UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Solução Digital Web Platform

Liliana Sofia da Mota Ramos

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Versão Pública

Trabalho de Projeto orientado por:
Prof. Doutor André Souto

2019

Agradecimentos

Em primeiro lugar quero agradecer à minha família, especialmente à minha Mãe, ao meu Pai e à minha Avó. Foram vocês que sempre me apoiaram nas minhas decisões e que me deram todos os valores para que eu chegasse até aqui. Sem vocês isto não seria possível.

Em segundo lugar quero agradecer a ti, Gonçalo, por me teres acompanhado ao longo de quase todo o meu percurso académico, por me ouvires sempre que preciso, por estares sempre presente e por seres um apoio incondicional.

Quero agradecer à Marta e ao Bruno por me terem dado a orientação necessária e fundamental para que este projeto acontecesse e chegasse a bom porto. Com isto, agradeço também ao Nuno Ferreira por toda a ajuda durante o estágio, especialmente na escrita deste relatório.

Agradeço ao meu orientador, Professor Doutor André Souto, por ter estado sempre disponível para ajudar, e pelo conhecimento transmitido ao longo deste estágio.

Agradeço também às minhas amigas, de há 9 anos, pela amizade que nos une, pela nossa união e por todos os momentos que passámos juntas. Delas destaco a Marta, a minha melhor amiga de uma vida, a quem agradeço muito.

Também tenho de agradecer aos meus colegas de trabalho, nomeadamente ao Artur, ao Beirão, ao José, ao Tomás, ao Paulo e à Joana, por me terem acolhido na empresa, por me terem também ajudado e terem tornado esta experiência ainda mais enriquecedora.

Às restantes pessoas que fizeram parte deste percurso direta ou indiretamente, o meu obrigada.

Aos meus.

Resumo

Este relatório está associado ao projeto realizado na empresa Accenture Portugal, em contexto de uma empresa de telecomunicações, no período de tempo de Outubro de 2018 a Julho de 2019, para a conclusão do Mestrado em Engenharia Informática, com especialização em Engenharia de Software, na Faculdade de Ciências da Universidade de Lisboa.

Este projeto consiste no desenvolvimento de uma aplicação enquanto prova de conceito para apresentação de uma fatura com suporte para a lógica de negócio prestada por um assistente virtual cujo objetivo é ajudar os clientes da empresa a esclarecer possíveis dúvidas relativamente à mesma. Esta prova de conceito representa uma solução *self-service* (ou *self-care*) que pode vir a ser disponibilizada no *website* do cliente, numa *box-TV*, via IVR (*Interactive Voice Response*) ou via App Móvel, com o intuito de atuar numa primeira instância como um agente de *call center* real. Este tipo de soluções permite reduzir o fluxo de chamadas para os tradicionais centros de atendimento o que, consequentemente, reduz os tempos de espera dos clientes, e os custos de operação da empresa.

A aplicação foi implementada com tecnologias *open-source* e dividida em duas camadas:

- uma camada de apresentação responsável por fornecer ao utilizador uma interface e
- uma outra camada que suporta a primeira sendo responsável pela comunicação com o assistente virtual e pelo processamento da informação a apresentar.

Para além do desenvolvimento da aplicação, também foi feita a sua integração com o assistente virtual, sendo que em alguns pontos do plano do projeto existiu colaboração. Este assistente virtual, desenvolvido por um colega, teve como objetivo receber questões feitas pelos clientes e, ao utilizar mecanismos de interpretação de linguagem natural e de *machine learning*, responder-lhes de forma adequada recorrendo por vezes a API's externas.

Palavras-chave: *Chatbot*, *Spring*, *Angular*, assistente virtual

Abstract

This report is associated with the project developed at Accenture Portugal, in the context of a telecommunications company, from October 2018 to July 2019, for the completion of the master's degree in Informatics Engineering, specializing in Software Engineering, at Faculdade de Ciências da Universidade de Lisboa.

This project aims to develop an application as a proof of concept for presenting an invoice with support for the business logic of a virtual assistant whose goal is to help customers clarify possible doubts. This proof of concept is a self-service (or self-care) solution that may be made available on the client's website, in a box-TV, through IVR (Interactive Voice Response) or through an App Mobile, to act in the first instance as a real call center agent. This type of solutions helps to reduce the flow of calls to the call centers and therefore, reduces customer wait time, and operating costs of the company.

The application was implemented with open-source technologies and divided in two layers:

- a presentation layer responsible for providing the user with an interface and
- another layer that supports the first one by being responsible for communicating with the virtual assistant and for processing the information to be presented.

In addition to the development of the application, its integration with the virtual assistant was also part of the project, and at some point in the project plan there was a collaboration. This virtual assistant, developed by my colleague, aims to receive questions made by customers and, by using mechanisms of natural language and machine learning, respond appropriately with possible use of external API's.

Keywords: *Chatbot*, *Spring, Angular, virtual assistant*

Conteúdo

Lista de Figuras

Acrónimos e Abreviaturas

IVR	Interactive Voice Response
API	Application Programming Interface
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
JSF	JavaServer Faces
GWT	Google Web Toolkit
MVC	Model-View-Controller
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
XML	Extensible Markup Language
URI	Uniform Resource Identifier
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
CRUD	Create, Read, Update and Delete
AOP	Aspect-Oriented Programming
IoC	Inversion of Control
JNDI	Java Naming and Directory Interface
Java EE	Java Enterprise Edition
EJB	Enterprise JavaBeans
JMX	Java Management Extensions
JDBC	Java Database Connectivity
ORM	Object-Relational Mapping
OXM	Object/XML Mapping
JMS	Java Message Service
SQL	Structured Query Language
JPA	Java Persistence API
JDO	Java Data Objects
JAXB	Java Architecture for XML Binding

JTA	Java Transaction API
JSP	JavaServer Pages
OOP	Object-Oriented Programming
URL	Uniform Resource Locator
RxJS	Reactive Extensions for JavaScript
MIME	Multipurpose Internet Mail Extensions

Capítulo 1

Introdução

Este projeto foi elaborado na empresa Accenture em contexto de uma empresa de telecomunicações. Com a vasta experiência da Accenture nas áreas de estratégia, consultoria, digital, tecnologia e operações, existe uma grande preocupação em oferecer resultados diferenciadores no que toca aos novos desafios do mundo digital [1]. Como tal, procedeu-se ao desenvolvimento de uma plataforma para apresentação de uma fatura de forma interativa e que dá suporte aos processos de negócio de um assistente virtual.

Neste capítulo é descrita a motivação do desenvolvimento deste projeto, os seus objetivos, o estado de arte e como este documento está organizado.

1.1 Motivação

A concorrência feroz que se faz sentir entre os vários operadores de telecomunicações aguça a procura de ofertas cada vez mais complexas em termos de *pricing*. O aumento da complexidade das ofertas leva inevitavelmente ao aumento da dificuldade de interpretação de uma fatura por parte do cliente. Esta dificuldade materializa-se sobretudo em questões como *"Qual é o verdadeiro preço da minha mensalidade?"* ou *"Qual é a razão para este mês estar a pagar mais do que o normal?"*. A necessidade de respostas às questões como as mencionadas anteriormente, tem como consequência a sobrelotação de chamadas para os tradicionais *call centers*. De modo a evitar a sobrecarga de afluência ao serviços de *call center*, a Accenture pensou numa solução que passa pelo desenvolvimento de uma aplicação para apresentação de uma fatura de forma mais intuitiva e interativa, com possibilidade de questionar um assistente virtual para esclarecer, de forma automática, as dúvidas mais comuns dos clientes relativamente às suas faturas e que tenham respostas padronizáveis. Esta aplicação foi

pensada para ser o mais apelativa possível, de forma a que a interação com o *chatbot* seja feita com um contexto visual associado, o que não acontece noutros *chatbots* do mercado cuja interação é feita com base num modelo de linha de comando.

1.2 Estado de Arte

1.2.1 *Assistente Virtual*

O texto desta secção é baseado na referência [2] e apresenta o conceito de assistente virtual e as suas principais caraterísticas.

Com a evolução da *web* e os avanços no que toca ao processamento da linguagem natural e à inteligência artificial, espera-se que os assistentes virtuais dominem o mercado aliviando a carga dos *call centers*, por exemplo. No entanto, antes do desenvolvimento de soluções de negócio para um problema concreto é necessário perceber o que é um assistente virtual, em que categoria se enquadram e perceber as características que os diferenciam e que podem ajudar no desenho da solução dos vários problemas. Só assim é possível perceber que potencialidades estas abordagens podem trazer para a solução e escolher as plataformas e ferramentas para o fazer (neste caso no que diz respeito à interface do utilizador). O objetivo de um assistente virtual é automatizar um serviço através de uma interface de conversação que o utilizador pode usar 24h por dia 7 dias por semana.

Este tema, relacionado com os assistentes virtuais, tem vindo a ser amplamente explorado uma vez que já existem tipos e características definidas sobre os objetivos e serviços a serem fornecidos por este tipo de assistentes. No que toca ao conhecimento de um assistente virtual, este pode ser caracterizado por ter um domínio aberto ou fechado. Em domínio aberto, os assistentes virtuais respondem apropriadamente sobre tópicos gerais. No domínio fechado, o conhecimento diz respeito a um tema específico, em que o assistente está preparado para as questões comuns desse tema, e pode falhar na resposta a outras questões que não estejam relacionadas com o tema para o qual foi concebido.

Relativamente ao serviço fornecido por um assistente virtual, existem três hipóteses possíveis:

- Assistente interpessoal que está relacionado com os assistentes que podem ter uma personalidade (e.g. ser amigáveis), porém não são obrigados a isso sendo que a sua função passa apenas por transmitir informação.

- Os assistentes intrapessoais acompanham o utilizador e compreendem-no como um humano.
- Assistente inter-agente cujo serviço passa pela comunicação entre dois ou mais assistentes virtuais com o intuito de integrar diferentes serviços num mesmo ecossistema de conversação.

Os assistentes virtuais podem também ser diferenciados quanto aos objetivos para os quais são desenhados. As três principais funções de um assistente virtual são transmissão de informação, manutenção de uma conversa e execução de uma tarefa. A primeira função corresponde a um assistente cujo objetivo é o de apenas fornecer informação que já se conhece previamente e que, por exemplo, se encontra armazenada numa base de dados. A função de manutenção de uma conversa passa por estabelecer um diálogo com o utilizador como um ser humano o faria, utilizando técnicas como a de perguntas cruzadas, a de evasão ou de respeito. A última função, a execução de uma tarefa, é autoexplicativo sendo que normalmente envolve ações já pré-determinadas que seguem fluxos de eventos fixos.

No que diz respeito ao processamento do *input* e ao método de geração da resposta um assistente virtual pode ser totalmente inteligente ou fazê-lo com base em regras. Se o sistema for totalmente inteligente, gera respostas e interpreta a questão do utilizador utilizando perceção da linguagem natural. Estes sistemas normalmente são usados quando o domínio é restrito e existem dados de treino. Um sistema com base em regras usa *pattern matching* e são rígidos nas respostas. Para um dado contexto, quando o número de resultados possíveis é fixo e os cenários são contáveis, este é o tipo de sistema que deve ser adotado como solução a ser implementada. No entanto, um sistema não tem obrigatoriamente de se enquadrar num destes dois tipos de processamento. Em particular, pode ser híbrido e, nesse caso, o sistema é um misto de *machine learning* com regras.

Tendo em conta todas estas características e a possibilidade de um assistente virtual não pertencer exclusivamente a uma categoria, podemos definir qual a utilidade e uso do assistente virtual no contexto deste projeto em concreto. O assistente virtual com que este projeto se integra tem um domínio fechado uma vez que apenas tratará assuntos relacionados com a fatura do cliente. O serviço prestado será interpessoal devido ao assistente ter sido pensado apenas para esclarecer dúvidas do cliente e não para manter uma conversa ocasional.

No que toca à função, este assistente é responsável por fornecer informação solicitada sobre a fatura do cliente, nomeadamente sobre consumos de serviços, e pode também ser responsável por executar tarefas como fornecer informação necessária para processar um pagamento ou agendar um contacto com um operador humano. Por último, relativamente ao processamento do *input* e ao método de geração de respostas, este assistente irá ser híbrido, ou seja, fará a perceção e o processamento da linguagem natural, e também poderá funcionar com base em regras definidas.

1.2.2 *Desenvolvimento da Interface de Utilizador*

Ao desenvolver uma interface de utilizador, especialmente para uma aplicação *web*, e no que toca a escolher uma *framework* para desenvolvimento com *JavaScript*, CSS e HTML, é necessário identificar todas as opções possíveis e qual melhor se adequa ao problema e ao contexto de desenvolvimento. Hoje em dia, para desenvolvimento *web* existe um conjunto vasto de ferramentas disponíveis que facilitam a organização do código e fornecem uma abstração relativamente a alguns aspetos de configuração. *Angular*, React.js, Node.js e ASP.NET são algumas das *frameworks* mais conhecidas para desenvolvimento de aplicações *web*. Para este projeto específico, a *framework* escolhida foi a *Angular* 7 pelos seguintes motivos: foi desenvolvida com base em componentes, que tem como benefícios a reutilização, legibilidade e manutenção do código; utiliza *TypeScript*, um superconjunto do *JavaScript* com mais escalabilidade; e é adaptável a múltiplas plataformas. Esta *framework* está explicada com mais detalhe no Secção 2.4.

1.2.3 *Formato Input e Output do cliente*

Neste projeto, a forma como é passado o *input* do utilizador e o *output* do assistente é um ponto fulcral da aplicação. Isto deve-se ao facto de um dos objetivos deste projeto ser recolher o *input* por voz (convertendo-o de seguida para texto) e responder ao utilizador igualmente por voz (sintetizando, neste caso, a resposta do agente dada em texto para voz). Relativamente a este mecanismo de sintetização de voz, já existem várias aplicações que implementam esta funcionalidade e não foi alvo de desenvolvimento no decorrer do projecto. A *Siri*, a *Cortana* ou a *Alexa* são exemplos bem conhecidos da utilização desta funcionalidade. Atualmente já é possível encontrar várias *API*'s que disponibilizam esta síntese e reconhecimento de voz. No âmbito deste projeto decidiu-se usar as *API's* da Google Cloud: *Speech-to-Text* e *Text-to-Speech*. Esta decisão deveu-se

sobretudo ao facto da possibilidade de utilizar a *API* em Português de Portugal, tanto para reconhecer voz como para sintetizá-la.

1.2.4 *Modularidade da Aplicação*

Um dos objetivos deste projeto passa pela implementação como uma *framework* genérica para ser possível a reutilização dos módulos em projetos futuros. Por esta razão, é importante separar a camada de lógica da interface propriamente dita. Primeiramente, foi tomada a decisão de que esta componente de suporte à interface do cliente seria implementada na linguagem de programação *Java*, por existir bastante documentação sobre a mesma e existirem também várias *frameworks* de desenvolvimento *web* com esta linguagem. JSF (*JavaServer Faces*), *Struts*, *Hibernate*, GWT (*Google Web Toolkit*) e *Spring* são algumas das *frameworks* aplicáveis a este caso. A escolha recaiu na *framework Spring*, apesar do *Struts* ser um concorrente direto, pelas seguintes razões: qualidade da documentação existente; incluir o padrão MVC, injeção de dependências e programação orientada a aspetos, que são úteis na abstração e modularidade; ter embebido o servidor *Tomcat*; e a possibilidade de integração com outras *frameworks* (neste caso, *Angular*). Mais à frente, na Secção 2.3 é explicado mais detalhadamente a *framework Spring* escolhida para este projeto.

1.2.5 *Comunicação*

O último aspeto de implementação a ser definido foi a comunicação com o assistente virtual e a comunicação entre a interface de utilizador e a camada de suporte à mesma. As opções mais comuns e mais utilizadas são a arquitetura REST (*REpresentational State Transfer*) e SOAP (*Simple Object Access Protocol*). Enquanto que SOAP utiliza o formato XML para todas as mensagens, REST pode utilizar outros formatos mais flexíveis, leves e consequentemente mais rápidos. Os serviços RESTful, isto é, sem existência de estado e com abstração entre cliente e servidor, são uma mais valia neste projeto especialmente devido à sua modularidade. Para além disto, a implementação de uma aplicação com o estilo arquitetural REST é mais simples. Com base nas razões mencionadas anteriormente, a arquitetura REST foi a escolhida, tanto para a comunicação entre o cliente e a camada de suporte a este, como para a comunicação entre esta camada e o assistente virtual. A Secção 2.2 apresenta informação mais aprofundada sobre este estilo arquitetural.

1.3 Objetivos

Para este projeto, os objetivos definidos foram:

- Desenvolvimento de uma aplicação para apresentação de uma fatura ao cliente de forma mais intuitiva e interativa.
- Desenvolvimento de um mecanismo interativo para explicação das funcionalidades numa primeira instância de interação com a aplicação.
- Desenvolvimento de uma aplicação que dê suporte a lógica de negócio de um assistente virtual.
- Desenvolvimento de um mecanismo de interação por texto e voz entre o cliente e o assistente virtual.
- Implementação da aplicação como uma *framework* genérica para possível reutilização de módulos em projetos futuros.
- Integração da aplicação desenvolvida com a aplicação responsável pela lógica de negócio de um assistente virtual.

1.4 Organização do documento

Este documento está organizado por capítulos que explicam a forma como foi desenvolvido o projeto, desde o seu contexto e objetivos, até às ferramentas usadas e todo o processo de desenvolvimento.

Um primeiro capítulo de **Introdução** que apresenta o contexto do trabalho, qual a motivação para este projeto, os objetivos e a própria estrutura do relatório.

O segundo capítulo diz respeito ao **Trabalho Relacionado** e indica as tecnologias utilizadas no desenvolvimento deste projeto, desde *frameworks*, padrões de desenho até metodologias de desenvolvimento.

O capítulo três designado de **Planeamento e desenho** descreve o planeamento feito para a realização deste projeto e o desenho da solução (das interfaces e técnico) adotado e implementado.

No capítulo quatro de **Desenvolvimento** é explicado todo o trabalho realizado (implementação dos requisitos funcionais e não funcionais) até ao momento de entrega deste relatório.

O capítulo cinco descreve os **Testes** efetuados para garantir que as funcionalidades desenvolvidas se comportam da maneira esperada.

No capítulo seis da **Conclusão** apresenta-se uma breve discussão sobre o que foi desenvolvido ao longo deste projeto, os desafios encontrados bem como as soluções para os mesmos, e o trabalho futuro.

O último capítulo é dedicado à **Bibliografia** onde são incluídas todas as fontes de informação utilizadas para a realização deste relatório.

Para além do capítulo anterior ainda existem capítulos específicos de anexo escritos com o intuito de complementar a informação deste relatório.

Capítulo 2

Trabalho relacionado

Neste capítulo são apresentadas as ferramentas usadas ao longo deste projeto. Este capítulo é dividido em secções, sendo que cada uma descreve uma ferramenta usada no desenvolvimento e/ou planeamento deste projeto.

2.1 Padrão de Arquitetura MVC

Model View Controller (MVC) é um padrão de arquitetura de *software* usado para implementar interfaces de utilizador, ou seja, uma mais valia no que diz respeito à arquitetura de aplicações *web* devido à clara separação de camadas aplicacionais. Inicialmente, o MVC era normalmente implementado no lado do servidor com o cliente a fazer pedidos através de *forms* e a receber de volta as *views* para mostrar no *browser*. No entanto, hoje em dia muita da lógica encontra-se do lado do cliente e, por isso, passou a ser usada esta arquitetura na implementação de uma interface *web*.

Esta arquitetura é composta por três componentes [3]: *Model*, *View* e *Controller*. Esta separação torna as aplicações mais flexíveis e mais recetivas a iterações, promovendo a sua modularidade e a sua capacidade de reutilização.

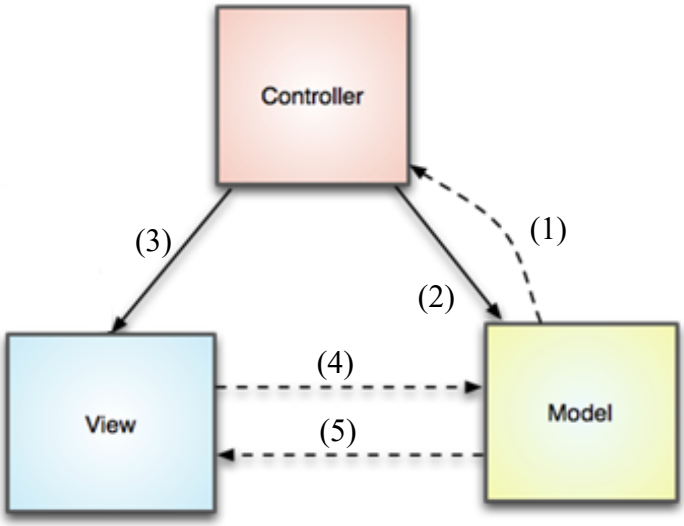

Figura 2.1 - Padrão de Arquitetura MVC. Retirada de [4]

A figura anterior, Figura 2.1, mostra a relação entre os três componentes. Estas relações são explicadas de seguida, com referência às legendas de cada uma das setas que

a figura contém. O *Model* é responsável por lidar com os dados que a aplicação deve conter, representando um objeto [5]. Se o estado destes dados for alterado, então o *Model* deve notificar a *View* (5), e possivelmente o C*ontroller* (1), para esta alteração ser mostrada [3]. A *View* define como é que os dados da aplicação devem ser mostrados ao utilizador, funcionando como uma interface. Esta transforma e dispõe os dados conforme necessário. O *Controller* contém toda a parte lógica usada para atualizar o *Model* (2) e/ou a *View* (3) em resposta ao *input* do utilizador [6]. É ele o responsável por separar a *View* e o *Model* [3].

As perspetivas seguintes assim como as vantagens apresentadas têm como base a referência [6]. Existem duas perspetivas diferentes sobre se o *Model* interage diretamente com a *View* ou não. No caso de não interagirem, todos os dados usados entre os dois são controlados pelo *Controller*. Caso interajam, então o *Controller* pode fazer com que a *View* peça dados ao *Model* (4). No fundo, a *View* passa a ter uma só funcionalidade se precisar de interagir com o *Model*.

Optar por esta arquitetura traz vantagens, principalmente no que toca à abstração entre o *Controller* e a *View*, ou entre o *Controller* e o *Model*. A abstração entre os dois primeiros permite que o *Controller* seja completamente alheio ao formato do *output* que a *View* usa. Isto significa que se for necessário alterar este formato basta criar uma nova *View* e o *Controller* mantém-se o mesmo. A lógica é a mesma no que toca à abstração entre o *Controller* e o *Model*. O formato em que os dados são guardados pode ser alterado sem necessidade de alterar o *Controller* ou a V*iew.*

A integração desta arquitetura com a *framework Spring* está explicada na Secção 2.3.3.

2.2 REST (*REpresentational State Transfer*)

REST é um estilo arquitetural destinado ao desenho de sistemas distribuídos facilitando a comunicação entre os mesmos [6]. Os sistemas compatíveis com esta arquitetura são usualmente chamados de sistemas RESTful. As suas características passam por não terem estados, por permitirem uma comunicação entre cliente e servidor e terem uma interface uniforme [8]. Para além disto, quatro princípios são aplicados a esta arquitetura:

- possibilidade de acesso aos recursos de forma percetível através de estruturas URI

- as representações transferem ficheiros JSON ou XML para representar objetos e atributos de dados
- mensagens usam métodos HTTP de forma explícita (apesar do REST não estar diretamente associado ao HTTP) e
- interações sem estado não armazenam nenhum contexto do cliente no servidor.

Nesta arquitetura, a implementação do cliente e do servidor pode ser feita de forma independente sem um ter conhecimento do outro [7]. Isto significa que a lógica do cliente pode ser alterada sem que com isso seja necessário alterar a lógica do servidor, desde que cada lado saiba o formato das mensagens a enviar. A utilização de uma interface com o estilo arquitetural REST torna possível que diferentes clientes atinjam os mesmos REST *endpoints*, executem as mesmas ações e recebam as mesmas respostas. Todas estas características dos sistemas RESTful fazem com que os sistemas sejam confiáveis, escaláveis, com desempenho rápido e com componentes que podem ser geridos, atualizados e reutilizados sem afetar o sistema como um todo.

Um pedido feito pelo cliente normalmente é constituído por: (i) uma operação HTTP que define o tipo de operação, (ii) um cabeçalho que permite passar informação sobre o pedido, (iii) um caminho para o recurso e (iv) uma mensagem opcional que contém dados [7].

Existem quatro tipos de operações que correspondem diretamente ao mapeamento CRUD (*create*, *retrieve*, *update*, *delete*) [8]. A operação POST faz um pedido ao recurso presente na URI para executar uma ação com a entidade fornecida, normalmente para criar ou atualizar uma entidade. GET é a operação responsável por retornar informação e deve ser segura e idempotente, isto é, independentemente do número de vezes em que o mesmo pedido é feito, os resultados são os mesmos. A terceira operação PUT faz parte do armazenamento de uma entidade num URI específico, sendo que pode criar ou atualizar uma entidade já existente. Esta operação é idempotente, ao contrário da primeira operação POST. No que toca à atualização de campos específicos de uma entidade num URI temos a operação PATCH. Esta operação pode não ser idempotente pois não consegue garantir que, entretanto, o recurso não tenha sido alterado. Por último, a operação DELETE faz um pedido para eliminar um recurso.

A integração destes pedidos HTTP com a *framework Angular* está explicado na Secção 2.4.7.

2.3 *Spring Framework*

Spring é uma ferramenta *open-source*, criada em 2002, que facilita o desenvolvimento de aplicações em Java. Atualmente a versão mais recente é a 5.1.2. [9]

Esta ferramenta está dividida em módulos o que permite escolher quais usar de acordo com as necessidades das aplicações. Na sua base esta ferramenta tem os módulos *Core Container*, *Data Access/Integration*, *Web*, *AOP* (*Aspect Oriented Programming*), *Instrumentation*, *Messaging* e *Test* [10]. Chama-se a atenção para os princípios fundamentais [9] que é preciso considerar para se poder usar corretamente a ferramenta:

- Existe a hipótese de escolha a todos os níveis, isto é, existem decisões de *design* que podem ser adiadas o mais possível sem que com isso se tenha de alterar o código.
- Há uma grande flexibilidade no que diz respeito à forma como as coisas devem ser feitas, ou seja, suporta um conjunto variado de necessidades de aplicações através de diferentes perspetivas.
- Existe uma grande retro compatibilidade. Este aspeto foi cuidadosamente pensado para serem necessárias poucas alterações entre versões.
- Disponibilização de *API*'s intuitivas e com versões com uma grande durabilidade.

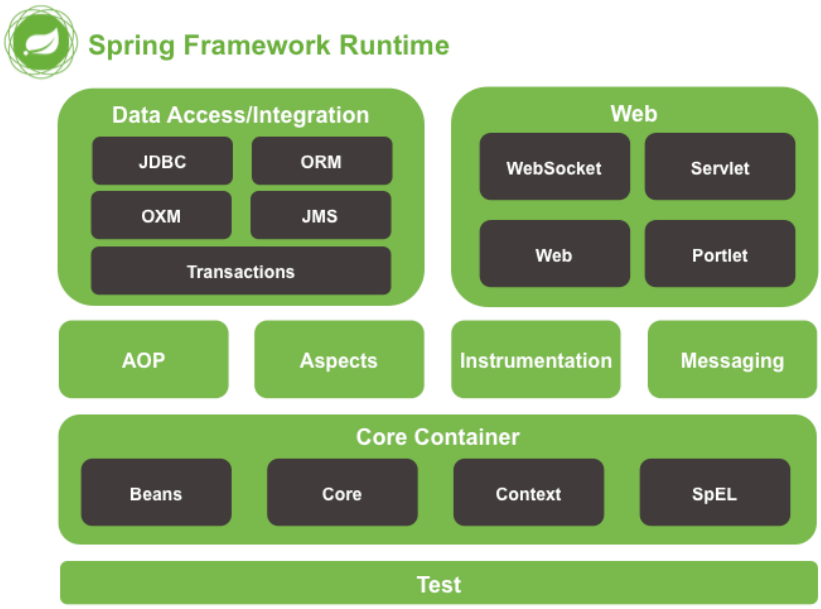

Figura 2.2 - Arquitetura da Framework Spring. Retirada de [10]

Os seus padrões de qualidade no que toca ao código são apontados como uma das grandes vantagens ao esta *framework*. Na Figura 2.2 é possível observar os módulos em que esta *framework* se divide: *Core Container*, *Data Access/Integration*, *Web*, *AOP (Aspects and Instrumentation)*, *Messaging* e *Test*. Cada um dos módulos é explicado mais detalhadamente nas subsecções seguintes.

2.3.1 *Core Container*

O *Core Container* é composto pelos módulos *core*, *beans*, *context* e *expression* (*Expression Language*) [10].

Os módulos *core* e *beans* fornecem as partes mais importantes da ferramenta, como injeção de dependências (IoC). IoC é um processo através do qual os objetos definem as suas dependências apenas com recurso aos argumentos do construtor, argumentos para um método *factory* ou propriedades que são definidas aquando da instanciação de um objeto [11]. Um componente fundamental nestes dois módulos é o *Bean Factory* que é uma implementação sofisticada do padrão de desenho *Factory*. Este padrão de desenho fornece uma interface para criação de objetos deixando que as subclasses definam qual a classe a instanciar. Com isto, é possível eliminar a necessidade de *singletons* e permite uma abstração da configuração e especificação das dependências da lógica do programa [10].

O módulo *context* é desenvolvido com base nos módulos mencionados anteriormente, isto é, fornece uma forma de aceder aos objetos semelhante ao registo *JNDI* (*Java Naming and Directory Interface*) [10]. Esta *interface* é uma *API* que permite a obtenção de dados ou objetos através do nome [12]. Este módulo herda as propriedades do módulo *beans* e adiciona suporte de propagação de eventos, carregamento de recursos e a criação transparente de contextos, como o *Servlet Container*. Para além disto, aqui também é adicionado o suporte para o Java EE (*Java Enterprise Edition*) com funcionalidades como o *EJB* (*Enterprise JavaBeans*), *JMX* (*Java Management Extensions*) e acesso remoto básico [10].

O último módulo *expression*, como o nome indica, fornece uma linguagem de expressão (*Expression Language*) para consultar e manipular um objeto gráfico em tempo de execução. Esta linguagem suporta a definição e obtenção de valores de propriedades, a atribuição de propriedades, a chamada de métodos, o acesso ao conteúdo de vetores, a operações lógicas e aritméticas, entre outras [10].

2.3.2 *Data Access/Integration*

Este módulo designado de *Data Access/Integration* é composto pelos módulos *JDBC* (*Java Database Connectivity*), *ORM* (*Object-relational mapping*), *OXM* (*Object/XML Mapping*), *JMS* (*Java Message Service*) e *Transaction* [13].

O módulo *JDBC* [14] fornece uma camada de abstração que elimina a necessidade de programar as definições da base de dados ou de analisar erros específicos, ou seja, todos os detalhes de baixo-nível. Com esta ferramenta, o programador apenas necessita de definir os parâmetros de conexão, especificar a *query SQL*, declarar os parâmetros e os valores dos mesmos e trabalhar com o resultado de cada iteração.

As API's *JPA* (*Java Persistence API*), *JDO* (*Java Data Objects*) e *Hibernate* são fornecidas pelo módulo *ORM.* Assim, é possível utilizar todas estas *frameworks* com todos os outros recursos por eles fornecidos [13].

O módulo *OXM*, este é responsável pela camada de abstração que suporta implementações como *JAXB* (*Java Architecture for XML Binding*), *Castor*, *XMLBeans*, JiBX e XStream [13].

O módulo *Java Message Service* (*JMS*) contém as funcionalidades de produção e consumo de mensagens [13]. A classe *JmsTemplate* é usada para a produção e receção de mensagens síncrona. A interface *MessageCreator* cria uma mensagem tendo em conta a sessão fornecida pelo código da classe anterior. Este módulo é o que fornece integração com o módulo *Messaging* [15].

Por fim, no módulo *Transaction* (tx) está a principal razão para se usar a *framework Spring* [16]. Esta fornece a abstração necessária para gestão de transações com os seguintes benefícios:

- modelo de programação consistente em diferentes *API*'s de transações como *JTA* (*Java Transaction API*), *JDBC*, *Hibernate*, *JPA* e *JDO*,
- suporta gestão de transações declarativas,
- *API* simples para gestão de transações programáticas e
- excelente integração com a abstração de acesso aos dados do *Spring*.

2.3.3 *Web*

O módulo *Web* é formado pelos módulos *web*, *webmvc*, *websocket* e *webmvc-portlet* [17].

O primeiro módulo *web* é o módulo onde estão contidas as funcionalidades de integração orientada à *web*, como carregamento de ficheiros de várias partes e a inicialização do IoC *Container* usando *Servlet listeners* e um contexto orientado à *web*. Para além disto também aqui está contido um cliente *HTTP* e tudo o que está relacionado com o suporte remoto do *Spring* [17].

O módulo *webmvc* (ou módulo *Web-Servlet*) é o módulo onde se encontra a implementação do *Model-View-Controller* (MVC) e os serviços *web* REST [17]. A *framework* MVC do *Spring* é desenhada com um *DispatcherServlet* que envia pedidos aos *handlers* [18]. O *handler default* é baseado em anotações *@Controller* e *@RequestMapping*. O mecanismo *@Controller* permite criar aplicações *web* RESTful, através da anotação *@PathVariable.* Este módulo fornece funcionalidades únicas de suporte à *web*:

- separação clara de papéis, isto é, a cada papel (*Controller*, *DispatcherServlet*, …) são atribuídos objetos especializados,
- configuração direta de classes da aplicação através de *JavaBeans*. Esta configuração inclui referências de contextos desde controladores da *web* a objetos de negócio e *validators*,
- código de negócio reutilizável, sem necessidade de duplicação,
- disponibilidade de uma biblioteca de etiquetas JSP que torna mais fácil o desenvolvimento de páginas JSP.

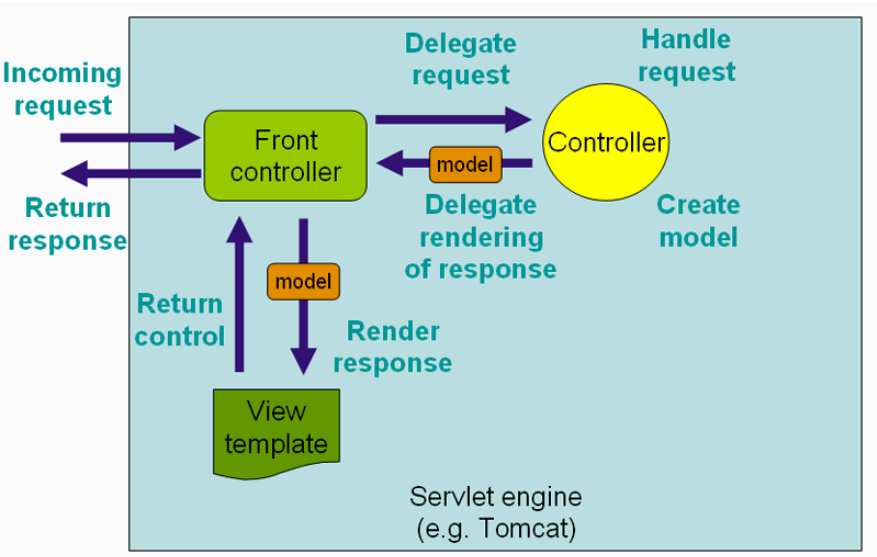

Figura 2.3 - Processamento pedido com Spring Web MVC. Retirada de [18]

Na figura acima (Figura 2.3) está ilustrado o fluxo de processamento de um pedido usando *Spring Web* MVC. Nesta figura o *Front Controller* é conhecido como o

DispatcherServlet. Quando um pedido é feito à aplicação *web* este é enviado para o *Front Controller.* Este decide para qual *Controller* deve reenviar o pedido, com base nas *headers* do mesmo. O *Controller*, após receber o pedido e processá-lo, envia-o para a classe de serviço apropriada. Quando este termina o processamento, o *Controller* recebe o modelo vindo da camada de serviço ou de acesso aos dados e envia-o para o *Front Controller.* Este encontra o *template* da *View* correspondente e envia-lhe o modelo. Através desta *View* a página é criada e enviada de volta para o *Front Controller* que a envia para o *browser* mostrando ao utilizador a resposta ao seu pedido [19].

2.3.4 *AOP (Aspect Oriented Programming), Aspects e Instrumentations*

O módulo AOP contém os módulos *aspects* e *instrument.* É responsável por fornecer uma implementação de programação orientada a aspetos, o que permite que seja possível definir intercetores de métodos e pontos de corte para separar o código que implemente uma funcionalidade com essa necessidade [20]. A explicação detalhada deste tipo de programação e da sua integração com a *framework Spring* encontra-se nos parágrafos seguintes e tem como base a referência [21].

Este tipo de programação funciona como um complemento à Programação Orientada a Objetos (OOP). Enquanto que a base da OOP é a classe, em AOP a unidade de modularidade é o aspeto. Os aspetos permitem a modularização de preocupações, como a gestão de transações em múltiplos tipos e objetos. Um aspeto é implementado através de uma classe normal anotada com *@Aspect*.

Uma das principais componentes do *Spring* é a *framework* AOP. Embora o IoC *Container* não seja dependente de AOP, acaba por complementar o IoC fornecendo uma solução de *middleware* eficiente.

Alguns dos conceitos importantes a reter são: *Join Point*, *Advice* e *Pointcut*. Um *Join Point* é um ponto durante a execução de um programa, como a execução de um método ou lidar com uma exceção. Em AOP aplicado ao *Spring*, um *Join Point* refere-se sempre à execução de um método. Um *Advice* é a ação de um aspeto num certo *Join Point*. Há diferentes tipos de *Advice*: *around*, *before* e *after*. Por último, um *Pointcut* é um predicado que corresponde a *Join Points*. Um *Advice* é associado a uma expressão *Pointcut* e é executado em qualquer *Join Point* correspondente ao *Pointcut* (por exemplo a execução de um método com um nome específico).

O módulo *aspects* é responsável por fornecer integração com o *AspectJ*.

O módulo *Instrumentations* dá suporte de instrumentação e implementação de classes a serem usadas em certos servidores de aplicações. O *Spring* contém o agente de instrumentação *Tomcat*.

2.3.5 *Messaging*

O *Spring* 4 inclui um módulo *Messaging* com abstrações do projeto como *Message*, *MessageChannel*, *MessageHandler* e outras que sirvam para aplicações baseadas em mensagens [22]. Este módulo também disponibiliza um conjunto de anotações para mapear mensagens para métodos, semelhante às anotações do modelo de programação *Spring MVC*.

2.3.6 *Test*

Este último módulo *Test* suporta a componente de *unit testing* e *integration testing* com *JUnit* ou *TestNG* [23]. *Unit Testing* é um nível de testes de *software* onde unidades individuais do *software* são testadas [24]. O objetivo é validar se cada unidade tem o comportamento desejado. A unidade é a menor parte testável de qualquer *software*, o que em programação orientada a objetos corresponde a um método. *Integration testing* é o segundo nível de testes de *software* onde as unidades individuais são combinadas e testadas em grupo [25]. O objetivo deste tipo de testes é expor falhas, caso existam, na interação entre as unidades integradas.

Para além das funcionalidades anteriores também são fornecidos *mock objects* que podem ser usados para testar o código de forma isolada [26]. Esta técnica consiste em instanciar uma versão de teste específica de um componente de *software*, que em vez de ter um comportamento normal fornece apenas resultados pré-computados, e confirmar que é invocada como esperado pelos objetos que estão a ser testados.

2.4 *Angular Framework*

Angular é uma *framework* para desenvolvimento de aplicações em HTML e *TypeScript*. Este projeto foi desenvolvido com a versão 7 do *Angular*. Os blocos básicos do desenvolvimento em *Angular* são os *NgModules*, que fornecem um contexto de compilação para os componentes. Os componentes definem as *Views* que são conjuntos de elementos do ecrã dos quais o *Angular* pode escolher e modificar consoante a lógica e

os dados do programa [27]. Nas subsecções seguintes são apresentados alguns dos blocos fundamentais usados ao longo deste projeto.

2.4.1 *Módulos*

Uma aplicação tem pelo menos um módulo - o *root module*, normalmente chamado de *AppModule* - que permite o uso da *framework bootstrap*. Como acontece em *JavaScript*, os *NgModules* podem importar funcionalidades de outros *NgModules* e permitir que as suas funcionalidades sejam exportadas e usadas por outros. A utilização de módulos ajuda na organização das funcionalidades, especialmente em aplicações mais complexas, assim como na reutilização do código [28].

2.4.2 *Componentes*

Tal como nos módulos, todas as aplicações têm pelo menos um componente – o *root component.* Cada componente é definido por uma classe que contém os dados e lógica da aplicação, e está associado a um *template* HTML que define a *View* a ser mostrada naquele ambiente. Para além disto, os componentes usam serviços que fornecem funcionalidades específicas sem ligação direta com as *Views*. Os fornecedores destes serviços podem ser injetados nos componentes como dependências, tornando o código modular, reutilizável e eficiente [29].

A classe que corresponde a um componente é anotada com o decorador *@Component()*. Estes decoradores são funções que modificam as classes do *JavaScript*. Ao usá-los são anexados tipos específicos de *metadata* às classes, que informam o sistema de quais as suas funções e como se devem comportar [29].

2.4.3 *Serviços e Injeção de Dependências*

Os serviços são usados quando existem certos dados ou lógica da aplicação, não associados a nenhuma *view* específica, que se querem partilhar entre componentes. A classe responsável por definir um serviço é anotada com o decorador *@Injectable()*. Este decorador fornece *metadata* necessários para permitir que o serviço seja injetado nas componentes como uma dependência [30].

2.4.4 *Routing*

O *NgModule Router* fornece um serviço que permite a definição de um caminho de navegação entre os vários estados da aplicação e visualizar as suas hierarquias. Esta definição é modelada através do *browser* onde o URL é colocado na barra de pesquisa e o *browser* navega para essa nova página. Para definir estas regras de navegação, é necessário associar *navigation paths* nas componentes correspondentes [31]. A figura seguinte (Figura 2.4) descreve a forma como todos os blocos mencionados anteriormente se relacionam e funcionam. Este processo, assim como mais detalhes sobre cada componente, está explicado mais à frente nesta secção.

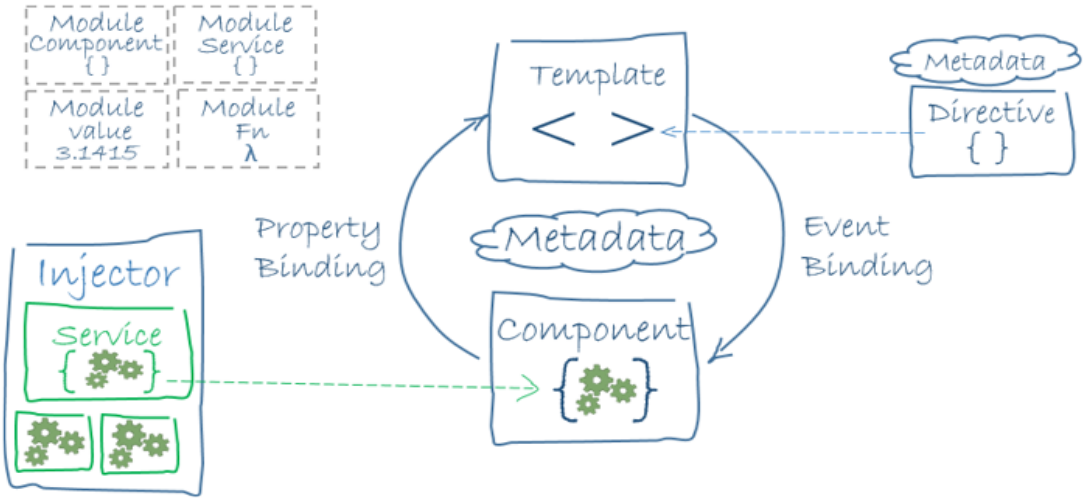

Figura 2.4 - Relação entre os vários blocos do Angular. Retirada de [32]

Um componente e um *template* definem uma *View* onde o decorador da classe componente adiciona *metadata*, assim como um ponteiro para o *template* associado. Para além disto, as diretivas e o *binding markup* no *template* de um componente modifica a *View* baseada na lógica e nos dados do programa. O injetor de dependência fornece serviços a um componente, como o serviço de *router* que permite definir caminhos de navegação entre *Views* [32]. De seguida, são apresentadas três funcionalidades do *Angular* e que foram úteis para o desenvolvimento deste projeto.

2.4.5 *Formulários (Forms)*

A descrição dos formulários aqui apresentada é adaptada da referência [33].

Lidar com o *input* do utilizador através de formulários é comum em muitas aplicações. Estes formulários podem ser utilizados tanto para permitir que os utilizadores

iniciem sessão, alterem campos do seu perfil ou outras tarefas que envolvam a inserção de dados.

O *Angular* fornece duas perspetivas diferentes para lidar com o *input* do utilizador: (i) formulários reativos ou (ii) formulários orientados por modelos. Ambas as perspectivas lidam com os eventos de *input* da *view*, validam-no, criam um modelo de formulário e um modelo de dados de atualização, e dão uma forma de controlar as alterações feitas. No entanto, a forma como processam e gerem os dados é diferente.

Os formulários reativos são mais escaláveis, reutilizáveis e testáveis. Estes devem ser usados caso o uso de formulários seja uma componente muito importante da aplicação ou já estejam a ser usados padrões reativos no desenvolvimento. Os formulários orientados por modelos são úteis quando se quer adicionar um formulário simples, como o registo de um utilizador a partir do seu e-mail. Este tipo de formulários é mais fácil de adicionar a uma aplicação, mas não escala da mesma forma que os do tipo reativo.

Ambos os tipos de formulários partilham blocos de desenvolvimento. O *FormControl* controla o valor e o estado de validação de um formulário individual. O *FormGroup* controla os mesmos valores e estado, porém desta vez a partir de uma coleção de formulários. O *FormArray* controla os mesmos valores e estado de um vetor de formulários.

2.4.6 *Observáveis (Observables)*

Os *Observables* dão suporte à passagem de mensagens entre *publishers* e *subscribers* na aplicação. No que toca a lidar com eventos, com programação assíncrona e com múltiplos valores, os *observables* têm benefícios mais significativos que outras técnicas. Estes componentes são declarativos, isto é, definem uma função para publicar valores, mas esta não é executada até ser subscrita. O *subscriber* depois passa a receber notificações até a função terminar ou até abandonar a subscrição [34].

Como toda a lógica e configuração são da responsabilidade do *Observable*, a aplicação apenas necessita de se preocupar em subscrever para poder consumir valores, e quando terminado, terminar a subscrição. Para definir esta subscrição recorre-se ao método *subscribe()* passando-lhe um *observer*. Este *observer* é um objeto *JavaScript* que define os *handlers* para as notificações a receber [35].

Existem três tipos de notificações que um observável pode enviar: *next*, *error* e *complete* [36]. O primeiro tipo de notificação (*next*) é obrigatória e é chamada por cada

valor a ser entregue e pode, depois da execução começar, ser chamada zero ou mais vezes. As notificações *error* e *complete* são opcionais, sendo que a primeira lida com uma notificação de erro e a segunda lida com uma notificação sobre a execução ter terminado.

2.4.7 *HTTPClient*

Muitas das aplicações *front-end* comunicam com serviços de *back-end* através do protocolo HTTP. O HTTPClient fornece uma *API* HTTP simplificada para aplicação em *Angular* com base na interface *XMLHttpRequest* exposta pelos *browsers*. Para usar esta *API* é necessário importar o módulo *HTTPClientModule.* Salientam-se as funcionalidades de teste, objetos de pedido e resposta, interceção do pedido e resposta, *API*'s de *observables* e manipulação simplificada de erros como sendo algumas das vantagens de utilizar o HTTPClient [37].

Muitas *API*'s do *Angular*, como o caso do HttpClient [38], produzem e consomem *observables* RxJS (*Reactive Extensions for JavaScript*). RxJS é uma biblioteca para escrita de código de forma assíncrona baseado em chamadas de retorno num estilo funcional e reativo. Associados a estas *API*'s estão os vários pedidos feitos ao servidor: *POST*, *DELETE* e *PUT*, explicados sucintamente na Secção 2.2. De notar que se não for aplicado previamente o método *subscribe()* sobre o *observable* que é retornado destes pedidos, o pedido não é executado de todo.

O pedido POST é usado para submeter um formulário [39]. Para além do parâmetro de tipo que recebe e o parâmetro URL, recebe mais dois parâmetros: os dados para submeter no pedido e as opções de método que especificam os cabeçalhos necessários (opcional). Estes cabeçalhos muitas vezes são necessários por parte dos servidores para poderem guardar dados [40]. Por exemplo, pode ser necessário um cabeçalho para declarar o tipo MIME (*Multipurpose Internet Mail Extensions*) do corpo de um pedido ou um *token* de autorização. O pedido DELETE tem a função de pedir para eliminar um recurso [41]. Os parâmetros necessários para executar este pedido são o URL, para onde é feito o pedido, e os dados para submeter o pedido. Por último, o pedido PUT é responsável por substituir completamente um recurso com dados atualizados [42]. No que toca a parâmetros, é composto exatamente da mesma forma que o pedido POST. Para além destes pedidos, ainda existe o pedido GET que é semelhante ao pedido POST [43], na medida em que recebe os mesmos dois primeiros parâmetros que este, porém em vez de submeter um formulário, faz um pedido de dados.

2.5 Metodologia V-Model

V-Model, também conhecido como modelo de Verificação e Validação [44], é uma metodologia de desenvolvimento linear usada durante o ciclo de vida do desenvolvimento de *software* [45]. Este modelo tem como base o método *waterfall*, onde são seguidas as várias fases de desenvolvimento passo-a-passo, porém com uma fase de testes associada diretamente a cada etapa de desenvolvimento. Na Figura 2.5, está esquematizada esta metodologia de desenvolvimento. O ciclo inicia-se do lado esquerdo do topo do V convergindo aos poucos para o topo do lado direito.

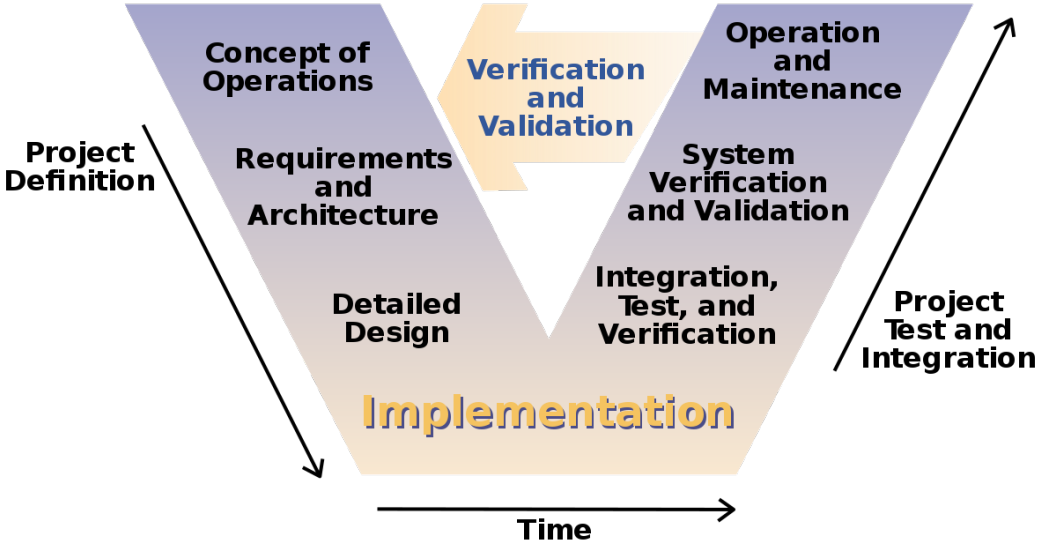

Figura 2.5 - Fases do modelo V-Model. Retirada de [44]

De um lado (o esquerdo do V) está toda a definição do projeto, isto é, definição dos requisitos, da arquitetura a ser usada e dos testes, assim como o desenho detalhado do sistema e dos seus módulos. Do outro lado (o direito do V) estão as ações que dizem respeito à realização de testes do projeto e à integração do mesmo com as componentes externas. Esta fase de testes corresponde à verificação de que o sistema está a comportar-se de acordo com o que foi pensado inicialmente. Na base deste modelo encontra-se a fase de implementação, que deve levar o tempo necessário para que tudo o que foi planeado seja codificado.

O uso desta metodologia permite a redução de riscos associados ao desenvolvimento do projeto, a garantia de qualidade, redução do custo total do projeto e a melhoria na comunicação entre todas as partes envolvidas [46], na medida em que a análise de um *deliverable* permite avaliar a evolução do mesmo. Esta metodologia, por oposição às metodologias denominadas por Agile, é mais adequada para aplicar em

projetos cujos requisitos estejam bem definidos e não sejam ambíguos ou indefinidos, bem como no caso em que a tecnologia a ser usada não seja dinâmica e seja bem compreendida pela equipa, e em casos em que a duração do projeto seja relativamente curta. Para entender se esta metodologia é a ideal a aplicar num certo projeto é necessário colocar lado a lado as vantagens e desvantagens no domínio do projeto em causa [44].

As vantagens deste modelo são:

- leva a que seja feita uma verificação e validação do produto logo em fases iniciais do desenvolvimento,
- cada etapa de desenvolvimento é testável,
- permite que o gestor de projeto acompanhe o seu progresso através de *milestones* e
- facilidade de entendimento, implementação e uso.

As desvantagens deste modelo são:

- não permite lidar com eventos concorrentes,
- não lida com iterações,
- não é apropriado para mudanças dinâmicas nos requisitos do projeto e
- não tem em conta uma análise ou mitigação de risco no que diz respeito à mudança.

Esta foi a metodologia escolhida para o desenvolvimento deste projeto. Para além de ser uma metodologia bastante usada na empresa onde este projeto está inserido, foi também considerada a ideal pelo facto de a duração do projeto ser relativamente curta (~9 meses), os requisitos estarem bem definidos logo desde início e permitir o teste do produto em cada fase de desenvolvimento, reduzindo desta forma os riscos de desvio dos objetivos de desenvolvimento. Apesar das desvantagens mencionadas anteriormente, aquando da adoção não se previa que nenhuma delas influenciasse o decurso do trabalho, e, portanto, não fossem considerados fatores decisivos na escolha da melhor metodologia. De referir que tal como se previu, e olhando retrospetivamente para as decisões tomadas no final do projeto, este método revelou-se adequado.

Capítulo 3

Planeamento e desenho

Este capítulo encontra-se omisso por confidencialidade.

Capítulo 4

Desenvolvimento

Este capítulo encontra-se omisso por confidencialidade.

Capítulo 5

Testes

Este capítulo encontra-se omisso por confidencialidade.

Capítulo 6

Conclusão

Ao longo deste projeto foi desenvolvida uma prova de conceito para apresentação de uma fatura de forma interativa com possibilidade de interação com um assistente virtual (desenvolvido em paralelo, mas fora do âmbito deste projeto). O facto deste projeto ter sido desenvolvido na Accenture, em ambiente empresarial, ajudou a ter uma maior noção de como são normalmente desenvolvidos estes projetos de maior envergadura em contexto real. Apesar de ser uma prova de conceito, existiu uma comunicação constante entre o supervisor do projeto e os *developers* (neste caso eu e o meu colega José Pedro Rodrigues), e também com outros elementos da empresa que foram fundamentais para uma melhor adaptação às normas da empresa e uma melhor compreensão dos sistemas legados.

Este projeto foi também uma fonte de aprendizagem no que diz respeito a novas tecnologias, neste caso o *Angular* e o *Spring*, tecnologias não usadas anteriormente durante o percurso académico. Foi necessário um período inicial dedicado à familiarização com as *frameworks* e para perceber como funcionam e como podiam interagir entre si. Com a primeira tecnologia foi possível abordar uma nova forma de desenvolver uma aplicação *front-end* e com a segunda abordaram-se aspetos como o padrão MVC, o módulo *JDBC* e *JPA*, e a programação por aspetos. Estes módulos eram já conhecidos em contexto académico.

Tendo em conta o plano inicial previsto para a execução das tarefas e os objetivos definidos no início deste projeto, e que foram apresentados na Secção 1.3 deste relatório, pode dizer-se que foram cumpridos na íntegra e totalmente. Com os testes de usabilidade feitos e com a análise dos resultados é possível concluir que os utilizadores acharam que, no geral, a aplicação é intuitiva, tem um aspeto atrativo e que recomendariam o seu uso. Um dos principais objetivos desta aplicação é dar um contexto visual à interação entre o cliente e o assistente virtual, e o facto dos utilizadores que participaram nos testes de usabilidade terem apontado que a aplicação era interativa e que a interação com o assistente virtual era uma mais valia para a completa perceção da fatura, reforça que esta prova de conceito foi concretizada com sucesso e para aquilo a que foi proposta. No que

diz respeito ao contexto empresarial, e sendo este projeto uma prova de conceito, ficou demonstrado que o conceito funciona, que é aplicável numa situação real, e que a aplicação cumpre com os objetivos estabelecidos inicialmente. Para além disto, este projeto tem potencial para funcionar como um projeto *omnichannel*, isto é, várias plataformas podem consumir este mesmo serviço, desde uma aplicação já existente, passando por uma área de cliente num *website* até uma *box* em casa do cliente.

Bibliografia

[1] - O que fazemos. (31 de Outubro de 2018). Obtido de accenture: https://www.accenture.com/pt-pt/company

[2] - Chatbots: An overview Types, Architecture, Tools and Future Possibilities. (2017). Em K. Nimavat, & T. Champanerla, *IJSRD - International Journal for Scientific Research and Development* (pp. 1019-1026). ResearchGate.

[3] - MVC architecture. (07 de Novembro de 2018). Obtido de MDN web docs: https://developer.mozilla.org/en-US/docs/Web/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture

[4] - MVC in Objective-C (I): Introduction. (07 de Novembro de 2018). Obtido de Angel G. Olloqui personal website: http://angelolloqui.com/blog/26-MVC-in-Objective-C-I-Introduction

[5] - Design Patterns - MVC Pattern. (07 de Novembro de 2018). Obtido de tutorialspoint: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm

[6] - The MVC pattern in theory and practice. (07 de Novembro de 2018). Obtido de Warp: http://warp.povusers.org/programming/mvc.html

[7] - What is REST? (08 de Novembro de 2018). Obtido de Codecademy: https://www.codecademy.com/articles/what-is-rest

[8] - Understanding REST. (08 de Novembro de 2018). Obtido de *Spring*: https://spring.io/understanding/REST

[9] - Spring Framework Overview. (02 de Novembro de 2018). Obtido de *Spring* Framework Overview: https://docs.spring.io/spring/docs/5.1.2.RELEASE/spring-framework-reference/overview.html#overview

[10] - 2.2.1 Core Container. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-core-container

[11] - 7.1 Introduction to the Spring IoC container and beans. (02 de Novembro de 2018). Obtido de Sprin Framework Reference Documentation:

https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#beans-introduction

[12] - Lesson: Overview of JNDI. (02 de Novembro de 2018). Obtido de Lesson: Overview of JNDI: https://docs.oracle.com/javase/tutorial/jndi/overview/index.html

[13] - 2.2.4 Data Access/Integration. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-data-access

[14] - 19.1 Introduction to Spring Framework JDBC. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#jdbc-introduction

[15] - 30. JMS (Java Message Service). (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#jms

[16] - 17. Transaction Management. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#transaction

[17] - 2.2.5 Web. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-web

[18] - 22.1 Introduction to Spring Web MVC framework. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#mvc-introduction

[19] - Spring Web MVC Framework Flow. (02 de Novembro de 2018). Obtido de *Spring* Web MVC Flow: https://www.onlinetutorialspoint.com/spring/spring-web-mvc-framework.html

[20] - 2.2.2 AOP and Instrumentation. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation:

https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-aop-instrumentation

[21] - 11.1 Introduction. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#aop-introduction

[22] - 2.2.3 Messaging. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-messaging

[23] - 2.2.6 Test. (02 de Novembro de 2018). Obtido de *Spring* Framework Reference Documentation: https://docs.spring.io/spring/docs/4.3.21.BUILD-SNAPSHOT/spring-framework-reference/htmlsingle/#overview-testing

[24] - Unit Testing. (2018 de Novembro de 2018). Obtido de Unit Testing - Software Testing Fundamentals: http://softwaretestingfundamentals.com/unit-testing/

[25] - Integration Testing. (02 de Novembro de 2018). Obtido de Integration Testing - Software Testing Fundamentals: http://softwaretestingfundamentals.com/integration-testing/

[26] - Mock Objects. (02 de Novembro de 2018). Obtido de What are Mock Objects?: https://www.agilealliance.org/glossary/mocks/

[27] - Architecture overview. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#architecture-overview

[28] - Modules. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#modules

[29] - Components. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#components

[30] - Services and dependency injection. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#services-and-dependency-injection

[31] - Routing. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#routing

[32] - What's next. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/architecture#whats-next

[33] - Introduction to forms in Angular. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/forms-overview

[34] - Observables. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/observables

[35] - Basic usage and terms. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/observables#basic-usage-and-terms

[36] - Defining observers. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/observables#defining-observers

[37] - HttpClient. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#httpclient

[38] - Observables and operators. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#observables-and-operators

[39] - Making a POST request. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#making-a-post-request

[40] - Adding headers. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#adding-headers

[41] - Making a DELETE request. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#making-a-delete-request

[42] - Making a PUT request. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#making-a-put-request

[43] - URL Parameters. (06 de Novembro de 2018). Obtido de *Angular*: https://angular.io/guide/http#url-parameters

[44] - SDLC - V-Model. (07 de Novembro de 2018). Obtido de tutorialspoint: https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm

[45] - V-Model: What Is It And How Do You Use It? (07 de Novembro de 2018). Obtido de Airbrake: https://airbrake.io/blog/sdlc/v-model

[46] - V Model. (07 de Novembro de 2018). Obtido de tutorialspoint: https://www.tutorialspoint.com/software_testing_dictionary/v_model.htm