframework.data.jpa.repository.JpaRepository<D,K> e uma instância para uma implementação de DaoModelMapper<D,M>.

Foram implementadas classes com anotação RestController que estendem a classe Controller para cada um dos objetos de domínio, excepto os objetos Log e UserCredentials. Os pedidos responsáveis pela criação, atualização e eliminação de UserCredentials são efetuados no controlador responsável por criar e eliminar utilizadores, dispondo também de um método que implementa o pedido para alterar a *password* do utilizador.

Na criação de um novo utilizador, uma nova *password* é gerada automaticamente através da classe PasswordGenerator sendo enviado um *email* para o utilizador com a mesma. A *password* gerada é encriptada utilizando a classe org.springframework.security.crypto.bcrypt.BcryptPasswordEncoder. A funcionalidade de envio de *emails* foi implementada na classe EmailService.

Para a funcionalidade de *log in* foi criada a classe Controller independente para o efeito, sendo que nesta fase do projeto, a autenticação consiste apenas na comparação entre a *password* introduzida e a *password* armazenada. No decorrer do projeto será adicionalmente introduzida a utilização de JWT (*Json Web Tokens*) para autenticar cada pedido efetuado à API.

Para implementar a funcionalidade de Log foi criada a classe LogFilter que implementa a interface javax.servlet.Filter e onde no seu método doFilter se registam os detalhes de cada pedido e correspondente resposta à API.

Para activar o protocolo HTTPS, foi utilizada a ferramenta Keytool, através da qual foi gerada uma KeyStore que foi posteriormente configurada no ficheiro application.properties.

2.4 - Testes

Utilizando a aplicação *Postman*, foram implementados testes para todos os pedidos implementados.

Os testes consistem em verificar se cada pedido retorna o *status* esperado, assim como para os pedidos que retornam uma resposta com corpo, validar que o objeto retornado possui todas as propriedades esperadas preenchidas com os valores corretos.

Foram implementados testes unitários com recurso à biblioteca JUnit para testar o mecanismo de geração automática de novas passwords para os novos utilizadores e clientes, assim como para testar as classes de mapeamento entre os objetos de acesso a dados e os objetos de domínio.

3 – Aplicação Web

A aplicação web foi construída com a *framework Angular* 9 e com a biblioteca *Bootstrap*.

A solução consistiu na implementação de componentes *Angular* para as páginas de:

- Detalhes do operador e lista de operadores;
- Detalhes de utilizador e lista de utilizadores;
- Detalhes de *Beacon* e lista de *Beacons*;
- Detalhes de *ServiceQueue* e lista de *ServiceQueue*;
- Detalhes de *Desk* e lista de *Desk*;
- Associação de utilizador a operador e lista dos utilizadores do operador;
- Associação de *Beacon* a operador e lista dos *Beacons* do operador;
- Associação de utilizador a *Desk*;
- Mudança de *password*,
- Visualização da senha atual / atendimento.

O componente raiz da aplicação (app.component) possui o formulário de autenticação. Quando esta é efetuada com sucesso, o estado do componente é atualizado e é apresentada uma página com uma *navigation bar*. Os *links* disponíveis na *navigation bar* são apresentados de acordo com o perfil do utilizador que se autenticou, sendo que o *link* para alteração de *password* é comum a todos os utilizadores.

O roteamento é efetuado através da utilização da diretiva *RouterOutlet* que faz parte do pacote de roteamento nativo do Angular (@angular/router). De acordo com a documentação oficial da framework, o *RouterOutlet* funciona como um *placeholder* que o Angular preenche dinamicamente com base no estado atual do *router*. As rotas e as suas associações aos componentes foram definidas no módulo AppRoutingModule.

Foi definido o serviço HttpService, onde foram implementados os métodos necessários para efetuar os pedidos GET, POST, PUT e DELETE à API, e que é utilizado nos vários componentes.

Para activar o protocolo HTTPS foi gerado o certificado iqueue.crt e a correspondente chave iqueue.key tendo sido adicionados às opções de configuração da aplicação no ficheiro angular.json.