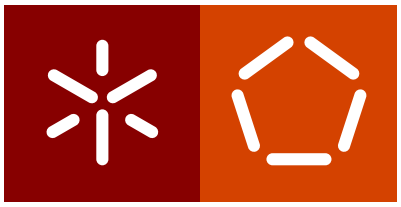
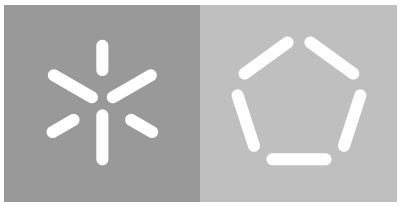


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel de Lima Pereira

Arquitetura orientada a serviços para suporte a um sistema de agendamentos online

outubro 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel de Lima Pereira

Arquitetura orientada a serviços para suporte a um sistema de agendamentos online

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
António Nestor Ribeiro

outubro 2016

Resumo

Título: *Arquitetura orientada a serviços para suporte a um sistema de agendamentos online*

A necessidade das empresas evoluírem as suas aplicações por forma a disponibilizarem mais recursos aos seus utilizadores é uma realidade da atualidade. A disponibilização de informação em tempo real é cada vez mais necessária, mesmo que isso implique a interação entre sistemas distintos, o que exige que essa comunicação seja completamente agnóstica de tecnologias.

Sendo uma das premissas da Q-Better - empresa que permitiu o desenvolvimento desta dissertação em contexto empresarial - proporcionar aos seus clientes uma melhor experiência de utilização aliado ao acompanhamento da evolução tecnológica, tornou-se imperativo a conceção de uma arquitetura que fornecesse suporte ao desenvolvimento de novas aplicações e também às já desenvolvidas, ainda que para tal seja necessária uma reformulação das mesmas.

Inicialmente foi feito um estudo sobre a temática das arquiteturas orientadas a serviços, incluindo os vários tipos de *web services* existentes, e também uma passagem pela temática da sincronização de dados para proporcionar a sincronização entre as várias aplicações da Q-Better.

A viabilidade da solução final - uma arquitetura orientada a serviços composta por um conjunto de *web services* REST - foi testada com a criação da aplicação *Bloom Appointments* cujo objetivo passa pela gestão de agendamentos a partir de qualquer dispositivo que tenha ligação à internet ou à rede onde o sistema esteja instalado.

Foi possível concluir que a escolha deste tipo de arquitetura se revelou acertada, uma vez que além de permitir a interoperabilidade entre os vários sistemas existentes na Q-Better, permite uma maior expansão não só da aplicação usada como *case study*, mas também de todo o *legacy software* e de futuras aplicações.

Palavras-chave: *Arquitetura orientada a serviços; SOA; Web services; REST; API;*

Abstract

Title: *A services oriented architecture to support an online scheduling system*

The market requirements increases the need of companies to update their applications in order to provide more resources to the users. The real time information availability is increasingly crucial even if it means interaction between different systems, which requires communication completely agnostic of technology.

One of Q-Better premises – enterprise which allows the development of this dissertation in business context – is to provide to their customers a better use experience allied to technologic evolution. For that, it has become imperative the conception of an architecture to support the development of new applications and support too the older ones, even if they needed to be reformulated.

Initially was realized a study about services oriented architectures, included different types of web services, and about data synchronization to support the synchronization between Q-Better applications.

The final solution's viability – one services oriented architecture composed by REST web services – was tested with the creation of *Bloom Appointments* application which the main goal is manage appointments from any device with internet or local network (in case of a local network installation) connection.

It was possible to conclude that the choice of this architecture was right because it allows the interoperability between different Q-Better systems, allows further expansion not only of the case study application but all legacy software and future applications too.

Keywords: *Services oriented architecture; SOA; Web services; REST; API;*

Agradecimentos

O início de qualquer caminhada é sempre repleto de medos, ansiedades mas sobretudo vontade de aprender, crescer e fazer melhor. Sabemos que o caminho irá ser difícil, cheio de contrariedades e desvios mas que a chegada à meta depende apenas de nós e daqueles que levamos connosco. Por isso, muito obrigada a todos aqueles que me ajudaram a atingir este objetivo, pois sozinho tal não seria possível.

Agradeço ao **Departamento de Informática da Universidade do Minho**, em particular a todos os professores que me acompanharam no decorrer do meu percurso académico.

Agradeço ao Professor **António Nestor Ribeiro**, orientador desta dissertação, pela disponibilidade, pelo apoio e orientações concedidas durante o desenvolvimento deste trabalho, fazendo com que encontrasse a melhor abordagem ao problema.

Agradeço à **Q-Better** por me ter permitido desenvolver todo o projeto, tornando-o também num produto vendável da empresa. Agradeço à direção pela compreensão e a toda a equipa de desenvolvimento pelo companheirismo, pelo acompanhamento, pela partilha de conhecimento e suporte durante as várias horas de trabalho.

Agradeço aos meus **Amigos** pelo incentivo que foram transmitindo ao longo dos vários meses e por me proporcionarem vários momentos de descontração.

Agradeço a toda a minha **Família** pela força e pela confiança.

De um modo especial, agradeço aos meus **Pais** por *TUDO*. Por me terem proporcionado este crescimento intelectual, por me terem apoiado e encorajado em todos os momentos, pelos valores que sempre me transmitiram, pela força, pelo carinho e pela energia que genuinamente me ofereceram e oferecem. É a eles a quem dedico este meu trabalho.

Agradeço de um modo ainda mais especial à **Juliana**. Por ter estado sempre presente, por ser a minha fonte de força e de energia, por me ter acompanhado em todos os momentos, por me ter apoiado e ajudado, por toda a paciência, por me ter transmitido conhecimento e coragem para enfrentar todos os momentos, principalmente quando a energia era escassa. Por todo o Amor. Por **TUDO**! É a ela também a quem dedico o meu trabalho.

Índice

Resumo **i**

Abstract **iii**

Agradecimentos **v**

Lista de Figuras **ix**

Listagens **xi**

Lista de Siglas e Acrónimos **xiii**

1 Introdução **1**

1.1 Objetivos 2

1.2 Estrutura do documento 3

2 Trabalho relacionado **5**

2.1 Arquitetura orientada a serviços 5

2.2 Web services 14

2.3 Web services RESTful 18

2.4 Bloom 20

2.5 Sincronização 26

3 Arquitetura proposta **29**

3.1 Frameworks 29

3.2 Detalhes da arquitetura 34

4 Análise de requisitos **41**

4.1 Levantamento de requisitos 41

4.2 Modelo de Domínio 50

4.3 Use cases 51

5 Desenvolvimento do caso de estudo **57**

5.1 Componente Servidor 57

5.2 Interface gráfica 67

6 Conclusões **81**

6.1 Trabalho futuro 84

Bibliografia **85**

Apêndices **89**

A Use cases **91**

B Diagrama da Base de Dados **93**

C API RESTful **95**

Lista de Figuras

2.1 Camadas da arquitetura SOA [1] 8

2.2 Exemplo de um processo de negócio 9

2.3 Componentes da arquitetura orientada a serviços 10

2.4 Funcionamento de um *web service* 15

2.5 QM-PAD durante um atendimento 22

2.6 Configuração de serviços, balcões, prioridades e associação de serviços com balcões no BQM . . . 23

2.7 Listagem de dispositivos adicionados a uma instalação do BQM 23

2.8 Diagrama de blocos do *Bloom* 25

3.1 Modelo de interação do padrão MVC [2] 30

3.2 Esquema lógico da arquitetura do *Bloom Appointments* 36

4.1 Visão dos *use cases* do sistema 45

4.2 Diagrama de sequência de alto nível da criação de um agendamento 47

4.3 Modelo de domínio do Bloom Appointments 50

4.4 Visão dos *use cases* do *Cliente* 52

4.5 Descrição textual do *use case* do Cliente: *Alterar dados do perfil* 53

4.6 Descrição textual do *use case* do Cliente: *Criar Agendamentos* 54

5.1 Exemplo de entidades do *Bloom Appointments* 58

5.2 Diagrama de sequência da integração do *Bloom Appointments* com o BQM 62

5.3 *API RESTful* para integração com o BQM 63

5.4 Painel para escolha de localizações por parte do cliente . . . 71
5.5 Painel para escolha de serviços por parte do cliente na sua página pessoal . . . 71
5.6 Painel para escolha de serviços por parte do cliente no dispensador de senhas . . . 72
5.7 Painel para escolha da data por parte do cliente . . . 72
5.8 Painel para escolha da hora por parte do cliente . . . 73
5.9 Painel para escolha da data por parte do cliente no dispensador de senhas . . . 73
5.10 Painel para escolha da hora por parte do cliente no dispensador de senhas . . . 74
5.11 Painel de confirmação do agendamento por parte do cliente . . . 74
5.12 Painel de confirmação do agendamento por parte do cliente no dispensador de senhas . . . 75
5.13 *Dashboard* do cliente com o agendamento criado . . . 75
5.14 *Dashboard* do cliente com a notificação de agendamento confirmado . . . 76
5.15 *Dashboard* do *Staff* . . . 77
5.16 Listagem de clientes registados no sistema . . . 77
5.17 Página de edição do serviço *Support* . . . 78
5.18 Vista do calendário de uma localização . . . 78
5.19 Vista detalhada de um agendamento num calendário . . . 79

A.1 Use Case do *Staff* . . . 91
A.2 Use Case do *Staff Admin* . . . 92

B.1 Diagrama da Base de Dados do *Bloom Appointments* . . . 94

C.1 *API RESTful* de ações relacionadas com os *Clientes* . . . 95
C.2 *API RESTful* de ações relacionadas com a criação de um agendamento (*Schedule*) . . . 95
C.3 *API RESTful* de ações relacionadas com o *Staff* . . . 96
C.4 *API RESTful* de ações relacionadas com o *Staff Admin* . . . 97

Listagens

2.1 Exemplo de um envelope SOAP [3] 16

2.2 Exemplo simplificado de um documento WSDL [4] 17

5.1 Código usado na criação da tabela (e respetiva classe) *customer* 59

5.2 Definição de *routing* do pedido POST para criação/sincronização de balcões do BQM no *Bloom Appointments* 65

5.3 Implementação do pedido POST para criação/sincronização de balcões do BQM no *Bloom Appointments* 66

5.4 Excerto da implementação da factory *StaffServices* num módulo de AngularJS 68

5.5 Código do botão para gravar serviços 69

5.6 Implementação na factory *StaffServices* do pedido POST para gravar serviços 69

5.7 Conteúdo do POST enviado para gravar informações sobre serviços 70

Lista de Siglas e Acrónimos

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

BES Bloom Enterprise Server

BQM Bloom Queue Management

CRUD Create, Read, Update, Delete

DAO Data Access Objects

DOM Document Object Model

FTP File Transfer Protocol

HATEOAS Hypermedia as the Engine of Application State

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IIOP Internet Inter-Orb Protocol

JSON Javascript Object Notation

MVC Model-View-Controller

OASIS Organization for the Advancement of Structural Information Standards

ORM Object-Relational Mapping

OSI Open Systems Interconnection

PDO PHP Data Objects

PHP PHP Hypertext Preprocesser

REST Representational State Transfer

RMI Java Remote Method Invocation

SMS Short Message Service

SMTP Simple Mail Transfer Protocol

SOAP Simple Object Access Protocol

SOA Service Oriented Architecture

SQL Structured Query Language

SSL Secure Sockets Layer

UDDI Universal Description Discovery and Integration

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

VPN Virtual Private Network

W3C World Wide Web Consortium

WSDL Web Services Description Language

XML eXtensible Markup Language

Capítulo 1

Introdução

O tecido empresarial na área das tecnologias de informação preocupa-se cada vez mais com o caminho a seguir no futuro. A constante evolução das tecnologias faz com que cada vez mais as empresas se preocupem em fornecer mais recursos para os seus clientes, aumentando também a produtividade dos seus quadros. Um dos problemas existentes no que diz respeito a este ramo da ciência prende-se no facto de muitas das empresas de produção de software continuarem a desenvolver soluções com base em *legacy software*, por forma a aproveitarem os vários anos de desenvolvimento e conhecimentos adquiridos outrora, sendo muitas vezes rentável o recurso a esta tática de gestão.

Deste modo, no desenvolvimento de novas soluções, a escolha das tecnologias e metodologias a adotar são sempre decisões sensíveis, uma vez que existe o receio da troca de paradigma, que quando é mal planeado e/ou mal desenvolvido, pode originar a perda de informações ou até mesmo provocar uma falha na transição para o novo paradigma. No entanto, além da redução do tempo de criação das soluções que permite uma redução nos encargos financeiros, a garantia de satisfação do cliente final é também um ponto de elevada relevância para a tomada de decisão.

A necessidade de ultrapassar os problemas enunciados e de se desenvolver uma base tecnológica sustentável para futuras aplicações fez com que surgisse o interesse por parte da Q-Better - empresa portuguesa reconhecida na área do *digital signage*[1] e na gestão de filas - no desenvolvimento de um novo projeto sustentado nesta dissertação.

O principal desafio da Q-Better passa por proporcionar uma melhor experiência quer para as empresas ou instituições, quer para os clientes. Assim tenta aliar a qualidade dos seus produtos a um conjunto de soluções desenvolvidas com recurso às novas tendências tecnológicas e com foco na experiência do seu uso.

[1] Painéis digitais colocados em espaços públicos com o principal objetivo de informar ou publicitar

Hoje em dia, empresas ou instituições orientadas ao atendimento de clientes confrontam-se com a possibilidade da existência de filas de espera nos seus serviços. Este efeito pode causar tensão e stress quer para os clientes, quer para os funcionários, baixando assim a eficiência dos serviços. Com o recurso a aplicações de gestão de filas de espera, os clientes ficam livres para usarem o tempo de outra forma enquanto aguardam pela sua vez. Além disso, possibilita à empresa ou instituição um conjunto de dados estatísticos que podem ser usados quer em tempo real no cálculo de estimativas de afluências a determinados serviços ou balcões, quer em dados previamente consolidados, permitindo gerir de melhor forma a alocação de recursos humanos.

Esta dissertação surge no âmbito da necessidade de criação de uma arquitetura que sirva de suporte ao desenvolvimento quer de novas aplicações, quer das aplicações previamente criadas na Q-Better, exigindo para tal uma readaptação das mesmas. Para testar a viabilidade desta arquitetura, será usada como *case study* uma aplicação *web* que permite fazer a gestão de agendamentos a partir de qualquer dispositivo com ligação à internet ou à rede onde o sistema esteja instalado. A solução do ponto de vista tecnológico passa pela conceção de uma arquitetura orientada a serviços, desenvolvendo para tal uma camada de *web services* capaz de dar resposta a todo o sistema de gestão de agendamentos, possibilitando a integração deste em interfaces gráficas distintas ou até mesmo com outros sistemas alheios à Q-Better. Esta aplicação deverá ter em conta o uso por parte de consumidores finais (quem faz os agendamentos) e o uso por parte de gestores e colaboradores das localizações onde os agendamentos podem ser feitos. Além do funcionamento em modo *standalone*, a aplicação deverá permitir a sincronização com soluções já existentes na Q-Better, tais como a solução de gestão de filas da Q-Better, o *Bloom Queue Management* (BQM) e o *Bloom Enterprise Server* (BES).

1.1 Objetivos

O objetivo principal desta dissertação passa pela criação e implementação de uma arquitetura orientada a serviços, através da criação de *web services*, por forma a suportar o desenvolvimento de novas soluções na empresa. Como *case study* será desenvolvida a aplicação *Bloom Appointments* usada para gestão de agendamentos e que também fará integração com soluções já existentes. Desta forma, estão definidos os seguintes objetivos da dissertação:

- Desenvolver um estudo sobre a temática das arquiteturas orientadas a serviços, identificando as suas características e várias formas de implementação;

- Analisar e recolher informações sobre vários tipos de *web services* existentes;
- Abordar a temática da sincronização entre sistemas distintos por forma a ter suporte no desenvolvimento da sincronização entre o *Bloom Appointments*, o BQM e o BES;
- Fazer um levantamento de requisitos para a implementação da arquitetura, por forma a perceber as competências e funcionalidades necessárias dos *web services* a desenvolver;
- Criar uma camada de *web services* que suportem todos os requisitos obtidos durante a análise das necessidades da aplicação;
- Conceber uma interface gráfica que permita fazer a gestão de agendamentos e que permita a interação do cliente final com a solução.

1.2 Estrutura do documento

O presente documento é composto por 6 capítulos e segue a seguinte estrutura:

- **Capítulo 2**: Neste capítulo é apresentado o estado atual da temática abordada nesta dissertação. São apresentados resultados da pesquisa sobre arquiteturas orientadas a serviços, sobre os *web services* como escolha fundamental para a implementação deste tipo de arquiteturas e sobre *web services RESTful*. É abordada também a sincronização entre sistemas e é feita uma introdução à *suite* de aplicações *Bloom*;
- **Capítulo 3**: A proposta para a implementação da arquitetura é abordada neste capítulo. É nele que estão concentrados todos os detalhes do sistema a desenvolver;
- **Capítulo 4**: É neste capítulo que é feito um levantamento de requisitos e um estudo da arquitetura existente, abordando assim toda a camada de negócio do projeto. Será feito também um estudo relativamente aos cenários a cobrir na implementação desta arquitetura;
- **Capítulo 5**: No penúltimo capítulo estão descritos os detalhes da implementação da solução abordada nos capítulos anteriores;

- **Capítulo 6**: Todas as conclusões sobre implementação do projeto estão detalhadas neste capítulo. É feita a relação e a avaliação entre os objetivos traçados e os objetivos realmente cumpridos. Por fim, são referenciadas também possíveis evoluções e mudanças tecnológicas a desenvolver no futuro.

Capítulo 2

Trabalho relacionado

Os métodos de desenvolvimento de aplicações, quer seja em pequena ou em grande escala, foram sofrendo alterações ao longo dos anos. O hábito de criar aplicações sem considerar o crescimento das redes de comunicações com outras aplicações, fez com que se tornasse imperativo desenvolver arquiteturas onde as mesmas comunicações sejam simplificadas e que a necessidade de manutenção seja menos prejudicial ao funcionamento do sistema.

As arquiteturas orientadas a serviços visam responder a esse problema, transformando as funcionalidades das aplicações em serviços com interfaces bem definidas para posteriormente poderem ser utilizados noutras aplicações, independentemente das plataformas utilizadas no desenvolvimento das mesmas.

Neste capítulo serão abordadas as temáticas da arquitetura orientada a serviços, os *web services* como implementação mais comum deste tipo de arquitetura e também uma abordagem sobre os *web services RESTful*, o exemplo mais comum de *web services*. Serão também introduzidas noções sobre sincronização entre sistemas distintos e também acerca da *suite* de aplicações *Bloom*, onde estará inserida esta arquitetura.

2.1 Arquitetura orientada a serviços

*"**Service Oriented Architecture** (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains"* [5]

2.1.1 Descrição

A arquitetura orientada a serviços trata-se de uma evolução da arquitetura de sistemas tradicional, utilizada para a adaptação das necessidades de desenvolvimento das novas normas do mercado. Esta abordagem permite a criação de serviços que comunicam entre si através de uma interface comum e que possam ser facilmente utilizados nas comunicações entre diferentes aplicações de vários sistemas, produzidos pela mesma empresa ou não, ou até mesmo utilizados na criação de serviços com complexidade superior. Os seus princípios fazem com que seja uma arquitetura amplamente utilizada no desenvolvimento de sistemas distribuídos de grande escala.

Normalmente, as entidades criam capacidade para dar resposta a uma necessidade. No entanto, dada a granularidade entre as necessidades e as capacidades, nem sempre existe uma correlação de um para um. Assim, uma necessidade pode requerer a combinação de várias capacidades enquanto que uma capacidade pode responder a mais que uma necessidade. O paradigma da SOA passa pelo desenvolvimento de uma framework capaz de relacionar necessidades e capacidades e de combinar as capacidades às necessidades correspondentes. Um **serviço** é um mecanismo capaz de aceder a uma ou mais capacidades, sendo o acesso exercido em conformidade com um conjunto de condições especificadas na descrição do serviço e feito com recurso a uma interface prescrita (*service interface*).

O fornecedor de serviços (*service provider*) é a entidade responsável por disponibilizar o serviço a outras entidades, mesmo que estas sejam desconhecidas por este. Considera-se que o serviço é opaco, uma vez que a sua implementação está oculta para o consumidor deste (*service consumer*), à exceção da informação existente na sua interface e da informação necessária para avaliar a utilidade do seu uso [5].

Segundo a OASIS (*Organization for the Advancement of Structured Information Standards*)[1] , existem três conceitos que podem ajudar a descrever o que está envolvido na interação entre serviços: a visibilidade entre os *service providers* e os *service consumers*, a interação entre eles e o efeito causados.

- **Visibilidade**: refere-se à aptidão das necessidades e das capacidades poderem-se observar/relacionar mutuamente. A visibilidade é garantida pelo fornecimento de alguns aspetos, tais como funções e requisitos técnicos, restrições e políticas e ainda por mecanismos para acesso ou resposta. As descrições serão apresentadas num formato com sintaxe e semânticas acessíveis e compreensíveis;

- **Interação**: trata-se da atividade de usar uma competência. A interação é tipicamente mediada pela troca de

[1] URL: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

mensagens e processa uma série de trocas de informação e ações invocadas. Apesar de existirem vários tipos de interação, todas são fundamentadas num contexto de execução, onde o conjunto de elementos técnicos e de negócio formam um *path* entre as necessidades e as capacidades, permitindo desta forma que os prestadores dos serviços e os consumidores possam interagir e fornecer um ponto de decisão para as políticas e contratos que possam estar em vigor;

- **Efeito**: trata-se do propósito de usar uma competência. O efeito pode ser o resultado de um pedido de informação, pode ser alteração de estados de entidades que estão envolvidas na interação ou a combinação dos dois [5].

A implementação desta arquitetura traz às empresas uma redução de custos através do aproveitamento de serviços já implementados noutros projetos e uma maior facilidade quer no planeamento das aplicações, quer no desenvolvimento das mesmas, uma vez que torna mais fácil a decisão de como implementar novas funcionalidades no sistema. Devido à baixa dependência entre os serviços, a resolução de problemas fica assim simplificada. *"Divide and conquer"* é também uma das máximas desta arquitetura, uma vez que o facto da estrutura dos serviços estar isolada, faz com que as alterações possam ser feitas com um grau de dificuldade menor.

No entanto, a implementação desta arquitetura obriga a ter um maior controlo no servidor e na rede onde os serviços são disponibilizados, uma vez que o seu desempenho pode limitar o acesso aos mesmos. É também imperativa a implementação de normas de segurança, uma vez que os serviços são disponibilizados na rede, podendo o sistema ser alvo de alterações inesperadas, o que pode levar ao bloqueio do mesmo.

2.1.2 Modelo arquitetural e componentes

Segundo *Ali Arsanjani* [1], uma arquitetura orientada a serviços pode ser dividida em sete camadas, representadas na Figura 2.1 e descritas de seguida.

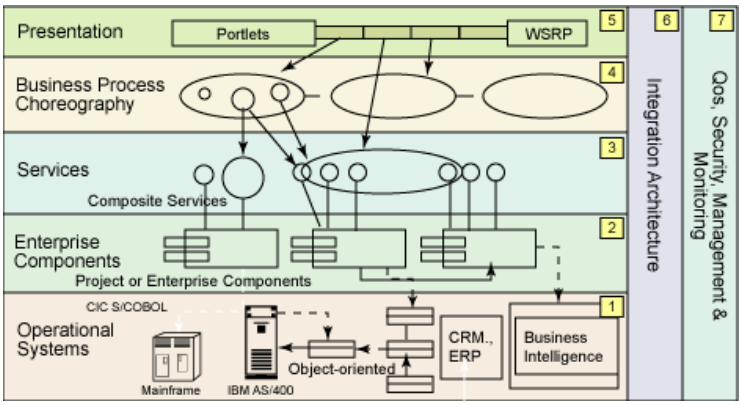


Figura 2.1: Camadas da arquitetura SOA [1]

- **Camada 1 (Sistemas operacionais)**: camada que contém o *legacy software* usado para integrar com sistemas mais atuais;

- **Camada 2 (Componentes)**: é a camada responsável por disponibilizar as funcionalidades de qualidade de serviço e está tipicamente implementada em servidores de aplicações;

- **Camada 3 (Serviços)**: é nesta camada que estão expostos os serviços que poderão ser invocados diretamente ou em conjunto com outros serviços em contextos específicos;

- **Camada 4 (Negócio)**: trata-se da camada onde os serviços podem ser agrupados por forma a que a chamada seja feita apenas através de um serviço. Na Figura 2.2 é possível observar a representação de um processo de negócio onde se demonstram as várias funções que um serviço pode desempenhar. Uma ou mais tarefas podem ser encapsuladas num serviço, podendo também dar origem a serviços que encapsulam outros serviços mais específicos, abrangendo assim diferentes quantidades de lógica;

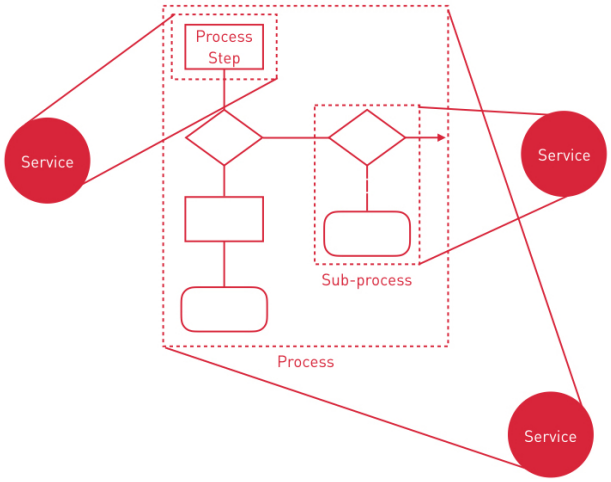


Figura 2.2: Exemplo de um processo de negócio

- **Camada 5 (Apresentação)**: Esta camada é responsável pela apresentação da informação ao utilizador;

- **Camada 6 (Integração)**: É a camada onde se podem encontrar tecnologias que tratam da integração de fim a fim;

- **Camada 7 (QoS)**: Trata-se da camada que providencia as capacidades necessárias para uma monitorização, gestão e manutenção da qualidade do serviço, incidindo sobre a segurança, desempenho e disponibilidade da arquitetura [1].

Dirk Krafzig, *Karl Banke* e *Dirk Slama*[6] descrevem que esta arquitetura é formada pelo *front end* das aplicações, pelo repositório e pelo barramento de serviços, pelos serviços e por tudo o que lhe é inerente, neste caso, o seu contrato, a sua interface e a própria implementação do serviço que inclui a lógica de negócio e os dados. Tal como demonstrado na Figura 2.3, estes elementos relacionam-se entre si de forma hierárquica e a sua função está descrita de seguida:

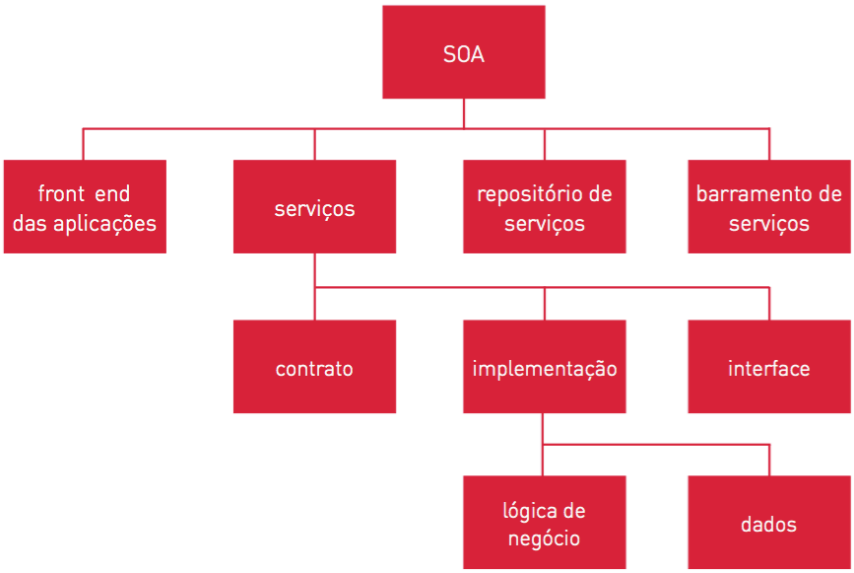


Figura 2.3: Componentes da arquitetura orientada a serviços

Front end **das aplicações**

É o responsável pela inicialização e controlo da execução dos serviços. O *front end* pode conter a interface gráfica onde existe interação direta dos utilizadores com a aplicação, ou pode ser um processo *batch* onde são feitas as invocações das funcionalidades ou eventos específicos.

Serviços

São componentes de software que representam um processo, uma entidade, uma atividade ou uma tarefa de negócio. Tratam-se de componentes de alto nível, orientados ao negócio (ex. "Atualizar o nome de um produto", "Criar novo utilizador").

Contrato

Fornece uma descrição agnóstica de tecnologias sobre o serviço, o seu modo de funcionamento, o seu propósito, contexto e possíveis restrições. Estes contratos são representados em linguagens baseadas em padrões abertos e estão disponibilizados nos repositórios dos serviços.

Interface

É o excerto de código disponibilizado ao utilizador do serviço de forma a que seja possível a abstração dos detalhes da implementação do serviço. O utilizador pode tratar-se de um utilizador de um *front end* de uma determinada aplicação ou pode ser outro serviço.

Implementação

É o elemento responsável pela execução do propósito do serviço. Uma vez que tem o acesso restringido ao consumidor do serviço, é possível (graças à não-dependência da interface) efetuar alterações sem causar impacto no uso do serviço. Para a execução dessas alterações, este elemento combina a camada de negócio com a camada de dados.

Lógica de negócio

Trata-se das regras de negócio envolvidas nas funcionalidades do serviço, usadas na execução do mesmo.

Dados

Informação que pode ser proveniente do consumidor do serviço caso a interface assim o exija, ou de aplicações de *back end*, tais como bases de dados ou outros sistemas de informação.

Repositório

Base de dados que fornece informações sobre os serviços de uma SOA. São responsáveis por informar os utilizadores sobre quais os serviços disponíveis e o modo de acesso descritos no contrato do serviço. Podem também fornecer informações sobre a localização física do serviço, custos de utilização, restrições ou até mesmo requisitos de segurança do serviço.

Barramento de serviços

Tal como o próprio nome indica, tecnicamente o barramento de serviços trata-se de um *middleware* que permite a intercomunicação dos vários componentes de uma SOA. Este componente pode ser constituído por várias tecnologias e permite abstrair a complexidade técnica existente nas camadas inferiores.

2.1.3 Princípios

Para tomar partido dos benefícios de uma arquitetura orientada a serviços, é necessário que a arquitetura esteja totalmente desenhada de uma forma consistente, por forma a que cada serviço dê a resposta pretendida ao seu objetivo. *Thomas Erl* [7] definiu 8 princípios para a criação de serviços que se transformaram em *guidelines* para o desenvolvimento de uma arquitetura deste género.

Baixa dependência

"Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment".

Serviços funcionam independentemente de outros serviços. Deste modo, os serviços devem ser coerentes e bem definidos, por forma que o acesso aos mesmos seja feito de um modo transparente. No entanto, não impede interação entre eles, tornando-se assim numa vantagem na utilização desta arquitetura.

Abstração

"Service contracts only contain essential information and information about services is limited to what is published in service contracts."

O desenvolvimento dos serviços deve tornar-se independente de linguagens, podendo a mesma ser alterada em qualquer momento, desde que não implique alterações na utilização deste. Além disso, a camada lógica não deve estar acessível aos consumidores dos serviços.

Composição

"Services are effective composition participants, regardless of the size and complexity of the composition."

Este princípio indica a possibilidade da existência de serviços que no decorrer da sua execução, possam juntar-se a outros serviços para a resolução de um determinado contexto de negócio. Desta forma, estes serviços não devem ter dependências e as suas operações devem ser o mais atómicas possíveis.

Contrato

"Services within the same service inventory are in compliance with the same contract design standards."

Tratam-se de documentos textuais que descrevem o *output* do serviço. Estes documentos existem em padrões tais como WSDL que é responsável por identificar o protocolo e o endereço no qual o serviço está publicado bem como os seus *inputs* e *outputs*, ou o padrão SOAP, utilizado pelos *web services*. São também responsáveis por definir o modelo da troca de mensagens.

Reutilização

"Services contain and express agnostic logic and can be positioned as reusable enterprise resources."

Um serviço deve possuir características no seu desenvolvimento que o torna genérico ao ponto de poder ser utilizado noutros projetos.

Autonomia

"Services exercise a high level of control over their underlying runtime execution environment."

O serviço possui autonomia suficiente para executar a sua lógica sem qualquer dependência de outros serviços.

Independência de estados

"Services minimize resource consumption by deferring the management of state information when necessary."

O serviço não deve armazenar informações de estados de outros pedidos por forma a poder aumentar a capacidade de reutilização. O facto de armazenar o estado de um pedido pode impedir o processamento de outros pedidos por parte de outros consumidores.

Localização

"Services are supplemented with communicative metadata by which they can be effectively discovered and interpreted."

Os serviços devem permitir que as suas interfaces sejam publicadas e tornadas visíveis para os consumidores dos serviços. Este princípio pressupõe que os serviços devem estar corretamente documentados, com a informação armazenada num repositório localizável e permitir que a pesquisa de informação sobre este seja encontrada de forma eficiente.

A aplicação de princípios como a *baixa dependência*, *abstração* e *composição* resultam em características que devem ser adicionadas aos serviços. Quando usados em conjunto com princípios reguladores como *contrato*, *reutilização*, *autonomia*, *independência de estados* e *localização*, garantem uma aplicação balanceada desta arquitetura como um todo.

Após ser abordado o tema da arquitetura orientada a serviços, onde foi possível perceber quais os seus fundamentos, os seus princípios e componentes bem como a pertinência da sua utilização, será discutida a temática dos *web services* - a vertente dos serviços utilizada na implementação da arquitetura orientada a serviços no projeto da dissertação.

2.2 Web services

Web Service é um *standard* para interoperabilidade entre vários sistemas [8]. Este *standard* é totalmente independente do hardware, do sistema e da tecnologia existente em cada um dos sistemas intervenientes na comunicação.

O termo *web service* comporta várias definições, normalmente associadas a um grupo específico de linguagens e protocolos.

Holt Adams et al. definem que um *web service* é uma aplicação identificada por um URI, cujas interfaces e ligações deverão ser definidas, descritas e localizáveis por XML e que suporta interações diretas com outro software via mensagens XML enviadas por protocolos *Internet-based* [9].

Jeffrey C. Broberg diz também que um *web service* é um componente de software descrito via XML e capaz de ser acedido por protocolos de rede, não estando limitados a SOAP via HTTP [10].

Por fim, o *World Wide Web Consortium* (W3C), consórcio responsável por estabelecer padrões para a criação e interpretação de conteúdos para a *web*, definem que um *web service* é um sistema de software desenhado para suportar a interação entre máquinas numa determinada rede, tendo este software uma interface descrita num formato processável, especificamente o WSDL. Outros sistemas interagem com um *web service* de acordo com a interface através de mensagens SOAP, tipicamente enviadas por HTTP e descritas em XML, em conjunto com outros *standards*

web [11].

Em conclusão, um *web service* trata-se de um serviço disponibilizado na internet ou em redes privadas que comunica com vários sistemas, através de protocolos tais como HTTP, SMTP, FTP, RMI/IIOP, sendo a mensagem convertida numa linguagem universal - XML - e encapsulada por um determinado protocolo de comunicação. Tem com o principal finalidade a comunicação de aplicações de diferentes naturezas através da rede, facilitando assim a integração de várias aplicações.

De acordo com o *W3C* [3], a arquitetura dos *web services* além de ser composta por várias tecnologias que passam pelos protocolos de mensagem de aplicação, por uma descrição em formato XML e por um protocolo ou uma coleção de protocolos de transporte, baseia-se em três elementos que comunicam entre si seguindo uma especificação SOAP:

- *Service consumer* - entidade que utiliza o serviço;
- *Service provider* - entidade que implementa o serviço e que transporta o mesmo para o cliente;
- *Service registry* - local onde estão listados os serviços disponíveis e que permite que o *service provider* publique os seus serviços, de forma a que o cliente possa fazer pesquisas sob eles.

Através destes elementos, pode-se representar o modo de funcionamento de um *web service* tal como demonstrado na Figura 2.4.

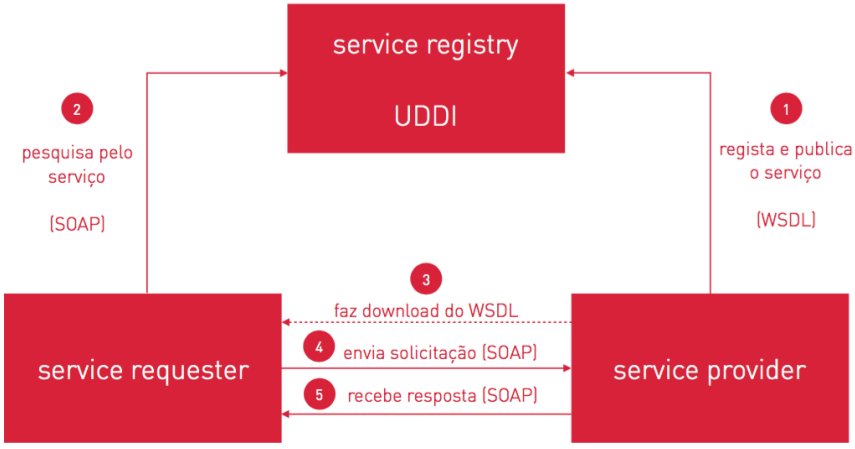


Figura 2.4: Funcionamento de um *web service*

Atualmente, a arquitetura dos *web services* é composta por várias especificações, sendo as especificações base a SOAP, WSDL e UDDI. Estas especificações são definidas de seguida:

- **SOAP** *(Simple Object Access Protocol)* - trata-se de um protocolo de comunicação que define um formato de envio de mensagens entre aplicações de uma forma descentralizada e distribuída. Todas as mensagens trocadas entre serviço e cliente passam a ser armazenadas em envelopes SOAP, que contêm documentos XML, e são enviadas através de protocolos da camada de aplicação tais como o HTTP ou SMTP. Cada mensagem SOAP (Listagem 2.1) é composta por um **Envelope** que identifica o documento como sendo uma mensagem SOAP, sendo que este contém (opcionalmente) um elemento **Header** onde se podem encontrar informações adicionais possivelmente necessárias na comunicação da mensagem. Por fim, a mensagem contém um elemento **Body** que contém o corpo da mensagem e os dados específicos da mesma, sendo que o seu conteúdo varia conforme as necessidades da implementação e tipicamente inclui os métodos e respetivos argumentos a invocar.

```
1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
2   <env:Header>
3    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
4     <n:priority>1</n:priority>
5     <n:expires>2001-06-22T14:00:00-05:00</n:expires>
6    </n:alertcontrol>
7   </env:Header>
8   <env:Body>
9    <m:alert xmlns:m="http://example.org/alert">
10    <m:msg>Pick up Mary at school at 2pm</m:msg>
11   </m:alert>
12  </env:Body>
13 </env:Envelope>
```

Listagem 2.1: Exemplo de um envelope SOAP [3]

O facto de toda a informação estar estruturada em documentos XML, faz com que seja possível o uso de linguagens de programação diferentes quer no lado do serviço, quer no lado do cliente.

Além do objetivo principal do protocolo, este providencia também mecanismos de extensibilidade, tornando possível adicionar funcionalidades aos serviços, tais como fiabilidade e segurança. As especificações associadas a estas funcionalidades são conhecidas como "WS-*", exemplificadas de seguida:

WS-Security: aplica mecanismos de segurança na troca de mensagens;

WS-Addressing: faz com que o endereçamento de mensagens e serviços seja agnóstico do protocolo de transporte usado na comunicação;

WS-AtomicTransaction: garante que a transação é aplicada na totalidade;

WS-ReliableMessaging: define mecanismos para uma troca fiável de mensagens;

- **WSDL** *(Web Services Description Language)* - tal como o próprio nome indica, WSDL é uma linguagem que serve para descrever um *web service*. Esta linguagem baseia-se em XML e funciona como o contrato do serviço, explicando não só o modo de funcionamento do mesmo, mas também a forma de o aceder e as operações disponíveis. Este tipo de documentos permite gerar automaticamente o código necessário para a implementação de uma interface no lado do consumidor do serviço em causa e estão normalmente associados a *web services* que usem mensagens SOAP. Um documento WSDL, exemplificado na Listagem 2.2, é composto por elementos **message** que definem os dados a serem trocados, podendo conter elementos que se assemelham aos parâmetros de uma função, denominados por **part**. A cada *part* está associado um nome e um tipo que indica o tipo de dados usados no serviço em questão. Dentro do mesmo documento podem encontrar-se também elementos **portType** que, tal como uma classe numa linguagem de programação tradicional, tratam-se do conjunto de operações que podem ser executadas e as mensagens envolvidas nas mesmas. Por fim, os elementos **binding** especificam o protocolo e o formato de dados a usar em cada *portType*.

```
1  <message name="getTermRequest">
2    <part name="term" type="xs:string"/>
3  </message>
4
5  <message name="getTermResponse">
6    <part name="value" type="xs:string"/>
7  </message>
8
9  <portType name="glossaryTerms">
10   <operation name="getTerm">
11     <input message="getTermRequest"/>
12     <output message="getTermResponse"/>
13   </operation>
14 </portType>
15
16 <binding type="glossaryTerms" name="b1">
17    <soap:binding style="document"
18    transport="http://schemas.xmlsoap.org/soap/http" />
19    <operation>
20      <soap:operation soapAction="http://example.com/getTerm"/>
21      <input><soap:body use="literal"/></input>
22      <output><soap:body use="literal"/></output>
23   </operation>
24 </binding>
25
```

Listagem 2.2: Exemplo simplificado de um documento WSDL [4]

- **UDDI** *(Universal Description Discovery and Integration)* - é um protocolo desenvolvido para organização e

registo de *web services*. Foi pensado como sendo um repositório de metadados com informações úteis para a utilização dos serviços tais como métodos padrões para publicação e localização de informações dos serviços. O UDDI pode também ser usado para armazenar arquivos WSDL [3].

No entanto, nem todos os serviços necessitam de mensagens SOAP e documentos WDSL. Existem maneiras mais simples de fazer a implementação dos serviços, aproveitando somente o uso do protocolo HTTP como principal protocolo de transporte. O uso de *web services* SOAP, devido à sua elevada quantidade de informação, pode também tornar-se pesado, uma vez que consome demasiados recursos. Desta forma, uma outra possível abordagem na construção de *web services* trata-se dos *web services RESTful* que serão apresentados de seguida.

2.3 Web services RESTful

REST - *REpresentational State Transfer* - é um estilo de arquitetura e uma abordagem de comunicação utilizada no desenvolvimento de *web services*. Este tipo de arquitetura, cuja primeira referência surge por *Roy Fielding* [12], investigador que esteve no desenvolvimento do protocolo HTTP, tem tido um aumento de popularidade graças à sua facilidade de utilização.

Tal como todos os *web services*, um *web service RESTful* recebe um pedido onde está indicado o processo a ser executado, retornando uma resposta com o resultado obtido. No entanto, existem dois pontos que diferenciam REST dos restantes *web services*: a forma como a ação a ser executada é discriminada no pedido e a forma como o objetivo da execução da ação é discriminado [13]. Este baseia-se no estilo cliente-servidor sem informação de estados, seguindo assim o princípio de independência de estados da abordagem SOA. Por forma a que os pedidos não causem redução no desempenho do servidor, é possível indicar se um pedido permite ou não o uso de *cache*. Assim, uma resposta dada a um pedido assinalado como *cachable*, ou seja, que possa ser guardado, será reutilizada para futuras requisições equivalentes.

O princípio de interface uniforme entre os dois componentes (cliente e servidor) trata-se da característica fundamental do REST que o distingue dos restantes estilos baseados em rede. Como tal, o REST indica que deverá existir uma sintaxe universal para a identificação de cada recurso, sendo este direcionado através do seu URI. Importa referir que um recurso pode ser referenciado por vários URI, sendo que caso não tenha qualquer referência, não pode ser considerado recurso.

Segundo *Fielding* [13], uma arquitetura *RESTful* segue o conceito HATEOAS (*Hypermedia as the Engine of Application State*), permitindo que todos os URIs sejam encontrados através de um único ponto de partida. No entanto, devido ao grau de complexidade, nem sempre esta filosofia é aplicada, o que faz com que na ausência do ponto de partida, seja necessária a existência de um documento com uma API pública para que a que o programador possa implementar as ligações aos recursos. Este princípio requer que a manipulação dos recursos seja feita através de representações. Deste modo, cada recurso REST é manipulado por 4 operações :

- **GET** - responsável pela leitura de um determinado recurso;
- **PUT** - responsável pela criação ou atualização de um recurso indicado no URIs;
- **POST** - responsável pela transferência de estados de um determinado recurso;
- **DELETE** - responsável por apagar um determinado recurso.

As mensagens transferidas deverão ser auto-descritivas, contendo informações que poderão ser utilizadas para, por exemplo, controlo da *cache*, deteção de erros de transmissão, negociação do formato de representação e para autenticar ou controlar o acesso ao recurso.

Por forma a aperfeiçoar o requisito de escalabilidade da internet, é possível a divisão de camadas fazendo com que seja possível separar diferentes funcionalidades. Embora a performance possa ficar comprometida devido ao *overhead* e à latência nos dados processados, pode tornar-se útil numa rede que suporte *cache*.

Por fim, o princípio *code-on-demand* trata-se de um princípio opcional que permite que os clientes possam transferir e executar o código do seu lado, em forma de *applets* ou *scripts*. Apesar de simplificar a parte cliente e de melhorar a extensibilidade do sistema, reduz a visibilidade, ou seja, o sistema deixa de ter a mesma regulação e monitorização do seu funcionamento quando comparado com a execução do código no lado do servidor [12, 14].

2.4 Bloom

O *Bloom* é um sistema de gestão de atendimento que pretende otimizar o serviço dos seus utilizadores. Dada a sua vasta gama de componentes, o *Bloom* vai de encontro às necessidades dos vários tipos de atividades e organizações, desde o retalho e grandes cadeias de distribuição, passando por organizações governamentais, educacionais, organizações de saúde, banca e restauração. Uma vez que é possível desenhar o fluxo que melhor se adequa ao negócio da organização (desde o início ao fim do atendimento), com o *Bloom* é possível:

- Controlar e prever o fluxo de clientes;
- Manter os clientes da organização informados;
- Aceder a informação relevante à otimização dos recursos da organização.

Como base para o funcionamento, cada organização necessita de ter por cada localização um computador com uma instalação do *Bloom Master* (referido no decorrer deste documento como *Master*). Este componente é a base fundamental do sistema e é vital no controlo e gestão das filas dando também acesso a informação em tempo real, incluindo estatísticas e histórico de dados. O *Master* permite ainda detetar, instalar e gerir os componentes adicionais do sistema, conferir ao sistema uma completa autonomia e independência de conexões externas e é também o responsável pelas comunicações com o servidor, o *Bloom Enterprise Server* (BES).

Para que se possa criar uma experiência de utilização ajustável ao negócio da organização, é possível escolher entre vários componentes e fazer a combinação dos mesmos. Os componentes estão categorizados da seguinte forma:

Componentes para registo de clientes no sistema

Antes de chegar ao local, o cliente não sabe o número de pessoas que estão à sua frente e, ao chegar à fila, não tem a certeza sobre qual o serviço que melhor se adequa às suas necessidades. Com o *Bloom*, os clientes através da aplicação *Virtual Ticket* podem registar-se previamente gerando uma senha para o serviço pretendido, obtendo também informações sobre o estado do processo de atendimento. Caso não tenha gerado previamente a senha e uma vez chegado ao local, o registo no sistema pode ser feito através de uma senha impressa por um dispensador de senhas ou por uma senha virtual obtida num *Bloom Tablet Kiosk* ou após a abordagem de um colaborador da localização que terá ao seu dispor a aplicação *Bloom Concierge*.

Componentes para tirar partido do tempo de espera

Com o *Bloom*, as condições de espera dos clientes são melhoradas pela apresentação de conteúdos dinâmicos e envio de notificações sobre o estado da fila de espera, permitindo que o cliente utilize o seu tempo de espera da forma que lhe for mais conveniente. Para responder a esta necessidade, o *Bloom* tem *players* com conteúdos totalmente personalizáveis e ainda permite a integração do módulo de gestão de filas em softwares de *digital signage*, usando para tal o seu módulo *Virtual Player*. Um complemento a estes *players* é a aplicação para tablets *Bloom Extender*, onde poderão ser transmitidos conteúdos multimédia.

Componentes para notificação de chamada

Quando chega a sua vez, o *Bloom* permite notificar o cliente de diversas formas: através dos *players*, através de alertas para a aplicação Virtual Ticket, através de um ecrã de LEDs e ainda através do *Bloom Extender*, onde poderá ser apresentado o número do balcão de atendimento, a última senha chamada ou o próprio nome do cliente. Será também possível receber a notificação de chamada via SMS, caso o cliente tenha preenchido o formulário ao chegar ao local.

Componentes para gestão de atendimentos

A gestão dos atendimentos está disponível em duas aplicações: o *QM-PAD* e o *Bloom Concierge*. O *QM-PAD*, mostrado na Figura 2.5, está incluído no BQM e trata-se de uma aplicação *web* que permite além de fazer a chamada, cancelar, pausar ou transferir para outro serviço/balcão/colaborador, permite ainda gerar senhas no próprio local e também preencher inquéritos no final de cada atendimento. O *Bloom Concierge* alia às suas características a mesma possibilidade de gestão de atendimentos do *QM-PAD*, com a vantagem de permitir alterar informações pré-preenchidas pelo cliente antes e durante o atendimento.

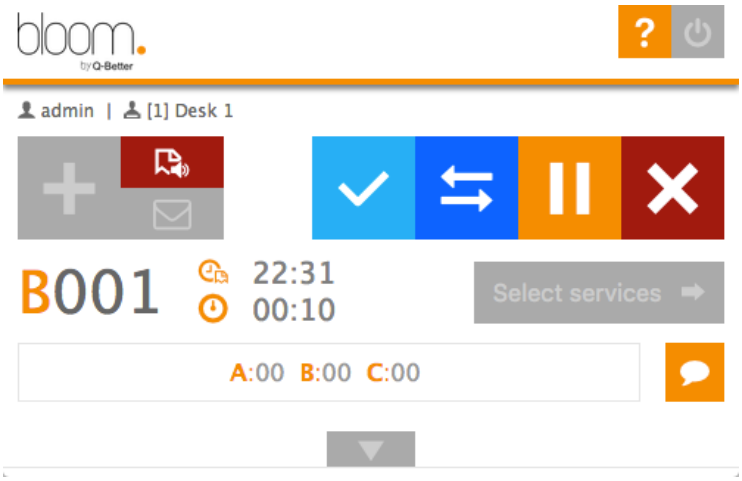


Figura 2.5: QM-PAD durante um atendimento

Componentes para obtenção de *feedback*

Após o atendimento, é possível através do *Bloom Extender* ou da aplicação *Virtual Ticket* a obtenção de *feedback* por parte dos clientes. Este formulário poderá servir para inquérito de satisfação ou outro tipo de informação que possa ser útil para o negócio.

Para efetuar a configuração do sistema basta aceder através de um browser ao endereço de rede configurado na máquina que está ser executado. É no back office que pode fazer configuração de serviços disponibilizados, balcões e colaboradores, conteúdos e modo de apresentação de informações nos componentes é feita através do *Master*, existindo uma gestão completamente centralizada do sistema.

A título de exemplo, a Figura 2.6 mostra a página de configurações de serviços e balcões, sendo que a Figura 2.7 mostra a listagem de dispositivos adicionados a uma instalação do BQM, o seu estado atual e as ligações às suas páginas de configuração.

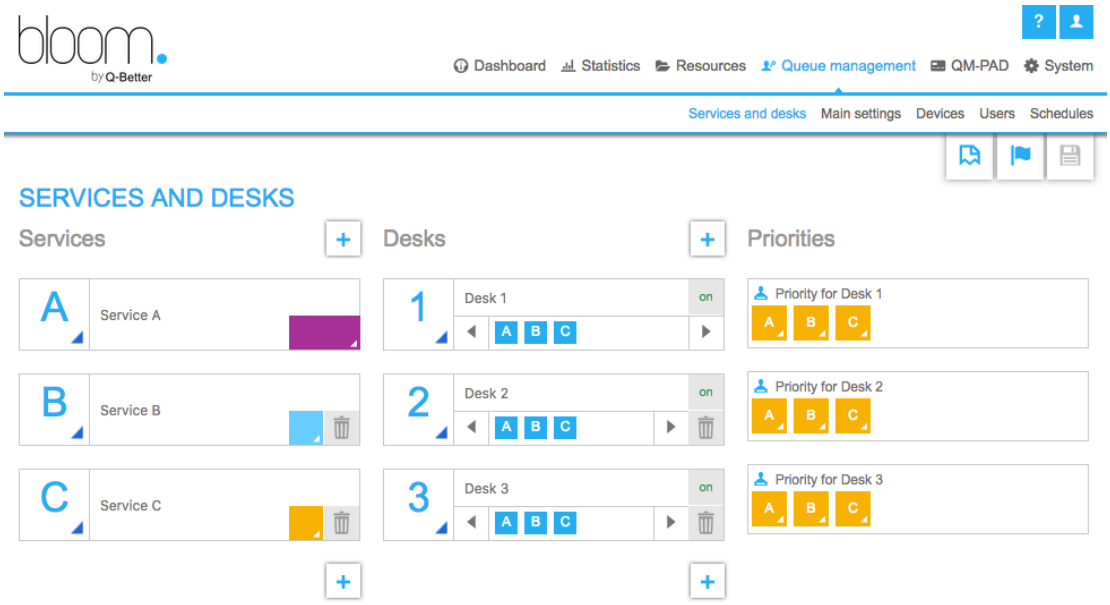


Figura 2.6: Configuração de serviços, balcões, prioridades e associação de serviços com balcões no BQM

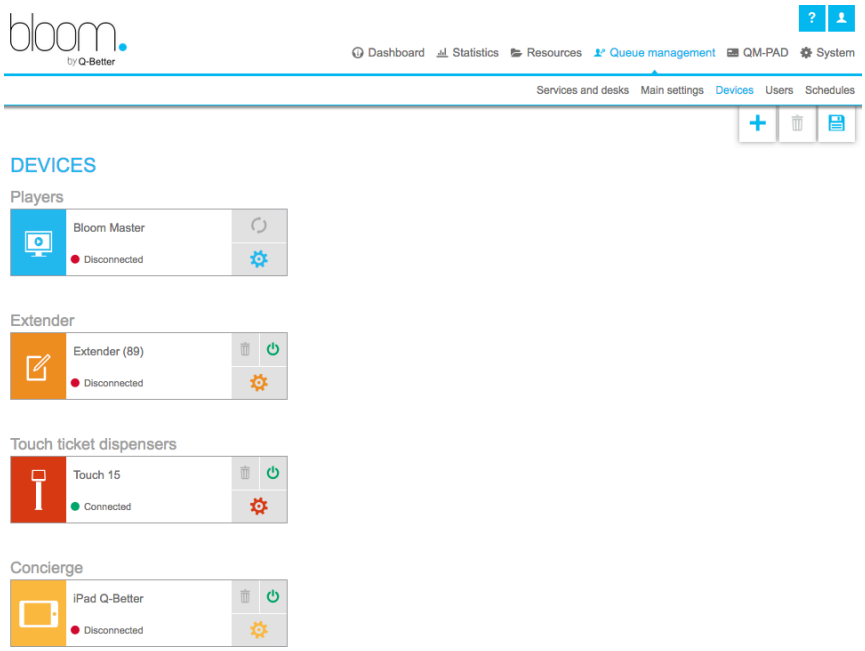


Figura 2.7: Listagem de dispositivos adicionados a uma instalação do BQM

Nos casos onde uma determinada organização tenha várias localizações com instalações do *Bloom*, é possível fazer uma gestão e monitorização de toda a rede em tempo real. O BES é a solução para redes grandes e descentralizadas. Este sistema permite fazer uma gestão central e remota de todos os *Masters* da rede *Bloom*, assim como conhecer a fundo o serviço de atendimento ao cliente através de análise de estatísticas. Permite também acompanhar o desempenho global do seu negócio, verificando as estatísticas de cada localização ou grupo de localizações, permite a gestão de conteúdos a mostrar nos vários dispositivos e permite a replicação de definições entre *Masters*.

Conhecido o *Bloom* e os seus vários componentes, é apresentada de seguida a descrição técnica do sistema, possibilitando assim perceber a forma como está construída a solução. A descrição técnica permitirá saber a atual composição do sistema por forma a que seja compreendido o enquadramento da arquitetura a desenvolver ao longo deste projeto.

2.4.1 Descrição Técnica

O *Bloom* segue uma filosofia de desenvolvimento por módulos, desenvolvidos em linguagens distintas. Cada módulo tem uma finalidade própria e existem quer no *Master*, quer nos restantes componentes do sistema. A comunicação entre eles é feita seguindo o modelo de comunicação *publish/subscribe*. Este modelo baseia-se na troca assíncrona de mensagens enviadas por clientes (tipicamente conhecidas por eventos), cuja sua recepção é feita pelos módulos que tenham registado o interesse em recebê-las. A implementação deste modelo está feita de um modo centralizado, onde existe o *Q-BUS* que é responsável por manter os registos de interesse em eventos e também é responsável pela sua disseminação. Cada evento é composto por um tipo e um corpo, sendo o conteúdo da corpo variável, uma vez que existem eventos que transportam informação (descrita em JSON ou XML) e eventos que funcionam apenas como *triggers*, não transportando assim qualquer conteúdo. A Figura 2.8 mostra os vários módulos, divididos em 3 categorias:

- *Bloom Master*: Uma vez que é o core de todo o sistema, contém a maioria dos módulos essenciais para o seu funcionamento;

- *Bloom Slave*: Trata-se de um player que tem apenas como propósito a reprodução de conteúdos;

- *Hardware devices*: Esta categoria abrange todos os dispositivos que permitem o registo dos clientes no sistema, abrange o ecrã de LEDs e restantes aplicações móveis.

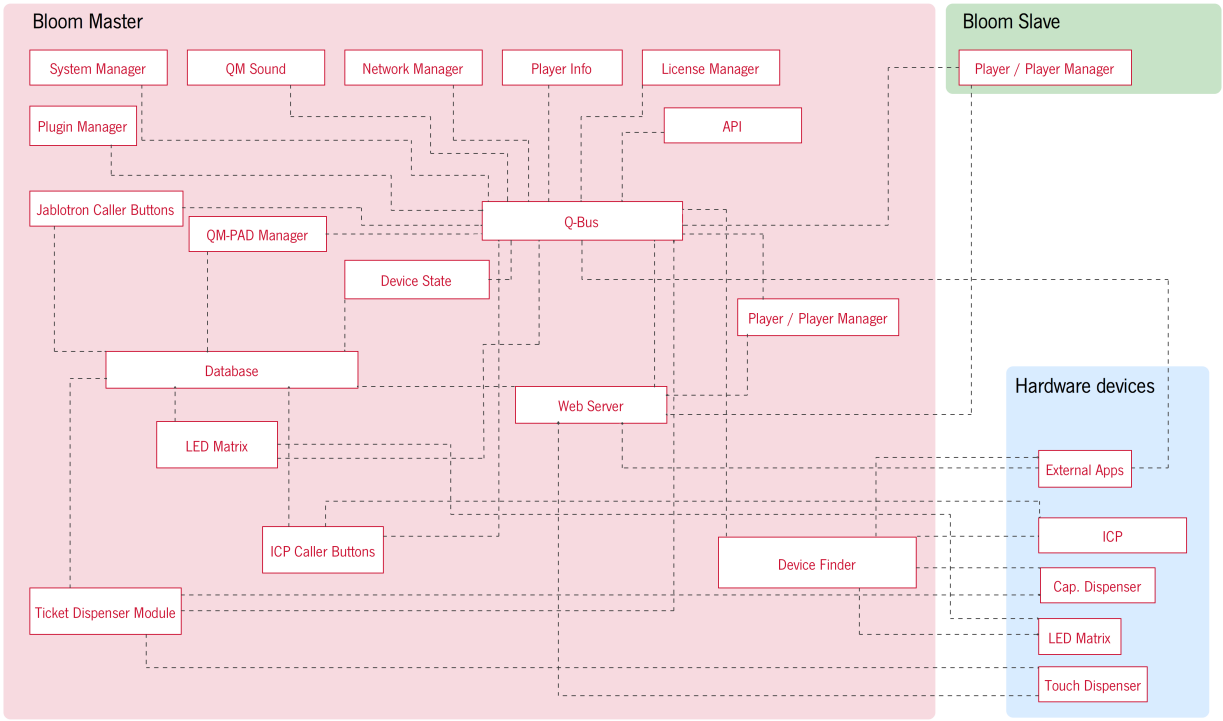


Figura 2.8: Diagrama de blocos do *Bloom*

A gestão de todo o sistema é feita através de um *browser* de qualquer dispositivo, uma vez que todo o *back office* está desenvolvido em linguagens compatíveis com *standards web*. Apesar de existir informação que é armazenada diretamente na base de dados sem recurso aos módulos, o facto de (por exemplo) adicionar um novo serviço ao sistema, faz com que seja emitido um evento que reinicia todos os dispositivos por forma a obterem nos seus conteúdos a informação sobre o serviço criado.

Embora o sistema esteja fechado para comunicações vindas do exterior para o *Master* (à exceção do BES), o sistema disponibiliza uma *API RESTful* para uma possível integração da gestão do atendimento em softwares de terceiros. Uma vez estabelecidas as devidas comunicações, é possível criar senhas e fazer toda a sua gestão desde a chamada, o cancelamento ou suspensão ou até mesmo a transferência da mesma para outro serviço, balcão ou colaborador.

Além disso, também é possível desenhar toda a componente gráfica da gestão de chamadas com recurso a um documento XML gerado em tempo real no *Master* que descreve todo o estado da fila no momento. Isto faz com que seja possível obter um modelo de apresentação personalizado e integrado noutro software de *digital signage*.

2.5 Sincronização

Sendo o *Bloom Appointments* um produto a adicionar ao conjunto de aplicações da *suite Bloom*, faz sentido pensar em como este poderá interagir com o sistema de gestão de filas já existente. A passagem de informação entre os dois sistemas necessita de um cuidado especial, nomeadamente na sincronização de agendamentos entre ambos, uma vez que se trata do ponto fulcral da interligação, já que uma falha pode dar origem a um agendamento sem efeito na fila de espera, ou até mesmo agendamento em duplicado, dando origem a entropia na própria fila.

Existem dois tipos de sincronização: dados ordenados e dados não-ordenados. Relativamente à sincronização de dados ordenados, trata-se de um problema de difícil resolução, uma vez que neste caso, é imprescindível manter a informação da ordem dos dados. Em relação à sincronização de dados não-ordenados, esta é conhecida por originar vários problemas de consolidação de dados, pelo que, foram desenvolvidas várias soluções para corrigir este problema estando a aplicabilidade destas soluções dependente do modelo de distribuição escolhido. Grande parte destas soluções funcionam num modelo onde os dados são sincronizados através de um servidor central, mas quando são aplicadas num modelo *peer-to-peer*, as soluções requerem modificações tendo em conta o facto de existirem alterações que precisam de estar sincronizadas com vários clientes [15].

De seguida, estão listadas as soluções mais relevantes no que diz respeito a este tipo de sincronização:

- *Wholesale synchronization*: Quando os dados são sincronizados, um dos dispositivos envolvidos na sincronização envia os dados para outro dispositivo para uma comparação. Este método torna-se ineficiente uma vez que na maioria dos casos, as alterações são residuais enquanto que os dados são enviados pela rede para comparação. No entanto, tem a vantagem de ter uma implementação simples e garante que todas as alterações são transmitidas;

- *Flags de estado da sincronização*: O cliente passa a ter informação sobre os dados em forma de *flags*. Estas *flags* indicam se um determinado item foi criado, alterado ou apagado. Na sincronização, o cliente pode simplesmente enviar os items que tenham a *flag* definida. Apesar de ser mais eficiente que o método anteriormente descrito, quando a sincronização envolve vários clientes, o seu funcionamento torna-se ineficaz;

- *Sincronização com timestamps*: Com este método de sincronização, todos os clientes mantêm o registo sobre a última vez que a informação foi alterada e também quando foi feita a última sincronização. Desta forma, a necessidade de envio de informação passa a estar restringida apenas aos items alterados desde a última

sincronização. Trata-se de uma melhoria comparativamente às *flags* de estado, uma vez que não existe a necessidade de manter informação sobre que alteração foi sincronizada e com quem. No entanto pode tornar-se ineficiente em situações onde existem dois clientes sincronizados com outros clientes pela primeira vez. Neste caso, têm de enviar toda a informação enquanto não existirem alterações entre eles;

- *Log Synchronization*: Esta técnica é bastante usada em bases de dados. Todas as alterações são executadas como uma transação e são gravadas em *logs*. Estes *logs* são também sincronizados com outros clientes. No entanto, esta técnica pode tornar-se prejudicial, uma vez que os *logs* crescem paralelamente à quantidade de dados sincronizados [16].

A sincronização pode funcionar quer direcional, quer bi-direcionalmente. No caso onde a informação é apenas lida pelo cliente e as alterações ocorrem apenas no servidor, os dados só podem ser obtidos via servidor, sendo este tipo de sincronização denominado por *Download-only synchronization*. Quando os dados são apenas usados e alterados pelos clientes, apenas os novos são enviados para o servidor. Este tipo de sincronização é descrito como *Upload-only synchronization*.

Capítulo 3

Arquitetura proposta

É neste capítulo que a arquitetura proposta para suportar o desenvolvimento do *Bloom Appointments* está retratada. Esta arquitetura segue os princípios de uma arquitetura orientada a serviços e tem em conta os requisitos levantados no capítulo anterior. Assim, serão abordadas ao longo deste capítulo as frameworks Laravel 5 e AngularJS, uma vez que em conjunto, são as que melhor servem o propósito de uma arquitetura orientada a serviços no contexto apresentado.

3.1 Frameworks

Na primeira secção deste capítulo, será abordado inicialmente o padrão de desenvolvimento *Model-View-Controller*, uma vez que se trata do padrão que dá suporte às duas frameworks que serão usadas para a implementação desta arquitetura, estando estas igualmente descritas nesta secção.

3.1.1 MVC

As frameworks Laravel 5 e AngularJS têm em comum a utilização do padrão de desenvolvimento *Model-View-Controller* - MVC. Este padrão foi inicialmente descrito por *Reenskaug* [17] e revisto por *Burbeck* [18] e é definido como um paradigma que permite a troca de mensagens entre os componentes da aplicação de uma forma organizada, separando as aplicações em 3 camadas lógicas: Models, Views e Controllers. A ideia central baseia-se na possibilidade de reutilizar código e na separação de tarefas durante o processo de desenvolvimento, tornando assim a aplicação mais

flexível, assegurando também que a expansão da mesma seja facilitada.

Seguindo uma abordagem *bottom-up*, encontramos a camada que representa as entidades da aplicação, os Models. Estes correspondem aos dados existentes nas tabelas da base de dados e podem gerir os mesmos através de pedidos de consulta e alterações de estado da informação.

De seguida, temos a camada dos Controllers, cuja responsabilidade é processar os dados recebidos por parte do utilizador e coordenar as ligações entre as Views e os Models. De um modo simplista, estes começam por extrair as informações dos Models fazendo um pré-processamento para as validar, tratam as mesmas e enviam os dados para a View, por forma a serem mostrados ao utilizador.

Por fim, a camada das Views é responsável por mostrar a resposta vinda de um Controller, ficando ao seu encargo os aspectos visuais e o modo como a informação será apresentada ao seu destinatário.

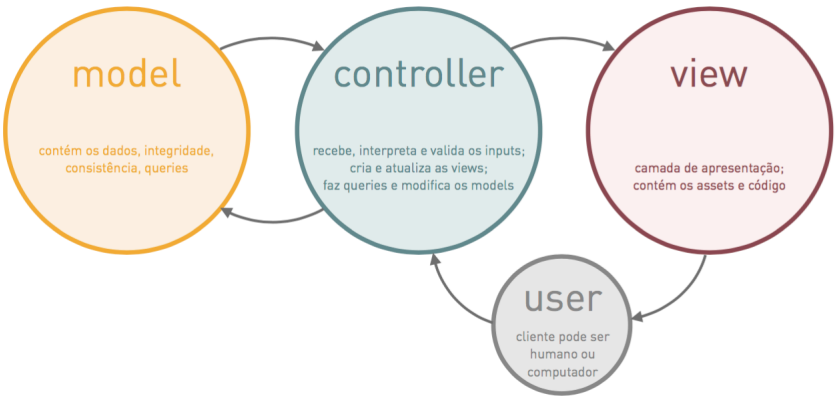


Figura 3.1: Modelo de interação do padrão MVC [2]

A implementação do padrão MVC, graças à separação em três componentes, permite um desenvolvimento mais especializado e focado, uma vez que é possível ter programadores a trabalharem em paralelo em componentes distintos. Isto faz com que os desenvolvimentos da camada de negócio possam ser feitos sem necessidade de intervenção por parte dos responsáveis pelo UI e vice-versa. Outra das vantagens passa pela possibilidade de existirem em várias Views para o mesmo tipo de informação e ao mesmo tempo, fazendo com que não exista a necessidade de replicação de código. Apesar da possível complexidade do desenvolvimento de uma aplicação baseada neste paradigma, este demonstra preencher os requisitos necessários para a implementação da arquitetura requerida no âmbito desta dissertação [18, 2].

3.1.2 Laravel 5

Tal como referido no início do secção anterior, a framework Laravel é baseada no paradigma MVC. Nesta framework, os Models são classes que extendem os Models do ORM incluido - *Eloquent* - e os seus nomes surgem normalmente em *CamelCase* (por exemplo *DomainRole*). Os Models, além de representarem tabelas da base de dados, podem também descrever relações que existam entre eles. Relativamente aos Controllers, esta framework permite a criação de Controllers específicos para *web services RESTful*, definindo através das rotas quais são as ações e respostas a dar para um determinado pedido HTTP. Por fim, as Views podem ser implementadas através do uso dos *templates Blade* ou através do uso básico do PHP, sendo que o Laravel distingue ambos através da extensão do ficheiro (ex. *dashboard.blade.php*).

As características desta framework que permitem potenciar a produtividade são apresentadas de seguida:

- *Modularidade*: Esta framework foi desenvolvida sobre 20 bibliotecas diferentes e a própria composição está dividida em módulos. Além disto, está integrada com o *Composer* que permite gerir e atualizar os vários componentes;

- *Testável*: O Laravel tem na sua génese várias ferramentas que permitem testar as aplicações de uma forma facilitada. É possível testar se todas as rotas estão bem definidas, se os métodos são chamados nas classes correspondentes, testar o HTML resultante e até fazer testes com utilizadores autenticados;

- *Routing*: Permite uma maior flexibilidade na definição das rotas da aplicação. Torna-se fácil criar e agrupar rotas, criar os recursos necessários para uma *API RESTful*, ou até mesmo a criação de versões da API;

- *Gestão de configurações*: É possível definir facilmente os vários ambientes de funcionamento da framework, bastando para tal utilizar um ficheiro de extensão *.env* que contem várias definições tais como: ligação à base de dados, configurações do servidor de email ou as possíveis mensagens de erro;

- *Query builder e ORM*: O Laravel inclui um *query builder* que permite a criação de *queries* na base de dados usando para tal métodos com sintaxe PHP, ao invés da escrita de SQL. Além disto, também contém o *Eloquent* que é o ORM que permite definir modelos que possam ser interligados;

- *Schema builder, migrações e seeding*: Estas características fazem com que seja possível definir o esquema da base de dados através do uso de código PHP, sendo que as migrações são uma forma fácil de descrever

alterações nesse mesmo esquema, incluindo até a reversão das mesmas. Com *seeding* é também possível popular a base de dados após a execução de uma migração;

- *Template engine*: O Blade é uma das ferramentas incluídas no Laravel que permite a criação hierárquica de *layouts* com recurso a blocos pré-definidos e com a possibilidade de injeção de conteúdo dinâmico;
- *E-mailing*: Graças à classe *Mail* que usa a libraria *SwiftMailer*, é fácil o envio de e-mails com conteúdo e até mesmo atalhos. Além disso, o Laravel traz *drivers* para vários serviços populares de e-mail;
- *Autenticação*: Uma vez que a autenticação tornou-se comum em aplicações *web*, o Laravel traz implementado por defeito mecanismos para registar, autenticar e até mesmo reenviar a palavra-passe [19].

3.1.3 AngularJS

O AngularJS é uma framework *open-source* desenvolvida pela *Google* e que teve origem no projeto *Google Feedback* em 2009. Trata-se de uma framework que permite construir e estruturar aplicações *web* dinâmicas, conhecidas por *single web page web applications* [20], por forma a que estas sejam reutilizáveis e modulares. Segundo *Adam Freeman* [21], o objetivo do AngularJS é trazer as ferramentas e capacidades que estavam apenas disponíveis no lado do servidor para o lado do cliente, facilitando o desenvolvimento, os testes e a manutenção de aplicações *web* complexas.

Um dos factos importantes de realçar é que esta framework segue o princípio de baixa dependência de uma arquitetura orientada a serviços apresentado na sub-secção 2.1.3, permitindo assim que exista uma separação de abordagens durante o desenvolvimento da aplicação: a abordagem ao comportamento da aplicação e a abordagem ao aspecto gráfico.

Uma aplicação que use AngularJS tem a sua arquitetura separada nas 3 camadas do padrão MVC:

- *Views* : correspondem à interface apresentada no *browser* ao utilizador;
- *Models* : correspondem aos dados apresentados ao utilizador nas Views;
- *Controllers* : corresponde à lógica que controla os dados apresentados ao utilizador. Estes estão responsáveis por determinar o comportamento dos elementos do DOM e são responsáveis por inicializar e/ou adicionar comportamentos ao *scope* que permite a comunicação entre a View e ao Controller [22].

Uma das características do AngularJS é a sua extensão do HTML, fazendo com que cada aplicação possa ter as suas diretivas, que são novas *tags* ou novos atributos para *tags* HTML, utilizadas para manter o código mais transparente ou para adicionar novos comportamentos a *tags* já existentes, podendo também interagir entre si.

Outra das grandes vantagens do uso desta framework trata-se da ligação bidirecional de dados (*two-way data binding*). Este componente é dos mais preponderantes do AngularJS e é responsável pela sincronização automática de dados entre duas camadas do MVC: a View e os Models. Ao contrário de outros *templates* de aplicações onde os *templates* e os dados são juntos uma só vez e as alterações dos dados não refletem alterações na View, no AngularJS o *template* passa a ser compilado no *browser*, produzindo assim uma View em tempo real, onde qualquer alteração à mesma é refletida no Model respectivo e vice-versa. Esta vantagem permite poupar linhas no desenvolvimento uma vez que deixa de ser necessária a preocupação com a manipulação do DOM. A sua implementação é bastante simples, bastando para tal a utilização dos caracteres {{}} ou da diretiva *ng-bind*.

Os serviços são também componentes do AngularJS com bastante utilidade, uma vez que é possível definir uma determinada funcionalidade uma vez e posteriormente usá-la em vários Controllers, com recurso a um dos *design patterns*: injeção de dependência (*dependency injection*). A injeção de dependência permite ao programador descrever as interligações entre os módulos da aplicação, podendo optar também por substituir os módulos de acordo com a necessidade do projeto e, uma vez que a declaração é explícita, a leitura e manutenção do código está facilitada.

Um possível complemento aos Controllers ou aos serviços são as *factories*. Uma vez que podem funcionar como os modelos da aplicação, o componente onde estejam injetadas pode instanciar novos objetos a partir dessa *factory*. Tal como os serviços, estes componentes são *singletons*, uma vez que o número de instâncias de uma classe está restrito a um único objeto.

O AngularJS inclui também um módulo de *routing*, o *ngRoute* que fornece serviços e diretivas de roteamento. Desta forma, cada vez que o módulo detetar alterações no URL da aplicação, o Controller respetivo é instanciado (incluindo possíveis carregamentos assíncronos antes do arranque deste) e a View é carregada.

Por fim, tal como o Laravel 5, o AngularJS também inclui mecanismos para testes que permitem assegurar uma maior fiabilidade do sistema [23, 24, 25].

Uma vez abordadas as frameworks que irão suportar o desenvolvimento não só da arquitetura proposta nesta dissertação, mas também do sistema como um todo, será descrita na próxima secção a arquitetura do sistema.

3.2 Detalhes da arquitetura

Os detalhes da infraestrutura da arquitetura e o seu esquema lógico estão explicados de seguida. Serão apresentados os vários componentes que a constituem, bem como os ambientes de execução da mesma.

3.2.1 Infraestrutura

O *Bloom Appointments* é uma solução criada de raiz que, além de se tornar em mais um produto produzido pela Q-Better, fará com que o paradigma de desenvolvimento da empresa siga novos caminhos, aproveitando as vantagens trazidas por uma arquitetura orientada a serviços.

No âmbito desta dissertação, a implementação da arquitetura orientada a serviços vem acompanhada do desenvolvimento de uma interface gráfica que interaja com os serviços disponibilizados. Para que tal aconteça, o sistema terá então 3 componentes base:

- O primeiro componente é a base de dados usada para gravar todas as informações inerentes ao sistema;
- O segundo componente trata-se da aplicação - é aqui que se encontra o motor de todo o sistema. Tal como os serviços, o *Q-BUS* (cuja descrição e motivação do seu uso pode ser encontrada na sub-secção seguinte) também pertence a este componente;
- O último componente trata-se da interface gráfica, cujo seu desenvolvimento neste projeto passa pela construção de uma interface *web* para os vários tipos de utilizadores.

A arquitetura deste sistema foi pensada por forma a que o produto resultante deste projeto possa ser executado em vários ambientes distintos:

- O *Bloom Appointments* é executado na íntegra através de uma só máquina, ficando ao encargo desta a execução de todos os componentes necessários para o funcionamento do mesmo;

- Outro ambiente possível ocorre quando o *Bloom Appointments* é apenas usado como motor de agendamentos, ficando a interface gráfica ou a interação com os serviços ao encargo de outra empresa/entidade, usando para tal aplicações previamente desenvolvidas (ex.: aplicação de gestão de hospitais,...). Desta forma, a máquina apenas executará a base de dados e a aplicação, sendo totalmente independente da escolha do tipo de interação com os serviços (seguindo uma lógica de cliente-servidor);

- O último ambiente de execução baseia-se na distribuição dos três componentes do sistema por várias máquinas distintas. A ideia deste ambiente de execução passa pela filosofia *scale out*, preparando assim a arquitetura para o seu previsível aumento, levando a pensar em mecanismos que garantam a execução perfeita do sistema. Ao separar todos os componentes do sistema, permitirá uma melhor gestão de cada componente, quer ao nível da base de dados, quer ao nível da aplicação (com recurso a *clusters* de alta performance e de uso ininterrupto), quer ao acesso à mesma, através do balanceamento de carga, por forma a garantir que todos os pedidos feitos aos serviços sejam atendidos de forma correta.

Apesar do tema da dissertação sugerir um funcionamento *online* do sistema, este está também pensado para funcionar sem qualquer ligação ao exterior, o que faz com que as questões de segurança sejam apenas restritas à própria rede local e à forma como esta é disponibilizada aos clientes da localização.

3.2.2 Esquema lógico

Após a análise dos requisitos da aplicação, é necessária a implementação de uma arquitetura orientada a serviços que siga os princípios referidos no Capítulo 2 e que responda ao modelo de domínio anteriormente descrito. Com recurso às framework indicadas na secção 3.1 e dando resposta à infraestrutura apresentada na sub-secção anterior, define-se então o seguinte esquema lógico da arquitetura, apresentado na Figura 3.2.

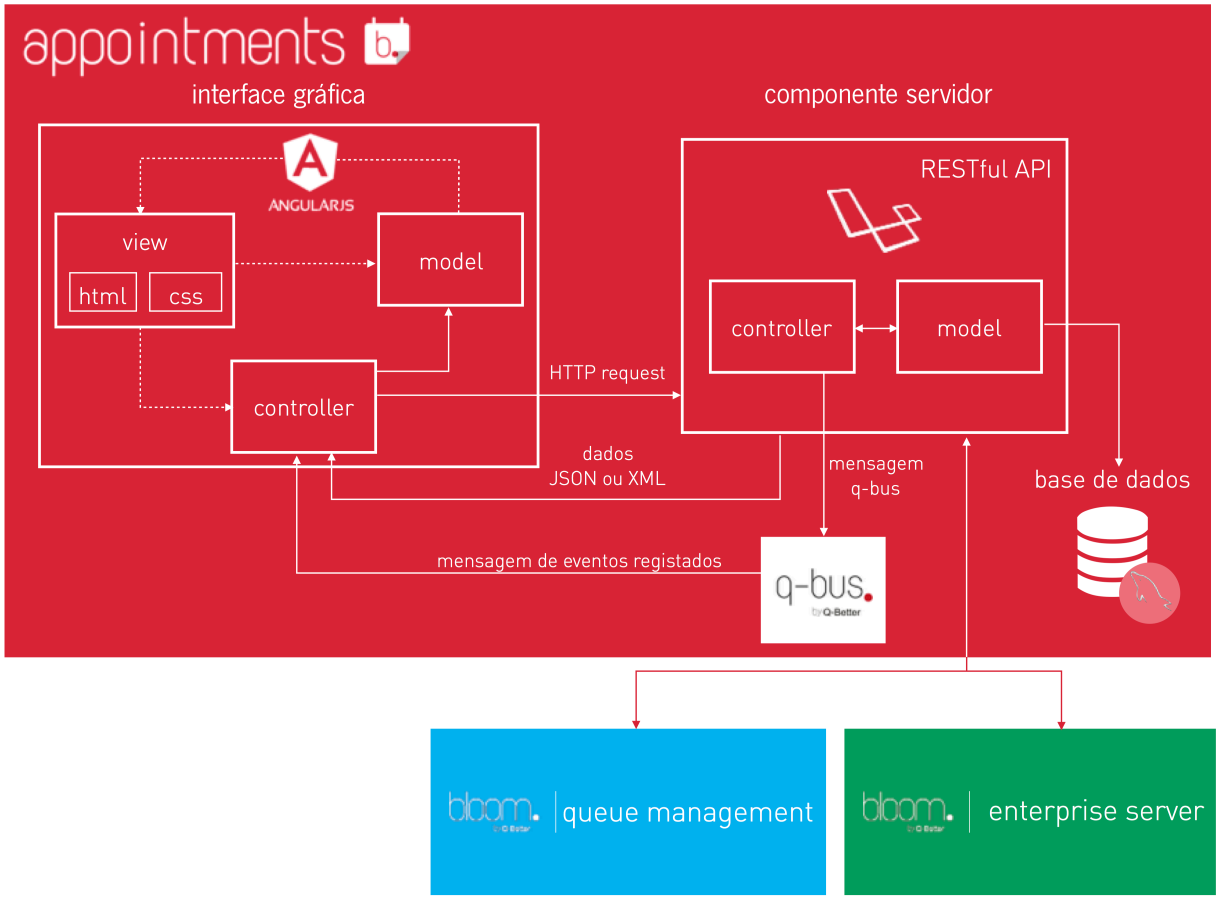


Figura 3.2: Esquema lógico da arquitetura do *Bloom Appointments*

Componente servidor

O *Bloom Appointments*, em semelhança aos restantes produtos *Bloom*, tem o seu componente servidor desenvolvido com tecnologia *web*, neste caso, com recurso à framework de PHP - Laravel 5. É aqui que está definida a *API RESTful*, onde se encontram implementados todos os serviços necessários para a execução da aplicação.

A base de dados *MySQL* contém as tabelas desenvolvidas mediante requisitos da modelação e que são criadas com recurso ao *Migrations* do Laravel. Através do *Migrations*, é possível criar toda a estrutura da base de dados e os respetivos Models automaticamente através de código PHP, de uma forma completamente agnóstica do tipo de base de dados escolhido, sendo que cada Model corresponde a um PDO (*PHP Data Objects*).

Os DAO (*Data Access Objects*) oferecem um acesso à base de dados independente do tipo escolhido, permitindo

assim alterar o tipo de base de dados sem necessidade de alterar o código usado para o acesso a esta. Os PDO tratam-se de uma extensão dos DAO para a linguagem de PHP. Desta forma, a comunicação e interação entre os Controllers e a base de dados fica assim assegurada.

Para garantir a integridade de cada transação na base de dados, é necessário garantir quatro propriedades conhecidas como **ACID**:

- **Atomicidade**: garantir que todas as modificações da transição são aplicadas. Caso contrário, nenhuma delas o serão;
- **Consistência**: garantir que após executar a transição, o estado da base de dados deverá manter-se igualmente consistente;
- **Isolamento**: garantir que não há dependência de transações na execução das mesmas;
- **Durabilidade**: garantir que após a execução com sucesso da transição, o seu resultado não possa ser alterado.

Com recurso aos PDO e para garantir a integridade de todos os pedidos, cada transação deve ser iniciada com o método *beginTransaction()*, sendo o método *commit()* responsável pela execução da transação e o *rollback()* para anular a mesma em caso de erro.

Cada Controller terá a implementação dos vários serviços disponíveis, fazendo assim uso dos PDO vindos dos Models necessários no desenrolar de cada um deles. Cabe também ao Controller a responsabilidade de comunicar via *sockets* com o *Q-BUS* para o envio de eventos que servirão essencialmente para alterações de estados e criação de notificações.

Uma vez que se trata de uma *API RESTful*, a comunicação com os serviços será feita com recurso a pedidos via HTTP, especificando no URI qual o serviço que pretende consumir. Dependendo da ação sobre o serviço, o pedido deve ter no cabeçalho o tipo da operação (GET, PUT, POST ou DELETE). O tipo de *output* de dados será também especificado no endereço, embora por omissão, o resultado da operação seja dado no formato JSON, uma vez tratar-se de uma conjugação bastante popular devido à facilidade de compreensão, à versatilidade e ao tamanho ocupado, sendo este um dos fatores no que diz respeito a comunicações por rede, nomeadamente via internet com dados móveis. No caso em que o *output* desejado seja em XML, este deverá ser obrigatoriamente especificado no endereço.

Interface gráfica

Por forma a manter uma coerência de utilização da linha de produtos *Bloom*, o *Bloom Appointments* deve disponibilizar interfaces *web* para os utilizadores desta aplicação.

Sendo assim, a interface gráfica é desenvolvida com recurso à framework AngularJS que, além de seguir o padrão MVC, permite que a aplicação seja alimentada pela *API RESTful* estabelecida no componente servidor. Cabe então a cada Controller alimentar os seus Models com os dados provenientes dos serviços disponibilizados, sendo posteriormente apresentados em cada View. A ligação com o *Q-BUS* para a receção de eventos é também da responsabilidade do Controller, estando este igualmente ligado com recurso a *sockets*.

Q-BUS

Tal como apresentado no sub-capítulo 2.5.2, o *Q-BUS* é o responsável por agrupar eventos enviados por vários módulos no BQM. No *Bloom Appointments*, o *Q-BUS* serve para o envio de notificações *push* para os vários utilizadores da aplicação e para a atualização de informação dos agendamentos em tempo real.

O seu uso permite que após ser concretizada uma determinada ação passível de notificação, seja enviada uma mensagem descritiva com o título do evento e o conteúdo necessário, por forma a que os Controllers que tenham esse evento registado, possam despoletar os mecanismos que geram o *pop-up* respetivo. Além disso, permite também que os blocos relativos a agendamentos sejam atualizados. A comunicação entre os Controllers da interface gráfica e o *Q-BUS* é feita através de *websockets*, existindo para tal uma ligação constante, o que permite a receção de eventos em qualquer momento.

Assim, tomando como exemplo o caso real de um agendamento feito por um cliente através da sua página, o *Q-BUS* permitirá que um elemento da organização receba na sua página a indicação de que foi criada um agendamento e apareça automaticamente listado. Caso a marcação seja confirmada, o cliente também receberá na sua página a notificação respectiva, sendo que o estado será automaticamente atualizado.

Integração com o BQM e BES

Para implementar a integração com o BQM e com o BES, o componente servidor disponibiliza serviços capazes de armazenar informações enviadas a partir deste.

Além das operações básicas como selecionar ou registar uma nova localização, desassociar o BQM, criar ou

atualizar a lista de serviços e horários de funcionamento, lista de balcões e as suas associações aos serviços e a lista de utilizadores, a API contempla serviços para o envio da listagem de agendamentos de um dado intervalo de datas e também para a receção de pedidos de check-in, caso estes sejam feitos a partir do BQM.

Como um complemento para instalações onde o BQM e o *Bloom Appointments* estejam disponíveis (quer estejam na mesma rede local, quer por recurso a uma VPN), é também possível guardar informações do endereço por forma a que o *Staff* habilitado a tal, possa utilizar o *QM-PAD* dentro do mesmo *back office*.

A integração com o BES tem um grau de complexidade menor, uma vez que apenas é possível a sincronização das entidades passíveis de marcação descritas no capítulo anterior.

Capítulo 4

Análise de requisitos

Terminado o estudo sobre as tecnologias associadas à temática desta dissertação e à concepção da arquitetura que dará suporte ao caso de estudo, neste caso a aplicação *Bloom Appointments*, iniciam-se os procedimentos para a criação da aplicação. Esta fase consiste no levantamento de requisitos, permitindo fazer o planeamento de todos os passos para a construção do produto final. A análise será feita com recurso a linguagens *standard* como o UML, através do desenho de vários diagramas apresentados ao longo deste capítulo.

4.1 Levantamento de requisitos

O *Bloom Appointments* surge da necessidade de aumentar a oferta de produtos da *suite Bloom*, por forma a dotar o sistema de gestão de filas da capacidade de cobrir casos reais como são os agendamentos. O sistema deverá ser capaz de cobrir diferentes cenários, tais como:

- Marcações de consultas em locais de saúde como hospitais, clínicas ou até mesmo consultórios privados, onde os agendamentos devem ser feitos por serviço ou por médico. A mesma filosofia de marcações aplica-se também em centros de estudo, centros de estética e outras empresas de prestação de serviços;

- Marcações de espaços como salas de aulas, salas de reuniões, auditórios, pavilhões, campos para prática de desportos, entre outros. Dependendo da organização dos locais, os agendamentos devem ser efetuados por serviço e/ou por espaço;

- Marcações de aulas em ginásios, de refeições em restaurantes, bem como de determinados serviços públicos existentes (por exemplo) em lojas do cidadão, centros de apoio ao Cliente ou de suporte técnico, ou até mesmo na requisição de equipamentos / instrumentos de trabalho em laboratórios, ou ainda marcações em centros de diagnóstico, os agendamentos devem ser efetuados apenas por serviço.

Para cada agendamento, o sistema deve mostrar (caso esteja associado) um formulário para recolha de informações adicionais, por forma a que a entidade para onde a marcação esteja a ser efetuada possa obter informação relevante sobre o Cliente.

O *Bloom Appointments* deverá permitir que para cada instalação seja possível a configuração das várias localizações pertencentes ao mesmo grupo. Usando como exemplo o grupo Trofa Saúde, deverá ser possível configurar todas as suas unidades de saúde (hospitais privados, hospitais de dia e instituto de radiologia) na mesma instalação. Esta ação, além de permitir uma gestão centralizada de todas as unidades, dará a possibilidade do Cliente fazer marcações para as várias unidades a partir do mesmo sítio e utilizando apenas uma credencial de acesso.

Por forma a padronizar a semântica usada nos produtos *Bloom* já existentes, devem usar-se os mesmos termos para descrever as várias entidades do sistema.

4.1.1 Entidades passíveis de marcação

Para uma maior organização do sistema, cada instituição/organização/empresa que pretenda adquirir esta solução, deverá criar uma localização. É na localização que estarão configuradas as várias entidades passíveis de marcações. Estas entidades contemplam os serviços, os balcões e os colaboradores.

Serviços

Seguindo a mesma lógica do BQM, os serviços deverão conter um nome, uma cor e uma letra identificativa. Adicionalmente deverão conter uma descrição e um formulário de informação adicional criado para a recolha de informação relevante. Deverão ser definidos o tempo de duração de um agendamento e a quantidade de agendamentos possíveis nesse mesmo bloco de tempo. Será também obrigatório definir horários de funcionamento, com possibilidade de especificação diária ou semanal, sendo que estes nunca poderão exceder o horário de funcionamento da localização onde o serviço está a ser criado.

Balcões

Os balcões devem ser obrigatoriamente descritos por um nome e por um número. Do mesmo modo dos serviços, é necessário definir os horários de funcionamento e a quantidade de agendamentos suportados em simultâneo. Deverá ser possível a indicação de quais serviços serão atendidos em cada balcão;

Colaboradores

Os colaboradores, além de um nome de utilizador e de uma palavra-passe, deverão ter uma ficha pessoal com informações relevantes tais como primeiro e último nome, morada, *email* e contacto telefónico para a sua identificação. O seu horário de trabalho, ao contrário das restantes entidades, passa a ser definido mediante o serviço que opera, podendo operar em vários serviços com horários completamente distintos. Deverá também ser possível indicar blocos de tempo em que não pode operar, como por exemplo, para marcação das respetivas férias.

4.1.2 Utilizadores do sistema

O *Bloom Appointments* também deverá permitir a definição de utilizadores com hierarquias distintas. Para responder a este requisito, deverão ser criados três tipos de utilizadores.

Clientes

É a classe de utilizadores que usufruirá do sistema. São eles que farão os agendamentos mediante a localização escolhida. Podem também desmarcar os mesmos e ter acesso ao seu perfil por forma a poder alterar a informação pessoal previamente respondida em formulários quer de registo, quer de informação adicional que possam surgir ao fazer uma marcação. Adicionalmente, deverão ter acesso às suas notificações e ao modo como as pretende receber.

Staff

São os colaboradores das localizações. É este o grupo responsável pelo desenrolar diário do sistema. Um elemento do *Staff* pode escolher em qualquer momento a localização onde deseja trabalhar mediante as localizações onde está associado, tendo também permissão para editar o horário de funcionamento e de fecho da localização no caso de férias ou feriados. Além disso, o *Staff* deverá ser capaz de fazer a gestão das entidades passíveis de marcação das suas

localizações, ou seja, dos balcões, dos serviços e dos seus formulários (se aplicável). Deverá ainda ser capaz de gerir o *Staff* que esteja associado à localização onde está a trabalhar, bem como os Clientes já registados no sistema, podendo inclusive criar novos Clientes.

O *Staff* poderá também ter acesso ao calendário de agendamentos da localização escolhida, podendo efetuar várias operações, tais como: criar e remover agendamentos, fazer a confirmação dos mesmos e fazer o *check-in* aquando da chegada do respetivo Cliente. Terá também acesso a um *dashboard* onde além de poder encontrar dados estatísticos que incluem a quantidade de agendamentos para o dia atual, a listagem do *Staff* que esteja online no momento e os gráficos relativos aos agendamentos num determinado intervalo de datas. Relativamente à localização, um utilizador *Staff* poderá gerir determinadas configurações: definir idioma e zona horária, alterar as tolerâncias relativas aos agendamentos (limites de tempo para fazer *check-in*, para remover o mesmo ou alterar os dados respondidos no formulário) e alterar permissões dos Clientes, restringindo a quantidade de agendamentos possíveis ou outras ações sobre os mesmos. A ordem dos passos com que o agendamento é feito também poderá ser alterada por um elemento do *Staff*.

Por último, tal como os Clientes, o *Staff* terá acesso à sua página de perfil e às notificações que lhe estiverem associadas. Apesar deste tipo de utilizadores suportar várias ações, estas poderão ser restringidas através da aplicação de regras, dando a possibilidade de definir vários elementos do *Staff* com limitações diferentes. Estas regras serão abordadas na sub-secção seguinte.

Staff Admin

São os administradores do sistema. São os responsáveis por gerir todo o *Staff* e todos os Clientes, por gerir as localizações e os formulários que surgirão no registo de novos Clientes. Tal como o *Staff*, poderão fazer uma gestão de configurações que se tornarão as configurações por defeito em cada localização (exceto quando estas são alteradas na página da própria localização). Caberá também ao administrador preencher os dados referentes à configuração do email para o envio de notificações e a criação da chave de sincronização com o BQM. A configuração do *widget* para colocar em páginas externas é também da sua responsabilidade. Por forma a ter acesso a vários dados estatísticos, o *Staff Admin* terá acesso a um *dashboard* onde poderá consultar dados de todas as localizações do sistema. Apesar de ser administrador, este poderá também ser *Staff* numa localização que poderá ser selecionada em qualquer instante, prestando o mesmo tipo de funções de um colaborador. Dentro do conjunto de administradores, poderão também ser

definidas regras que limitam a ação de cada um deles.

A Figura 4.1 mostra uma visão geral de todos os *use cases* do sistema, cuja função será descrita na secção 4.3.

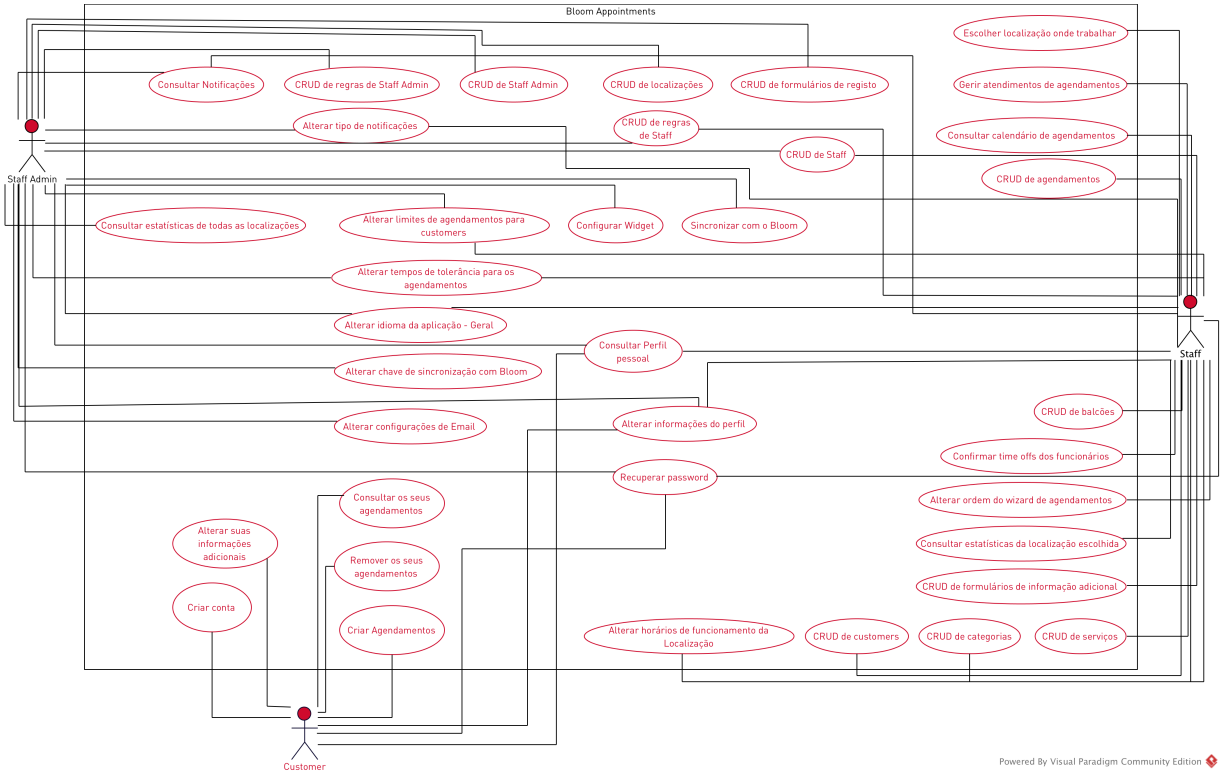


Figura 4.1: Visão dos *use cases* do sistema

4.1.3 Regras

Apesar da hierarquia de utilizadores, deverão existir também diferentes restrições para os elementos pertencentes ao *Staff* e ao *Staff Admin*. Do total de regras unitárias previamente definidas, poderão ser formados conjuntos de regras específicos para cada cargo que seja desempenhado na localização.

Relativamente aos administradores, estas devem restringir alterações de localizações, de formulários de registo, de regras e dos próprios elementos do *Staff* e alterações nos agendamentos.

Quanto ao *Staff*, as restrições passarão pela edição de serviços e seus formulários,pelos balcões e Clientes, pela gestão de agendamentos, pela gestão da localização e administração de outros elementos do *Staff*.

Resta referir que o mesmo utilizador só pode ter um conjunto de regras associado, podendo este ser distinto por

cada localização onde esteja inserido.

4.1.4 Agendamentos

Os agendamentos poderão ser marcados quer pelos Clientes, quer pelo *Staff* habilitado para tal. O processo de marcação de agendamentos deverá ser configurável em cada localização por um elemento do *Staff* que possua a respetiva permissão. No entanto, deve seguir os passos obrigatórios representados no diagrama de sequência da Figura 4.2 e descritos de seguida:

- Escolha da localização como primeiro passo, sendo que deverá ser automaticamente preenchido no caso de só existir uma localização;

- O segundo e terceiro passo (caso aplicável) responderão à organização da localização. Assim, a escolha do serviço ou escolha do balcão/colaborador será opcional bem como a ordem de apresentação destes. Uma vez que as associações entre entidades para os agendamentos são apenas entre serviços e balcões ou serviços e colaboradores, não deve ser possível a escolha do balcão e do colaborador em simultâneo;

- Escolha da data e hora como passo seguinte e obrigatório;

- Por fim, antes do ecrã de confirmação, caso o serviço tenha algum tipo de formulário adicional associado, este deverá ser preenchido neste momento.

Após marcação do agendamento, este passará para o estado *"Por confirmar"*, sendo esta confirmação feita por qualquer *Staff* ou *Staff Admin*, desde que associado à localização em questão. No entanto, esta marcação poderá passar automaticamente para o estado *"Confirmado"* caso assim seja configurado na respetiva página de configuração da localização. Em qualquer das situações, o Cliente receberá um código que lhe permitirá realizar o *check-in*.

Uma vez presente no local, o Cliente poderá solicitar a um elemento do *Staff* para que processe o seu *check-in* ou então, poderá efetuar o mesmo num dispensador de senhas instalado na localização. A marcação passará então a integrar a lista de marcações passíveis de serem chamadas. A gestão de chamadas das marcações estará disponível numa *dashboard* própria, que permitirá que o elemento do *Staff* proceda à chamada, ao término da mesma, ou ao seu cancelamento. Todos os estados deverão ser gravados, por forma a integrarem os dados estatísticos do sistema.

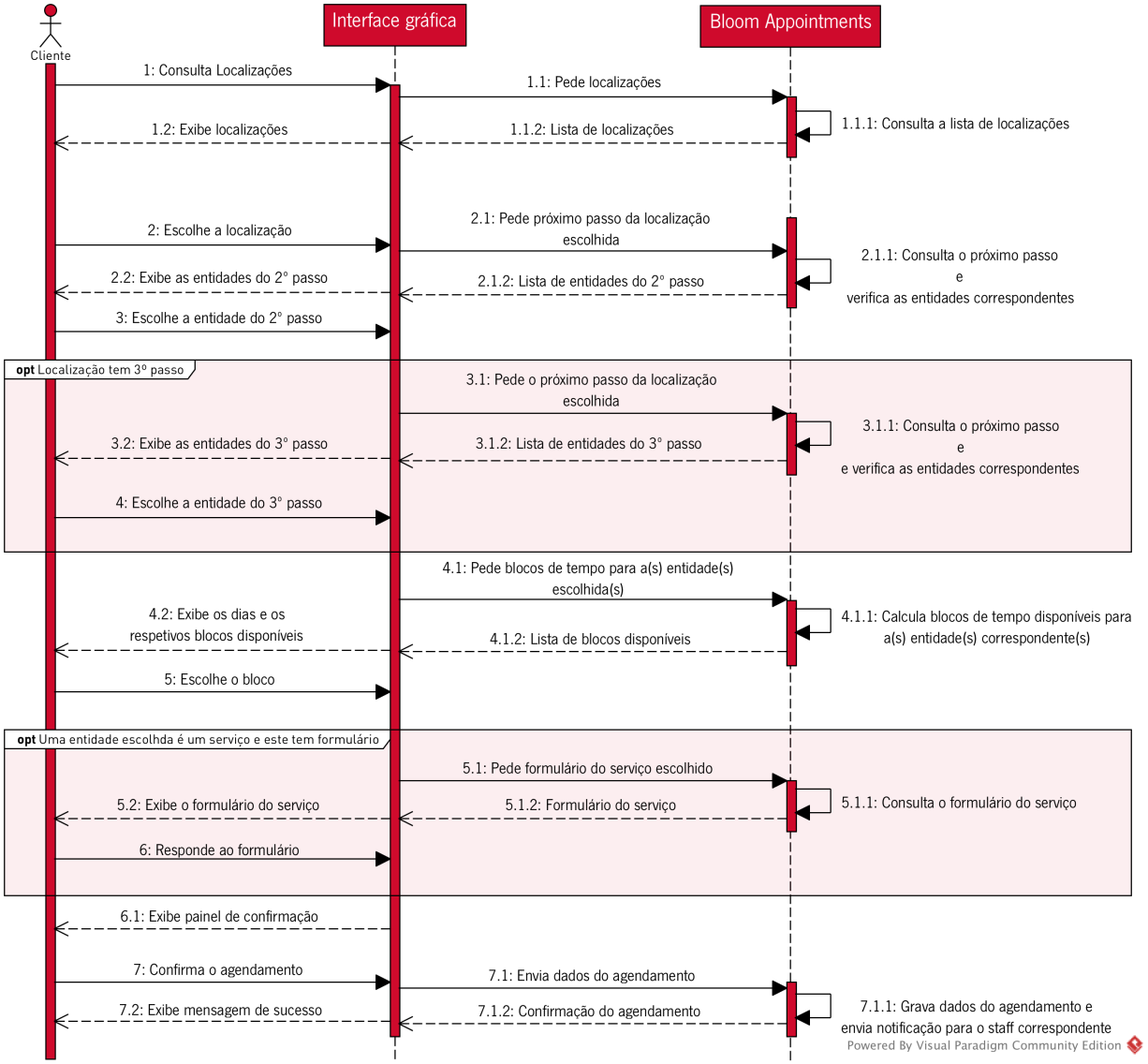


Figura 4.2: Diagrama de sequência de alto nível da criação de um agendamento

4.1.5 Widget

O sistema deverá disponibilizar um *widget* em HTML por forma a que as organizações/instituições utilizadoras do *Bloom Appointments* permitam que por exemplo, na sua página oficial, seja possível criar agendamentos sem necessidade de registo. Este *widget* deverá ser configurável, fazendo com que respeite as paletas de cores das organizações/instituições. Esta configuração apenas estará acessível a utilizadores do tipo *Staff Admin*.

4.1.6 Integração com o Bloom Queue Management

O *Bloom Appointments*, além do seu modo de funcionamento *standalone*, deverá permitir a integração com o software de gestão de filas BQM. Este tipo de integração permite que os Clientes tenham acesso prioritário numa fila de espera, caso tenham efetuado um agendamento para uma determinada data. A sincronização entre as duas aplicações deverá ser feita de um modo bidirecional, embora o tipo de informação seja diferente, mediante o sentido da comunicação: o BQM será o responsável por enviar informações sobre os seus utilizadores, serviços e balcões, por forma a que estes sejam criados (ou atualizados após a primeira sincronização) no *Bloom Appointments*. Por outro lado, a aplicação de agendamentos estará responsável por enviar informações sobre os agendamentos existentes para um determinado intervalo de tempo, tornando assim possível que o *check-in* possa ser feito localmente no BQM, evitando que possíveis falhas de comunicação não permitam a entrada do Cliente na fila com a devida prioridade.

O canal de sincronização deverá ser configurado a partir do BQM, sendo necessária a indicação do endereço da aplicação de agendamentos, a porta de comunicação, a chave de acesso gerada no painel de gestão do *Bloom Appointments* e as credenciais de acesso de um utilizador do tipo *Staff Admin*.

Após o primeiro *handshake* que deve incluir a verificação das credenciais, devem ser listadas as localizações geridas pelo respetivo *Staff Admin* por forma a que seja possível escolher qual a localização a sincronizar. No caso em que a localização não esteja já criada no *Bloom Appointments*, deverá ser possível que o utilizador crie a mesma, sendo atribuído a esta um horário de funcionamento por defeito (9h-18h por exemplo). Após a escolha da localização, deve ser mostrada uma lista com as regras disponíveis para atribuição a elementos do *Staff*, para que os utilizadores sincronizados tenham essa regra por defeito. Todas as entidades sincronizadas terão um horário de funcionamento igual ao horário da localização onde foram sincronizados, podendo ser alterado posteriormente a partir do *Bloom Appointments*.

Após o envio destas informações, o BQM deverá iniciar um processo responsável por fazer um pedido de listagem de agendamentos de uma forma periódica, com um intervalo de tempo reduzido. O *Bloom Appointments* deverá listar os agendamentos mediante um intervalo dado, enviando também informações sobre os tempos de tolerância, por forma a que o *check-in* do Cliente seja respeitado.

O *check-in* passará também a estar disponível nos dispensadores de senhas com ecrã touch, bastando indicar o código obtido no final do processo de marcação. Após o *check-in*, será gerada a respetiva senha e o BQM deverá comunicar ao *Bloom Appointments* de que foi feito um determinado *check-in*, procedendo assim ao pedido de atualização

de estados dos agendamentos.

Esta sincronização permite também que todos os elementos do *Staff* que estejam sincronizados com o BQM tenham ativo o *dashboard* para gestão de marcações, possam ter acesso direto ao *QM-PAD* respetivo e possam desta forma fazer a gestão de todos os atendimentos na mesma interface, sejam senhas associadas a marcações ou não. Caso a senha esteja associada a uma marcação, deverá receber essa notificação no ecrã, estando listados também todos os campos de informação adicional preenchidos (caso existam) pelo Cliente.

4.1.7 Integração com o Bloom Enterprise Server

Por forma a complementar a integração com todos os produtos da suite, é ainda possível o BQM estar ligado ao BES e ao *Bloom Appointments* através de conetores. Após a ligação estar concluída, os objetos passíveis de sincronização passam a ter essa informação mediante o modo escolhido. Caso o modo *"Super"* esteja selecionado, o *Bloom Appointments* e o BQM passam a obter os nomes e restante informação das entidades a partir do BES. No caso do modo escolhido ser o modo *"Normal"*, todas as informações passam do BQM para o servidor e embora seja possível haver informação repetida (ex. o mesmo nome de serviço em BQMs diferentes), estes serão considerados serviços distintos. No entanto, existe um modo intermédio onde no BQM, ao invés de ter os valores dados diretamente pelo servidor, estes serão listados no BQM, ficando ao critério do administrador a escolha da opção pretendida.

4.1.8 Sistema de notificações

O *Bloom Appointments* deverá estar dotado da capacidade do envio de notificações para todos os utilizadores. As notificações deverão ser configuráveis por utilizador através de um painel de configuração onde estarão listados todos os tipos de notificações e o seu modo de alerta. Além dos alertas obrigatórios por push, deve ser possível que o utilizador escolha se pretende receber as notificações por email ou por notificações do browser.

4.2 Modelo de Domínio

Com a análise de todos os requisitos da aplicação é possível fazer uma definição de todos os conceitos principais para o funcionamento na mesma.

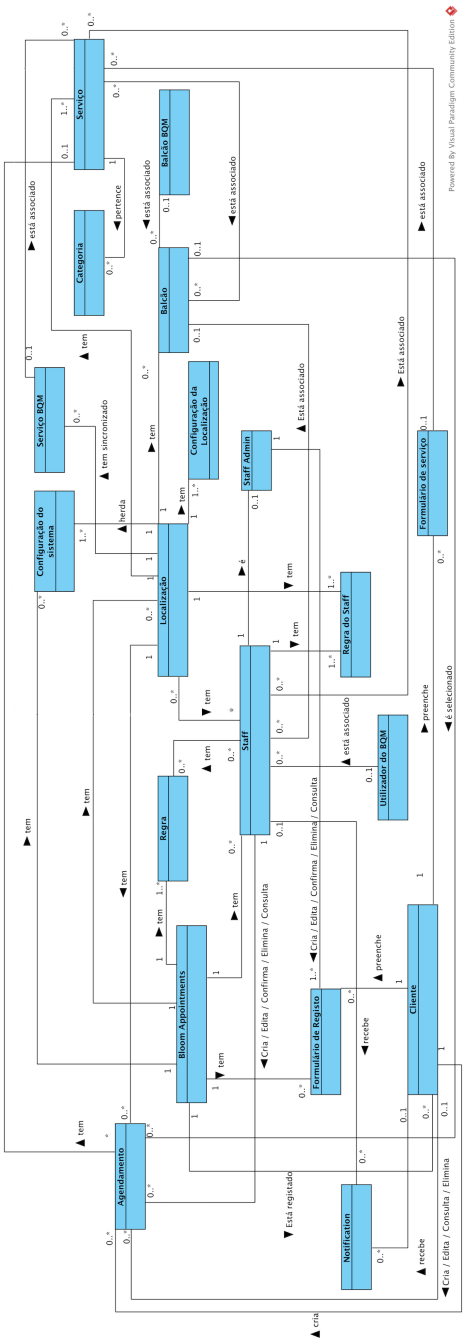


Figura 4.3: Modelo de domínio do Bloom Appointments

A Figura 4.3 mostra a representação gráfica dos conceitos principais e das relações entre eles. Considerando o conceito do sistema *Bloom Appointments*, pode-se encontrar as relações diretas com os vários tipos de *utilizadores do sistema*, com as *regras* e com as *localizações*. Dentro de cada tipo de utilizador, é possível observar quais as suas principais funções dentro do sistema bem como configurações que estejam possibilitados a alterar. Por fim, também se podem encontrar as várias entidades passíveis de marcação, incluíndo a relação entre elas, nomeadamente a associação entre balcões e serviços e elementos do *Staff*.

4.3 Use cases

Após o levantamento das funcionalidades da aplicação através de exaustivas tarefas de análise, foram criados os *use cases*. Estes permitem capturar de uma forma intuitiva e sistemática os requisitos funcionais do sistema, uma vez que ajuda a detetar todas as funcionalidades requeridas pelos utilizadores. Estes indicam que serviços deve o sistema fornecer e a quem os deve fornecer, permitindo identificar melhor as tarefas que são os objetivos dos utilizadores do sistema. Além disso, especificam também todas as possíveis utilizações do sistema e , uma vez que se tratam de instrumentos de diálogo entre Clientes e projetistas, permitem desenvolver protótipos da interface da aplicação com o utilizador.

A Figura 4.1 mostrada anteriormente mostra uma visão geral de todo o sistema, por forma a que seja possível ter uma maior noção da extensibilidade das ações que têm de ser implementadas. Desse diagrama de *use cases* geral, é possível dividir em três mais específicos: um diagrama para o *Cliente*, um diagrama para o *Staff* e por fim, um diagrama para o *Staff Admin*.

A Figura 4.4 trata-se do diagrama de *use cases* específico do Cliente onde mostra as possíveis interações que um Cliente pode ter com o *Bloom Appointments*. Os restantes diagramas podem ser consultados no Apêndice A.

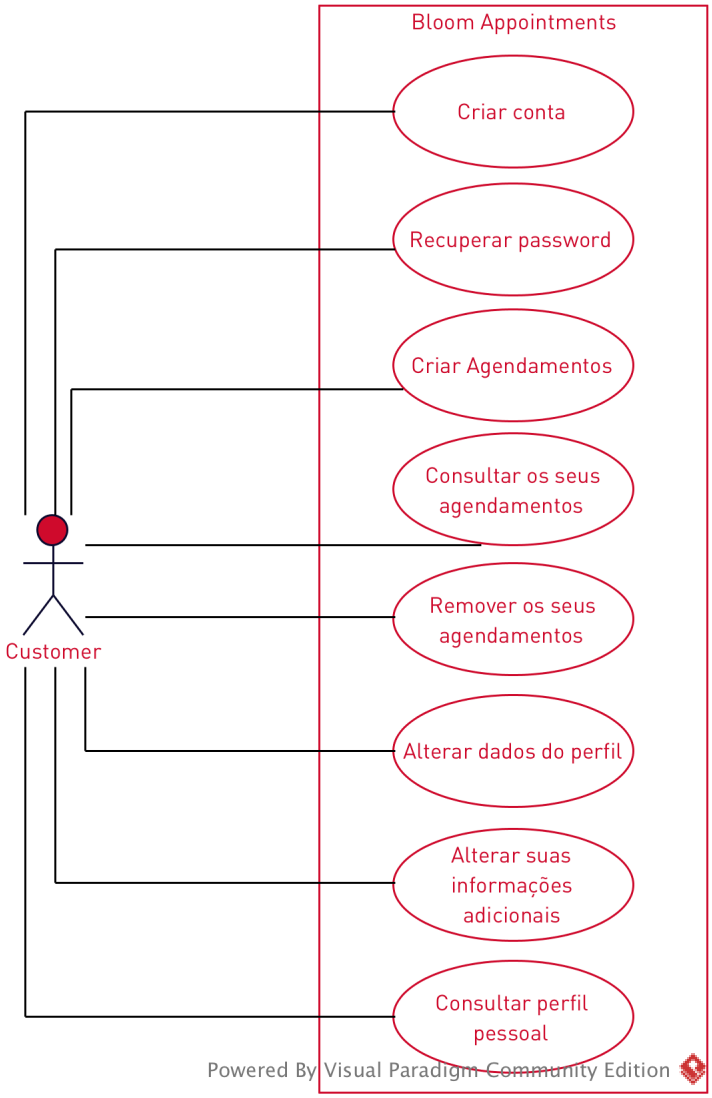


Figura 4.4: Visão dos *use cases* do *Cliente*

Após a conceção dos diagramas, foram criados os *use cases* textuais. Estes tratam-se de textos simples que fixam responsabilidades funcionais do sistema. Estas especificações textuais permitem também explorar alternativas e variações das ações no sistema. Como exemplo dessas especificações, a Figura 4.5 mostra a especificação de um dos casos de uso referente ao diagrama apresentado na Figura 4.4.

Título	Alterar dados do perfil		
Ator	Cliente		
Descrição	O cliente altera os dados do seu perfil na sua página pessoal.		
Pré-condição	O utilizador deve estar autenticado. O utilizador encontra-se na página de perfil.		
Pós-condição	SUCESSO: Os dados ficam gravados no sistema e perfil do utilizador fica atualizado com as novas informações.		
Fluxo normal de eventos		Ator	Sistema
	1	O cliente insere novas informações sobre o seu perfil	
	2	O cliente grava as alterações feitas no seu perfil carregando no botão para esse propósito	
	3		O sistema verifica validade das informações
	4		O sistema guarda as informações
	5		O sistema retorna mensagem de sucesso
Fluxo alternativo 3a		Ator	Sistema
	1		O sistema deteta que tem informações de preenchimento obrigatório vazias
	2		O sistema não guarda as informações
	3		O sistema retorna mensagem de erro

Figura 4.5: Descrição textual do *use case* do Cliente: *Alterar dados do perfil*

A especificação da Figura 4.6 mostra com detalhe o *use case* da criação de agendamentos por parte do Cliente. Esta apresenta o fluxo normal para o caso onde existe apenas uma entidade passível de marcação sem formulário associado. O primeiro fluxo alternativo descreve os passos necessários caso exista a necessidade de seleção de nova entidade para criar o agendamento, ao passo que o segundo fluxo alternativo mostra o caso de o Cliente ter selecionado um serviço que exija o preenchimento de um formulário.

Título	Criar Agendamentos		
Ator	Cliente		
Descrição	O cliente cria um novo agendamento		
Pré-condição	O utilizador deve estar registado O utilizador está no painel de marcação de agendamentos		
Pós-condição	SUCESSO: É criado um novo agendamento no sistema e o cliente consegue ver o mesmo na sua dashboard		
Fluxo normal de eventos		Ator	Sistema
	1	O cliente escolhe a localização que pretende	
	2		O sistema regista a localização
	3		O sistema consulta o próximo passo
	4		O sistema devolve as entidades correspondentes
	5	O cliente escolhe a entidade que pretende	
	6		O sistema regista a entidade
	7		O sistema consulta os blocos de tempo disponíveis para a entidade seleccionada
	8		O sistema devolve os blocos de tempo disponíveis
	9	O cliente escolhe o bloco que pretende	
	10		O sistema mostra painel de confirmação
	11	O cliente confirma o agendamento	
	12		O sistema cria o agendamento
	13		O sistema envia notificações para o staff respectivo
	14		O sistema envia notificação de sucesso
Fluxo alternativo 7a		Ator	Sistema
	1		O sistema consulta o próximo passo
	2		O sistema devolve as entidades correspondentes
	3	O cliente escolhe a entidade que pretende	
	4		O sistema regista a entidade
	5		O sistema consulta os blocos de tempo disponíveis para a entidade seleccionada
Fluxo alternativo 10a		Ator	Sistema
	1		O sistema consulta o formulário do serviço
	2		O sistema devolve o formulário
	3	O cliente preenche o formulário	
	4		O sistema regista as respostas enviadas
	5		O sistema mostra painel de confirmação

Figura 4.6: Descrição textual do *use case* do Cliente: *Criar Agendamentos*

Este capítulo documentou o levantamento e análise de requisitos da aplicação de gestão de agendamentos usada como *case study* nesta dissertação. Inicialmente foi feito um levantamento das entidades, dos vários tipos de utilizadores do sistema e das restantes opções cuja implementação é imperativa para o funcionamento desejado do sistema.

A modelação de todos os requisitos foi feita com recurso a um modelo de domínio que apresenta as ligações entre os conceitos principais do sistema e a diagramas de *use cases* que mostram, através de uma representação gráfica, as funcionalidades associadas aos vários tipos de utilizadores.

O próximo capítulo dá continuidade a todo o processo exaustivo de análise de requisitos, mostrando de que forma a concepção de uma arquitetura orientada a serviços pode ser usada como suporte à implementação do *Bloom Appointments*.

Capítulo 5

Desenvolvimento do caso de estudo

Após a planificação de toda a arquitetura do sistema a concretização da mesma foi o passo seguinte. É neste capítulo que serão detalhados vários aspectos da sua implementação, nomeadamente ao nível das tecnologias usadas, dos vários desenvolvimentos e dos vários ambientes concebidos com recurso aos serviços criados.

Serão também mostrados vários excertos de código desenvolvidos, bem como serão realçadas também algumas das interfaces concebidas para interação com o sistema.

5.1 Componente Servidor

Nesta secção serão abordadas as decisões com relevância da implementação da arquitetura do *Bloom Appointments*. Inicialmente será apresentada a base de dados criada, incluindo uma demonstração da criação da mesma. Criada a estrutura de dados e respetivas classes, será demonstrada a camada de negócios com uma introdução à *API RESTful* desenvolvida (incluindo para tal um exemplo representativo desta) fechando assim a parte alusiva ao componente servidor da solução.

Com base no modelo de domínio e na análise de requisitos apresentada no capítulo 4, criou-se toda a estrutura na base de dados, tendo sido esta gerada através do *Migrations*, componente integrante do Laravel que com recurso a código PHP, gera todas as tabelas e respetivas ligações, criando também as classes de negócio, sendo automático todo o mapeamento ORM. Todas as classes têm 2 campos fixos *created_at* e *updated_at* criados automaticamente pelo Laravel que servem apenas como *timestamp* de controlo.

A título de exemplo, na Figura 5.1 estão representadas as tabelas *desk*, *desk_bloom*, *customer* e *staff_user*, estando o restante diagrama da base de dados no Apêndice B.

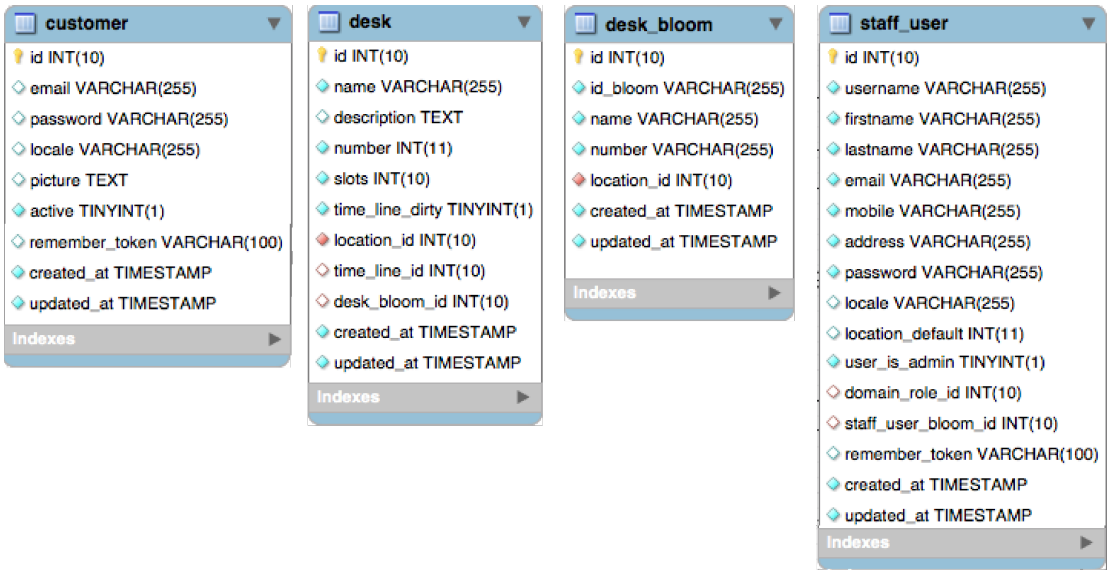


Figura 5.1: Exemplo de entidades do *Bloom Appointments*

A classe *customer* faz parte da lista das entidades mais importantes do sistema, tendo em conta que se trata da entidade correspondente aos clientes. Os vários campos que constituem esta tabela são:

- **id** : chave única que identifica o cliente no sistema;
- **email** : email do cliente;
- **password** : palavra-passe (encriptada) do cliente;
- **locale** : contém o identificador do idioma usado pelo cliente (por exemplo, *pt_PT*);
- **picture** : contém a imagem do cliente codificada no formato *base64*;
- **active** : indica o estado do cliente;
- **remember_token** : usado quando o cliente pretende o inicio de sessão automático;

A Listagem 5.1 mostra a criação da tabela *customer*, que dará origem à classe com o mesmo nome. Pode observar-se a criação dos campos acima descritos, bem como a tipagem dos mesmos.

```php
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateCustomersTable extends Migration {
  /**
   * Run the migrations.
   * @return void
   */
  public function up()
  {
    Schema::create('customer', function(Blueprint $table)
    {
          $table->increments('id');
          $table->string('email')->unique()->nullable();
          $table->string('password')->nullable();
          $table->string('locale')->nullable();
          $table->text('picture')->nullable();
          $table->boolean('active')->default(false);
          $table->rememberToken();
          $table->timestamps();
    });
  }
  /**
   * Reverse the migrations.
   * @return void
   */
  public function down()
  {
    Schema::drop('customer');
  }
}

```

Listagem 5.1: Código usado na criação da tabela (e respetiva classe) *customer*

As tabelas *desk* e *desk_bloom* são também de elevada importância no sistema, uma vez que são usadas para guardar informações dos balcões, sendo a última está relacionada com os balcões provenientes do BQM. A tabela *desk* é composta por:

- **id**: é a chave única que identifica o balcão no sistema;
- **name**: o nome do balcão;
- **description**: descrição textual do balcão;
- **number**: número identificativo do balcão;
- **location_id**: chave estrangeira da localização onde o balcão se encontra;
- **desk_bloom_id**: chave estrangeira associada ao *id* da tabela que representa o balcão proveniente do BQM;

- **slots, time_line_dirty, time_line_id**: permitem guardar informações sobre os blocos de tempo passíveis de marcação de um agendamento.

Já a tabela *desk_bloom* contém menos informação uma vez que serve apenas como mapeamento entre a entidade do *Bloom Appointments* e a do BQM. Apesar de não ter os campos associados a marcações e o campo da descrição, tem o campo com o *id_bloom* que serve para guardar o *id* proveniente do BQM.

Por fim, a tabela *staff_user* contém a informação dos elementos do *Staff*, sendo esta obviamente importante para o funcionamento do sistema. Eis os campos que a constitui:

- **id**: chave única que identifica o *Staff*;
- **username**: nome de utilizador com o qual fará login;
- **firstname** e **lastname**: primeiro e último nome respetivamente;
- **email**, **mobile**: contactos do elemento do *Staff*;
- **address**: morada do *Staff*;
- **password**: palavra-passe (encriptada) deste;
- **locale**: tal como no *customer*, contém o identificador do idioma usado;
- **location_default**: contém o *id* da localização que este utilizador pretenda abrir por defeito sempre que iniciar sessão;
- **user_is_admin**: indica se pertence ao grupo de utilizadores *Staff Admin*;
- **domain_role_id**: chave estrangeira associada ao *id* da *role* que lhe foi atribuída;
- **staff_user_bloom_id**: chave estrangeira associada ao *id* da tabela que representa os utilizadores proveniente do BQM;
- **remember_token**: do mesmo modo que no cliente, é usado para gravar o início automático da sessão;

Após a criação de todas as classes necessárias, foi então desenvolvida toda a camada de negócio que segue a filosofia da temática desta dissertação. Os serviços disponibilizados através da *API RESTful* de toda a solução contemplam os requisitos apontados no capítulo anterior.

Para categorizar os serviços (nomeadamente no seu endereçamento), foram criadas várias categorias:

- **Clientes** : contém os serviços associados às ações dos clientes com o sistema. Os serviços disponibilizados variam desde a obtenção de dados, quer para a página de perfil, quer para o registo, quer para as listas de notificações e de agendamentos, passando pela possibilidade de gravar informações do seu perfil ou até mesmo para a gestão de notificações;

- **Staff** : é nesta categoria que se encontram os serviços associados aos utilizadores do tipo *Staff*, tais como listagem de informação para preenchimento da *dashboard*, informações sobre as suas localizações, o CRUD de serviços e formulários de informação adicional, balcões e clientes. Tem também os serviços necessários para fazer operações sobre os agendamentos à excepção da sua criação;

- **Staff Admin** : esta categoria serve como complemento à categoria *Staff*, uma vez que as restantes configurações do sistema ficam ao encargo dos utilizadores desta categoria. Sendo assim, encontram-se aqui os serviços de CRUD de localizações, regras, de elementos do *Staff*, formulários de registos, configurações do *Widget* e do sistema;

- **Schedule** : categoria que agrega todos os serviços necessários para a criação de um agendamento e para o seu *check-in*. A grande maioria dos serviços são de listagem de informações, uma vez que os únicos passos onde existe um envio da informação são na confirmação final do agendamento e no momento do *check-in*. Agrega a listagem de localizações, serviços, balcões, colaboradores e informações sobre os blocos de tempo disponíveis. Tem também um serviço que permite, na escolha do cliente para o agendamento, obter todas as informações adicionais do mesmo para o pré-preenchimento do formulário;

- **Integração com o BQM** : de todas as categorias, é a única que não tem qualquer interface visual associada. Esta contém todos os serviços necessários para a integração com o BQM.

Para garantir autenticidade em todos os pedidos do tipo POST (independentemente da categoria a que pertençam), estes requerem o argumento *token* que pode ser obtido, a par do *id* do servidor, por um pedido GET ao endereço *http://baseurl/token*.

Como exemplo de implementação das categorias, será demonstrada a categoria *Integração com o BQM*, podendo ser encontrado no Apêndice C uma descrição mais pormenorizada dos serviços disponibilizados por cada categoria restante.

O diagrama de sequência representado na Figura 5.2 demonstra a sequência de passos necessários para criar o conetor entre o BQM e o *Bloom Appointments*, abrindo assim possibilidade de interação entre os 2 sistemas.

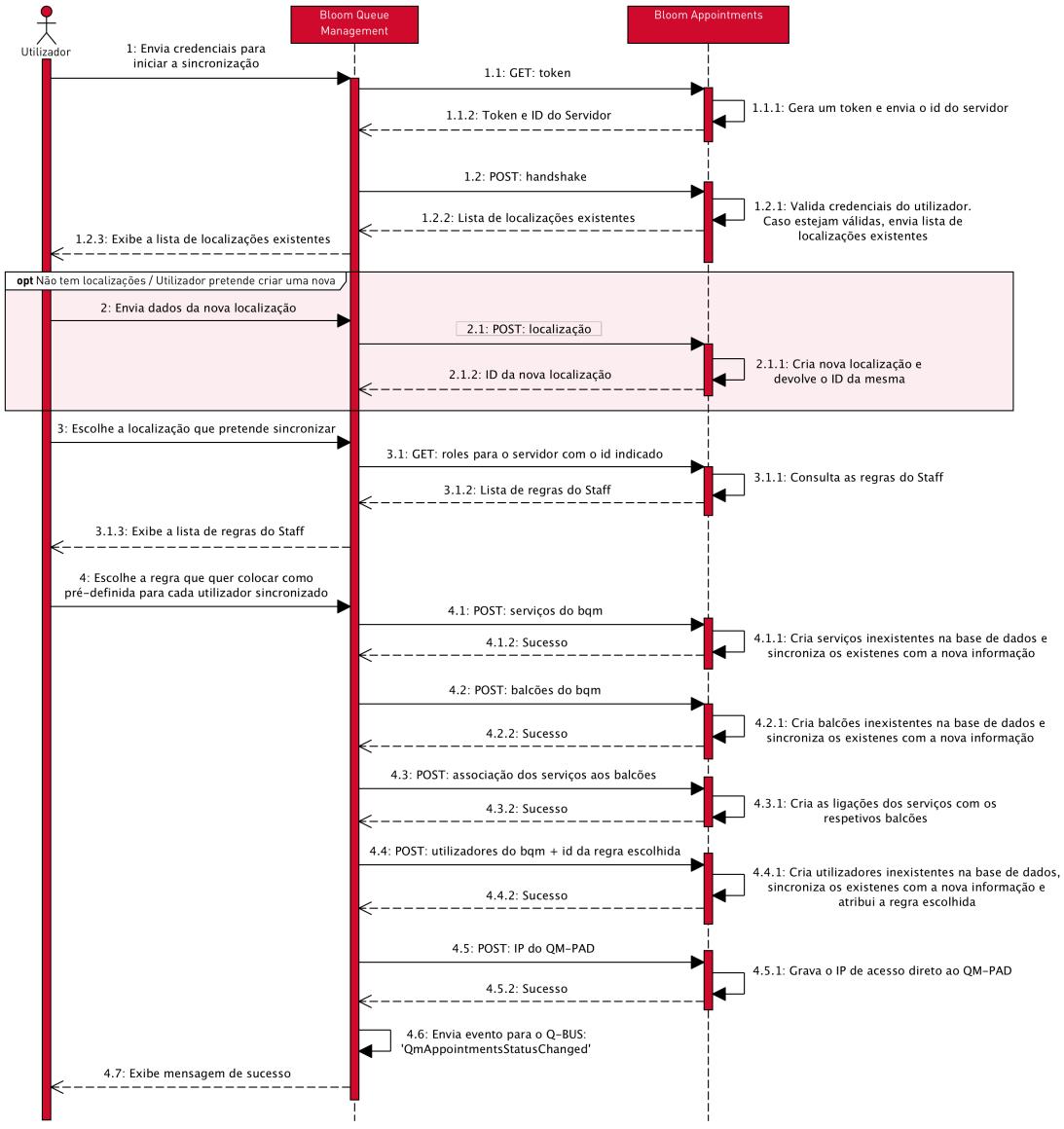


Figura 5.2: Diagrama de sequência da integração do *Bloom Appointments* com o BQM

Para respeitar as indicações do diagrama, foram então desenvolvidos os serviços para completar todas as operações necessárias. Todos os pedidos HTTP a esta *API RESTful* estão apresentados na Tabela 5.3 e estão categorizados mediante a sua função.

Registo	
POST	/api/{version}/{format}/handshake
GET	/api/{version}/{format}/roles?server_id=[server_id]
POST	/api/{version}/{format}/location
Criação / Sincronização de entidades	
POST	/api/{version}/{format}/bloomservices
POST	/api/{version}/{format}/bloomdesks
POST	/api/{version}/{format}/bloomassoc
POST	/api/{version}/{format}/bloomusers
POST	/api/{version}/{format}/qmpadip
Agendamentos	
POST	/api/{version}/{format}/checkinappointment
POST	/schedule/getmenuconf/api/{version}/{format}/changestatusig
GET	/api/{version}/{format}/appointments?startDate=[Y-m-d]&endDate=[Y-m-d]&lastUpdate=[Y-m-d]
Cancelar o Registo	
POST	/api/{version}/{format}/unregister

Figura 5.3: *API RESTful* para integração com o BQM

Na categoria Registo, encontra-se o pedido *handshake* que permite que seja dado o primeiro passo nas comunicações entre os dois sistemas. Para obter resposta positiva, deverão ser enviados como argumentos os dados de um utilizador do tipo *Staff Admin*. Uma vez validados os dados, será devolvida como resposta a lista de localizações existentes para que no BQM seja definida a localização que vai sincronizar. O pedido *location* foi criado para que seja possível criar uma nova localização a partir do BQM, devolvendo o *id* da mesma no caso do pedido ser feito corretamente. Por fim, o pedido *roles* serve para obter a listagem de regras da instalação indicada no argumento *server_id*,

para que os utilizadores sincronizados do BQM tenham automaticamente uma regra básica para que possam trabalhar no imediato no sistema.

Relativamente à categoria Criação / Sincronização de entidades, todos os pedidos são do mesmo tipo (POST), bastando que para cada um deles, seja enviado em formato JSON os valores respectivos:

- *bloomservices* : *Array* que contenha o *id*, a *tag*, o *nome*, a *cor*, e as *horas de funcionamento* de cada serviço;
- *bloomdesks* : *Array* que contenha o *id*, a *tag* e o *nome* de cada balcão;
- *bloomassoc* : *Array* que associa o *id* dos serviços a cada *id* do balcão que o atenda;
- *bloomusers* : *Array* que contenha o *id*, o *nome de utilizador* e a *palavra-passe* dos utilizadores pretendidos;
- *qmpadip* : endereço completo do *QM-PAD* (ferramenta de chamada de senhas do BQM) apenas para colocar atalho direto na *dashboard* do *Staff*.

Além das operações demonstradas, foi também necessário criar operações que permitam que a informação circule entre ambos os sistemas, quer para o *Bloom Appointments* enviar informações sobre os agendamentos existentes num intervalo de datas (pedido *appointments*), quer para o BQM enviar informações sobre check-ins efetuados (pedido *checkinappointment*) e sobre alteração do estado de um determinado agendamento (pedido *changestatus*).

Para a implementação dos vários pedidos, é necessário indicar primeiro a sua rota, por forma a que o pedido seja mapeado para o Controller e função respetivos. Assim, tal como demonstrado na Listagem 5.2, inicia-se com a definição do tipo do pedido, indicando de seguida o endereço respectivo. A indicação {*version?*}/{*format?*} permite receber esses campos como argumentos de função, sendo usados para validação dos mesmos. Neste caso, tornam-se úteis para verificar quer a versão introduzida no endereço que permite manter a retro-compatibilidade da API após o lançamento de novas versões, quer o formato do *output* que permite optar por mostrar a informação em XML ou JSON.

No início da função é criado um objeto do tipo *ApiBloom* que é a classe que contém todas as funções associadas à integração com o BQM, sendo de seguida verificada a versão indicada no endereço. Para cada versão, é devolvido o resultado da chamada da função respetiva, neste caso, a função *apiBloomDesks* que permite criar/sincronizar balcões do BQM no *Bloom Appointments*. Se for indicada de uma versão errada, é devolvida uma mensagem de erro.

```
1  <?php
2  Route::post('/api/{version?}/{format?}/bloomdesks',function(Request $request,$
       version='last',$format='json'){
3    $function=new ApiBloom();
4      switch($version){
5      case "last":
6      case "0.0":
7        return $function->apiBloomDesks($request,$format);
8      default:
9        error_log($format);
10       if(strcmp($format,'json')==0){
11         return Response::json(['success'=>false,'errno'=>"1"]);
12       }else{
13         return Restable::single(['success'=>false,'errno'=>"1"])->render('xml');
14       }
15   }
16 });
17 Route::post('/api/{format?}/bloomdesks','ApiBloom@apiBloomDesks');
```

Listagem 5.2: Definição de *routing* do pedido POST para criação/sincronização de balcões do BQM no *Bloom Appointments*

Uma vez definida a rota, é então implementada a função *ApiBloomDesks*, tal como demonstra a Listagem 5.3. Neste caso, o pedido requer como parâmetros o *id da localização* e um *array codificado em JSON* que contenha objetos com o *id* do balcão, a sua *tag* (número) e *descrição*.

As linhas 2 e 3 permitem obter os dados recebidos como parâmetros do pedido e de seguida, fazer a descodificação do JSON associado aos balcões. A linha seguinte indica o início de uma transação na base de dados, seguindo as indicações dadas na sub-secção 3.2.2.

A condição da linha 5 verifica se foram enviados balcões no parâmetro respetivo. No caso positivo, inicia-se um ciclo iterativo que percorre todos os balcões recebidos e para cada um deles verifica se já existe quer na tabela da listagem de balcões do BQM (linha 7), quer na tabela da listagem de balcões do*Bloom Appointments* (linha 22). Caso não exista no primeiro caso, é então criado um novo balcão, sendo de imediato preenchido o *id* vindo do BQM (linhas 8 a 11). As linhas seguintes (12 a 14) mostram que o objeto *deskBloom* é preenchido com valores recebidos nos parâmetros e gravado de seguida. Caso a gravação falhe, é feito um *rollback* da transação e devolvida uma mensagem de erro no formato indicado no endereço (linhas 15 a 21). O objeto *deskAppoint* contém o balcão do sistema de agendamentos e segue os mesmos passos do objeto anterior. É feita inicialmente uma verificação da existência ou não do mesmo na base de dados (linha 22) e caso não exista, é criado um novo (linhas 23 e 24). De seguida, os seus campos são preenchidos (linhas 25 a 29) e o objeto é posteriormente gravado, sendo feita também a verificação de sucesso ou insucesso da gravação (linhas 30 a 36).

Terminado o ciclo, é feito o *commit* que executa a transação e posteriormente é devolvido conforme o formato

indicado a indicação de sucesso (linhas 38 a 43). Caso não tenham sido enviados balcões no parâmetro devido, é feito um *rollback* e é devolvida a mensagem de erro respetiva (linhas 44 a 49).

```
1  public function apiBloomDesks(Request $request,$format='json'){
2    $data=$request->only('location','desks');
3    $desks=json_decode($data['desks']);
4    DB::beginTransaction();
5    if($desks && count($desks)>0){
6      foreach($desks as $desk){
7        $deskBloom=DeskBloom::all()->where('id_bloom',$desk->id)->first();
8        if(!$deskBloom){
9          $deskBloom=new DeskBloom();
10         $deskBloom->id_bloom=$desk->id;
11       }
12       $deskBloom->name=$desk->description;
13       $deskBloom->number=$desk->tag;
14       $deskBloom->location_id=intval($data['location']);
15       if(!$deskBloom->save()){
16         DB::rollback();
17         if(strcmp('json',$format)==0)
18           return Response::json(['success'=>false,'errno'=>"4"]);
19         else
20           return Restable::single(['success'=>false,'errno'=>"4"])->render('xml')
                 ;
21       }
22       $deskAppoint=Desk::all()->where('desk_bloom_id',$deskBloom->id)->first();
23       if(!$deskAppoint)
24         $deskAppoint=new Desk();
25       $deskAppoint->name=$deskBloom->name;
26       $deskAppoint->number=$deskBloom->number;
27       $deskAppoint->name=$deskBloom->name;
28       $deskAppoint->location_id=$deskBloom->location_id;
29       $deskAppoint->desk_bloom_id=$deskBloom->id;
30       if(!$deskAppoint->save()){
31         DB::rollback();
32         if(strcmp('json',$format)==0)
33           return Response::json(['success'=>false,'errno'=>"4"]);
34         else
35           return Restable::single(['success'=>false,'errno'=>"4"])->render('xml')
                 ;
36       }
37     }
38     DB::commit();
39     if(strcmp($format,'json')==0)
40       return Response::json(['success'=>true]);
41     else
42       return Restable::single(['success'=>true])->render('xml');
43   }else{
44     DB::rollback();
45     if(strcmp('json',$format)==0)
46       return Response::json(['success'=>false,'errno'=>"4"]);
47     else
48       return Restable::single(['success'=>false,'errno'=>"4"])->render('xml');
49   }
50 }
```

```
51 ?>
```

Listagem 5.3: Implementação do pedido POST para criação/sincronização de balcões do BQM no *Bloom Appointments*

Uma vez implementada a arquitetura de suporte à aplicação de agendamentos online, é então desenvolvido a interface gráfica da aplicação. Devido à enorme extensibilidade da solução, a operação abordada na secção seguinte foi escolhida para representar os vários tipos de operações executadas com recurso aos serviços disponibilizados.

5.2 Interface gráfica

Tal como está indicado no capítulo anterior, para esta aplicação foi desenvolvido uma interface gráfica capaz de dar resposta aos vários *use cases* definidos para os vários tipos de utilizador através de uma interface gráfica. Este foi implementado de forma a consumir os serviços desenvolvidos e demonstrados na secção anterior.

Sendo assim, nesta secção estão descritos os principais detalhes da implementação da interface gráfica do *Bloom Appointments*, surgindo em primeiro lugar uma explicação sobre esta e por fim, serão demonstradas algumas das interfaces quer do cliente final da solução, quer do próprio *Staff*.

Seguindo a arquitetura lógica apresentada na Figura 3.2, todo a interface gráfica da aplicação foi concebida usando como principais linguagens o HTML e *JavaScript*, com recurso à framework AngularJS.

Para a construção da interface gráfica foram definidas várias Views, estando estas categorizadas por:

- **Customer** - Agrupa todas as Views relacionadas com o cliente, nomeadamente a sua página de registo e de login, página de perfil e *dashboard* e o *wizard* para marcação de um agendamento;
- **Staff** - Tal como o próprio nome sugere, contém as Views pertencentes ao *Staff*: páginas para configuração de serviços, balcões, formulários, regras, definições da localização, *dashboard* e calendário para gestão de agendamentos, etc.
- **Staff Admin** - Apesar de um administrador de *Staff* poder pertencer ao próprio *Staff*, este também tem Views próprias tais como: regras do domínio, gestão de localizações, gestão do *Staff* e ainda possibilidade de gerir o formulário destinado ao registo de novos utilizadores, entre outras;
- **Dispenser** - Esta View está diretamente relacionada com o dispensador de senhas do BQM, por forma a que possam ser feitos agendamentos a partir do próprio dispositivo, ao invés da página dos clientes;

- **Widget** - A View do *Widget* é usada em dois locais distintos:

  - É usada no próprio*Bloom Appointments* quando um utilizador pretende colocar o wizard de marcação de agendamentos no seu próprio site;

  - É usada no *Bloom Concierge*, por forma a que o utilizador que tenha feito a sincronização com o BQM possa usufruir do sistema de marcações sem ter que sair do ambiente de trabalho do BQM.

Cada categoria descrita contém o seu próprio módulo AngularJS onde estão declarados os Controllers, as *factories* e as várias diretivas que possam ser necessárias em cada um dos módulos.

Tomando como exemplo a página de gestão de serviços por parte do *Staff*, serão demonstrados de seguida alguns pontos onde os serviços REST disponibilizados pelo componente servidor são utilizados.

Abertura da View

No momento em que a View é carregada, o Controller definido para esta (*StaffServicesController*) é alimentado por um pedido *GET* à API que irá devolver o conjunto de informações necessárias, estando neste caso todos os dados serializados em JSON, tal como demonstra a Listagem 5.4. Este pedido está inserido dentro da *factory StaffServices* que agrupa todas as funções necessárias para inicializar, validar, gravar e remover serviços.

```
1  StaffApp.factory('StaffServices',['$rootScope','$http','$route','$location','
       StaffAlert',function($rootScope,$http,$route,$location,StaffAlert){
2    return{
3       initServices: function () {
4            var configs={};
5            $.ajaxSetup({async:false});
6            $.get('staff/getallservices',{'user':{'id':$rootScope.user['id'],'
                 username':$rootScope.user['username']},'location':$rootScope.
                 location},
7                function(data) {
8                   configs=data;
9                });
10           return configs;
11        }
12   }]);
```

Listagem 5.4: Excerto da implementação da factory *StaffServices* num módulo de AngularJS

Gravação dos dados dos serviços

Os dados relativos aos serviços serão gravados sempre que o utilizador clicar no botão destinado a tal. Todos os valores estão colocados na variável *$scope.services*, sendo esta editada diretamente na View através do atributo *ng-model* colocado em cada input.

```
$scope.saveService= function () {
  $scope.save=true;
    showMainLoader();
    var errors=StaffServices.validateServices($scope.services);
    if(errors.length>0){
       var str='';
       errors.forEach(function (entry) {
          str+=entry+'<br/>'
       });
       hideMainLoader();
       StaffAlert.danger(str,4880);
    }else{
       var token=document.getElementById('_token').val;
       StaffServices.saveServices($scope.services,token);
    }
};
```

Listagem 5.5: Código do botão para gravar serviços

Ao clicar no botão, a primeira ação despoletada é a validação de todos os campos. As regras de validação estão definidas dentro da *factory StaffServices*. Mediante o resultado deste teste, é invocado o pedido POST para gravar todas as informações relativas ao serviços. O pedido está também definido na mesma *factory* (Listagem 5.6), bastando que para tal seja passado o *token* para validação do pedido e os dados dos serviços.

```
saveServices: function (services,token) {
  $http({
     method:'POST',
       url:'staff/saveservices',
       data:{'_token':$rootScope.token,'user':{'id':$rootScope.user['id'],'
           username':$rootScope.user['username']},'location':$rootScope.location
           ,'services':JSON.stringify(services)},
       contentType:false,
       processData: false
    }).then(function(data) {
       data=data['data'];
       hideMainLoader();
       if(data['status'])
         StaffAlert.success(data['reason']);
       else
          StaffAlert.danger(data['reason']);
       setTimeout(function(){$route.reload();},1000);
  }
);
```

Listagem 5.6: Implementação na factory *StaffServices* do pedido POST para gravar serviços

Uma vez que os dados encontram-se em forma de objeto, estes são convertidos em JSON para que possam ser enviados para o serviço correspondente, tal como demonstra a Listagem 5.7.

Após a recepção da resposta, será mostrado um alerta referente ao sucesso ou insucesso da concretização deste.

```
1  Accept: application/json, text/plain, * / *
2  Accept-Encoding: gzip, deflate
3  Accept-Language: pt-PT,pt;q=0.8,en;q=0.5,en-US;q=0.3
4  Content-Length: 3117
5  Content-Type: application/json;charset=utf-8
6  Cookie: _ga=GA1.2.244397768.1429004742(...)UxIn0
7  Host: appointments.q-better.com
8  Referer: http://appointments.q-better.com/staff
9  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:47.0) Gecko/20100
       101 Firefox/47.0
10
11 {"_token":"JlpEziKqPqXri6wDj4DK6XBgdh9BsEVuRGocvilP","user":{"id":1,"username"
       :"BloomStaff"},"location":1,"services":"[{"id":1,"name":"Exams","
       description":"","category":"0","tter":"A","appointment_confirmed":false,"
       slots":"1","duration":"01:00","staff_only":null,"form_service_id":"0","
       bloom_service_id":null,"bloom_sync":false,"color":"#d03b2f","days_available
       ":[{"type":0,"name":"All days","visible":false},{"type":1,"name":"Monday","
       visible":true},{"type":2,"name":"Tuesday","visible":true},{"type":3,"name":
       "Wednesday","visible":true},{"type":4,"name":"Thursday","visible":true},{"
       type":5,"name":"Friday","visible":true,},{"type":6,"name":"Saturday","
       visible":true,},{"type":7,"name":"Sunday","visible":true,}],"dirty":0,"save
       ":true,"hours":[{"weekday":{"type":0,"name":"All days","visible":true},"
       closed":false,"times":[{"init_time":"09:00","end_time":"18:00"}]}],"
       day_to_add":"8"}]"}
```

Listagem 5.7: Conteúdo do POST enviado para gravar informações sobre serviços

Interface gráfica - Marcação de um agendamento

A interface gráfica implementada surge para dar seguimento aos requisitos apontados na sub-secção 4.1.4.

Tomando como referência a criação de um agendamento num Hospital que disponibiliza apenas serviços passíveis de marcação (sem possibilidade de escolha de balcão e colaborador) e sem formulário, um cliente poderá fazer autonomamente um agendamento a partir da sua página pessoal ou através de um dispensador de senhas (caso o*Bloom Appointments* esteja sincronizado com o BQM).

Por forma a mostrar as potencialidades da arquitetura, serão demonstradas de seguida as duas interfaces gráficas desenvolvidas para atender às duas formas possíveis de criação de agendamentos acima indicadas. Ambas as interfaces são completamente distintas em termos gráficos. No entanto, utilizam a mesma API para obter as informações a mostrar e para criar o agendamento no sistema.

No caso em que o cliente utiliza a sua página pessoal no*Bloom Appointments*, cabe-lhe escolher a localização para onde pretende criar o agendamento, selecionando a mesma no painel demonstrado na Figura 5.4. Relativamente à interface no dispensador de senhas, esta não lista as localizações, uma vez que na sincronização entre o *Bloom Appointments* e o BQM já está a ser indicada a localização específica.

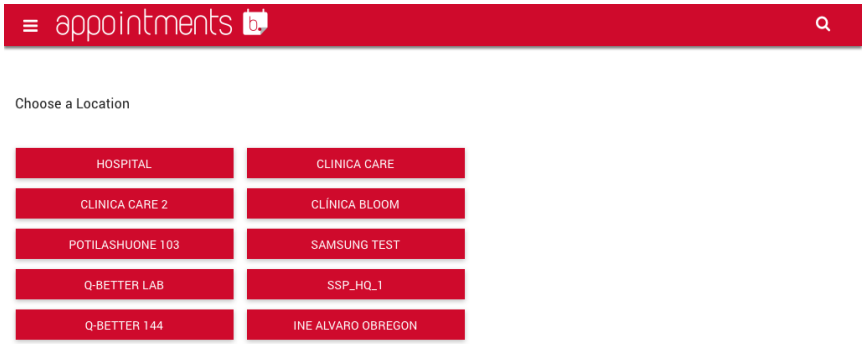


Figura 5.4: Painel para escolha de localizações por parte do cliente

Mediante a localização escolhida, são listados os serviços disponíveis. Para avançar para o passo seguinte, o cliente terá apenas de selecionar no seguinte painel (Figura 5.5) o serviço desejado. Do mesmo modo, no dispensador de senhas, o cliente apenas precisa de selecionar o serviço listado no painel da Figura 5.6.

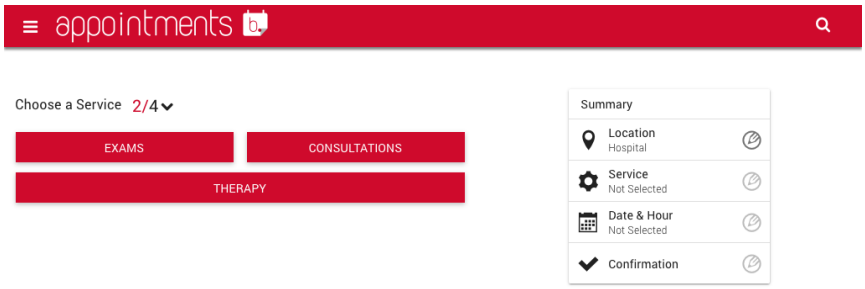


Figura 5.5: Painel para escolha de serviços por parte do cliente na sua página pessoal

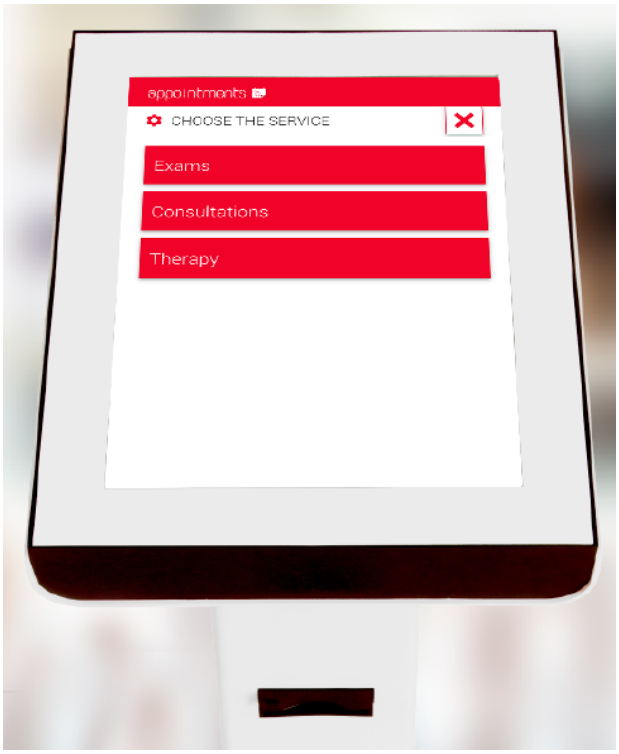


Figura 5.6: Painel para escolha de serviços por parte do cliente no dispensador de senhas

Como último passo, o cliente terá ao seu dispor um calendário com os dias disponíveis assinalados (Figura 5.7), sendo listadas as horas disponíveis após a seleção do dia (Figura 5.8).

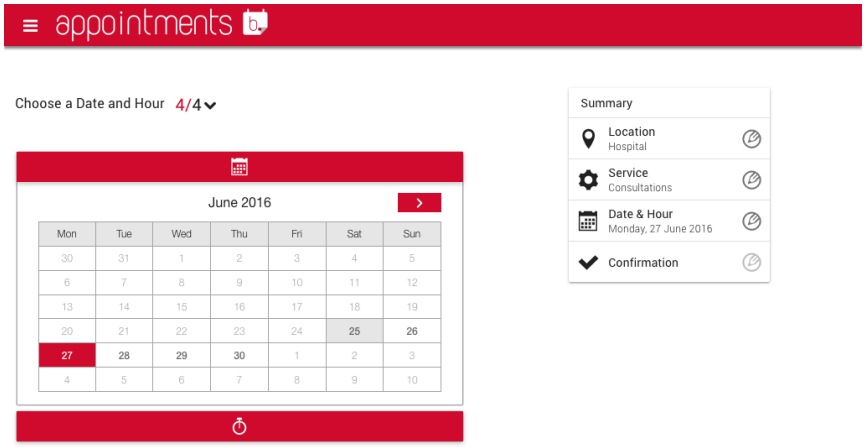


Figura 5.7: Painel para escolha da data por parte do cliente

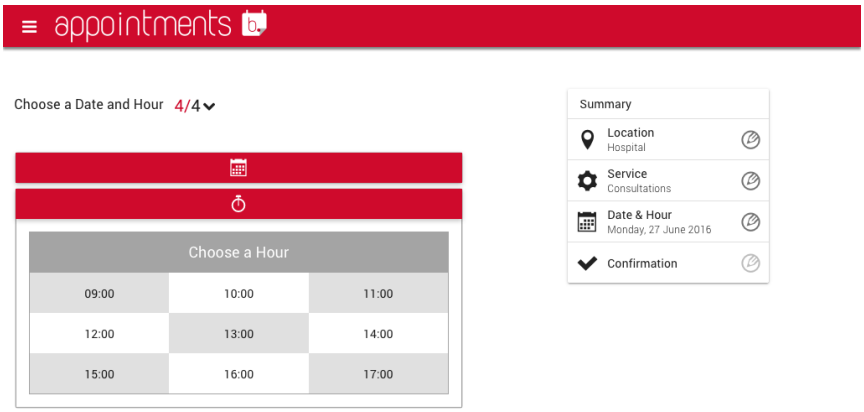


Figura 5.8: Painel para escolha da hora por parte do cliente

Comparativamente ao dispensador de senhas, os 2 passos supra-mencionados estão divididos em 2 painéis distintos, sendo mostrado inicialmente o calendário com os dias disponíveis (Figura 5.9) e posteriormente as horas disponíveis (Figura 5.10).

Figura 5.9: Painel para escolha da data por parte do cliente no dispensador de senhas

Figura 5.10: Painel para escolha da hora por parte do cliente no dispensador de senhas

Após a seleção da data e hora, é exibido no ecrã um painel de confirmação, havendo sempre a possibilidade do utilizador editar os passos anteriores, bastando para tal clicar no botão referente à opção que pretende alterar (Figura 5.11 no caso da página pessoal e Figura 5.12 no caso do dispensador de senhas).

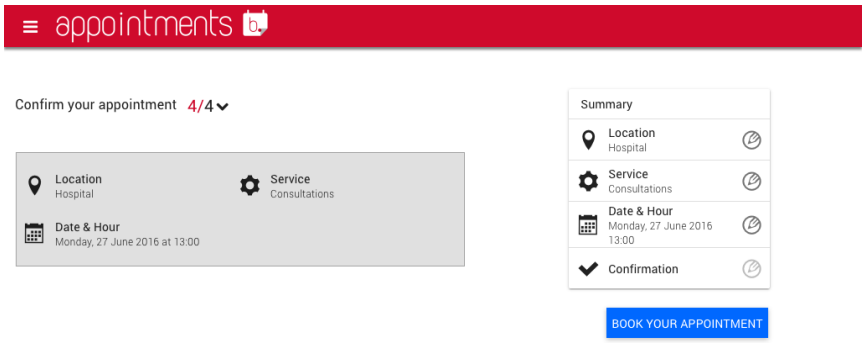


Figura 5.11: Painel de confirmação do agendamento por parte do cliente

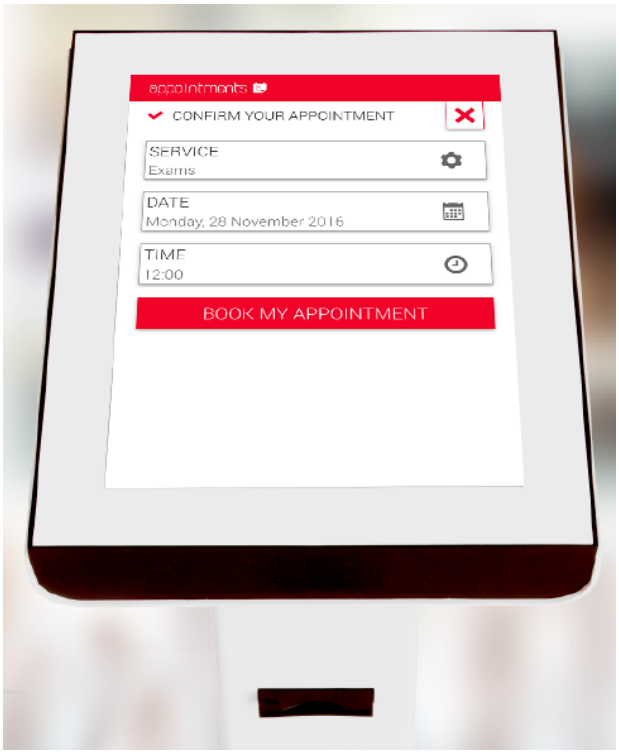


Figura 5.12: Painel de confirmação do agendamento por parte do cliente no dispensador de senhas

Quando o cliente confirma, é exibida na página pessoal a mensagem de confirmação e o agendamento é mostrado automaticamente na sua *dashboard* (Figura 5.13). No caso do dispensador de senhas, é imprimida uma senha com o código de *check-in* referente ao agendamento.

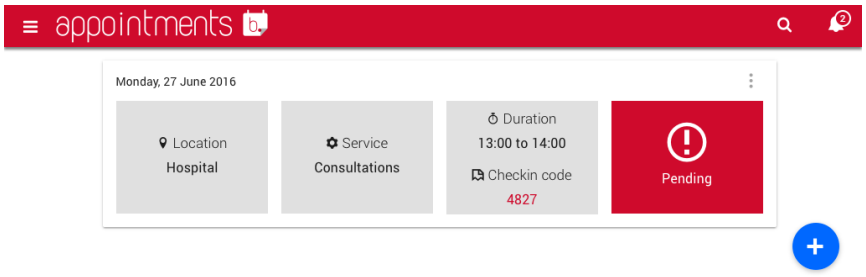


Figura 5.13: *Dashboard* do cliente com o agendamento criado

Apesar da criação do agendamento ser efetivada nas duas formas, o mesmo não pode ser consultado na página pessoal caso tenha sido criado num dispensador, uma vez que neste caso de exemplo não existiu nenhum formulário que exigisse a identificação por parte do cliente.

Após a confirmação por parte do *Staff* sobre o agendamento criado, o cliente receberá uma notificação *push* na sua página pessoal e o estado será automaticamente atualizado, tal como demonstra a Figura 5.14.

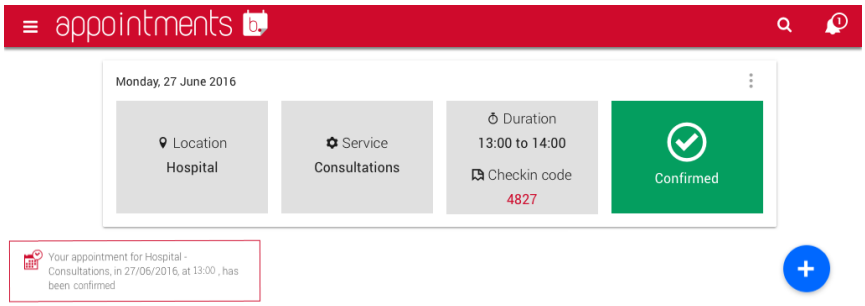


Figura 5.14: *Dashboard* do cliente com a notificação de agendamento confirmado

Interface gráfica - Páginas do Staff

Os elementos do *Staff* têm interfaces distintas em relação aos clientes. Estas são igualmente alimentadas pelos serviços definidos no componente servidor.

A Figura 5.15 mostra a *Dashboard* com informações básicas sobre o estado diário da localização. O utilizador pode através deste painel ter acesso a estatísticas e realizar operações básicas nos agendamentos, tais como cancelar, confirmar e fazer *check-in*.

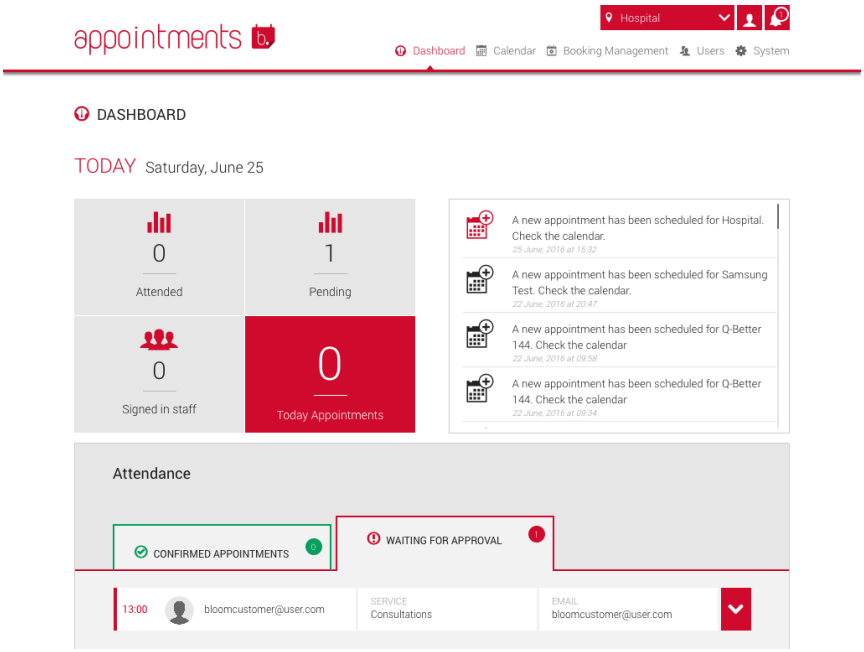


Figura 5.15: *Dashboard* do *Staff*

A Figura 5.16 mostra um exemplo de listagem, neste caso uma listagem de clientes registados no sistema. Todas as páginas de listagem e CRUD têm aspecto gráfico semelhante, por forma a tornar-se mais *user-friendly*, permitindo que o utilizador da aplicação se familiarize com as ações que tem de fazer. Enquadram-se então as páginas das localizações, balcões, clientes e categorias.

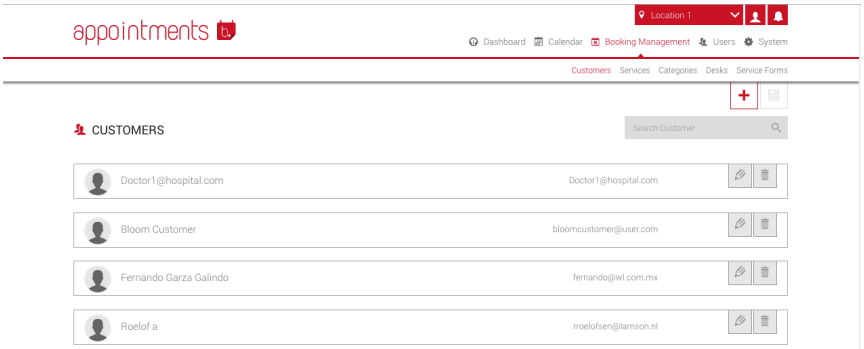


Figura 5.16: Listagem de clientes registados no sistema

A Figura 5.17 dá continuidade à figura anterior, mostrando agora um exemplo da edição de um serviço.

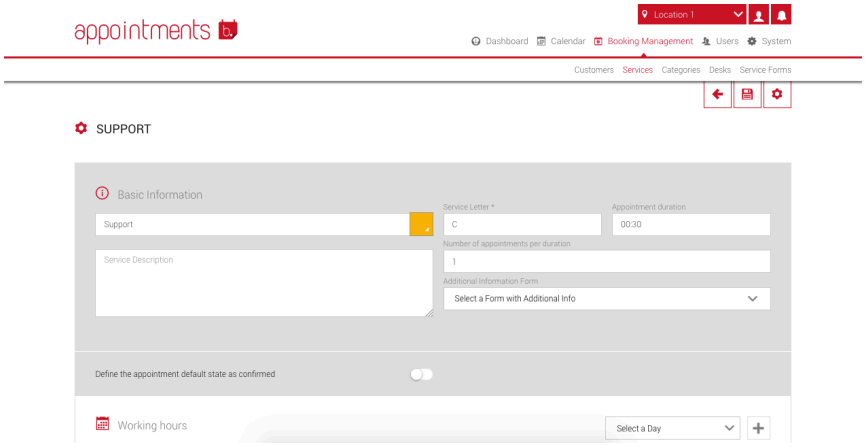


Figura 5.17: Página de edição do serviço *Support*

As Figuras 5.18 e 5.19 mostram o calendário de uma localização. Neste calendário, o elemento do *Staff* tem uma visão sobre o panorama mensal, semanal e diário dos agendamentos. Além disso, pode também marcar um novo agendamento, ou então verificar detalhes de um agendamento específico, permitindo também fazer ações rápidas sob este, nomeadamente fazer check-in, confirmar ou apagar.

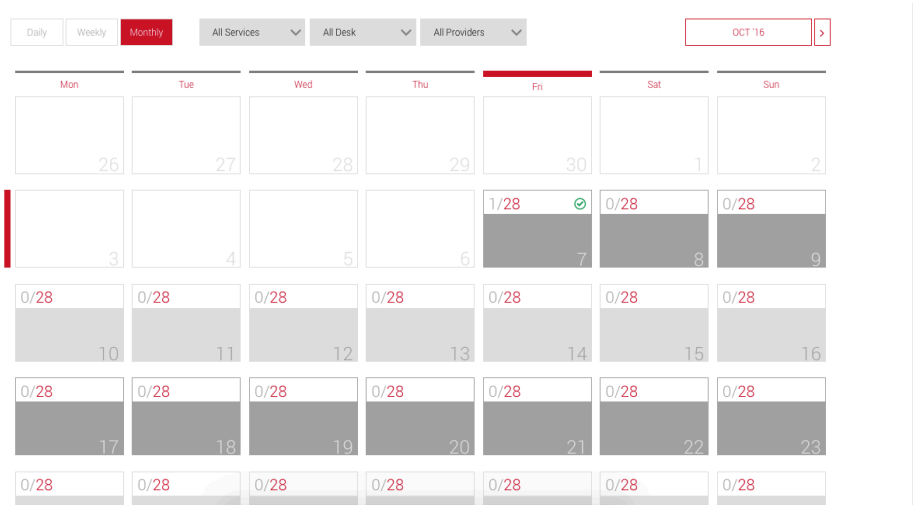


Figura 5.18: Vista do calendário de uma localização

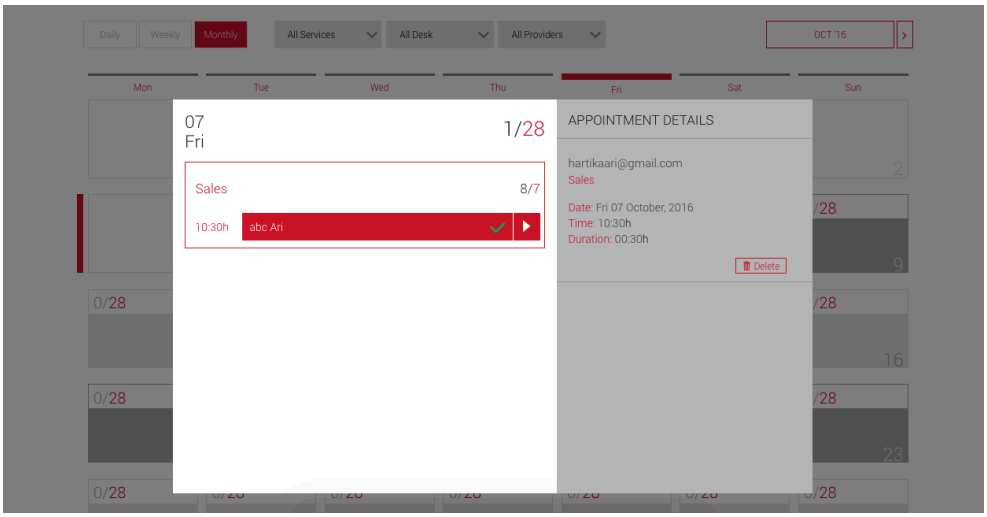


Figura 5.19: Vista detalhada de um agendamento num calendário

Neste capítulo foi demonstrado o processo de desenvolvimento do caso de estudo, descrevendo e dando exemplos das implementações quer do componente servidor - o *core* de toda a arquitetura orientada a serviços - sendo o responsável pela disponibilização dos serviços, quer da interface gráfica desenvolvida de forma a interagir com o sistema através dos serviços disponibilizados.

Capítulo 6

Conclusões

Hoje em dia pode afirmar-se que as arquiteturas orientadas a serviços são cada vez mais usadas na criação de novas aplicações. O facto de existirem infinitas ligações à internet em todo o mundo a um ritmo elevado com um tráfego completamente disperso, distribuído por vários dispositivos e pelas várias formas de consumir essa mesma informação, obrigou a que os métodos de desenvolvimento baseados em arquiteturas locais e isoladas se tornassem cada vez mais obsoletos, deixando de fazer sentido pensar numa aplicação cujo funcionamento fica restrito ao uso local e controlado, mas sim num ambiente de utilização com recurso de serviços através da rede. No entanto, surgem preocupações inerentes a este novo método tais como: a necessidade de prever e solucionar falhas de comunicação que podem originar perdas de informação ou respostas com demasiado tempo de atraso, ou a necessidade de implementação de políticas de segurança para evitar possíveis ataques ao sistema.

Uma vez encontrados problemas ao nível de manutenção de código antigo e ao nível de criação de novas soluções escaláveis, tornou-se necessária a criação de uma arquitetura de suporte capaz de resolver estas questões. Surge então esta dissertação, que permitiu desenvolver um estudo sobre arquiteturas orientadas a serviços por forma a perceber as suas características e quais os aspetos necessários para a implementação da mesma.

As arquiteturas orientadas a serviços surgem não só como uma abordagem para a conceção de aplicações que requerem comunicação entre si para troca de dados, mas que também permitem obter benefícios importantes a ter em conta no desenvolvimento de aplicações, tornando mais fácil integrar aplicações de plataformas de natureza diferente, com recurso a protocolos padrão tais como *web services*. Devido à baixa dependência dos serviços, à reutilização de código e suas funcionalidades, esta pode também ser utilizada para redução de custos de desenvolvimento e integração

de novas soluções. Como acrescento a este ponto, é ainda possível escalar os serviços dinamicamente.

Após o estudo sobre este tipo de arquiteturas e *web services*, pode concluir-se que apesar das semelhanças entre ambos, estes têm funções diferentes. SOA trata-se de uma arquitetura, ao contrário dos *web services* que cujas especificações permitem definir plataformas de comunicação implementadas numa arquitetura orientada a serviços. Apesar de se tratar da implementação mais comum quando se pretende construir serviços de negócios agnósticos a tecnologias, o uso de *web services* para implementação desta arquitetura pode ser alterada, por exemplo, por mecanismos de integração por um ESB (*Enterprise Service Bus*) cujo assunto não foi abordado nesta dissertação. Outro ponto de distinção está relacionado com os objetivos de ambos. Os objetivos da arquitetura orientada a serviços são estratégicos e orientados para o negócio. Já os objetivos de *web services* estão centrados na tecnologia, podendo ajudar a atingir objetivos estratégicos de negócios.

Outro dos objetivos desta dissertação passava pela comparação das diferentes versões de *web services* existentes. Em termos práticos, não foi possível testar outro tipo de implementação de *web services* que não os *web services RESTful*. A primeira grande conclusão a tirar passa pela facilidade quer de implementação, quer da própria interação com estes. Um dos grandes fatores que permite a fácil utilização deste tipo de serviços, independentemente da natureza das aplicações em questão, deve-se à larga utilização do seu protocolo base, o HTTP.

Ao implementar um *web service RESTful*, um dos cuidados a ter reside na definição da API. Esta definição requer atenção na sua criação, uma vez que o é o URI que indica quais são os recursos acessíveis. Uma má definição pode levar a uma insustentabilidade do próprio sistema, uma vez que, ou versões diferentes dos recursos ou um encaminhamento errado pode levar a que nas aplicações cliente sejam obtidos resultados errados. Outro fator que simplifica o processo de comunicação reduzindo a sua complexidade é a não obrigatoriedade recorrer a WSDL, uma vez que a comunicação é feita através do protocolo HTTP e com recurso à interface previamente conhecida. Contudo, este fator faz com que os dados previamente serializados em XML ou JSON tenham uma estrutura conhecida no lado da aplicação que consome o serviço.

Apesar da distinção entre *web services* e e SOA, grande parte das características de *web services RESTful* complementam as características da arquitetura. Todos os objetivos do REST favorecem direta e indiretamente a interoperabilidade entre vários serviços. Ambos são defensores da abstração de detalhes da implementação dos serviços aos consumidores para evitar formas negativas de dependência que podem inibir a independência do produto desenvolvido. Um maior alinhamento entre o negócio e a tecnologia é o fundamento principal da arquitetura orientada a serviços que pode ser apoiada pelos *web services RESTful* graças ao seu ênfase tecnológico. O aumento do retorno de investimento

torna-se também natural graças à possibilidade de reutilização implementada igualmente em REST. Embora o negócio não seja um objetivo para REST, cada objetivo deste pode contribuir diretamente para aumentar a capacidade de resposta das organizações [26].

A abordagem sobre a temática da sincronização foi também definida como objectivo para esta dissertação. Uma vez que se tratou de um requisito na implementação do *Bloom Appointments*, o estudo sobre várias soluções de sincronização permitiu ter ideias sobre os métodos a aplicar para interligar o BQM e o sistema de agendamentos. No caso prático descrito no capítulo anterior, demonstrou-se que o método escolhido passou pela sincronização com recurso a *timestamps*. Este método tornou-se eficaz, na medida em que a única desvantagem seria na primeira sincronização entre os sistemas, uma vez que seria necessário o envio de toda a informação, o que se conseguiu implementar sem intercorrências.

O desenvolvimento do *Bloom Appointments* foi o *case study* utilizado nesta dissertação para demonstrar a construção de uma aplicação cujo suporte de informação é proveniente de uma arquitetura orientada a serviços. Graças às exaustivas tarefas de análise, foi possível especificar com detalhe todas as funcionalidades da aplicação, permitindo assim obter a listagem de serviços a desenvolver para suportar a aplicação. Após este trabalho, foram então desenvolvidos os serviços do componente servidor, criando assim uma extensa *API RESTful* que cobrem as funcionalidades requeridas. Posteriormente, foi então desenvolvido a interface gráfica da aplicação que, seguindo a linha de produtos *Bloom*, foi construído com linguagens compatíveis com *standards web*. O AngularJS revelou-se boa escolha na implementação desta interface, uma vez que as suas características permitiram uma construção rápida de uma interface dinâmica que usasse todos os serviços disponibilizados pela arquitetura desenvolvida.

É assim possível concluir que a escolha deste tipo de arquitetura para a realização deste projeto revelou-se acertada, uma vez que consegue juntar todas as suas vantagens com os objetivos da própria Q-Better, aliar uma melhor experiência de uso com o desenvolvimento das soluções com as mais recentes tendências tecnológicas. O *Bloom Appointments* veio criar na Q-Better um novo paradigma de desenvolvimento, abrindo não só as portas a novos métodos de expansão da solução *Bloom* de um modo mais facilitado e mais aberto, mas também para uma futura transição de soluções previamente concebidas, tentando adaptar o *legacy software* já existente.

6.1 Trabalho futuro

Com a conclusão deste projeto, é possível determinar trabalho futuro por forma a melhorar e acrescentar novas características à arquitetura desenvolvida.

O primeiro passo será a implementação de mecanismos de segurança na arquitetura, diminuindo assim o risco de interseção dos pedidos por parte de terceiros, por forma a evitar a adulteração de todo o sistema.

Uma vez implementado o ponto anterior, o trabalho seguinte será uma conversão das restantes aplicações *Bloom* para que estas passem a ser suportadas pela arquitetura criada. Apesar de se tratar de um trabalho de complexidade elevada, as vantagens do uso da arquitetura farão com que o *Bloom* se torne ainda mais versátil e com um crescimento mais facilitado.

Relativamente ao *Bloom Appointments*, uma das possibilidades passa por dotar a arquitetura de mecanismos que permitam a sincronização com outros sistemas de gestão de filas, aumentando a compatibilidade com outros sistemas distintos.

Outro ponto a avaliar seria a interoperabilidade com outros sistemas de agendamentos já existentes em várias organizações. Dando como exemplo a área da saúde, dotar a arquitetura da capacidade de comunicação com sistemas de informação através do conjunto de normas HL7, daria a possibilidade de usar apenas a arquitetura como ”motor de agendamentos”de toda a organização.

Bibliografia

[1] A. Arsanjani, "Service-oriented modeling and architecture," *IBM developer works*, 2004. ISBN 10: 0-7695-2670-5, DOI: 10.1109/SCC.2006.93, Visitado em: 2016-05-10.

[2] S. Delamore, "Essence of mvc." http://www.essenceandartifact.com/2012/12/the-essence-of-mvc.html. Visitado em: 2016-01-20.

[3] H. M. e. a. Gudgin, M., "W3c recommendation - soap version 1.2." https://www.w3.org/TR/soap12/. Visitado em: 2016-01-07.

[4] "W3schools.com - xml wsdl." http://www.w3schools.com/xml/xml_wsdl.asp. Visitado em: 2016-01-07.

[5] C. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz, *OASIS Standard - Reference model for service oriented architecture*. 2006.

[6] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN 10: 0-131-46575-9.

[7] T. Erl, *SOA Design Patterns*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2009. ISBN 10: 0-136-13516-1.

[8] "W3c - web services architecture." https://www.w3.org/TR/ws-arch/. Visitado em: 2016-23-04.

[9] H. Adams, D. Gisolfi, J. Snell, and R. Varadan, "Best practices for web services: Part 1, back to the basics," *Retrieved February*, 2002.

[10] J. C. Broberg, "Glossary for the oasis web service interactive applications (wsia/wsrp)," *Retrieved February*, vol. 15, 2002.

[11] D. Booth, H. Haas, and A. Brown, "Web services glossary - w3c working group note 11 february 2004," tech. rep., World Wide Web Consortium (W3C), 2004.

[12] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. ISBN 10: 0-599-87118-0.

[13] O. F. F. Filho and M. A. G. V. Ferreira, "Semantic web services: A restful approach," in *Proceedings of the IADIS International Conference WWWInternet 2009*, pp. 169–180, IADIS, 2009.

[14] R. Ramanathan and T. Korte, "Software service architecture to access weather data using restful web services," in *Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on*, pp. 1–8, 2014. DOI: 10.1109/ICCCNT.2014.6963122.

[15] J. Laanstra, *Offline Data and Synchronization for a Mobile Backend as a Service system*. PhD thesis, Faculty EEMCS, Deft University of Techonology, 2013.

[16] D. Starobinski, A. Trachtenberg, and S. Agarwal, "Efficient pda synchronization," *IEEE Transactions on Mobile Computing*, vol. 2, no. 1, pp. 40–51, 2003. ISSN: 1536-1233.

[17] T. M. H. Reenskaug, "The original mvc reports," 1979.

[18] S. Burbeck, "Application programming in smalltalk-80: How to use model-view-controller (mvc)," *Smalltalk-80 v2*, vol. 5, 1992.

[19] M. Bean, *Laravel 5 Essentials*. Packt Publishing Ltd., 2015. ISBN 10: 1-785-28301-4.

[20] P. M. N. Jain and D. Mehta, "Angularjs: A modern mvc framework in javascript," vol. 5, no. 12, pp. 17–23, 2014. ISSN 2229-371X.

[21] A. Freeman, *Pro AngularJS*. Apress, 1st ed., 2014. ISBN 10: 1-430-26448-9.

[22] K. Williamson, *Learning AngularJS: A Guide to AngularJS Development*. O'Reilly Media, Inc., 1st ed., 2015. ISBN 10: 1-491-91675-3.

[23] L. Ruebbelke and B. Ford, *AngularJS in Action*. Manning Publications, 2015. ISBN 10: 1-617-29133-1.

[24] "Angularjs developer guide." http://docs.angularjs.org/guide. Visitado em: 2016-02-15.

[25] A. Lerner, *ng-book - The Complete Book on AngularJS*. Fullstack io, 2013. ISBN 10: 0-991-34460-X.

[26] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, "Soa with rest - principles, patterns and constraints for building enterprise solutions with rest," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 3, pp. 32–33, 2013. ISBN 10: 9-332-52384-3.

Apêndices

Apêndice A

Use cases

Os *use cases* apresentados na Figura A.1 e na Figura A.2 surgem na sequência do diagrama de *use cases* da Figura 4.1 e mostram de forma mais específica para estes dois tipos de utilizadores as possíveis interações que podem ter com o *Bloom Appointments*.

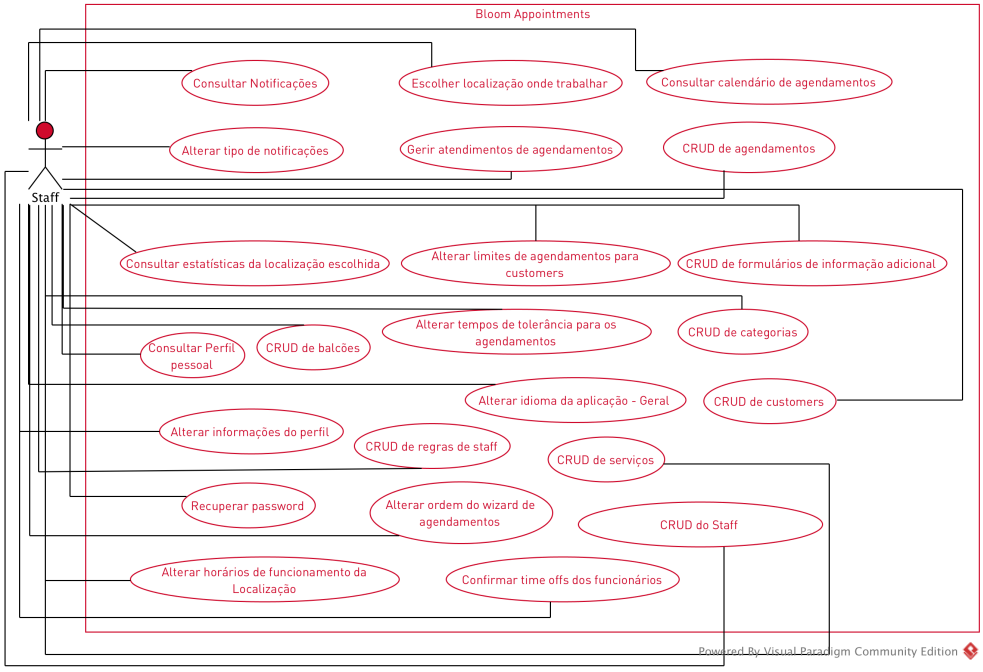


Figura A.1: Use Case do *Staff*

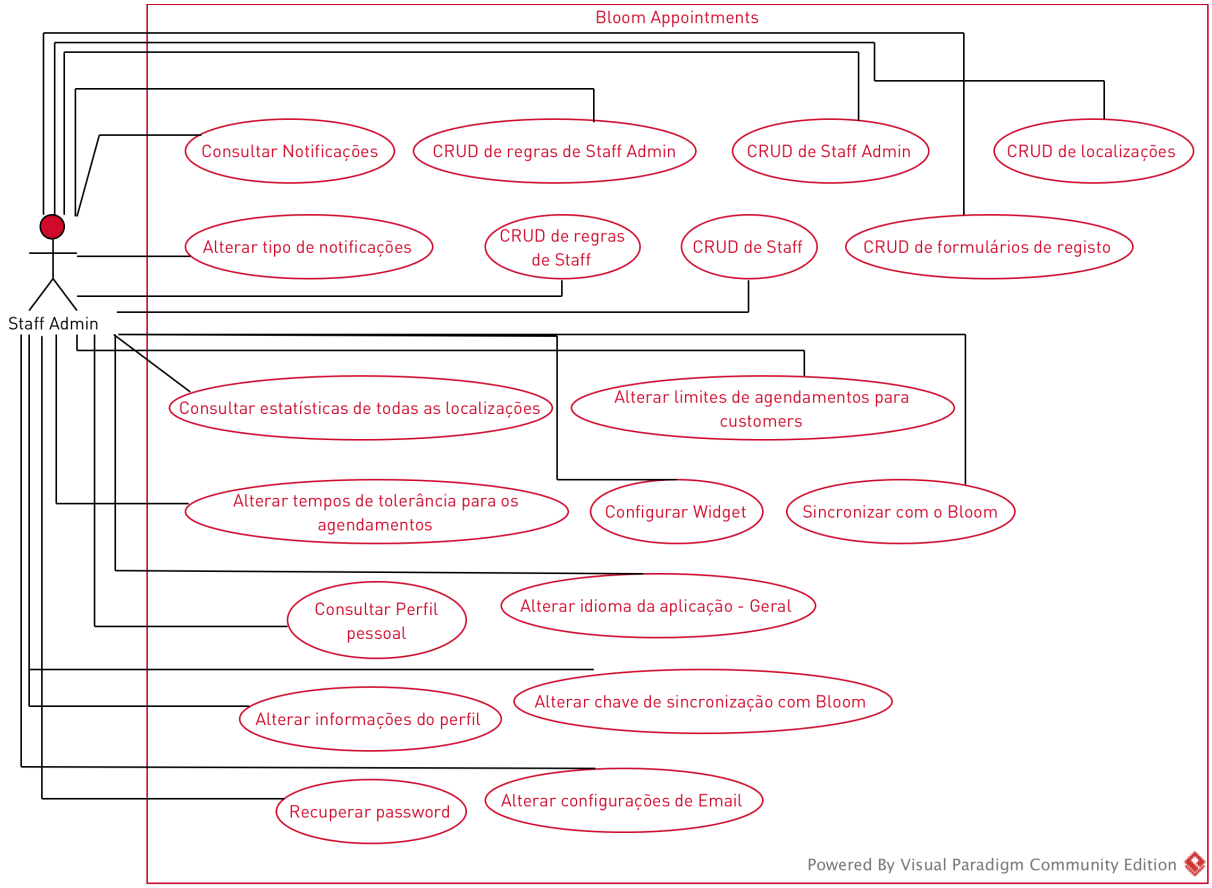


Figura A.2: Use Case do *Staff Admin*

Apêndice B

Diagrama da Base de Dados

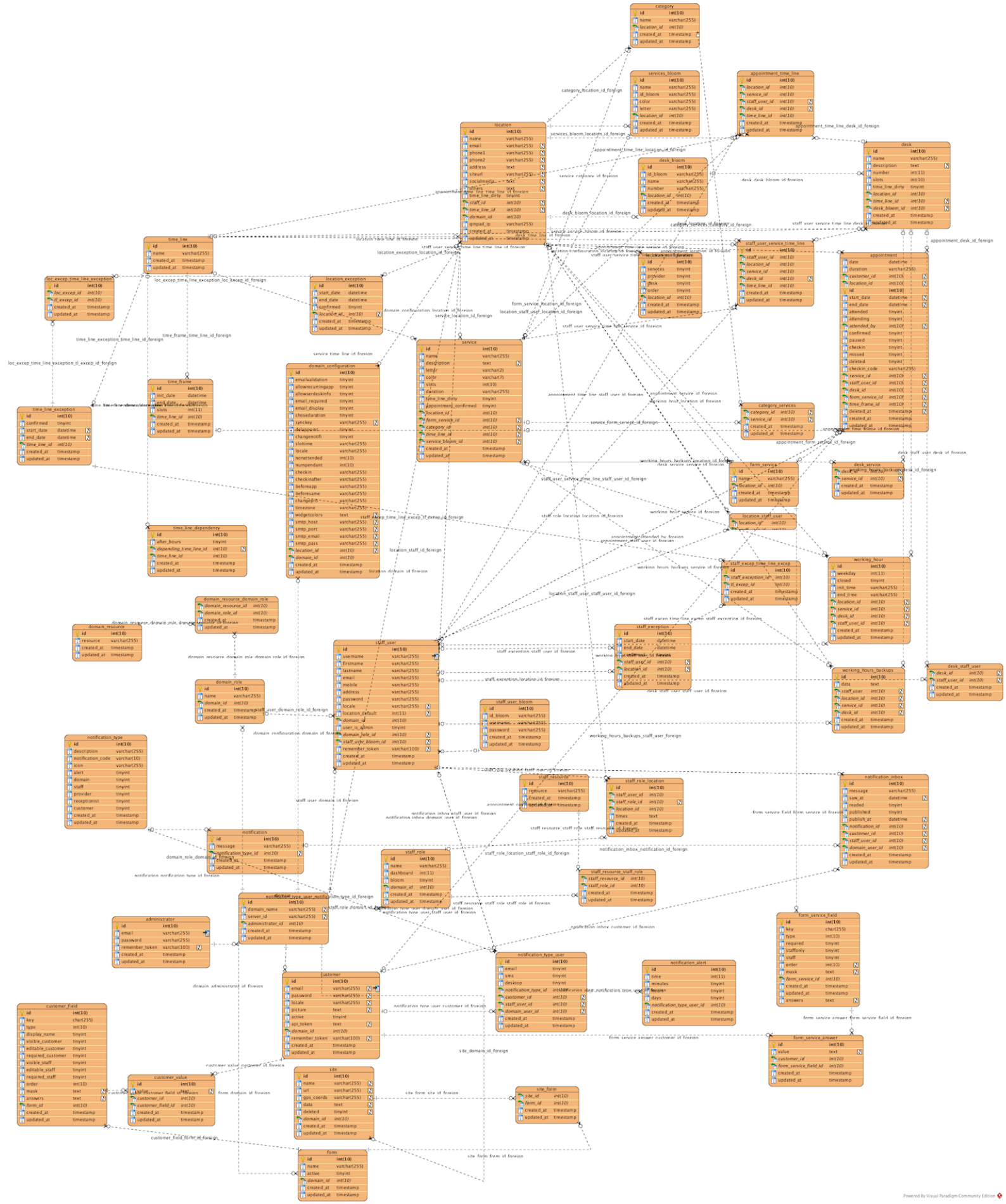


Figura B.1: Diagrama da Base de Dados do *Bloom Appointments*

Apêndice C

API RESTful

Registo / Recuperação de password	
GET	/customer/getregistrationgfields
POST	/customer/registration
POST	/customer/restorepassword
Ações gerais	
GET	/customer/getprofileinfo
GET	/customer/getnotificationsuser
DELETE	/customer/deleteallnotification
GET	/customer/getappointments
POST	/customer/updateprofile
DELETE	/customer/deleteappointment

Figura C.1: *API RESTful* de ações relacionadas com os *Clientes*

Criação de um agendamento	
GET	/schedule/getlocations
GET	/schedule/getservices
GET	/schedule/getdesks
GET	/schedule/getproviders
GET	/schedule/gethours
GET	/schedule/init_calendar
GET	/schedule/addinformation
GET	/schedule/getUserInfo
POST	/schedule/confirm_appointment
POST	/staff/chekinappointment
GET	/appointment_activation
GET	/schedule/getmenuconfig
GET	/schedule/blockdays

Figura C.2: *API RESTful* de ações relacionadas com a criação de um agendamento (*Schedule*)

Dashboard	
GET	/staff/getdashproviderinformation
GET	/staff/hasqmpad
GET	/staff/getdashmanagerinformation
GET	/staff/getservicegraphdata
GET	/staff/getstaffusergraphdata
POST	/staff/attendedappointment
Localizações	
GET	/staff/getlocations
GET	/staff/getlocationsbyuser
GET	/staff/getlocationsoverview
GET	/staff/addinformation
Serviços	
GET	/staff/getallservices
POST	/staff/saveservices
DELETE	/staff/removeservices
Categorias	
GET	/staff/getallcategories
POST	/staff/deletecategory
POST	/staff/savecategories

Perfil	
GET	/staff/getprofile
POST	/staff/saveprofile
POST	/staff/confirmexceptionstaff
Regras do Staff	
GET	/staff/getallroles
POST	/staff/savestaffroles
POST	/staff/deletestaffrole
Clientes	
GET	/staff/getallcustomers
POST	/staff/savecustomers
DELETE	/staff/deletecustomers
Formulários de serviço	
GET	/staff/getformservices
POST	/staff/saveformservice
DELETE	/staff/deleteformservice
DELETE	/staff/deleteformservicefield

Calendário	
POST	/staff/cancelappointment
POST	/staff/callappointment
POST	/staff/pauseappointment
Definições da Localização	
GET	/staff/getsettings
POST	/staff/savesettings
DELETE	/staff/deletelocationexception
POST	/staff/confirmlocationexception
Notificações	
GET	/staff/getnotifications
GET	/staff/getnotificationstaff
POST	/staff/savenotifications
DELETE	/staff/deletenotificationstaff
DELETE	/staff/deleteallnotifications
POST	/staff/seennotificationstaff

Calendário	
GET	/staff/getmenuconfig
GET	/staff/getcalendar
GET	/staff/getfilters
GET	/staff/gethoursavailablestaff
POST	/staff/confirmappointment
DELETE	/staff/deleteappointment
POST	/staff/createcustomer
POST	/staff/makeappointment
POST	/staff/callappointment
Balcões	
GET	/staff/getalldesks
POST	/staff/savealldesk
DELETE	/staff/removedesks
Staff	
GET	/staff/getallstaffusers
POST	/staff/restorepassword
POST	/staff/savestaff
DELETE	/staff/removestaff
DELETE	/staff/removeexception

Figura C.3: *API RESTful* de ações relacionadas com o *Staff*

Dashboard	
GET	/staff_admin/getdashinformation
GET	/staff_admin/getservicegraphdata
GET	/staff_admin/getdeskgraphdata
DELETE	/staff_admin/deleteappointment
POST	/staff_admin/confirmappointment
GET	/staff_admin/getstaffusergraphdata
Definições gerais	
GET	/staff_admin/getrolesresources
POST	/staff_admin/savedomainroles
GET	/staff_admin/verifystmp
DELETE	/staff_admin/deletedomainrole
GET	/staff_admin/domainsettings
POST	/staff_admin/savedomainsettings
Formulário de Registo	
GET	/form/getcustomerfields
POST	/form/savecustomerfields
DELETE	/form/removecustomerfield

Perfil	
GET	/staff_admin/getprofile
POST	/staff_admin/saveprofile
Notificações	
GET	/staff_admin/getnotifications
POST	/staff_admin/savenotifications
DELETE	/staff_admin/deletenotificationstaff
DELETE	/staff_admin/deleteallnotifications
POST	/staff_admin/seennotificationstaff
Localizações	
GET	/staff_admin/getalllocations
POST	/staff_admin/savelocations
DELETE	/staff_admin/removelocation
POST	/staff_admin/createstaff

Widget	
GET	/staff_admin/getwidgetconfigurations
GET	/staff_admin/initwidgetpreviewconfigurations
POST	/staff_admin/savewidgetconfigurations
POST	/staff_admin/loadpreviewconfigurations
Gestão de Staff	
GET	/staff_admin/getallstaffusers
POST	/staff_admin/confirmexception
DELETE	/staff_admin/removeexception
DELETE	/staff_admin/removestaff
POST	/staff_admin/save_staff_users

Figura C.4: *API RESTful* de ações relacionadas com o *Staff Admin*