Instituto Politécnico de Tomar

Escola Superior de Tecnologia de Tomar

Paulo Jorge Borges da Costa

Garantia de Qualidade de *Software* (SQA)

Projecto e Implementação do Processo de Automatização de *Release* de *Software*

Relatório de Estágio

Orientado por:

Prof. Doutor Paulo Coelho - Instituto Politécnico de Tomar

Relatório de Estágio apresentado ao Instituto Politécnico de Tomar
para cumprimento dos requisitos necessários à
obtenção do grau de Mestre em
Engenharia Electrotécnica – Especialização em Controlo e Electrónica Industrial

Tomar, Novembro 2019

Resumo

A tecnologia tornou-se de tal forma parte integral da nossa vida que todos os dias recorremos a meios tecnológicos que funcionam com base em *software* criado para satisfazer múltiplas funções. É, portanto, normal que se tenha vindo a observar um aumento exponencial no desenvolvimento de *software*. Em determinados momentos, chegamos mesmo a confiar as nossas vidas ao *software*, pode-se olhar para exemplos onde tal acontece, na área da saúde, na economia ou até mesmo na mobilidade onde os nossos próprios automóveis têm vários sistemas de controlo e segurança baseados em *software*. Como resultado, tem-se atribuído cada vez mais importância à área da garantia de qualidade. Nos últimos anos, este campo tem demonstrado resultados positivos nos processos de desenvolvimento de *software* ao obter melhor eficiência e reduzindo o risco, que, por sua vez, se traduzem em benefícios para a economia e segurança.

Este relatório, tem como propósito, contextualizar de que forma funcionam os processos de desenvolvimento de *software* e como são aplicadas sistematicamente, na Engenharia, as atividades de garantia de qualidade nesses processos. Finalmente, o trabalho de garantia de qualidade realizado na empresa, *Bosch* Termotecnologia (Aveiro), no âmbito deste estágio de mestrado, é descrito dando ênfase ao projecto e implementação do processo de automatização de *release* de *software*, e consequentemente a todas as tarefas inerentes e paralelas à sua implementação.

Palavras-chave: Automação, Internet das Coisas, Qualidade, *Software*, *Release*.

Abstract

Technology has become such an integral part of our daily life that every day we resort to technological means which work based on software created to satisfy multiple functions. Therefore, it is normal that an exponential increase in software development has been and continues to be observed. In certain moments, we even trust our lives to software such as in the areas of healthcare, economy or even mobility where our own vehicles have multiple control and security systems based in software. As a result, increasingly more importance has been attributed to the quality assurance field. In recent years, this field has shown positive results in software development processes such as obtaining greater efficiency and reducing risks, which ultimately, translates to economic and security benefits.

*This report's purpose is to concisely explain how software development processes function and how the quality assurance activities are applied systematically, in Engineering, to these processes. Finally, the quality assurance work accomplished in the company, Bosch Thermotechnology (*Aveiro*), under this master's degree internship, is described with emphasis on the design and implementation of the process of automating the release of software, and consequently on all inherent and parallel tasks in its implementation.*

Keywords: *Automation, Internet of Things, Quality, Software, Release.*

Agradecimentos

À minha família, pelas oportunidades que me proporcionaram, entre elas a de adquirir uma formação académica, e por todo o apoio e incentivo incondicional que me deram ao longo da vida.

Ao meu orientador, o Prof. Doutor Paulo Coelho pelo apoio, acompanhamento e disponibilidade prestados. Bem como aos restantes docentes de Engenharia Electrotécnica do IPT responsáveis pela minha formação ao longo deste mestrado.

À direcção das equipas TT-RHC/EPQ e TT/EPQ-CE da *Bosch* Termotecnologia, e em especial ao Engenheiro Rafael Peixinho, pela oportunidade da realização do estágio de mestrado e respectivo acompanhamento.

A todos os colaboradores da *Bosch* Termotecnologia de Aveiro, pela disponibilidade, acolhimento, partilha de informação e transmissão de conhecimento durante o período de estágio, com um especial agradecimento aos elementos das equipas TT-RHC/EIS2-Av e TT/EIS5-Av.

Índice Principal

Resumo V
Abstract VII
Agradecimentos IX
Índice Principal XI
Índice de Figuras XV
Índice de Tabelas XIX
Lista de Abreviaturas XXI
Capítulo 1. Introdução 1
1.1. Enquadramento e Objectivos do Estágio 1
1.2. Estrutura do Relatório 2
Capítulo 2. A Empresa 3
2.1. A Bosch no Mundo 3
2.2. Unidades de Negócio 4
2.3. A Bosch em Portugal 5
2.4. Divisão de Termotecnologia 5
Capítulo 3. A Internet das Coisas (IoT) 7
3.1. Que problemas podem advir com IoT? 8
3.2. IoT na Bosch 9
Capítulo 4. Desenvolvimento de *Software* 15
4.1. Fases Gerais do Desenvolvimento de *Software* 15
4.2. SQA (Garantia de Qualidade de *Software*) 18
4.3. *Quality Gates* (Critérios de Aceitação) 20
4.4. *Definition of Done* (Definição de Pronto) 20
4.5. Integração, Entrega e Passagem à Produção Contínuas (CI/CD) 23
4.5.1. *Continuous Integration* (Integração Contínua) 23
4.5.2. *Continuous Delivery* (Entrega Contínua) 24
4.5.3. *Continuous Deployment* (Passagem à Produção Contínua) 24
4.6. REST API 25
4.6.1. API (Interface de Programação de Aplicações) 26
4.6.2. REST (Transferência de Estado Representativa) 27

4.6.3. URI, URL e URN (Identificador, Localizador e Nome de Recurso Uniformes) 29
4.6.4. HTTP (Protocolo de Transferência de Hipertexto) 30
4.6.5. REST API 34
4.7. *Software* (Tecnologias) Utilizado no Estágio 36
4.7.1. Sistema Operativo 36
4.7.2. Sistema de Virtualização 37
4.7.3. Sistema de Controlo de Versão 37
4.7.4. Repositório e Gestão do Código Fonte 39
4.7.5. Gestão de Projectos 39
4.7.6. Gestão de Testes 41
4.7.7. Análise Estática do Código Fonte 43
4.7.8. Análise das Vulnerabilidades a Nível de Segurança 43
4.7.9. Repositório de Artefactos / Binários 44
4.7.10. Documentação de Projectos 45
4.7.11. Ambiente de Desenvolvimento Integrado (IDE) 45
4.7.12. *Framework* de Testes 46
4.7.13. *Build Automation Tool* 47
4.7.14. Integração, Entrega e Deploy Contínuos 48
4.7.15. Colaboração e Desenvolvimento de APIs 49
4.7.16. Outros 49
Capítulo 5. Trabalho Desenvolvido 51
5.1. Desenho e Automatização do Processo de *Release* de *Software* 52
5.1.1. Porquê automatizar este processo? 52
5.1.2. Como automatizar? 53
5.1.3. Análise ao Estado das *Pipelines* das Equipas (Recolha dos *Jenkinsfiles*) 55
5.1.4. Criação do Conceito de *Release* Automática de *Software* 56
5.1.5. Criação da Biblioteca Partilhada de Funções em *Groovy* 60
5.1.6. Adopção de um Modelo Jenkins Eficiente 64
5.1.7. Teste do Conceito com Recurso a um Projecto de Demonstração 66
5.1.8. Integração de Ferramentas e Criação de Tutoriais 67
5.2. Mapeamento de Requisitos com Testes no *Xray* 67
Capítulo 6. Conclusões 75

Referências 77

Índice de Figuras

Figura 1 – **Esquentador Bosch Greentherm 9000** [11]. 11
Figura 2 – **Ilustração do Controlo Remoto do Aquecimento Central através de um Smartphone** [14]. 13
Figura 3 – **Cronotermóstato Inteligente Bosch EasyControl** [15] & **Exemplo da Aplicação EasyControl na Programação de uma Rotina** [16]. 13
Figura 4 – **Válvula Termostática de Radiador Inteligente (Smart TRV)** [15]. 14
Figura 5 – **Ciclo de Vida do Desenvolvimento de Software** [19]. 15
Figura 6 – **Fluxo Geral de um Projecto submetido a Quality Gates** [24]. 20
Figura 7 – **Ilustração do envolvimento entre Integração, Entrega e Passagem à Produção Contínuas** [26]. 25
Figura 8 – **Representação gráfica da definição ou função de uma API** [28]. 26
Figura 9 – **Comparação entre as estruturas de um ficheiro XML e de um ficheiro JSON**. 28
Figura 10 – **Exemplo da estrutura constituinte de um URL** [31]. 30
Figura 11 – **O que representam URI, URL e URN e a forma como se relacionam**. 30
Figura 12 – **Esquemático de um ciclo de Pedido e Resposta via HTTP entre utilizador e servidor**. Imagem adaptada de [33]. 31
Figura 13 – **Estrutura típica de uma mensagem HTTP**. Imagem adaptada de [33]. 31
Figura 14 – **Estrutura típica e respectivo exemplo de um pedido HTTP**. Imagem adaptada de [33]. 32
Figura 15 – **Estrutura típica e respectivo exemplo de uma resposta HTTP**. Imagem adaptada de [33]. 32
Figura 16 – **Tabela de operações CRUD com as respectivas declarações em SQL e verbos em HTTP** [35]. 33
Figura 17 – **Esquemático de uma comunicação feita através de uma REST API**. Imagem adaptada de [36]. 35
Figura 18 – **Exemplo e estrutura de um pedido HTTP**. Imagem adaptada de [36]. 35
Figura 19 – **Logotipos dos Sistemas Operativos Windows 10** [37] & **Linux Ubuntu** [38]. 36
Figura 20 – **Logotipo do Sistema de Virtualização Virtual Box** [39]. 37

Figura 21 – **Logotipo do Sistema de Controlo de Versão Git** [40]. 37
Figura 22 – **Modelo do Gitflow criado por Vincent Driessen** [42]. 38
Figura 23 – **Logotipo do Repositório Bitbucket** [43]. 39
Figura 24 – **Logotipo do Gestor de Projectos JIRA** [44]. 39
Figura 25 – **Exemplo de um quadro Scrum no JIRA** [45]. 40
Figura 26 – **Exemplo de vários gráficos que ilustram o progresso de um projecto no JIRA** [45]. 40
Figura 27 – **Logotipo do Gestor de Testes Xray** [46]. 41
Figura 28 – **Exemplo de organização das tarefas (issues) do Xray com base na sua função**. 42
Figura 29 – **Exemplo da vista dos Sprints de um projecto no JIRA Xray** [47]. 42
Figura 30 – **Exemplo da rastreabilidade mantida pelo JIRA Xray entre as várias tarefas (issues)** [48]. 42
Figura 31 – **Logotipo do Software de Análise Estática SonarQube** [49]. 43
Figura 32 – **Logotipo do Software de Verificação de Segurança Checkmarx** [50]. 43
Figura 33 – **Logotipo do Repositório Artifactory** [51]. 44
Figura 34 – **Logotipo da Plataforma de Documentação Confluence** [52]. 45
Figura 35 – **Logotipo do IDE IntelliJ IDEA** [53]. 45
Figura 36 – **Logotipo da Framework de Testes JUnit** [54]. 46
Figura 37 – **Logotipo Alternativo da Framework de Testes TestNG** [55]. 47
Figura 38 – **Logotipo da Ferramenta de Automatização Maven** [56]. 47
Figura 39 – **Logotipo da Ferramenta Jenkins** [57]. 48
Figura 40 – **Exemplo ilustrativo dos benefícios da implementação de integração contínua num projecto**. 49
Figura 41 – **Logotipo da Ferramenta de Desenvolvimento de APIs Postman** [58]. 49
Figura 42 – **Logotipos das Ferramentas de Comunicação Outlook** [59]**, Skype** [60] **e Rocket Chat** [61]. 49
Figura 43 – **Representação de uma pipeline do Jenkins e respectivas etapas configuradas no Jenkinsfile**. 54
Figura 44 – **Esquema ilustrativo de um projecto com vários ramos (multibranch) e de como estes se podem comportar de forma diferente no Jenkins dependendo das suas configurações**. 55

Figura 45 – **Conceito de lançamento de software automático proposto**. 57
Figura 46 – **Realce de algumas tarefas necessárias (e ferramentas associadas) no desenvolvimento de um software para que este possa ser lançado. Base sólida para uma definição de pronto (DoD)**..... 58
Figura 47 – **Ilustração de como se pretende cumprir com os requisitos base necessários para a garantia de qualidade de forma automática através de uma pipeline do Jenkins e com recurso às funções desenvolvidas presentes na biblioteca partilhada – se algum dos passos falhar, não é possível avançar com o lançamento do software**. 60
Figura 48 – **Exemplo do funcionamento do POSTMAN com um pedido e resposta HTTP** [63]. 61
Figura 49 – **Exemplo de um método desenvolvido para a biblioteca de funções partilhada – este método tinha como intuito exportar a documentação definida no formato HTML**. 62
Figura 50 – **Exemplo de documentação acerca dos métodos desenvolvidos referente aos pré-requisitos necessários à utilização desses métodos**. 63
Figura 51 – **Exemplo de documentação acerca dos métodos desenvolvidos que contém uma breve descrição sobre os mesmos**..... 64
Figura 52 – **Exemplo de um Jenkinsfile que fica ao nível dos projectos e invoca o verdadeiro Jenkinsfile que contém as configurações da pipeline do Jenkins**. 65
Figura 53 – **Exemplo de um Jenkinsfile que contém as configurações para uma pipeline do Jenkins e que faz uso dos métodos disponibilizados na biblioteca de funções partilhada**. 65
Figura 54 – **Modelo Jenkins com uma camada de abstração e forma como se relaciona com os projectos e com a biblioteca de funções partilhada**. 66
Figura 55 – **Pipeline do Jenkins para o projecto de demonstração em acção**..... 67
Figura 56 – **Requisito de demonstração sem qualquer Teste associado (na plataforma JIRA Xray)**. 68
Figura 57 – **Classe com o código fonte para demonstração correspondente às funcionalidades aritméticas básicas de uma calculadora**. 69
Figura 58 – **Classe de testes (contém um teste para cada método presente no código fonte) com realce dos elementos que farão a associação automática entre Testes e Requisitos na plataforma de gestão de testes**. 70

Figura 59 – **Classe com anotações específicas para o Xray**. 71
Figura 60 – **Classe de interface que processará as anotações do Xray e as passa como atributos ao TestNG**. 72
Figura 61 – **Relatório gerado em XML com os resultados dos testes e com realce para os atributos passados**..... 73
Figura 62 – **Momento em que o Xray processa o relatório de resultados, numa pipeline do Jenkins**..... 74
Figura 63 – **Requisito de demonstração com todos os Testes associados automaticamente (na plataforma do JIRA Xray)**. 74

Índice de Tabelas

Tabela 1 – **Resumo das tarefas realizadas durante o período de estágio**. 51

Tabela 2 – **Exemplo de documentação acerca dos métodos desenvolvidos referente aos parâmetros presentes em cada método e respectiva descrição**..... 63

Lista de Abreviaturas

AI – *Artificial Intelligence*;

API – *Application Programming Interface*;

ATDD – *Acceptance Test Driven Development*;

BDD – *Behaviour Driven Development*;

CD – *Continuous Deployment* ou *Continuous Delivery*;

CI – *Continuous Integration*;

CSS – *Cascading Style Sheets*;

CSV – *Comma-Separated Values*;

DDS – *Design Document Specifications*;

DevOps – *Development & Operations*;

e2e – *End to End*;

GmbH – *Gesellschaft mit beschränkter Haftung* (Sociedade Limitada);

GSEP – *Global Scientists and Engineers Program*;

GUI – *Graphical User Interface*;

HTML – *HyperText Markup Language*;

IDE – *Integrated Development Environment*;

IEEE – *Institute of Electrical and Electronics Engineers*;

IoT – *Internet of Things*;

ISO – *International Organization for Standardization*;

ISTQB – *International Software Testing Qualifications Board*;

ISQI – *International Software Quality Institute*;

JDK – *Java Development Kit*;

JSON – *JavaScript Object Notation*;

KG – *Kommanditgesellschaft* (Parceria Limitada);

ML – *Machine Learning*;

OS – *Operative System*;

OSS – *Open-Source Software*;

PHP – *Hypertext Preprocessor* (originalmente *Personal Home Page*);

PO – *Product Owner*;

POM – *Project Object Model*;

REST – *Representational State Transfer*;

RoI – *Return of Investment*;

SDK – *Software Development Kit*;

SDLC – *Software Development Life Cycle*;

SQA – *Software Quality Assurance*;

SRS – *Software Requirement Specifications*;

TDD – *Test Driven Development*;

TMap – *Test Management Approach*;

TT – *Thermotechnology*;

UAT – *User Acceptance Testing*;

UI – *User Interface*;

URL – *Uniform Resource Locator*;

US – *User Story*;

VAN – *Verfahrensanwnweisung* (Instruções Processuais);

XML – *Extensible Markup Language*;

Capítulo 1. Introdução

No âmbito do curso de Mestrado em Engenharia Electrotécnica (Especialização em Controlo e Electrónica Industrial) surgiu a possibilidade de realização de um estágio curricular em ambiente industrial na *Bosch* Termotecnologia em Aveiro. A escolha do estágio curricular é bastante gratificante, pois assim é possível usufruir de uma formação em contexto de trabalho e adquirir outros conhecimentos, para além dos adquiridos durante a formação académica.

Com o presente relatório pretende-se apresentar um pequeno enquadramento em torno do desenvolvimento de *software* bem como das actividades que levam à garantia de qualidade do mesmo. Por fim, serão descritas resumidamente as actividades realizadas, ao longo do período de estágio.

1.1. Enquadramento e Objectivos do Estágio

O estágio teve início com a integração numa equipa de desenvolvimento de *software* (TT-RHC/EIS2-Av[1]), constituída por 6 elementos (2 dos quais em estágio). Dentro desta equipa, os elementos em estágio trabalhavam junto da equipa responsável pela garantia de qualidade (TT/EPQ-CE[2]), que abrangia entre outras tarefas, implementação de testes e suporte para todas as equipas do Centro de Investigação e Desenvolvimento da *Bosch* Termotecnologia.

O estágio de mestrado teve lugar nas instalações da *Bosch* Termotecnologia em Aveiro, e foi subordinado ao tema "Projecto e Implementação do Processo de Automatização de *Release* de *Software*". De modo a levar a cabo o tema que foi sugerido, o estágio, de uma forma genérica, foi dividido em:

- Preparação do ambiente de trabalho;

[1] Acrónimo da equipa: *Thermotechnology / Engineering IoT Systems & Solutions 2* – Aveiro
[2] Acrónimo da equipa: *Thermotechnology / Engineering Product Quality – Center of Engineering*

- Levantamento, análise e recolha de informações sobre critérios e padrões para a garantia de qualidade no *software*;
- Criação de um conceito de *release* automática de *software* que cumpra com os padrões de qualidade, e respectivos testes, de forma a agilizar o lançamento (*release*) de *software* para produção.

De uma forma mais discriminada, as várias tarefas desempenhadas no estágio, estão detalhadas no capítulo 5.

1.2. Estrutura do Relatório

O presente relatório divide-se em 6 capítulos. O primeiro capítulo é composto pela introdução, enquadramento e objectivos do estágio e pela estrutura do relatório. No capítulo 2, será feita uma pequena apresentação da empresa *Bosch* Termotecnologia. O Capítulo 3 diz respeito à Internet das Coisas (IoT) e de que forma é que se interliga com o trabalho desenvolvido no estágio. No capítulo 4 – Desenvolvimento de *Software* – serão apresentados os passos necessários para um desenvolvimento de *software* com sucesso; serão ainda apresentadas as ferramentas e programas utilizados na sua implementação. No capítulo 5 – Trabalho Desenvolvido – são apresentadas com detalhe as várias tarefas desempenhadas e desenvolvidas durante o estágio de mestrado. Por fim, no capítulo 6 serão apresentadas as conclusões, onde são evidenciados os aspectos mais importantes do trabalho e do estágio de mestrado.

Capítulo 2. A Empresa

A empresa foi fundada em Estugarda no ano de 1886 por *Robert Bosch* (1861-1942) como uma "oficina de mecânica de precisão e eletricidade". A estrutura acionista da *Robert Bosch GmbH* garante a autonomia empresarial do Grupo *Bosch*, tornando possível o planeamento a longo prazo e a realização de investimentos significativos para salvaguarda do seu futuro. Noventa e dois por cento das ações da *Robert Bosch GmbH* são detidas pela Fundação *Robert Bosch*, uma fundação beneficente. A maioria dos direitos de voto é detida pela *Robert Bosch Industrieltreuhand KG*, uma *trust* industrial a quem está confiada a gestão dos activos empresariais. As restantes acções são detidas pela família *Bosch* e pela *Robert Bosch GmbH* [1].

2.1. A Bosch no Mundo

O Grupo *Bosch* é líder mundial no fornecimento de tecnologia e serviços. A empresa emprega aproximadamente 410.000 colaboradores em todo o mundo (a 31.12.2018), que contribuíram para gerar uma faturação de 78,5 mil milhões de euros em 2018. As operações do Grupo estão divididas em quatro áreas de negócio: Soluções de Mobilidade, Tecnologia Industrial, Bens de Consumo, e Tecnologia de Energia e Edifícios. Líder em IoT, a *Bosch* oferece soluções inovadoras para casas e cidades inteligentes, mobilidade e indústria conectada. A empresa utiliza o seu conhecimento em tecnologia de sensores, *software* e serviços, bem como a sua própria *cloud* IoT para oferecer aos seus clientes soluções conectadas e em diversos domínios a partir de uma única fonte. O objetivo estratégico da *Bosch* é fornecer inovações para uma vida conectada. Os produtos e serviços do Grupo *Bosch* são concebidos para cativar e melhorar a qualidade de vida das pessoas através de soluções inovadoras e úteis. Desta forma, a empresa oferece mundialmente "Tecnologia para a Vida". O Grupo *Bosch* é composto pela *Robert Bosch GmbH* e por cerca de 460 subsidiárias e empresas regionais presentes em aproximadamente 60 países. Se for incluído os representantes de vendas e serviços, a rede mundial de desenvolvimento, a produção e distribuição da *Bosch*, pode-se dizer que está presente em quase todos os países. A sua força inovadora é a base para a continuidade do crescimento da empresa. Em cerca de 130

localizações em todo o mundo, a *Bosch* emprega 68.700 colaboradores em investigação e desenvolvimento [2] [3].

2.2. Unidades de Negócio

Como referido na secção anterior, a empresa divide-se em 4 principais áreas de negócio, cada uma com as respectivas subdivisões, nomeadamente [2] [3]:

- **Soluções de Mobilidade**
  - **PS** – *Powertrain Solutions* (Soluções para Cadeia Cinemática)
  - **CC** – *Chassis Systems Control* (Controlo de Sistemas dos Chassis)
  - **ED** – *Electrical Drives* (Accionamentos Eléctricos)
  - **CM** – *Car Multimedia* (Multimédia do Automóvel)
  - **AE** – *Automotive Electronics* (Electrónica Automóvel)
  - **AA** – *Automotive Aftermarket* (Mercado de Pós-Venda Automóvel)
  - **AS** – *Automotive Steering* (Sistemas de Direcção Automóvel)
  - *Connected Mobility Solutions* (Soluções de Mobilidade Conectada)
- **Tecnologia Industrial**
  - **DC** – *Drive and Control Technology* (Tecnologia de Accionamentos e Controlo)
  - **PA** – *Packaging Technology* (Tecnologia de Embalagens)
- **Bens de Consumo**
  - **PT** – *Power Tools* (Ferramentas Eléctricas)
  - **BSH Hausgeräte GmbH** – *Household Appliances* (Electrodomésticos)

- **<u>Tecnologia de Energia e Construção</u>**
  - **BT** – *Building Technologies* (Tecnologias de Construção)
  - **TT** – *Thermotechnology* (Termotecnologia)
  - *Bosch Global Service Solutions* (Soluções para Serviços Globais da Bosch)
  - *Robert Bosch Smart Home GmbH* (Soluções IoT)

2.3. A Bosch em Portugal

A *Bosch* é representada em Portugal pela *Bosch* Termotecnologia, em Aveiro, a *Bosch Car Multimedia* Portugal, em Braga, e a *Bosch Security* Systems – Sistemas de Segurança, em Ovar. Nestas localizações, a empresa desenvolve e fabrica, respectivamente, soluções de água quente, multimédia automóvel e sistemas de segurança e comunicação, 95% dos quais exportadas para os mercados internacionais. A sede do Grupo no país está em Lisboa, onde são realizadas atividades de vendas, *marketing*, contabilidade e comunicação, bem como serviços partilhados de recursos humanos e comunicação para o Grupo *Bosch*. Além disso, a empresa possui ainda uma subsidiária da BSH Eletrodomésticos, em Lisboa. Com mais de 5.300 colaboradores, a *Bosch* é um dos maiores empregadores industriais de Portugal e gerou, em 2017, 1,5 mil milhões de euros em vendas internas [2] [3] [4].

2.4. Divisão de Termotecnologia

A divisão de termotecnologia fornece produtos de aquecimento com eficiência energética e soluções de água quente (essencialmente a nível residencial), principalmente na Europa. Os produtos da divisão são vendidos sob marcas internacionais e regionais, como *Bosch*, *Buderus*, *Worcester* e *Junkers*. O portfólio de produtos inclui desde aquecedores,

bombas de calor, sistemas solares térmicos e caldeiras de combustível sólido, até processos de cogeração e caldeiras industriais [2] [3].

Tendo em vista funcionalidades como a monitorização e diagnóstico remotos, os dispositivos preparados para se ligarem à internet tornam-se cada vez mais importantes. É aqui que são desenvolvidas soluções tecnológicas na área da termotecnologia focadas na conectividade, eficiência energética e em casas inteligentes e sustentáveis. Entre as inovações criadas no Centro de Competências estão o esquentador conectado *Geos* e vários tipos de *software* para plataformas Web e *mobile* de várias áreas de negócio da *Bosch*.

Na visita a esta unidade, assim como em Braga, foi possível observar como todas as linhas de montagem estão ligadas. Os dados recolhidos são, depois, analisados para dar «importantes *insights*» que ajudam a «melhorar a eficiência e ajudar os gestores na tomada de decisão». Em Aveiro, a empresa usa ainda cinco robôs autónomos AGV (*Automated Guided Vehicle*) que transportam produtos terminados para o armazém e trazem componentes para as linhas de montagem. «A Indústria 4.0 é já uma realidade nas nossas fábricas», referiu Carlos Ribas, que enalteceu ainda o facto de os dados serem vitais para avaliar a qualidade de tudo o que é produzido pela *Bosch* [5].

Capítulo 3. A Internet das Coisas (IoT)

O conceito IoT, do inglês *Internet of Things*, assenta, de uma forma mais ampla, numa ideia de simples explicação, que passa por ligar todas as coisas no mundo à internet. Dito assim, poderá parecer algo absurdo (dada a complexidade intrínseca dessa tarefa e opiniões menos consensuais). Mas, na verdade, é uma ideia que se tem vindo a expandir cada vez mais e que hoje em dia conta já com diversos aparelhos que se podem conectar entre si através de uma rede, para partilhar informação, criando oportunidades para mais integrações do mundo físico a sistemas computadorizados, com vista a optimizar várias tarefas do dia-a-dia, resultando em melhorias de eficiência, benefícios económicos e reduzindo esforço humano.

De uma forma geral, tudo o que se liga conta com um ou mais sensores e/ou actuadores. Assim, pode-se pensar em 3 categorias:

- Coisas que reúnem informação e que a enviam posteriormente (sensores);
- Coisas que recebem informação e que agem com base nela (actuadores);
- Coisas que fazem ambas as tarefas anteriores.

Assim como os seres humanos possuem vários sentidos para perceber o mundo em redor, as máquinas contam com os sensores para perceber o que se passa à sua volta, como por exemplo sensores de temperatura, humidade, luminosidade, entre muitos outros. Os sensores são fundamentais para recolher informações sobre uma variedade de dados, e quando conectados a uma rede local ou à internet, estes podem transmitir essa mesma informação com a finalidade de ser interpretada e de conduzir a decisões mais inteligentes.

Essas decisões tornam-se em acções por meio dos chamados actuadores. Quando conectados a uma rede local ou à internet, estes recebem a informação e actuam com base nela, como por exemplo um portão automático que abre ou fecha com base na instrução do utilizador, ou uma impressora que imprime documentos quando comandada para tal.

Existem, também, sistemas mais complexos compostos por ambas as partes, sensores e actuadores, e que podem estar ainda conectados à rede para permitir monitorização/controlo ou o envio de comandos à distância. Um exemplo muito comum utilizado na agricultura passa por sistemas de irrigação automática, onde a rega se vai ligar consoante a informação enviada pelos sensores de humidade, mas como este exemplo, existem inúmeros outros casos [6].

3.1. Que problemas podem advir com IoT?

Apesar das inúmeras vantagens que a Internet das Coisas pode trazer, existem também alguns aspectos menos positivos a ter em consideração.

Um dos desafios para os programadores de aplicativos IoT passa por filtrar, processar e interpretar a vasta quantidade de dados que são recolhidos pelos sensores, e depois disso, o armazenamento dessa grande quantidade de dados. Dependendo da aplicação, poderá ser necessária uma elevada quantia de dados adquiridos, o que implicaria a existência de grandes espaços de armazenamento. Note-se que para armazenar grandes quantidades de informação são necessários centros de processamento de dados (também conhecidos por CPDs ou *data centers*), instalados em espaços adequados e devidamente protegidos por sistemas de segurança. Estes centros de processamento de dados são alimentados pelas concessionárias locais de energia, geradores de energia de grande capacidade e fontes de alimentação ininterruptas (UPS). Estes *data centers* consomem até cem vezes mais energia do que um escritório comum. Sabendo que a internet consiste em incontáveis centros de armazenamento de dados que contêm toda a informação *online*, estima-se que a internet, actualmente, seja já responsável por cerca de 5% do total de energia gerada (consumida) [7].

A nível de segurança, foram levantadas várias preocupações em torno do rápido desenvolvimento da IoT, sem que fossem tomadas, ou sequer consideradas, as devidas medidas de segurança e mudanças de regulamento que poderão ser necessárias. A maioria dos problemas de segurança técnicos assemelham-se aos de servidores convencionais, computadores e telemóveis, pois as *firewalls*, actualizações de segurança e sistemas *anti-malware* usados para estes são, geralmente, inapropriados para os muito mais pequenos, e

menos capazes, dispositivos de IoT. Para além disso, as correções dos defeitos encontrados nos sistemas, raramente chegam aos utilizadores de aparelhos mais velhos e mais baratos.

Outro problema passa pela privacidade, autonomia e controlo dos utilizadores. A IoT oferece um imenso potencial para dar mais poder aos cidadãos, tornando os governos transparentes e expandindo o acesso à informação (como o caso da Estónia que aplicou IoT no seu país a nível governamental [8]). No entanto, as ameaças à privacidade são enormes, bem como o potencial para controlo da sociedade e manipulação política. Há quem expresse preocupação, afirmando que estão prestes a perder a sua privacidade. Outros demonstram preocupações relativamente à habilidade da IoT extinguir o controlo das pessoas sobre as suas vidas.

Finalmente, aquele que será talvez o maior desafio actual, diz respeito à fragmentação de plataformas e à falta de normas técnicas. Isto deve-se ao facto de os sensores e actuadores da Internet das Coisas usarem um vasto leque de diferentes tecnologias para recolher informação, comunicar entre si, e transmitir dados para computadores que armazenam e processam os dados. Os diferentes dispositivos, plataformas, protocolos e portas de comunicação tornam difícil desenvolver aplicações que funcionem de forma consistente em diferentes ambientes inconsistentes, e leva a problemas de desempenho, fiabilidade, interoperacionabilidade e escalabilidade [9] [10].

É aqui que a garantia de qualidade de *software*, em particular a área de testes, marca a diferença.

3.2. IoT na Bosch

A *Bosch* tem apostado bastante na área de IoT, e na da inteligência artificial. Com cerca de 10 anos de envolvimento em actividades de IoT, tem definido como objectivo tornar-se numa das líderes mundiais nesta ideologia. Numa fase inicial, criaram a *Bosch IoT Suite*, sistema que continuam a expandir, e que tem registados todos os aparelhos com ligação à internet. Como tal, tem vindo a trabalhar no sentido de tornar todos os produtos *Bosch* passíveis de se ligarem à internet e vai manter esta tendência nos próximos anos. Em

2016, lançou a *Bosch IoT Cloud*, que para além de fornecer a infraestrutura técnica, inclui plataformas e *software* para as áreas de mobilidade, indústria conectada, e construção.

Para além disso, foi criado um centro de competências em inteligência artificial (IA) e *machine learning* em 2017, sediado no Centro de Inteligência Artificial da *Bosch* (em Tübingen, na Alemanha), e já está planeado, com a ajuda desta, criar novas áreas de negócio em torno de sensores para análise de dados, deixando como exemplo a agricultura conectada, que pretende reduzir a necessidade de pesticidas, desperdício de recursos e minimizar o impacto ambiental, etc.

Na área da mobilidade, a *Bosch* planeia também conectar veículos à rede, com vista a desenvolver a condução autónoma, e têm já protótipos de estacionamento autónomo [3].

No que diz respeito à Termotecnologia, equipamentos com a capacidade de se ligarem à rede são já uma realidade, e é para estes que o Centro de Engenharia desenvolve por exemplo as interfaces entre homem e máquina, e respectivo *software*.

Quando se fala em soluções de água quente residenciais, pensa-se essencialmente em equipamentos que providenciam aquecimento ambiente (climatização) ou aquecimento de água sanitária (para torneiras e chuveiros) numa habitação. Estes serviços podem ser fornecidos separadamente por equipamentos diferentes, sendo o esquentador um dos mais comuns quando se pretende apenas aquecimento de águas sanitárias, pois contam com apenas um circuito (permutador de calor) de água que consiste numa serpentina de cobre colocada sobre uma chama. A água fria entra por um lado da serpentina e sai quente do outro lado. A título de exemplo, na *Bosch* Termotecnologia em Aveiro foi desenvolvido e é fabricado um dos mais avançados esquentadores do mundo, o *Bosch Greentherm 9000* (Figura 1), que, com uma potência de 80 CV e a atingir uma eficiência na ordem dos 96% UEF (*Uniform Energy Factor*), conta com versões dotadas de avançadas funcionalidades de conectividade permitindo total controlo quer ao instalador, quer ao utilizador [11].

Figura 1 – **Esquentador *Bosch Greentherm 9000*** *[11].*

No entanto, é possível combinar as duas funções, ou seja, com o mesmo sistema pode-se aquecer a casa e ter água quente para o duche ou para tarefas na cozinha. A estes sistemas, habitualmente dá-se o nome de aquecimento central. O aquecimento central conjuga o máximo conforto com a máxima economia, sendo a solução mais racional do ponto de vista energético. Regra geral, estes sistemas são constituídos por:

- Uma caldeira destinada ao aquecimento de águas;
- Uma rede de tubos para circulação da água;
- Radiadores e toalheiros destinados ao aquecimento do ar ambiente;
- Depósitos de acumulação de água quente (opcional);
- Reguladores e programadores (controladores).

A peça principal nestes sistemas é a caldeira. À semelhança dos esquentadores, as caldeiras também têm um circuito de água para aquecimento das águas sanitárias. No entanto, estas contam com uma segunda serpentina para a água do aquecimento ambiente que corre em circuito fechado (a água que sai da caldeira para os radiadores volta a entrar na caldeira) e esta circulação é feita com recurso a uma bomba eléctrica. Os queimadores podem ser alimentados por diferentes tipos de combustíveis (gás natural, propano ou butano, gasóleo, lenha ou *pellets*) sendo o gás o mais comum. A nível de extração de fumos, existem

caldeiras atmosféricas e de estanque. As atmosféricas fazem a admissão do ar necessário para a combustão a partir do local onde está instalada e extrai os gases queimados para o exterior através de uma chaminé convencional (pode ser instalada em locais que tenham uma boa renovação do ar). As caldeiras de estanque, vão buscar o ar necessário para a combustão ao exterior da habitação e expulsa os gases queimados para o exterior através de uma chaminé de tubos coaxiais (a combustão dá-se numa câmara hermética, não correndo risco do oxigénio do local onde está instalada a caldeira, ser abusivamente retirado ou consumido. A exaustão dos gases é auxiliada por um ventilador). Existem também caldeiras de condensação, numa tentativa de aumentar a eficiência do equipamento, que ao contrário das caldeiras convencionais, que permitem que os gases e o vapor de água gerados na combustão saiam livremente para a atmosfera, aproveitam o seu calor transmitindo-o aos circuitos de água quente [12] [13].

Os controlos da caldeira fazem uma grande diferença num sistema de aquecimento central. Com as caldeiras a operar já no seu pico de eficiência, os controlos oferecem uma forma adicional de melhorar o desempenho geral do sistema de aquecimento, permitindo economizar na factura. Existem essencialmente duas formas de controlar o aquecimento em nossa casa. A primeira, e provavelmente a mais popular, são os termóstatos, que monitorizam a temperatura dentro de casa e a vão ajustar ao valor definido pelo utilizador. A segunda forma é recorrendo a temporizadores programáveis que permitem ao utilizador ligar ou desligar o aquecimento a determinados momentos do dia (oferecem a possibilidade de programar rotinas diárias ou semanais com diferentes tempos de funcionamento). Estes controlos são colocados ao dispor sob a forma dos tradicionais controlos mecânicos ou sob a forma de controlos digitais programáveis. Uma forma complementar de controlar o aquecimento ambiente são as válvulas colocadas em cada um dos radiadores espalhados pela habitação, que permitem ajustar diferentes temperaturas para diferentes divisões.

Muitas vezes as caldeiras estão em locais pouco cómodos de acesso, como sótãos, caves ou garagens e por isso foram disponibilizados termóstatos programáveis de interior para colocar em qualquer divisão da casa e permitir um fácil acesso aos controlos da caldeira. E de forma a estender essa comodidade foi conferida a habilidade de se ligar à internet, tornado estes equipamentos inteligentes. Assim, todas as funcionalidades podem estar ao alcance de um toque no *smartphone* (Figura 2), através dos (crono)termóstatos programáveis inteligentes e de aplicações de *smartphone* ou *tablet* dedicadas a esse fim.

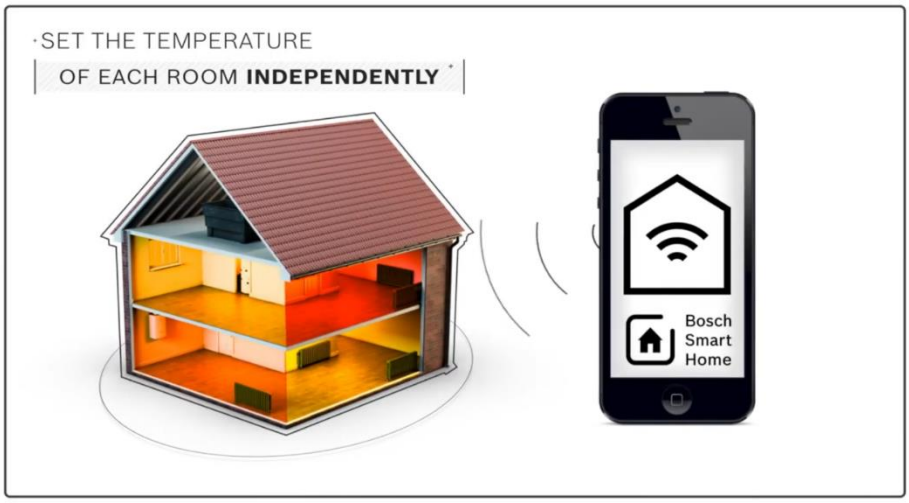

Figura 2 – **Ilustração do Controlo Remoto do Aquecimento Central através de um *Smartphone*** *[14]*.

Um excelente exemplo disso, é o *Bosch EasyControl* (Figura 3), cujo software é desenvolvido no Centro de Engenharia da *Bosch* Termotecnologia de Aveiro [15].

Figura 3 – **Cronotermóstato Inteligente *Bosch EasyControl*** *[15]* & **Exemplo da Aplicação *EasyControl* na Programação de uma Rotina** *[16]*.

Este equipamento permite controlar e ajustar o aquecimento remotamente, vem provido de um sistema de deteção de presença, faz compensação do aquecimento em função da temperatura exterior, mantém um histórico de consumos e permite controlar o aquecimento individualmente em diferentes divisões da casa (com a adição de válvulas

termostáticas de radiador inteligentes (Figura 4 - *smart TRVs – thermostatic radiator valves*)).

Figura 4 – **Válvula Termostática de Radiador Inteligente (*Smart TRV*)** *[15].*

Capítulo 4. Desenvolvimento de *Software*

Como já é do conhecimento do público em geral, ao longo das duas últimas décadas, a indústria do *software* tem vindo a crescer imenso, a tornar-se mais competitiva e a angariar mais utilizadores. O *software* é hoje um componente essencial de aplicações mais exigentes, tais como aviões, naves espaciais ou sistemas de controlo para tráfego aéreo, bem como em aplicações mundanas como relógios, fornos, automóveis, leitores de DVD, portões eléctricos, telemóveis ou controlos remotos [17]. Os dispositivos orientados para a Internet das Coisas não fogem à regra, e para estes, é também necessário desenvolver *software* (inclusive *firmware*).

Como tal, o desenvolvimento de *software* é hoje uma área de engenharia, que de uma forma geral, segue uma ordem de ideias em comum, com pequenas variações de programador para programador. A esta ordem de ideias, dá-se o nome de Ciclo de Vida do Desenvolvimento de *Software* (em inglês, *Software Development Life Cycle*, SDLC).

O SDLC é uma *framework* que engloba a descrição da sequência de fases pelas quais um determinado produto/*software* tem de passar desde o seu projecto inicial até à sua implementação e entrega final [18].

4.1. Fases Gerais do Desenvolvimento de *Software*

Figura 5 – **Ciclo de Vida do Desenvolvimento de *Software*** *[19].*

O Ciclo de Vida do Desenvolvimento de *Software* (Figura 5) é um processo usada na indústria do *software* para projectar, desenvolver e testar os produtos. Tem, essencialmente, como objectivo produzir *software* de alta qualidade que vai ao encontro das expectativas dos clientes, dentro das estimativas de custos e tempo estipuladas. Abaixo será apresentada uma breve descrição de cada uma dessas fases [20].

Etapa 1: Planeamento e Análise de Requisitos

O Planeamento e Análise de Requisitos é uma etapa fundamental e talvez a mais importante deste processo. Aqui são reunidas ideias e informações da parte do cliente, do departamento de vendas e de especialistas no domínio da indústria, que são posteriormente usadas para planear uma abordagem base ao projecto e para conduzir ao estudo da viabilidade económica, operacional e técnica do produto.

O planeamento dos requisitos para garantia de qualidade e a identificação dos riscos associados ao projecto também surgem nesta fase. Os resultados do estudo da viabilidade técnica do produto irão ajudar a definir as abordagens técnicas que podem ser seguidas para implementar o projecto com sucesso e com o mínimo de risco.

Etapa 2: Definição dos Requisitos

Assim que a etapa anterior termina, o próximo passo consiste em definir claramente, e documentar, os requisitos do projecto e submetê-los à aprovação do cliente e dos analistas de mercado. Isto é feito através de um documento (SRS, *Software Requirement Specification*) que consiste em todos os requisitos do produto a serem implementados durante o ciclo de vida do projecto.

Etapa 3: Projecto e Arquitectura do Produto

O SRS é a referência para os arquitectos do projecto perceberem qual a melhor arquitectura a desenvolver para o produto. Com base nos requisitos especificados no SRS, habitualmente mais do que uma abordagem para a arquitectura do produto é proposta e são documentadas num documento chamado de DDS, *Design Document Specification*.

Este documento é revisto por todas as partes envolvidas e com base em vários parâmetros como avaliação do risco, robustez do produto, orçamentos e restrições de tempo, a melhor abordagem é escolhida.

Essa abordagem deve definir claramente todos os módulos da arquitectura do produto, bem como representações das comunicações e fluxo de dados entre o produto e qualquer módulo externo existente. O projecto interno de todos os módulos da arquitectura proposta devem estar claramente definidos o mais detalhadamente possível no DDS.

Etapa 4: Desenvolvimento do Produto

Nesta etapa, começa o verdadeiro desenvolvimento e o produto é construído. O código fonte é gerado de acordo com o DDS. Se o projecto foi feito de uma forma detalhada e organizada, então a escrita do código não deverá ser muito complicada.

Os programadores devem seguir as directrizes definidas pela organização e são usadas ferramentas para a escrita desse código, tais como editores de texto, compiladores, interpretadores, depuradores, entre outros (normalmente todas estas ferramentas vêm incluídas nos Ambientes de Desenvolvimento Integrados, do inglês *Integrated Development Environment*, IDE). Diferentes linguagens de programação, como C, C++, Pascal, Java e PHP são usadas. A escolha de qual a linguagem de programação a utilizar normalmente é decidida de acordo com o tipo de *software* a ser desenvolvido.

Etapa 5: Testes ao Produto

Esta etapa acaba por ser um anexo em todas as fases mencionadas, pois actualmente, as actividades de testes estão geralmente, envolvidas em todo o ciclo de vida do desenvolvimento de *software*. No entanto, esta etapa refere-se unicamente à fase de testar o produto/*software* final, onde os defeitos do produto que sejam encontrados, são registados, corrigidos e testados novamente, até que o produto esteja de acordo com os padrões de qualidade definidos no SRS.

Etapa 6: Lançamento no Mercado e Manutenção

Uma vez concluídos os testes ao produto, e chegada à conclusão de que este está pronto, é então formalmente lançado no mercado apropriado. O produto pode ser, numa primeira fase, lançado de forma limitada com o intuito de ser testado num ambiente real (UAT, *User Acceptance Testing*, os testes de aceitação do utilizador). Depois, com base nas opiniões, o produto pode ser lançado realmente ou podem ser aplicadas as melhorias sugeridas. Depois do lançamento, é necessário efectuar a sua manutenção, em prol da base de utilizadores existente.

4.2. SQA (Garantia de Qualidade de *Software*)

SQA, terminologia do termo *Software Quality Assurance*, em português Garantia de Qualidade de *Software.* De acordo com o glossário do IEEE o SQA define-se em primeiro lugar por um padrão sistemático e planeado de todas as acções necessárias para garantir a confiança num produto de que este cumpre com todas os requisitos técnicos e também pelo conjunto de actividades concebidas para avaliar os processos pelos quais os produtos são desenvolvidos ou fabricados (em contraste com o controlo de qualidade) [21]. Já Galin defende que esta definição peca por não ter em conta os aspectos temporais e orçamentais

das operações que serão definidos pelos requisitos de gestão, alargando então a definição acima para conter estes mesmos aspectos [21].

É portanto um tema de extrema importância nas empresas tecnológicas e embora esta importância seja amplamente referida na prática as coisas ainda não são perfeitas continuando por vezes a persistir a percepção de que é mais importante cumprir prazos do que solucionar problemas existentes [22], o que leva muitas vezes as equipas de desenvolvimento a "atropelarem" um pouco o tema da qualidade.

Com esta insistência no tema da qualidade será também importante esclarecer o que é que será um *software* de qualidade. De acordo com a definição de Pressman's citada por Galin, um *software* de qualidade é dado pelo grau de conformidade perante os requisitos funcionais definidos, pelos *standards* de especificados para a qualidade do *software* e pela aplicação de boas práticas de engenharia de *software* (GSEP[3]) [21]. É também expectável que a esperança de vida útil do *software* atinja as expectativas para satisfazer os seus requisitos. Para este efeito dever-se-á ter em conta vários atributos, intrínsecos ao *software* ou aos sistemas a que se destina. Atributos estes como: segurança, escalabilidade, previsibilidade, durabilidade, interoperacionabilidade, portabilidade, reputação, etc [23].

A nível de reputação, tem-se um exemplo dado numa das formações, que diz que 1 cliente satisfeito influencia positivamente em média 8 pessoas, e 1 cliente insatisfeito influencia em média 25 pessoas. Claro que ainda assim o balanço seria positivo, mas aos olhos do público, estes produtos defeituosos teriam um grande impacto na imagem e nos lucros da empresa.

Para ilustrar melhor, imaginem-se 3000 aparelhos fabricados por dia, com uma taxa de produtos sem defeito de 99,9%. Pode parecer um valor muito positivo, no entanto, significa que estão 3 produtos defeituosos a serem entregues por dia ao cliente, que no final de uma semana serão 15, ao final de um mês são 60 e ao final do ano são 720 aparelhos com defeito. Isso resultaria em aproximadamente 18 000 pessoas influenciadas negativamente pela marca ao final de um ano.

Por isso mesmo, a *Bosch* é uma empresa que leva este tema muito a sério e percebe a importância das actividades de garantia de qualidade nos vários ciclos de desenvolvimento

[3] GSEP: *Global Scientists and Engineers Program*

de um produto, fazendo um esforço para estudar as várias soluções existentes no mercado e tentar minimizar o número de incompatibilidades entre departamentos e ferramentas com vista a maximizar a eficiência e qualidade dos seus produtos.

4.3. *Quality Gates* (Critérios de Aceitação)

Como já se viu numa das secções anteriores, os projectos repartem-se em diferentes fases. Geralmente estas consistem na fase de planeamento, desenvolvimento, testes, lançamento e manutenção. Cada uma destas fases, está sujeita aos chamados *Quality Gates* que são uma espécie de *checkpoints* a atingir antes de se avançar para a próxima etapa (Figura 6). Habitualmente a verificação destes *Quality Gates* é feita com recurso a uma lista de pontos a cumprir onde são colocadas questões relativas à fase do projecto em avaliação. O elemento da equipa responsável pela garantia de qualidade deve então confirmar se todas as ações necessárias ao cumprimento desses pontos foram levadas a cabo ou não, e geralmente avalia o cumprimento destes pontos com recurso a 3 cores, verde se estiver tudo em ordem, amarelo se houverem tópicos em aberto ou em falta mas que não representem risco, e vermelho nos piores cenários, que podem obrigar à paragem do projecto até que todas as medidas necessárias sejam tomadas [24].

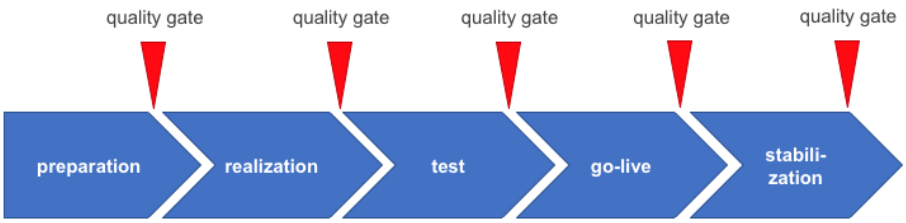

Figura 6 – **Fluxo Geral de um Projecto submetido a *Quality Gates*** *[24].*

4.4. *Definition of Done* (Definição de Pronto)

Antes de se começar a desenvolver um produto (ou *software*) é necessário decidir o que 'Pronto' significa para esse produto e para a empresa responsável. Uma boa definição

de 'Pronto' (muitas vezes abreviada para DoD, do inglês *Definition of Done*), inclui tudo aquilo que uma empresa tem de fazer para entregar o produto finalizado aos clientes, ou seja, uma lista acordada de critérios que o produto deverá cumprir para cada item do conjunto de tarefas necessárias ao seu desenvolvimento (no caso de um *software* esse conjunto de tarefas chama-se *backlog*). Dependendo do tipo de produto essa definição pode ser feita de forma muito diferente. Por exemplo, no caso de um produto que possa servir de suporte vital a um paciente (como um *pacemaker*) essa definição deve ser feita de forma muito rigorosa, enquanto que no caso de um portal de vendas *online*, essa definição pode ser menos rigorosa [25].

Não se deve confundir a definição de 'Pronto' com os critérios de aceitação (*quality gates*) referidos na secção anterior, que tal como foi referido, consistem nas condições específicas que um item individual tem de cumprir para ser aceite. A definição de 'Pronto' aplica-se uniformemente a todos os itens do *backlog* de um produto.

Seguem-se algumas características fundamentais que deve ter uma definição de 'Pronto':

- Uma lista de verificações mensurável e relativamente curta: deve-se almejar ter coisas mensuráveis na definição de 'Pronto', das quais se possa testar o resultado, preferencialmente, de forma automática;
- Uma versão preparada para lançamento: ainda que a equipa não tenha lançado o seu produto, essa opção deve ser disponibilizada. O proprietário do produto (*product owner*) deve ter a possibilidade de decidir lançar o produto caso este seja do seu agrado. Idealmente, devem ter um processo totalmente automatizado para entrega de *software*;
- Sem mais tarefas: não deverão existir tarefas restantes para a equipa de desenvolvimento. Qualquer trabalho adicional significa que não estão prontos, e vai ser necessário utilizar mais recursos do proprietário do produto na próxima iteração.

A primeira definição de 'Pronto' não aparece do nada e fazer com que um *software* esteja em conformidade com uma boa definição de 'Pronto' não é tarefa fácil. Essa definição

deve crescer gradualmente, no entanto, para isso acontecer, deve ser criado algo por onde começar.

Para chegar a essa primeira definição é recomendável realizar uma reunião de refinamento inicial do *backlog* do *software* com toda a equipa e, de preferência, com especialistas de várias áreas (código, testes, segurança, UX (*user experience*), UI (*user interface*), arquitectura, etc) a fim de tentar perceber tudo o que será necessário implementar.

Alguns exemplos que podem constar numa primeira definição de 'Pronto' são:

- O produto passa as verificações da análise estática ao código sem muitos erros críticos (começando por exemplo em algo que não ultrapasse os 50 erros críticos e ir elevando a fasquia ao longo do tempo);
- A percentagem de código abrangido por testes mantém-se igual à anterior ou sobe (podendo começar nos 80% e ir subindo ao longo do tempo);
- O produto vai de encontro aos padrões do desenvolvimento de *software*: devem adoptar-se normas e boas práticas como nomear métodos, testes, variáveis e colocar toda essa informação em documentação numa *wiki* e ir melhorando ao longo do tempo;
- Os testes de aceitação ao produto são automatizados: é altamente recomendável automatizar os testes ao *software*, para evitar que ao implementar novo código, se introduzam problemas ao código anterior;
- Os testes à segurança do produto passaram sem graves vulnerabilidades detectadas: deve usar-se uma ferramenta automatizada para verificar possíveis vulnerabilidades no *software*. Nem todas serão detectadas, mas pelo menos garante-se um mínimo exigível;
- Etc.

Qualquer que seja a definição de 'Pronto' acordada inicialmente, é improvável que o produto nesse momento cumpra com todos os critérios. Portanto, é importante garantir que todos os critérios são cumpridos pela versão do *software* actualmente em desenvolvimento (produto viável mínimo). Só depois se deve começar a fazê-lo de forma regular com tempos de desenvolvimento definidos e datas de entrega acordadas (os chamados *Sprints*).

É importante melhorar continuamente a qualidade, e isso significa que a equipa deve reflectir sobre a sua definição de 'Pronto' numa base regular, idealmente a cada *Sprint*, até atingir uma definição perfeita. Normalmente essa definição é implementada via uma *pipeline* (um agregado de tarefas) que executa todos os processos de integração e entrega contínuas de forma automática [25].

4.5. Integração, Entrega e Passagem à Produção Contínuas (CI/CD)

4.5.1. *Continuous Integration* (Integração Contínua)

Inicialmente, no modelo mais 'tradicional' de desenvolvimento, os programadores de *software* introduziam alterações ao código de uma forma esporádica, ou seja, pouco frequente. Isto significava que iam acumulando alterações no código, e quando chegava o momento de combinar o código modificado com o código base através dos *merges*, tinham já uma quantidade significativa de alterações que geralmente abria uma porta para o surgimento de inúmeros conflitos. Conflitos estes, que teriam de ser resolvidos antes de avançar, para garantir a estabilidade e a integridade do *software*. Isto é conhecido como o 'inferno da integração' e tem uma perniciosa tendência para ocorrer nas fases de lançamento de uma nova versão do *software*, levando inevitavelmente a atrasos.

A integração contínua surgiu precisamente para evitar isso. Programadores que praticam esta modalidade, fazem o *merge* das suas alterações com o código base o mais frequentemente possível, respeitando simultaneamente uma hierarquia definida de ramos de desenvolvimento. Assim essas alterações são validadas imediatamente ao criar uma *build*, por via de testes automáticos, concebidos para testar a integridade desse trecho de código/funcionalidade (Figura 7). Desta forma, os programadores podem detectar e corrigir os erros na hora, evitando assim um efeito de 'bola de neve' que esses erros iriam originar se fizessem a integração de forma menos frequente. Em última análise, isto traduz-se num aceleramento do processo de desenvolvimento e da entrega.

Para adoptar um modelo de integração contínua é necessário acrescentar algumas coisas ao processo de desenvolvimento, como por exemplo, implementar conjuntos de testes

em código e preparar um servidor de automação (do qual é exemplo, o *software* Jenkins) para executar esses testes após cada alteração introduzida. Naturalmente, numa primeira fase, de estruturação dos testes, isto vai, aparentemente, tornar todo o processo mais demorado dado o tempo despendido para preparar as novas funcionalidades, contudo, a longo prazo, as vantagens são por demais evidentes tornando todo o processo mais simples e eficiente.

Em suma, a integração contínua coloca grande ênfase na automatização de testes para verificar que uma aplicação não deixou de funcionar quando novos *commits* (alterações e introdução de novo código) são integrados no ramo principal [26] [27].

4.5.2. *Continuous Delivery* (Entrega Contínua)

A entrega contínua é uma extensão da integração contínua para garantir que se podem entregar novas alterações rapidamente e de uma forma sustentável. Isto significa que para além de ter testes automatizados, também há uma automatização do processo de lançamento e o *software* pode ser colocado em execução a qualquer momento com um simples clique se assim entenderem (Figura 7).

Em teoria, com a entrega contínua, é possível decidir entregar *software* diariamente, semanalmente, quinzenalmente, ou o que mais se adequar aos requisitos do negócio. No entanto, para tirar o máximo proveito da entrega contínua, e apesar da pré-garantia de integridade com os testes automatizados, deve colocar-se o *software* em produção (*deploy*) o mais cedo possível por forma a mitigar o impacto que poderão ter eventuais problemas, permitindo que estes sejam solucionados de forma mais rápida [26] [27].

4.5.3. *Continuous Deployment* (Passagem à Produção Contínua)

A passagem à produção contínua (*deployment*) vai um passo mais à frente que a entrega contínua. Com esta prática, todas as alterações que concluam com sucesso todas as etapas de uma *pipeline* (cadeia de eventos) de produção são lançadas para um ambiente de

produção de forma automática onde as restantes fases do ciclo de testes podem ser levadas a cabo (Figura 7). Diz-se de forma automática pois não há intervenção humana (a nível de acção e decisão) neste passo em concreto, e apenas um teste falhado ou qualquer erro na *pipeline* vai prevenir que uma nova alteração com falhas seja colocada em execução na produção.

Esta é uma excelente forma de acelerar o ciclo de *feedback* dos clientes e retirar pressão da equipa uma vez que se anula o tradicional prazo/*deadline* do dia do lançamento do *software*. Os programadores podem focar-se no desenvolvimento do *software*, e ver o seu trabalho entrar em acção minutos depois de terem terminado, com a segurança adicional que este controlo de qualidade garante [26] [27].

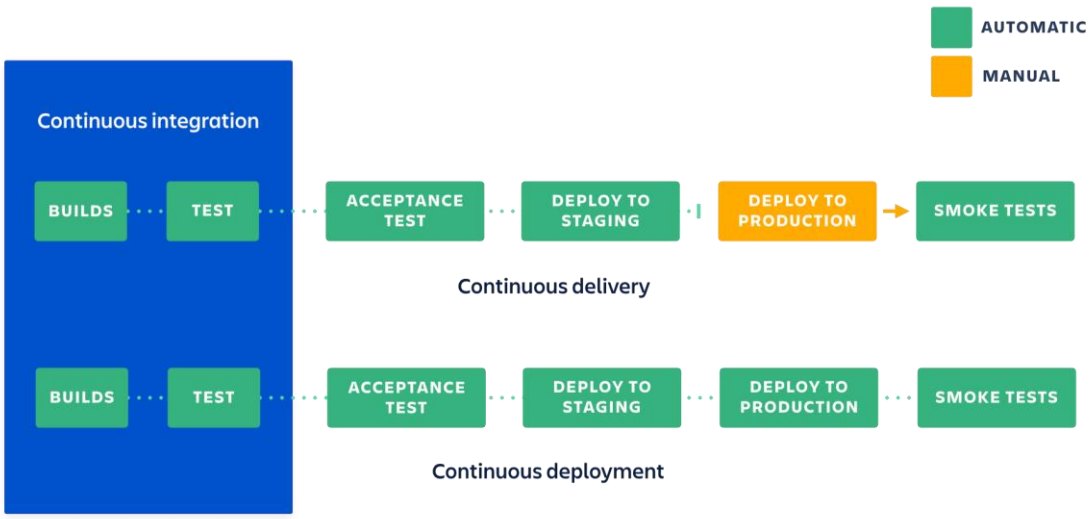

Figura 7 – **Ilustração do envolvimento entre Integração, Entrega e Passagem à Produção Contínuas** *[26].*

4.6. REST API

As REST APIs de vários *softwares* foram muito utilizadas ao longo do trabalho desenvolvido no estágio, com vista à automatização dos processos, recorrendo à ferramenta *Jenkins*.

A fim de perceber o que é uma REST API, é primeiro necessário perceber alguns conceitos base.

4.6.1. API (Interface de Programação de Aplicações)

Uma API (*Application Programming Interface*) é, simplificando, uma forma de permitir que componentes de *software* comuniquem entre si. Ou se for preferível, um conjunto de rotinas e padrões documentados que permitem a serviços externos aceder e obter recursos de uma determinada aplicação, sem necessitar de conhecer os seus detalhes.

Figura 8 – **Representação gráfica da definição ou função de uma API** *[28].*

Por exemplo, em linguagem orientada a objectos (como Java), os objectos contêm métodos e atributos, que podem ser considerados uma API para esses mesmos objectos. Para comunicar ou utilizar uma funcionalidade de um determinado objecto, isto é feito através da sua API, ou seja, neste caso, através dos seus métodos e atributos que são definidos como acessíveis externamente permitindo a interacção. É ao conjunto desses métodos e atributos que se dá o nome API. Portanto, uma API poderia ser qualquer coisa sob qualquer formato, desde que seja uma forma de comunicação entre componentes de *software* (Figura 8). Um exemplo mais familiar de uma API para o utilizador comum, é a forma como as pessoas interagem com o seu sistema operativo através da linha de comandos, para desligar, reiniciar ou qualquer outra funcionalidade disponibilizada e permitida pelo sistema. A forma como uma API é implementada e em que consiste não é importante, apenas necessita de fornecer uma forma de comunicação para ser designada API.

No contexto da *web*, as APIs podem assumir a designação de Serviços *Web* (*Web Services*). Isto porque são um conjunto de regras e tecnologias que permitem que dois ou mais componentes comuniquem entre si através da *web*.

4.6.2. REST (Transferência de Estado Representativa)

Este conceito foi sugerido por Roy Fielding, que é também um dos autores do protocolo de comunicação HTTP. Segundo Roy Fielding, o autor do conceito de REST (*Representational State Transfer*):

«Durante o processo de uniformização do HTTP, fui chamado a defender as escolhas de design da *web*. Isto é algo extremamente difícil de fazer num processo que aceita propostas de qualquer pessoa num tópico que estava rapidamente a tornar-se no centro de toda a indústria. Tinha comentários de mais de 500 programadores, muitos dos quais engenheiros distinguidos com décadas de experiência, e eu tive de explicar desde as noções mais abstractas das interações da web aos detalhes mais sucintos da sintaxe HTTP. Este processo resumiu o meu modelo a um conjunto chave de princípios, propriedades, e restrições que são agora chamados de REST.» [29]

A definição que se pode retirar das suas palavras, é que REST é uma arquitectura/especificação que dita como sistemas distribuídos na internet devem comunicar entre si. Posto de forma simples, REST é uma forma de comunicação entre componentes na web.

A REST não terá sido algo criado à parte, ou seja, é algo tão antigo quanto a própria *web*. Representa apenas a forma como as comunicações na *web* devem funcionar. Roy Fielding adicionou alguns elementos e enalteceu as boas práticas, mas a REST em si, é a arquitectura subjacente da *web*. Poderá, eventualmente, não ser a melhor arquitectura, mas é certamente melhor do que fazer as coisas arbitrariamente.

O significado de Transferência de Estados Representativa

Denomina-se de representativa porque há uma transferência de representações de recursos. Os recursos podem ser qualquer coisa passível de ser nomeada na internet, como um utilizador, uma lista de utilizadores, uma fotografia, uma lista de fotografias, comentários, publicações, artigos, páginas, vídeos, livros, perfis, entre outros. Pode-se pensar na internet como um enorme conjunto de recursos. Para perceber exactamente como se obtém essa representação de recursos, basta olhar para o modo como as pessoas interagem com páginas *web* em modelos cliente-servidor e com recurso ao protocolo de comunicação HTTP. O cliente faz o pedido colocando um URL numa aplicação (geralmente *browser*) e o servidor envia de volta a resposta com um recurso sob a forma de página web ou qualquer outro formato em que este esteja. Mas na verdade, o servidor não está a enviar o verdadeiro recurso, e sim uma representação do mesmo. Portanto pode-se pensar num recurso com algo físico localizado na base de dados de um servidor e representações desses recursos como sendo cópias legíveis para um humano, sob a forma de HTML, de uma imagem (JPEG) ou formatos fáceis de trabalhar para programadores, como JSON ou XML (Figura 9).

XML

```
<empresa>
   <empregados>
      <empregado>
         <nome>Helena</nome>
         <idade>35</idade>
      </empregado>
      <empregado>
         <nome>Isabel</nome>
         <idade>42</idade>
      </empregado>
      <empregado>
         <nome>Manuel</nome>
         <idade>39</idade>
      </empregado>
   </empregados>
</empresa>
```

JSON

```
{
  "empregados": [
    {
      "nome": "Helena",
      "idade": "35"
    },
    {
      "nome": "Isabel",
      "idade": "42"
    },
    {
      "nome": "Manuel",
      "idade": "39"
    }
  ]
}
```

Figura 9 – **Comparação entre as estruturas de um ficheiro XML e de um ficheiro JSON**.

O estado refere-se ao estado de uma aplicação, quando se clica em algum menu, o estado da aplicação muda e o utilizador é confrontado com novos recursos, e a cada clique,

o estado muda e volta a mudar. Na resposta pode-se obter um recurso, no entanto, também se pode receber o estado de uma aplicação.

Transferência implica simplesmente uma troca de informações entre cliente e servidor, através de um pedido e de uma resposta.

A arquitectura REST segue 6 restrições chave [30]:

- **Interface Uniforme**: neste caso feita com recurso aos verbos e respostas (estados e corpo da mensagem) do HTTP e URIs dos recursos;
- **Sem Estado**: significa que o servidor não guarda o estado do cliente, obrigando às mensagens (pedidos e respostas) serem autossuficientes;
- **Arquitectura Cliente-Servidor**: Assume-se um sistema desconectado, há separação de assuntos, a interface uniforme é o elo entre os dois;
- **Armazenável em *Cache***: as respostas (representações) do servidor são armazenáveis em *cache*, agilizando processos.
- **Sistema de Camadas**: não existe uma ligação directa entre o cliente e o servidor, há *software* ou *hardware* intermediário pelo meio, o que promove escalabilidade.
- **Código Sob Demanda** (opcional): o servidor pode temporariamente estender-se ao cliente, transferir a lógica ao cliente, e o cliente executa a lógica (como por exemplo, no JavaScript).

4.6.3. URI, URL e URN (Identificador, Localizador e Nome de Recurso Uniformes)

Um URI (*Uniform Resource Identifier*) é um identificador de recurso uniforme, que consiste numa *string* de caracteres utilizada para identificar um recurso na internet, seja por localização, por nome, ou por ambos.

Já um URL (*Uniform Resource Locator*) é um localizador de recurso uniforme (sob a forma de uma *string* de caracteres), que tal como o nome indica se refere ao endereço (localização) de um recurso. É o meio mais utilizado na internet para localizar recursos. O URL consiste em dois componentes mandatórios, o protocolo de comunicação e o domínio

(Figura 10). Existem outros componentes opcionais, como o caminho da directoria no servidor, a porta, *query strings*, *fragmentos*, etc.

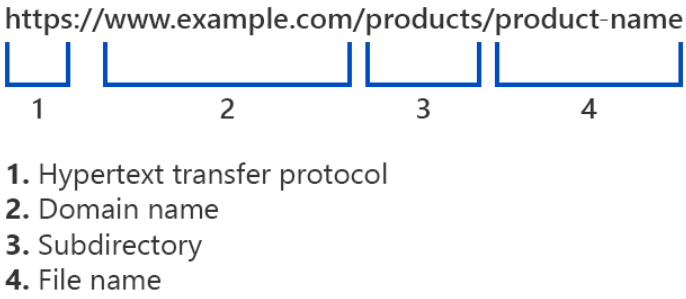

Figura 10 – **Exemplo da estrutura constituinte de um URL** *[31]*.

Finalmente, um URN (*Uniform Resource Name*) é um nome de recurso uniforme, que à semelhança dos anteriores, vem também expresso sob a forma de uma *string* de caracteres, referindo-se apenas ao nome de um recurso. Uma vez que dois ou mais recursos podem ter o mesmo nome, é ineficaz usar URNs, e por isso utilizam-se mais URLs.

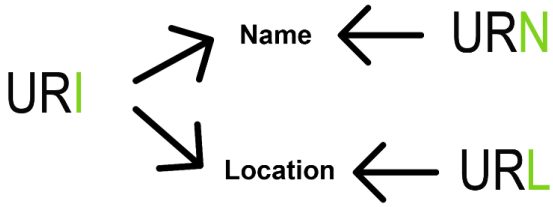

Figura 11 – **O que representam URI, URL e URN e a forma como se relacionam**.

Na verdade, URL e URN são ambos um URI (Figura 11), e estes termos foram inventados porque antigamente havia dois tipos de URI, os que designavam apenas a localização, e os que designavam apenas o nome.

4.6.4. HTTP (Protocolo de Transferência de Hipertexto)

É o protocolo de comunicação ao nível de aplicações mais utilizado no mundo, que permite às aplicações baseadas na *web* comunicar entre si e trocar informação.

O HTTP (*HyperText Transfer Protocol*) é baseado no protocolo TCP/IP. É utlizado para recolher e entregar conteúdos, como imagens, vídeos, ficheiros de áudio, documentos, etc [32].

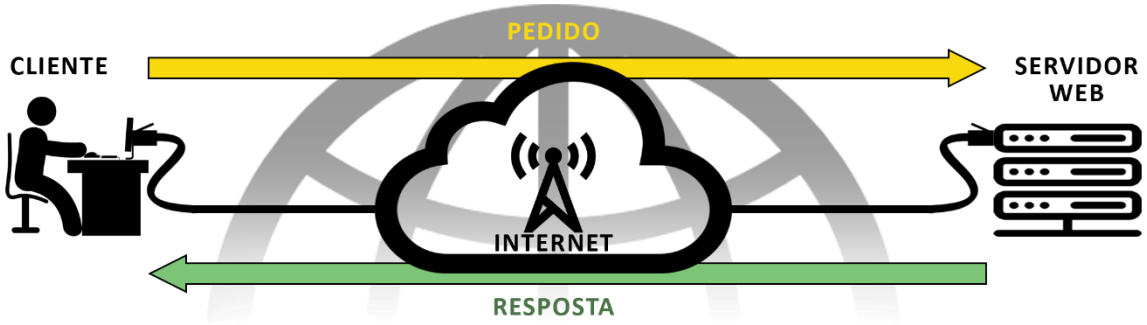

Figura 12 – **Esquemático de um ciclo de Pedido e Resposta via HTTP entre utilizador e servidor**. Imagem adaptada de *[33]*.

Basicamente, há dois extremos, o cliente e o servidor que podem estar em partes distintas do globo. O cliente tenta aceder a um *website* (sítio na internet), através de um URL (protocolo + domínio + caminho) inserido num programa, habitualmente um *browser*. Mas antes disso, eles devem estar fisicamente conectados (computador do cliente e servidor), é esse o propósito da internet, que através dos protocolos TCP/IP estabelece uma combinação de ligações por cabo e sem fios por todo o mundo, permitindo que as duas máquinas comuniquem (computador do cliente e servidor). Assim que essa conexão seja estabelecida, o cliente envia um pedido através do protocolo HTTP e, quando a resposta está pronta, o servidor envia a respectiva resposta (Figura 12).

Tipicamente, as mensagens em HTTP são constituídas pela linha inicial, cabeçalhos e corpo da mensagem (em texto simples ou dados binários) (Figura 13).

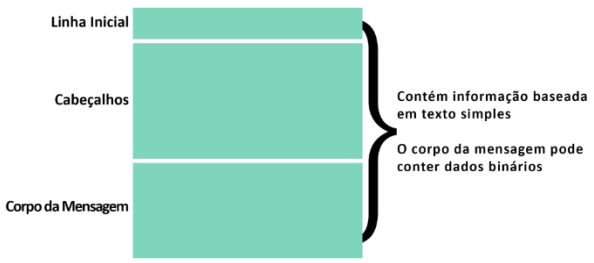

Figura 13 – **Estrutura típica de uma mensagem HTTP**. Imagem adaptada de *[33]*.

A informação nas três secções varia dependendo da mensagem HTTP, se for um pedido, ou uma resposta. No caso dos pedidos, a linha inicial contém o método, o URI e a versão HTTP, e os cabeçalhos (que são pares nome:valor), e não existe corpo pois não é necessário (Figura 14). O método é um comando que diz ao servidor o que fazer (GET, POST, PUT, DELETE, etc). O URI é um conjunto de caracteres que constituem uma forma de localizar um recurso. Os cabeçalhos especificam informação e regras.

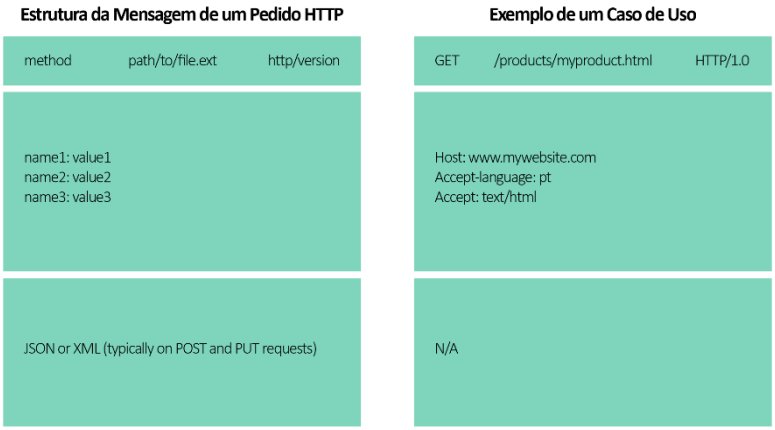

Figura 14 – **Estrutura típica e respectivo exemplo de um pedido HTTP**. Imagem adaptada de *[33]*.

No caso da resposta, a linha inicial, não contém método nem URI pois esses são específicos dos pedidos. No entanto, contém o código de estado, que diz que se o pedido foi bem-sucedido ou não (e respectivo problema) (exemplo 200 OK, 404 *File Not Found*). E na resposta existe o corpo da mensagem contendo o ficheiro requisitado (Figura 15).

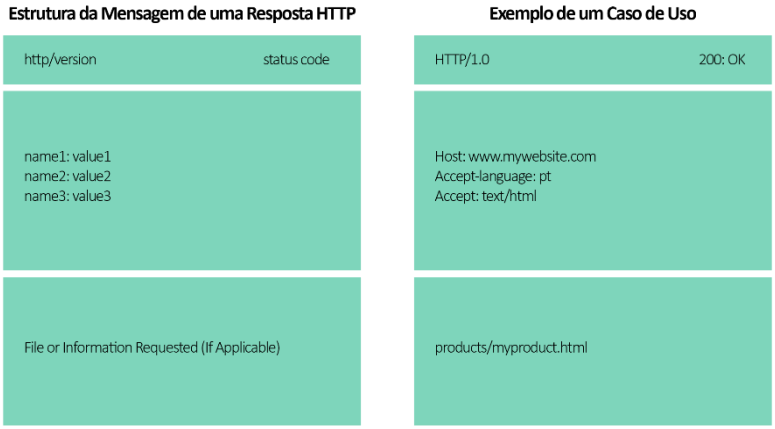

Figura 15 – **Estrutura típica e respectivo exemplo de uma resposta HTTP**. Imagem adaptada de *[33]*.

Em suma, é uma forma muito conveniente e segura de mover informação na internet.

Relativamente aos verbos HTTP [34]**:**

Figura 16 – **Tabela de operações CRUD com as respectivas declarações em SQL e verbos em HTTP** *[35].*

- **GET** – este método é utilizado para ler (ou obter) a representação de um recurso, que habitualmente vêm em formato XML ou JSON. Isto é feito sem modificar qualquer tipo de dados;
- **POST** – é o método mais utilizado para criar recursos através de um pedido HTTP onde podem ser passados dados alusivos a essa criação sob a forma de ficheiro XML ou JSON;
- **PUT** – este método é utilizado para actualizar recursos, mas em alguns casos pode também servir para criar recursos ao passar dados em XML ou JSON no corpo da mensagem do pedido HTTP;
- **DELETE** – tal como o nome indica, este método serve simplesmente para apagar um determinado recurso.

Existem alguns outros métodos, no entanto, aqueles que merecem maior destaque são os mencionados anteriormente.

Relativamente aos códigos de estado HTTP [34]**:**

- **1xx**: Informativo – utilizado para enviar informações para o cliente de que o seu pedido foi recebido e está a ser processado;
- **2xx**: Sucesso – indica que o pedido do cliente foi bem-sucedido;
- **3xx**: Redireccionamento – informa acções adicionais que devem ser tomadas para completar o processo;
- **4xx**: Erro no Cliente – avisa que o cliente fez um pedido que não pode ser atendido;
- **5xx**: Erro no Servidor – ocorreu um erro no servidor ao atender a um pedido válido.

4.6.5. REST API

O que é uma REST API?

Finalmente, após uma breve análise aos termos API e REST, é possível afirmar que uma REST API, é basicamente uma forma de dois *softwares* comunicarem entre si (API) obedecendo às regras da arquitectura REST. É também habitual designar estas APIs como RESTful APIs.

Como funciona uma REST API?

Para uma API funcionar é necessário perceber como vão ser resolvidas algumas questões, para as quais existem várias respostas. No entanto, se é pretendido que uma API seja RESTful devem resolver-se as questões de acordo com as regras REST.

Assim, resumindo [30]:

- Para o cliente dizer ao fornecedor de serviço que operação pretende realizar (informação do método), este deve utilizar operações padrão e para isso, a

operação deve ser expressa através dos verbos HTTP adequados a cada situação (GET para ler informação, POST para criar informação, PUT para actualizar informação, DELETE para apagar informação, entre outros).

- O cliente deve dizer ao fornecedor de serviço quais os dados nos quais deve operar (informação do foco) passando essa informação no URI. Em REST, tudo o que um cliente pode manipular, é considerado um recurso (entidade, objecto ou um conjunto destes) e cada recurso é identificado por um único URI que pode ser passado como parâmetro.

- A comunicação não tem estado (*stateless*). Significa que o servidor não mantém o estado dos clientes. Ou seja, não existe início de sessão onde os dados de autenticação do cliente ficariam guardados do lado do servidor consumindo recursos, o que poderia levar a problemas de escalabilidade e desempenho. Os pedidos REST são autossuficientes, pois levam contidos os dados de autenticação em cada pedido.

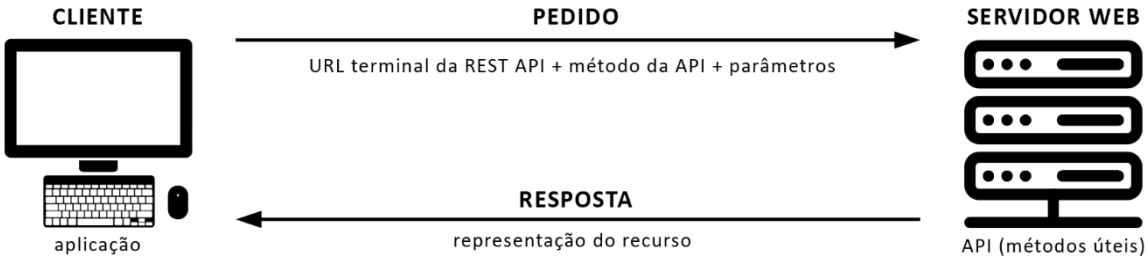

Figura 17 – **Esquemático de uma comunicação feita através de uma REST API**. Imagem adaptada de *[36]*.

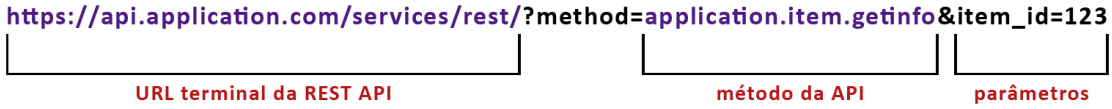

Figura 18 – **Exemplo e estrutura de um pedido HTTP**. Imagem adaptada de *[36]*.

- A representação dos recursos deve ser definida por formatos de dados comuns (sendo as mais comuns entre programadores o JSON, ou XML, por serem facilmente interpretados e manipulados).

- Todo o processamento inerente a este tipo de comunicações é feito do lado do servidor.

Embora este tema seja teoricamente mais complexo, os conceitos atrás referidos de forma resumida, permitem o entendimento das questões de maior relevância.

4.7. *Software* (Tecnologias) Utilizado no Estágio

4.7.1. Sistema Operativo

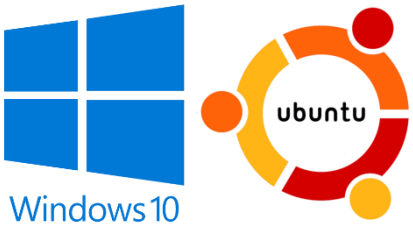

Figura 19 – **Logotipos dos Sistemas Operativos *Windows 10*** *[37]* & ***Linux Ubuntu*** *[38].*

Windows 10 & **Linux Ubuntu**

A nível de sistema operativo, foi mais utilizado o **Windows 10** da Microsoft em actividades de gestão e comunicação. Para o desenvolvimento foi mais utilizado o sistema operativo **Linux Ubuntu** 18.04 LTS da Canonical.

Ambos os sistemas operativos corriam na mesma máquina (computador), sendo que o Windows 10 funcionou como hospedeiro (*Host OS*). O sistema operativo de desenvolvimento Linux Ubuntu 18.04 (*Guest OS*) foi instalado e configurado através de uma máquina virtual, por sua vez criada com recurso ao *software* VirtualBox da Oracle.

4.7.2. Sistema de Virtualização

Figura 20 – **Logotipo do Sistema de Virtualização *Virtual Box*** *[39].*

Virtual Box

O **Virtual Box** é um virtualizador de sistemas operativos que está disponível gratuitamente como *software* de código aberto (*open source software*).

Este pode ser instalado em diferentes sistemas operativos e permite virtualizar igualmente esses sistemas, criando assim uma camada de protecção adicional para o utilizador, e criando cópias de segurança para as quais se pode reverter facilmente se algo correr mal no sistema.

4.7.3. Sistema de Controlo de Versão

Figura 21 – **Logotipo do Sistema de Controlo de Versão *Git*** *[40].*

Git

Criado em 2005 por Linus Torvalds, o **Git** é um sistema de controlo de versões amplamente utilizado no desenvolvimento de *software* que permite gerir o código fonte, rastrear ou acompanhar mudanças a ficheiros. É *open-source* e, portanto, gratuito. Basicamente funciona por ramos, permite a criação de novos ramos que vão clonar o ramo de onde têm origem, e nestes ramos poderão ser feitas alterações e podem ser efectuados testes, sem qualquer interferência com o ramo original. Da mesma forma, permite a união de ramos, que consiste em introduzir as alterações feitas nos ramos secundários ao ramo

original. Todas estas operações são feitas com recurso a vários comandos próprios do Git. [41]

Para tentar visualizar melhor esta ideia, segue-se uma imagem de uma das metodologias aplicadas ao Git para desenvolvimento de *software*, chamada de ***Gitflow*** (Figura 22), criada por Vincent Driessen. Esta metodologia permite desenvolvimento paralelo do mesmo código. Basicamente o ramo *master* é aquele que permanece na linha de produção e contém o produto final. Deste é feita uma cópia que estará sujeita a alterações, o ramo *develop*. Deste ramo *develop*, saem os ramos *feature*, onde são trabalhadas novas funcionalidades e quando prontas, voltam a unir-se ao ramo *develop* actualizando o mesmo. Quando o ramo *develop* está pronto para um novo lançamento, cria-se uma cópia deste a que se dá o nome de ramo *release*, com o intuito de lançar em ambiente de testes, corrigir eventuais defeitos encontrados e então lançar para produção. Existe ainda o ramo *hotfixes* com o intuito de efectuar correcções de emergência.

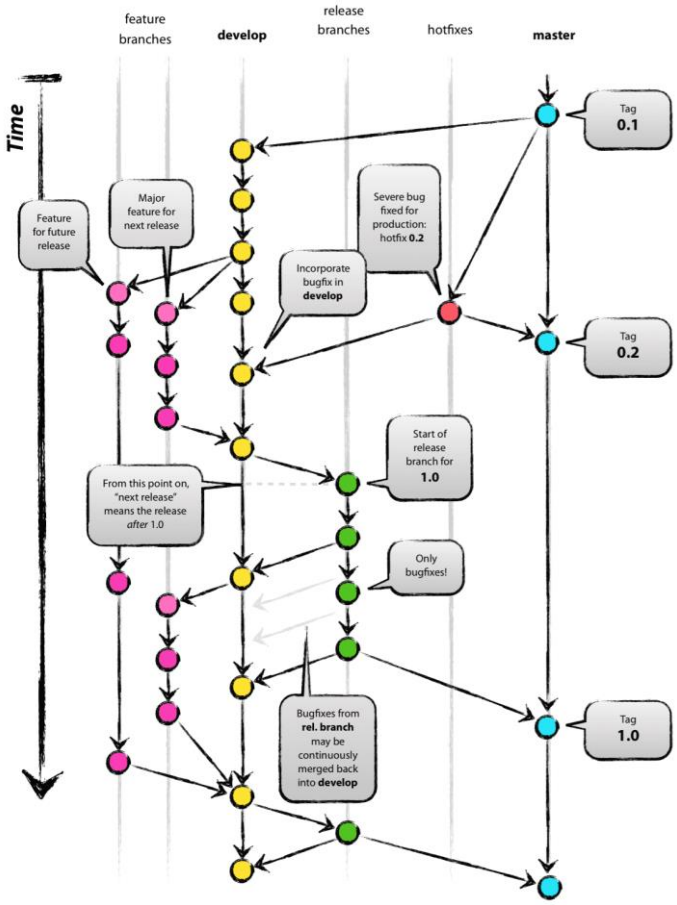

Figura 22 – **Modelo do *Gitflow* criado por Vincent Driessen** *[42].*

4.7.4. Repositório e Gestão do Código Fonte

Figura 23 – **Logotipo do Repositório *Bitbucket*** *[43].*

Bitbucket

A função dos repositórios não é nada mais do que armazenar o código fonte de projectos. E estes recorrem aos sistemas de controlo de versão para gerir os mesmos. Requer a criação de uma conta de utilizador, sob a qual serão associados projectos. Neste caso, trabalhou-se mais de perto com o **Bitbucket**, que pertence à Atlassian, e suporta Git e Mercurial como sistemas de controlo de versões.

Outros repositórios incluem o GitHub ou o GitLab.

4.7.5. Gestão de Projectos

Figura 24 – **Logotipo do Gestor de Projectos JIRA** *[44].*

JIRA

O **JIRA**, da Atlassian, é uma ferramenta que permite gerir projectos e actividades relacionadas num único lugar, por várias pessoas. Nestes projectos são agrupadas e organizadas tarefas definidas pela equipa de acordo com as suas necessidades. Esta organização é feita com recurso a painéis editáveis e a ferramenta conta também com filtros e com apresentação de estatísticas relativas ao desenvolvimento dos projectos.

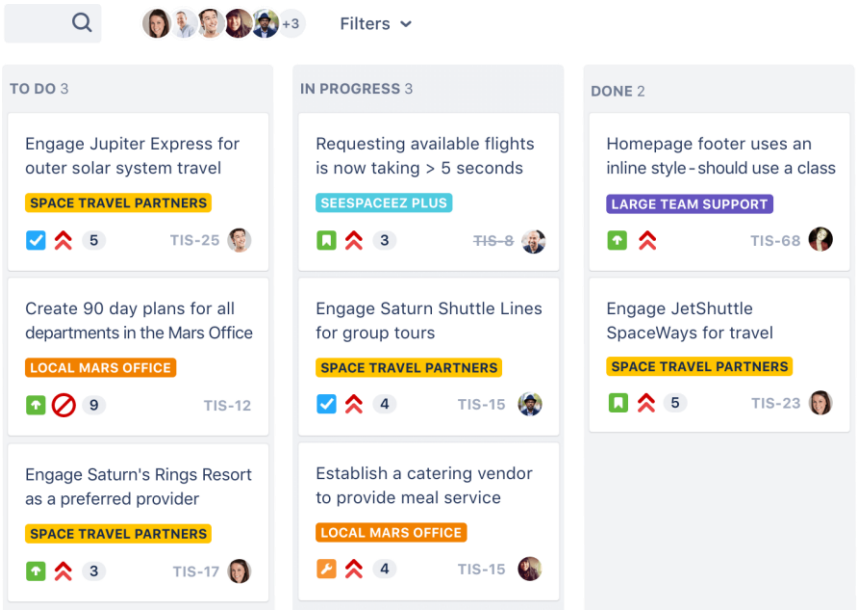

Figura 25 – **Exemplo de um quadro *Scrum* no JIRA** *[45]*.

Figura 26 – **Exemplo de vários gráficos que ilustram o progresso de um projecto no JIRA** *[45]*.

Este *software* permite a integração com outras ferramentas, e, uma vez que se está no campo dos testes, a empresa recorreu a uma ferramenta de gestão de testes, ferramenta

essa chamada de **Xray**, que se integra na perfeição e acrescenta assim funcionalidades adicionais ao **JIRA**.

4.7.6. Gestão de Testes

Figura 27 – **Logotipo do Gestor de Testes *Xray*** *[46].*

Xray for JIRA

Este *add-on* foi desenvolvido pela XpandIT, uma empresa portuguesa, que para além deste, conta com outros produtos desenvolvidos por ela e que são já usados a nível internacional por várias empresas de renome.

O **Xray** permite efectuar a gestão de testes de uma forma mais concisa à semelhança das tarefas no JIRA (Figura 29), apresentado os seus resultados visualmente, organizando-os em grupos de testes e planos de testes. Permite reportar facilmente defeitos encontrados durante os testes.

Permite a importação de resultados através de ficheiros XML (por integração com *frameworks* de teste como Selenium, JUnit e Nunit), JSON e CSV (*comma-separated values*, comum nos ficheiros de Excel).

Foca-se no preenchimento de requisitos, apresentando gráficos e relatórios dos mesmos.

Mantém a rastreabilidade entre requisitos, testes, defeitos e execuções de testes (Figura 30).

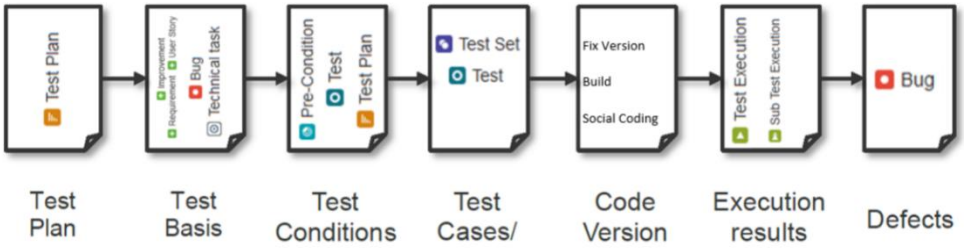

Figura 28 – **Exemplo de organização das tarefas (*issues*) do Xray com base na sua função**.

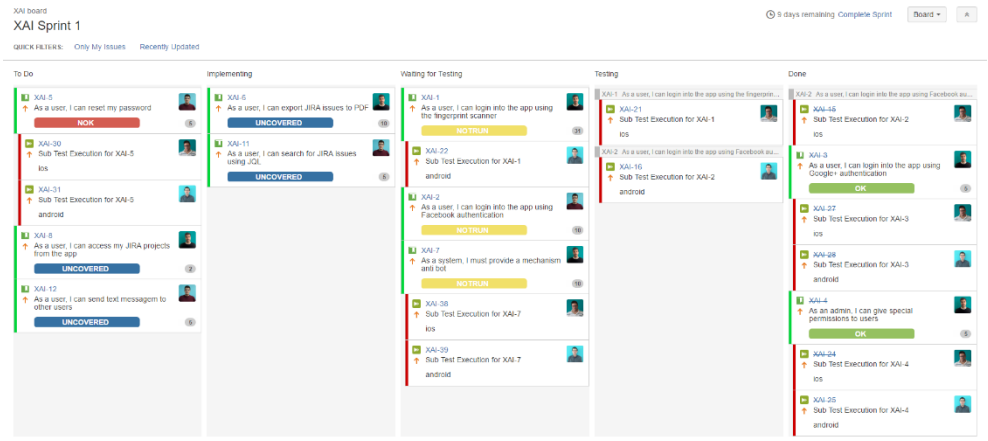

Figura 29 – **Exemplo da vista dos *Sprints* de um projecto no JIRA Xray** *[47]*.

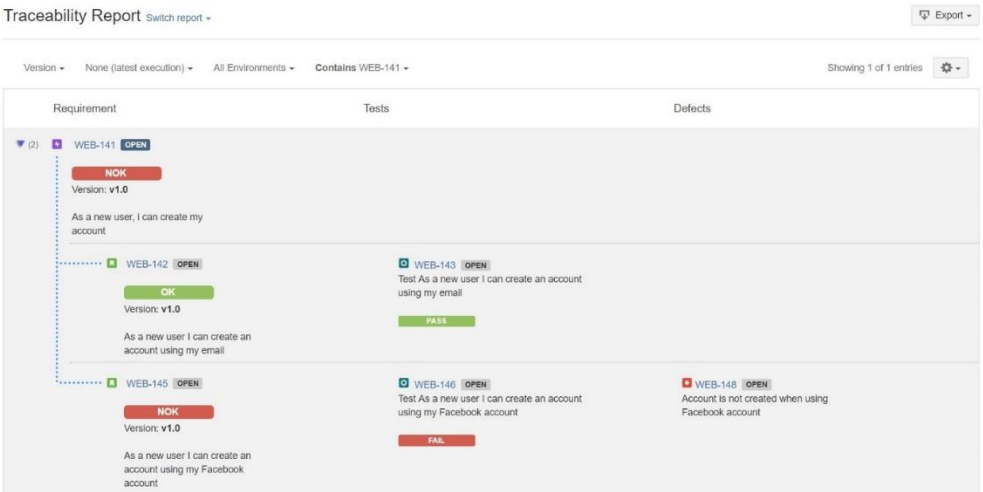

Figura 30 – **Exemplo da rastreabilidade mantida pelo JIRA Xray entre as várias tarefas (*issues*)** *[48]*.

4.7.7. Análise Estática do Código Fonte

Figura 31 – **Logotipo do Software de Análise Estática *SonarQube*** *[49].*

SonarQube

O **SonarQube** é uma ferramenta *open source* que permite efectuar revisões automáticas ao código através de uma análise estática para detectar *bugs*, problemas mais extensos e vulnerabilidades em várias linguagens de programação.

Esta plataforma disponibiliza resultados sobre código duplicado, o padrão e boas práticas de uma linguagem de programação, quantidade de testes unitários implementados, a percentagem de código abrangido por testes, a complexidade do código, alertas sobre comentários, pequenos *bugs* e vulnerabilidades de segurança. [49]

Adicionalmente, a ferramenta mantém um histórico de resultados e com isto fornece métricas e gráficos da evolução do *software* em desenvolvimento.

Outro ponto importante, é a possibilidade de integração com várias ferramentas da área de desenvolvimento e automação, como Bitbucket, Maven ou Jenkins.

4.7.8. Análise das Vulnerabilidades a Nível de Segurança

Figura 32 – **Logotipo do Software de Verificação de Segurança *Checkmarx*** *[50].*

Checkmarx

Esta ferramenta, à semelhança do SonarQube, recorre a um tipo de análise estática ao código fonte. No entanto, esta foca-se nas vulnerabilidades de segurança, desde cedo no

desenvolvimento de um *software*, pois hoje em dia, nos ambientes e aplicações amplamente conectados, geridos por APIs, qualquer componente de um *software* pode ser invocado e potencialmente explorado de várias formas. O **Checkmarx** pretende prevenir isso.

É uma solução flexível, precisa e suporta todas as principais linguagens de programação. Conta também com uma modesta API, ideal se pensarmos em automatizar alguns dos processos que a envolvem.

4.7.9. Repositório de Artefactos / Binários

Figura 33 – **Logotipo do Repositório *Artifactory*** *[51].*

Artifactory

O **Artifactory** foi o primeiro, e é ainda apontado como o principal gestor universal e repositório para artefactos (ou binários) no mercado do *software*. É muito utilizado para armazenar e facilitar a partilha de bibliotecas e componentes de um *software*, em especial nas grandes equipas de desenvolvimento, de forma organizada, com recurso a versões, e disponibiliza uma pesquisa de artefactos fácil.

Ao mesmo tempo, mantém um nível de segurança sobre todos os conteúdos colocados neste repositório, permitindo configurar permissões para quem pode ou não aceder aos mesmos. Também garante uma enorme estabilidade face a grandes cargas nos seus servidores.

Este oferece uma grande capacidade de integração com praticamente todas as ferramentas, linguagens de programação e tecnologias, pois é amplamente utilizado e isso leva a que se desenvolvam novas funcionalidades para a ferramenta mais rapidamente, contando, assim, com uma poderosa API que é também ideal para automatizar processos.

4.7.10. Documentação de Projectos

Figura 34 – **Logotipo da Plataforma de Documentação Confluence** *[52].*

Confluence

Classificada como um *software* de colaboração, esta plataforma foi desenvolvida e publicada pela companhia australiana Atlassian.

É um dos principais meios de informação, onde as equipas podem colocar documentação sobre todos os seus processos e produtos, desde a fase do planeamento à sua manutenção, ou tutoriais, entre outros tópicos, de uma forma organizada e facilmente acessível. A plataforma permite aos utilizadores controlar quem pode aceder ou não a determinado espaço ou página.

Os conteúdos são altamente personalizáveis e existem múltiplas funcionalidades adicionais, como a criação de tabelas com filtros ou a exportação dos conteúdos em formato PDF.

Como esta ferramenta foi desenvolvida pela Atlassian, integra-se facilmente com outras ferramentas do mesmo desenvolvedor, em particular com o JIRA, que é bastante útil para criar ligações com as tarefas nessa plataforma.

O **Confluence** conta ainda com uma extensa API, que facilita a automatização de alguns processos que envolvem a ferramenta.

4.7.11. Ambiente de Desenvolvimento Integrado (IDE)

Figura 35 – **Logotipo do IDE *IntelliJ IDEA*** *[53].*

IntelliJ IDEA

É um tipo de *software* que já é familiar a muitos, reunindo as ferramentas necessárias para desenvolver *software*, entre as quais o editor de código fonte, o compilador que transforma a linguagem escrita em código binário, o depurador (*debugger*) que auxilia no processo de encontrar e corrigir defeitos no código fonte.

Neste caso o mais usado foi o **IntelliJ IDEA** tendo também passado pelo Android Studio.

Outros IDE incluem o Xcode.

4.7.12. *Framework* de Testes

Figura 36 – **Logotipo da *Framework* de Testes *JUnit*** *[54]*.

JUnit

As *frameworks* de testes providenciam um conjunto de bibliotecas, que por sua vez contêm várias funções, anotações e asserções que nos permitem escrever os nossos testes no código. Contam também com executores de testes, que instanciam as nossas classes de teste, executam os nossos métodos de teste presentes nessas classes, e reportam os resultados.

No caso da *framework* **JUnit**, esta está direcionada à linguagem de programação Java, e serve essencialmente a criação de testes unitários e/ou testes de integração.

Todas as asserções estão na classe *Assert*. Esta classe fornece um conjunto de métodos de asserção, úteis para escrever testes. Apenas as asserções falhadas são registadas.

As anotações são como etiquetas que se podem adicionar ao código, e aplicar aos métodos ou classe. Estas fornecem informação sobre que métodos vão ser executados antes e depois dos métodos de teste, antes e depois de todos os métodos e que métodos ou classes vão ser ignorados durante a execução.

Outras *frameworks* de testes incluem o Espresso, para testes UI (*User Interface*) em Android, e o XCTest, para testes UI em iOS.

Figura 37 – **Logotipo Alternativo da *Framework* de Testes *TestNG*** *[55].*

O **TestNG** é uma *framework* de testes baseada na JUnit e NUnit, mas com novas funcionalidades que a tornam mais poderosa e fácil de usar. Uma dessas funcionalidades foi de resto explorada durante o estágio, e possibilita o mapeamento ou associação automática entre requisitos e testes na plataforma do Xray, processo esse que será descrito no próximo capítulo.

Para além disso, foi projectada para abranger todos os tipos de testes, desde testes unitários, a testes de integração ou testes de sistema.

4.7.13. *Build Automation Tool*

Figura 38 – **Logotipo da Ferramenta de Automatização *Maven*** *[56].*

Maven

No caso do **Maven**, o projecto é configurado através do POM (*Project Object Model*) guardado no formato XML (*pom.xml*), que contém informação como as propriedades do projecto (campos que são passados para o Xray), as suas dependências em outras ferramentas, os *plugins* necessários, entre outros. Quando executado, o Maven procura todas as ferramentas necessárias para a *build* do projecto (desde a compilação do código, à instalação e configuração do mesmo) e corre os testes automaticamente.

No fundo é nisso que consiste o processo de automatizar a *build*, é o processo de automatizar a criação de uma *build* de *software* e os processos associados incluindo: compilar o código fonte em código binário, efectuar o *packaging* desse código binário e correr os testes automatizados.

Esta ferramenta conta também com *plugins* que possibilitam gerar relatórios de resultados em ficheiros do tipo XML e em ficheiros de texto (TXT).

Surefire no caso de testes unitários e *Failsafe* no caso de testes de integração.

Esses ficheiros são, por defeito, gerados na seguinte directoria: *${basedir}/target/failsafe-reports/TEST-*.xml*

Outros exemplos de ferramentas que automatizam a compilação incluem o Gradle (mais usado para projectos Android).

4.7.14. Integração, Entrega e Deploy Contínuos

Figura 39 – **Logotipo da Ferramenta *Jenkins*** *[57].*

Jenkins

Jenkins é uma ferramenta *open source* de automação. Consiste num servidor escrito em Java, que ajuda a automatizar a parte não dependente da "mão humana" do processo de desenvolvimento de *software*. Assim, permite ganhar tempo que antes era perdido em tarefas redundantes, e reduz a probabilidade de falhas, através da integração contínua (Figura 40).

As execuções deste podem ser activadas de várias formas, uma das quais passar por fazer um simples *commit* no sistema de controlo de versão como o Git.

Podem ser acrescentadas funcionalidades ao Jenkins com recurso a *plugins*.

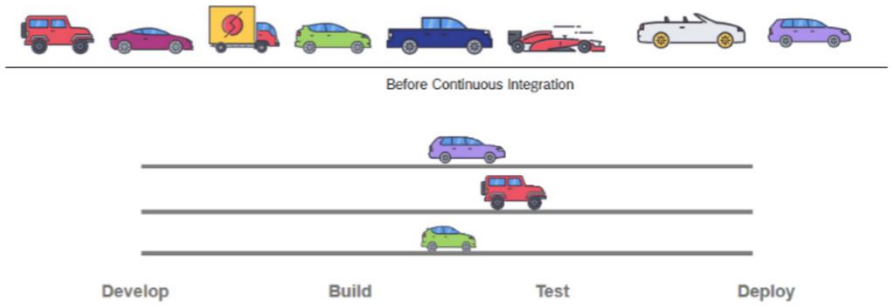

Figura 40 – **Exemplo ilustrativo dos benefícios da implementação de integração contínua num projecto**.

4.7.15. Colaboração e Desenvolvimento de APIs

Figura 41 – **Logotipo da Ferramenta de Desenvolvimento de APIs *Postman*** *[58]*.

Postman

Aquando da automatização de alguns processos que requerem a utilização das APIs, o **Postman** é a ferramenta ideal para desenvolver e testar pedidos REST ou SOAP[4] antes de os colocar em código para o Jenkins executar, facilitando o despiste de problemas e economizando tempo.

Permite ainda guardar todos esses pedidos de forma organizada e é possível partilhar essas coleções com outros utilizadores.

4.7.16. Outros

Figura 42 – **Logotipos das Ferramentas de Comunicação *Outlook*** *[59]*, ***Skype*** *[60]* **e *Rocket Chat*** *[61]*.

[4] SOAP: *Simple Object Access Protocol*, é um protocolo para troca de informações estruturadas.

Aplicativos de Comunicação / *Chats*

Em ambiente empresarial é vital a existência de ferramentas para comunicação. Aqui o mais utilizado terá sido o **Outlook** da Microsoft para a gestão de correio electrónico, no entanto também se recorreu a aplicativos de mensagens como o **Skype** em videoconferências ou o **Rocket.Chat** que consiste num *software* gratuito, *open-source* ideal para comunicação em equipa. De referir, que internamente, existiam inúmeras plataformas de comunicação como fóruns, *wikis*, blogues, entre outras.

Capítulo 5. Trabalho Desenvolvido

Neste capítulo pretende-se descrever as atividades realizadas na empresa ao longo dos meses em que decorreu o estágio. Dado que já havia concluído um período na empresa, dedicado à automatização de testes em *software*, num passado recente (antes do estágio de mestrado), já estava familiarizado com as instalações e com as equipas com as quais trabalhei. Portanto, o período inicial foi, essencialmente, dedicado à preparação do ambiente de trabalho. De seguida, iniciaram-se os trabalhos para a garantia de qualidade no *software*, que desta vez, para além de dar continuidade aos testes em *software*, se estenderam a outros campos. Mantendo a atenção na automação, desta vez o principal foco foi tentar automatizar o processo de verificação de conformidade com os padrões de qualidade da empresa, de forma a agilizar o lançamento (*release*) de *software* no mercado. De uma forma mais discriminada, seguem na tabela seguinte, as várias tarefas desempenhadas no estágio, sem nenhuma ordem em particular:

Tabela 1 – **Resumo das tarefas realizadas durante o período de estágio**.

Tarefas Desempenhadas
Preparação do ambiente de trabalho
Conhecer o processo de qualidade nas equipas de desenvolvimento de *software* em Aveiro
Recolha dos ficheiros de configuração das *pipelines* (*jenkinsfiles*) de todas as equipas de desenvolvimento de *software* em Aveiro
Análise da documentação interna da empresa relativamente a critérios de conformidade com os padrões de qualidade e definição do seu cumprimento através de ferramentas utilizadas na empresa
Criação de um conceito de *release* automática de *software* que cumpra com os padrões de qualidade
Contribuição no desenvolvimento de uma biblioteca de funções para Jenkins, disponível a todas as equipas do departamento
Análise e avaliação de possíveis melhorias a essa biblioteca de funções para Jenkins
Teste ao conceito de *release* automática de *software* junto de uma equipa com recurso a um projecto de demonstração

Documentação de todos os procedimentos para disponibilização interna de forma a que todas as equipas possam consultar e realização de um tutorial acerca do tema
Análise à possibilidade de associar automaticamente requisitos (*user stories*) com os resultados dos testes importados para a plataforma do Xray
Participação numa fase de testes de uma versão interna do *software* SourceTree
Suporte (apoio) às várias equipas em questões relacionadas com a integração das ferramentas utilizadas na garantia de qualidade

5.1. Desenho e Automatização do Processo de *Release* de *Software*

5.1.1. Porquê automatizar este processo?

Essencialmente, esta automatização pretende agilizar a *release* de *software*, evitando os longos períodos de espera inerentes às reuniões formais dependentes de oportunidade de calendário, que no fundo seriam para uma verificação exaustiva a toda a documentação e aos passos definidos e acordados pela empresa, necessários à aprovação do projecto.

É também importante salientar, que o foco desta automatização seriam as pequenas actualizações, sendo que as grandes *releases* teriam de passar sempre, inicialmente, por uma aprovação formal. Mas daí em diante, e com o aval dos departamentos de qualidade, as pequenas actualizações seguiriam com o processo automático, sem necessidade de espera, permitindo agilizar as actualizações que muitas vezes se querem rápidas (exemplo os *hotfixes*).

Para se perceber melhor a diferença entre os tipos de actualizações, estas, geralmente seguem regras base de sistema de controlo de versões (ou versionamento) chamadas de 'Versão Semântica', num formato X.Y.Z (*Major*.*Minor*.*Patch*) [62]. A forma como cada número incrementa irá depender da alteração introduzida. As correções de erros simples que não afectam a API incrementam a versão do *patch* (geralmente associado ao ramo *hotfix*), as alterações feitas à API que mantêm a compatibilidade com versões anteriores (desenvolvimento de novas funcionalidades), incrementam a versão secundária *minor*, e as alterações incompatíveis com a API (*refactoring* ou alterações na tecnologia que levam à

falta de retro-compatibilidade) incrementam a versão principal *major* (que estão geralmente associados ao ramo de *release*). Seguindo esta abordagem, os números da versão e a forma como eles se alteram transmitem o significado sobre o código subjacente e o que foi modificado de uma versão anterior para a seguinte.

Existe também o termo MVP (*Minimum Viable Product*, produto viável mínimo) que consiste na versão mais simples de um produto que pode ser lançada, que já cumpre com o seu propósito básico, tendo valor suficiente para que os utilizadores possam começar a usufruir do mesmo, fornecendo ao mesmo tempo, dados e impressões da parte dos utilizadores para ser melhorada nas versões seguintes. Geralmente corresponde à versão 1.0.0.

5.1.2. Como automatizar?

A peça fundamental na automatização de todo o processo é o Jenkins, ferramenta de integração contínua. E para entender melhor o seu funcionamento será feita uma breve introdução.

Basicamente, todos os projectos cujos processos se pretendam automatizar, devem conter um ficheiro associado ao Jenkins na sua estrutura, que normalmente assume o nome de *Jenkinsfile*. É este ficheiro que irá conter todas as instruções (*pipeline*/*workflow*) a executar pela máquina do Jenkins (Figura 43). O arranque de cada execução, pode ser determinado de diferentes maneiras, desde intervalos de tempo definidos pelo utilizador a eventos gerados pelos programadores, como os *commits* (método muito comum no desenvolvimento, pois permite ao utilizador saber se estão a ser introduzidos erros ou problemas no projecto com a última alteração efectuada).

Uma *pipeline* do Jenkins é uma coleção ou, se preferirmos, um agregado de tarefas / passos que levam o *software* desde o sistema de controlo de versão (do lado do desenvolvimento) até às mãos do utilizador final (do lado da produção) com a ajuda de ferramentas de automação de processos no desenvolvimento de *software*.

Assim como qualquer outro problema, existem diferentes formas de chegar a uma solução, e as *pipelines* do Jenkins são uma delas. Para fazer uso das *pipelines* do Jenkins é

necessário a instalação de um *plugin* no Jenkins, chamado de *Pipeline* (anteriormente *Workflow Aggregator*).

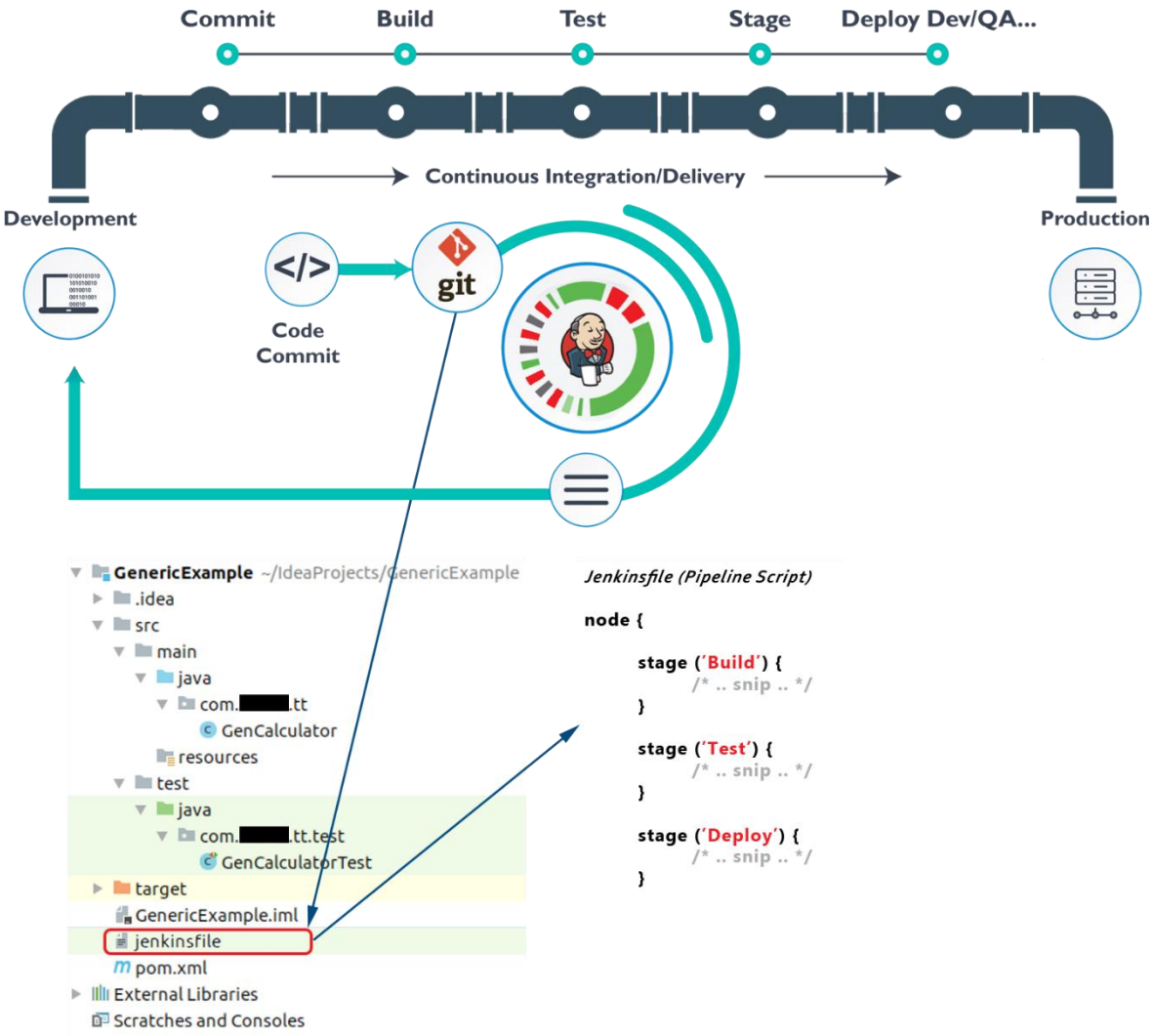

Figura 43 – **Representação de uma *pipeline* do Jenkins e respectivas etapas configuradas no *Jenkinsfile*.**

Quanto aos *Jenkinsfiles*, não são nada mais nada menos do que os ficheiros que contêm todas as configurações que se pretendem para a *pipeline*. O servidor do Jenkins irá então procurar por esse ficheiro contido algures dentro do projecto (deve existir apenas um destes ficheiros por projecto), e executa todas as instruções no seu interior. É importante fazer notar que este ficheiro irá ser replicado em todos os ramos existentes no projecto, mas

pode conter configurações diferentes dependendo do mesmo, como vem demonstrado na figura seguinte (Figura 44).

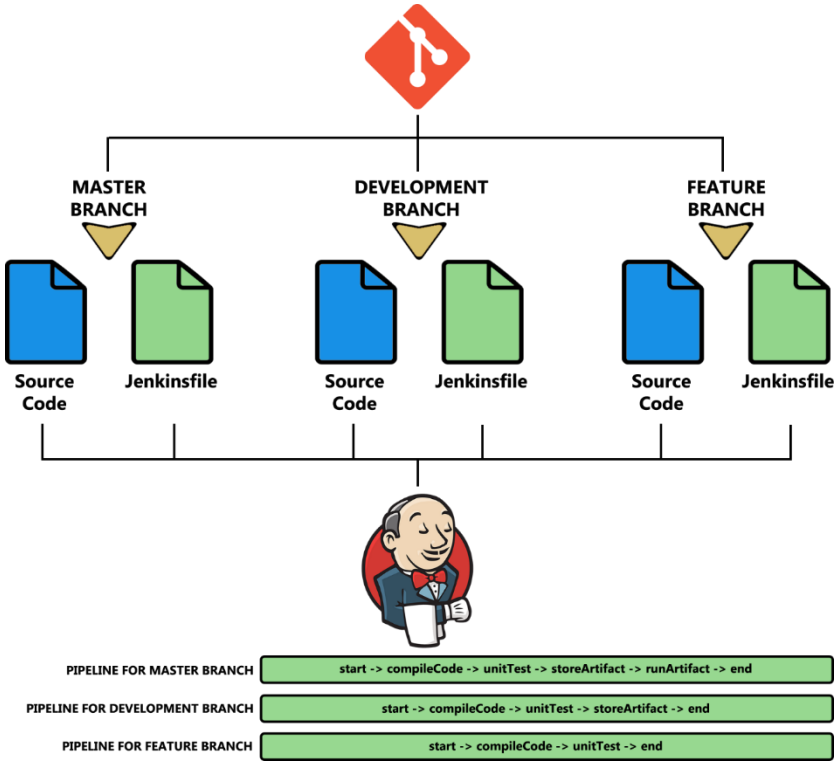

Figura 44 – **Esquema ilustrativo de um projecto com vários ramos (*multibranch*) e de como estes se podem comportar de forma diferente no Jenkins dependendo das suas configurações**.

5.1.3. Análise ao Estado das *Pipelines* das Equipas (Recolha dos *Jenkinsfiles*)

Nesta fase, foi feita uma recolha dos *Jenkinsfiles* das várias equipas de desenvolvimento de *software* para analisar as suas *pipelines*, ou seja, para perceber quais eram os passos fundamentais ao desenvolvimento do *software* e de que forma foram implementadas as etapas que garantem um controlo da qualidade sobre o seu produto.

Após uma análise sobre a forma como as várias equipas de desenvolvimento de *software* em Aveiro implementam os processos de garantia de qualidade nos seus projectos é possível afirmar que existem vários casos distintos, desde equipas novas ou com novos projectos em mãos que procuram começar a implementar estes processos seguindo a

ideologia da empresa e por isso procuram orientação junto dos elementos das equipas de qualidade, a equipas que já têm processos de qualidade implementados com um estado de maturidade algo avançado, mas que por vezes diferem das ideologias mais recentemente adoptadas pela empresa ou que simplesmente procuram continuar a melhorar os seus processos, por exemplo, através da automatização. De notar ainda que a automatização destes processos é algo ainda relativamente recente e nem todas as equipas têm estes processos optimizados. No entanto, após recolha e observação dos *scripts* das várias equipas de desenvolvimento foi possível perceber de uma forma geral os pontos fundamentais para uma base sólida daquilo que se pretende implementar e o que se pode ou não automatizar no presente, colocando numa *pipeline* do Jenkins.

Actualmente, é cedo para pensar numa automatização total, principalmente se pensarmos no planeamento do projecto e de todo o desenvolvimento da documentação necessária ao mesmo, pois ainda depende muito do pensamento / criatividade e da execução humana. No entanto, em campos de verificação e execução repetitiva, pode-se tentar partir para alguma automatização, e ainda que não seja total, é um começo nesse sentido. Assim, pensou-se em automatizar o recolher dos recursos necessários à avaliação da qualidade de *software* em diversos pontos.

5.1.4. Criação do Conceito de *Release* Automática de *Software*

Numa primeira etapa, depois de analisados os *scripts* das equipas, foi necessário efectuar uma análise extensiva à documentação interna da empresa, em particular às instruções processuais acerca dos *Quality Gates*, e da *Framework* de *Software* (TT-VAN005 e TT-VAN137, respectivamente), e desta análise retirar alguns pontos fundamentais numa tentativa de tornar a sua interpretação mais amigável para as equipas, mantendo todos os elementos necessários ao cumprimento dos *quality gates* (verificações / padrões de qualidade).

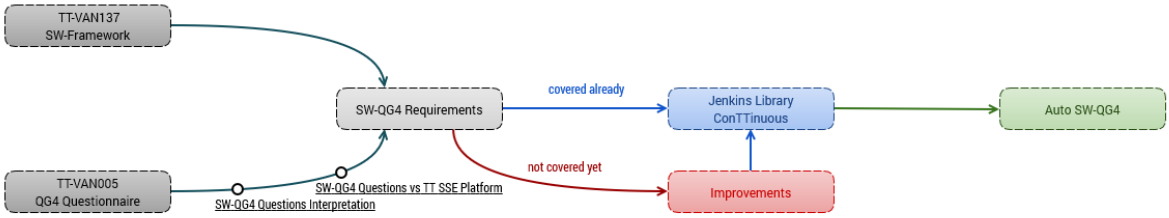

Figura 45 – **Conceito de lançamento de *software* automático proposto**.

Entre os principais pontos a abordar sobre um projecto de forma a garantir que toda a informação acerca do mesmo está disponível bem como os passos de garantia de qualidade sejam efectuados, tem-se a arquitectura do *software* e a sua monitorização, desenvolvimento e gestão de requisitos, análise aos tipos de licenças das ferramentas utilizadas (*open source software scan*), engenharia do produto e análise de foco, implementação de uma estratégia de testes, mapeamento de todos os relatórios de resultados e histórico de desenvolvimento, documentação técnica, notas de lançamento da versão (*release notes*), entrega e colocação em execução de artefactos e notificação de todas as partes envolvidas. Dentro de cada um destes pontos, poderão existir várias subcategorias.

Para cada um destes pontos, foi pensada uma resposta, de preferência, com recurso a ferramentas já utilizadas na empresa, e quando tal não era possível, foram sugeridas melhorias futuras (Figura 45). Algumas dessas melhorias incluíam a realização de uma análise estática da arquitectura do *software* ou de testes de segurança dinâmicos por meio de ferramentas não existentes na empresa, a implementação de métodos que, de forma automática, permitissem disponibilizar e anexar relatórios das análises estáticas à documentação ou apurar a percentagem de requisitos (*user stories*) abrangidos por pelo menos um teste ou ainda a criação de uma *dashboard* que reunisse, visualmente, todos os dados estatísticos e resultados, para uma melhor percepção do panorama geral do estado do *software*.

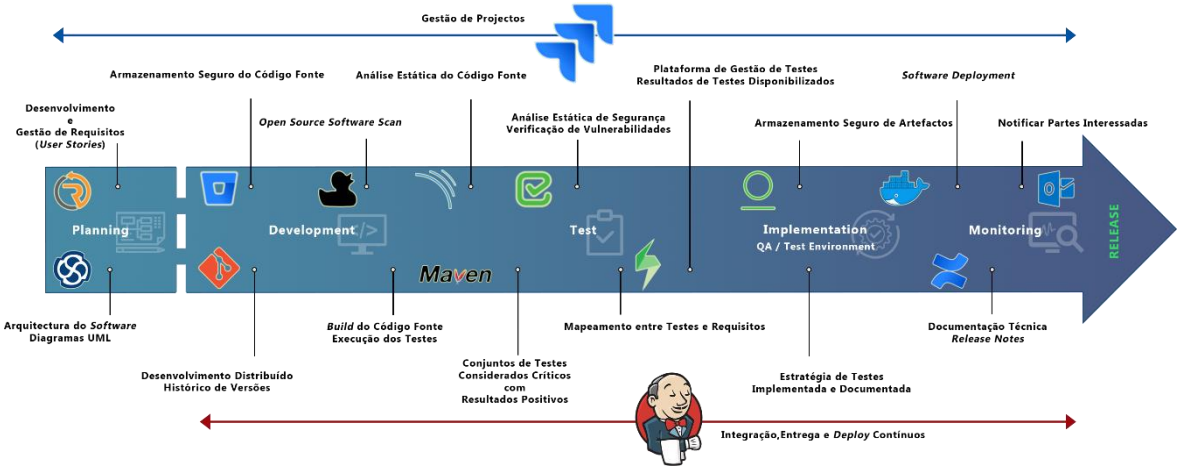

Figura 46 – **Realce de algumas tarefas necessárias (e ferramentas associadas) no desenvolvimento de um *software* para que este possa ser lançado. Base sólida para uma definição de pronto (DoD)**.

Pela observação da figura anterior (Figura 46), pode-se perceber algumas das ferramentas utilizadas e respectivas funções no ciclo de desenvolvimento de um *software*.

Começando do planeamento e projecto do *software*, tem-se por exemplo a plataforma de desenvolvimento e gestão de requisitos (*user stories*), R4J (*Requirements For Jira*), que é um *add-on* à plataforma do JIRA, ou o *Enterprise Architect*, uma ferramenta de modelação e projecto baseada em diagramas (UML, *unified modelling language*) que permite caracterizar todos os constituintes e interfaces de um *software* desde cedo.

Passando à fase de desenvolvimento, começa-se a lidar com o *Bitbucket*, o repositório de código fonte, que o armazena de forma segura, permitindo gerir acessos e que não permite alterações directas ao código, se assim entendermos, tendo de passar por revisão e aprovação de terceiros (a partir dos *pull requests*) e com o Git, o sistema de controlo de versão, que oferece uma abordagem fácil e segura de desenvolver código sem comprometer o já existente, a partir da criação de novos ramos. Esta ferramenta mantém ainda um histórico de versões, permitindo reverter alterações em caso de necessidade.

Na transição da fase de desenvolvimento para a fase de testes existem várias etapas a acontecer, como por exemplo o *Black Duck* para o *open source scan*, que irá garantir conformidade a nível de licenças, usando apenas bibliotecas permitidas. O *SonarQube* efectua uma análise estática ao código por comparação aos padrões da linguagem em questão (no caso do Java, usa-se o JaCoCo, *Java Code Coverage*), tentando encontrar *bugs*,

duplicados e possíveis vulnerabilidades (como código comentado), e verificando a percentagem de código abrangido por testes. O *Checkmarx* recorre também a uma análise estática (SAST, *static application security testing*), mas concentra-se mais nas vulnerabilidades a nível de segurança, prevenindo que terceiros possam aproveitar-se de determinadas funcionalidades do *software* em seu benefício. O *Maven* automatiza a compilação do código e executa os testes (unitários, integração ou sistema) ao mesmo, através dos seus *plugins*, geralmente utilizado na linguagem Java. Ao automatizar este passo, as *builds* do *Maven* são feitas remotamente, mas todo o processo fica registado nos *logs* do Jenkins para posterior revisão.

Nesta fase a estratégia de testes já deverá estar implementada e documentada, e os conjuntos de testes considerados críticos devem passar os limites definidos pela equipa (aplicável a cada tipo de teste). Todos os requisitos possíveis de ser testados, devem estar associados aos respectivos testes, e esse mapeamento será feito pela plataforma de gestão de testes do Xray (*add-on* do JIRA) para onde são enviados todos os resultados, ficando disponíveis para consulta.

De seguida os artefactos são guardados num repositório, o Artifactory, protegidos por autenticação, e, onde permissões podem ser definidas. É também nesta altura que os *softwares* são colocados em execução em ambientes de teste e garantia de qualidade (através do Docker ou DTR, *Docker Trusted Registry*), de forma a que se possa proceder aos restantes tipos de testes (UAT, *user acceptance tests* ou testes exploratórios) e análises dinâmicas (DAST, *dynamic application security testing*). Note-se que estes ambientes não têm qualquer tipo de conectividade com os ambientes de produção. Nesta fase os *softwares* estão também sujeitos a monitorização.

Finalmente, toda a documentação técnica e notas de *release* são disponibilizadas no Confluence, e as partes interessadas do projecto são notificadas com toda a informação relativa ao projecto e à sua garantia de qualidade através de *email*. É nesta altura que a *release* do *software* é feita.

5.1.5. Criação da Biblioteca Partilhada de Funções em *Groovy*

Em paralelo com o exposto na secção anterior, foi criado um projecto (no qual participei) destinado a conter todas as funções / métodos utilizados na automatização das várias etapas da garantia de qualidade, de forma a possibilitar às equipas de desenvolvimento reutilizar esses mesmos métodos sem ter de os voltar a desenvolver. Métodos estes o mais genéricos possível para não criar problemas de compatibilidade com diferentes tecnologias, havendo ainda assim vários módulos devido a contemplar situações de quando os métodos são demasiado específicos para uma tecnologia. No fundo, uma espécie de biblioteca do Jenkins, ferramenta central na automatização das várias tarefas por meio das suas *pipelines* / *workflows*.

Como já foi referido anteriormente, após obter os pontos essenciais ao cumprimento dos *quality gates*, a ideia seria implementar métodos nessa biblioteca de métodos (*scripts* em Jenkins) que foi referida antes para tentar automatizar a resposta a cada um dos requisitos fundamentais. Se já existirem métodos alusivos a determinada questão, então seria possível reutilizar os mesmos, adaptando-os se necessário. Caso ainda não existisse nenhum, seria adicionado a uma lista de melhorias a implementar futuramente, e estes viriam a ser desenvolvidos gradualmente.

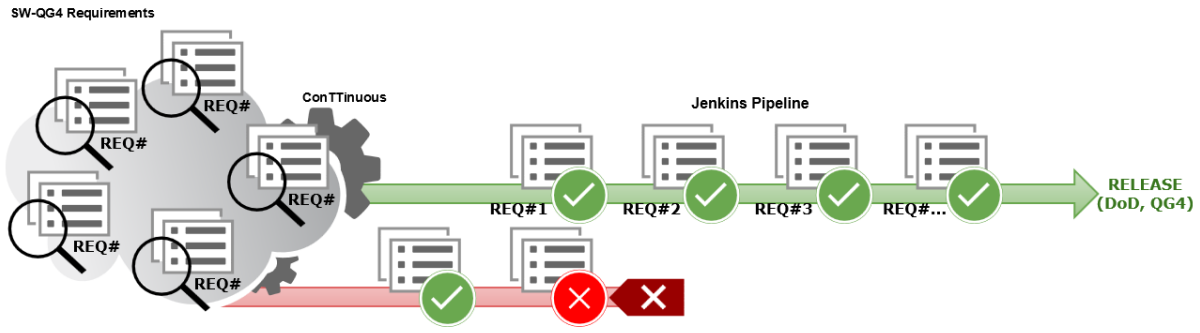

Figura 47 – **Ilustração de como se pretende cumprir com os requisitos base necessários para a garantia de qualidade de forma automática através de uma pipeline do Jenkins e com recurso às funções desenvolvidas presentes na biblioteca partilhada – se algum dos passos falhar, não é possível avançar com o lançamento do *software*.**

No final, a ideia era ter essa biblioteca do Jenkins a fornecer capacidade de resposta automática aos requisitos essenciais ao cumprimento dos *quality gates*, sob a forma de uma *pipeline* que teria de passar com sucesso em todas as etapas (realizadas pelas respectivas ferramentas), caso contrário, seria necessário corrigir os erros existentes ou completar a informação em falta, antes de chegar à *release* (Figura 47).

Para começar a desenvolver métodos, é importante perceber se para o método em desenvolvimento, vai haver interação com alguma ferramenta / *software*. Em caso afirmativo deve ser feita uma pesquisa junto da documentação técnica dessa ferramenta, em particular sobre como se comporta a REST API, de forma a perceber que funcionalidades os programadores dessa ferramenta disponibilizam.

Depois são feitos pedidos via REST API e testados, com recurso a uma ferramenta muito útil, a *Postman* (Figura 48). Esta ferramenta, como já se viu antes, permite enviar um pedido via REST, para receber informação de volta na resposta. Se o pedido for bem-sucedido, recebe-se uma resposta com código 200 SUCCESS e toda a informação que se pretende. Caso contrário, é necessário interpretar o tipo de erro ou problema retornado e proceder à sua correção.

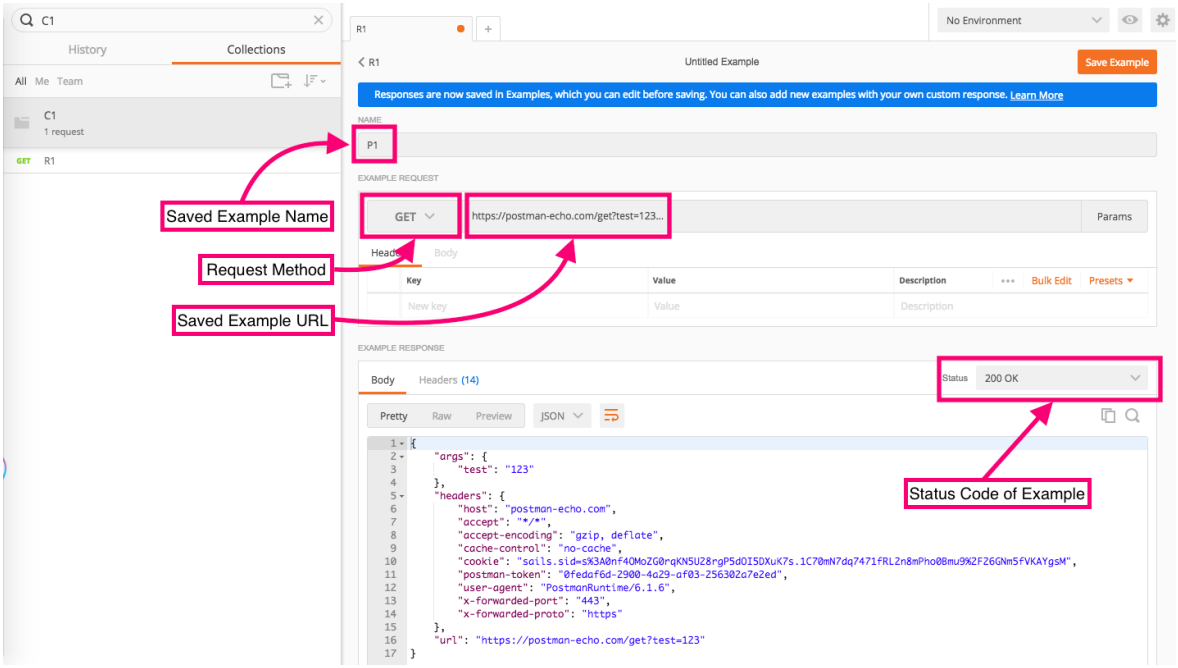

Figura 48 – **Exemplo do funcionamento do POSTMAN com um pedido e resposta HTTP** *[63].*

Após efectuar com sucesso o teste aos pedidos REST para a ferramenta em questão, passa-se à fase seguinte, o desenvolvimento da lógica e do código em *Groovy* (linguagem de programação que a ferramenta Jenkins irá interpretar e executar) que irá representar o método que se está a criar. Este desenvolvimento seria feito com recurso a um IDE, no caso foi utilizado o *IntelliJ IDEA*, da JetBrains, também utilizado para desenvolver em Java, aplicações e testes.

```
def exportHTML(confluenceCredentialsId, confluenceApiKeyId, confluenceURL, spaceKey, pageName, schemeSettingsPath) {

    withCredentials([string(credentialsId: "$confluenceApiKeyId", variable: 'secretApiKeyId')]) {
        withCredentials([[$class: 'UsernamePasswordMultiBinding', credentialsId: "$confluenceCredentialsId", usernameVariable: 'username', passwordVariable: 'password']]) {

            pageNameSpaceImmune = "$pageName".replaceAll(" ", "+")
            pageSearchResults = sh(returnStdout: true, script: "curl -D- -H 'KeyID: ${secretApiKeyId}' -X GET -k -u ${username}:${password} '… '

            pageJsonResponse = separateJsonFromHttpResponse(pageSearchResults)
            pageJsonContent = readJSON text: "${pageJsonResponse}"
            pageId = "${pageJsonContent.results.id[0]}".toInteger()

            def updateExportSchemeHTML = readJSON file: "${schemeSettingsPath}"
            updateExportSchemeHTML."rootPageId" = "$pageId" as String
            writeJSON file: 'updatedExportSchemeHTML.json', json: updateExportSchemeHTML, pretty: 4

            sh("curl -H 'Content-Type:application/json' -d '@updatedExportSchemeHTML.json' -J -O -k -u ${username}:${password} '… '

        }
    }
}
```

Figura 49 – **Exemplo de um método desenvolvido para a biblioteca de funções partilhada – este método tinha como intuito exportar a documentação definida no formato HTML**.

Na figura acima (Figura 49), como exemplo, está um método criado para efectuar a exportação da documentação pretendida em formato HTML a partir da plataforma *Confluence*.

É definido o nome do método e os parâmetros que este deverá receber para ser executado (no caso, as credenciais para autenticação na plataforma, uma chave para a API, o URL da plataforma, o espaço onde se encontra a documentação, o nome da página principal, e as definições para o esquema de exportação). De seguida, recolhe a informação relativa às credenciais e chave, mascaradas pelo Jenkins. É efectuado um pedido HTTP do tipo GET (verbo HTTP já descrito no capítulo anterior), para encontrar informação acerca da página definida por parâmetro, em particular o ID atribuído à página pela plataforma, necessário às definições de exportação. Finalmente esse ID é passado juntamente com as restantes definições num *template* JSON, através de um pedido específico fornecido pelos programadores da plataforma, que permite exportar em HTML a documentação.

Para cada método criado e colocado na biblioteca de Jenkins, foi criada documentação técnica (descrevendo os parâmetros e o funcionamento da função) e documentação prática exemplificando na prática o seu funcionamento (Figura 50, Tabela 2 e Figura 51).

exportHTML
Pre-Requisites
Enough read and edit accesses to allow the exporting of pages in a certain Docupedia space.

A Confluence API Key is needed for the user. In some cases, for Jenkins user. This can be requested via CI-Hotline ticket as described here: First Steps with Confluence API.

A temporary export scheme with whatever settings to be saved. The purpose of it will be to provide its ID, to be used on all further custom scheme settings. Then it can be deleted.

This function also calls one other functions in it, to separate the HTTP response header from the JSON data, called 'separateJsonFromHttpResponse'.

Figura 50 – **Exemplo de documentação acerca dos métodos desenvolvidos referente aos pré-requisitos necessários à utilização desses métodos**.

Tabela 2 – **Exemplo de documentação acerca dos métodos desenvolvidos referente aos parâmetros presentes em cada método e respectiva descrição**.

Parameters

Parameter	Mandatory	Type	Description
confluenceCredentialsId	YES	String	Your Confluence credentials ID, which is given by the Jenkins you stored them in. Usually the same credentials as portal.
confluenceApiKeyId	YES	String	Your Confluence API Key ID, requested from the CI as mentioned in the Pre-Requisites.
confluenceURL	YES	String	The Confluence URL. Careful not to use the root URL, but the alternative API Gateway Frontend as an additional security layer, described in: Confluence API Architecture.
spaceKey	YES	String	The space key from the Docupedia space where the pages you want to export as HTML are.
pageName	YES	String	The name of the parent page you want to export as HTML.
schemeSettingsPath	YES	String	The export settings you wish to use. These are contained in a JSON file.

This method is meant to export a section of your choice from Docupedia as HTML in a .zip file.

You pass the page name that will work as parent for the hierarchic section of pages you will be exporting. The method will then search that page and obtain the parent page ID to pass on the JSON scheme settings file.

There are all the additional configurations for the exporting. This method can be used along with the labeling mechanism which will allow filtering of the pages that are to be or not exported. Those label filters are also defined in the JSON scheme settings file.

Figura 51 – **Exemplo de documentação acerca dos métodos desenvolvidos que contém uma breve descrição sobre os mesmos**.

No final, estes métodos foram testados numa instância de Jenkins à parte (no caso específico foi utilizada a da equipa da qualidade que tinha esse propósito) e submetidos para aprovação por parte dos administradores da biblioteca do Jenkins, sendo então colocada num ambiente disponível a todas as equipas para que possam usufruir desses métodos.

5.1.6. Adopção de um Modelo Jenkins Eficiente

Para projectos simples, ou numa fase inicial, poderá ser suficiente e sustentável editar um ou dois *Jenkinsfiles*, cada um no seu respectivo projecto. No entanto, se se pensar numa escala maior, em equipas de desenvolvimento numerosas, com vários projectos ou módulos em mãos, e se o processo de garantia de qualidade a implementar em cada um destes for idêntico, não faz sentido editar ficheiro a ficheiro, pois seria pouco eficiente a nível de tempo e esforço.

Então pensou-se em criar uma camada de abstração para os verdadeiros *Jenkinsfiles*, de forma a que quando se altera um *Jenkinsfile*, esta alteração se possa reflectir em todos os projectos que fazem uso dessa configuração. Para recorrer a este método, o verdadeiro *Jenkinsfile* deve ser mantido à parte, sendo que nos projectos irá estar apenas um ficheiro *Jenkinsfile* (*loader*) que servirá para invocar o verdadeiro (*Jenkinsfile*).

Jenkinsfile.groovy

```groovy
node {

    def jenkinsfile_repository = 'Repository'
    def jenkinsfile_repository_branch = 'Repository Branch'
    def jenkins_ssh_id = 'SSH Key ID'
    def jenkinsfile_location = 'File Path'

    try {

        // Load Real Jenkinsfile

        jenkinsfile_project = fileLoader.fromGit("${jenkinsfile_location}", "${jenkinsfile_repository}", "${jenkinsfile_repository_branch}", "${jenkins_ssh_id}", '')
        jenkinsfile_project.jenkinsFileStart()

    } catch (error) {

        throw error

    }
}
```

Figura 52 – **Exemplo de um *Jenkinsfile* que fica ao nível dos projectos e invoca o verdadeiro *Jenkinsfile* que contém as configurações da *pipeline* do Jenkins**.

Como se pode ver, como exemplo, na figura acima, está presente um *Jenkinsfile* a funcionar como *loader* (Figura 52), o qual irá estar colocado dentro de um projecto. Este simplesmente aponta para o verdadeiro *Jenkinsfile* que está contido em outro local à parte (Figura 53).

Jenkinsfile.groovy

```groovy
def jenkinsFileStart() {

    // Load Library Dynamically

    JenkinsLib = library('Jenkins_Lib@master')

    // Import Required Modules

    commonsLib = JenkinsLib.com.bosch.tt.Commons.new()

    try {

        stage('Checkout') {
            checkout scm
        }

        stage('Import Results') {
            commonsLib.importXrayResults('Credentials ID', 'XRAY URL', 'File Path', 'Project Key', 'Test Execution Key', 'Test Plan Key', 'Test Environment', 'Revision')
        }

    } catch (error) {

        throw error

    }

}

return this
```

Figura 53 – **Exemplo de um *Jenkinsfile* que contém as configurações para uma *pipeline* do Jenkins e que faz uso dos métodos disponibilizados na biblioteca de funções partilhada**.

Por sua vez, o verdadeiro *Jenkinsfile* (a verde na figura seguinte - Figura 54), faz uso das bibliotecas partilhadas (no bloco cinzento), indo buscar métodos disponibilizados pelas mesmas.

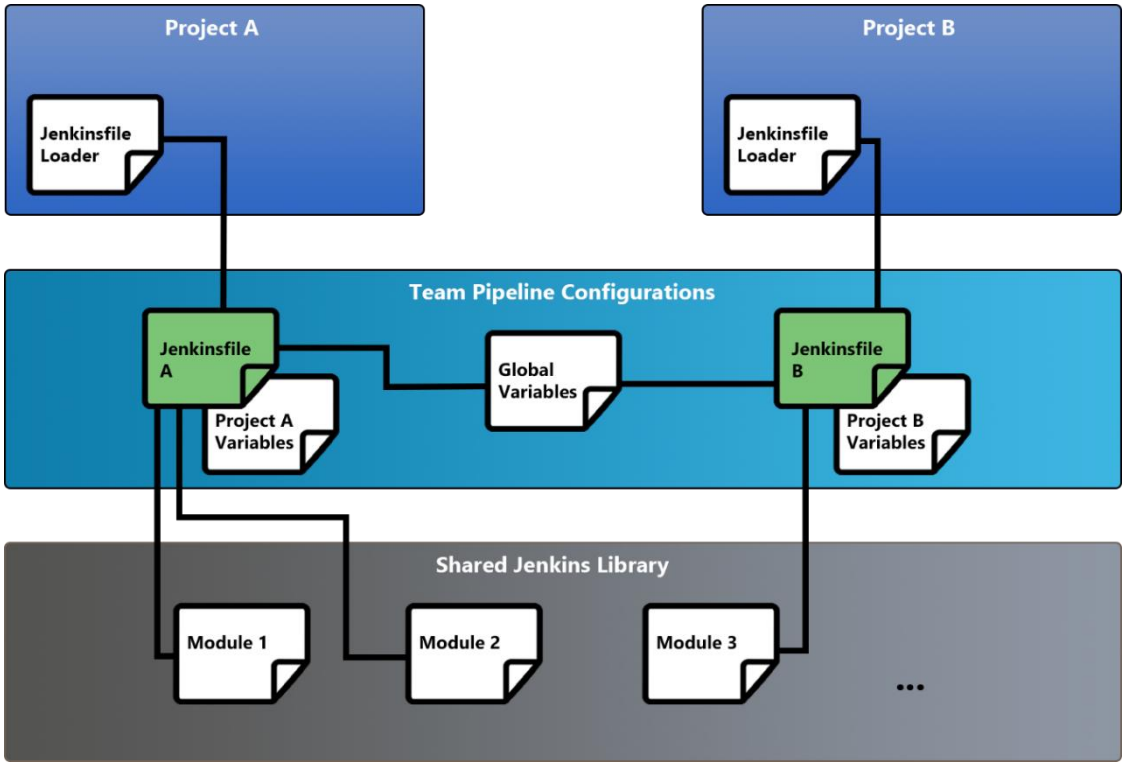

Figura 54 – **Modelo Jenkins com uma camada de abstração e forma como se relaciona com os projectos e com a biblioteca de funções partilhada**.

Para activar estas funcionalidades, foi necessário instalar alguns *plugins* para o Jenkins, nomeadamente o *Pipeline Remote Loader Plugin* (para o *Jenkinsfile loader*) e o *Pipeline Shared Groovy Libraries Plugin* (para as bibliotecas partilhadas), e efectuar a respectiva configuração.

5.1.7. Teste do Conceito com Recurso a um Projecto de Demonstração

Paralelamente, foi criado um projecto Demo, para que se pudesse colocar em prática os vários métodos desenvolvidos e o conceito de *release* automática de *software* numa *pipeline* do Jenkins, que seria activada com cada *commit* por parte dos programadores de *software* (*developers*), e teria como intuito, para além de efectuar uma demonstração, funcionar como modelo base a seguir pelas restantes equipas de desenvolvimento.

Figura 55 – ***Pipeline* do Jenkins para o projecto de demonstração em acção**.

Como é observável na figura anterior (Figura 55), a pipeline base do projecto de demonstração consistiu em 11 etapas, de entre as quais, a preparação do ambiente de trabalho (que garante a limpeza de eventuais ficheiros de execuções anteriores), a obtenção do código fonte, a compilação do mesmo, a execução dos testes, a análise estática do código, a verificação de conformidade dessa análise, a importação dos resultados dos testes para a plataforma de gestão, a colocação do artefacto no repositório destinado, a recolha de informações sobre o projecto, a exportação da documentação em HTML e a notificação às partes interessadas. Se todas as etapas forem concluídas com sucesso, então, em teoria, significa que há luz verde para a uma *release*.

5.1.8. Integração de Ferramentas e Criação de Tutoriais

Durante o desenvolvimento do conceito de *release* automática de *software*, da biblioteca partilhada de funções do Jenkins e do projecto de demonstração houve necessidade de integrar as várias ferramentas utilizadas (como *Bitbucket*, *Jenkins*, *SonarQube*, *Artifactory*). Todas essas integrações foram documentadas e deram lugar a extensos tutoriais para que qualquer elemento da empresa pudesse recorrer aos mesmos em caso de necessidade.

5.2. Mapeamento de Requisitos com Testes no *Xray*

Relativamente a testes ao *software*, foi feita uma pesquisa sobre a possibilidade de mapeamento de requisitos (*user stories*) com os resultados dos testes realizados, na

plataforma de gestão *Xray* inserida no JIRA, de forma a agilizar o trabalho dos programadores. Isto surgiu também como pedido por parte da equipa de desenvolvimento onde estive inserido.

Chegou-se à conclusão de que tal era possível adaptando os testes à *framework* de testes TestNG, compatível com testes implementados segundo a *framework* JUnit muito comum para testes unitários e de integração.

Fazendo uso das capacidades da *framework* de testes TestNG e da plataforma de gestão de testes *Xray*, é possível pensar em mapear automaticamente testes com os respectivos requisitos (*user stories*) a que pretendem dar resposta. Isto foi feito com a ajuda do TestNG, passando meta-dados adicionais aos métodos de teste no código fonte, que por sua vez serão passados como atributos no relatório de resultados dos testes. A plataforma *Xray* é depois capaz de processar a informação contida nos relatórios XML.

De forma a exemplificar, veja-se o que acontecia antes de implementar esta abordagem. Na imagem seguinte (Figura 56) pode-se ver um exemplo de requisito (*user story*) na plataforma de gestão de testes do *Xray*, sem qualquer teste associado. Aquando da importação dos testes para esta plataforma, estes por sua vez, também não tinham qualquer tipo de associação, e esta, tinha de ser feita manualmente e à vez.

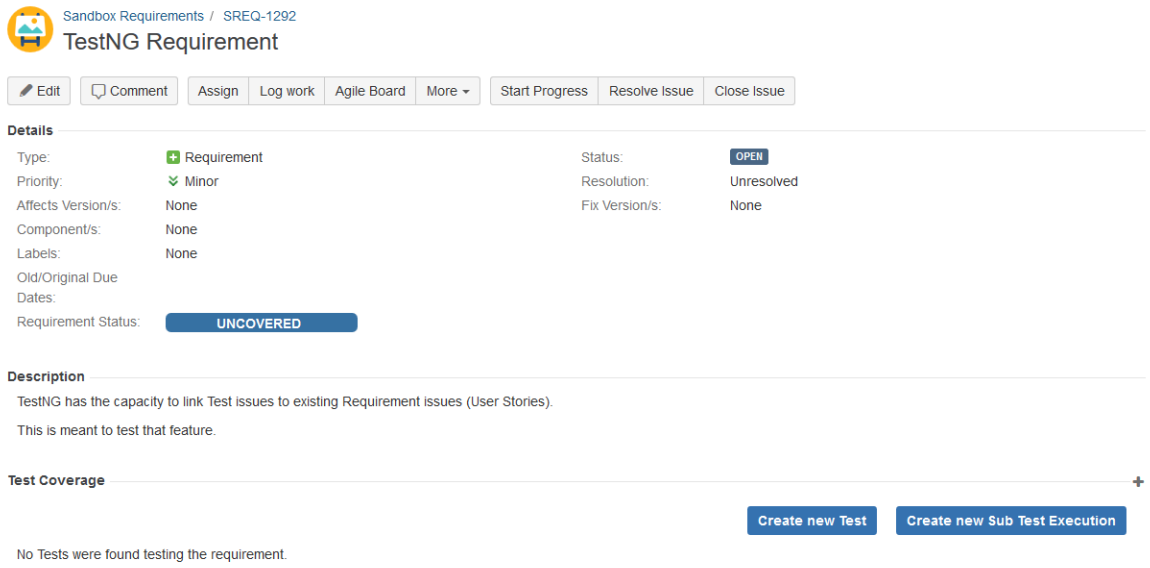

Figura 56 – **Requisito de demonstração sem qualquer Teste associado (na plataforma JIRA Xray)**.

Utilizando um projecto de demonstração, de uma calculadora, pode-se perceber o processo de mapear os requisitos aos testes automaticamente. Tem-se então a classe com o código fonte de uma simples calculadora sem interface (Figura 57):

Calculator.java

```
1  package com.     .tt;
2
3  public class Calculator {
4
5      private static int result;
6
7      public static int add(int arg1, int arg2) {
8          result = arg1 + arg2;
9
10         return result;
11     }
12
13     public static int subtract(int arg1, int arg2) {
14         result = arg1 - arg2;
15
16         return result;
17     }
18
19     public static int multiply(int arg1, int arg2) {
20         result = arg1 * arg2;
21
22         return result;
23     }
24
25     public static int divide(int arg1, int arg2) {
26         if (arg2 == 0) {
27             throw new ArithmeticException("Not possible to divide by 0.");
28
29         } else {
30             result = arg1 / arg2;
31
32             return result;
33         }
34     }
35
36 }
```

Figura 57 – **Classe com o código fonte para demonstração correspondente às funcionalidades aritméticas básicas de uma calculadora**.

Para testar esta classe, foi criada uma classe de testes, que contém os métodos de teste que comprovam que o código fonte da aplicação está a funcionar como pretendido (Figura 58). Esses métodos podem ser declarados através do TestNG ou do JUnit. A *framework* TestNG é mais poderosa pois oferece mais funcionalidades, no entanto, JUnit é a mais popular e pode ser problemático passar grandes quantidades de testes para outra

framework, então, para evitar isso, é possível adaptar os métodos de teste em JUnit para serem lidos e reportados pelo TestNG. Neste exemplo foram escritos através do JUnit.

CalculatorTest.java

```
1  public class CalculatorTest {
2
3      private Calculator classUnderTest;
4
5      @Before
6      public void setUp() throws Exception {
7          this.classUnderTest = new Calculator();
8      }
9
10     @After
11     public void tearDown() throws Exception {
12         classUnderTest = null;
13     }
14
15     @Test
16     @Xray(requirement = "SREQ-1292", test = "SREQ-1210", labels = "Core Addition")
17     public void testAdd() {
18         Assert.assertEquals("The addition of 3 and 4 should result in 7.", 7,
       classUnderTest.add(3, 4));
19     }
20
21     @Test
22     @Xray(requirement = "SREQ-1292", test = "SREQ-1206", labels = "Core Addition")
23     public void testSubtract() {
24         Assert.assertEquals("The subtraction of 11 to 17 should result in 6.", 6,
       classUnderTest.subtract(17, 11));
25     }
26
27     @Test
28     @Xray(requirement = "SREQ-1292", test = "SREQ-1208", labels = "Core Addition")
29     public void testMultiply() {
30         Assert.assertEquals("The multiplication of 7 and 8 should result in 56.", 56,
       classUnderTest.multiply(7, 8));
31     }
32
33     @Test
34     @Xray(requirement = "SREQ-1292", test = "SREQ-1207", labels = "Core Addition")
35     public void testDivide() {
36         Assert.assertEquals("The division of 16 by 4 should result in 4.", 4,
       classUnderTest.divide(16, 4));
37     }
38
39     @Test(expected = ArithmeticException.class)
40     @Xray(requirement = "SREQ-1292", test = "SREQ-1209", labels = "Core Addition")
41     // "The division of any number by 0 should return an error message."
42     public void testDivideByZero() throws Exception {
43         classUnderTest.divide(2, 0);
44     }
45
46 }
```

Figura 58 – **Classe de testes (contém um teste para cada método presente no código fonte) com realce dos elementos que farão a associação automática entre Testes e Requisitos na plataforma de gestão de testes**.

Como se pode ver (Figura 58), a informação adicional sobre os métodos de teste (como o requisito associado e etiquetas informativas) pode ser passada de uma forma fácil e elegante. Isto foi feito com recurso a uma classe personalizada (designada por *listener*).

Primeiro, é criada uma classe de anotações específicas para o *Xray* (Figura 59), de acordo com a documentação disponibilizada pela *Xpand-It* (programadores do *Xray*), onde são definidos 3 campos para passar como informação adicional em cada método de teste (*requirement*, *test* e *label*):

Xray.java

```
1  package com.     .tt.annotations;
2
3  import java.lang.annotation.Retention;
4  import java.lang.annotation.RetentionPolicy;
5
6  /**
7   * A Custom Annotation to inject additional information into a TestNG Test
8   */
9
10 @Retention(RetentionPolicy.RUNTIME)
11 public @interface Xray {
12
13     String requirement() default "";
14
15     String test() default "";
16
17     String labels() default "";
18
19 }
```

Figura 59 – **Classe com anotações específicas para o Xray**.

Depois é criada uma classe (*listener*) que servirá de interface e será responsável por processar as anotações do *Xray* e passá-las como atributos ao objecto ITestResult do TestNG (Figura 60):

XrayListener.java

```
1  package com.     .tt.annotations;
2
3  import org.testng.IInvokedMethod;
4  import org.testng.IInvokedMethodListener;
5  import org.testng.ITestContext;
6  import org.testng.ITestListener;
7  import org.testng.ITestResult;
8
9  public class XrayListener implements IInvokedMethodListener, ITestListener  {
10
11     boolean testSuccess = true;
12
13     public void beforeInvocation(IInvokedMethod method, ITestResult testResult) {
14         if(method.isTestMethod() && annotationPresent(method, Xray.class) ) {
15             testResult.setAttribute("requirement",
   method.getTestMethod().getConstructorOrMethod().getMethod().getAnnotation(Xray.class).requirement(
   ));
16             testResult.setAttribute("test",
   method.getTestMethod().getConstructorOrMethod().getMethod().getAnnotation(Xray.class).test());
17             testResult.setAttribute("labels",
   method.getTestMethod().getConstructorOrMethod().getMethod().getAnnotation(Xray.class).labels());
18         }
19     }
20
21     private boolean annotationPresent(IInvokedMethod method, Class clazz) {
22         boolean retVal =
   method.getTestMethod().getConstructorOrMethod().getMethod().isAnnotationPresent(clazz) ? true :
   false;
23         return retVal;
24     }
25
26     public void afterInvocation(IInvokedMethod method, ITestResult testResult) {
27         if(method.isTestMethod()) {
28             if( !testSuccess ) {
29                 testResult.setStatus(ITestResult.FAILURE);
30             }
31         }
32     }
33
```

Figura 60 – **Classe de interface que processará as anotações do Xray e as passa como atributos ao TestNG**.

Depois, esta classe personalizada (*listener*) pode ser invocada de pelo menos 3 formas diferentes, seja declarando como serviço, chamando no início da classe de testes ou declarando-o no ficheiro XML que serve de definições de teste (*test suite*).

O resultado deverá então ser visível no relatório de resultados dos testes (Figura 61), onde os 3 campos são passados como atributos, reconhecidos pelo *Xray*.

testng-results.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <testng-results ignored="0" total="5" passed="5" failed="0" skipped="0">
3    <reporter-output>
4    </reporter-output>
5    <suite started-at="2019-04-10T09:12:33Z" name="TestNG on JUnit" finished-
   at="2019-04-10T09:12:33Z" duration-ms="230">
6      <groups>
7      </groups>
8      <test started-at="2019-04-10T09:12:33Z" name="CalculatorTest Class Test" finished-
   at="2019-04-10T09:12:33Z" duration-ms="230">
9        <class name="com.bosch.tt.test.CalculatorTest">
10         <test-method signature="testDivideByZero() throws java.lang.Exception" started-
   at="2019-04-10T10:12:33Z" name="testDivideByZero" finished-at="2019-04-10T10:12:33Z" duration-
   ms="78" status="PASS">
11           <reporter-output>
12           </reporter-output>
13           <attributes>
14             <attribute name="test">
15               <![CDATA[SREQ-1209]]>
16             </attribute> <!-- test -->
17             <attribute name="requirement">
18               <![CDATA[SREQ-1292]]>
19             </attribute> <!-- requirement -->
20             <attribute name="labels">
21               <![CDATA[Core Addition]]>
22             </attribute> <!-- labels -->
23           </attributes>
24         </test-method> <!-- testDivideByZero -->
25         <test-method signature="testAdd()" started-at="2019-04-10T10:12:33Z" name="testAdd"
   finished-at="2019-04-10T10:12:33Z" duration-ms="37" status="PASS">
26           <reporter-output>
27           </reporter-output>
28           <attributes>
29             <attribute name="test">
30               <![CDATA[SREQ-1210]]>
31             </attribute> <!-- test -->
32             <attribute name="requirement">
33               <![CDATA[SREQ-1292]]>
34             </attribute> <!-- requirement -->
35             <attribute name="labels">
36               <![CDATA[Core Addition]]>
37             </attribute> <!-- labels -->
38           </attributes>
39         </test-method> <!-- testAdd -->
40   ...
```

Figura 61 – **Relatório gerado em XML com os resultados dos testes e com realce para os atributos passados**.

Este será depois importado para a plataforma de gestão de testes do *Xray*, levando consigo a informação adicional.

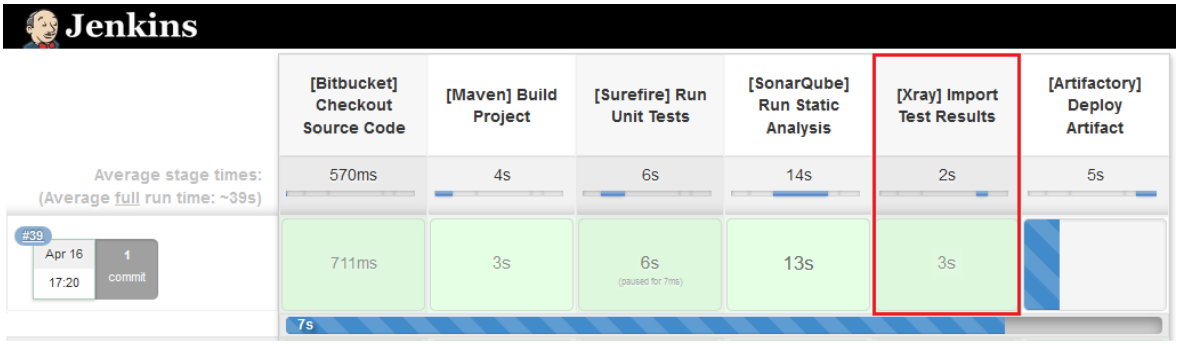

Figura 62 – **Momento em que o Xray processa o relatório de resultados, numa *pipeline* do Jenkins**.

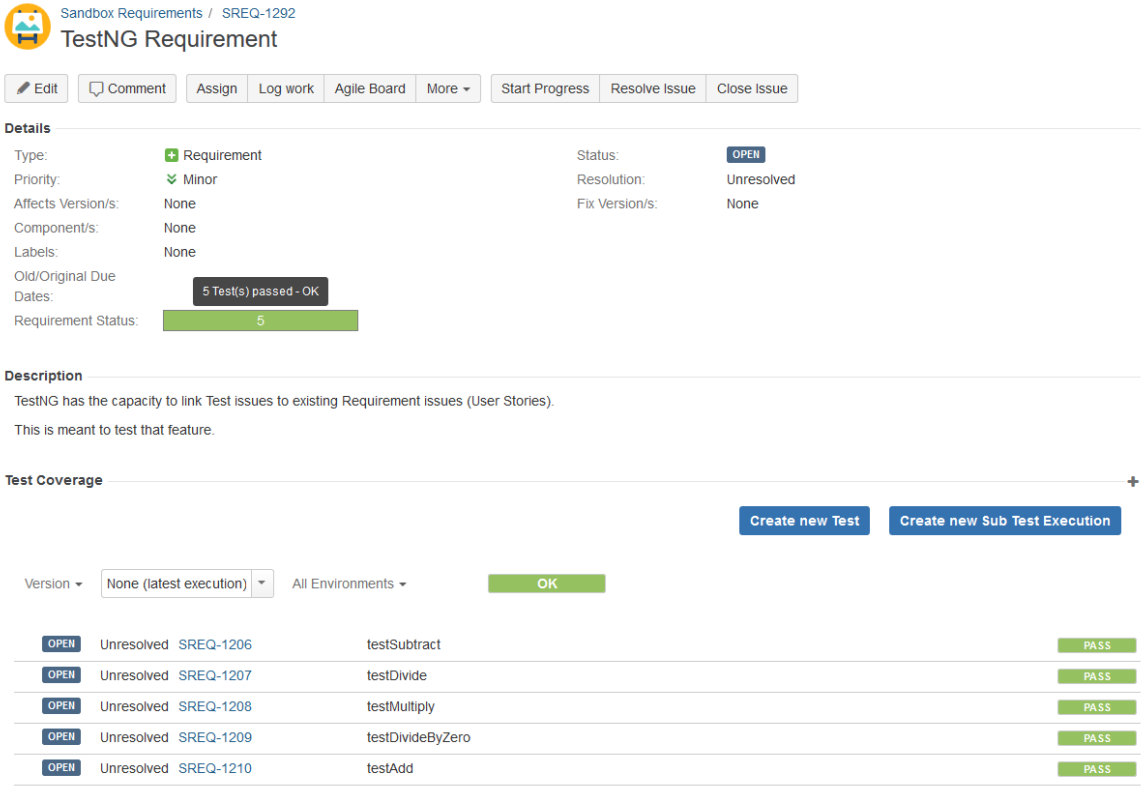

Figura 63 – **Requisito de demonstração com todos os Testes associados automaticamente (na plataforma do JIRA Xray)**.

E como se pode observar (Figura 63), a associação entre testes e requisitos é feita automaticamente, poupando tempo e esforço aos programadores, podendo dedicar-se a outras actividades.

Capítulo 6. Conclusões

O período de permanência na empresa no âmbito do estágio de mestrado, veio no seguimento de uma estadia anterior nesta mesma empresa, dedicados à automatização de testes em *software*. Assim, foi decidido dar continuidade ao trabalho anterior, desta vez abrangendo outras práticas, nomeadamente a de automatizar o processo de *release* de *software*, que, no fundo, é um agregado de várias tarefas associadas à garantia de qualidade no desenvolvimento de *software*. A garantia de qualidade no desenvolvimento de *software* é fundamental neste novo paradigma da Engenharia, onde a IoT e o conceito de Indústria 4.0 têm um papel relevante no alavancar da indústria do século XXI. Sendo eu Engenheiro Electrotécnico de formação, foi aliciante o despertar para estas novas abordagens que permitem a "ligação" entre o *hardware* e o *software* e que contribuíram para o enriquecimento pessoal e para aprofundar conhecimentos em outras áreas científicas.

Dada a minha anterior experiência na empresa, a familiarização com as instalações e com as equipas com as quais trabalhei, foi bastante rápida. Assim, o período inicial do estágio foi dedicado à preparação do ambiente de trabalho. Seguiu-se uma análise aos procedimentos levados a cabo pela empresa para que os seus produtos estejam em conformidade com os padrões de qualidade segundo a documentação e regras estabelecidas internamente e com base nessa análise partiu-se para a criação de um conceito de *release* automática de *software* para pequenas actualizações ou *patches*, mas com vista a alargar, eventualmente, a sua aplicação para grandes *releases*. Ao mesmo tempo, houve uma contribuição para o desenvolvimento de um projecto partilhado com vista a beneficiar todas as equipas, que consistia numa biblioteca de funções com o intuito de automatizar ao máximo alguns dos processos de desenvolvimento de *software* e garantia de qualidade. É importante salientar que todas estas acções foram documentadas sob a forma de um tutorial para posterior consulta. Para além disso foram prestados assistência e suporte às equipas do departamento, sempre que possível. Outras tarefas aliciantes foram ainda a participação e o apoio na preparação de eventos na empresa, como as *Tech Talks* e *Workshops*, com o objectivo de compartilhar o conhecimento adquirido.

Em jeito de conclusão, a ideia principal e globalizante passou por tentar demonstrar a importância dos bons hábitos e práticas de testar os produtos em desenvolvimento nas

várias fases desse processo, na tentativa de introduzir metodologias mais consistentes e optimizadas nas várias equipas, assim como promover a uniformização dentro do leque de ferramentas utilizadas sempre que possível. Isto poderia fomentar o fortalecimento e partilha de conhecimento envolto nessas ferramentas, que em última análise resulta num aumento da eficiência de recursos. No entanto, na adopção de novas medidas e tecnologias deve haver vontade e empenho de ambas as partes (geralmente dos elementos da equipa de garantia de qualidade e das equipas de desenvolvimento) em implementá-las, o que muitas vezes se revela um desafio, substituir práticas que já estão demasiado enraizadas e funcionais, mas que podem já estar ultrapassadas e onde há espaço para melhorar.

Foi possível, portanto, perceber a importância das actividades de garantia de qualidade de *software*, e em particular, o facto de estas serem uma responsabilidade de todos, ainda que existam pessoas qualificadas com a tarefa específica de auxiliar e direcionar na realização das mesmas. O trabalho em equipa e a constante partilha de conhecimentos são uma absoluta mais valia para o enriquecimento pessoal e para o crescimento da empresa.

Ao longo da investigação realizada durante este estágio/projecto, foi também possível perceber que as grandes empresas que promovem a qualidade têm vindo a apostar cada vez mais na aplicação da integração e entrega contínuas aos seus projectos, na automatização de processos e na filosofia de *DevOps* com vista a minimizar o tempo de produção e optimizar os recursos. A contínua tendência em aumentar o investimento nestas estratégias, comprova, com os resultados positivos que se têm vindo a obter assim como o valor acrescentado aos produtos desenvolvidos, que é o caminho certo para aumentar a eficiência a par da qualidade.

Além disso, tive a oportunidade de estar inserido num ambiente empresarial, que permitiu interagir com diversos colegas, adquirir experiência e desenvolver competências profissionais nas áreas de desenvolvimento de *software* e *testing*. Proporcionou também a aquisição/melhoramento das chamadas *soft skills* necessárias para o trabalho em equipa e no mercado de trabalho.

De um modo geral, este estágio foi muito enriquecedor para o meu conhecimento, pois entrei num mundo novo.

Referências

[1] Bosch, Bosch, [Online]. Available: http://www.bosch.pt/pt/pt/newsroom_11/news_10/news-detail-page_128576.php. [Acedido em 2018].

[2] Bosch, "Bosch today 2019," Bosch, Stuttgart, 2019.

[3] Bosch, "Bosch annual report 2018," Bosch, Sttutgart, 2019.

[4] Bosch, "Bosch em Portugal," 2019. [Online]. Available: https://www.bosch.pt/a-nossa-empresa/bosch-em-portugal/. [Acedido em Outubro 2019].

[5] M. Freire, "IoT e soluções de mobilidade impulsionam resultados da Bosch," business.IT, 2018.

[6] C. McClelland, "What is IoT?," Medium, 2017.

[7] J. Gubbi, R. Buyya, S. Marusic e M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," Melbourne, 2012.

[8] e-estonia, "Internet of Things paves the way for smart B2G solutions," Fevereiro 2018. [Online]. Available: https://e-estonia.com/internet-of-things-way-for-b2g-solutions/. [Acedido em 2018].

[9] K. Wieland, "IoT experts fret over fragmentation," Mobile World Live, 2016.

[10] M. Wallace, "Fragmentation is the enemy of the Internet of Things," Qualcomm - OnQ Blog, 2016.

[11] P. Pinto, "Bosch produz em Aveiro o mais avançado esquentador do mundo," *pplware,* 19 Março 2017.

[12] Classe Superior, "Como comprar caldeira ideal para a sua casa?," [Online]. Available: http://classesuperior.pt/primeira-dica/. [Acedido em 2019].

[13] Solar Condicionado, "Caldeiras e Esquentadores - Instalamos todo o tipo de caldeiras e esquentadores," [Online]. Available: http://www.solarcondicionado.pt/caldeiras-porto-povoavarzim.html. [Acedido em 2019].

[14] Worcester, Bosch Group, "Boiler Controls explained - What you need to know | Worcester-Bosch.co.uk," 2018.

[15] Bosch, "Bosch EasyControl - Smart Thermostat," [Online]. Available: https://www.worcester-bosch.co.uk/products/boiler-controls/bosch-easycontrol. [Acedido em Outubro 2019].

[16] Junkers, Bosch Group, "Junkers EasyControl CT 100," [Online]. Available: https://www.junkers.pt/consumer/produtos_consumidor/catalogo_consumidor/product_22016. [Acedido em 2019].

[17] P. Ammann e J. Offutt, Introduction to Software Testing, 2nd ed. ed., New York: Cambridge University Press, 2017.

[18] P. Isaias e T. Issa, High Level Models and Methodologies for Information Systems, New York: Springer, 2015.

[19] B. Tripathi, "10 Questions to Ask a Client When Developing Software," 2017. [Online]. Available: https://www.synotive.com/blog/software-development-client-questionnaire. [Acedido em 2019].

[20] tutorials point, "SDLC - Overview," [Online]. Available: https://www.tutorialspoint.com/sdlc/sdlc_overview.htm. [Acedido em 2019].

[21] D. Galin, "What is software quality?," em *Software Quality Assurance: From theory to implementation*, Essex , Pearson, 2004, p. 26.

[22] B. R. Maxim e M. Kessentini, "An introduction to modern software quality assurance," em *Software Quality Assurance: In Large Scale and Complex Software-Intensive Systems*, I. Mistrik, R. Soley, N. Ali, J. Grundy e B. Tekinerdogan, Edits., Waltham, Elsevier Inc, 2016, p. 19.

[23] P. Eeles, R. Bahsoon, I. Mistrik, R. Roshandel e M. Stal, "Foundations and approaches," em *Relating System Quality and Software Architecture*, Waltham, Elsevier Inc., 2014, p. 1.

[24] A. Neumeyer, "What I learned as a project manager at BOSCH - the German industrial giant," [Online]. Available: https://www.tacticalprojectmanager.com/reports/project-manager-bosch-experiences/. [Acedido em Outubro 2019].

[25] M. Hinshelwood, "Getting Started with a Definition of Done (DoD)," 8 Janeiro 2018. [Online]. Available: https://dzone.com/articles/getting-started-with-a-definition-of-done-dod. [Acedido em Junho 2019].

[26] S. Pittet, "Continuous integration vs. continuous delivery vs. continuas deployment," [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. [Acedido em Outubro 2019].

[27] B. Pepper, "Continuous Integration, Delivery, and Deployment: CI/CD," 24 Dezembro 2018. [Online]. Available: https://dzone.com/articles/continuous-integration-delivery-and-deployment-cic. [Acedido em Outubro 2019].

[28] G. Levin, "An API-First Development Approach," 15 Novembro 2016. [Online]. Available: https://dzone.com/articles/an-api-first-development-approach-1. [Acedido em Outubro 2019].

[29] R. T. Fielding, *REST Discussion Mailing List,* Yahoo Tech Groups, 2006.

[30] R. T. Fielding, "Representational State Transfer (REST)," 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Acedido em 2018].

[31] G. Svash, "How to Optimize URL Structure to Win Additional SEO Points," 2 Agosto 2017. [Online]. Available: https://rankactive.com/blog/how-to-optimize-url-structure-to-win-additional-seo-points. [Acedido em Outubro 2019].

[32] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," RFC (Request for Comments), 1999.

[33] The TechCave, "The HTTP and the Web | HTTP explained | Request-Response Cycle," 2016.

[34] T. Fredrich, "Learn REST: A RESTful Tutorial," [Online]. Available: https://www.restapitutorial.com/. [Acedido em Outubro 2019].

[35] RailsBridge Boston Job Board, "CRUD And Resourceful Routing," [Online]. Available: https://docs.railsbridgeboston.org/job-board/crud_and_resourceful_routing. [Acedido em 2019].

[36] Clever Techie, "REST API & RESTful Web Services Explained," 2017.

[37] Microsoft, "Microsoft Windows 10," [Online]. Available: https://www.microsoft.com/pt-pt/windows/. [Acedido em 2019].

[38] Canonical, "Ubuntu," [Online]. Available: https://ubuntu.com/. [Acedido em 2019].

[39] Oracle, "VirtualBox," [Online]. Available: https://www.virtualbox.org/. [Acedido em 2019].

[40] Git, "git --fast-version-control," [Online]. Available: https://git-scm.com/. [Acedido em 2019].

[41] Wikipedia, "Git".

[42] V. Driessen, "A successful Git branching model," 5 Janeiro 2010. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/. [Acedido em 2019].

[43] Atlassian, "Atlassian Bitbucket," [Online]. Available: https://bitbucket.org/. [Acedido em 2019].

[44] Atlassian, "Atlassian Jira Software," [Online]. Available: https://www.atlassian.com/software/jira. [Acedido em 2019].

[45] Atlassian, "Features for Software Development," [Online]. Available: https://www.atlassian.com/software/jira/features. [Acedido em 2019].

[46] Xpand IT, "Xray Documentation Home," [Online]. Available: https://confluence.xpand-it.com/display/public/XRAY/Xray+Documentation+Home?key=XRAY. [Acedido em 2019].

[47] Xpand IT, "Xray v2.0 Sneak Peak - Xray and JIRA Agile," 2016. [Online]. Available: https://www.xpand-addons.com/xray-v2-0-sneak-peak-xray-and-jira-agile/. [Acedido em 2019].

[48] Xpand IT, "Built-in Reports," 2017. [Online]. Available: https://confluence.xpand-it.com/display/public/XRAY/Built-in+Reports. [Acedido em 2019].

[49] SonarQube S.A., "Your Teammate for Code Quality and Security," [Online]. Available: https://www.sonarqube.org/. [Acedido em 2019].

[50] Checkmarx Ltd., "The Security Platform Built for CI/CD," [Online]. Available: https://www.checkmarx.com/. [Acedido em 2019].

[51] JFrog Ltd., "JFrog Artifactory - Enterprise Universal Repository Manager," [Online]. Available: https://jfrog.com/artifactory/. [Acedido em 2019].

[52] Atlassian, "Confluence - Team Collaboration Software," [Online]. Available: https://www.atlassian.com/software/confluence. [Acedido em 2019].

[53] JetBrains, "IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains," [Online]. Available: https://www.jetbrains.com/idea/. [Acedido em 2019].

[54] JUnit, "JUnit - About," [Online]. Available: https://junit.org/junit4/. [Acedido em 2019].

[55] C. Beust, "TestNG - Welcome," [Online]. Available: https://testng.org/doc/. [Acedido em 2019].

[56] The Apache Software Foundation, "Welcome to Apache Maven," [Online]. Available: https://maven.apache.org/. [Acedido em 2019].

[57] Jenkins, "Jenkins - Build great things at any scale," [Online]. Available: https://jenkins.io/. [Acedido em 2019].

[58] Postman, “Postman | The Collaboration Platform for API Development,” [Online]. Available: https://www.getpostman.com/. [Acedido em 2019].

[59] Microsoft, “Outlook: e-mail e calendário pessoais e gratuitos da Microsoft,” [Online]. Available: https://outlook.live.com/owa/. [Acedido em 2019].

[60] Microsoft, “Skype | Ferramenta de comunicação para chamadas e conversas gratuita,” [Online]. Available: https://www.skype.com/pt/. [Acedido em 2019].

[61] Rocket.Chat, “Rocket.Chat - Free, Open Source, Enterprise Team Chat,” [Online]. Available: https://rocket.chat/. [Acedido em 2019].

[62] T. Preston-Werner, “Semantic Versioning 2.0.0”.

[63] Postman, “Mocking with examples,” [Online]. Available: https://learning.getpostman.com/docs/postman/mock-servers/mocking-with-examples/. [Acedido em 2019].