



Pipeline de testes automatizados para integração e entrega contínua de software B2B em desenvolvimento Agile

Por

Luís Filipe da Costa Miranda Brochado

Orientador: Professor Doutor André Filipe Esteves de Sousa

Co-orientador: Engenheiro Andreas Carocha Vilela

Relatório de estágio submetido à
UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO
para obtenção do grau de
MESTRE
em Engenharia Informática, de acordo com o disposto no
DR – I série–A, Decreto-Lei n.º 74/2006 de 24 de Março e no
Regulamento de Estudos Pós-Graduados da UTAD
DR, 2.ª série – Deliberação n.º 2391/2007

Pipeline de testes automatizados para integração e entrega contínua de software B2B em desenvolvimento Agile

Por

Luís Filipe da Costa Miranda Brochado

Orientador: Professor Doutor André Filipe Esteves de Sousa

Co-orientador: Engenheiro Andreas Carocha Vilela

Relatório de estágio submetido à
UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO
para obtenção do grau de
MESTRE
em Engenharia Informática, de acordo com o disposto no
DR – I série–A, Decreto-Lei n.º 74/2006 de 24 de Março e no
Regulamento de Estudos Pós-Graduados da UTAD
DR, 2.^a série – Deliberação n.º 2391/2007

Orientação Científica :

Professor Doutor André Filipe Esteves de Sousa

Professor Auxiliar Convidado do
Departamento de Engenharia
Escola de Ciência e Tecnologia
da Universidade de Trás-os-Montes e Alto Douro

Engenheiro Andreas Carocha Vilela

CEO @ Izilabs Software & Product owner @ YUGOUP do
Regia Douro Park
Vila Real
Trás-os-Montes

“Technology is just a tool. In terms of getting the kids working together and motivating them, the teacher is the most important”

B. Gates (1955 -)

“I think that it’s when we step out of the road, step outside the box, become our own person, and we walk fearlessly down paths other people wouldn’t look at, that true progress comes. And sometimes true beauty as well.”

J. McAfee (1945 -)

À minha família e amigos

Pipeline de testes automatizados para integração e entrega contínua de software B2B em desenvolvimento Agile

Resumo — No desenvolvimento de software de acordo com a metodologia *Agile*, a satisfação do cliente é a principal prioridade. Assim sendo, a entrega contínua de software funcional, compatível com os requisitos e em formatos confiáveis, é o que caracteriza esta técnica de gestão de projetos.

Uma vez que os objetivos da garantia de qualidade vão de encontro aos pressupostos dos princípios do manifesto *Agile*, a implementação de uma *pipeline* de integração e entrega contínua pode ser uma das soluções para dar resposta às necessidade de adaptação a constantes mudanças sentidas pelas empresas. Nos casos em que este tipo de práticas são comuns verifica-se efetivamente um aumento da resiliência.

A existência de uma *pipeline* de entrega contínua é bastante benéfica na medida em que permite possuir um ecossistema de apoio ao desenvolvimento, com um sistema de controlo de versões, um orquestrador de processos que, por sua vez, vai coordenar uma ferramenta de integração contínua que realiza análise estática, testes unitários e testes de integração. Posteriormente, um sistema de transição de estados, atualiza as fases dos projetos de *development* para *staging*, *pre-live* e *live*. Todas estas ferramentas e automatizações têm como principais objetivos o suporte no desenvolvimento de software com qualidade, a otimização do tempo de desenvolvimento e, sempre que acontecem problemas, o apoio na tomada de decisão para aumentar a celeridade da resposta.

Keywords: Quality assurance, Integração contínua, Entrega contínua, DevOps, Gestão de projetos, Agile.

Automated testing pipeline for B2B continuous integration and deployment of software in Agile environment

Abstract — In software development according to the Agile philosophy, consumer satisfaction is the priority. Therefore, the continuous delivery of functional software, compliant with the requirements in reliable formats, supports this technique of project management.

Once the focus of quality assurance meets the assumptions of the principles of the Agile manifesto, implementing a pipeline of continuous integration and continuous delivery is one of the possible solutions to answer the needs felt by the organizations to constantly adapt to market changes. In most of the cases, these practices are verified to improve resilience.

The existence of a continuous delivery pipeline brings much benefit since it supports the existence of an ecosystem with a version control system, a process orchestrator that coordinates a continuous integration tool that performs static analyzes, unit tests and integration tests. In another stage, a state transition system updates the phases of the projects from the testing phase, to staging, pre-live and live. All these tools and automations combined have the purpose of supporting high quality software development, the optimization of the development process and, whenever problems occur, support decision makers to improve their response.

Keywords: Quality assurance, Continuous integration, Continuous delivery, DevOps, Project Management, Agile.

Agradecimentos

Desde 2012, esta jornada revelou-se gratificante e retributiva. O sentimento de conclusão do meu curso é de completa satisfação e de dever cumprido, nos níveis pessoal e académico, graças à Universidade de Trás-os-Montes e Alto Douro. Um enorme obrigado a esta grande instituição e votos de continuação na formação de estudantes e seres humanos de excelência.

Uma palavra de agradecimento em especial a toda a minha família, que sempre me apoiou incondicionalmente desde o primeiro dia. Aos meus pais, Fátima e Rui e à minha irmã Andreia, que fizeram grandes esforços para que tudo isto fosse possível. Um muito obrigado pela paciência, amor e compreensão.

A investigação não seria possível sem a ajuda indispensável da Izilabs Software. Endereço os meus cumprimentos ao Andreas Vilela, ao Álvaro Almeida, ao António Botelho e ao José Guimarães por toda a ajuda dispensada. Endereço também os meus cumprimentos ao Professor Doutor André Sousa não só pela orientação e revisão técnica mas também pela confiança, determinação e objetividade durante e após os desenvolvimentos da dissertação. O seu conhecimento, experiência e persistência foram cruciais ao longo do último ano. Um muito obrigado a todos e votos de sucesso e felicidade.

Em último lugar - mas não menos importante - gostaria de agradecer a todos

os docentes da ECT (Escola de Ciência e Tecnologia), assim como aos membros da Reitoria da Universidade de Trás-os-Montes e Alto Douro que conheci ao longo do meu percurso académico. Endereço os meus cumprimentos a todos aqueles que depositaram confiança nos projetos da UTAD Solutions Consulting durante o ano letivo transato. Muito obrigado a todos aqueles que são merecedores de uma menção honrosa pelo acompanhamento e progressão ao longo dos últimos dois anos de pesquisa e desenvolvimento na Júnior Empresa.

UTAD,

Luís Filipe da Costa Miranda Brochado

Vila Real, 4 de Setembro de 2019

Índice geral

Resumo	ix
<i>Abstract</i>	xi
Agradecimentos	xiii
Índice de tabelas	xix
Índice de figuras	xix
Glossário, acrónimos e abreviações	xxiii
1 Introdução	1
1.1 Contexto e Enquadramento	1
1.1.1 A cultura de DevOps	2
1.2 <i>Izilabs Software</i>	4
1.2.1 <i>Yugoup – spinoff</i>	5
1.3 Motivação e Objetivos	5
1.4 Organização do Documento	6
2 Estado da Arte	9
2.1 Hábitos comportamentais	9
2.2 Práticas correntes	11
2.2.1 Integração Contínua	11
2.2.2 Entrega Contínua	14

2.3	Tecnologias utilizadas	17
2.3.1	Orquestração da <i>pipeline</i>	19
2.3.2	Orquestrador de processos	19
2.3.3	<i>Version Control System</i>	20
2.3.4	Análise estática	22
2.3.5	Repositório de artefactos	22
2.3.6	Orquestração de <i>containers</i>	23
2.3.7	Workflow da pipeline de CI/CD	25
3	Construção da <i>pipeline</i> de integração contínua	29
3.1	Estudo de viabilidade das <i>frameworks</i> de teste	29
3.1.1	<i>Arrange, Act & Assert</i>	30
3.1.2	Tempo de execução dos testes	31
3.1.3	Considerações	32
3.2	Construção da <i>pipeline</i> de integração contínua	32
3.2.1	Configuração do sistema de controlo de versões	33
3.2.2	Desenvolvimento de testes unitários	35
3.2.3	Publicação do código no sistema de controlo de versões	41
3.2.4	Orquestração com <i>Jenkins</i>	43
3.2.5	Configuração da <i>pipeline</i>	46
3.2.6	Integração dos serviços	48
3.2.7	Desenvolvimento da Web API	56
3.2.8	Publicação da Web API	67
3.3	Considerações	71
4	Aplicação do protótipo aos serviços da empresa	75
4.1	Serviço de Gestão de tarefas	75
4.1.1	Arquitetura da aplicação	76
4.1.2	Criação do <i>Dockerfile</i>	80
4.1.3	Criação do <i>docker-compose.yml</i>	81
4.1.4	Considerações	84
4.2	Plataforma Yugoup	85
4.2.1	Integração contínua do serviço <i>Tenant</i>	86
4.2.2	Desenvolvimento dos testes de integração e performance	86
4.2.3	<i>Pipeline</i>	88
4.2.4	Geração da imagem do serviço <i>Tenant</i>	93
4.2.5	Resultados	96
4.2.6	Considerações	97
4.3	Integração de sistemas de visualização	100
4.3.1	Sistema de visualização	100

4.3.2	Integração do sistema de visualização	101
4.3.3	Instalação do sistema de visualização	102
4.3.4	Configuração do sistema de visualização	102
4.3.5	Apresentação do estado das builds	103
4.3.6	Pontos críticos	105
5	Conclusão e trabalho futuro	109
5.1	Conclusão	109
5.2	Trabalho futuro	112
	Referências bibliográficas	115

Índice de figuras

1.1	DevOps nas empresas (Fonte: Webinar – Using VSM to Optimize DevOps Workflow)	3
2.1	<i>Workflow</i> de um <i>job</i> da fase de <i>development</i>	25
2.2	<i>Workflow</i> de um <i>job</i> da fase de <i>staging</i>	26
2.3	<i>Workflow</i> de um <i>job</i> da fase de <i>pre-live</i>	27
2.4	<i>Workflow</i> de um <i>job</i> da fase de <i>production</i>	27
3.1	Exemplo de uma classe de teste	30
3.2	Estrutura da solução	31
3.3	Explorador de testes do <i>Visual Studio 2017</i>	32
3.4	Consola de Git Bash	34
3.5	Menu de definições do utilizador do GitLab	35
3.6	Menu de criação do projeto no <i>Visual Studio 2017</i>	36
3.7	Biblioteca de <i>packages</i> do <i>Visual Studio 2017</i>	36
3.8	Classe <i>UnitTestingInXUnit</i>	37
3.9	Método <i>shouldReturnASum</i>	37
3.10	Método <i>Add</i>	37
3.11	Estrutura exemplo de um método	38

3.12	Atributos [InlineData()]	38
3.13	Resultado da execução dos testes	38
3.14	Estrutura completa de uma <i>Theory</i>	39
3.15	Estrutura da segunda <i>Theory</i>	40
3.16	Estrutura da classe Calculator após alterações	40
3.17	Estrutura da <i>Theory ShouldNotAddNULL</i> com os dois casos de teste .	41
3.18	Mostrador do <i>Test Explorer</i>	41
3.19	Mostrador do <i>Team Explorer</i>	42
3.20	Solicitador do SSH	42
3.21	<i>Plugin Credentials</i>	44
3.22	Configuração das credenciais de acesso	45
3.23	Caminho de instalação do MSBuild	45
3.24	Configuração da ligação ao GitLab	46
3.25	Menu de criação do <i>Personal Access Token</i>	47
3.26	<i>Source Code Management</i>	47
3.27	<i>Build Environment</i>	48
3.28	Primeira instrução do SonarQube	50
3.29	Última instrução do SonarQube	50
3.30	SonarQube – Quality Gate	51
3.31	Instruções de <i>build</i> dos projetos	52
3.32	Resultado dos testes unitários	52
3.33	Configuração do repositório NuGet proxy	54
3.34	Configuração do repositório NuGet Public	55
3.35	Configurações de segurança do servidor Nexus	55
3.36	Instrução de publicação do <i>package</i> para o Nexus	56
3.37	NuGet Package Manager	57
3.38	Estrutura do método de configuração do serviço	58
3.39	Estrutura do método de configuração da aplicação especificada	58
3.40	Estrutura do método <i>PostNumbers</i>	59
3.41	Interface do Swagger	60
3.42	Interface do Swagger com resposta 200 OK	60

3.43	Estrutura do <i>TestClientProvider</i>	62
3.44	Estrutura do <i>TestForResponseTypePOST</i>	63
3.45	Configuração da descrição e do <i>access token</i>	64
3.46	Configuração dos <i>Build Triggers</i>	65
3.47	Configuração dos <i>Build Steps</i>	65
3.48	Configuração do Jenkins CI	66
3.49	Configuração do Slack Notifier	66
3.50	<i>Post-build action</i> de Slack Notification	67
3.51	Estrutura do <i>Dockerfile</i>	68
3.52	Instrução de <i>build</i> à solução	69
3.53	Criação, publicação e lançamento da imagem da Web API	70
3.54	<i>Requests</i> à Web API	71
3.55	Instrução de paragem do <i>container</i>	71
3.56	Estrutura do ficheiro <i>docker-compose.yml</i>	72
4.1	Modelo do Serviço de gestão de tarefas	76
4.2	Organização do serviço em camadas	77
4.3	Organização dos projetos	77
4.4	Interface <i>Swagger</i> do serviço de gestão de tarefas	78
4.5	Estrutura da base de dados do <i>container</i>	78
4.6	Estrutura da classe <i>startup.cs</i>	79
4.7	Estrutura do script de <i>Docker</i>	80
4.8	Interface <i>Swagger</i> do serviço de gestão de tarefa	81
4.9	Serviço de base de dados da aplicação	82
4.10	Verificação da configuração do Serviço de Base de dados	82
4.11	Serviço de apresentação da Web API	83
4.12	Serviço de <i>load balancing</i>	84
4.13	Classe <i>ClientExtensions</i>	86
4.14	Teste à presença do <i>Default GUID</i>	87
4.15	Teste à presença de um <i>GUID</i> inexistente	87
4.16	Teste de acesso autorizado	88
4.17	Teste de acesso não autorizado	88

4.18	<i>Script de setup</i>	89
4.19	<i>Multiple Source Code Management</i>	90
4.20	Análise estática, <i>Restore</i> e <i>Build</i>	90
4.21	Compilação dos projetos de teste	91
4.22	Execução dos projetos de teste	91
4.23	Remoção das dependências	92
4.24	Compilação e execução dos testes de integração e performance	92
4.25	Dockerfile do serviço <i>Tenant</i>	93
4.26	Dockerfile do serviço <i>Tenant</i>	94
4.27	Dockerfile do serviço <i>Tenant</i>	94
4.28	<i>Connection String</i> da Base de Dados	95
4.29	Conexão ao serviço de base de dados pelo Azure Data Studio	95
4.30	Resultado dos testes	96
4.31	<i>Overview</i> da análise estática	98
4.32	<i>Screenshot</i> do serviço <i>Tenant</i> executado a partir do IDE	99
4.33	<i>Screenshot</i> do serviço <i>Tenant</i> executado a partir da imagem Docker	99
4.34	Ambiente de trabalho do <i>rasbpian</i>	103
4.35	Configuração da <i>view</i>	104
4.36	Estado do <i>job</i>	104

Glossário, acrónimos e abreviações

Initials	Expansion
API	<i>Application Programming Interface</i>
B2B	<i>Business-to-Business</i>
BAT	<i>Build Acceptance Tests</i>
BDD	<i>Behavioural Driven Development</i>
CD	<i>Continuous Deployment</i>
CEO	<i>Chief Executive Officer</i>
CI	<i>Continuous Integration</i>
DLL	<i>Dynamically-linked library</i>
DP	<i>Delivery Pipelines</i>
DevOps	<i>Development and Operations</i>
FTP	<i>File Transfer Protocol</i>
GUI	<i>Graphical User Interface</i>
GUID	<i>Global Unique Identifier</i>
HTTP	<i>Hypertext Transfer Protocol</i>
HTTPS	<i>Hypertext Transfer Protocol Secure</i>
IDE	<i>Integrated Development Environment</i>
IP	<i>Internet Protocol</i>
IT	<i>Information Technologies</i>

Initials	Expansion
ITIL	<i>Information Technology Infrastructure Library</i>
ITSM	<i>Information Technology Service Management</i>
LCD	<i>Liquid Crystal Display</i>
NOOBS	<i>New Out Of the Box Software</i>
ORM	<i>Object Relational Mapper</i>
PO	<i>Product Owner</i>
REST	<i>Representational State Transfer</i>
SCM	<i>Source Code Manager</i>
SDLC	<i>Software Development Life Cycle</i>
SME	<i>Small and Medium Enterprises</i>
SSH	<i>Secure Shell</i>
SSL	<i>Secure Socket Layer</i>
STLC	<i>Software Testing Life Cycle</i>
TDD	<i>Test Driven Development</i>
URL	<i>Uniform Resource Locator</i>
VCS	<i>Version Control System</i>
VSM	<i>Value Stream Management</i>



Introdução

A indústria está constantemente à procura de novas soluções para melhorar os seus processos produtivos com o objetivo de se colocar à frente dos seus competidores. Se os níveis de competitividade na indústria são elevados, fruto do desenvolvimento tecnológico e da melhoria constante dos processos produtivos, é natural que surja a necessidade de desenvolvimento e melhoria constante nas organizações do setor das tecnologias da informação. A comunicação da informação em tempo útil é fundamental para que qualquer organização tenha sucesso.

Existe a necessidade de melhorar constantemente a qualidade, a fluidez e a assertividade da comunicação. Da mesma forma exige-se melhor desempenho global das metodologias de desenvolvimento de novos processos, novos produtos e novas tecnologias.

1.1 Contexto e Enquadramento

Os enormes volumes de capital monetário, humano e de tempo investidos pelas grandes empresas tecnológicas em satisfação do consumidor, *quality assurance* e redução do tempo de entrega são, provavelmente, os fatores com mais peso nas

decisões tomadas pelos quadros executivos quando se trata de melhoria contínua dos processos produtivos (Goldratt e Cox, 2016). Esta é uma premissa que se verifica em qualquer setor dada a transversalidade e a presença crescente da tecnologia quer em processos industriais produtivos, quer na prestação de serviços de consultoria IT, como é o caso do desenvolvimento de software. O desenvolvimento de software pode ser estruturado de várias formas e pode seguir várias metodologias. Para qualquer caso de desenvolvimento, existem as etapas de *software development lifecycle* (SDLC) e de *software testing lifecycle* (STLC), onde todos os eventos que consomem mais tempo do que aquele que é considerado aceitável, que sejam sistematicamente repetidos, são processos que o *business* deve considerar automatizar.

Automação de processos não é um conceito novo na indústria. O mesmo se pode aplicar ao desenvolvimento e controlo de qualidade de software. O que acontece, de um ponto de vista objetivo, é que se quer salvaguardar o negócio tomando partido de um conjunto de vantagens competitivas que permitem economizar tempo no desenvolvimento e na entrega de soluções. Tanto na fase de **SDLC** como na fase de **STLC** o ideal será a automatização de todo o processo – excepcionando o de desenvolvimento de código – onde os testes e a validação das fases de *development*, *staging* e *production* é realizado de forma automática.

Segundo o manifesto *Agile* (K. Beck et al., 2001), para o desenvolvimento de software deve ter-se em conta que podem ser considerados quaisquer tipos de mecanismos que providenciem agilidade ao sistema, em termos de flexibilidade no reajuste aos requisitos do cliente. Seguidamente, caso venham a ser aprovados, serão incluídos no projeto. Isto inclui as práticas da cultura *DevOps* (ex 1.1).

1.1.1 A cultura de DevOps

O movimento DevOps, os seus princípios e práticas podem ser definidos de várias formas. Certos autores, como é o caso de Ebert et al. (2016), definem este conceito através de uma visão romântica, quase utópica: “*DevOps é desenvolvimento flexível e rápido, fornecendo processos de negócio. Integra eficientemente o*

DevOps is a good thing, right?



Figura 1.1 – DevOps nas empresas (Fonte: Webinar – Using VSM to Optimize DevOps Workflow)

desenvolvimento, a entrega e as operações, traduzindo-se numa ligação elegante e fluída destes três SILOS tradicionalmente separados". Outros autores, como Bass et al. (2015), descrevem este conceito através de uma visão racional, focada no propósito afirmando que: “*DevOps é um conjunto de práticas que têm como objetivo reduzir o tempo entre a implementação de uma mudança no sistema e a colocação dessa mesma mudança em produção normal, preservando elevada qualidade*”. Tem-se ainda, segundo o autor Hüttermann (2012) uma definição do conceito de DevOps mais prática e equilibrada, vejamos: “*O Termo DevOps é uma mistura de desenvolvimento (representando os software developers, incluindo programadores, testers e técnicos de quality assurance) e operações (representando os peritos que colocam o software em produção, incluindo os administradores de sistemas, administradores de bases de dados e os técnicos de redes). DevOps descreve práticas que agilizam o processo de entrega de software, realçando a aprendizagem transmitindo feedback, desde a produção ao desenvolvimento, melhorando o tempo do ciclo*”.

Com os devidos pressupostos, cada uma das explicações dadas acerca deste conceito podem ser consideradas como corretas. Isto acontece porque DevOps envolve várias áreas dentro do desenvolvimento (*Dev*) e operações (*Ops*). Faz-se menção à lógica de negócio, às camadas de apresentação – *back-end* e *front-end* – passando ainda pela segurança do sistema. Segundo o autor Gruver (2016), o que se pode afirmar com toda a certeza, é que *DevOps* deve ser definido pelos resultados. De

forma a que fique claro, *DevOps* pode ser definido pelo conjunto de normas culturais e práticas tecnológicas que conferem aos projetos um fluxo rápido de trabalho planeado desde – entre outras fases – o desenvolvimento através dos testes até às operações, enquanto é preservada confiabilidade, operação e segurança de classe mundial.

1.2 *Izilabs Software*

A *Izilabs Software* é uma empresa com base tecnológica localizada no *REGIA DOURO PARK*, em Vila Real. A empresa foca-se no desenvolvimento de aplicações web, aplicações móveis e de *desktop*, contabilizando vários projetos bem sucedidos, com mais de 5 milhões de downloads e mais de 35 milhões de sessões de utilizadores em todo o mundo, nos últimos anos.

A história da *Izilabs Software* começou em 2010, quando o seu *CEO* Andreas Vilela, ainda estudante, desenvolveu um jogo para *Windows phone*, chamado “Kill the Duck”, que ficou classificado em segundo lugar na lista de transferências da loja de aplicações, tornando-se no melhor ranking alguma vez obtido por uma aplicação desenvolvida por estudantes. O jogo contabiliza mais de 1.5 milhões de downloads desde 2010.

Em 2012, o Andreas Vilela estabeleceu uma parceria com a Microsoft Portugal com o objetivo de publicar uma aplicação para *Windows 8* na *Windows PC Store*, chamado “Background Wallpapers HD”. Este aplicativo está presente na lista das melhores aplicações grátis desde 2012, conta com mais de 5 milhões de downloads e aproximadamente 250 000 utilizadores mensais ativos desde a última leitura. A marca *Izilabs Software* foi protegida em 2012 e foram desenvolvidas mais 10 aplicações móveis.

Mais tarde, em 2015, a *Izilabs Software* foi convidada para colaborar no desenvolvimento do “MB WAY”. Esta aplicação móvel de pagamento permite que

os utilizadores façam pagamentos e transferências através da utilização dos seus números telefónicos, que estão associados às suas contas bancárias.

Enquanto que as experiências e projetos da *Izilabs* foram relacionados com soluções B2C (business to consumer), a empresa decidiu entrar no mercado das soluções B2B. No começo do ano de 2016, o Fernando Novais foi convidado como membro do *REGIA DOURO PARK* para conhecer o Andreas Vilela e, em conjunto, formaram um novo projeto (**Yugoup** – Let’s grow up! – *Digital marketing platform*).

1.2.1 *Yugoup* – *spinoff*

A plataforma **Yugoup** tem como missão trazer as *SME* – *Small and Medium Enterprises* para o mundo digital, tornando o marketing digital acessível a todas as organizações. A plataforma conta com seis principais áreas: *Web*, *Mobile Apps*, *E-Commerce*, *Digital Marketing*, *Social Networking* e *Media*.

A plataforma foi construída para considerar os seguintes atributos: Combate à iliteracia tecnológica, Simplicidade, Automatização, Serviço universal, Integração, *User Friendliness*, Dinamismo, Economia digital, Comércio.

1.3 Motivação e Objetivos

Este estágio vai consistir na construção do protótipo de uma *pipeline* de integração e entrega contínua de forma a permitir avaliar as vantagens de uma implementação deste tipo tendo em vista a comparação com o processo atual de desenvolvimento de software da empresa.

A introdução de *pipelines* de integração e entrega contínua no desenvolvimento de software requer investigação prévia e realizada de forma periódica. Os autores Eddy et al. (2017), provaram que a curva de aprendizagem deste tipo de ferramentas depende muito do grau de conhecimento que cada um possui. Assim sendo, o primeiro objetivo será compreender – através de uma pesquisa alargada – resumir

e criar uma base de conhecimento que permita dotar todos os *developers* sobre o comportamento das *pipelines*, *DevOps*, e *CI/CD*.

Na fase de construção do protótipo, durante o desenvolvimento dos processos que vão compôr a *pipeline*, é necessário ter em atenção o contexto das políticas da empresa. Portanto, os segundos e terceiro objetivos são: assegurar que a *pipeline* de testes está sob monitorização permanente e garantir que o *output* do resultado da *pipeline* é útil para a empresa em termos de informação apresentada aos *developers*.

Em termos de *feedback*, Dunne et al. (2015) provaram que os defeitos com menos impacto nas aplicações são, na maioria dos casos, encontrados pelos utilizadores. Por outras palavras, pode existir a necessidade de se optar por desenvolver um mecanismo de feedback retroativo entre clientes e a equipa de desenvolvimento. Contudo, não parece ser esse o caso. Para uma empresa que tem como objetivo reportar *code smells*, *bugs* e vulnerabilidades no sistema (para que possam ser rapidamente identificadas e resolvidas) vai ser feita análise estática, vão ser executados testes unitários e testes de integração e performance para mitigar a probabilidade de ocorrerem falhas no sistema. Posteriormente será criada e publicada uma versão funcional da aplicação à qual serão adicionados um conjunto de serviços, desde serviços de bases de dados até a serviços de *load balancing*.

1.4 Organização do Documento

Este documento é composto por cinco capítulos e foi escrito tendo em vista a produção de mais um bloco documentado de apoio para os colaboradores da *Izilabs*. É uma base de documentação do processo de desenvolvimento, integração e entrega contínua de um serviço da plataforma, pretende aumentar o conhecimento em temas como *DevOps* e orquestração de micro-serviços tendo como objetivo providenciar uma nova visão e um conjunto de técnicas passíveis de serem aplicadas a novos projetos. No presente capítulo, foi apresentado o contexto e o enquadramento deste documento, a empresa onde o estágio será realizado, seguido pela motivação, pelas contribuições e pelos objetivos para o desenrolar do estágio.

No segundo capítulo é feito um levantamento ao estado da arte em três domínios, o domínio comportamental, o domínio das práticas correntes e o domínio das tecnologias utilizadas. São apresentados os conceitos-chave sobre os quais o trabalho será fundamentado e são analisados os pontos críticos do estado da arte. Neste capítulo é ainda introduzida a teoria de *pipelines*, é feita uma breve introdução a cada um dos componentes utilizados na configuração da pipeline e é apresentada uma alternativa para cada um dos componentes, uma vez que o foco é a configuração e a construção da *pipeline*.

No terceiro capítulo é apresentado um estudo de viabilidade das *frameworks* de teste. São apresentadas as tecnologias utilizadas para realizar o estudo, são apresentadas as metodologias de organização e estruturação dos testes e é medido o tempo de execução para cada uma das *frameworks* utilizadas. Em seguida é configurada a *pipeline* de integração e entrega contínua. É abordado o desenvolvimento dos testes unitários em termos de funcionamento, estrutura e processos de validação. Por fim é apresentado um modelo de **Web API**, que será publicado após a construção de uma versão de *Release*.

O quarto capítulo está dividido em três fases, começando pelo o serviço de gestão de tarefas seguido pela aplicação da *pipeline* a um serviço da empresa e terminando com a integração de um sistema de visualização no ciclo de desenvolvimento. Na primeira fase é apresentada a arquitetura base do serviço e é criada uma imagem com todos os componentes necessários para o funcionamento da aplicação. Depois é composto um ficheiro com todos os serviços que a aplicação necessita e são analisados os pontos críticos deste tipo de metodologia de publicação. Na fase seguinte é criado um *job* para análise de um serviço componente da plataforma **Yugoup**. É abordada a questão de desenvolvimento de testes de integração e performance que, de seguida, são incluídos na *pipeline* juntamente com análise estática e os testes unitários já existentes. Ainda na segunda fase, é criada uma imagem da **Web API** do serviço que a empresa disponibilizou, são apresentados o *output* da *pipeline*, os resultados e os pontos críticos. Na última fase é apresentado um guia para a integração de um sistema de visualização no ciclo de desenvolvimento e teste de software em desenvolvimento *Agile*. Aqui é explicada a sua utilidade, são

apresentados os seus objetivos e, por fim, é discutida a sua otimização.

Por fim, no último capítulo, são tiradas as conclusões sobre o impacto global do estágio, sobre o trabalho desenvolvido na empresa e sobre os principais fatores associados à redução do *time-to-market*. São tecidas algumas recomendações relativamente à implementação do processo na empresa e, neste capítulo, são também apresentados alguns caminhos futuros sobre como é que a empresa pode continuar, a partir deste trabalho, a aumentar a qualidade do seu processo de desenvolvimento de software.



Estado da Arte

Nas *small and medium enterprises* (SME) do setor das Tecnologias da informação (IT), uma alteração ao software nas fases tardias de desenvolvimento é uma aposta que – no cenário mais pessimista – pode colocar tudo em jogo. Tendo em conta que as SME são a espinha dorsal da economia Europeia, surge um desafio do seu lado para dar resposta às necessidades dos consumidores de forma rápida e conectada (Dunne et al., 2015). É necessário apresentar uma solução de melhoria contínua para o típico consumidor que está a amadurecer tecnologicamente e exige cada vez mais qualidade dos serviços e produtos que consome.

2.1 Hábitos comportamentais

Quando se produz software, para dar resposta aos requisitos dos clientes, os longos períodos de espera entre *releases* são um risco que pode levar à perda de quota de mercado. A recolha e análise de feedback em tempo útil, sobre se o software – em desenvolvimento – vai de encontro aos requisitos de grandes números de utilizadores, é praticamente impossível. Com a introdução de metodologias *Agile*, o software a ser desenvolvido vai sendo testado muito mais cedo no processo de

desenvolvimento pelo cliente final e assim é possível analisar o seu comportamento e recolher feedback sobre a sua adequação em termos de solução. Assim sendo, a redução do *time-to-market* e dos *feedback-loops* vai permitir diminuir a discrepância entre aquilo que os *Product Owners* (PO's), *developers* e utilizadores entendem como boas ou más propostas de valor (Udd, 2016). Evidentemente, nenhuma organização, *developer* ou cliente quer investir tempo e recursos financeiros no desenvolvimento do produto errado. Para além disso, é reconhecido pela indústria de software que os ciclos de *releases* podem ser períodos tensos e frustrantes. Estes sentimentos são conduzidos pelo risco de ocorrerem falhas associadas ao processo de publicação de software para produção. Em projetos de software onde as *releases* são processos manuais intensivos, os ambientes que albergam o software são construídos individualmente, normalmente pelas equipas de operações sendo que, em muitos casos, a equipa de desenvolvimento também intervém nos processos de *deployment*. O software de terceiros que a aplicação necessita para funcionar é instalado, os artefactos das aplicações são copiados para o(s) ambiente(s) de produção e a informação sobre a configuração é copiada ou criada através da consola de administração dos *web servers*, *application servers* ou componentes isolados que integram o sistema. Só depois, os dados das referências são copiados e, finalmente, a aplicação é executada serviço a serviço, componente a componente. Os sentimentos de tensão e nervosismo estão presentes por razões claras. Citando os autores Farley e Humble (2010): “*Muitas fases podem correr mal neste processo. Se cada um dos passos não for executado de forma perfeita, a aplicação não vai ser executada corretamente. E, neste ponto, pode não ser claro – de todo – qual é o erro, ou qual foi o passo que falhou.*”

Esta é a principal razão pela qual, com o decorrer do tempo, os *deployments* devem tender para ser o mais automatizados possível, não só pela redução do tempo ocupado pelo processo, mas também pelo aumento da resiliência do sistema e pela sucessiva eliminação de falhas provocadas pelo fator humano. Quando não são totalmente ou maioritariamente automatizados, os erros podem (potencialmente) ocorrer sempre que um *deploy* manual for feito. A solução não passa só por perceber se os erros são ou não significantes. Mesmo com excelentes testes de *deployment* os bugs podem ser difíceis de encontrar. Quando os *deployments* não são automáticos o

risco de falha aumenta e perde-se a capacidade de serem repetidos de maneira exata – que leva a desperdícios de tempo – e, em alguns casos, resulta na impossibilidade de identificar o problema por se desconhecer o processo exato. Um *deployment* manual é uma tarefa complexa, que consome imenso tempo e envolve a colaboração de várias pessoas. É natural que, a cada passo, a documentação criada esteja incompleta ou desatualizada. Por outro lado, em *deploys* automáticos, os *scripts* desenvolvidos podem servir de documentação e vão estar sempre o mais atualizados possível. Se fosse de outra forma, o *deployment* não iria ser concluído com sucesso (Farley e Humble, 2010).

2.2 Práticas correntes

Ainda de acordo com os autores Farley e Humble (2010), integração contínua (CI – *continous integration*) permite-nos manter a aplicação funcional depois de iteração de novo código e o seu complemento, entrega contínua (CD – *continuous delivery/deployment*) dá-nos a possibilidade de lançarmos versões novas e funcionais do software, várias vezes por dia. Isto significa que o código produzido e colocado no repositório – no *branch* principal – está com qualidade para permitir que as aplicações sejam entregues em qualquer instante. Uma *delivery pipeline* (DP) compreende tanto *continuous integration* de código – numa primeira fase – como *continuous delivery e deployment* de software para ambientes de teste ou de produção.

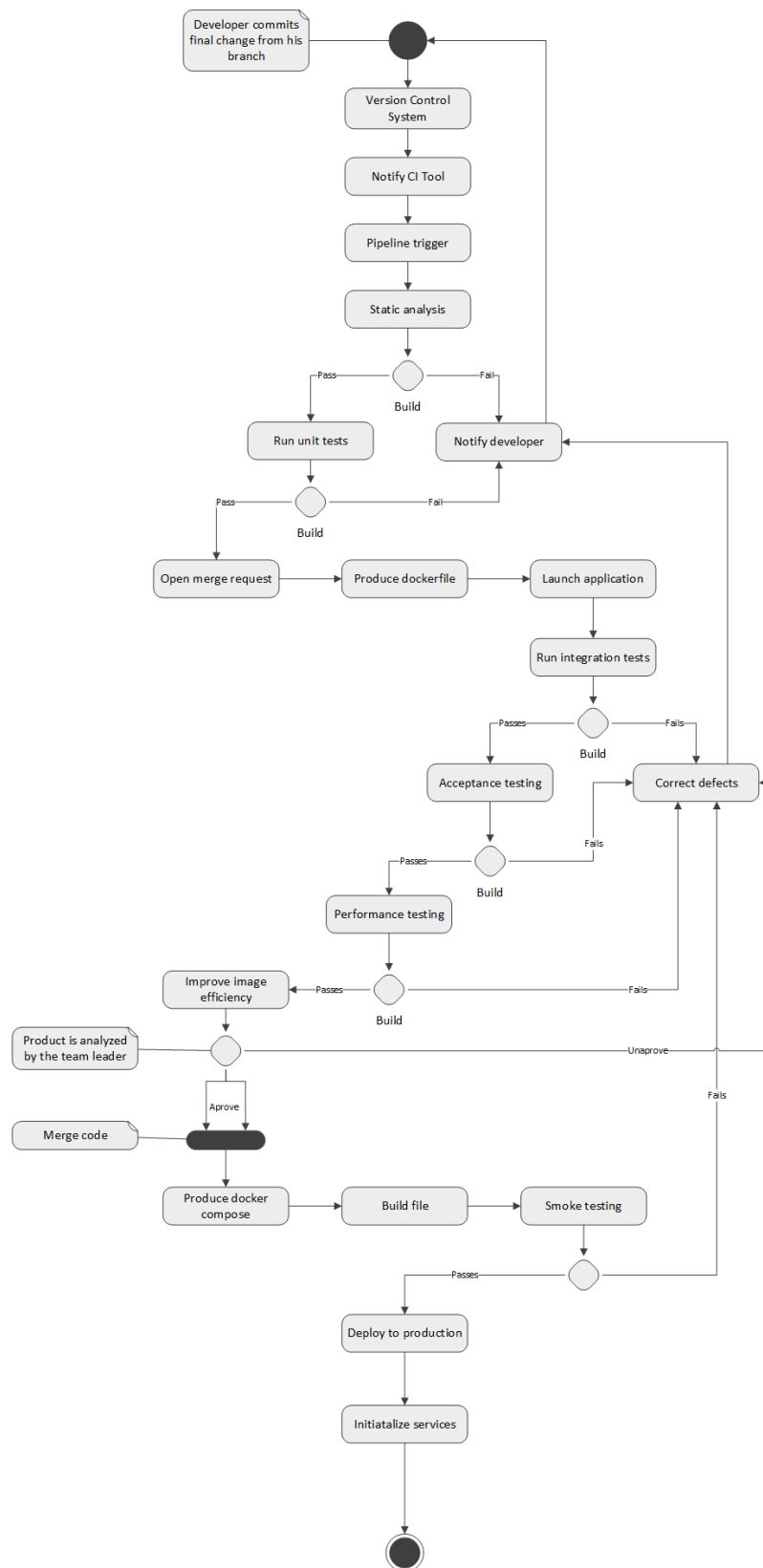
2.2.1 Integração Contínua

Uma das primeiras propostas de integração contínua foi feita por K. Beck e Gamma (2000) como uma das muitas práticas integrantes de *Extreme Programming* (<http://www.extremeprogramming.org/>). Integração contínua é o primeiro passo na implementação de uma DP e, apesar de não depender do tamanho da equipa de desenvolvimento de software, torna-se especialmente útil com o aumento do número

de *developers* a colaborar em conjunto. O objetivo desta mudança é a união de todos os códigos produzidos por eles, existindo um ponto central com o código final e a garantia de que estes módulos se integram devidamente. Com a existência deste ponto central, com o código completo, é possível a implementação de processos automáticos que permitam validar desde processos básicos, como se o código é compilável sem erros, até a processos mais avançados, como testar se o código em si funciona como é esperado. Os resultados destes processos podem permitir ou impedir que o processo continue para fase seguinte. É dada prioridade às *builds* bem sucedidas e ao desenvolvimento de forma que nos permita integrar código sem quebrar o fluxo, ou seja, é recomendado que cada *developer* valide previamente o processo de junção do código antes de o partilhar com os restantes colegas, de forma a evitar que estes fiquem impedidos de trabalhar devido a falhas existentes. Esta mudança é um desafio cultural para equipas de desenvolvimento habituadas a métodos tradicionais. Segundo Gruver (2016): *“Até ser feito, não se consegue imaginar que alguma vez vá funcionar mas, assim que é feito desta forma, não se imagina trabalhar de outra forma. O desafio é assegurarmo-nos que as mudanças culturais ocorrem e que as equipas estão a abraçar esta nova forma de trabalhar.”*

Esta é uma prática que requer que os *developers* façam partilhas (*commit*) dos seus códigos para um sistema de controlo de versões (VCS) pelo menos uma vez por dia, como os autores Fowler (2006) sugerem. Depois do *commit* das mudanças no VCS, um orquestrador de processos inicia a *pipeline* de integração contínua para integrar as mudanças do código recebido com o resto do código, é disparado um *trigger* e é feita uma *build* que executa análise estática primeiro, para depois compilar e executar todos os testes unitários disponíveis (Udd, 2016). Um exemplo de uma *pipeline* de integração e entrega contínua (CI/CD) pode ser visto na figura 2.2.1.

Se, em qualquer fase da *pipeline*, existir uma falha durante o processo de *build* o seu estado é considerado como *failed*. Em todo o caso, o *developer* é informado acerca do resultado, quer através de feedback sob forma de notificação nos canais de comunicação da empresa (Slack/SMS, ou e-mail para notificações em tempo real), quer através do sistema de visualização integrado no processo de desenvolvimento.



Quando existe mais do que um *developer*, ou naqueles casos em que se está a desenvolver uma aplicação modular, a possibilidade de introdução de *branches* – uma espécie de réplica da linha principal – permite que seja realizado desenvolvimento de uma determinada funcionalidade de forma independente e facilita que o código seja depois unido à linha principal sem que outros programadores sejam forçados a aguardar a sua conclusão (Farley e Humble, 2010).

Fundamentalmente, a integração contínua exige a capacidade de manter a aplicação funcional depois de iterar novo código. Por forma a aumentar a qualidade do desenvolvimento do software, existem algumas abordagens ao desenvolvimento que podem ser úteis para aumentar a qualidade do produto final. Uma delas é o Desenvolvimento Orientado a Testes (*TDD – Test Driven Development*) proposto por Beck (2002), que propõe o desenvolvimento orientado a testes como sendo a ideia de que, quando estamos a desenvolver novas funcionalidades ou corrigimos erros, os *developers* criam primeiro um teste que é uma especificação executável para o comportamento esperado do código a ser escrito. Estes testes motivam o *design* das aplicações servindo de testes de regressão e, simultaneamente, de documentação de código e do comportamento esperado da aplicação (Farley e Humble, 2010).

2.2.2 Entrega Contínua

Entrega contínua exige que sejamos capazes de fazer *deploys* da aplicação a qualquer momento tanto para ambientes de teste como para ambientes de produção (Farley e Humble, 2010). Uma das técnicas chave para mantermos as aplicações publicáveis a qualquer instante – mesmo apesar das mudanças constantes – é a modelação dos componentes da aplicação e utilização de interfaces de comunicação. Imagine-se um componente como um grande bloco de código dentro de uma aplicação (com uma Web API bem documentada) que pode eventualmente ser trocada por outra implementação desde que respeite a interface definida. Um sistema de software baseado em componentes é distinguido pelo facto de que a sua base de código está dividida em pedaços discretos que providenciam comportamento sob forma de interações limitadas, através dessas interfaces bem definidas, com outros componentes

(Farley e Humble, 2010).

A utilização de *pipelines* de integração contínua em produção e desenvolvimento de software, pode poupar muitas horas de trabalho. Partindo do código-fonte no **VCS**, construir a aplicação, executar análise estática, testes unitários, empacotar e publicar a aplicação para um repositório remoto, são processos que consomem algum tempo. Este tempo aumenta com a complexidade e tamanho da aplicação e, por vezes, é tempo que podia ser investido no desenvolvimento de uma funcionalidade em vez de desperdiçado na execução repetitiva de um conjunto de instruções.

De qualquer das formas, a automatização deste processo deve ser planeada de acordo com o tamanho e modularidade de cada aplicação. Para aplicações mais complexas, podem ser necessárias várias imagens para que a aplicação seja executada. Podem existir inúmeros *deployments* em múltiplos locais – dependendo da quantidade de fases existentes – e pode ser necessário ajustar o número de *deployments* dependendo da infraestrutura que a organização tem ao seu dispor.

Imagine-se um contexto em que um projeto exige a integração de código de cinco *developers* diferentes, sendo cada um responsável por funções diferentes dentro da *stack* de tecnologias a usar (BD, backend, frontend, etc). Se o *Product Owner* dá preferência à utilização das práticas de *DevOps* com o projeto em fase final, o projeto vai precisar não só de um ajuste em termos de estrutura organizacional, mas também de uma revisão geral de todos os processos. Neste caso em particular, a equipa está a desenvolver uma plataforma de *e-commerce* e as várias arquiteturas do sistema estão fechadas, sendo apenas necessárias pequenas correções em alguns serviços que a compõem. Aqui, evidentemente, dado o grau de complexidade do sistema em desenvolvimento e o tempo investido na sua modelação – não só em termos macro (plataforma), como também em termos micro (serviços) – é mais vantajoso manter as práticas correntes e continuar com o mesmo método de trabalho. Nem sequer se coloca em cima da mesa outro conjunto de fatores como por exemplo a barreira cultural da resistência à mudança. O projeto iria precisar de uma revisão geral tão extensa que todo o tempo de todos os integrantes da equipa teria de ser investido na revisão de metodologias de trabalho. Isto representaria largos

períodos de tempo que, obviamente, são cruciais para a finalização dos serviços que compõem a plataforma no imediato e que não podem sequer ser considerados em termos monetários.

Noutro contexto, dispõe-se exatamente da mesma equipa de *developers* a trabalhar no desenvolvimento de uma aplicação web de submissão de currículos. E um dos requisitos do *business*, com o projeto em fase de arranque, indica que o desenvolvimento será feito mediante as práticas da cultura *DevOps*. Neste caso, a equipa não tem qualquer tipo de ideia de qual será o alinhamento – em termos de arquitetura – do sistema. Aqui, contrariamente ao exemplo anterior, não tendo ainda sido investido tempo na modelação do sistema nem no seu planeamento, faz sentido a adoção das práticas de *DevOps* uma vez que a mudança na cultura de trabalho será feita a partir do começo do projeto, abrindo-se espaço a um processo gradual de adaptação.

A cultura *DevOps* é portanto um complemento fundamental ao desenvolvimento *Agile* e os benefícios das suas práticas são promissores. A implementação deste tipo de práticas traz imensas vantagens para o desenvolvimento de software em equipa. Em contrapartida, quanto mais tardia for a mudança para este conjunto de normas ou práticas diárias, mais difícil será a adaptação. A filosofia *Agile*, por si só, é muita das vezes considerada insuficiente ou ineficaz na gestão de projetos de desenvolvimento de software. Daí ser comum encontrar outros conjuntos de técnicas e estratégias de gestão aliados a esta filosofia, como é o caso do *Scrum* (<https://www.atlassian.com/agile/scrum>) para gestão do trabalho de vários elementos e do *Kanban* (<https://www.atlassian.com/agile/kanban>) para representação visual do fluxo de trabalho.

Considerando que as metodologias de gestão e planeamento devem contabilizar vários fatores internos e externos à equipa responsável pela execução do projeto, desde as políticas da internas da empresa até ao contexto do negócio, é importante ter em conta que a produção deste protótipo tem como objetivo ir de encontro à prestação de serviços *business-to-business* (B2B).

Naquilo que à prestação de serviços diz respeito, é o *Information Technology*

Infrastructure Library (ITIL) conhecido como um dos conjuntos detalhados de boas práticas que se concentra no alinhamento de serviços de IT com as necessidades dos negócios. Resumidamente, o **ITIL** baseia-se em quatro disciplinas – *service request*, *incident*, *change* e *problem*.

O **ITIL** atua sobre três pontos cruciais de *IT Service Management* (ITSM). São eles os processos, a comunicação e a transparência. Este conjunto de práticas *post-deployment* atua sobre **processos** pouco (ou nada) definidos e implementados com as ferramentas erradas. Atua sobre **comunicação**, em equipas que trabalham de forma isolada, que não se focam na comunicação fora do seu *SILLO* e atua sobre a **transparência** onde as equipas de clientes ou de parceiros nem sempre têm visibilidade dos itens de trabalho ou dos processos pertinentes.

Apesar de ser pouco comum a comparação entre filosofias e *frameworks* – como é o caso do *Agile* em contraposição com o **ITIL** – pode ser útil aliar uma filosofia, diga-se, anárquica a um conjunto de boas práticas bem definido, com atenção ao detalhe e com definição concreta no que diz respeito ao processo. Portanto, aliar estes dois pólos opostos e aplicar **ITIL** em processos cirúrgicos para estruturação das metodologias de trabalho pode ser vantajoso para tirar partido dos seus pontos fortes. Nos aspetos em que a filosofia *Agile* sai mais fragilizada pode eventualmente ser aplicado o **ITIL** para nutrir os processos com resiliência e para providenciar bases sólidas, bem documentadas, em melhoria contínua do serviço (Axelos, 2018).

2.3 Tecnologias utilizadas

A técnica de *pipelining* é muitas vezes comparada a uma linha de montagem industrial onde são produzidos vários tipos de componentes ao mesmo tempo, dentro de uma sequência lógica de acontecimentos, para um determinado processo produtivo. Mesmo existindo uma certa dependência em termos sequenciais, no global, o processo tira vantagem de operações que acontecem em paralelo.

O conceito de *pipelining* em engenharia informática advém de arquitetura de

computadores. Consiste numa técnica de decomposição de processos sequenciais de controlo em subprocessos, sendo que cada um desses subprocessos é executado num segmento especialmente dedicado que opera concorrentemente com os outros segmentos.

As primeiras aplicações práticas de *pipelining* em arquitetura de computadores surgiram no início da década de 80 e foram avanços extraordinários para a indústria de processamento digital de informação. Este tipo de arquitetura flexível – baseada em operações paralelas – foi adaptada aos requisitos funcionais de um conjunto de modelos de computadores, o que permitiu aumentar gradualmente a capacidade de processamento de informação. Estes avanços foram baseados na adição de novos componentes responsáveis principalmente pelo melhoramento da execução sequencial de instruções e pela mitigação e prevenção de *stack overflows* (Potash, Levin, e Genter, 1984). Mais próximo do final da década, aliado a outro conceito inovador, surgiu o primeiro computador multi-nó com processamento paralelo reconfigurável. Esta inovação tinha por base a computação feita através de vários nós. Foi impulsionada pelo desenvolvimento de uma topologia em *hypercube* e revolucionou a indústria da computação através da adição de novos componentes aos processadores – como os *multiplexers*, os *memory-ALU switches* e os mecanismos de *caching* – que eram orquestrados pelos micro-controladores e pelos micro-sequenciadores (Nosenchuck e Littman, 1989). Para termos uma base de comparação, imaginemos a descoberta dos autores Potash et al. (1984) como uma *pipeline* e a descoberta dos autores Nosenchuck e Littman (1989) como uma *pipeline de pipelines*.

Mais tarde, no início da década de 90, surgiram alguns dos princípios que ainda hoje são aplicados em computação sequencial. Através da administração de um *crossbar switch*, os elementos unitários de processamento e os módulos de memória paralelos passaram a poder ser alterados dinamicamente – ciclo após ciclo – de acordo com os requisitos de cada um dos algoritmos em execução. Esta inovação, composta por duas secções – um *multiplexer* e uma secção de controlo – trouxe melhorias principalmente para a área do processamento digital de imagem (Hiller et al., 1992). Depois, através da reintrodução de componentes estáticos

aliados aos componentes dinâmicos já existentes, surgiram métodos focados na otimização da performance e na conservação de energia. A introdução dos componentes estáticos era promissora por duas razões. Primeiro permitiria preservar informação que daria ao sistema a capacidade de resumir a execução dos processos mais tarde, depois de um *system clock* ser parado (*halted*), sem que os dados fossem perdidos. Segundo, para além da redução no consumo – comparativamente com um sistema implementado inteiramente em lógica dinâmica – também era minimizado o custo de operação, de produção e a área do circuito eletrónico (Donner, 1993). Ou seja, o processador ficava assim mais pequeno, a sua produção ficava menos dispendiosa e a sua operação mais sustentável.

2.3.1 Orquestração da *pipeline*

A *pipeline* faz orquestração sequencial de análise estática, testes unitários e testes de integração/performance do projeto. Depois é produzida uma imagem da versão executável do projeto que vai ser publicada num registo privado, onde vai ser armazenada. O último passo antes da passagem para produção é a criação de um ficheiro de composição de todos os serviços que a aplicação necessita para ser executada de forma automática através da execução de uma única instrução. Existe um conjunto de ferramentas cuja utilização é comum a todas as fases, como é o caso do **Jenkins** (orquestrador de processos) e do **GitLab** (VCS). Dependendo das fases de **SDLC**, ou **STLC**, vão ser utilizadas outras ferramentas para análise estática, repositórios de artefactos, ferramentas de virtualização em *containers* e registos privados de imagens.

2.3.2 Orquestrador de processos

Um orquestrador de processos é simultaneamente o coração e o cérebro da *pipeline* de integração e entrega contínua. Faz a automatização *end-to-end* do sistema. Para além puxar para si o código dos repositórios, executa instruções, faz a manutenção do código através da compilação do mesmo e da execução dos testes.

No fundo, pode – dependendo da complexidade das instruções – desempenhar uma panóplia de funções de automação.

O **Jenkins** (<https://jenkins.io/>) é o líder do mercado no que toca a servidores *open-source* de automação de projetos. Focado na automação contínua das *builds* e dos testes dos projetos, o **Jenkins** acrescenta valor através do aumento da produtividade e da integração dos *logs* de cada *build* na sua **GUI**.

Outros serviços de integração contínua, como o **Travis-CI** (<https://travis-ci.com/>) e o **Azure DevOps** (<https://azure.microsoft.com/pt-br/services/devops/>), requerem a criação de um *script* com todos os passos para definir os processos a ser executados pela *pipeline*. Para além da configuração em forma de *script*, estes serviços de integração contínua requerem um nível de conhecimento e à vontade mais elevados para poderem ser configurados de acordo com os objetivos definidos. Apesar de serem ambas opções viáveis devido à elevada quantidade de informação disponibilizada pela comunidade, o **Jenkins** será a ferramenta elegida uma vez que é *open source* e pode ser utilizado para a apresentação da informação dos estados das builds *on the fly*. Comparativamente com outros servidores de automação, como é o caso do **GitLab CI** (<https://docs.gitlab.com/ee/ci/>), o **Jenkins** é das ferramentas de integração contínua cuja informação é mais fácil de encontrar uma vez que era a ferramenta com que a empresa já detinha alguma experiência em termos de utilização. O **GitLab** foi utilizado como forma de armazenamento na *cloud* do **VCS**.

2.3.3 *Version Control System*

O sistema de controlo de versões é um sistema que permite armazenar todas as alterações realizadas a um determinado conjunto de ficheiros, e armazenar todas essas alterações num histórico, o que permite voltar atrás caso seja necessário. Esta é uma ferramenta fundamental para os *developers*, pois permite armazenar todas as alterações ao código realizadas não só por um programador, mas por vários. E como o que é armazenado pelo **VCS** é a diferença entre os ficheiros iniciais e final (delta)

– ou seja, as linhas adicionadas e as linhas removidas – torna o processo de união dos ficheiros de múltiplos programadores muito mais eficiente e, em muitos casos, quase automático.

O **GitLab** (<https://gitlab.com/>) é uma ferramenta de gestão de repositórios de software com suporte a *Wikis*, a gestão de tarefas e a integração e entrega contínua. A empresa utiliza esta ferramenta de gestão de repositórios de software para o armazenamento de praticamente todos os projetos que desenvolve. Isto quer dizer que todo o código que foi desenvolvido durante o período de estágio foi publicado e está armazenado em repositórios **GitLab**. O **GitHub** (<https://github.com/>) é outro bom exemplo de uma ferramenta de gestão de repositórios de software com suporte a *Wikis* e gestão de tarefas, focado na partilha de código e na interação social entre os *developers*. Existem imensos serviços de armazenamento de código na *cloud*, entre os quais se podem destacar o **Bitbucket** (<https://bitbucket.org/>), o **Subversion** (<https://subversion.apache.org/>), o **Gogs** (<https://gogs.io/>) entre outros, dos mais respeitados e conhecidos do mercado. Para este caso, tendo em este contexto empresarial da atual metodologia de desenvolvimento de código, o **GitLab** foi a ferramenta é utilizada no que diz respeito à componente de armazenamento de código na *cloud* do **VCS**.

Podemos comunicar com os servidores de formas distintas. Através da utilização de (<https://>), ou através da utilização de um túnel, o *secure shell* (SSH). Neste caso, para publicarmos o código no repositório, vai ser utilizado o protocolo **SSH**.

O protocolo **SSH** encripta as ligações entre um cliente e um servidor. Encripta todas as mensagens, desde instruções até às credenciais de autenticação do utilizador. Tudo aquilo que é comunicado entre os dois *peers* é encriptado por um par de chaves. Uma chave é pública – fica sempre do lado do servidor – e outra chave, que dá acesso ao servidor a quem tiver uma cópia, é privada. Posteriormente, a chave privada será utilizada para que o orquestrador de processos possa puxar o código que publicamos no repositório.

2.3.4 Análise estática

A análise estática, é uma análise léxica (estrutural) e sintática (declarativa) para identificar más práticas no código desenvolvido ou declarações consideradas perigosas. Quando integrada numa *pipeline* de integração e entrega contínua, a análise ao código é feita de forma 100% automática garantindo a verificação das duas primeiras camadas da **stack** enquanto que as verificações semânticas do contexto da funcionalidade são feitas pelos testes unitários. A gravidade dos elementos possivelmente identificados pela análise estática, depende essencialmente de políticas existentes numa determinada organização. Isto quer dizer que duas empresas podem ter interpretações diferentes do mesmo relatório de análise estática.

O **SonarQube** (<https://www.sonarqube.org/>) é uma plataforma de gestão dedicada à análise contínua de qualidade do código. Já contém milhares de regras de análise estática de código e é utilizada para analisar várias linguagens. À semelhança desta plataforma, também o **Codacy** (<https://www.codacy.com/>) pode ser utilizado para automatizar estes parâmetros das *code reviews* e monitorizar a qualidade do código elaborando relatórios do impacto de cada *commit* ou *pull request*. Apesar de existirem outras ferramentas de análise estática no mercado com as mesmas funcionalidades, é o **SonarQube** que vai ser utilizado para revisão da qualidade do código uma vez que é facilmente integrável na *pipeline* e pode ser virtualizado em *containers*.

2.3.5 Repositório de artefactos

O repositório de artefactos é um órgão vital para o funcionamento da *pipeline* de integração e entrega contínua. É lá que são publicados os artefactos que produzimos através da execução do código desenvolvido. Estes componentes, modulares e simples de manter, podem desempenhar uma função específica dentro de **Web API**, de uma **Web Application** ou de uma **console application** que se pretenda desenvolver. O repositório utilizado para armazenar artefactos irá ser, mais tarde, reintegrado no ambiente integrado de desenvolvimento (IDE) e é a partir de lá que

podem ser descarregados e utilizados os *NuGet packages*.

A **Sonatype** disponibiliza uma ferramenta grátis para armazenamento de artefactos. O **Nexus Repository Manager OSS** (<https://www.sonatype.com/nexus-repository-oss>) é um repositório de artefactos compatível com os formatos de artefactos mais conhecidos. Também foram avaliadas outras hipóteses, como por exemplo o **JFrog artifactory** (<https://jfrog.com/artifactory/>) – um repositório de artefactos com características semelhantes ao supra-mencionado **Nexus** – o **Whitesource** (<https://www.whitesourcesoftware.com/>), o **ProGet** (<https://inedo.com/proget>) e o **MyGet** (<https://www.myget.org/>), que têm o mesmo tipo de metodologia de armazenamento de artefactos com estratégias de caching.

Durante as fases de desenvolvimento do protótipo da *pipeline* será utilizado o **Nexus Repository Manager OSS**. Mais tarde, será utilizado o **MyGet** para injeção de dependências num contexto ligeiramente diferente.

2.3.6 Orquestração de *containers*

A orquestração de *containers* é a capacidade de provisionar automaticamente a infraestrutura necessária para atender às necessidades de um projeto, sem a necessidade de utilização de máquinas físicas. Conceptualmente semelhante a máquinas virtuais, mas com o foco em serviço, cada *container* é um serviço em si e apenas um só. Esta abordagem promove a independência dos serviços, aumenta a resiliência do sistema e, principalmente, facilita o desenvolvimento e *deployment* dos serviços em produção.

O **Docker** é – à semelhança do **rkt** (<https://coreos.com/rkt/>) – uma tecnologia de virtualização em *containers*. Segundo o **Docker (2019e)**, os *containers* são métodos de *packaging* de aplicações que são executadas, juntamente com as suas dependências, isoladas de outros processos. O **Docker** funciona através da criação de imagens (**Dockerfiles**) que posteriormente são executadas em *runtime* dentro de um *container*. Comparativamente com outras ferramentas, como o **Podman** (<https://podman.io/>) e o **Buildah** (<https://buildah.io/>), que são utilizados para

orquestrar *containers* e gerar imagens respetivamente, o **Docker** pode ser utilizado para estas duas tarefas. Vai ser tirado partido desta vantagem uma vez que, neste contexto, é preferível utilizar a mesma tecnologia tanto na construção de imagens, como na orquestração de *containers*.

Inicialmente, o **Docker** era uma ferramenta desconhecida cujo conceito foi explorado no decorrer do estágio. À medida que o protótipo foi desenvolvido, a utilidade do **Docker** na orquestração dos serviços foi ganhando importância. Verificou-se que, no repositório público (<https://hub.docker.com/>), existiam imagens para serviços de **VCS**, serviços de bases de dados, de análise estática, de armazenamento de artefactos, armazenamento de imagens e por aí em diante. O que é curioso, é que a grande maioria dos serviços que as organizações utilizam no seu dia-a-dia disponibilizam imagens de **Docker** para uso da comunidade. Neste repositório conseguimos encontrar imagens de serviços como o **MS SQL Server**, **SonarQube**, **Sonatype Nexus**, **Docker Registry 2.0** entre outros.

O **Docker Swarm** (<https://docs.docker.com/engine/swarm/>) é uma ferramenta nativa do **Docker** que permite colocar um *cluster* de *containers* em ambiente de teste ou produção e controlar esse mesmo conjunto de *hosts*. O **Kubernetes** (<https://kubernetes.io/>) é outro orquestrador de *containers Docker* que permite gerir *workloads* de forma a garantir que o estado dos *clusters* vai de encontro às necessidades dos utilizadores. Através da utilização de dois conceitos “*labels*” e “*pods*”, os *containers* são agrupados em *logical units* para facilitar a sua gestão. Outras tecnologias, como o **Openshift** (<https://www.openshift.com/>), podem ser utilizadas para *deployments* em conjunto com o **Kubernetes** em sistemas operativos como o **RHEL** ou o **CentOS**. Estes três mecanismos de orquestração de *containers* podem e devem ser comparados mais tarde, noutra estudo, em termos de custo de execução e de operação aquando dos *deployments* da plataforma para produção.

2.3.7 Workflow da pipeline de CI/CD

Concluído o levantamento do estado da arte, tudo indica que a *pipeline* CI/CD será composta por quatro fases: *development*, *staging*, *pre-live* e *production*.

Development Environment

Na fase de *development* da *pipeline* (2.1), um *job* faz *fetch* ao código presente no **VCS** e, através de um conjunto de instruções, faz *build* à solução, executa análise estática e todos os testes unitários. Durante este processo são interpretados e apresentados os seus resultados.

Pela sequência lógica dos acontecimentos, a fase de *development* deve terminar com um *deployment* para *staging*. Como é possível constatar nos próximos capítulos, podem existir vários cenários de *deployment*. Neste caso serão considerados dois cenários diferentes. Um primeiro cenário onde o objetivo é validar um conjunto de funcionalidades através da execução dos testes unitários em conjunção com a análise estática e um outro cenário onde se pode adicionar *packaging* e *push* desse conjunto de funcionalidades ao *flow*. Apesar de existirem outras possibilidades para o *deploy*, serão trabalhados maioritariamente os dois cenários acima discutidos.

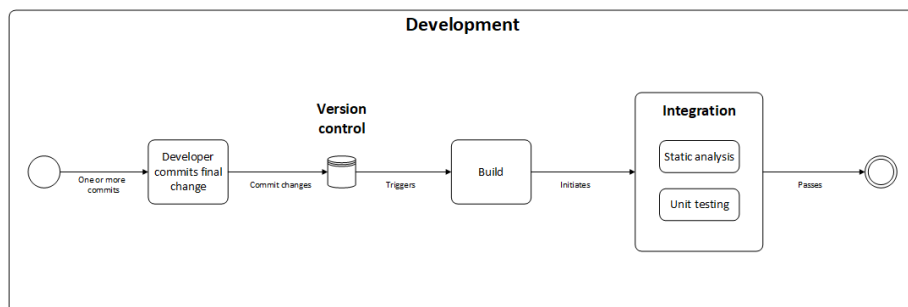


Figura 2.1 – Workflow de um job da fase de *development*

Staging Environment

Na fase de *staging* (2.2) é produzida uma imagem da aplicação que depois será colocada no repositório privado de imagens. Depois de lançada e validada a aplicação, são feitos testes de integração e performance, a aplicação é validada pelo *product owner* e – opcionalmente – pode ser incluída nesta fase uma análise à qualidade de compressão da imagem através da utilização do **Dive** (Goodman, 2018).

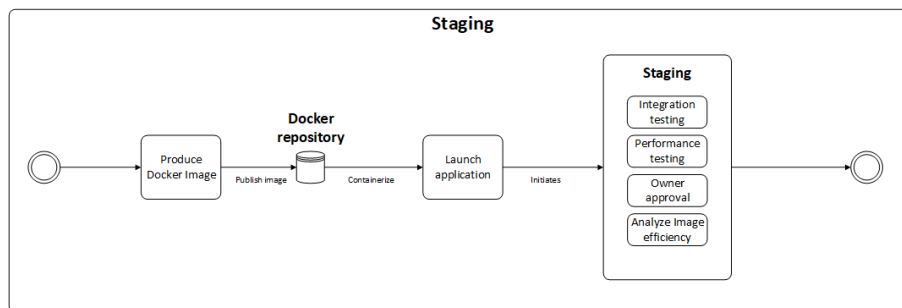


Figura 2.2 – Workflow de um job da fase de *staging*

Faz sentido incluir análise estática nesta fase apesar de não ser obrigatório. A análise estática pode ser utilizada como uma fase extra de verificação ao código da **Web API** desenvolvida em **C#**, uma vez que o código presente em **VCS** terá de ser validado e aprovado para que o processo de criação da imagem seja automatizado.

Pre-live Environment

A fase de *pre-live* assemelha-se com um ambiente de produção. O objetivo é precisamente configurar um *job* com um conjunto de processos que simulem a publicação de uma aplicação para produção, juntamente com todos os serviços dos quais a aplicação depende, executar um conjunto de verificações – nomeadamente validar que o comportamento da aplicação vai de encontro ao esperado – e submeter a aplicação a uma fase final de aprovação pela mão do *product owner*.

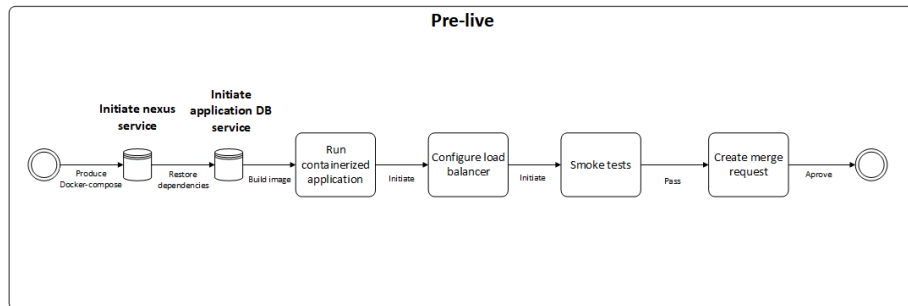


Figura 2.3 – Workflow de um *job* da fase de *pre-live*

Como se pode ver pela figura 2.3, a configuração do ambiente começa, à semelhança de uma fase de produção, com a configuração de um documento de composição de todos os serviços (`docker-compose.yml`) da aplicação. O documento terá de incluir a configuração sequencial dos serviços por ordem cronológica de acontecimentos, depois são feitos testes à aplicação e opcionalmente pode ser criado um *merge request*.

Production Environment

A única diferença entre este *job* e o anterior é a remoção dos *tests*, do *merge request* e das validações do *product owner* uma vez que teriam sido feitos anteriormente. Para esta fase pode ser considerada a adição de serviços de monitorização (2.4) como é o exemplo do **ELK Stack** (<https://www.elastic.co/pt/elk-stack>) e do **Nagios** (<https://www.nagios.org/>).

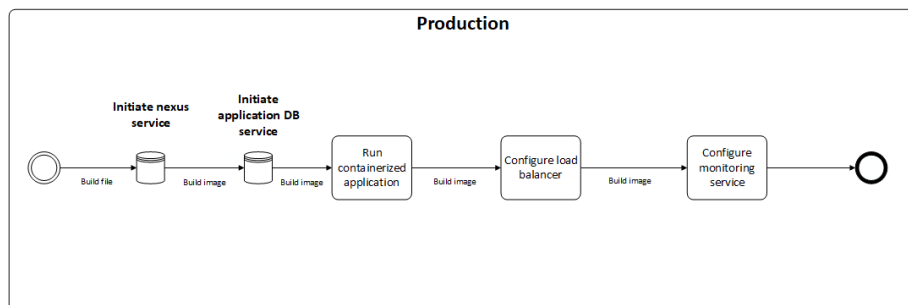


Figura 2.4 – Workflow de um *job* da fase de *production*

Caso estajamos a considerar a adição destes ou de novos serviços de monitorização na fase de produção é importante ter em mente que todos os serviços devem ser testados e validados previamente durante a fase de *pre-live*.

É também importante mencionar que estes *jobs*, desenvolvidos para cumprir um conjunto de objetivos pré-estabelecidos num contexto específico, podem não ter o mesmo tipo de resultado quando aplicados a outros contextos de negócio. Ao longo do estágio, durante a construção do protótipo da pipeline, as considerações e as decisões tomadas estiveram de acordo com aquilo que são as necessidades da empresa.



Construção da *pipeline* de integração contínua

3.1 Estudo de viabilidade das *frameworks* de teste

Uma das vantagens da implementação de uma *pipeline* de testes automatizada é a redução do tempo de validação e entrega de código funcional. Para que o código esteja funcional é necessário que, em primeiro lugar, corresponda aos requisitos para o qual está a ser desenvolvido. Uma vez verificada a funcionalidade – e tendo sido verificado que os seus requisitos estão de acordo com as expectativas iniciais – podemos então pensar no segundo ponto que consiste na otimização da *pipeline*, isto é, na redução da duração do tempo de compilação dos projetos, dos testes e de outros processos. Sabendo que existem várias *frameworks* à disposição, dependendo da linguagem de programação utilizada para desenvolvimento, foram escolhidas aquelas que se melhor se ajustam ao contexto da execução dos testes.

No desenvolvimento orientado a testes há uma metodologia de organização e estruturação do código bastante interessante, que é dividida em três fases: *Arrange*, *Act* e *Assert*. Esta abordagem aos testes é conhecida no mundo da programação como AAA ou *triple A*.

3.1.1 *Arrange, Act & Assert*

Na primeira fase, **Arrange**, apenas temos o código necessário para o *setup* daquele *test case*. É aqui que são criados os objetos ou *mocks* (caso existam) e também é nesta fase que é definido o resultado esperado. Na segunda fase, **Act**, é onde normalmente são invocados os métodos que vão ser testados. Na última fase, **Assert**, vai ser verificado se o resultado obtido na execução das duas fases anteriores vai de encontro ao resultado esperado. Na figura 3.1 encontra-se um exemplo de uma classe de teste desenvolvida em **MSTest**. Esta classe de teste tem três *test cases* diferentes e todos verificam que o resultado obtido é igual ao resultado esperado.

```
...[TestClass]
...public class UnitTestingInMSTest
...{
...    ...[TestMethod]
...    ...[DataRow(3, 2, 5)]
...    ...[DataRow(1, 1, 2)]
...    ...[DataRow(2, 0, 2)]
...    ...//[DataRow(int.MaxValue, 0, 2)]
...    ...public void shouldReturnASum(int n1, int n2, int sum)
...    ...{
...    ...    var sut = new Calculator(); //Arrange
...    ...    var result = sut.Add(n1, n2); //Act
...    ...    Assert.AreEqual(sum, result); //Assert
...    ...}
```

Figura 3.1 – Exemplo de uma classe de teste

De seguida, para dar início a esta *guideline*, irá ser delineada a estratégia para a execução dos testes num contexto próximo da realidade. Para tal, foi verificada, através do desenvolvimento de testes unitários, a funcionalidade da soma de dois números inteiros e foi verificado também que caso exista um número *null* a soma não será executada. Este procedimento foi executado em cada *framework* de testes e terá vários cenários de teste, dentro do mesmo contexto, com objetivos semelhantes. Tendo em conta que o desenvolvimento será orientado aos testes, será dado início através da criação de um projeto de teste. O tipo de *framework* selecionada para o primeiro desenvolvimento é indiferente pelo que fica à escolha do leitor. No entanto, para estudar a viabilidade de cada uma das *frameworks* de teste, foi utilizado:

- C#, mais concretamente, **dotnet core** v2.2;
- Test Frameworks: **xUnit (2.4.0)**, **NUnit (3.11.0)** e **MSTest (1.4.0)**;

A classe que se pretende implementar apenas possui um método de adição que aceita dois inteiros caso sejam *non-nullable* e retorna o valor da soma de ambos. É um método simples, cujo objetivo é somente a validação da funcionalidade de teste para permitir posterior utilização numa aplicação mais complexa.

A solução contém o projeto *ModelClasses* que mais tarde será o *package* – com a classe *Calculator* – juntamente com os três projetos de teste das três diferentes *frameworks* (3.2).

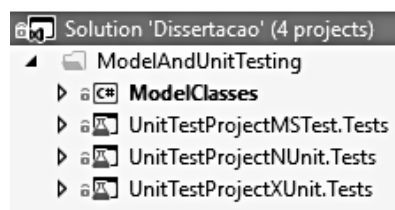


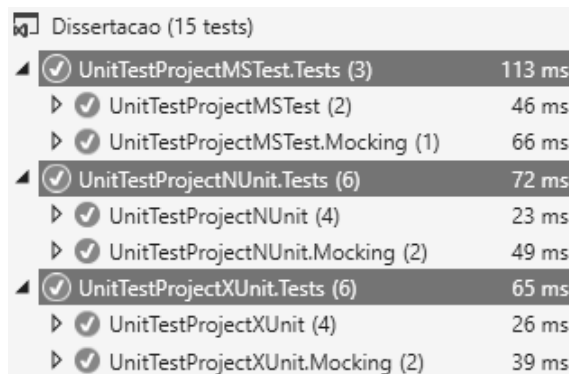
Figura 3.2 – Estrutura da solução

3.1.2 Tempo de execução dos testes

O tempo de execução das *frameworks* de teste (t_n) será medido através de 9 tentativas. Este intervalo de tempo será estimado em milissegundos e será feita a média de tempo de execução para cada uma das *frameworks*.

<i>Framework</i>	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	Média (ms)
MSTest	87	79	90	86	90	88	86	100	116	91,33
NUnit	76	59	58	60	55	55	64	60	56	60,33
XUnit	62	50	42	54	44	43	53	46	48	49,11

A *framework* de teste com melhor performance foi o **XUnit** com um tempo médio de execução de testes de 49,11 (ms), seguido pelo **NUnit** com um tempo médio de execução de 60,33 (ms) tendo ficado o **MSTest** em último lugar, com um tempo médio de execução de 91,33 (ms). O tempo de testes foi retirado do tempo calculado pelo **IDE**, neste caso o *Visual Studio 2017*, como pode ser visto na figura 3.3.



Dissertacao (15 tests)	
▶ ✓	UnitTestProjectMSTest.Tests (3) 113 ms
▶ ✓	UnitTestProjectMSTest (2) 46 ms
▶ ✓	UnitTestProjectMSTest.Mocking (1) 66 ms
▶ ✓	UnitTestProjectNUnit.Tests (6) 72 ms
▶ ✓	UnitTestProjectNUnit (4) 23 ms
▶ ✓	UnitTestProjectNUnit.Mocking (2) 49 ms
▶ ✓	UnitTestProjectXUnit.Tests (6) 65 ms
▶ ✓	UnitTestProjectXUnit (4) 26 ms
▶ ✓	UnitTestProjectXUnit.Mocking (2) 39 ms

Figura 3.3 – Explorador de testes do *Visual Studio 2017*

3.1.3 Considerações

A empresa utiliza atualmente **MSTest** como *framework* para desenvolvimento de testes. No entanto, a utilização de **XUnit** representaria uma melhoria de aproximadamente 46% no tempo médio de execução de testes. Já a utilização de **NUnit** representaria uma melhoria de aproximadamente 34% no tempo médio de execução de testes. Tendo em conta que o **MSTest** foi a *framework* com pior performance e estando à disposição no mercado mais do que uma solução *open-source* com performance evidentemente superior, o tempo de execução dos testes unitários pode ser otimizado quer com a utilização de **XUnit**, quer com a utilização de **NUnit**.

Sendo o próximo objetivo a construção de uma *pipeline* de integração e entrega contínua, que implica o desenvolvimento de testes de integração e performance, foi necessário eleger uma *framework* para o seu desenvolvimento. Uma vez que se está ainda dentro da temática dos testes, ficou decidido – juntamente com a orientação – que seria utilizado **XUnit** daqui em diante, no desenvolvimento dos testes de integração, dada a sua rapidez de execução.

3.2 Construção da *pipeline* de integração contínua

Neste capítulo foi anteriormente desenvolvido um módulo de soma de dois números inteiros, através da utilização de **TDD**. Neste segmento, vai ser explicado

todo o processo de criação de testes unitários e implementação de funcionalidades através da utilização de **XUnit** como *framework* de desenvolvimento orientado a testes. De seguida, será delineado o processo de criação da *pipeline*, com a comunicação com o sistema de controlo de versões (**GitLab**) em primeiro lugar, seguida pelo desenvolvimento da funcionalidade e da comunicação com **Jenkins** via **SSH**. Por último serão iniciadas duas instâncias, uma do **Nexus Repository OSS** e outra do **SonarQube** que serão o repositório de artefactos e ferramenta de análise estática respetivamente.

3.2.1 Configuração do sistema de controlo de versões

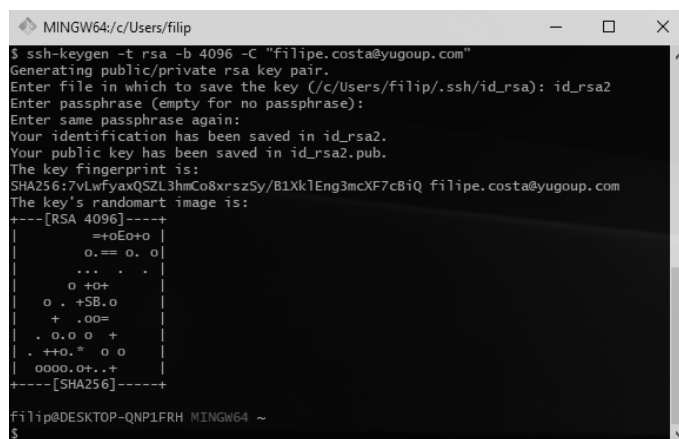
A configuração dos repositórios varia consoante o tipo de projeto. Para este caso, em particular, é pretendido implementar-se uma *pipeline* de testes automatizada com execução de testes unitários, análise estática, *packaging* e *push* da funcionalidade para um repositório remoto. Como vai ser necessário um repositório **.git**, onde será publicado o código da funcionalidade desenvolvida em formato não executável, faz sentido dar-se início pela comunicação entre o ambiente integrado de desenvolvimento – onde é desenvolvida a funcionalidade – e o repositório online que vai armazenar o código.

Criação de *SSH keys*

Como foi explicado anteriormente, a comunicação via **SSH** exige que seja criado um *key pair*. Para se criar o par de chaves, depois de estar instalado o **.git**, abre-se a linha de comandos, o **Git Bash**, e executa-se a instrução de criação de chaves `$ ssh-keygen -t rsa -b 4096 -C e-mail@exemplo.com` que vai criar uma chave nova, utilizando o e-mail providenciado como um parâmetro de configuração que é único para cada utilizador. De seguida somos solicitados a dar um nome ao ficheiro onde se pretende guardar a chave recentemente criada e, caso se pressione *Enter*, será utilizada a localização por defeito para guardar ficheiros. Com o destino seja validado, somos novamente solicitados, desta vez para criar uma password e reinserir

a password. Quando o processo de criação do par de chaves é bem sucedido, é gerada uma *fingerprint*, com uma imagem *randomart* como mostrado na figura 3.4.

Deste processo resultam dois ficheiros, um ficheiro contém a chave privada e outro contém a chave pública. A chave pública irá ser utilizada pelo **GitLab** para comunicar com a máquina onde está instalada a instância do **Jenkins**. A chave privada deve ser armazenada de forma segura (Atlassian, 2019).



```
MINGW64: c:/Users/filip
$ ssh-keygen -t rsa -b 4096 -C "filipe.costa@yugoup.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/filip/.ssh/id_rsa): id_rsa2
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa2.
Your public key has been saved in id_rsa2.pub.
The key fingerprint is:
SHA256:7vLwFyaxQ5ZL3hmCo8xrszSy/B1Xk1Eng3mcXF7cBiQ filipe.costa@yugoup.com
The key's randomart image is:
+---[RSA 4096]-----+
|
|  oEo+o
| o,== o. o|
| ... ..|
| o +o+
| o . +SB,o
| + .oo=
| . o.o o +
| . ++o,* o o
| oooo.o+..+
+---[SHA256]-----+
filip@DESKTOP-QNP1FRH MINGW64 ~
$
```

Figura 3.4 – Consola de **Git Bash**

Comunicação via *SSH*

Para ser adicionada a chave pública no **GitLab**, após serem acedidas as definições do utilizador, clica-se na opção *SSH Keys* (3.5) e insere-se a chave pública – o ficheiro com a terminação **.pub** – que é um dos ficheiros gerados anteriormente. Pode optar-se também pela atribuição de um nome à chave e adição dessa chave ao conjunto de chaves. Depois de criado um par de chaves, é criado um novo repositório na interface web do **GitLab** que será clonado para a máquina.

Para clonar o repositório para a máquina é aberta a linha de comandos e executada a instrução `$ git clone git@gitlab.com:nome_utilizador/repositorio.git`. De seguida seremos solicitados para inserir a password com a qual foi anteriormente criada a chave, ficando assim criada a pasta do projeto com ligação ao sistema de repositórios. A partir deste instante, todo o desenvolvimento da funcionalidade irá

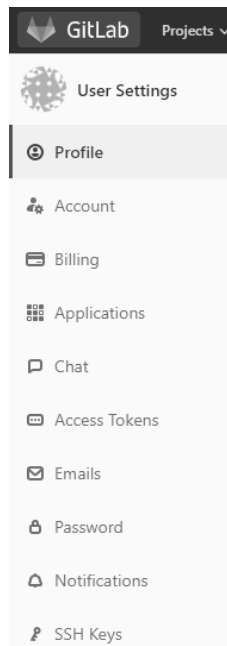


Figura 3.5 – Menu de definições do utilizador do **GitLab**

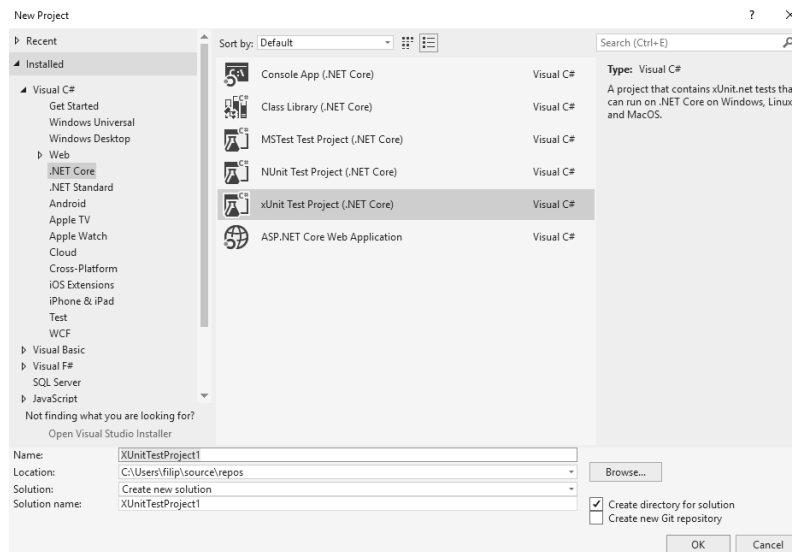
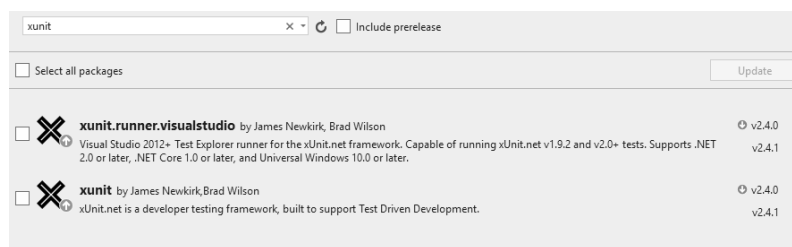
estar sincronizado entre o **GitLab** e o **Visual Studio** através do *Team Explorer* que é uma ferramenta de integração de código desenvolvida e mantida pela equipa da **Microsoft** (Git, 2019).

3.2.2 Desenvolvimento de testes unitários

Se o objetivo for desenvolver um *package* seguindo a metodologia **TDD** o ideal será começar pela criação do projeto de teste. Como pode ser visto na figura 3.6, terá de se alterar o nome do projeto para o nome pretendido.

Após a criação do projeto de teste, são adicionados ao projeto os dois packages que são necessários para o desenvolvimento dos testes. São eles o **xUnit** e o **xUnit.runners.visualstudio** (3.7).

No desenvolvimento dos testes serão criados dois métodos. Um será utilizado para verificar se o resultado da soma é igual ao resultado esperado, outro para verificar se existe algum parâmetro com valor **null**.

Figura 3.6 – Menu de criação do projeto no *Visual Studio 2017*Figura 3.7 – Biblioteca de *packages* do *Visual Studio 2017*

Criação da classe e estruturação dos métodos

Depois dos *packages* estarem concluídos e – antes de ser definida a classe como pública – para começar o desenvolvimento da *Theory*, é muito útil o apoio da documentação oficial que a **Microsoft** e a comunidade disponibilizam para se ter uma ideia mais consolidada sobre os métodos de desenvolvimento (**AAA**) de código para testes (Microsoft, 2019d).

Theory

No contexto do desenvolvimento orientado a testes, a *Theory* é, à semelhança dos atributos que vemos nos métodos de uma classe **controller**, um atributo que

vai definir que tipo de testes serão desenvolvidos. Uma *Theory* pode ser aplicada quando são desenvolvidos múltiplos casos de teste para o mesmo método. Se o objetivo for desenvolver apenas um caso de teste para um determinado método, deve ser utilizado o atributo *Fact* (Microsoft, 2019e).

Adiciona-se uma classe para o desenvolvimento dos testes, *UnitTestingInXUnit*, muda-se o nível de acesso para **public** e é iniciado o desenvolvimento da *Theory* (3.8).

```
using System;
using Xunit;

public class UnitTestingInXUnit
{
    [Theory]
```

Figura 3.8 – Classe *UnitTestingInXUnit*

De seguida dá-se um nome, por exemplo *shouldReturnASum*, ao método que é criado dentro da *Theory* e passam-se três argumentos inteiros, **n1**, **n2** e **sum** (3.9).

```
public void shouldReturnASum(int n1, int n2, int sum)
```

Figura 3.9 – Método *shouldReturnASum*

Dentro do método, é definida uma variável – **sut** (*system under test*) – que é onde o sistema a ser testado estará contido. Uma vez que o objetivo é criar um método para calcular a soma de dois números, o sistema a ser testado – **sut** – será o objeto *Calculator*.

Esta será a fase de *Arrange*. O **Visual Studio**, com a ajuda do *intellisense*, permite-nos criar uma classe, num ficheiro novo, com o nome do objeto, a classe *Calculator* e o método *Add* (3.10).

```
public int Add(int? n1, int? n2)
{
    return n1.Value + n2.Value;
}
```

Figura 3.10 – Método *Add*

Após a criação da classe num ficheiro à parte, de volta ao projeto de teste, é iniciada a implementação da fase de *Act*. Para esta fase é necessário definir outra

variável chamada *resultado* que vai receber o resultado da soma dos dois números adicionados no método de adição.

Na implementação do teste, é invocado o método *Add* na classe *Calculator* para, só depois, se prosseguir para a fase de *Assert* onde é verificado que o resultado obtido é igual ao esperado. A classe de teste deve ter sempre uma estrutura simples e organizada, com métodos à semelhança daquilo que pode ser visto na figura 3.11.

```
public void shouldReturnASum(int n1, int n2, int sum)
{
    -----
    var sut = new Calculator(); //Arrange
    var result = sut.Add(n1, n2); //Act
    Assert.Equal(sum, result); //Assert
}
```

Figura 3.11 – Estrutura exemplo de um método

InlineData

Para que a *Theory* possa funcionar de forma correta, adiciona-se *InlineData*. Estes atributos são colocados logo a seguir ao atributo *Theory*. Começa-se pela adição de um caso de teste cujo resultado seja falso (e.g. $1 + 1 = 1$) e outro caso de teste verdadeiro (e.g. $1 + 1 = 2$) para ser feito um despiste e assim verificar-se que o método está a funcionar de acordo com o esperado (3.12).

```
public class UnitTestingInXUnit
{
    [Theory]
    [InlineData(1, 1, 1)]
    [InlineData(1, 1, 2)]
}
```

Figura 3.12 – Atributos [*InlineData()*]

Como pode ser visto dentro do *test explorer* (3.13) do ambiente integrado de desenvolvimento, um dos testes do método *shouldReturnASum* passou e o outro teste falhou conforme esperado.

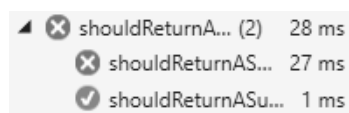


Figura 3.13 – Resultado da execução dos testes

Independentemente do número (*n*) de testes que são executados, caso pelo menos um dos testes tenha resultado numa falha – isto é caso a *assertion* não se verifique – a execução daquele método de teste (neste caso da *Theory*) vai falhar completamente.

A estrutura final da *Theory*, para o caso de teste onde são adicionados dois números é relativamente simples. No entanto à medida que, para o mesmo caso, se aumenta o número cenários de teste em termos de *InlineData*, aumenta também o nível de complexidade da *Theory*. A *Theory*, para todo o caso, ficará estruturada à semelhança daquilo que se pode ver na figura 3.14.

```
using ModelClasses;
using System;
using Xunit;

public class UnitTestingInXUnit
{
    [Theory]
    [InlineData(3, 1, 4)]
    [InlineData(3, 0, 3)]
    // [InlineData(int.MaxValue, 0, 2)]
    public void shouldReturnASum(int n1, int n2, int sum)
    {
        var sut = new Calculator(); // Arrange
        var result = sut.Add(n1, n2); // Act
        Assert.Equal(sum, result); // Assert
    }
}
```

Figura 3.14 – Estrutura completa de uma *Theory*

Passagem de argumentos nulos

Pensemos no caso em que o utilizador passe um valor *null* e espere que o método de adição faça *output* a um resultado válido. Esta é uma validação que deve ser feita para dar seguimento à construção dos testes. Através da criação de um método de teste – *ShouldNotAddNULL* – que simula este cenário, terá de se verificar que, caso exista um argumento *null*, esse mesmo argumento será apanhado por uma exceção do tipo *ArgumentNullException*.

Para se validar que a classe **Calculator** lança uma exceção do tipo *ArgumentNullException* quando algum dos dois parâmetros é nulo, é necessário em primeiro

lugar adaptar a *Theory* do método *ShouldNotAddNULL* para que aceite valores inteiros que possam ou não existir. Para tal serão passados dois parâmetros do tipo “*int?*”. A seguir é iniciada uma instância do objeto **Calculator** – que será o **sut** – na fase de *Arrange* para depois, nas fases de *Act* e *Assert* ser possível verificar que a exceção é lançada. Neste caso em particular, visível na figura 3.15, foram agrupadas as fases de *Act* e *Assert* numa função **lambda**.

```
public void ShouldNotAddNULL(int? n1, int? n2)
{
    //Arrange
    var sut = new Calculator();
    //Assert
    Assert.Throws<ArgumentNullException>(() => sut.Add(n1, n2));
}
```

Figura 3.15 – Estrutura da segunda *Theory*

Em segundo lugar, para se poderem passar números inteiros *nullable*, terá de ser feita uma pequena alteração à classe **Calculator**. Em termos de estrutura, ficaria implementada – dentro do método de adição – uma condição (*if*) que verificaria se os argumentos não são nulos. Caso exista algum argumento sem valor, é feito um *throw* à exceção do tipo *ArgumentNullException*. Caso contrário é retornado o resultado da soma dos dois valores (3.16).

```
using System;
using log4net;

public class Calculator
{
    public int? n1 { get; set; }
    public int? n2 { get; set; }
    public int Add(int? n1, int? n2)
    {
        if (!n1.HasValue || !n2.HasValue)
        {
            throw new ArgumentNullException();
        }
        return n1.Value + n2.Value;
    }
}
```

Figura 3.16 – Estrutura da classe **Calculator** após alterações

De seguida, para validar que a *Theory* executa os dois cenários de teste, são atribuídas duas linhas de *InlineData()*. Por outras palavras, são criados dois casos de teste. O primeiro, que vai passar um valor nulo no lugar do primeiro argumento

e o segundo que irá passar um valor nulo no lugar do segundo argumento. No final da adição destes casos de teste a *Theory* terá aspeto semelhante ao da figura 3.17.

```
[Theory]
[InlineData(null, 1)]
[InlineData(1, null)]
public void ShouldNotAddNULL(int? n1, int? n2)
{
    //Arrange
    var sut = new Calculator();
    //Assert
    Assert.Throws<ArgumentNullException>(() => sut.Add(n1, n2));
}
```

Figura 3.17 – Estrutura da *Theory ShouldNotAddNULL* com os dois casos de teste

Nesta fase ambas as *Theorys* estão devidamente desenvolvidas, com vários *test cases*, com verificação e validação de funcionalidade. Por último lugar, para fechar a questão do desenvolvimento dos testes, são executados todos os testes para se verificar que passam como esperado (3.18). O código está pronto a ser publicado no sistema de repositórios.

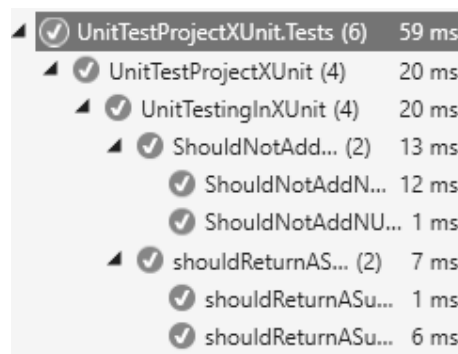


Figura 3.18 – Mostrador do *Test Explorer*

3.2.3 Publicação do código no sistema de controlo de versões

No *Team Explorer* podem ser publicadas as alterações feitas ao projeto, caso existam, clicando nas **Changes** (3.19). Depois terá de ser inserida uma mensagem que será publicada juntamente com o *commit*. Nesta mensagem, é política da empresa colocar um *briefing* dos ficheiros alterados, ou adicionados e, opcionalmente, pode ser especificada a funcionalidade adicionada.

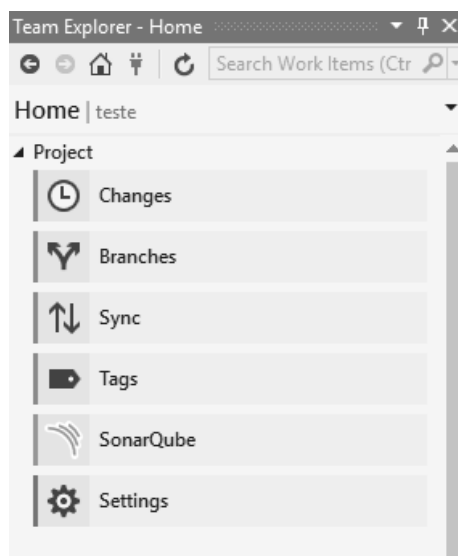


Figura 3.19 – Mostrador do *Team Explorer*

Clica-se em **Commit** e insere-se a password (3.20), gerada no momento da criação das chaves **SSH**, para que o código do projeto seja publicado no **VCS**.

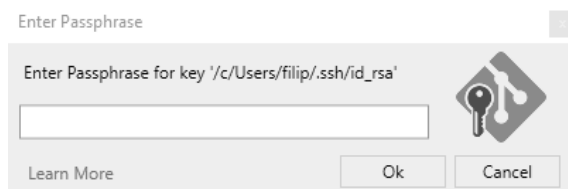


Figura 3.20 – Solicitador do **SSH**

Para a publicação do código deve ser utilizado um ficheiro que inclui ou exclui outro conjunto de ficheiros do projeto. Nem todos os ficheiros gerados são necessários para que o código possa ser executado. Alguns desses ficheiros são gerados automaticamente e não têm utilidade em termos de compilação. Portanto a inclusão ou exclusão deste conjunto de ficheiros – através da utilização do **.gitignore** – é indispensável para que numa fase mais avançada o código esteja compilável e seja reutilizável (<https://github.com/github/gitignore>).

3.2.4 Orquestração com *Jenkins*

A configuração do **Jenkins** será feita na máquina de *staging* disponibilizada pela empresa. A máquina está disposta em rede local. Isto torna o contacto com os dispositivos de visualização do estado das *builds* mais facilitado. Mais à frente este tema é devidamente abordado.

Instalação dos plugins

Para que possam ser executadas as *builds* da compilação de código do módulo de soma, juntamente com os respetivos testes unitários e a análise estática, é necessária a instalação de um conjunto de *Plugins* no **Jenkins**, nomeadamente:

- Credentials;
- MSBuild Plugin;
- Gitlab Plugin;
- Blue Ocean (opcional);
- Slack Notification (opcional);

Da lista acima é obrigatório o uso dos *plugins* **Credentials**, **MSBuild** e **Gitlab**, utilizados para compilação de código do projeto C#. O **Credentials** faz a gestão das credenciais de acesso ao repositório do sistema de controlo de versões sem que estas sejam expostas. O **MSBuild** é a plataforma utilizada para fazer *build* de projetos .NET. O **Gitlab** é utilizado para estabelecer uma ligação segura ao **VCS**. Os restantes *plugins* são opcionais dado que o **Blue Ocean** é uma interface de visualização do **Jenkins** que apenas apresenta uma interface gráfica mais apelativa para o utilizador. Já o **Slack Notification** é um componente adicional para o sistema de visualização, caso se pretenda que os *developers* recebam notificações no *Slack* – canal de comunicação utilizado pela empresa – sobre os diferentes acontecimentos durante as fases da *pipeline*.

Instalação dos módulos de desenvolvimento do C#

Será necessário descarregar o **.NET Core SDK**, que já inclui o **.NET Core Runtime**, tendo sempre em mente o sistema operativo que é utilizado. Neste caso é utilizado o **Windows 10**. Estes componentes podem ser descarregados a partir do site oficial da **Microsoft**.

Com o *Software Development Kit* (SDK) devidamente instalado, neste caso na máquina de *staging*, procede-se à configuração do **Jenkins**. Como dito anteriormente, esta ferramenta de integração contínua fará *fetch* ao código presente no sistema de controlo de versões via **SSH** para depois poderem ser feitas operações sobre o mesmo. Para que tal seja possível, providencia-se ao orquestrador de processos a chave pública para que possa comunicar com o **GitLab** e é indicado, nas opções de configuração do **Jenkins**, o caminho da localização do **MSBuild** instalado anteriormente na máquina.

Configuração das credenciais de acesso ao GitLab

Com o *plugin* de gestão de credenciais (**Credentials**) instalado, é adicionada uma nova credencial de acesso. Para tal, clica-se em cima do *plugin* **Credentials** e acede-se à opção que foi expandida, “System” (3.21).

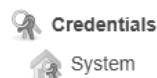


Figura 3.21 – Plugin Credentials

De seguida clica-se em *Global credentials* e adiciona-se uma nova credencial (*Add Credentials*). Depois, selecciona-se o tipo de credencial na *dropdownlist* como “SSH Username with private key” e insere-se diretamente a chave privada, seleccionando a opção “Enter directly”, no campo “Private Key”, como pode ser visto na figura 3.22.

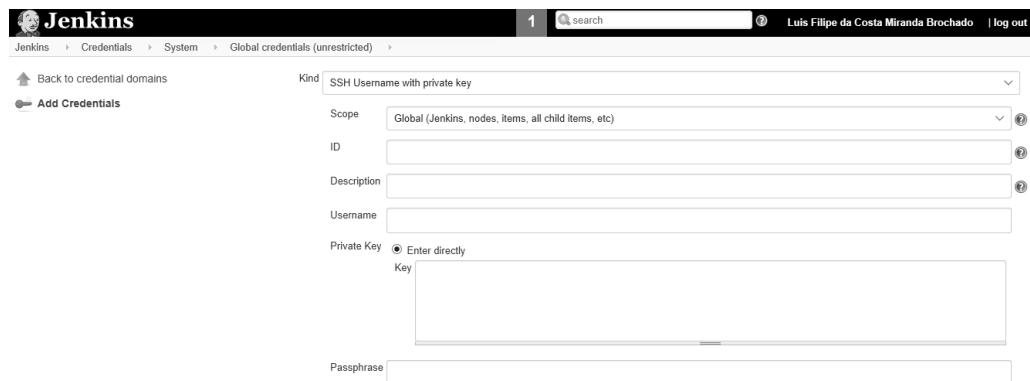
The screenshot shows the Jenkins web interface. The breadcrumb trail is 'Jenkins > Credentials > System > Global credentials (unrestricted)'. On the left, there are links for 'Back to credential domains' and 'Add Credentials'. The main form is titled 'SSH Username with private key'. It includes a 'Kind' dropdown set to 'SSH Username with private key', a 'Scope' dropdown set to 'Global (Jenkins, nodes, items, all child items, etc)', and input fields for 'ID', 'Description', 'Username', 'Private Key' (with a radio button for 'Enter directly'), and 'Passphrase'.

Figura 3.22 – Configuração das credenciais de acesso

Configuração do MSBuild

Nas opções de configuração do **Jenkins**, em **Manage Jenkins**, existe um conjunto de opções à disposição. Será selecionada a opção **Global Tool Configuration**. Se o *plugin* **MSBuild** tiver sido instalado, deverá aparecer nas opções de instalação uma imagem semelhante à da figura 3.23.

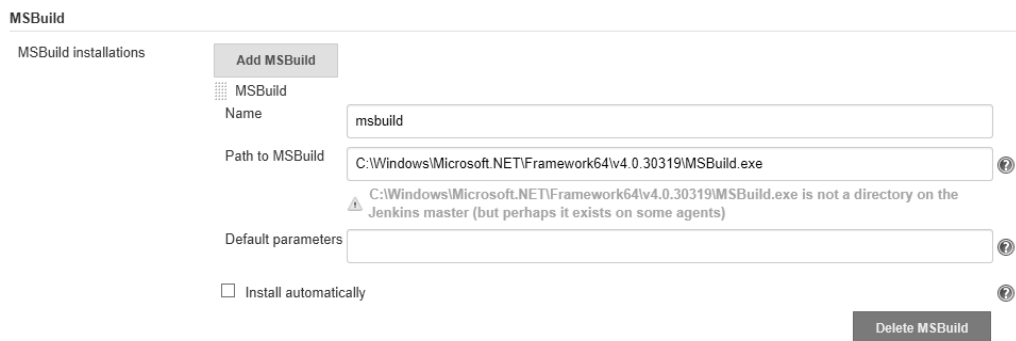
The screenshot shows the 'MSBuild' configuration page in Jenkins. It has a header 'MSBuild' and a sub-header 'MSBuild installations'. There is an 'Add MSBuild' button. Below, there is a table with one entry for 'MSBuild'. The table has columns for 'Name' (containing 'msbuild'), 'Path to MSBuild' (containing 'C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe'), and 'Default parameters' (empty). A warning icon is next to the path, with a message: 'C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe is not a directory on the Jenkins master (but perhaps it exists on some agents)'. There is an 'Install automatically' checkbox and a 'Delete MSBuild' button.

Figura 3.23 – Caminho de instalação do **MSBuild**

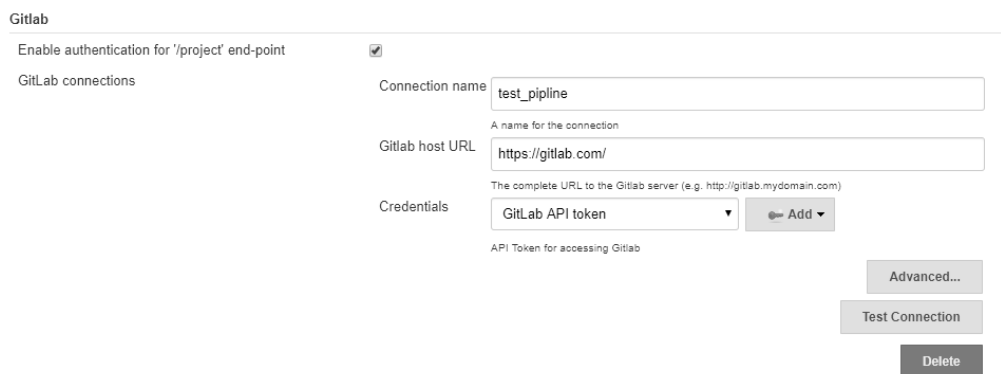
Com o caminho absoluto do ficheiro executável do **MSBuild** colocado na opção “Path to MSBuild”, o orquestrador de processos está pronto a compilar o código dos projetos **.NET**. Nesta fase a configuração do orquestrador está concluída. De seguida será abordada a configuração da *pipeline* testes automatizados.

3.2.5 Configuração da *pipeline*

Começando pela criação de um *New Item*, é atribuído um nome à escolha do utilizador seleccionando a opção *Freestyle project*. O primeiro *job* está criado e segue-se sua configuração. Este *job* será o primeiro componente da *pipeline*.

Estrutura dos *jobs*

Para a configuração começa-se por dar uma descrição ao *job* e por seleccionar a opção para descartar builds anteriores – “Discard old build” – para que sejam eliminados todos os ficheiros e artefactos resultantes de compilações de código anteriores. Depois adiciona-se o **GitLab Plugin** à lista de *plugins* para que seja possível configurar uma ligação ao **VCS** através da criação de um **API token**. Para configurar esta ligação é necessário aceder novamente às propriedades de configuração do sistema – **Configure System** – e navegar até aparecer a secção de configuração do **GitLab** (3.24).



The screenshot shows the 'GitLab' configuration section in a web interface. At the top, there is a checkbox labeled 'Enable authentication for "/p>

Figura 3.24 – Configuração da ligação ao **GitLab**

Antes de ser adicionado um *Access Token* ao orquestrador de processos é necessário requisitar, dentro do sistema de controlo de versões, um novo *Personal Access Token*. Para tal, acede-se à plataforma **GitLab**, navega-se até aos *user settings*, clica-se em *Access Tokens* e adiciona-se um nome, uma data de validade e os **scopes** do *token* como mostrado na figura 3.25.

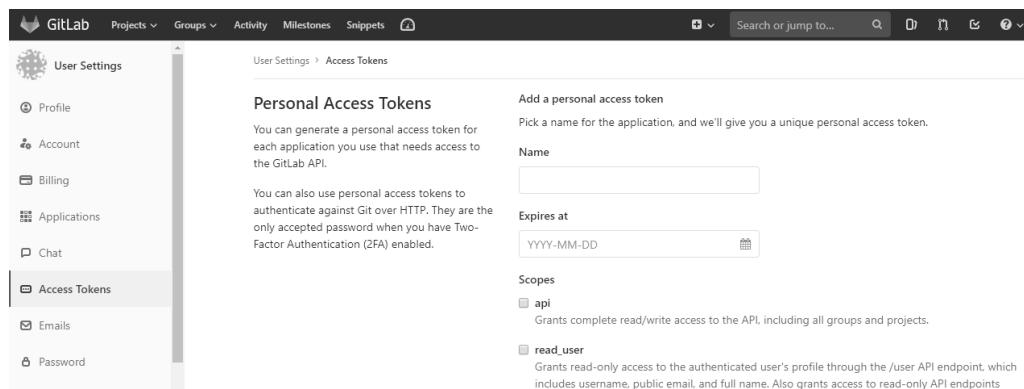


Figura 3.25 – Menu de criação do *Personal Access Token*

O *access token* criado será depois adicionado nas credenciais do **Jenkins**. Este *access token* só pode ser visualizado uma única vez por questões de segurança e, por boa prática, não deve ser utilizado em mais que um serviço.

Voltando à configuração do *job*, com a *GitLab Connection* configurada, seguem-se as configurações de **Source Code Management**. É aqui que serão configuradas as ligações via **SSH** aos respectivos repositórios criados que foram armazenados no **VCS**. Assim sendo seleciona-se a opção **Git** e preenche-se o **URL** do repositório com a hiperligação do protocolo **SSH**, com as respetivas credenciais de acesso, que é o meio de ligação pretendido para estabelecer a conexão. A *branch* que será utilizada para fazer *build* é a única existente no projeto (3.26), **master**.

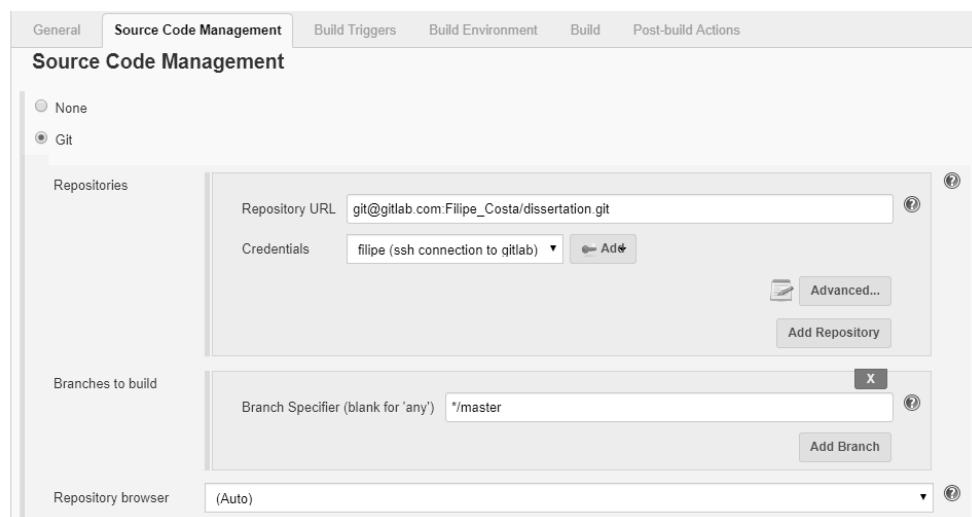


Figura 3.26 – *Source Code Management*

O próximo separador, **Build Triggers**, vai ser passado à frente uma vez que não é necessário qualquer tipo de configuração neste campo. Este segmento – **Build Triggers** – é onde são agendados *triggers* automáticos a novas *builds* mediante certos eventos. Será utilizado mais à frente, por exemplo, na configuração dos *jobs* que serão dependentes da execução deste primeiro *job*.

No próximo segmento, **Build Environment**, é selecionada a opção “Delete workspace before build starts”. Esta opção, segundo a tradução literal do Inglês, significa que é apagado o *workspace* gerado pelo **Jenkins** sempre que começar uma *build* (3.27).

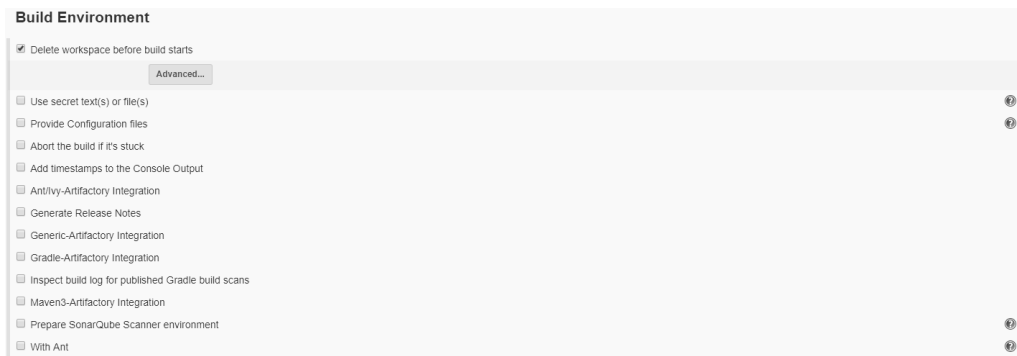


Figura 3.27 – *Build Environment*

Esta funcionalidade tem como objetivo evitar que execuções anteriores da *pipeline* provoquem erros devido a versões de serviços e/ou dependências antigas que tenham sido usados e não tenham sido atualizados corretamente para a nova execução.

3.2.6 Integração dos serviços

Para a integração de serviços externos na *pipeline* vão ser utilizados:

- Docker;
- SonarQube;
- Nexus Repository Manager OSS 2;

- **Docker Registry 2.0;**

Na fase de *Build*, vão ser inseridas as instruções que normalmente seriam introduzidas numa linha de comandos. Para que estejamos mais contextualizados com as instruções que são utilizadas aconselha-se a consulta da documentação oficial de apoio da **Microsoft** referente ao **.NET Core CLI** (Docs.microsoft.com, 2017). Toda a informação que é necessária está disponível nesta documentação.

Integração com SonarQube

Começemos por integrar a análise estática na *pipeline*. Para fazer a análise estática do código é utilizada uma *tool* de revisão de código chamada **SonarQube**. Esta ferramenta de análise estática já integra com **Docker** portanto vai ser executada uma instância de uma versão do **SonarQube** dentro de um *container* que será publicado na porta 9000. Todos os dados serão guardados dentro de um *container* do **Docker** que foi anteriormente criado através de uma técnica de persistência de dados em volume (Docker, 2019d). Desta forma, sempre que for necessário, os dados podem ser acedidos através do *attachment* de um *container* a este volume de dados conferindo assim a possibilidade e a garantia de se utilizar uma instância para aceder às informações guardados sobre os projetos.

A configuração de um projeto dentro da instância do **SonarQube** é relativamente simples. As credenciais de autenticação, por defeito, são `admin` para o **username** e `admin` para **password**. Todos os projetos são criados da mesma forma, independentemente da linguagem utilizada. Neste caso é seleccionada a opção “criar novo projeto” e, após a atribuição de uma chave e de um nome ao projeto, adiciona-se um *token* de identificação. De seguida é seleccionada a linguagem em que o projeto foi desenvolvido. É importante ter em consideração que tanto a atribuição do *token* como a seleção da linguagem em que o projeto foi desenvolvido são dois passos que também têm relevância para a integração do **SonarQube** com a *pipeline*. Mais tarde será necessário a chave que atribuída ao projeto nesta fase.

A instrução que se vai utilizar para dar início à análise estática é a seguinte:

`$ dotnet SonarScanner.MSBuild.dll begin /k:"project-key"` e caso se pretenda, por exemplo, executar a análise estática a partir de um *container* dentro de uma máquina virtual, podem ser adicionadas como opções `/d:sonar.host.url="hostname"` para indicar o *hostname* onde será feita análise e `/d:sonar.login="access_token"` para indicar o *access token* que permite aceder ao *container*. É necessário ter em conta que o **Jenkins** não reconhece nem o comando *dotnet*, nem a ".dll" do *SonarScanner*, portanto são necessários os *absolute paths* destes dois ficheiros dentro da máquina principal para que o orquestrador os consiga encontrar (3.28).

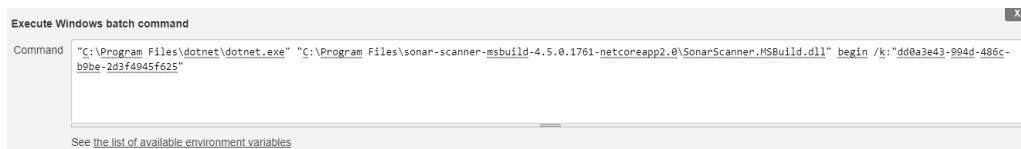


Figura 3.28 – Primeira instrução do **SonarQube**

Para concluir a análise estática é necessário utilizar como última instrução: `$ dotnet SonarScanner.MSBuild.dll end` (3.29). No caso em que a análise estática é iniciada com as opções extra, é necessário executar a instrução descrita neste parágrafo com a adição da opção `/d:sonar.login="access_token"` para encerrar corretamente a execução da análise estática.

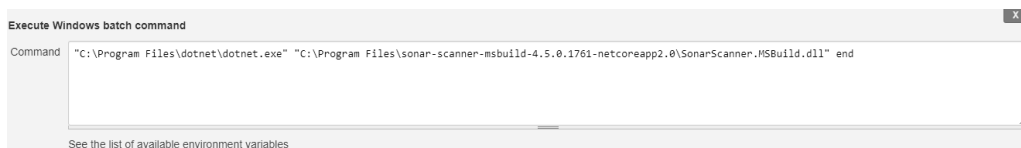


Figura 3.29 – Última instrução do **SonarQube**

Executa-se a instrução para iniciar a ferramenta de análise estática no início do *job* para que – durante as fases de *build* e *testing* – todo o código da aplicação seja revisto. É executada a instrução para terminar a ferramenta antes do final do *job* para que seja parada a execução desta *tool* (Loureiro, 2019). Os resultados da análise estática serão apresentados à semelhança daquilo que se pode ser visto na figura 3.30.

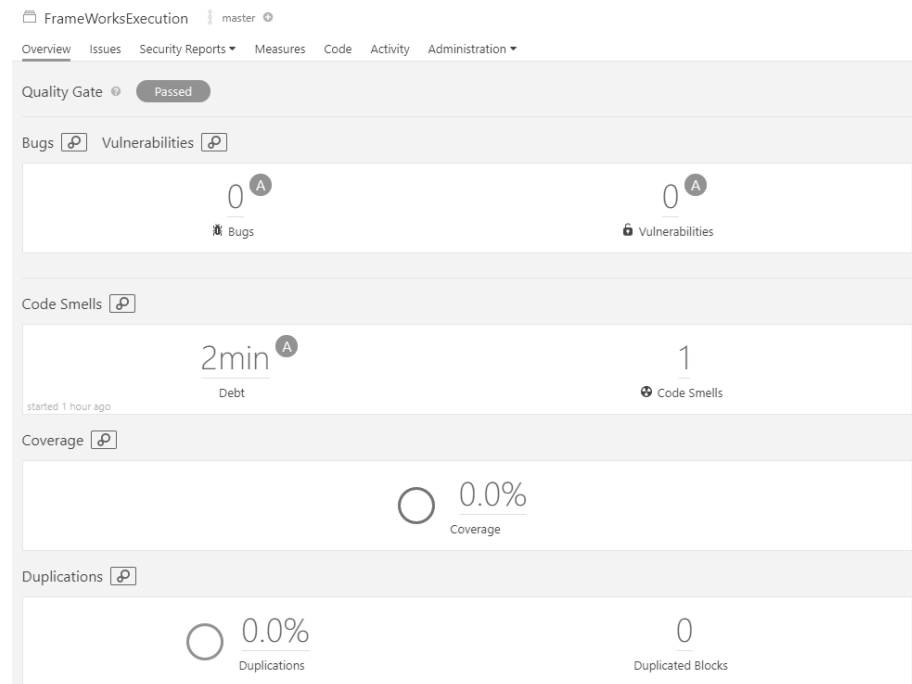


Figura 3.30 – SonarQube – Quality Gate

Build do projeto e dos testes

Para se fazer *build* a um projeto (.csproj) ou a uma solução (.sln) a instrução é: `$ dotnet build project-path` (Docs.microsoft.com, 2019a). Aqui entramos na fase de *build* (3.31) do projeto e dos testes unitários que foram criados para a aplicação calculadora.

Após a fase de *build*, são executados os testes para o **Jenkins** interpretar os seus resultados e produzir um gráfico com a estatística dos *test runs*. A instrução utilizada para produzir os resultados dos testes em formato interpretável (.trx) foi a seguinte: `$ dotnet test project-path --no-build --logger trx` (Docs.microsoft.com, 2019d).

Esta instrução produz um ficheiro (.trx) com o resultado dos testes que é interpretado pelo orquestrador e apresentado na GUI (3.32).



Figura 3.31 – Instruções de *build* dos projetos

Test Result

0 failures (±0)

19 tests (±0)

Took 0 ms.

[add description](#)

All Tests

Package	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
UnitTestProjectMSTest	10 ms	0	0	5	5
UnitTestProjectMSTest.Mocking	22 ms	0	0	2	2
UnitTestProjectNUnit	52 ms	0	0	4	4
UnitTestProjectNUnit.Mocking	63 ms	0	0	2	2
UnitTestProjectXUnit	22 ms	0	0	4	4
UnitTestProjectXUnit.Mocking	47 ms	0	0	2	2

Figura 3.32 – Resultado dos testes unitários

Integração com o repositório *Nexus Repository Manager OSS*

A ferramenta de armazenamento de artefactos utilizada é o **Nexus Repository Manager OSS**. Este sistema de gestão de artefactos tem integração com **Docker** – que foi o motivo com mais peso na sua seleção – e apresenta uma **GUI** de simples utilização. Para a *pipeline* foi escolhida uma versão mais antiga, a versão **2.14.12-02**, uma vez que durante a escrita do documento cumpria com os requisitos estabelecidos pela orientação. A versão **2.14.13-01** também pode ser utilizada uma vez que não existem diferenças tanto em termos de utilização como de configuração.

Para que seja possível publicar artefactos dentro de um *container* com uma instância do **Nexus** é necessário realizar um conjunto de operações. Por outras palavras, é preciso empacotar o módulo de soma, *ModelClasses*, para um formato específico e assim transformar o módulo de soma de dois números inteiros num **NuGet Package**. Esta operação pode ser feita através da utilização da instrução: `$ dotnet pack project-path` (Docs.microsoft.com, 2019c). Esta instrução gera um ficheiro do tipo “**.nupkg**” que é o módulo *plug-and-play* da funcionalidade desenvolvida.

O próximo passo é a configuração do repositório. Para configurar corretamente o repositório aconselha-se a leitura da documentação produzida pela **Sonatype** (Help.sonatype.com, 2019) e disponibilizada no repositório público de imagens do **Docker** (Sonatype, 2018). A configuração do repositório **NuGet** requer a criação de dois repositórios, um *proxy* e um *hosted*, que mais tarde vão ser colocados dentro de um grupo, *public*, resultando no total de três repositórios criados. As credenciais de autenticação por defeito são `admin` para o **username** e `admin123` para **password**. A criação dos repositórios do **Nexus** é ligeiramente mais complexa em termos processuais e, comparativamente com a configuração do **SonarQube**, requer alguns cuidados. Por defeito, o **Nexus** traz consigo um conjunto de repositórios **Maven** pré-configurados que podem ser eliminados.

Começando pela adição de um repositório *proxy*, é atribuído o ID “nuget-gallery-proxy” e o nome “Nuget Gallery Proxy”. Como pode ser visto na figura 3.33, no campo *Provider* seleciona-se **NuGet** como formato do artefacto e coloca-se a conexão à **API** pública do **NuGet** no campo *Remote Storage Location*.

De seguida cria-se o repositório *hosted*, atribuindo como ID “nuget-releases” e como nome “Nuget Internal Releases”. À semelhança da configuração anterior, o *Provider* é o mesmo, ou seja, *NuGet*. Nas definições de acesso podem ser selecionadas as opções *Allow Redeploy* ou *Disable Redeploy* consoante se pretenda ou não permitir *overwriting* a pacotes de versões semelhantes. Para testar o repositório aconselha-se a *Policy* de *Allow Redeploy* uma vez que não bloqueia a entrada a **NuGet packages** da mesma versão sendo mais fácil de verificar o seu funcionamento.

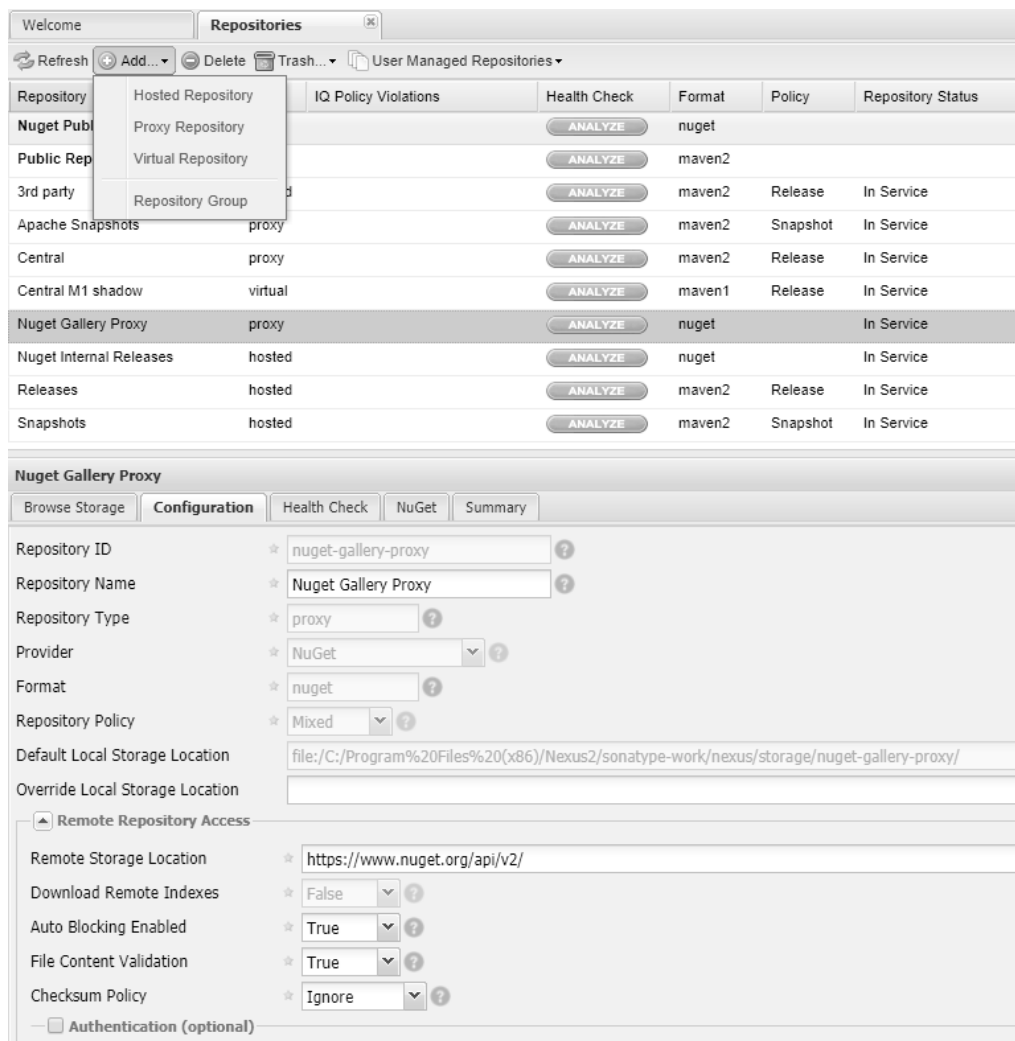


Figura 3.33 – Configuração do repositório NuGet proxy

Por último, o grupo que vai conter os dois repositórios. Para configurar o grupo de repositórios no **Nexus** são apenas necessários um ID, por exemplo “nuget-public”, e um nome semelhante, “Nuget Public”. De seguida, passam-se os repositórios *hosted* e *proxy* – presentes nos *Available Repositories* – para os *Ordered Group Repositories* (3.34).

A configuração do repositório de **NuGet packages** está praticamente concluída. Para publicar **Packages** no repositório é necessário aceder à API Key gerada pelo **Nexus**. Para tal, vai-se ao painel de administração na opção *server* e, nas *security settings*, coloca-se a *NuGet API-Key Realm* nos **Selected Realms** (3.35).

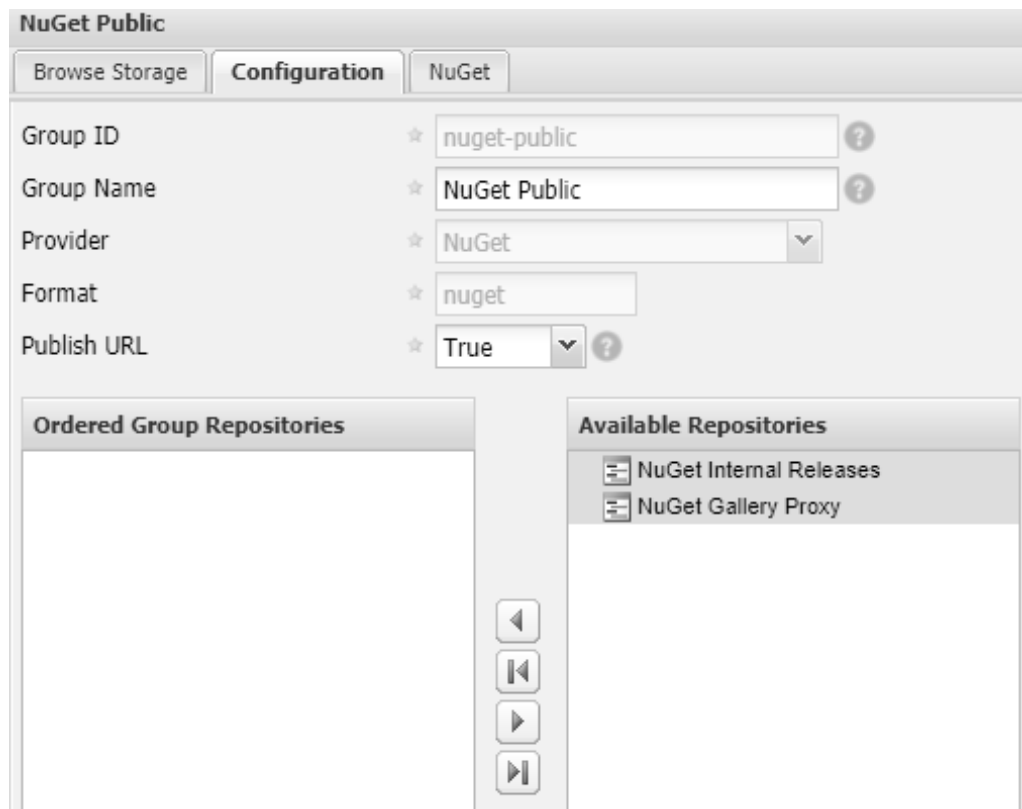


Figura 3.34 – Configuração do repositório NuGet Public

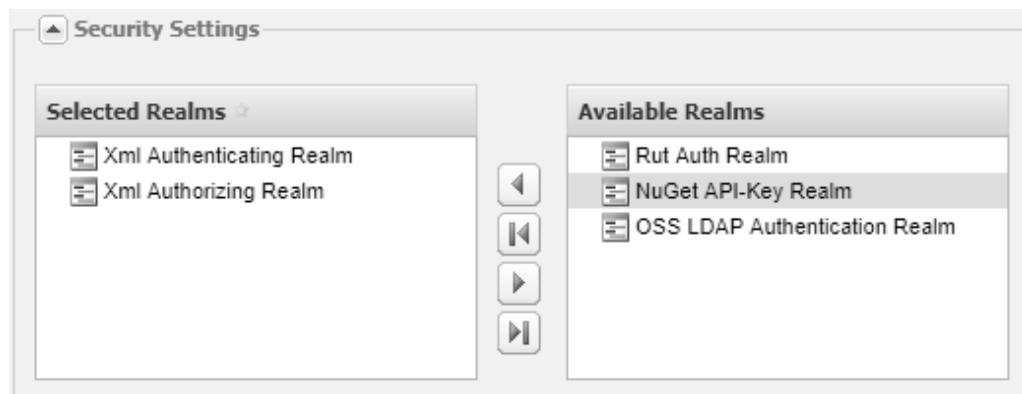


Figura 3.35 – Configurações de segurança do servidor Nexus

A **Sonatype** disponibiliza no seu canal oficial de **Youtube** um vídeo mais detalhado sobre a gestão de componentes .NET com o **Nexus** (Sonatype, 2012). A última instrução da fase de *build* vai colocar o *package* no repositório que foi configurado. Esta instrução necessita da **API Key** para autenticar contra o repositório

do **Nexus**, precisa também da hiperligação (*source*) para onde será enviado o pacote e precisa do caminho absoluto do ficheiro produzido com a instrução anterior. Pode utilizar-se uma instrução semelhante à seguinte: `dotnet nuget push`, com o caminho do *package* `package-path`, com a **API Key** `-k nexus-api-key` e com a *source* `-s hosted-repository` como opções (Docs.microsoft.com, 2019b).

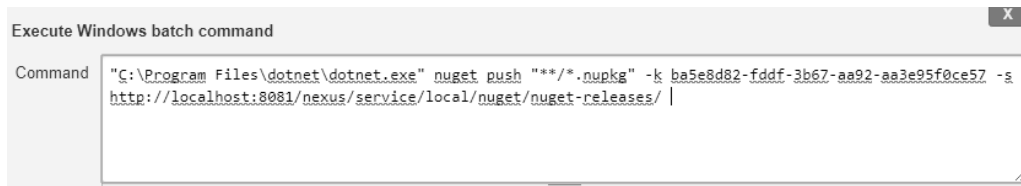


Figura 3.36 – Instrução de publicação do *package* para o Nexus

A instrução é semelhante aquela visível na figura 3.36 sendo que se pode recorrer a *wildcards* (*) para procurar ficheiros pelo seu formato. Está tudo configurado para se fazer *build* do primeiro *job* da *pipeline*. O próximo passo é o desenvolvimento de uma **Web API** que implemente a funcionalidade de soma.

3.2.7 Desenvolvimento da Web API

Para o desenvolvimento da **Web API** será utilizada uma ferramenta de documentação, o **Swagger** (Smartbear, 2019). Esta ferramenta *open source* é utilizada pela comunidade de *developers* principalmente no desenvolvimento de **APIs REST**.

Como dito anteriormente, será utilizado o **NuGet package** – *ModelClasses* – e são também necessárias algumas alterações ao código, nomeadamente a adição do serviço de documentação do **Swagger** na classe *Startup.cs*. O serviço que se vai desenvolver não incorpora uma base de dados uma vez que para testar os serviços em funcionamento não seria necessário armazenar dados da **Web API** nesta fase de desenvolvimento.

Depois do desenvolvimento da API, ainda dentro da mesma solução, será criado um projeto de teste onde vão ser desenvolvidos os testes de integração e performance aos *endpoints* disponíveis.

Pré-requisitos:

- Swashbuckle.AspNetCore (4.0.1);
- Swashbuckle.AspNetCore.Swagger (4.0.1);
- ModelClasses (1.0.3.4);
- Microsoft.AspNetCore.App;
- Microsoft.AspNetCore.App.Design;

Configuração da nova *source* no *Visual Studio*

Para ser adicionada a nova *source* no **Visual Studio** é necessário aceder a “*Tools*” e clicar em “*options*”. Depois navega-se novamente até se encontrarem as definições de configuração do “*NuGet Package Manager*” e adiciona-se a nova fonte na opção “*Package Sources*” para que possam ser incluídos no projeto os pacotes com as dependências necessárias (3.37).

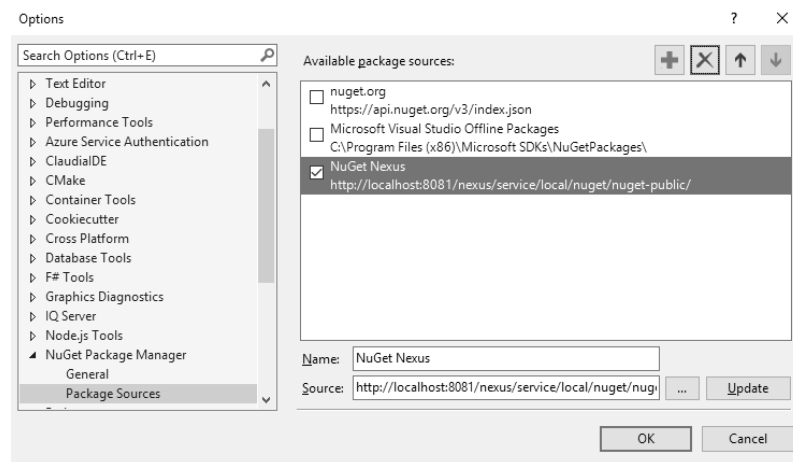


Figura 3.37 – NuGet Package Manager

Todos os pacotes estão incluídos na nova *source*, mesmo aqueles que são disponibilizados na API do **NuGet** (<https://www.nuget.org/api/v2>), porque o repositório *proxy* foi configurado para incluir a API pública de **NuGet Packages**.

Alterações ao serviço da Web API

No serviço, começa-se por editar a classe “Startup.cs” para incluir a interface do **Swagger**. Em primeiro lugar são incluídos os *packages* do **Swashbuckle** nas dependências e, de seguida, dentro da classe incluem-se os serviços do **Swagger** nos métodos de configuração do serviço (3.38) e da aplicação especificada (3.39). A configuração do **Swagger** está, uma vez mais, em detalhe na documentação oficial (Addie e Boyer, 2019) do .NET pelo que se aconselha a sua leitura.

```
public void ConfigureServices(IServiceCollection services)
{
    ... services.AddMvc()
    ... .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    ... .ConfigureApiBehaviorOptions(options =>
    ... {
    ...     options.SuppressMapClientErrors = true; //removes the full error log
    ... });
    ... services.AddSwaggerGen(c =>
    ... {
    ...     c.SwaggerDoc("v1",
    ...     new Info
    ...     {
    ...         Title = "Calculator Web App" + hosting.EnvironmentName,
    ...         Version = "v1",
    ...         Description = "Web API example, created with ASP.NET CORE",
    ...         Contact = new Contact
    ...         {
    ...             Name = "Filipe Costa",
    ...             Url = "http://localhost"
    ...         }
    ...     });
    ... });
}
```

Figura 3.38 – Estrutura do método de configuração do serviço

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ... if (env.IsDevelopment())
    ... {
    ...     app.UseDeveloperExceptionPage();
    ... }
    ... else
    ... {
    ...     // The default HSTS value is 30 days. You may want to change this for prod
    ...     // app.UseHsts();
    ... }

    ... //app.UseHttpsRedirection();
    ... app.UseMvc();

    ... // Enable middleware to serve generated Swagger as a JSON endpoint.
    ... app.UseSwagger();
    ... app.UseSwaggerUI(c =>
    ... {
    ...     c.SwaggerEndpoint("/swagger/v1/swagger.json", "Calculator App");
    ... });
}
```

Figura 3.39 – Estrutura do método de configuração da aplicação especificada

Na classe *HomeController* está incluído por defeito um método *GET* que retorna duas strings, “value1” e “value2”. Este método vai permanecer tal como está sendo apenas alterado o nome do controlador para *CalcController*. Com o **NuGet Package** incluído na classe, está tudo pronto para ser implementada a funcionalidade de soma na **Web API**.

O proximo objetivo será criar um método *POST* para somar dois números inteiros. Este método, chamado *PostNumbers*, vai receber dois números inteiros (*int n1*, *int n2*) e vai retornar o resultado da sua soma. A estrutura do método é bastante simples. É criado um novo objeto da classe *Calculator*, proveniente do **NuGet Package** (*ModelClasses*), depois é retornado o resultado do método *Add* daquele objeto. Como se pode ver na figura 3.40, os parâmetros que são enviados pela *string*, para serem adicionados, são aqueles que serão introduzidos pelo utilizador.

```
/// <summary>
/// Add two number and return the result
/// </summary>
/// <param name="n1"></param>
/// <param name="n2"></param>
/// <returns>200 OK; 404 Bad Request</returns>
// POST api/values
[HttpPost("add/{n1}/{n2}")]
public ActionResult<int> PostNumbers(int n1, int n2)
{
    Calculator calculator = new Calculator();
    return new ObjectResult(calculator.Add(n1, n2));
}
```

Figura 3.40 – Estrutura do método *PostNumbers*

Para testar a **Web API** através da interface gráfica do **Swagger** deve ser executado (F5) o projeto e no *browser*, após ser expandido o método *POST*, clica-se em “Try it out” de forma a que seja possível inserir dois números inteiros para executar a operação (3.41). Se a operação for bem sucedida é recebida uma resposta do servidor do **Kestrel** (Dykstra, Halter, e Ross, 2019) com a resposta **200 OK** juntamente com o resultado da soma, caso contrário será retornada uma resposta do tipo **404 Bad Request**.

No caso foi retornada a resposta **200 OK** acompanhada pelo resultado da soma (3.42). A implementação da funcionalidade que foi desenvolvida, testada, empacotada e instalada na Web API está concluída.

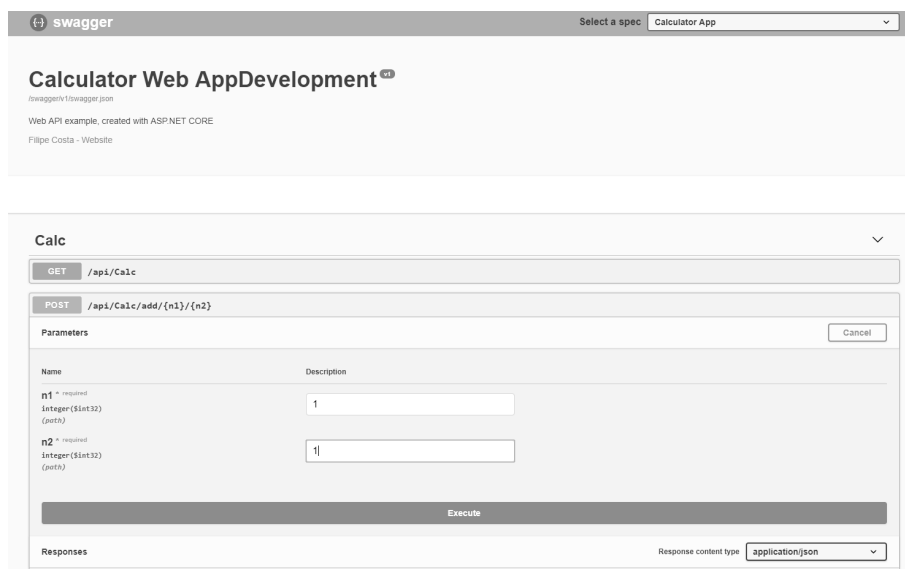


Figura 3.41 – Interface do Swagger

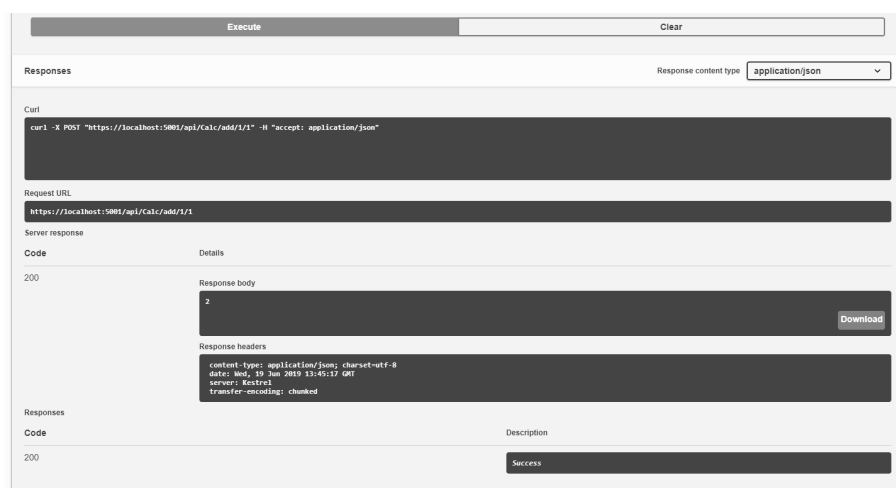


Figura 3.42 – Interface do Swagger com resposta 200 OK

A próxima fase diz respeito aos testes de integração e performance.

Criação dos testes de integração e performance

Os testes de integração e performance serão acompanhados pela criação de uma classe *TestClientProvider*. Esta classe utiliza o servidor de testes da **Microsoft** – o *TestServer* – para simular um cliente que faz pedidos à aplicação. O projeto

de teste será criado em **xUnit** e vai incluir uma referência ao projeto da **Web API**. À semelhança dos testes unitários, os testes de integração – que podem ou não incluir testes de performance – seguem a mesma metodologia, *Triple A*. No entanto, a estrutura dos testes de integração é ligeiramente mais complexa e, caso se pretenda incluir testes de performance, a sua complexidade aumenta. Para a implementação dos testes de integração serviram de apoio as páginas da documentação oficial da **Microsoft** sobre testes de integração (Latham e Smith, 2019), o blogue do **Andrew Lock** (Lock, 2016) e alguns vídeos disponibilizados no canal **Dot Net Core Central** (Choudhury, 2018) sobre criação de testes de integração.

Pré-requisitos:

- xUnit (2.4.0);
- xUnit.runner.visualstudio (2.4.0);
- ModelClasses (1.0.3.4);
- FluentAssertions (5.6.0);
- Microsoft.AspNetCore.TestHost (2.2.0);
- Microsoft.AspNetCore.HttpsPolicy (2.2.0);
- Microsoft.AspNetCore.Diagnostics (2.2.0);
- Microsoft.AspNetCore.Mvc (2.2.0);
- Newtonsoft.Json (12.0.1);

Objetivo dos testes de integração e performance

Os testes de integração são desenvolvidos quando pretendemos verificar as respostas de um pedido a um determinado *endpoint*. Imaginemos que se pretende testar o comportamento de um método **POST**, semelhante aquele implementado na

Web API, verificando que são efetivamente adicionados os dois números inteiros. Para tal, terá de se verificar que a resposta retornada é do tipo **200 OK** e que o resultado obtido é igual ao resultado esperado.

Os testes de performance vão ser incluídos na execução dos testes de integração e terão como objetivo medir o seu tempo de execução. É necessário ter em conta que a primeira execução dos testes de integração é sempre mais demorada que as seguintes uma vez que, neste contexto, estará a ser inicializado o servidor de testes antes de ser criado um cliente.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.TestHost;
using MyCalculatorWebApp;
using System;
using System.Net.Http;

public class TestClientProvider : IDisposable
{
    ... private readonly TestServer Server;

    ... public HttpClient Client { get; private set; }

    ... public TestClientProvider()
    ... {
    ...     Server = new TestServer(new WebHostBuilder().UseStartup<Startup>());
    ...     Client = Server.CreateClient();
    ... }

    ... public void Dispose()
    ... {
    ...     Server?.Dispose();
    ...     Client?.Dispose();
    ... }
}
```

Figura 3.43 – Estrutura do *TestClientProvider*

Como dito anteriormente, para simular a criação do cliente utiliza-se a classe *TestServer* que foi desenvolvida pela **Microsoft**. Desta forma, é criada uma nova classe, dentro do projeto de **xUnit**, chamada *TestClientProvider* (3.43). Esta classe vai herdar as propriedades da classe *IDisposable*, vai ter uma propriedade privada do tipo *TestServer* – que será o servidor de testes – e uma propriedade pública do tipo *HttpClient* – que será o cliente – cujo objetivo será fazer pedidos à aplicação *MyCalculatorWebApp*. Ainda dentro da classe vão constar dois métodos. O primeiro método será utilizado para criar o servidor de testes com a configuração da classe *Startup.cs* e o cliente que lhe fará pedidos. O segundo método fará *Dispose* ao

cliente e ao servidor de testes.

Para ser testada a integração do método *POST*, a *Theory* vai ter apenas um caso de teste – chamado *TestForResponseTypePOST* (3.44) – com três parâmetros inteiros, *n1*, *n2* e *result*. Na fase de *Arrange* será instanciado um novo cliente da classe *TestClientProvider*. De seguida, na fase de *Act*, é dado início à contagem do tempo de execução do método *POST* juntamente com o tempo da resposta do servidor. No final, na fase de *Assert*, verifica-se que o tempo demorado é menor do que um valor pré-estabelecido (no caso dois segundos) e depois, com base nessa condição, são feitas duas verificações. É feita uma verificação ao tipo de resposta e outra verificação ao conteúdo da mensagem. Tal como é possível ver na figura 3.44, uma das verificações é feita com o *xUnit* ao passo que a outra é feita em *FluentAssertions*.

```
[Theory]
[InlineData(1, 1, 2)]
public async Task TestForResponseTypePOST(int n1, int n2, int result)
{
    //Arrange
    using (var client = new TestClientProvider().Client) //Initializes the client
    {
        //Act
        var Initial = DateTime.UtcNow;
        var request = await client.PostAsync($"api/calc/add/{n1}/{n2}", new StringContent(
            JsonConvert.SerializeObject(new Calculator().Add(n1, n2)),
            Encoding.UTF8, "application/json"));
        request.EnsureSuccessStatusCode();
        //Test for the response
        var response = request.Content.ReadAsStringAsync().Result;
        var dif = DateTime.Now - Initial;

        //Assert
        if (dif.TotalSeconds < 2)
        {
            request.StatusCode.Should().Be(HttpStatusCode.OK); //Asserting that the request
            Assert.Equal(result.ToString(), response); //Asserting that the calculation of t
        }
    }
}
```

Figura 3.44 – Estrutura do *TestForResponseTypePOST*

A próxima fase passará pela integração do código da **Web API** numa *pipeline*, com análise estática e execução de testes de integração. Pode-se ou não integrar as notificações acerca do *status* do *job* com as plataformas de comunicação que a empresa utiliza, neste caso o **Slack**, mas o principal objetivo é fazer *build* ao projeto da **Web API** e *restore* às dependências do projeto.

Configuração do *job*

Partindo do pressuposto que o código da **Web API** foi publicado no **VCS**, serão repetidos os passos realizados na criação do *job* anterior. No **Jenkins** será criado um **new item**, do tipo *freestyle job*, podendo ser adicionada uma descrição. Depois será necessário adicionar a ligação ao **VCS**, a **GitLab Connection** que foi criada anteriormente, para que seja possível trazer para o **Jenkins** o código armazenado no repositório (3.45).

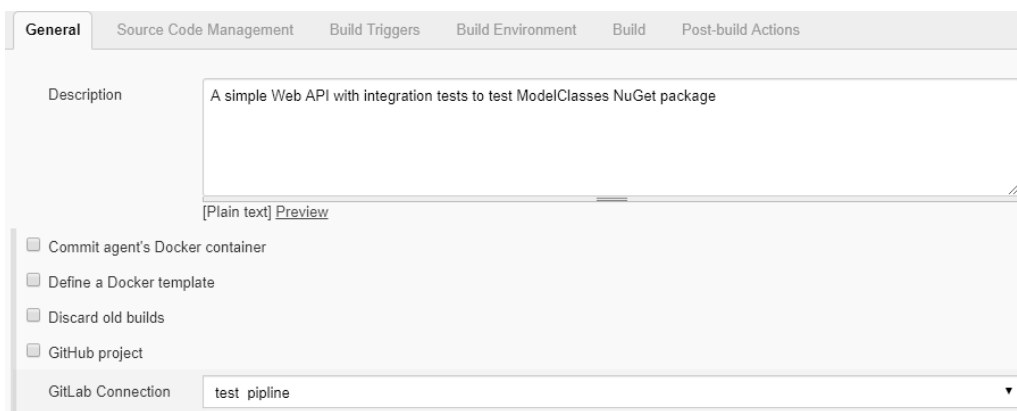


Figura 3.45 – Configuração da descrição e do *access token*

Em **Source Code Management** selecciona-se o repositório **GitLab** para onde foi publicado o código. Mais uma vez a comunicação entre o sistema de controlo de versões com **Jenkins** será feita via *SSH*.

Para que o processo de *build* do *job* que está a ser configurado esteja encadeado com o processo de *build* do *job* anterior será preciso fazer-se uma configuração na secção de **Build Triggers**. Para tal, é necessário seleccionar a opção “*Build after other projects are built*” (3.46) e, como se pretende que apenas seja feito um *trigger* caso a *build* anterior seja bem sucedida, selecciona-se também a opção “*Trigger only if build is stable*”.

No próximo segmento, em **Build Environments**, é importante não esquecer de se seleccionar a opção de “*Delete workspace before build starts*”, para que seja apagado o *workspace* criado pelo **Jenkins** antes do início de cada *build*.

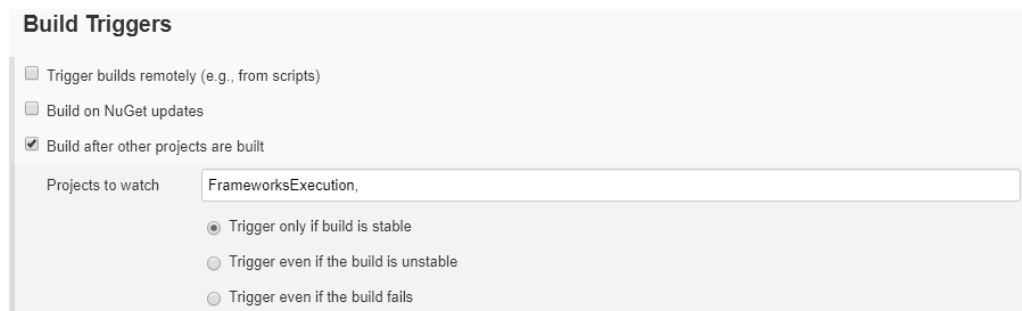


Figura 3.46 – Configuração dos *Build Triggers*

Na fase de *build* passa-se à configuração de um conjunto de instruções para integrar a análise estática, a *build* à solução da **Web API**, a execução dos testes de integração e a interpretação do resultado dos testes, tal como no *job* anterior. À semelhança daquilo que se fez, a integração da análise estática inicia-se através da instrução inicial `$ dotnet SonarScanner.MSBuild.dll begin /k:"project-key"` e termina-se com a instrução final `$ dotnet SonarScanner.MSBuild.dll end`. A *build* ao projeto é feita com a instrução `$ dotnet build project-path` com a adição das opções `--source http://package-source` onde “http://package-source” será o repositório de **NuGet Packages**. Para se executarem os testes de integração utiliza-se o mesmo `$ dotnet test project-path --no-build --logger trx`. Na figura 3.47 vêm-se as instruções de *build* à solução assim como de execução dos testes de integração.



```
"C:\Program Files\dotnet\dotnet.exe" build "%WORKSPACE%\src\MyCalculatorWebApp\ModelClassWebApp.sln" --source "http://localhost:8081/nexus/service/local/nuget/nuget-public/"
```

```
"C:\Program Files\dotnet\dotnet.exe" test "%WORKSPACE%\src\MyCalculatorWebApp\XUnitIntegrationTest\XUnitIntegrationTest.Tests.csproj" --no-build --logger trx
```

Figura 3.47 – Configuração dos *Build Steps*

Integração com Slack

Para a integração do **Jenkins** com a plataforma de comunicação interna da empresa – **Slack** – é necessária a instalação do *plugin* **Slack Notifications** (<https://plugins.jenkins.io/slack>).

Do lado do **Slack** a configuração é relativamente simples. Primeiro é necessário pedir um *token* de acesso à plataforma de comunicação. Isto é feito através do painel de configuração das aplicações **Slack**. Neste painel (3.48) pode ser criado o canal com o qual a aplicação – **Jenkins CI** – vai comunicar o estado das *builds* e é também aqui que se gera o *token* de acesso à plataforma de comunicação.

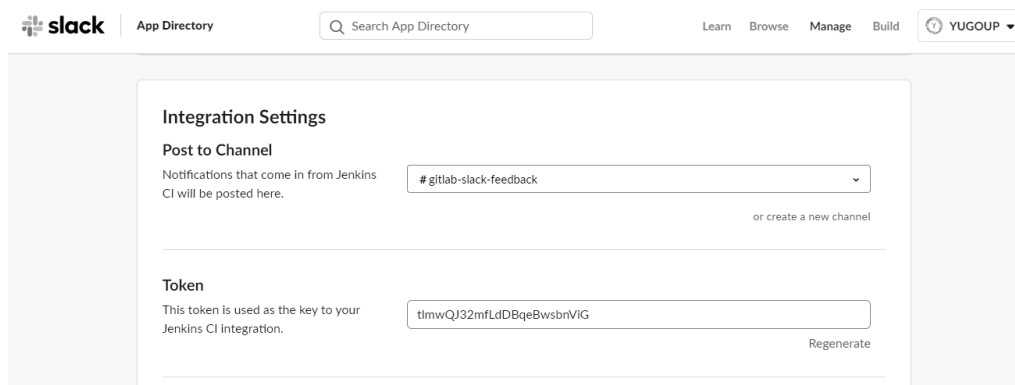


Figura 3.48 – Configuração do Jenkins CI

Do lado do **Jenkins**, como se pode ver na figura 3.49, antes da configuração da *pipeline*, é necessário adicionar o *token* de acesso gerado anteriormente, juntamente com o nome do canal para o qual será transmitida a informação.

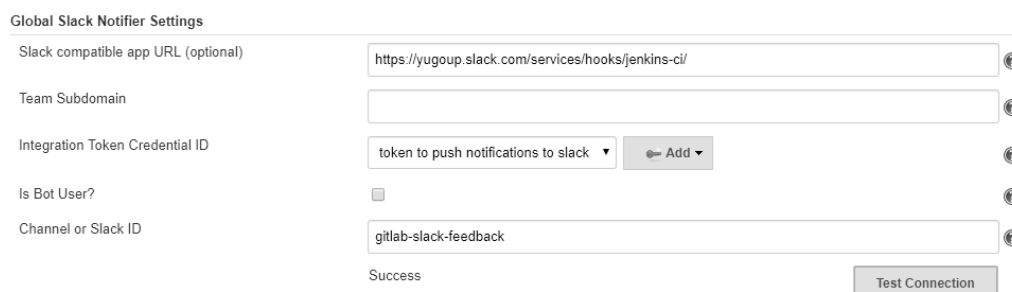


Figura 3.49 – Configuração do Slack Notifier

De seguida, e para finalizar a configuração das **Slack Notifications**, será necessário voltar ao *job*. No *job* é na fase de **Post-build Actions** que, para além de serem agregados todos dos resultados da execução dos testes, serão também adicionadas as *Slack Notifications*. Aqui seleccionam-se as opções pretendidas, desde notificações de sucesso/insucesso da *build* até a informações sobre os *commits* (3.50).

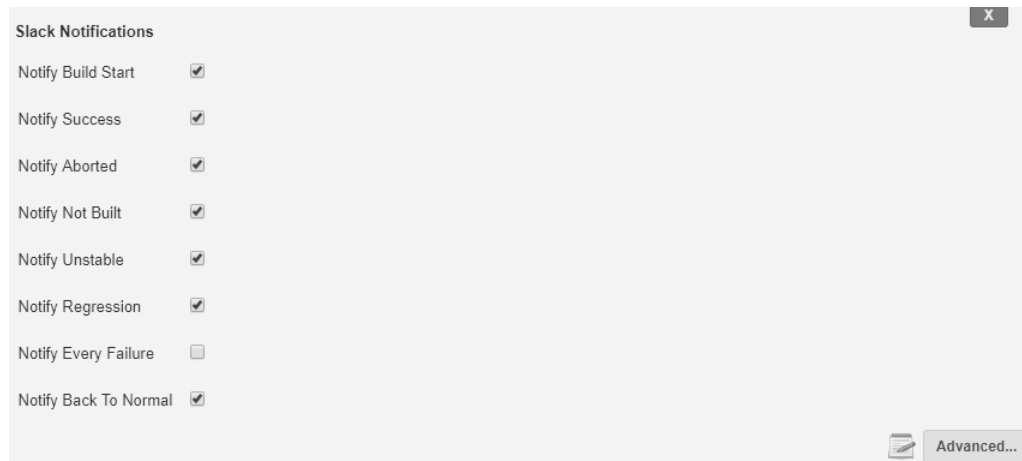


Figura 3.50 – *Post-build action de Slack Notification*

Nesta fase a *pipeline* contabiliza dois *jobs*. O primeiro *job* encarrega-se da automatização das fases de *build*, *static analysis*, *unit testing*, *packaging* e *push* do módulo de soma para um repositório de artefactos. O segundo *job* encarrega-se das fases de *build*, *static analysis*, *integration testing* e notifica o developer acerca do estado da *build* da **Web API**. A última fase da *pipeline* envolve a criação de um *job* com o propósito de automatizar a construção e publicação de uma imagem da aplicação. A imagem vai ser criada através de um ficheiro – **Dockerfile** – e a aplicação vai ser publicada através da utilização do **Docker**.

3.2.8 Publicação da Web API

Para a publicação da aplicação vai ser utilizado o **Docker**. É através desta tecnologia de orquestração de *containers* que será criada e publicada uma imagem da aplicação com recurso a registo privado de imagens. Com este registo privado podem ser geridas as imagens das aplicações que são criadas sem que mais ninguém

tenha acesso. Caso estivessem no registo público do **Docker**, estariam disponíveis para uso público.

Criação de um *Dockerfile*

O **Dockerfile** é o documento que vai ser utilizado para executar um conjunto de instruções que vão gerar uma versão de *Release* da **Web API** que foi criada (3.51). Para a imagem da **Web API** estar totalmente funcional é necessário copiar os ficheiros **.csproj** para a *working directory* – neste caso **/app** – para depois resolver as dependências do projeto que estão guardadas no repositório de artefactos. De seguida, a pasta com as dependências resolvidas é copiada para a *working directory*. Depois, novamente a partir da pasta da **Web API**, é feita *build* à solução com a configuração de **Release** e com o *output* a ser enviado para a pasta **/app**.

```
Dockerfile
1 FROM microsoft/dotnet:2.2-aspnetcore-runtime AS base
2 WORKDIR /app
3 EXPOSE 80
4
5 FROM microsoft/dotnet:2.2-sdk AS build
6 WORKDIR /src/mycalculatorwebapp/
7 COPY *.csproj ./
8 RUN dotnet restore --source http://127.0.0.1:8081/nexus/service/local/nuget/nuget-public/
9 COPY . .
10 WORKDIR /src/mycalculatorwebapp/
11 RUN dotnet build -c Release -o /app
12
13 FROM build AS publish
14 RUN dotnet publish -c Release -o /app
15
16 FROM base AS final
17 WORKDIR /app
18 COPY --from=publish /app .
19 ENTRYPOINT ["dotnet", "ModelClassWebApp.Service.dll"]
```

Figura 3.51 – Estrutura do *Dockerfile*

A partir do resultado da instrução gerada na fase de *build* vai publicar-se o conteúdo gerado para a pasta **/app** com a configuração de *Release*. Para finalizar, a partir da imagem base, volta-se novamente para a pasta **/app**, copia-se o conteúdo da instrução anterior (*publish*) para a pasta atual e passa-se o *entrypoint* que é a *dynamically linked library* da **Web API**.

Após a criação da primeira imagem vai automatizar-se todo o processo de

geração da mesma na *pipeline* de integração contínua. Claro está que para tal será necessário recorrer ao *script* gerado anteriormente.

Criação do job

O *workflow* inicial do novo *job* é ligeiramente diferente do anterior. Até à fase de **Build**, não existe qualquer alteração nas configurações. Contudo, quando se executam as instruções necessárias para automatizar o processo de geração, publicação e lançamento da imagem, terão de utilizar-se dois tipos de *build steps* distintos. Até agora têm sido utilizado maioritariamente **Windows batch command** mas, para este caso, terá de se utilizar **shell scripts**.

Setup do ambiente

Para o **setup** ao ambiente irá fazer-se *build* à solução da **Web API**. À semelhança daquilo que foi feito no *script*, também terá de ser indicada na instrução a *source* dos **NuGet Packages** (3.52). O primeiro **Build step** vai definir o *WORKSPACE* dentro do **Jenkins**. Com o ambiente definido será necessário entrar na pasta onde está presente o **Dockerfile**. A instrução para mudar de diretorias no **Windows** é `$ cd directory-path` originado de *change directory*.

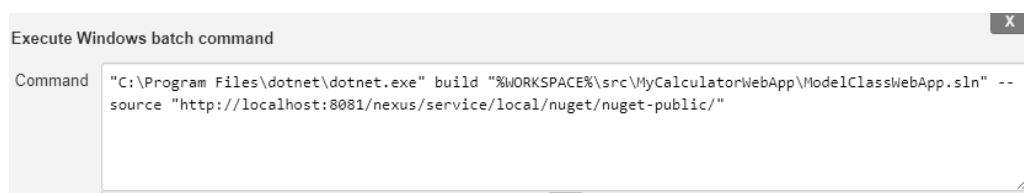
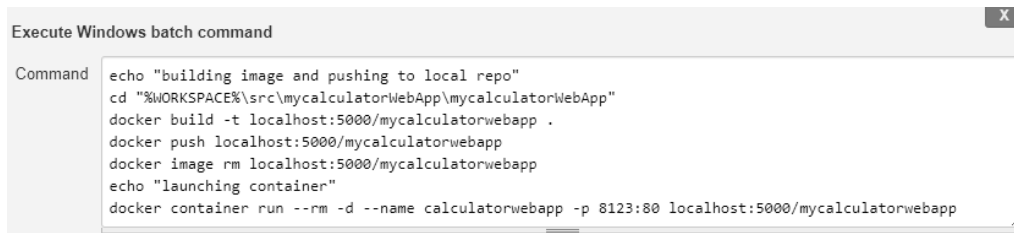


Figura 3.52 – Instrução de *build* à solução

Criação, publicação e lançamento da imagem da Web API

Para que se tenha uma ideia de como é gerada, publicada e instanciada uma imagem de **Docker** é importante que seja lida a documentação sobre estas instruções na página oficial do **Docker** sobre imagens (Docker, 2019b) e *containers* (Docker,

2019a). A sequência cronológica de instruções de publicação de uma aplicação através da utilização de **Docker CLI** pode ser vista na figura 3.53.



```
Execute Windows batch command
Command
echo "building image and pushing to local repo"
cd "%WORKSPACE%\src\mycalculatorWebApp\mycalculatorWebApp"
docker build -t localhost:5000/mycalculatorwebapp .
docker push localhost:5000/mycalculatorwebapp
docker image rm localhost:5000/mycalculatorwebapp
echo "launching container"
docker container run --rm -d --name calculatorwebapp -p 8123:80 localhost:5000/mycalculatorwebapp
```

Figura 3.53 – Criação, publicação e lançamento da imagem da Web API

A imagem da **Web API** vai ser gerada com uma **tag** do endereço local do serviço de *registry* seguido pelo seu nome `$ docker build -t registry-address/image .`. O serviço de *registry* é uma aplicação *server side stateless* escalável que nos permite armazenar e distribuir imagens **Docker** (Docker, 2019c). Existem outras alternativas para armazenamento de imagens, no entanto a implementação implicaria a emissão de um certificado **SSL self-signed** e para o caso pretende manter-se o processo o mais simples possível.

A próxima instrução será utilizada para colocar a imagem dentro do **Docker registry**. Para tal, utiliza-se `$ docker push registry-address/image` e, para garantir que a instrução funciona corretamente, é necessário verificar que a **tag** da imagem é igual à **tag** da instrução anterior.

De seguida, remove-se a imagem que foi criada na máquina com recurso à instrução `$ docker image rm registry-address/image`. Pode ser necessário utilizar a opção `--force` para que a imagem seja completamente removida do **Docker** local. Depois, para publicar num *container* a **Web API** a partir da imagem armazenada no registo, será utilizada a instrução `$ docker container run [options] image` juntamente com algumas opções para lançar o *container* na porta **8123**.

Pedidos à Web API

Depois de publicada a imagem vão ser feitos alguns pedidos à aplicação. Para tal, são utilizadas as instruções do **curl** (<https://curl.haxx.se/>) tal como pode ser

visto na figura 3.54. Para garantir que a aplicação é devidamente publicada e só depois são feitos os pedidos aguarda-se durante 15 segundos com a instrução `$ sleep 15`. Só depois são executadas as instruções para os pedidos **GET** e **POST** sendo que o *output* pode ser verificado diretamente no **Jenkins**.

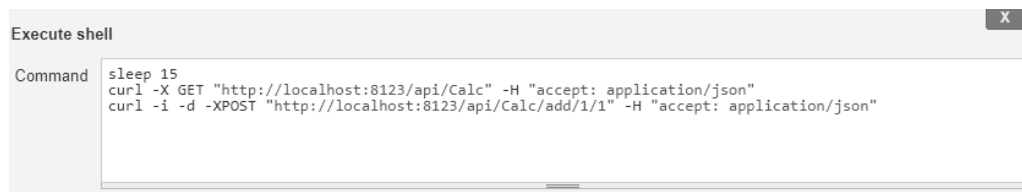


Figura 3.54 – Requests à Web API

Depois de feitos os *requests* e validado o comportamento da **Web API**, irá ser parado o *container* que foi lançado através da utilização da instrução de paragem `$ docker container stop container-name` (3.55).

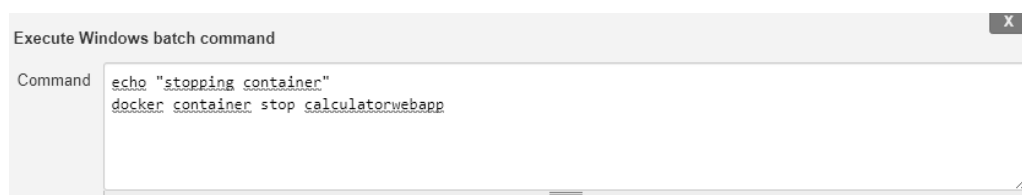


Figura 3.55 – Instrução de paragem do *container*

3.3 Considerações

Recapitulando o trabalho feito neste capítulo, que envolveu:

- Configuração de um sistema de controlo de versões;
- Desenvolvimento um conjunto de testes unitários para validação do funcionamento do módulo de soma;
- Configuração um orquestrador de processos (**Jenkins**);
- Configuração de um *job* para automatizar as fases de *build*, *static analysis*, *test*, *pack* e *push* de um **NuGet Package** para um repositório de artefactos;

- Descrição do processo de desenvolvimento de uma **Web API**;
- Configuração de um *job* para automatizar as fases de *build*, *static analysis*, *integration testing* e notificação do *developer* acerca do estado da *build* e dos detalhes da **Web API**;
- Descrição do processo de geração, publicação e lançamento de uma imagem da **Web API**;
- Configuração de um *job* para automatizar a fase de *build*, *publish* e *run* da imagem **Docker** da aplicação;

O último passo consiste na criação de um ficheiro de composição dos serviços. Este ficheiro, semelhante ao da figura (3.56), iria conter dois serviços da **Web API** que foi criada, com a adição de um serviço de *load balancing* para distribuição dos pedidos. Para fazer a distribuição dos pedidos à aplicação por dois *containers* diferentes foi utilizado como *Web Server* o **Nginx** (<https://www.nginx.com/>).

```
mycalculatorwebapp ▸ mycalculatorwebapp ▸ docker-compose.yml
1  version: '3.4'
2
3  services:
4    nginx:
5      image: localhost:5000/loadbalancer
6      build:
7        context: ../nginx
8      depends_on:
9        - app
10       - app2
11      ports:
12        - 80:80
13
14    app:
15      working_dir: /app
16      image: localhost:5000/mycalculatorwebapp
17      environment:
18        - ASPNETCORE_ENVIRONMENT=test
19      build:
20        context: .
21      ports:
22        - 8123:80
23
24    app2:
25      working_dir: /app
26      image: localhost:5000/mycalculatorwebapp
27      environment:
28        - ASPNETCORE_ENVIRONMENT=prod
29      build:
30        context: .
31      ports:
32        - 8124:80
```

Figura 3.56 – Estrutura do ficheiro *docker-compose.yml*

Em termos de *workflow* a **Web API** construída incorpora três fases de desenvolvimento. A primeira fase é a fase de *development* onde se desenvolveu o módulo de soma através de práticas de **TDD**. Os testes são incorporados mais tarde numa *pipeline* de integração contínua, são analisados pelo servidor de automação e, caso sejam bem-sucedidos, é gerado um artefacto daquele módulo que posteriormente será colocado no repositório de artefactos.

Na fase seguinte o módulo de soma foi utilizado por uma **Web API** desenvolvida para se verificar que os testes de integração podiam ser feitos através da utilização de uma tecnologia da **Microsoft**, o *TestServer*. Os testes de integração também testam a performance da aplicação, mas é necessário ter em conta que a primeira execução vai ser sempre mais lenta dado que é necessário inicializar o servidor de testes.

O **Jenkins** produz depois um *report* detalhado da análise dos testes unitários e de integração que é apresentado ao utilizador para fins estatísticos.

Na terceira fase de desenvolvimento – a fase de *staging* – é produzido o **Dockerfile** que é utilizado para gerar uma imagem. Esta imagem, que no fundo é a versão de *release* (executável) da aplicação, vai ser publicada no registo privado do **Docker** que foi criado para servir de repositório de imagens. Ainda neste fase, a imagem da aplicação é lançada dentro de um *container* e são feitos dois pedidos à aplicação para verificar que funciona de forma correta. Caso estes pedidos retornem *StatusCodes* diferentes do esperado a aplicação não estará a funcionar como é esperado e requer investigação.

Na última fase – *Pre-live* – coloca-se a imagem da aplicação criada, juntamente com uma imagem do *load balancer*, dentro de um ficheiro **docker-compose.yml**.

O próximo passo é a criação de uma imagem funcional de um serviço composto por uma arquitetura semelhante aquela utilizada pela empresa no desenvolvimento dos seus serviços. Depois da apresentação da solução é ainda apresentado um conjunto de serviços que o **Docker** disponibiliza.



Aplicação do protótipo aos serviços da empresa

4.1 Serviço de Gestão de tarefas

Para evidenciar os benefícios da utilização do **Docker** na orquestração de serviços foi criado um serviço de gestão de tarefas. Este serviço utiliza uma arquitetura base semelhante aquela utilizada pela equipa de desenvolvimento da **Yugoup** na construção dos serviços quem compõem a plataforma.

Pré-requisitos

- Microsoft.AspNetCore.App (2.2.0);
- Microsoft.AspNetCore.Razor.Design (2.2.0);
- Microsoft.EntityFrameworkCore (2.2.4);
- Microsoft.EntityFrameworkCore.SqlServer (2.2.4);
- Newtonsoft.Json (12.0.2);
- Swashbuckle.AspNetCore (4.0.1);

- Swashbuckle.AspNetCore.Swagger (1.0.1);

Em termos de modelação o serviço de gestão de tarefas tem uma estrutura de dados semelhante à da figura 4.1, com uma tabela *Employee* e outra tabela *Task*. Cada *Employee* pode ter uma ou mais *Tasks* associadas e cada task só pode estar atribuída a um *Employee*.

Employee		
Id	integer	
NOT NULL	FirstName	varchar(100)
NOT NULL	LastName	varchar(100)
NOT NULL	Address	varchar(250)
	HomePhone	varchar(20)
	CellPhone	varchar(20)
NOT NULL	TaskList	varchar(Task)
Add field		

Task		
Id	integer	
NOT NULL	TaskName	varchar(100)
Add field		

Figura 4.1 – Modelo do Serviço de gestão de tarefas

4.1.1 Arquitetura da aplicação

A arquitetura da aplicação não é 100% idêntica à arquitetura que a empresa utiliza nos serviços da plataforma **Yugoup**. No entanto, o objetivo não é replicar um serviço com precisamente a mesma arquitetura do serviço *Tenant*. O objetivo é sim publicar um serviço que inclua as camadas de tratamento de dados que a **Web API** da calculadora não inclui enquanto se aproxima, em termos de complexidade, do nível de desenvolvimento pretendido. A aplicação está organizada em seis *layers* (4.2): **Client SDK**, **Presentation**, **Application**, **Domain**, **Data** e **Infrastructure** sendo que as camadas **Client SDK** e **Infrastructure** não serão utilizadas. É na camada **Presentation** que estará contida a API do serviço de gestão de tarefas e é a camada de **Data** vai modelar a base de dados do serviço. As camadas – **Application** e **Domain** – são o *middleware* entre as camadas de **Presentation** e **Data**.

Não será aprofundado o desenvolvimento do serviço em si, nem é esse o objetivo da apresentação deste serviço. Com a construção deste serviço pretende-se

demonstrar que, em termos de conceito, o desenvolvimento com **Docker** pode ser facilmente integrado nas práticas de qualquer empresa de desenvolvimento de software.

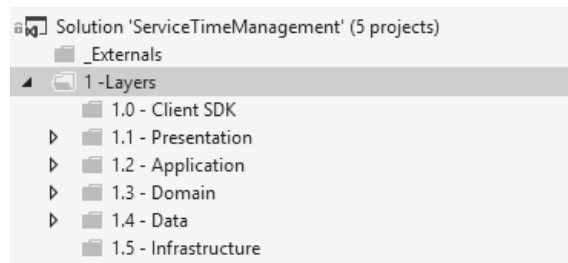


Figura 4.2 – Organização do serviço em camadas

Para que possa ser criada uma imagem do *Docker* são necessários os *paths* dos projetos de **.NET** (**.csproj**) que compõem a solução. Uma vez que estes projetos estão associados a ficheiros que são utilizados para executar a aplicação, os mesmos terão de estar referenciados algures no *script*. Como se pode ver na figura 4.3, a solução contabiliza um total de cinco destes ficheiros. Os projetos serão utilizados no *Dockerfile*, juntamente com a solução do serviço, para se poder criar uma versão de *Release* da **Web API** do serviço de gestão de tarefas e assim gerar a versão final de publicação da aplicação.

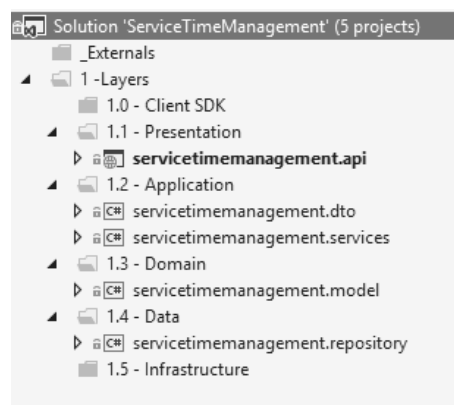


Figura 4.3 – Organização dos projetos

A execução da aplicação a partir do ambiente integrado de desenvolvimento é apresentada no *browser* através da interface gráfica do **Swagger** (4.4), que é semelhante à apresentação do serviço *Tenant*, com os métodos da respetiva **Web**

API do serviço de gestão de tarefas. A interface gráfica comunica com o serviço de armazenamento de dados do **SQL Server** dentro de um *container*. Este método também foi testado no serviço *Tenant*, como poderá ser visto mais à frente.

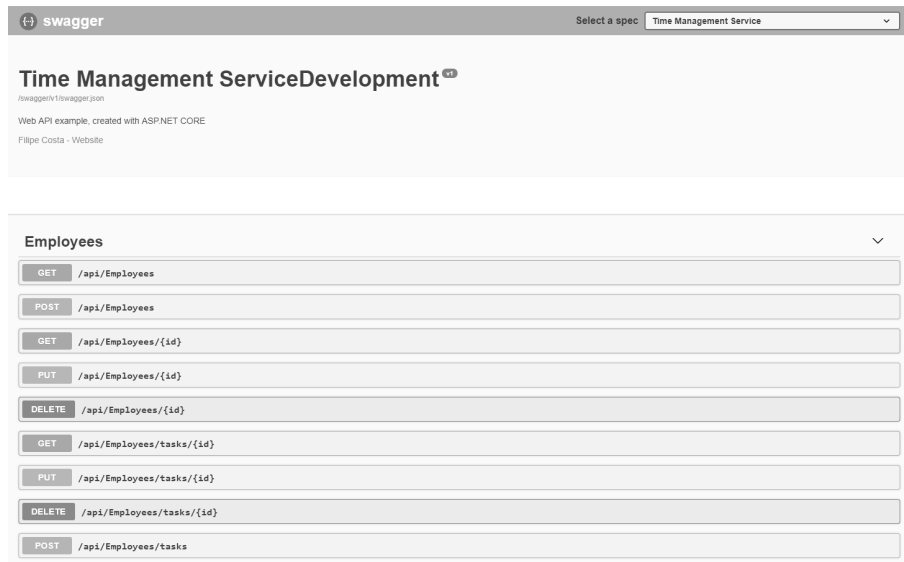


Figura 4.4 – Interface *Swagger* do serviço de gestão de tarefas

Quando o serviço é inicializado as migrações são feitas para a instância do **SQL Server** que está a ser executada no *container*, sendo gerada a base de dados **ServiceTimeManagementContext** com três tabelas (4.5), “*Employee*”, “*Task*” e “*EFMigrationsHistory*”, a tabela das migrações do **Entity Framework Core**.

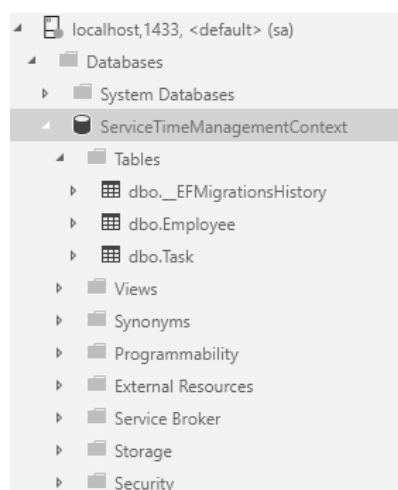


Figura 4.5 – Estrutura da base de dados do *container*

Tal como pode ser visto na figura 4.6, na classe *startup.cs* do serviço de gestão de tarefas, foi adicionado um sufixo ao título da interface gráfica do **Swagger** com o ambiente atual da aplicação. Neste caso, a aplicação executada a partir do **IDE** é lançada com as configurações do ambiente de **Development** (4.4), “*Time Management ServiceDevelopment*”.

```
public class Startup
{
    /// <summary>The constructor of the startup class</summary>
    /// <param name="configuration"></param>
    private readonly IHostingEnvironment hosting;
    public Startup(IConfiguration configuration, IHostingEnvironment _hosting)
    {
        Configuration = configuration;
        hosting = _hosting;
    }

    /// <summary>IConfiguration interface's instance</summary>
    public IConfiguration Configuration { get; }

    /// This method gets called by the runtime. Use this method to add services to the container.
    /// <summary>Adds the database context with the defined connection string</summary>
    /// <param name="services"></param>
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2) // Give me the 2.2 behaviours
            .ConfigureApiBehaviorOptions(options =>
            {
                options.SuppressMapClientErrors = true; // removes the full error log
            });
        services.AddDbContext<ServiceTimeManagementContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddScoped<IService, ServiceTimeManagement.Services.Service>();
        services.AddScoped<IRepository, ServiceTimeManagement.Repository.Repository>();

        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1",
                new Info
                {
                    Title = "Time Management Service" + hosting.EnvironmentName,
                    Version = "v1",
                    Description = "Web API example, created with ASP.NET CORE",
                    Contact = new Contact
                    {
                        Name = "Filipe Costa",
                        Url = "url"
                    }
                });
        });
    }
}
```

Figura 4.6 – Estrutura da classe *startup.cs*

4.1.2 Criação do *Dockerfile*

Na criação do *Dockerfile* do serviço foi utilizado o *script* que se pode ver na figura 4.7. A imagem gerada terá a **tag** do registo seguido do seu nome – **localhost:5000/service-time-management** – e será, mais tarde, publicada no registo privado de imagens.

```
Dockerfile
1  FROM microsoft/dotnet:2.2-aspnetcore-runtime AS base
2  WORKDIR /app
3  EXPOSE 80
4
5
6  FROM microsoft/dotnet:2.2-sdk AS build
7  WORKDIR /src/
8  # Copy all .csproj files to the app folder
9  COPY ServiceTimeManagement.sln ./
10 COPY servicetimemanagement/*.csproj ./servicetimemanagement/
11 COPY servicetimemanagement.dto/*.csproj ./servicetimemanagement.dto/
12 COPY servicetimemanagement.model/*.csproj ./servicetimemanagement.model/
13 COPY servicetimemanagement.repository/*.csproj ./servicetimemanagement.repository/
14 COPY servicetimemanagement.services/*.csproj ./servicetimemanagement.services/
15
16 COPY . .
17
18 WORKDIR /src/servicetimemanagement/
19 RUN dotnet build -c Release -o /app
20
21 FROM build AS publish
22 RUN dotnet publish -c Release -o /app
23
24 FROM base AS final
25 WORKDIR /app
26 COPY --from=publish /app .
27 ENTRYPOINT ["dotnet","servicetimemanagement.api.dll"]
```

Figura 4.7 – Estrutura do script de *Docker*

O serviço de gestão de tarefas, depois de publicado, apresenta o sufixo **Production** junto ao título (4.8) uma vez que é este o valor que está definido como variável de ambiente para o **HostingEnvironment** de *Release*.

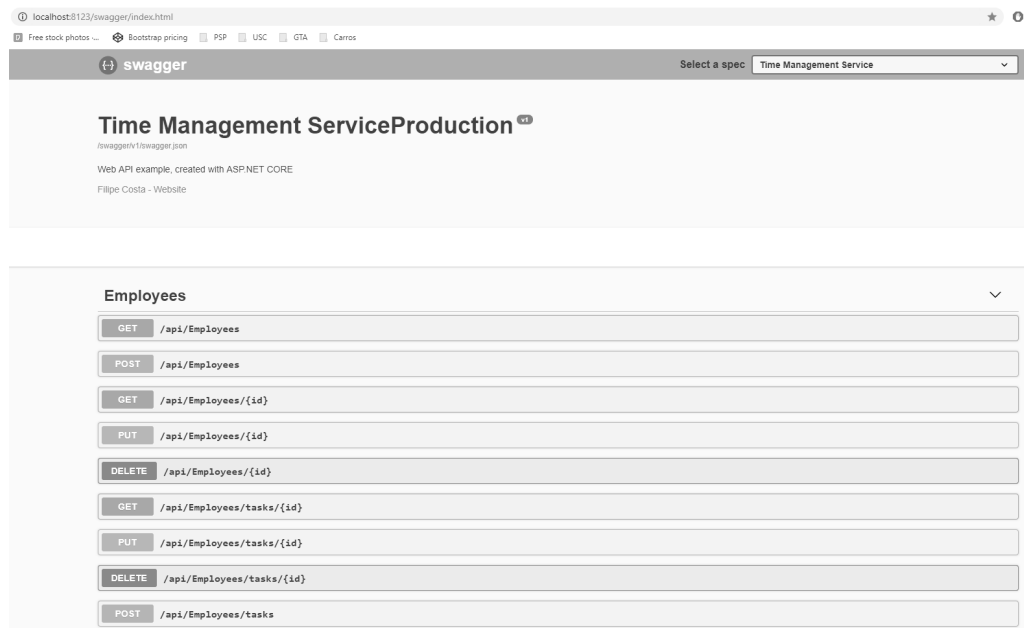


Figura 4.8 – Interface *Swagger* do serviço de gestão de tarefa

4.1.3 Criação do `docker-compose.yml`

A criação de um ficheiro de lançamento da aplicação para *pre-live* vai envolver a execução de quatro *containers*. O primeiro *container* a ser lançado será o da base de dados porque sem ela a aplicação simplesmente não funciona. De seguida são executados dois *containers* com duas aplicações da **Web API** do serviço de gestão de tarefas, em duas portas diferentes e com *HostingEnvironments* diferentes. Por último lugar, é executado o serviço de *load balancing* da aplicação.

Serviço de base de dados

O serviço de base de dados utilizado – **MS SQL Server** – é o primeiro a ser executado e todos os restantes serviços dependem do seu funcionamento. Convém ter em conta que os dois serviços da **Web API** devem aguardar que a base de dados esteja lançada (4.9) e devidamente configurada para, só depois, poderem arrancar. Para ultrapassar este problema é necessário garantir que o serviço da **Web API** só é iniciado depois de ser estabelecida uma conexão ao *container* da base de dados.

```

1  version: '3.4'
2
3  services:
4    db:
5      container_name: sqlserver
6      image: microsoft/mssql-server-linux:2017-latest
7      environment:
8        - ACCEPT_EULA=Y
9        - SA_PASSWORD=DockerSql2017!
10       - MSSQL_PID=Express
11 | # command: ["//waitforit.sh"] necessario criar uma imagem sqlserver personalizada com este file https://github.com/mcmoe/mssqldocker
12 ports:
13   - 1433:1433

```

Figura 4.9 – Serviço de base de dados da aplicação

Para este caso a solução passou pela implementação de um ciclo de tentativas de conexão ao serviço de base de dados com recurso a um *try-catch*. Como pode ser visto na figura 4.10, no bloco *try* vai ser utilizado o contexto da base de dados que foi configurado para serem feitas as migrações. Enquanto que na cláusula *catch* é feito um ciclo de 10 tentativas de conexão à base de dados.

```

public static void Main(string[] args)
{
    ... var host = CreateWebHostBuilder(args);
    ... using (var scope = host.Services.CreateScope())
    ... {
    ...     var services = scope.ServiceProvider;
    ...     int tries = 0;
    ...     while (tries < 10)
    ...     {
    ...         try
    ...         {
    ...             ... var applicationDbContext = services.GetRequiredService<ServiceTimeManagementContext>(); //muda ApplicationDbContext
    ...             ... applicationDbContext.Database.Migrate();
    ...             ... break;
    ...         }
    ...         catch (Exception ex)
    ...         {
    ...             ... var logger = services.GetRequiredService<ILogger<Program>>();
    ...             ... logger.LogError(ex, "An error occurred seeding the DB.");
    ...             ... if (tries > 10)
    ...             ... {
    ...                 ... break; //sai do ciclo
    ...             ... }
    ...             ... else
    ...             ... {
    ...                 ... tries++; //continua
    ...                 ... logger.LogError(ex, "\n-----Retrying Database Upgrade-----\n");
    ...                 ... Thread.Sleep(5000);
    ...             ... }
    ...         }
    ...     }
    ... }
    ... host.Run();
}

```

Figura 4.10 – Verificação da configuração do Serviço de Base de dados

Serviço de Web API

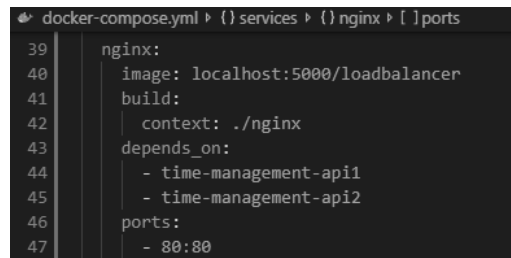
Os dois serviços da **Web API** dependem do serviço de BD para poderem funcionar. O serviço *time-management-api1* apresenta a variável de ambiente com a configuração de *Release* por defeito. Já o serviço *time-management-api2* apresenta uma variável de ambiente diferente ao utilizador apesar de ter a mesma configuração de *Release* do seu par. Como se pode ver pela figura 4.11, nas variáveis de ambiente **time-management-api2**, o valor de *HostingEnvironment* daquela instância é definido como “*Test*” através da utilização da variável de ambiente **ASPNETCORE_ENVIRONMENT=Test**. Assim sendo, quando for utilizado o *load balancer* para distribuir os pedidos pelas duas **Web APIs** sabe-se, através do sufixo apresentado no título, qual a instância da API que está a ser apresentada no *browser*.

```
docker-compose.yml > {} services > {} nginx > [] depends_on
15 |   time-management-api1:
16 |     container_name: service-time-management-prod
17 |     working_dir: /app
18 |     image: localhost:5000/service-time-management
19 |     build:
20 |       context: .
21 |     ports:
22 |       - 8123:80
23 |     depends_on:
24 |       - db
25 |
26 |   time-management-api2:
27 |     container_name: service-time-management-testing
28 |     working_dir: /app
29 |     image: localhost:5000/service-time-management
30 |     environment:
31 |       - ASPNETCORE_ENVIRONMENT=Test
32 |     build:
33 |       context: .
34 |     ports:
35 |       - 8124:80
36 |     depends_on:
37 |       - db
```

Figura 4.11 – Serviço de apresentação da **Web API**

Serviço de *Load Balancing*

Da mesma forma que os serviços da **Web API** dependem do serviço de base de dados para poderem ser inicializados, os serviço de *load balancing* vai esperar que as duas APIs estejam lançadas para poder ser executado. Pela figura 4.12 pode verificar-se que o **nginx** é o último serviço a ser executado.



```
39 nginx:
40   image: localhost:5000/loadbalancer
41   build:
42     context: ./nginx
43   depends_on:
44     - time-management-api1
45     - time-management-api2
46   ports:
47     - 80:80
```

Figura 4.12 – Serviço de *load balancing*

Após a fase de *development*, os passos para a publicação de um serviço repetem-se sistematicamente. A seguir à criação das imagens que vão compor o serviço nas fases de *pre-live* e *production*, é sempre produzido um ficheiro de composição (**docker-compose.yml**) que vai orquestrar a publicação da aplicação, serviço a serviço, pela sequência cronológica de instruções de publicação que lhe são transmitidas.

4.1.4 Considerações

Este tipo de metodologia de publicação pode ser utilizado localmente ou online, com uma condição. Na máquina onde o serviço é publicado, terá de estar instalado o **Docker**.

Caso o serviço esteja publicado em rede local, como por exemplo, na máquina de *staging* disponibilizada pela empresa, os custos de operação passam a ser única e exclusivamente a energia consumida pela máquina, contrariamente ao que acontece quando um serviço é publicado online para ser testado. Uma das principais vantagens é a facilidade com que se consegue replicar toda a infra-estrutura, o que facilita não só o desenvolvimento em si, mas também a *pipeline* de teste, porque desta forma é possível criar e destruir os ambientes facilmente.

Um dos contratempos da utilização do **Docker** (versão do **Windows**) é a perda de dados em situações como, por exemplo, quando uma máquina é reiniciada sem existir persistência de dados. Estes dados nunca mais serão recuperados, uma vez que por definição os *containers* são voláteis. Assim sendo, para contornar este

problema é necessário utilizar *containers* para guardar os dados das aplicações que estão a ser utilizadas, e utilizar os seus volumes como repositórios de dados. Desta forma os dados ficarão sempre persistidos na memória dos outros *containers*, mesmo que o serviço do **Docker** falhe inesperadamente.

Os próximos passos serão a implementação de uma *pipeline* de integração e entrega contínua para um serviço da plataforma Yugoup. Para tal, será criada uma *branch* paralela à *Master* – chamada **cicd** – que permitirá trabalhar em conformidade com os objetivos estabelecidos pela equipa de orientação sem interferir com os desenvolvimentos da equipa da empresa. O serviço mais importante da plataforma, o *Tenant*, foi o escolhido para percorrer todas as fases de integração contínua.

4.2 Plataforma Yugoup

A plataforma de comércio eletrónico **Yugoup** foi desenvolvida segundo o estilo de arquitetura orientada a serviços (por camadas) e contabiliza 36 serviços, no momento de escrita do documento. A arquitetura dos serviços é constituída por 6 camadas – *top-to-bottom*: **Client SDK**, **Presentation**, **Application**, **Domain**, **Data** e **Infrastructure** – e inclui testes unitários às camadas de **Application**, **Domain** e **Data**. Para esta fase, o objetivo seria construir uma *pipeline* de testes automatizada. Os desafios seriam a integração de análise estática e o desenvolvimento de testes de integração e performance num serviço disponibilizado, mais concretamente na camada de **Presentation**.

A empresa disponibilizou um serviço (*Tenant*) para integração contínua numa *pipeline* de testes automatizada com a execução dos testes unitários pré-existentes. Os serviços foram publicados, pela equipa da **Yugoup** numa máquina, em rede local, dentro do espaço de trabalho o que significa que os testes de integração e performance seriam feitos diretamente contra o serviço *t-yugoup-tenant* publicado em *staging*.

4.2.1 Integração contínua do serviço *Tenant*

Os testes de integração e performance serão desenvolvidos e integrados, juntamente com a análise estática ao código, na instância do **Jenkins** previamente instalada e configurada na máquina de *staging*.

4.2.2 Desenvolvimento dos testes de integração e performance

Foram desenvolvidos quatro testes de integração e dois testes de performance ao serviço *Tenant*. Como é possível ver na figura 4.13, os testes de integração utilizam uma classe – **ClientExtensions** – cuja função é criar o cliente. Esta classe contém um método – **CreateClient** – que recebe como parâmetro uma *string*. Essa *string* é o **URL** do serviço do *Tenant* que foi publicado localmente em *staging*.

```
public class ClientExtensions
{
    ... public async static Task<HttpClient> CreateClient(string url)
    ... {
        ... string client = "system";
        ... string secret = "secret";

        ... HttpClient _result = new HttpClient().{ .BaseAddress = new Uri(url) };

        ... HttpClient defaultClient = new HttpClient().{ .BaseAddress = new Uri("http://192.168.12.150:60001") };
        ... var response = await defaultClient.GetAsync($"api/v0.1/tools/token/{client}/{secret}");

        ... if (response.IsSuccessStatusCode)
        ... {
            ... var token = JsonConvert.DeserializeObject<Token>(await response.Content.ReadAsStringAsync());
            ... _result.DefaultRequestHeaders.Add("Authorization", $"Bearer {token.access_token}");
            ... }

        ... return _result;
    }
}

public class Token
{
    ... public string access_token;
}
```

Figura 4.13 – Classe *ClientExtensions*

O primeiro teste de integração foi criado para testar a existência do **GUID** – *Global Unique Identifier* – que é gerado automaticamente quando são feitas as

migrações, logo após a criação das bases de dados. O cliente gera um *token* de autenticação para o **Tenant** e as verificações são feitas ao tipo de resposta e ao conteúdo da mensagem (4.14).

```
[Theory]
[InlineData(HttpStatusCode.OK, "00000000-0000-0000-0000-000000000001")]
public async Task TestingDefaultGuid(HttpStatusCode statusCode, string tenantId)
{
    ...HttpClient client = ClientExtensions.CreateClient("http://192.168.12.150:60035").GetAwaiter().GetResult();
    ...//Arrange
    ...using (var testClient = client)
    ...{
    ...    ...//Act
    ...    ...var response = await client.GetAsync($"api/v0.1/tenants/{tenantId}");

    ...    ...string content = response.Content.ReadAsStringAsync().GetAwaiter().GetResult();
    ...    ...//Assert
    ...    ...Assert.Equal(response.StatusCode, statusCode);
    ...    ...Assert.Equal("00000000-0000-0000-0000-000000000001", tenantId);
    ...}
}
```

Figura 4.14 – Teste à presença do *Default GUID*

O segundo teste de integração foi criado para testar um GUID inexistente. O cliente gera um *token* de autenticação para o **Tenant** e as verificações são feitas ao tipo de resposta e ao conteúdo da mensagem (4.15) à semelhança do teste anterior.

```
[Theory]
[InlineData(HttpStatusCode.NotFound, "00000000-0000-0000-0000-000000000000")]
public async Task TestingMissingGuid(HttpStatusCode statusCode, string tenantId)
{
    ...HttpClient client = ClientExtensions.CreateClient("http://192.168.12.150:60035").GetAwaiter().GetResult();
    ...//Arrange
    ...using (var testClient = client)
    ...{
    ...    ...//Act
    ...    ...var response = await client.GetAsync($"api/v0.1/tenants/{tenantId}");

    ...    ...string content = response.Content.ReadAsStringAsync().GetAwaiter().GetResult();
    ...    ...//Assert
    ...    ...Assert.Equal(response.StatusCode, statusCode);
    ...    ...Assert.Equal("00000000-0000-0000-0000-000000000000", tenantId);
    ...}
}
```

Figura 4.15 – Teste à presença de um *GUID* inexistente

O terceiro teste de integração tem uma componente de teste à performance na medida em que é testado o tempo de resposta do pedido de acesso à aplicação com a *property* **enabled=true** presente na mensagem (4.16).

O quarto teste de integração tem uma componente de teste à performance, semelhante ao teste anterior, na medida em que é testado o tempo de resposta do

```
[Fact]
public async Task<TestingPostRequestAuthorizingResponse>()
{
    ...HttpClient client = ClientExtensions.CreateClient("http://192.168.12.150:60035").GetAwaiter().GetResult();
    ...//Arrange
    ...using (var testclient = client)
    ...{
    ...    //Act
    ...    var Initial = DateTime.UtcNow;
    ...    var request = await client.PostAsync($"/api/v0.1/tenants", new StringContent(JsonConvert.SerializeObject(new { enabled = true }, Encoding.UTF8, "application/json")));
    ...    var response = request.Content.ReadAsStringAsync().Result;
    ...
    ...    var dif = DateTime.UtcNow - Initial;
    ...    request.StatusCode.Should().Be(HttpStatusCode.Created);
    ...    dif.TotalMilliseconds.Should().BeLessOrEqualTo(1000);
    ...}
}
```

Figura 4.16 – Teste de acesso autorizado

pedido de acesso à aplicação sem a *property* `enabled=true` presente na mensagem (4.17).

```
[Fact]
public async Task<TestingPostRequestUnauthorizedResponse>()
{
    ...HttpClient client = ClientExtensions.CreateClient("http://192.168.12.150:60035").GetAwaiter().GetResult();
    ...//Arrange
    ...using (var testclient = client)
    ...{
    ...    //Act
    ...    var Initial = DateTime.UtcNow;
    ...    var request = await client.PostAsync($"/api/v0.1/tenants", new StringContent(JsonConvert.SerializeObject(null), Encoding.UTF8, "application/json"));
    ...    var response = request.Content.ReadAsStringAsync().Result;
    ...
    ...    var dif = DateTime.UtcNow - Initial;
    ...    //Assert
    ...    request.StatusCode.Should().Be(HttpStatusCode.BadRequest);
    ...    dif.TotalMilliseconds.Should().BeLessOrEqualTo(1000);
    ...}
}
```

Figura 4.17 – Teste de acesso não autorizado

Os testes estão todos organizados de forma a que as *Assertions* sejam feitas ao tipo de resposta esperado, portanto todos os testes foram validados. O próximo desafio é alinhar os testes de integração/performance com todos os testes unitários já existentes. Para tal, na *pipeline* serão necessários todos os serviços com os quais o **Tenant** comunica.

4.2.3 Pipeline

Os serviços da plataforma que estão referenciados no projeto do serviço *t-yugoup-tenant* são necessários para que o projeto possa ser devidamente integrado na *pipeline*. Quer isto dizer que, para que seja possível compilar o código do projeto, seria também necessário integrar na *pipeline* o código de todos os projetos com os quais o *Tenant* tem dependências ou referências. Será portanto necessário gerir vários repositórios e incluir todas as soluções das dependências do projeto numa pasta, que posteriormente será reorganizada, e só depois poderão ser feitas as

operações sobre o código. De seguida será então possível integrar análise estática, restaurar as dependências do projeto do serviço *Tenant*, fazer *build* ao projeto, executar testes unitários, testes de integração/performance e, por fim, interpretar os resultados dos testes. Foi também necessário instalar um *plugin* para que se possam analisar vários repositórios em simultâneo.

Para o *setup* do ambiente do **Jenkins** foi utilizado um *script* semelhante ao da figura 4.18. Este *script* vai criar uma pasta (final) e uma subpasta (src), vai fazer *clone* aos repositórios que fazem parte da plataforma e, por fim, *build* ao projeto. Isto acontece porque, como dito anteriormente, são necessários todos projetos da plataforma *Yugoup*, que estão referenciados no projeto *t-yugoup-tenant*, para que possa ser feita *build* ao serviço da **Web API** do *Tenant*.

```
1 #start-ssh-agent.cmd
2 mkdir final\src
3
4 git clone repository_name
5 robocopy repository_name\src final\src /COPYALL /e /NFL /NDL /NJH /nc /ns /np
6
7 Echo Build
8
9 "C:\Program Files\dotnet\dotnet.exe" build "final\src\repository_name\repository_name.csproj"
10
```

Figura 4.18 – Script de setup

No **Jenkins**, dá-se uma descrição à *pipeline*. Em **Source Code Management**, selecciona-se o *plugin Multiple SCMs* e configura-se a ligação aos repositórios via **SSH**. Todos os repositórios que vão ser adicionados à *pipeline* vão seguir a mesma metodologia. Primeiro será inserido o **URL** do repositório e é seleccionada a credencial de acesso por **SSH** e só depois é especificada a *branch* sob a qual serão feitas as operações. Por fim, dá-se um nome à pasta onde serão colocados os ficheiros dos projetos (4.19). As configurações restantes serão idênticas, não existirão **Build Triggers**, apenas será eliminado o *workspace* antes de cada *build*.

Em **Build**, todos os *build steps* utilizados para criar o *setup* serão feitos à semelhança do *script* que se pode visualizar na figura 4.18. À semelhança do *script*, como dito anteriormente, foi criada um diretoria – final – e uma subdiretoria – src – para onde foram passados os ficheiros dos serviços. Foi utilizado o **robocopy** para

copiar os ficheiros para a subdiretoria: `$ robocopy project-name folder-path` juntamente com as opções `/COPYALL /e /NFL /NDL /NJH /nc /ns /np` (Microsoft, 2019c).

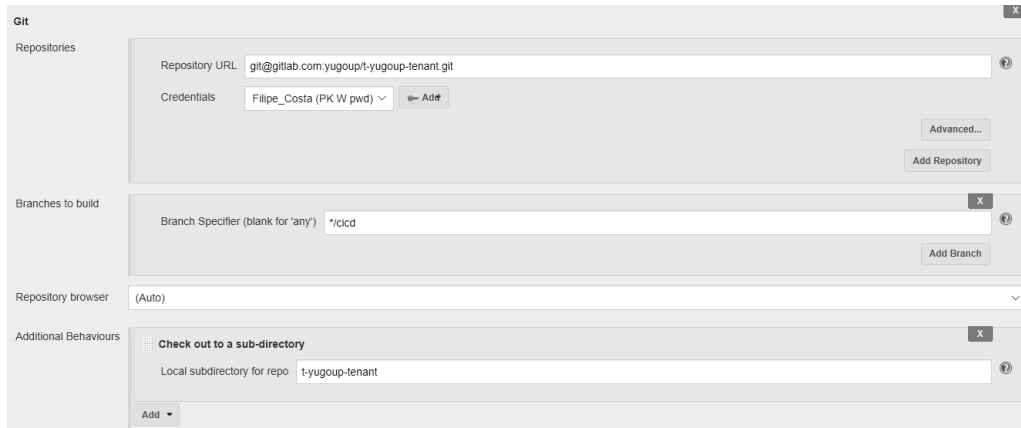


Figura 4.19 – Multiple Source Code Management

Fase de Build do projeto

As primeiras instruções na fase de *build* do projeto *tenant* são a instrução de arranque da análise estática e a instalação dos **NuGet Packages** presentes num repositório online. Posteriormente é feita *build* à solução do *Tenant* (4.20).



Figura 4.20 – Análise estática, Restore e Build

Para iniciar a análise estática foram repetidos os passos do capítulo anterior.

Depois de iniciada a análise estática e antes e antes de ser feita *build* à solução é executado um *script* que vai instalar todas as dependências através da utilização de um ficheiro **batch** (.bat) desenvolvido pela equipa da **Yugoup**. Na fase de *build* da solução – e após o *setup* do ambiente – todas as soluções serão revistas pela análise estática. Isto acontece porque se está a fazer *build* à solução do *Tenant* em conjunto com as soluções das suas dependências.

Build e Run dos testes unitários

Também à semelhança daquilo que foi feito no capítulo anterior, os testes unitários terão de ser compilados (4.21) e executados (4.22) para que sejam gerados resultados. Só depois se podem armazenar e apresentar estes resultados. O mesmo acontece com os testes de integração e performance.

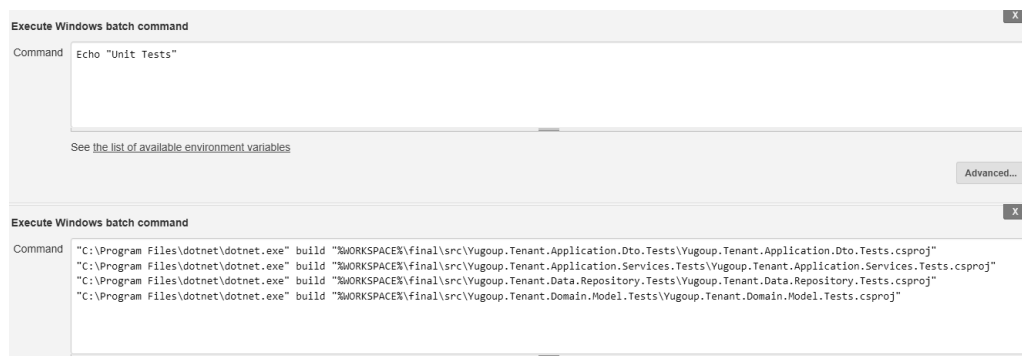


Figura 4.21 – Compilação dos projetos de teste

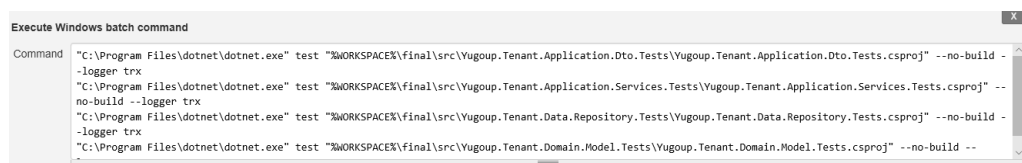


Figura 4.22 – Execução dos projetos de teste

As instruções de execução dos testes já são conhecidas e, como pode ser visto pela figura (4.23), antes da execução dos testes de integração é necessário executar outro *script*, através de um ficheiro **batch** (.bat), para remover o *feed* das dependências do projeto. Resta parar a execução da análise estática e pode então proceder-se à execução dos testes de integração e performance.

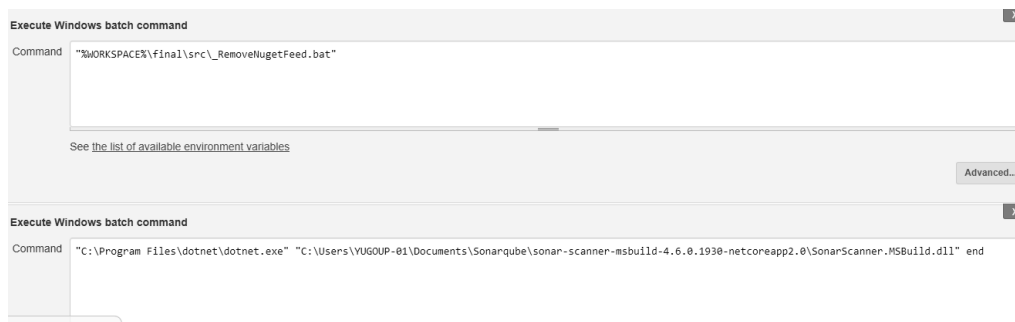


Figura 4.23 – Remoção das dependências

Build e Run dos testes integração e performance

Os testes de integração e performance são compilados e executados (4.24) à semelhança dos testes unitários. Na fase de **Post-build Actions** vão interpretar-se os resultados de todos os testes em simultâneo.



Figura 4.24 – Compilação e execução dos testes de integração e performance

Um complemento interessante à *pipeline* seria a geração automática de uma imagem de **Docker** do serviço **t-yugoup-tenant**. A imagem foi gerada e exposta numa porta pública mas o seu processo de geração não foi incluído na *pipeline*. Apesar da aplicação ter sido publicada, o seu endereço não estava autorizado a aceder aos serviços da plataforma Yugoup. Em baixo é apresentada a estrutura do *Dockerfile* que deu origem à imagem do serviço e são delineados um conjunto de **Next Steps** para o futuro, caso a equipa pretenda integrar na *pipeline* o processo de geração automática da imagem através de um *Dockerfile*.

4.2.4 Geração da imagem do serviço *Tenant*

A versão de *Release* da **Web API** do serviço *Tenant* sobre a qual estão a ser feitos os testes de integração foi publicada na máquina de *staging* através de **Web Deploy** (Microsoft, 2019f). Contudo, seria interessante publicar uma imagem de **Docker** do serviço *Tenant* semelhante à imagem da **Web API** da Calculadora.

Foram incluídos todos os projetos (4.25) que compõem o serviço *Tenant* e foi feito *restore* a partir do repositório de artefactos que foi disponibilizado pela equipa do projeto (4.26). De seguida foi copiada a pasta de configuração do ficheiro **NuGet.config** para a pasta **/app**, foram executadas as intruções de `$ dotnet build` e `$ dotnet publish` (4.27) com a configuração de *Staging* e o *output* a ser colocado na pasta **/app**. Por fim foi publicada a aplicação com a *dynamically linked library* da **Web API** do serviço de *Tenant* tal como pode ser visto na figura 4.27.

```
Dockerfile x
source › repos › t-yugoup-tenant › src › Dockerfile
1 FROM microsoft/dotnet:2.2-aspnetcore-runtime AS base
2 WORKDIR /app
3 EXPOSE 80
4 expose 61030
5
6 ENV ASPNETCORE_ENVIRONMENT="Staging"
7
8 FROM microsoft/dotnet:2.2-sdk AS build
9 WORKDIR /
10
11 # Copy all .csproj files to the app folder
12 COPY Yugoup.Tenant.sln ./
13 COPY Yugoup.Tenant.Application.Dto/*.csproj ./Yugoup.Tenant.Application.Dto/
14 COPY Yugoup.Tenant.Application.Dto.Tests/*.csproj ./Yugoup.Tenant.Application.Dto.Tests/
15 COPY Yugoup.Tenant.Application.Services/*.csproj ./Yugoup.Tenant.Application.Services/
16 COPY Yugoup.Tenant.Application.Services.Tests/*.csproj ./Yugoup.Tenant.Application.Services.Tests/
17 COPY Yugoup.Tenant.Client/*.csproj ./Yugoup.Tenant.Client/
18 COPY Yugoup.Tenant.Data.Gateway/*.csproj ./Yugoup.Tenant.Data.Gateway/
19 COPY Yugoup.Tenant.Data.Repository/*.csproj ./Yugoup.Tenant.Data.Repository/
20 COPY Yugoup.Tenant.Data.Repository.Tests/*.csproj ./Yugoup.Tenant.Data.Repository.Tests/
21 COPY Yugoup.Tenant.Domain.Core/*.csproj ./Yugoup.Tenant.Domain.Core/
22 COPY Yugoup.Tenant.Domain.Model/*.csproj ./Yugoup.Tenant.Domain.Model/
23 COPY Yugoup.Tenant.Domain.Model.Tests/*.csproj ./Yugoup.Tenant.Domain.Model.Tests/
24 COPY Yugoup.Tenant.Domain.Services/*.csproj ./Yugoup.Tenant.Domain.Services/
25 COPY Yugoup.Tenant.Infrastructure.CrossCutting/*.csproj ./Yugoup.Tenant.Infrastructure.CrossCutting/
26 COPY Yugoup.Tenant.WebApi/*.csproj ./Yugoup.Tenant.WebApi/
27 COPY Yugoup.Tenant.WebApi.IntegrationTest.Tests/*.csproj ./Yugoup.Tenant.WebApi.IntegrationTest.Tests/
28 COPY Yugoup.Tenant.Application.Dto.Tests/*.csproj ./Yugoup.Tenant.Application.Dto.Tests/
29
```

Figura 4.25 – Dockerfile do serviço *Tenant*

Sem alterar o código da plataforma, uma vez que não eram permitidas alterações à sua base, e sem interferir com os tipos de bases de dados que a plataforma

```

78
79 RUN echo '<?xml version="1.0" encoding="utf-8"?> \\
80 <configuration><packageSources> \\
81 <add key="myget.org" value="https://www.myget.org/F/yugoup/api/v3/index.json" />\\
82 </packageSources><packageSourceCredentials><myget.org><add key="Username" value="yugoup.packages" />\\
83 <add key="ClearTextPassword" value="Q!w2345" />\\
84 </myget.org></packageSourceCredentials>\\
85 <apikeys><add key="myget.org" value="77771ba6-d6f6-4fa4-bcf0-3e316d1e74ce" />\\
86 </configuration>' > NuGet.config
87
88
89 COPY NuGet.config /root/.nuget/NuGet/NuGet.config
90
91 COPY . .

```

Figura 4.26 – Dockerfile do serviço *Tenant*

```

83 COPY . .
84
85 WORKDIR /Yugoup.Tenant.WebApi/
86 #Copy /Yugoup.Tenant.WebApi/appsettings.Development.json /Yugoup.Tenant.WebApi/appsettings.Production.json
87 RUN dotnet build -c Staging -o /app
88 #--source "https://www.myget.org/F/yugoup/api/v3/index.json" --k 77771ba6-d6f6-4fa4-bcf0-3e316d1e74ce
89
90 FROM build AS publish
91 RUN dotnet publish -c Staging -o /app
92 #--source "https://www.myget.org/F/yugoup/api/v3/index.json" --k 77771ba6-d6f6-4fa4-bcf0-3e316d1e74ce
93
94 FROM base AS final
95 WORKDIR /app
96 COPY --from=publish /app .
97 ENTRYPOINT ["dotnet", "Yugoup.Tenant.WebApi.dll"]

```

Figura 4.27 – Dockerfile do serviço *Tenant*

está a utilizar, foi criada uma base de dados do mesmo tipo que o serviço utiliza – **SQL Server** – através do uso de uma imagem de **Docker** disponibilizada no repositório oficial que a **Microsoft** criou para imagens do **Microsoft SQL Server** (Microsoft, 2019a). Também é necessário fazer uma pequena alteração à *connection string* do serviço *Tenant* uma vez que terá de ser ligeiramente diferente daquela que o serviço que está publicado na máquina de *staging* utiliza atualmente para que a base de dados a ser executada dentro de um *container* possa ser acedida e manipulada.

Tal como é apresentado na figura 4.28, pode ser vista a *connection string* de ligação ao *container* do **Docker** onde está a ser executada a imagem do **SQL Server**. Comentada, na linha anterior, também pode ser vista a *string* de ligação à base de dados antiga. Para verificar que a base de dados de **SQL Server** está

a funcionar é necessário aceder ao *container* via **Azure Data Studio**. São assim necessárias as credenciais de acesso com as quais foi lançado o serviço de base de dados e é necessário colocar no campo **Server**: *localhost, port-number* (4.29). Quando a conexão à base de dados for estabelecida será possível ver uma luz verde (semelhante à da figura 4.32) ao lado do nome da base de dados e, depois de se iniciar o serviço do tenant, poderá ser então vista a base de dados criada pelo **EF Core – TenantRepository** – dentro da diretoria das bases de dados.

```
1 {
2   "Service": {
3     "Gateway": "http://192.168.12.150:60000",
4     "Uri": "http://s.Tenant.yugoup.com"
5   },
6   "ConnectionStrings": {
7     "DefaultConnection": "192.168.12.150\\STAGING;Database=TenantRepository;Uid=developer;Password=q1w2e3r4t5y6"
8     "DefaultConnection": "Server=192.168.12.32,1433;Database=TenantRepository;User-Id=sa;Password=DockerSql2017!"
9   },
10 }
```

Figura 4.28 – Connection String da Base de Dados

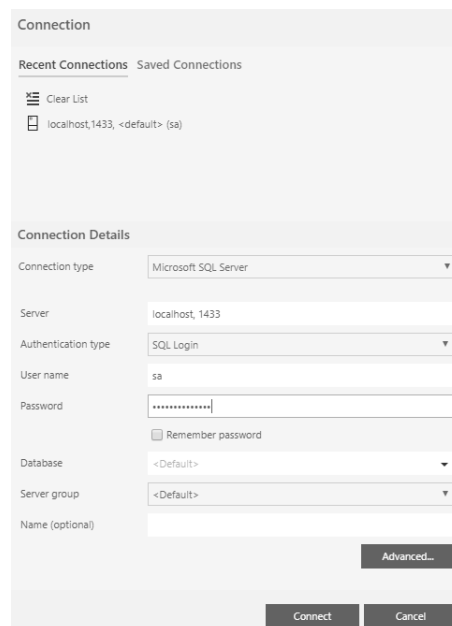


Figura 4.29 – Conexão ao serviço de base de dados pelo **Azure Data Studio**

A imagem do serviço foi depois publicada com a *tag* **t-yugoup-tenant** e posteriormente foi lançado um *container* com o projeto da **Web API** que mais tarde foi acedido pelo *browser*.

4.2.5 Resultados

A apresentação dos resultados dos testes e da análise estática vai ser o produto final da *pipeline* de testes automatizada.

Apresentação dos resultados dos testes

A apresentação dos resultados dos testes no **Jenkins** é feita com um gráfico. Este gráfico contém o número de testes – *Passed*, *Failed* e *Skipped* – no eixo dos *y* e o número da *build* no eixo dos *x*. Na figura 4.30 são apresentados os resultados dos testes, na sua maioria, por um tom de cinza intermédio.



Figura 4.30 – Resultado dos testes

Quer isto dizer que grande parte dos testes são executados com sucesso. Na *build* número 187 foi introduzida a análise aos quatro testes de integração, dois dos quais com testes de performance, cujo resultado inicial da execução não foi de encontro ao esperado, sendo possível ver pela cor cinza escuro. No entanto duas *builds* depois

(189) os testes foram compilados e executados com sucesso, o que resultou numa subida no número total de testes *Passed*. Para ser vista a representação dos testes *Skipped* foi criado, a título de exemplo, um teste unitário em **MSTest** com o atributo **Skip** – representado a cinza claro – para ser conhecida a sua interpretação gráfica.

No total, o serviço do **Tenant** contabiliza dezoito testes. Destes dezoito testes, um é passado à frente (*Skipped*). Quatro desses dezoito testes são testes feitos à integração dos *endpoints* da plataforma e dois desses quatro testes são testados em termos de performance. Os restantes treze testes são testes unitários, feitos às camadas de manipulação e tratamento de dados da plataforma.

Resultados da análise estática

A análise estática apresentou um conjunto alertas relativos a falhas de segurança, *bugs*, *Code Smells* e vulnerabilidades dentro da plataforma, no entanto o **quality gate** foi positivo, como é possível verificar-se na figura 4.31.

Foram identificados 27 *Bugs*, 37 *Vulnerabilities*, 634 *code smells* e 193 *Security Hotspots*. Em termos de gravidade, foram identificados 5 *Issues* do tipo *Blocker*, 198 do tipo *Minor*, 105 do tipo *Critical*, 385 do tipo *Major* e 5 do tipo *Info*. Toda a informação acerca dos *Issues* identificados pela análise estática do **Sonarqube** é descrita em detalhe na documentação oficial (SonarQube, 2019).

4.2.6 Considerações

Após a conclusão da execução da *pipeline* verificamos que o serviço do *Tenant* está validado com uma bateria de testes e com a análise estática. Numa primeira fase, depois da junção de todos os projetos numa pasta final, é executada análise estática. Inicialmente a análise estática iria ser executada contra o serviço do *Tenant*. No entanto, apesar da *build* ser feita apenas ao serviço que pretendíamos analisar inicialmente, todos os projetos que estão dependentes do serviço vão ser analisados. Como muitos projetos dependem, direta ou indiretamente, do serviço

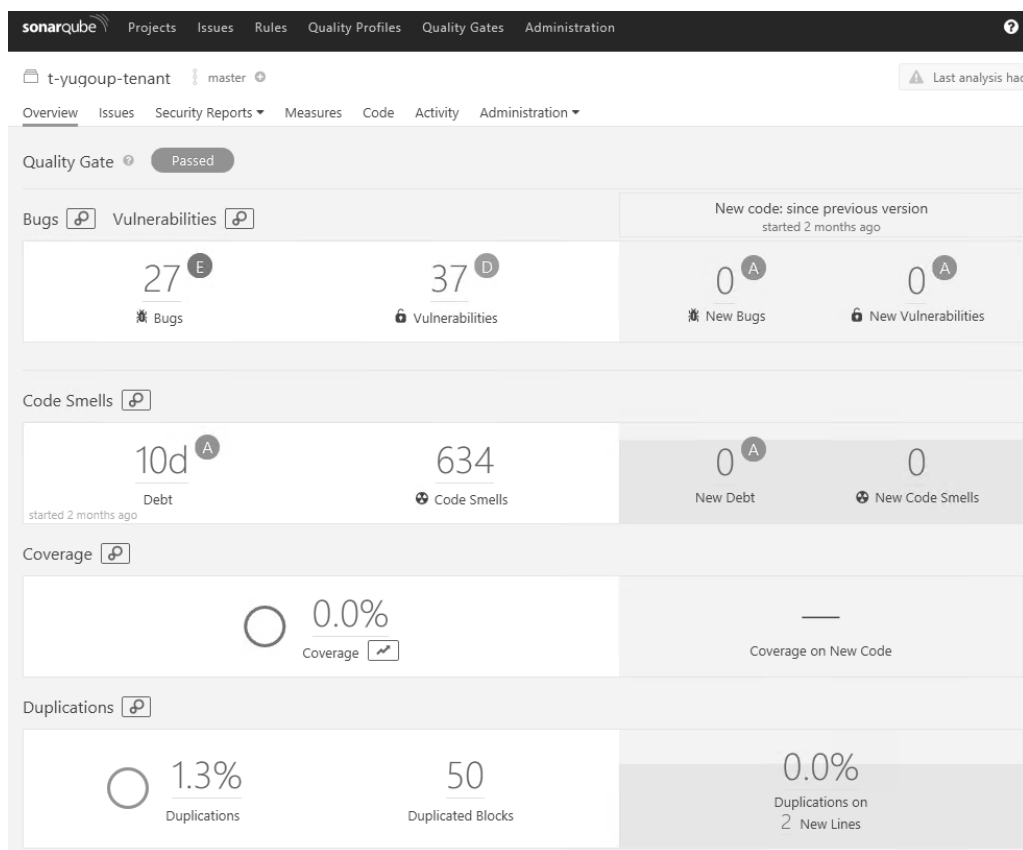


Figura 4.31 – Overview da análise estática

do *Tenant* vamos ter a plataforma da **Yugoup** analisada por completo. Apesar de ir contra o objetivo inicial, a análise estática faz mais do que aquilo que inicialmente foi estabelecido. A análise estática é fundamentalmente uma análise léxica e sintática do código desenvolvido pela equipa da Yugoup. É possível afirmar também que a análise estática pode incorporar análise semântica tendo em conta o contexto do negócio. Do ponto de vista funcional, o **Sonarqube** é incluído na *pipeline* porque é fundamental que todo o código seja revisto e analisado de acordo com as boas práticas de programação e para que seja verificado que não existem variáveis, métodos ou chamadas de funções não implementados ou não utilizados dentro da plataforma.

A execução dos testes unitários é importante do ponto de vista semântico e de compilação. Com os testes unitários, são feitas verificações ao modelo da aplicação que é testado de acordo com um conjunto de especificações, tendo em conta as

funcionalidades implementadas, as quais foram pedidas com regras pelo *business*. Os testes unitários também procuram testar se essas mesmas regras são aplicadas e se os resultados obtidos vão de encontro aos resultados esperados.

Os testes de integração e performance são utilizados para testar a integração entre serviços através da comunicação com *endpoints* na máquina de *staging*. Junto com testes de integração, são aproveitados os tempos de resposta, tempos esses que são – após execução dos testes – dados como testes de performance.

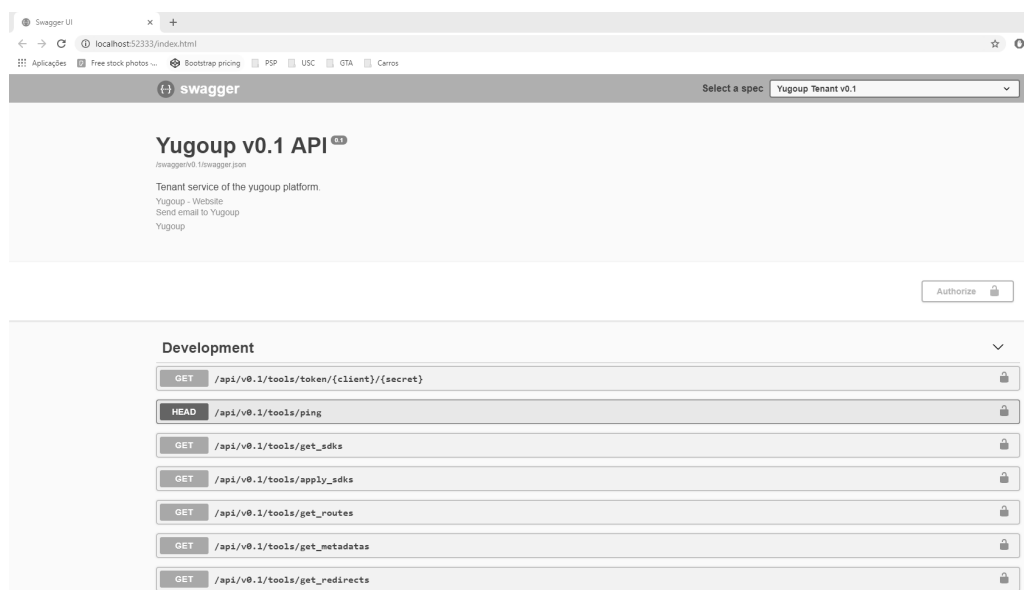


Figura 4.32 – Screenshot do serviço *Tenant* executado a partir do IDE

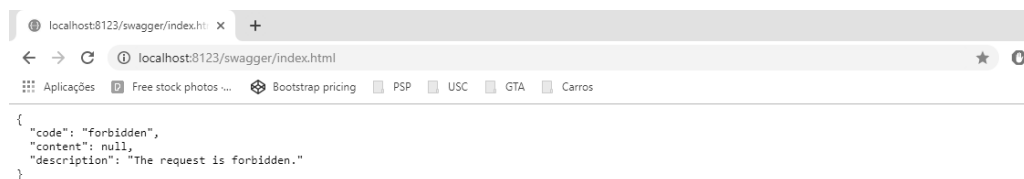


Figura 4.33 – Screenshot do serviço *Tenant* executado a partir da imagem **Docker**

O *Tenant* funciona como esperado quando executado a partir do ambiente integrado de desenvolvimento (4.32). Mesmo com a base de dados de **SQL Server** servida a partir do *container*. Por outro lado, o comportamento do serviço *Tenant*, quando executado a partir da imagem gerada anteriormente através do *Dockerfile* –

que pode ser visualizado nas figuras 4.25, 4.26 e 4.27 – é diferente (4.33) do esperado. O problema apresentado é uma medida de segurança, a qual pode ser desativada, que foi implementada para proibir interações que não estejam configuradas ou esperadas. Esta medida de segurança usa-se para colocar um serviço por detrás de um *Gateway*, o qual é o único a ter acesso a essa API. A configuração garante que apenas e só – e só mesmo – o serviço *Gateway* consegue comunicar com esses serviços. Este mecanismo apresenta também a vantagem de prevenir **DDoS** pois, em termos de *stack* de chamada, faz com que o pedido falhe imediatamente sem ocupar grande quantidade de recursos.

4.3 Integração de sistemas de visualização

Como forma de obter feedback mais rápido acerca do resultado da *pipeline* surgiu a necessidade de implementação de um sistema de visualização integrado no ciclo de desenvolvimento e teste de software. O principal objetivo da integração deste sistema de visualização seria a colocação do estado de uma *build* num monitor LCD disponibilizado pela empresa. O sistema servirá para notificar os *developers* acerca do estado das *builds* e estará ligado, em rede local, à máquina de *staging* que, por sua vez, conta com uma instância do *Jenkins* pronta a executar um conjunto de *Jobs*.

4.3.1 Sistema de visualização

As especificações do sistema, em termos de requisitos funcionais, seriam as seguintes:

- O sistema deve apresentar o resultado final de uma *build*;
- O sistema deve notificar os *developers*;
- O sistema deve utilizar o LCD disponibilizado pela empresa;
- O sistema deve utilizar um Raspberry PI;

- O sistema deve estar conectado à rede local;

Para além da monitorização – quase instantânea – dos *jobs*, o sistema de visualização estará disponível via FTP (*File Transfer Protocol*). Isto permitirá a inclusão de um conjunto de ficheiros, como os ficheiros com as credenciais de acesso à máquina de *staging*, para facilitar a conexão ao sistema central.

4.3.2 Integração do sistema de visualização

Para a integração de um sistema idêntico de visualização em rede – local ou online – é necessário que o sistema central conte com:

- Uma instância do *Jenkins*;
- O *Plugin Build Monitor*;
- Um *Job*;

Para a integração do sistema de visualização vão ser necessários:

- Um *Raspberry PI*, com o respetivo carregador;
- Um *SD Card* com, pelo menos, 8GB de memória;
- Um teclado;
- Um rato ótico (opcional);
- Um monitor, como referido anteriormente;
- Um cabo de rede;
- Um cabo HDMI;

4.3.3 Instalação do sistema de visualização

O primeiro passo para que seja possível integrar o sistema de visualização no ciclo de desenvolvimento de software é a formatação do cartão de memória que irá conter o sistema operativo do mini-computador, o **Raspberry PI**. Para a formatação do **SD Card** existem vários softwares disponíveis mas, neste caso, foi utilizado o software **SD Card Formatter**.

De seguida, procede-se à instalação do sistema operativo no **SD Card** pré-formatado. No site oficial da **Raspberry** são disponibilizados vários sistemas operativos, juntamente com um instalador de sistemas operativos, na secção de downloads. Neste caso, foi utilizado o **NOOBS** (*New Out-of-the Box Software*), que é um instalador de sistemas operativos *user-friendly*.

4.3.4 Configuração do sistema de visualização

Para proceder à configuração do sistema de visualização certifica-se, primeiramente, que o dispositivo contém uma fonte de alimentação com a potência adequada. Após essa verificação, conecta-se o teclado, o rato (caso se disponha de um), o cabo HDMI e o cabo de rede. Antes de se proceder à primeira execução do dispositivo, insere-se o **SD Card** na parte inferior do **Raspberry PI** com o instalador de sistemas operativos, ou com a imagem do sistema operativo que vai ser utilizado. Só depois de se completar os passos anteriores é ligado o dispositivo à corrente. Se todos os passos tiverem sido devidamente executados, o sistema de configuração do dispositivo deve arrancar dentro de alguns momentos.

A primeira vez que o **Raspberry PI** arranca é mais demorada e requer a instalação dos módulos e componentes do sistema. Dito isto, quanto mais rápida for a conexão à rede, mais rápido será o download do sistema operativo e dos seus módulos e, consequentemente, mais rápida será a configuração do dispositivo.

Após a instalação dos módulos e do sistema operativo, o dispositivo terá de ser reiniciado para que sejam aplicadas as alterações. Neste caso, em particular, foi



Figura 4.34 – Ambiente de trabalho do *rasbpian*

instalado o **raspbian**, o sistema operativo desenvolvido pela marca **Raspberry PI**, que é baseado no **Debian**, um sistema operativo *open source*. Este sistema operativo apresenta uma *Graphical User Interface* semelhante ao ambiente de trabalho, *user-friendly*, dos seus homólogos mais conhecidos – **Windows** e **Ubuntu** – que facilita a sua aprendizagem (4.34).

4.3.5 Apresentação do estado das builds

O estado das builds, que será mostrado no LCD, fará uso do *plugin* **Build Monitor** que foi previamente instalado na instância do **Jenkins** da máquina de *staging* que a empresa disponibilizou. Tendo em conta que um dos pressupostos para a implementação deste sistema passava pelo seu funcionamento em rede local, é necessário aceder ao endereço de IP da máquina de *staging*.

Para ser possível criar a *view* do **Build Monitor**, na página principal do orquestrador de processos, clica-se na *tab* que contém um sinal “+”, por cima dos *jobs*, selecciona-se o *radiobutton* com a opção “*Build Monitor View*” e atribui-se um nome (4.35).

Já dentro do painel de configuração da *view* podemos ser definidos um conjunto de critérios. Desde uma descrição, um filtro por estado do *job*, o nome do último *job*, o seu tempo de execução, etc. Neste caso, será apenas testado o seu funcionamento, portanto, selecciona-se o *job* “teste” que foi anteriormente criado com esse propósito (4.36).

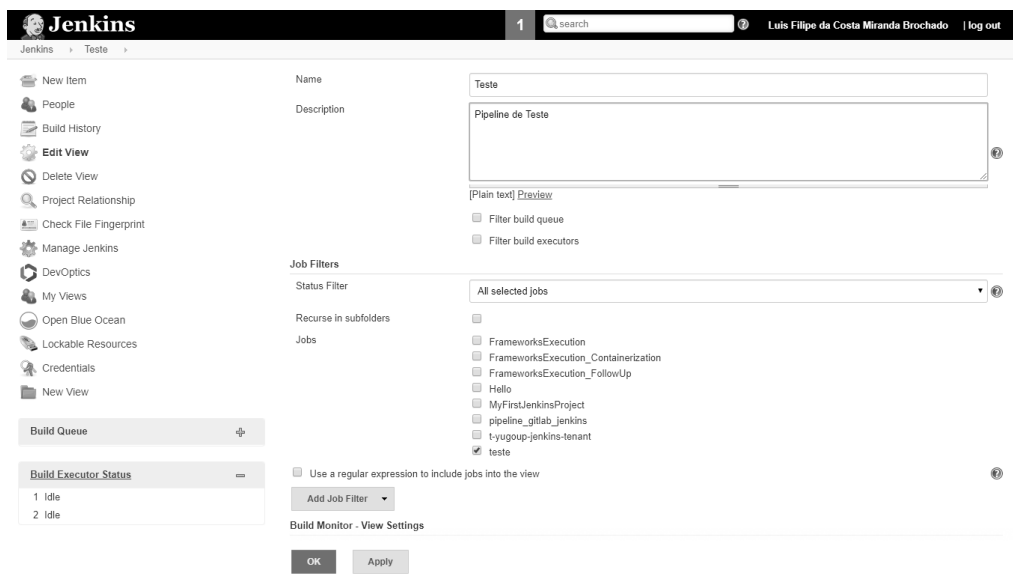


Figura 4.35 – Configuração da view

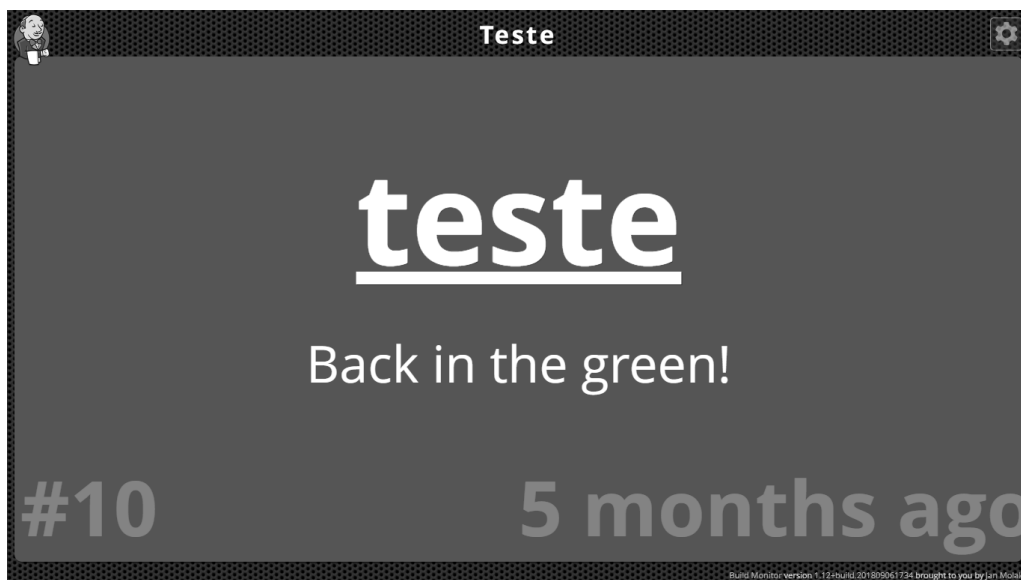


Figura 4.36 – Estado do job

Tendo em conta que o objetivo da instalação de um sistema deste tipo é a sua mostragem constante no ecrã, para efeitos de monitorização, será necessária uma configuração adicional ao **Raspberry PI** para que o dispositivo não entre em hibernação. Para tal, após alguma pesquisa, verificou-se que existiam um conjunto de soluções nos fóruns oficiais da **RasperryPI** nos *posts* (*Raspberrypi.org, 2014b*) e (*Raspberrypi.org, 2014a*). A solução passou por alterar algumas linhas nos ficheiros de configuração do sistema operativo.

4.3.6 Pontos críticos

A utilização de um pequeno dispositivo com baixa capacidade de computação foi suficiente para criar algum *engagement* entre a equipa de desenvolvimento. Contudo, caso se pretenda um dispositivo com melhor performance, devem ter-se em atenção alguns aspetos. Em termos comparativos, o dispositivo utilizado tem um conjunto de sucessores com performance superior. A versão do **Raspberry PI** (*Raspberrypi.org, 2016*), segundo o website da marca, conta com as seguintes especificações base:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU 1GB RAM;
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board;
- 100 Base Ethernet;
- 40-pin extended GPIO;
- 4 USB 2 ports;
- 4 Pole stereo output and composite video port;
- Full size HDMI;
- CSI camera port for connecting a Raspberry Pi camera;
- DSI display port for connecting a Raspberry Pi touchscreen display;

- Micro SD port for loading your operating system and storing data;
- Upgraded switched Micro USB power source up to 2.5A;

Apesar de existirem dispositivos com performance algo superior, um dispositivo semelhante ao utilizado é mais do que suficiente para o tipo de trabalho que foi designado.

Um dos fatores que contribui imenso para a performance do dispositivo, para além da velocidade de conexão à internet, é o tipo de **SD Card** utilizado. Para melhor performance, recomenda-se a utilização de um cartão classe 10 (C-10) de 16 ou 32GB. A capacidade do cartão não afeta diretamente o tempo de resposta. Contudo, o tempo de leitura e o tempo de escrita podem diferir entre cartões da mesma classe. Na tabela apresenta-se a performance do mesmo **Raspberry PI** com cartões diferentes. Os tempos apresentados serviram para medir o tempo desde que o dispositivo inicia até que está 100% operacional.

Tempos de espera até iniciar o sistema operativo		
Tentativa	SD Card 16GB	SD Card 32GB
t_1	1:55.84 (min)	1:31:56 (min)
t_2	1:55.30 (min)	1:26:67 (min)
t_3	1:55.64 (min)	1:34:15 (min)
t_4	1:52.33 (min)	1:30:10 (min)
t_5	1:56.32 (min)	1:28:75 (min)
t_6	1:58.67 (min)	1:35:18 (min)
t_7	2:00.67 (min)	1:35:43 (min)
t_8	1:55.90 (min)	1:35:57 (min)
t_9	1:57.55 (min)	1:37:41 (min)
Média (min)	1:55.46 (min)	1:32.76 (min)

O primeiro cartão testado com 16GB de capacidade tinha, segundo as especificações, velocidade de leitura na ordem dos 18.6 MB/s e velocidade de escrita na

ordem dos 10.2 MB/s. Já o segundo cartão testado tinha 32GB de capacidade e, segundo as especificações, contava com velocidade de leitura e de escrita bastante superiores, 95 MB/s e 25 MB/s, respetivamente.

Pode concluir-se que, por si só, a utilização de um *SD Card* com melhor performance apresenta melhores resultados para este caso, onde o objetivo foi medir o tempo de espera até o sistema operativo do dispositivo estar "*fully operational*". Como é evidente, este intervalo de tempo não contabiliza a abertura do *Browser* e a creditação no *Jenkins* para a apresentação do estado das *builds*. Contudo, é importante referir que o *SD Card* cuja velocidade (estimada) de leitura e escrita se diz de melhor performance foi mais rápido, em média, 23.30 segundos do que o *SD Card* disponibilizado pela empresa. Este *upgrade* representa uma melhoria de 20.3% no que diz respeito à demora ao iniciar o sistema operativo para o mesmo dispositivo.

Como referido anteriormente, este dispositivo – cedido pela empresa – é mais do que suficiente para a função que está a desempenhar. No entanto, caso se pretenda um dispositivo com melhor *performance*, existem no mercado várias soluções que podem ser estudadas. Existe por exemplo a versão mais recente do dispositivo, o **Raspberry PI model 3 b+** (Raspberrypi.org, 2018) cuja ficha técnica apresenta, segundo a marca, especificações base superiores.



Conclusão e trabalho futuro

5.1 Conclusão

O impacto positivo esperado através da utilização da *pipeline* é notório principalmente na redução do tempo total de validação da qualidade do código para acelerar o processo de colocação em produção. A automatização dos testes e a *framework* apresentada (**xUnit**) como alternativa são mais céleres na execução dos testes quando comparadas com a *framework* atualmente em utilização na empresa. Com a implementação dos testes unitários em **xUnit** é possível obter-se uma redução para quase metade do tempo na sua execução e, ao ser acrescentada na *pipeline* de testes, liberta-se o *developer* para a realização de outras tarefas enquanto a *suíte* de testes está a decorrer. Esta melhoria aplica-se não só na execução dos testes unitários, mas também na execução dos testes de integração e performance.

A análise estática é um processo um pouco mais demorado, sendo que o serviço em si, necessita de alguma configuração inicial, que tanto pode ser feita *on demand*, como pré-configurada. A primeira abordagem tem a vantagem de não necessitar de nenhum recurso físico. Tudo o que é necessário é gerado na *pipeline* com recurso a *containers docker*, no entanto, é necessário lançar a imagem em cada execução da

pipeline, gerar a chave com o projeto – processo que pode demorar até 3 minutos em *setup* – e dependendo do tamanho do projeto, vários minutos a analisar, sendo que neste caso demorou em média 5 minutos. A segunda abordagem requer que o serviço esteja configurado numa máquina (física ou virtual), no entanto permite reduzir o tempo de *setup* e de análise devido a estratégias de *cache* realizadas pelo **SonarQube**. De qualquer das formas, apesar de esta análise ser um pouco demorada, em termos práticos esta execução não vai afetar a utilidade da *pipeline*, pelo contrário, uma vez que ela pode ser configurada para ser executada em paralelo ao processo de testes e assim apresentar feedback informativo sobre o projeto em si.

A utilização de *containers* para a validação de aplicações também representa uma melhoria significativa em termos de tempo. O processo de *deploy* manual de uma aplicação numa máquina de *staging* para validação do PO (*product owner*) ou do cliente é mais demorado e implica a existência de uma máquina para onde será feito o *deploy*. Esta abordagem pode trazer problemas em termos de dependências uma vez que a atualização dos serviços é manual, podendo ficar alguma dependência por atualizar causando problemas no serviço. O *deploy* de uma aplicação para vários *containers* pode ser visto como uma forma mais fácil e eficaz de manter a aplicação e os serviços que a compõem, e resolve a problemática de dependências e versões antigas uma vez que toda a infraestrutura é construída apenas para aquela versão de código.

Tendo em conta todos os fatores que podem ser considerados para a redução do tempo de entrega, a empresa pode reduzir o tempo de execução dos testes unitários, de integração e de performance para aproximadamente metade (50%). Caso a execução dos testes consuma 10 minutos por execução, e assumindo a execução completa de testes 2 vezes por dia – o que é bastante inferior à realidade mas muitas das execuções dos testes são apenas parciais – estaríamos a falar de uma redução diária de 10 minutos, o que se transforma em reduções mensais de quase 4 horas (isto para apenas um *developer*). Tendo em conta que o projeto tem entre 2-3 *developers*, as poupanças de tempo sobem para dias de tempo poupado. Mas na realidade, o uso de uma *pipeline* leva a uma poupança de tempo muito superior, como a execução

dos testes e as análises realizadas são feitas de modo assíncrono ao trabalho do *developer*, ele não fica bloqueado à espera que o processo termine, ficando assim livre para continuar as suas tarefas, sendo notificado e apenas tomando acções quando os resultados dos testes falham.

Os *deploys* de artefactos, de imagens e da aplicação também são um facto que contribui diretamente para a redução do tempo de entrega. O *deploy* de um artefacto para um repositório de artefactos é relativamente simples, no entanto, pode ser necessário consultar documentação para fazer este processo manualmente. O mesmo acontece com o *deploy* de uma imagem para um registo privado de imagens e com a *containerização* de uma aplicação para aprovação. Quando automatizado, este processo reduz imenso tempo à validação do software para produção que, por sua vez, reduz o *time-to-market* que normalmente estaria dependente de um *deploy* manual sempre sujeito a erros. Apesar de ser difícil prever com exatidão, no cenário mais pessimista a *pipeline* de testes automatizados poderia reduzir anualmente em entre 18 horas e meia e 2 dias e 5 horas ao tempo de desenvolvimento de cada serviço – somando ainda as reduções inerentes à automatização dos *deploys* – e ao tempo de entrega de software, não esquecendo a redução nas falhas na execução manual de instruções onde é propícia a ocorrência de falha humana. Este valor foi obtido através da multiplicação daquilo que são considerados os principais fatores de redução do tempo de entrega pelo número total de dias úteis anuais em comparação com o processo atual de teste e validação de código para produção.

A dinâmica criada pelo sistema de visualização também tem um impacto global no tempo de execução da *pipeline*. Com feedback em tempo real, a equipa de desenvolvimento vai monitorizar constantemente o desenvolvimento das aplicações. Pode ainda concluir-se que a utilização do sistema de visualização iria contribuir para a redução do *time-to-market*, através do aumento do *awareness* global, apesar de ser difícil estimar um número em quantidades de tempo. Através da ajuda das ferramentas de análise da qualidade do código os *developers* terão uma ideia mais concreta sobre o desenvolvimento e a manutenção das aplicações mais facilitada, podendo identificar e corrigir as falhas com maior rapidez.

5.2 Trabalho futuro

Para este trabalho consideraram-se apenas os componentes indispensáveis para que o *workflow* seja semelhante ao de uma *pipeline* de integração e entrega contínua. Dos componentes mencionados no documento, aqueles que podem ser considerados essenciais para o funcionamento de uma *pipeline* de testes automatizados são:

- O orquestrador de processos;
- O *version control system*;
- A *tool* de análise estática;
- O repositório de artefactos;
- O repositório de imagens;
- O mecanismo de *load-balancing*;
- A *tool* de containerização;
- A *tool* de comunicação interna;

Dependendo da complexidade do processo de análise da qualidade do código e da validação do código para produção, podem ser adicionados outros componentes, quer durante a fase de validação, na *pipeline*, quer depois de validado o software, já na fase de produção. Por este motivo, existem várias linhas de melhoria deste projeto que podem ser seguidas pela empresa no sentido de aumentar tanto a celeridade no desenvolvimento como a resiliência e a eficácia na resolução de problemas que possam surgir em produção.

Executar todos os testes unitários existentes, criar testes de integração e performance na plataforma *Yugoup* seria, evidentemente, o próximo passo para dar continuidade ao trabalho desenvolvido pela equipa. Numa fase de transição para testes de integração automatizados poderiam também ser executadas baterias de

testes de integração gerados automaticamente (em nUnit) através da utilização do **Swagger Codegen** (<https://swagger.io/tools/swagger-codegen/>).

Podem ser incluídos na *pipeline smoke tests* à plataforma através da utilização de ferramentas como o **Katalon Studio** (<https://www.katalon.com/>) ou mesmo o **Selenium** (<https://www.seleniumhq.org/>). A criação destes testes não é exigente em termos desenvolvimento uma vez que são gerados automaticamente através de uma metodologia de *recording* dos casos de testes, permitindo uma posterior replicação num ciclo de extenso de tentativas. Isto seria útil para testar a resistência da plataforma a, por exemplo, submissões consecutivas de pedidos de consulta, ou até mesmo a submissões consecutivas de dados.

No que diz respeito à produção e análise de versões executáveis de uma aplicação, outro dos caminhos seria o *upgrade* do **Nexus Repository Manager** para a versão mais atualizada e a utilização do **Dive**. Tendo em vista a melhoria contínua da performance da *pipeline*, é importante mencionar que a versão do repositório de artefactos utilizada (versão 2) apenas suporta a inclusão de um mecanismo de armazenamento de artefactos – neste caso em particular de *NuGet packages*. No entanto a versão mais recente do **Nexus Repository Manager** já suporta o armazenamento de imagens Docker. Por sua vez, o **Docker Dive** pode ser utilizado como uma ferramenta de análise das camadas de uma imagem, com o objetivo de encontrar novas formas para comprimir o seu tamanho final.

Como os projetos também falham mesmo depois de todos os testes serem bem sucedidos, em produção faz sentido utilizar as ferramentas de monitorização de plataformas, tanto para monitorizar os pedidos dos utilizadores como os dados das consultas ou das bases de dados da plataforma. Aconselha-se para isso a utilização do **ELK Stack** (<https://www.elastic.co/pt/elk-stack>) que inclui o *Kibana*, o *Elasticsearch*, o *Logstash* e o *Beats*. Existe também a possibilidade de utilização do **Nagios** (<https://www.nagios.org/>) para monitorizar o estado da infraestrutura utilizada em tempo real. Estas ferramentas vão produzir gráficos que podem ser consultados e analisados pelo *business*, com dados reais e em tempo real, sobre as consultas, vendas, ou o que quer que seja útil em termos de dados sobre a utilização da plataforma,

mesmo a nível técnico. Para monitorização e *caching* de dados pode ser estudada a utilização do **Redis** (<https://redis.io/>) a fim de reduzir a carga sobre a base de dados. Outra possibilidade de migração será a mudança de base de dados do motor **Microsoft SQL Server** para uma outra que seja Open source, uma vez que vai diminuir os custos significativamente. Existe a opção de migração para bases de dados em **PostgreSQL** (<https://www.postgresql.org/>) uma vez que é compatível com o **.NET Entity Framework Core** e integra facilmente com as *Web APIs* desenvolvidas, não sendo necessária modificações significativas no projecto.

Referências bibliográficas

- Addie, S., e Boyer, S. (2019). *Get started with swashbuckle and asp.net core*. Retrieved from <https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.2&tabs=visual-studio>
- Arachchi, S., e Perera, I. (2018). Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 moratuwa engineering research conference (mercon)* (pp. 156–161).
- Atlassian. (2019). *Creating ssh keys*. Retrieved from <https://confluence.atlassian.com/bitbucketserver/creating-ssh-keys-776639788.html>
- Axelos. (2018). *Bringing agile and itil together in an agile organization — axelos*. Retrieved from <https://www.axelos.com/news/blogs/june-2018/bringing-agile-ital-together-in-agile-organization>
- Bass, L., Weber, I., e Zhu, L. (2015). *Devops: A software architect's perspective*. Addison-Wesley Professional.
- Beck. (2002). *Test driven development: By example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... others (2001). Manifesto for agile software development.
- Beck, K., e Gamma, E. (2000). *Extreme programming explained: embrace change*. addison-wesley professional.
- Choudhury, N. (2018). *.net core central - everything .net core*. Retrieved from

- <https://www.youtube.com/channel/UCjiKyvYzAYCEh4NSfsPG77A>
- Docker. (2019a). *Docker container run — docker documentation*. Retrieved from https://docs.docker.com/engine/reference/commandline/container_run/
- Docker. (2019b). *Docker image build — docker documentation*. Retrieved from <https://docs.docker.com/engine/reference/commandline/image.build/>
- Docker. (2019c). *Docker registry — docker documentation*. Retrieved from <https://docs.docker.com/registry/>
- Docker. (2019d). *Use volumes — docker documentation*. Retrieved from <https://docs.docker.com/storage/volumes/>
- Docker. (2019e). *What is a container? — docker*. Retrieved from <https://www.docker.com/resources/what-container>
- Docs.microsoft.com. (2017). *.net core command-line interface (cli) tools - .net core cli. — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/tools/?tabs=netcore2x>
- Docs.microsoft.com. (2019a). *dotnet build command - .net core cli — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-build>
- Docs.microsoft.com. (2019b). *dotnet nuget push*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-nuget-push>
- Docs.microsoft.com. (2019c). *dotnet pack command - .net core cli — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-pack>
- Docs.microsoft.com. (2019d). *dotnet test command - .net core cli — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-test?tabs=netcore21>
- Donner, R. W. (1993, April 13). *Computer architecture for conserving power by using shared resources and method for suspending processor execution in pipeline*. Google Patents. (US Patent 5,203,003)
- Dunne, J., Malone, D., e Flood, J. (2015). Social testing: A framework to support adoption of continuous delivery by small medium enterprises. In *2015 second international conference on computer science, computer engineering, and social media (cscesm)* (pp. 49–54).
- Dykstra, T., Halter, S., e Ross, C. (2019). *Kestrel web server implementation in asp.net core*. Retrieved from <https://docs.microsoft.com/en-us/aspnet/>

- core/fundamentals/servers/kestrel?view=aspnetcore-2.2
- Ebert, C., Gallardo, G., Hernantes, J., e Serrano, N. (2016). Devops. *Ieee Software*, 33(3), 94–100.
- Eddy, B. P., Wilde, N., Cooper, N. A., Mishra, B., Gamboa, V. S., Shah, K. M., ... Shields, N. A. (2017, Nov). A pilot study on introducing continuous integration and delivery into undergraduate software engineering courses. In *2017 iee 30th conference on software engineering education and training (csee&t)* (p. 47-56). doi: 10.1109/CSEET.2017.18
- Farley, e Humble. (2010). *continuous delivery: reliable software releases through build, test, and deployment automation*. addison-wesley professional.
- Fowler, M. (2006, May). *Continuous integration*. Retrieved from <https://martinfowler.com/articles/continuousIntegration.html>
- Git. (2019). *Git - git-clone documentation*. Retrieved from <https://git-scm.com/docs/git-clone>
- Goldratt, E. M., e Cox, J. (2016). *The goal: a process of ongoing improvement*. Routledge.
- Goodman, A. (2018). *Github - wagooodman/dive: A tool for exploring each layer in a docker image*. Retrieved from <https://github.com/wagooodman/dive>
- Gruver, G. (2016). Starting and scaling devops in the enterprise. In *Starting and scaling devops in the enterprise* (pp. 8–16).
- Help.sonatype.com. (2019). *Repository manager 2*. Retrieved from <https://help.sonatype.com/repomanager2>
- Hiller, J., Johnsen, H., Mason, J., Mulhearn, B., Petzinger, J., Rosal, J., ... others (1992, January 14). *Highly parallel computer architecture employing crossbar switch with selectable pipeline delay*. Google Patents. (US Patent 5,081,575)
- Hüttermann, M. (2012). *Devops for developers*. Apress.
- Latham, L., e Smith, S. (2019). *Integration tests in asp.net core*. Retrieved from <https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-2.2>
- Lock, A. (2016). *Introduction to integration testing with xunit and testserver in asp.net core*. Retrieved from <https://andrewlock.net/introduction-to-integration-testing-with-xunit-and-testserver-in-asp-net-core/>
- Loureiro, T. (2019). *Code analysis with sonarqube + docker + .net core*. Retrieved from <https://medium.com/@thiagoloureiro/code-analysis-with-sonarqube-docker-net-core-aee521ee8931>
- Microsoft. (2019a). *Docker hub - microsoft/mssql-server-linux*. Retrieved from

- <https://hub.docker.com/r/microsoft/mssql-server-linux>
- Microsoft. (2019b). *Entity framework — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/ef/>
- Microsoft. (2019c). *robocopy — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/robocopy>
- Microsoft. (2019d). *Unit test basics — microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019>
- Microsoft. (2019e). *Unit testing c in .net core using dotnet test and xunit microsoft docs*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>
- Microsoft. (2019f). *Web deploy 3.6: The official microsoft iis site*. Retrieved from <https://www.iis.net/downloads/microsoft/web-deploy>
- Nosenchuck, D. M., e Littman, M. G. (1989, March 7). *Multinode reconfigurable pipeline computer*. Google Patents. (US Patent 4,811,214)
- Potash, H., Levin, B. L., e Genter, M. E. (1984, August 21). *Flexible computer architecture using arrays of standardized microprocessors customized for pipeline and parallel operations*. Google Patents. (US Patent 4,467,409)
- Raspberrypi.org. (2014a). *Disable sleep mode - raspberry pi forums*. Retrieved from <https://www.raspberrypi.org/forums/viewtopic.php?t=144833>
- Raspberrypi.org. (2014b). *prevent hdmi from sleeping - raspberry pi forums*. Retrieved from <https://www.raspberrypi.org/forums/viewtopic.php?t=91135>
- Raspberrypi.org. (2016). *Raspberry pi 3 model b*. Retrieved from <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>
- Raspberrypi.org. (2018). *Raspberry pi 3 model b +*. Retrieved from <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>
- Smartbear, I. (2019). *About swagger specification — documentation — swagger — swagger*. Retrieved from <https://swagger.io/docs/specification/about/>
- SonarQube. (2019). *Issues — sonarqube docs*. Retrieved from <https://docs.sonarqube.org/latest/user-guide/issues/>
- Sonatype. (2012). *Managing your .net components with nexus nuget*. Retrieved from <https://www.youtube.com/watch?v=FvurW9XsLN0&t=1078s>
- Sonatype. (2018). *Docker hub - sonatype/docker-nexus*. Retrieved from <https://>

`hub.docker.com/r/sonatype/nexus`

- Udd, R. (2016). Adopting continuous delivery: A case study. In *Adopting continuous delivery: A case study* (pp. 8–).
- Yiran, W., Tongyang, Z., e Yidong, G. (2018). Design and implementation of continuous integration scheme based on jenkins and ansible. In *2018 international conference on artificial intelligence and big data (icaibd)* (pp. 245–249).