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.

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 um). 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, associações e entidades associativas a criar na base de dados. Foram criados dois scripts em SQL Server, um para definir todas as tabelas necessárias e outro para implementar todos as Stored Procedures.

Foi criada uma aplicação em Java Spring, onde foram definidos todos os pedidos HTTP que a API suporta e foi utilizado o software 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)			
phoneNumber	int			
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
ibeaconUuid	varchar(32)		X	
ibeaconMajor	int		X	X
ibeaconMinor	int		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. 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	
phoneNumber	int			
address	varchar(200)			
userProfileId	int		X	

Tabela 3 – Definição da entidade User.

Client:

Esta entidade é utilizada para representar os dados dos clientes dos operadores. Na Tabela 4 podemos ver os detalhes da sua definição técnica.

Coluna	Tipo	Chave	Obrigatório	Único
clientId	int identity(1,1)	X	X	X
clientName	varchar(100)		X	
email	varchar(100)		X	

Tabela 4 – Definição da entidade Client.

OperatorServiceQueue:

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 OperatorServiceQueue.

ServiceQueueDesk:

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
operatorId	int		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
operatorId	int		X	
serviceQueueId	int		X	
deskId	int		X	
clientId	int		X	
startWaitingTime	datetime		X	
endWaitingTime	datetime			
startAttendanceTime	datetime			
endAttendanceTime	datetime			
attendanceStatusId	int		X	
attendanceUserId	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
classificationCreationTime	datetime		X	
rate	int		X	
observations	varchar(200)			

Tabela 8 – Definição da entidade AttendanceClassification.

2.1.2 – Associações e Entidades Associativas

Foram definidas as associações OperatorBeacon e OperatorUser 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 e operatorId e userId, respectivamente.

Foi também definida a entidade associativa ServiceQueueDeskUser, cuja definição técnica se encontra na Tabela 9.

Coluna	Tipo	Chave	Obrigatório	Único
operatorId	int	X	X	X
serviceQueueId	Int	X	X	X
deskId	Int	X	X	X
userId	Int	X	X	X
date	date	X	X	X

Tabela 9 – Definição da entidade associativa ServiceQueueDeskUser.

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.

2.1.4 – *Stored Procedures*

Foram definidas *stored procedures* para executar as operações CRUD (*Create*, *Read*, *Update* e *Delete*) sobre as tabelas e associações definidas. Nas *stored procedures* que envolvem a alteração de estado (*Create*, *Update* e Delete) foi definido o nível de isolamento *Serializable* para garantir que múltiplas transações possam ocorrer independentemente sem interferência umas das outras.

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 secções 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 – OperatorServiceQueue

GET /api/iqueue/operator/{operadorId}/servicequeue
GET /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}
GET /api/iqueue/operator/servicequeue
POST /api/iqueue/operator/servicequeue
PUT /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}
DELETE /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}

2.2.10 – Client

GET /api/iqueue/client/{clientId}
GET /api/iqueue/client/
POST /api/iqueue/client
PUT /api/iqueue/client

DELETE /api/iqueue/client/{clientId}

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 – OperatorServiceQueueDesk

GET /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}/desk
GET /api/iqueue/operator/servicequeue/desk/
POST /api/iqueue/operator/servicequeue/desk
DELETE /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}/desk/{deskid}

2.2.13 – OperatorServiceQueueDeskUser

GET
/api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}/desk/{deskid}/user?date={date}
GET /api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}/desk/{deskid}/user/{userid}
GET /api/iqueue/operator/servicequeue/desk/user/{userid}?date={date}
POST /api/iqueue/operator/servicequeue/desk/user
DELETE
/api/iqueue/operator/{operadorId}/servicequeue/{serviceQueueId}/desk/{deskid}/user/{userid}?date={date}

2.2.14 – Attendance

GET
/api/iqueue/attendance?operatorid={operatorId}&servicequeueid={serviceQueueId}&deskid={deskid}&clientId={clientId}
GET /api/iqueue/attendance/{attendanceId}
GET /api/iqueue/attendance
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

2.3 - Implementação

Utilizando *Java Spring*, foi implementado um modelo de objetos paralelo ao modelo de dados da base de dados.

Utilizando a biblioteca JDBC Template, foi criada para cada um dos objetos de domínio, uma classe que implementa a interface org.springframework.jdbc.core.RowMapper. Através destas implementações, é feito o mapeamento entre as entidades da base de dados e os objetos de domínio da API.

Foi definida a classe abstrata e genérica Repository, que tem como estado:

- Uma referência para uma instância de org.springframework.jdbc.core.JDBCTemplate
- Uma referência para uma instância de org.springframework.jdbc.core.RowMapper
- Quatro referências para quatro instâncias de String que representam os Query Templates utilizados para ser executada a chamada às *stored procedures* para cada uma das operações CRUD.
- Esta classe abstrata é posteriormente estendida em concretizações específicas para cada objeto de domínio.

Foram implementadas classes com anotação RestController, cujo estado é uma referência para uma instância de Repository, correspondente à entidade que está a ser processada, e implementados os métodos necessários para executar os pedidos HTTP definidos no ponto 2.2.

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.