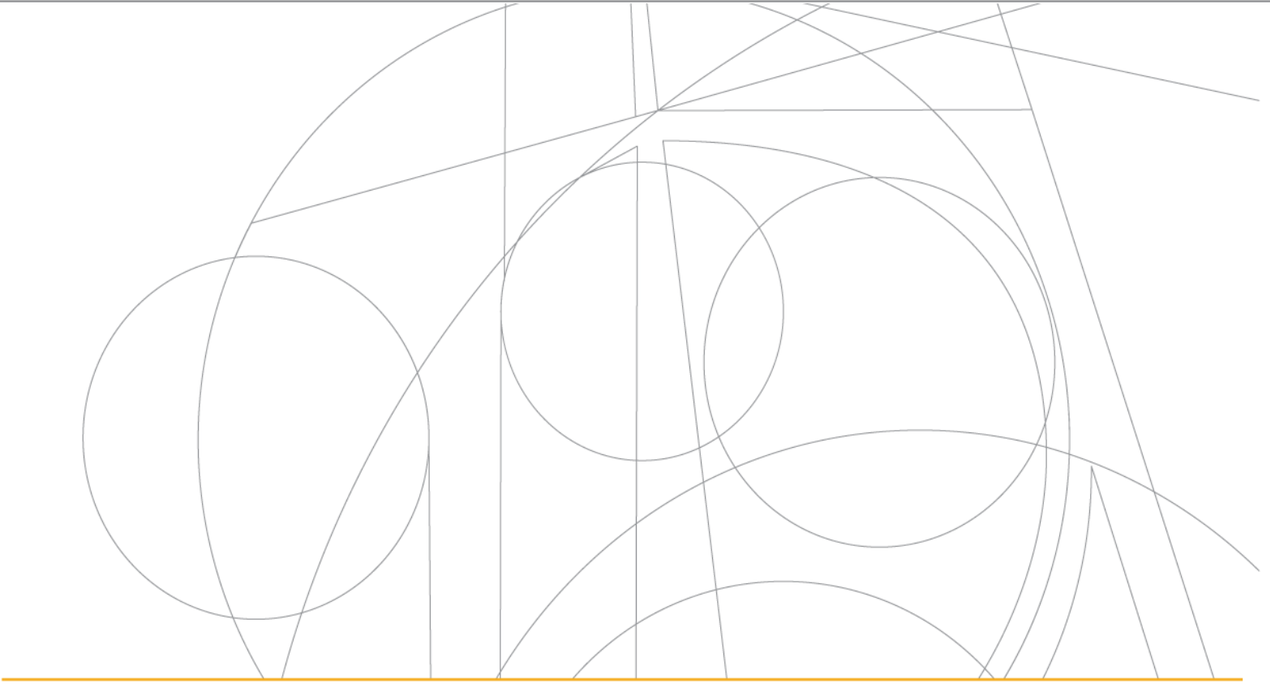
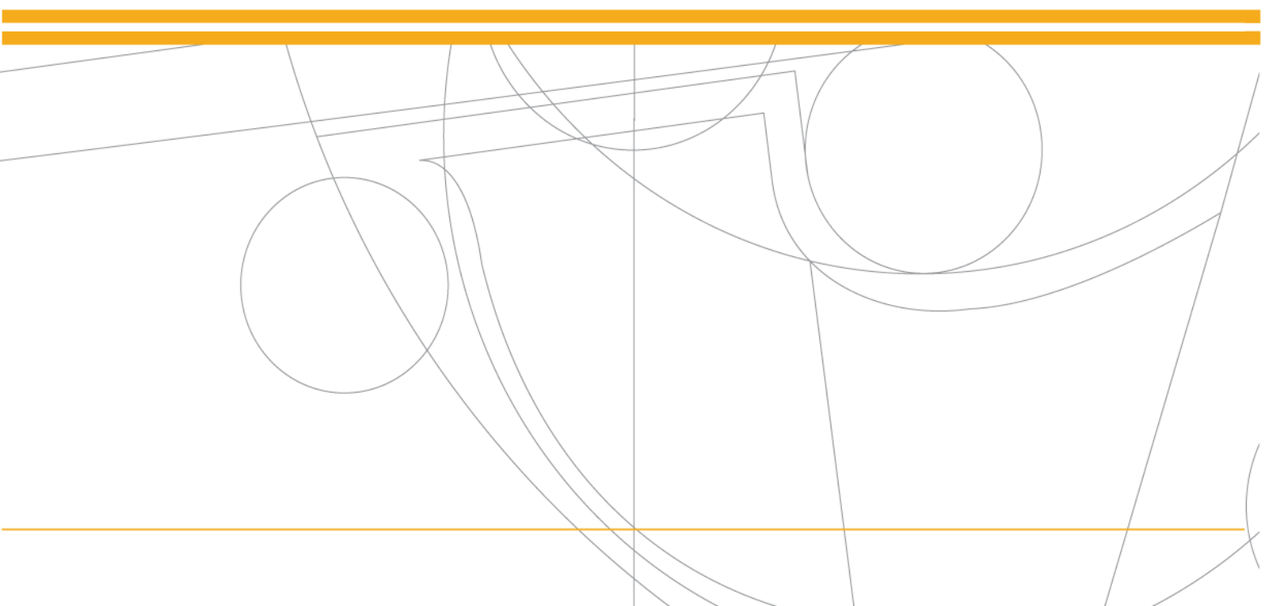


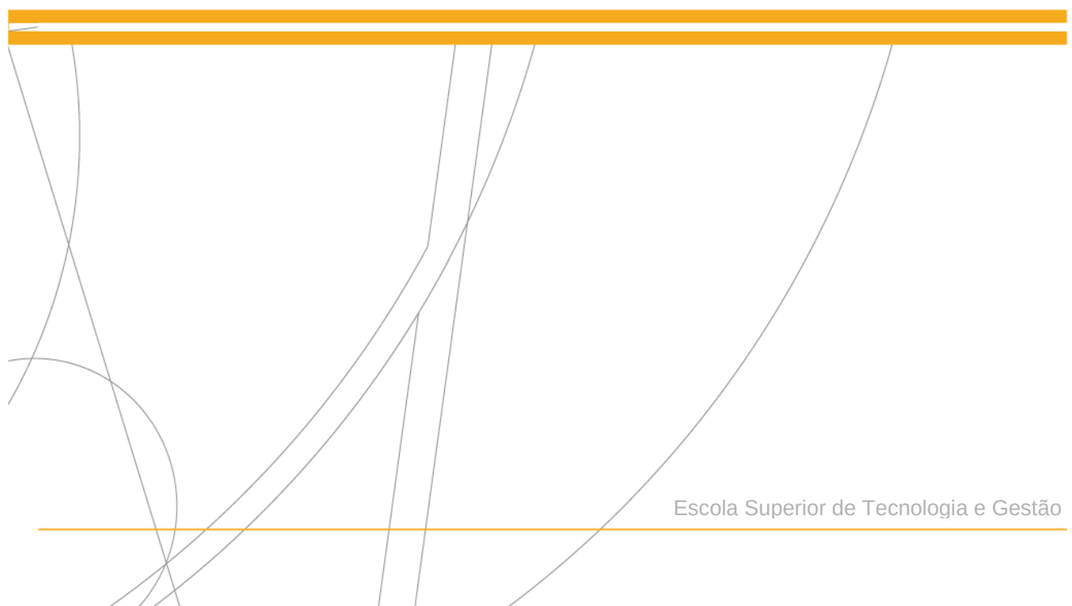
ESTG

INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

MODERNIZAÇÃO DE ARQUITECTURA MONOLÍTICA NUMA ARQUITECTURA DE MICRO-SERVIÇOS

Nuno Amaro Beito Barros





Escola Superior de Tecnologia e Gestão

MODERNIZAÇÃO DE ARQUITECTURA MONOLÍTICA NUMA ARQUITECTURA DE MICRO-SERVIÇOS

2019

INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

Nuno Amaro Beito Barros

MODERNIZAÇÃO DE ARQUITECTURA MONOLÍTICA NUMA ARQUITECTURA DE MICRO-SERVIÇOS

Mestrado em Engenharia Informática

Trabalho de Projeto efetuado sob a orientação de
Doutor António Miguel Cruz
Doutor Pedro Miguel Moreira

Novembro de 2019

RESUMO

Palavras-Chave: Pagamentos, Monólito, Microserviços, Spring, Hibernate, Java, Kafka, Arquitectura de sistemas

O desenvolvimento de software em arquitectura monolítica é tipicamente o primeiro passo aquando do desenvolvimento de uma prova de conceito, devido à sua rapidez de implementação e consequente reduzido *time-to-market*. Actualmente, com a conjuntura de mercado existente e consequente mutação de mercados, tendências e ideias, opta-se frequentemente por uma solução inicial monolítica, efectuando compromissos técnicos com vista a uma rápida adopção de determinada plataforma, produto ou serviço de informática que esteja em desenvolvimento. No entanto, passada a prova inicial do conceito, é constatado frequentemente que o projecto desenvolvido numa arquitectura monolítica assume dimensões cuja natureza o tornam difícil de manter, escalar e estender. Este foi o caso do JumiaPay, que, após uma solução monolítica inicial, se depara agora com este problema e, com vista a maior escalabilidade e manutenibilidade, adoptou uma abordagem para dividir o seu monólito em micro-serviços e, de igual modo, na tentativa de corrigir algumas decisões tomadas que são menos eficientes, efectuadas ao longo do tempo, que foram fruto das várias influências, de natureza humana, técnica ou económica. Este Relatório de Mestrado trata dessa abordagem de reestruturação do JumiaPay numa arquitetura distribuída baseada em micro-serviços.

Foi conseguida a migração da parte do sistema que está responsável pelas integrações com os parceiros, pois foi identificada como a parte que iria escalar mais rapidamente. Neste caso, foi imperativo identificar os pontos comuns entre todas as integrações para que fosse possível integrá-las com o restante do sistema. Esta tarefa revelou-se complexa devido às diferenças nos fluxos entre os parceiros. A separação deste código responsável pela integração com os parceiros permitiu-nos uma separação mais clara de responsabilidades, podendo facilmente aumentar escalabilidade quando necessário e consequentemente, a fiabilidade.

Novembro de 2019

ABSTRACT

Keywords: Payments, Monolith, Microservices, Spring, Hibernate, Java, Kafka, System architecture

Developing a software product using a monolithic architecture is typically the first step for the creation of a proof of concept, due to its reduced implementation time and consequently its reduced time to market. Currently, with the existing market tendencies and constant mutation and shifting of tendencies and ideas, frequently it is defined to start with a monolithic solution, creating technical compromises with the final goal of a specific platform, product or informatic service that's under development. However, after the initial proof of concept it is often noted that the aforementioned project that was originally developed using a monolithic architecture is very difficult to maintain, scale and extend. This was the case of JumiaPay that after an initial monolithic solution now faces these problems, and with the goal of increasing scalability and maintainability adopted an approach to attempt the division of its monolith into microservices, and at the same time change some less-efficient decisions taken throughout time, caused by technical, human or economic factors. This master's report demonstrates the restructuring approach of JumiaPay towards a microservice oriented distributed architecture.

It was possible to migrate part of the current system that's responsible for the payment service provider integrations, as it was identified as the component with the most urgent need for scalability. It was imperative to identify the common points within all the existing integrations so that it would be possible to integrate with the current system. This task was complex due to the different flows of the payment service provider partners. The separation of this code responsible for the integrations allowed us to achieve a clearer separation of responsibilities, allowing us to easily increase scalability when required, and consequently its reliability.

November 2019

ÍNDICE

1. INTRODUÇÃO

O grupo Jumia está com uma presença forte em 14 países [1], com planos de expansão para todo o continente africano. A motivação por parte do grupo Jumia está no desenvolvimento de uma plataforma para que os seus clientes possam comodamente efectuar pagamentos, adicionando assim mais uma peça ao seu ecossistema.

O grupo fornece uma série de serviços para facilitar o dia-a-dia do seu público alvo, como o Jumia Mall onde os clientes podem comprar produtos de vários tipos, desde telemóveis a fraldas, passando por maquilhagem e ferramentas [2], o Jumia One, onde os utilizadores podem pagar as suas facturas, efectuar doações, pedir empréstimos, carregar o telemóvel, entre outros serviços [3]. O Jumia Travel, que agrega na mesma plataforma hotéis e vôos [4] e o Jumia Food, que os clientes podem usar para efectuar encomendas de comida nas principais cidades dos países onde está actualmente implementada esta plataforma [5].

Para facilitar a vida aos clientes, criando hábitos de confiança, criando uma fidelização dos mesmos e para ajudar a garantir a segurança dos colaboradores do grupo [6], foi apresentado o JumiaPay. Esta plataforma, assente num sistema de carteira virtual, nasceu por necessidade de contrariar a fraca adopção dos pagamentos online em África [7] [8]. Este continente é caracterizado pela heterogeneidade dos seus países, legislação, hábitos e costumes. Este serviço de pagamento online, fornecido pela Jumia e actualmente em utilização no continente africano, agrega um número cada vez maior de utilizadores, de acordo com a Bowker, J. [9] e Krishnakumar, A [10].

O propósito deste relatório de projecto de Mestrado é desenvolver um estudo sobre a modernização de arquitecturas monolíticas com vista à sua reestruturação em arquitecturas de micro-serviços. Estas maneiras de desenhar e estruturar software serão analisadas, reflectindo sobre vantagens e desvantagens de uma e outra.

A base para o desenvolvimento deste relatório de projecto é precisamente o JumiaPay, plataforma desenvolvida com base numa arquitectura monolítica, que reflecte a necessidade de uma modernização com vista à sua escalabilidade, manutenibilidade, fiabilidade e extensibilidade. Esta necessidade surge da crescente dificuldade sentida actualmente aquando da implementação de novas funcionalidades nesta plataforma, dada a necessidade de escalar rapidamente e a heterogeneidade geográfica da equipa de desenvolvimento.

Neste relatório de projecto pretende-se abordar de uma forma prática as metodologias que foram sendo escolhidas e adoptadas com base nas necessidades reais e diárias deste projecto, abordando também os compromissos que foram sendo adoptados, quer do ponto de vista do negócio, quer do ponto de vista técnico.

O presente trabalho encontra-se dividido nos seguintes capítulos:

- **Definições e conceitos:** Descrição de conceitos base e siglas, utilizados ao longo do projecto, para simplificação da compreensão do documento.

- **Análise do problema:** Vista geral sobre o estado da arte da solução, os problemas encontrados, e o porquê da necessidade da melhoria do sistema em estudo.

- **Desenvolvimento:** Trabalho desenvolvido com base nos problemas identificados, objectivos delineados e prazos estabelecidos.

- **Discussão dos resultados:** Análise sobre os resultados obtidos, problemas encontrados e as formas de os ultrapassar. Levantamento de possíveis problemas a resolver numa fase futura.

- **Trabalho futuro:** Identificação de falhas encontradas, melhorias identificadas e possíveis atalhos tomados para cumprir prazos. Agregação destes dados e desenvolvimento de uma análise com vista a identificar os pontos a melhorar numa fase posterior.

- **Conclusões finais:** Análise posterior ao trabalho desenvolvido.

2. DEFINIÇÕES E CONCEITOS

2.1. INTRODUÇÃO

Este capítulo apresenta diversos conceitos e definições usados ao longo deste documento, e necessários para a sua cabal compreensão.

2.2. CONCEITOS

2.2.1. STARTUP

Uma startup é uma equipa de talento empresarial com o objectivo de desenvolver inovação, numa forma que seja possível de investir e identificar valor com a ambição de crescer depressa e criar impacto, com um modelo de negócio escalável [11]. Este modelo de negócio assenta numa ideia identificada pelos seus fundadores, com o objectivo de ter um grande impacto no seu mercado alvo, e com vista a evoluir para um negócio sustentável.

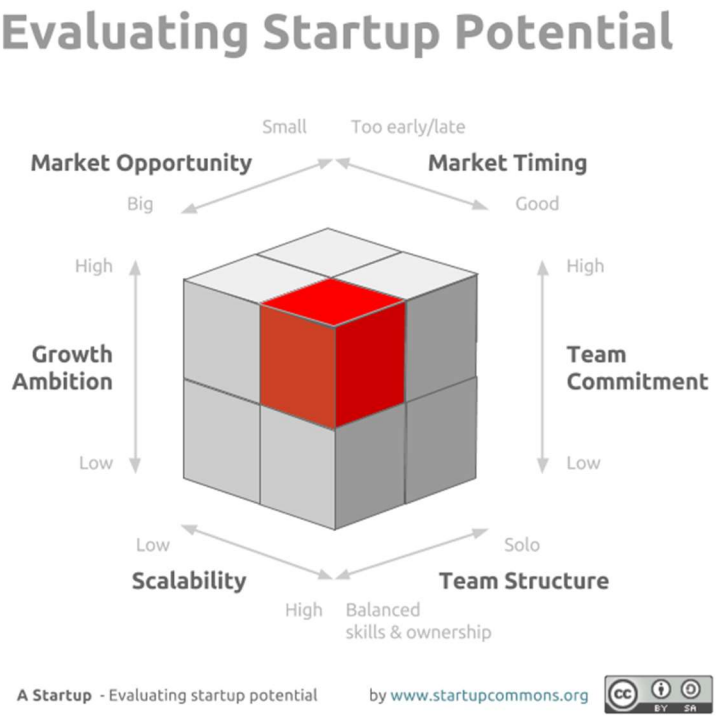


Figura 1-Cubo de avaliação de potencial de startup.

Na Figura 1 podemos identificar que, neste cubo de avaliação de potencial de startups, o cenário ideal depende da existência de uma grande oportunidade de negócio, com um bom timing, uma grande ambição e dedicação da equipa, sendo esta uma equipa equilibrada e com uma grande ambição de crescimento. Com estas condições reunidas, a startup terá as ferramentas necessárias para se tornar num negócio sustentável, sem a necessidade de investimento externo. Estes investimentos

são necessários durante a fase da prova de conceito. Segundo Steve Blank, uma startup é uma organização cujo objectivo é identificar um modelo de negócio repetível e escalável [12]

2.2.2. STANDALONE

O dicionário de Cambridge define que software standalone, ou um computador standalone, é capaz de funcionar independentemente, sem fazer parte de um conjunto ou estar ligado a outros computadores [13]. Neste âmbito, apenas nos referimos ao software, ou seja, a definição de standalone neste contexto é um software que é independente de outros, não sendo requerida a sua conexão a outros sistemas.

2.2.3. ROADMAP

Um roadmap é um plano estratégico que define um objectivo, incluindo os principais passos para alcançar o resultado [14]. Também serve como documento de alto nível que assiste ao pensamento estratégico, apoiando na clarificação dos requisitos gerais e nos prazos das tarefas planeadas. Este documento é um documento de alto nível, não contendo dele requisitos técnicos, nem o levantamento de casos de uso. O roadmap pode ter uma visão tecnológica, ou uma visão de produto. Neste caso, o objectivo é transparecer a visão de produto no roadmap [15].

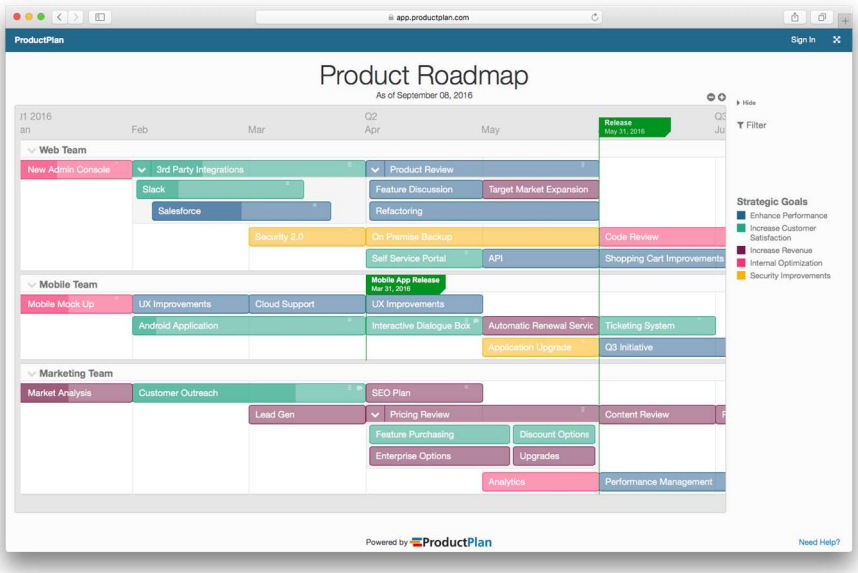


Figura 2 - Exemplo de Roadmap de produto.

Na Figura 2 podemos ver o exemplo de um roadmap dividido por três equipas, a equipa Web, a equipa Mobile e a equipa de Marketing. O roadmap pode ser único entre todas as equipas, ou pode existir um roadmap por equipa, sendo este critério definido pela empresa e pelas equipas.

2.2.4. E-MONEY

E-money, moeda digital, ou dinheiro electrónico, é uma representação virtual de uma unidade de moeda. Esta representação existe no nosso dia-a-dia, com a utilização de cartões bancários, ou os sistemas de home-banking. Esta representação virtual de moeda permite ao utilizador efectuar transacções sem a necessidade de utilizar dinheiro físico [16].

2.2.5. API

API é uma sigla que significa Application Programming Interface, ou interface de programação de aplicações. Segundo Clarke S., o propósito de uma API é a simplificação da programação pela abstracção da implementação existente e expondo apenas os objectos ou acções que o programador precisa. Enquanto, por exemplo um interface gráfico de um cliente de email pode providenciar ao utilizador um botão que concretiza todos os passos para receber e destacar novos emails, uma API para manuseamento de ficheiros facilita ao programador esta tarefa, sem ser necessário ter conhecimento sobre as operações de baixo nível efectuadas pelo sistema de ficheiros [17].

2.2.6. WEB SERVICES (SERVIÇOS WEB)

Um Web Service, ou serviço web é um serviço oferecido por um dispositivo electrónico a outro dispositivo electrónico, utilizando a internet para efectuar a comunicação. De forma simplificada, este serviço providencia uma interface objectual para acesso a dados, com vista a serem consumidos por outro serviço que providenciará a interface para consumo pelo utilizador final. Os exemplos de web services mais comuns são REST e WSDL, habitualmente utilizado em serviços SOAP.

2.2.7. REST

REST, ou Representational State Transfer (em português, Transferência de Estados Representacionais) é um estilo de arquitectura de software que define uma série de restrições para serem aplicadas aquando da criação de serviços web com recurso a operações desprovidas de estado. Estas operações *stateless* permitem a um sistema melhorias de performance, fiabilidade, e a capacidade de crescer reutilizando componentes que podem ser actualizados enquanto um sistema se encontra em execução [18]. Serviços web que respeitem este estilo de arquitectura, providenciam melhor interoperabilidade entre sistemas informáticos na internet [19].

2.2.8. FRAMEWORK

De acordo com Riehle D., uma framework é uma abstracção na qual um produto de software pode obter a sua funcionalidade base, sendo esta estendida ou modificada. Uma framework contém um conjunto de ferramentas e standards, com os quais um software pode ser desenvolvido e estruturado usando práticas habituais, facilitando o desenvolvimento e reduzindo assim o custo deste [20].

2.2.9. Injecção de Dependências

Injecção de dependências é o termo pelo qual a inversão de controle é mais conhecida. A inversão de controle é um princípio da programação, o qual permite aumentar a modularidade de um produto de software, tornando-o mais extensível. A extensibilidade com este padrão de desenvolvimento deve-se á ligação fracamente ligada entre os objectos, permitindo que em vez de criar uns objectos dentro de outros, possamos ter uma instância de um objecto a ser injectada para dentro dos restantes objectos que precisam dela. Com isto, conseguimos que o comportamento dos objectos seja uniforme, com um consumo reduzido de recursos, e com uma estrutura aplicacional mais clara. Podem também definir-se interfaces, utilizá-los como referência e injectar implementações de interfaces consoante as necessidades de cada um desses objectos que recebem outros via injecção [21].

2.2.10. Padrão MVC

Segundo Trygve Reenskaug, T. e Coplien J., o padrão MVC é um padrão arquitectural habitualmente utilizado para desenvolver aplicações interactivas, que dividem uma aplicação em três componentes interligados [22]:

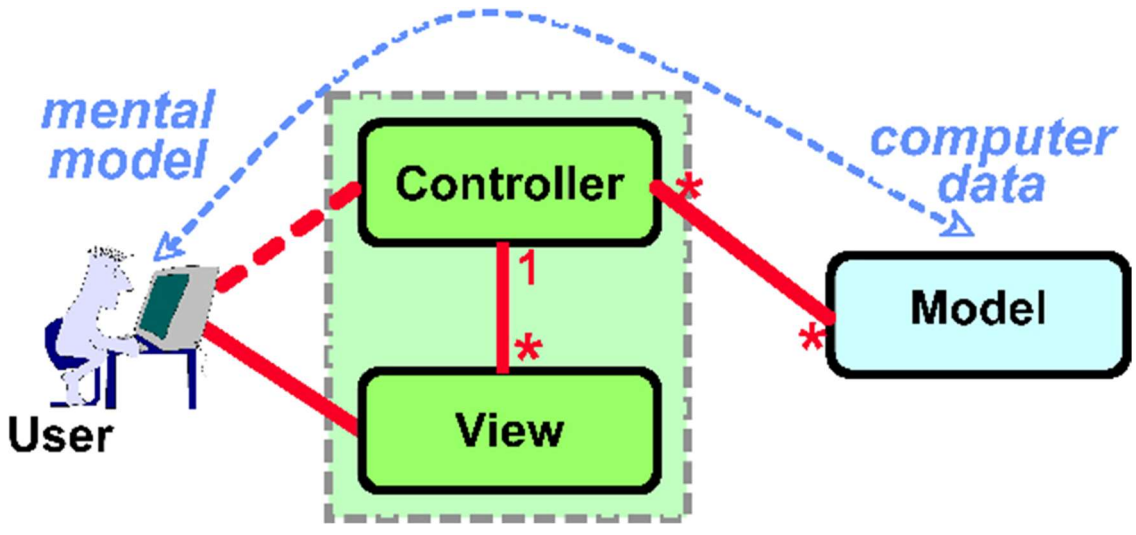


Figura 3 – Padrão MVC.

O modelo é o componente principal do padrão. Neste componente existe a estrutura dinâmica de dados da aplicação, independentemente do interface. Este gere os dados, lógica e regras da aplicação. Na visualização (*view*) temos a representação visual dos dados, podendo isto ser um conjunto de controladores para acesso aos dados, um gráfico, um diagrama, uma tabela por exemplo. O controlador, por fim, é responsável por aceitar *input* de utilizador e convertê-lo em comandos para o modelo ou a *view*.

2.2.11. SPRING, SPRING BOOT

Spring é uma framework aplicacional, com o propósito de facilitar o desenvolvimento de aplicações java. Esta framework, que também permite a injecção de dependências, contém módulos que permitem de forma fácil o manuseamento de dependências, ligação a base de dados, gestão de autorização e autenticação, a implementação do padrão MVC e testes unitários [23].

2.2.12. LEGACY (LEGACY CODE, OU CÓDIGO LEGADO)

Segundo Lopian E., código legado refere-se a código fonte antigo, que pode até ter sido obtido e adaptado de outro projecto, código que já não reflecte as necessidades actuais e que acarreta uma grande resistência à modificação por parte dos programadores [24]. Um sistema legado por outro lado refere-se a um sistema cuja tecnologia já está desactualizada mas ainda continua em utilização [25]. Isto pode significar também que o sistema em questão possa precisar de uma actualização. Um exemplo de um sistema legado é a utilização em 2011 de uma aplicação em MS-DOS (como termo de comparação, o suporte ao Windows XP acabou a 8 de abril de 2014 [26]) para a gestão dos recursos alimentares da marinha dos Estados Unidos. Este sistema, entretanto, foi substituído por um mais recente [27].

2.2.13. SERVIDOR APLICACIONAL

Um servidor aplicacional é uma *framework* de software que providencia os meios para criar aplicações web, e um ambiente de servidor para a sua execução [28]. No caso da linguagem Java, um servidor aplicacional pode servir para a execução de projectos Java EE (Java Platform, Enterprise Edition). A infraestrutura do Java EE está dividida em contentores lógicos. Um contentor gere as transacções usando EJB, *Enterprise Java Beans*. Nestes é contida a lógica de negócio da aplicação. Outro dos contentores está

encarregue da lógica relacionada com web, incluindo servlets (nos quais se estendem as capacidades do servidor aplicacional, alojando aplicações web). De seguida, um contentor utilizado para optimizar a ligação entre sistemas de informação empresarial (EIS), e por último um contentor responsável pela providência de JMS (*Java Message Service*). Esta é uma tecnologia que permite o envio de mensagens entre clientes [29]. Para o caso descrito neste relatório de projecto, apenas o contentor responsável pelo alojamento de aplicações web é de facto relevante. Existem vários servidores aplicacionais, como por exemplo o Apache Tomcat. Este é uma implementação open-source, mantido pela Apache Foundation [30]. Existe também o Glassfish, que foi a primeira implementação da especificação Java EE 6 [31] e também o WildFly, que é mantido pela Redhat [32].

2.2.14. SINGLETON

Na engenharia de software, o padrão *Singleton* é um padrão de desenho de software que restringe a instanciação de uma classe a uma única instância. Isto é útil quando um e apenas um objecto é necessário para coordenar acções em todo o sistema [33]. Alguns críticos consideram que o *singleton* é aplicado frequentemente em cenários nos quais não é vantajoso, introduzindo restrições desnecessárias em cenários nos quais não se justifica [34] [35].

2.2.15. BEANS (SPRING BEANS)

Quando nos referimos a *beans*, ou *spring beans*, referimo-nos a *enterprise java beans* (EJB). Esta é uma de várias funcionalidades da API do Java para a construção modular de software. Estes EJB são componentes que se encontram do lado do servidor de uma estrutura aplicacional, nos quais se encontra lógica de negócio. Um contentor web de enterprise java beans fornece a estrutura necessária para a gestão e utilização dos mesmos [36]. O caso mais habitual de um *bean* é a implementação de um *singleton*, que são objectos de negócio que contém um estado global partilhado dentro de uma *Java Virtual Machine* (JVM). O acesso concorrente a este bean é gerido pelo contentor que gere a execução da aplicação web, e existe uma e só uma instância deste bean, que sendo singular nos remete para a sua definição na engenharia de software, *singleton* [37] [38].

2.2.16. BEAN MANAGER

O *bean manager* é um interface Java que permite a uma aplicação interagir directamente com o seu container [39]. Por outras palavras, a utilização deste interface permite-nos manusear directamente no servidor aplicacional os *beans* que nele existem.

Este interface permite-nos também obter informação sobre o contexto aplicacional, obter referências aos *beans* e referências aos tipos de *beans* e, entre outros métodos, validar um ponto de injecção para verificar se existe um problema com um determinado bean. Caso esta validação falhe, uma excepção é devolvida.

2.2.17. JWT

JWT, ou *JSON Web Tokens,* são um padrão da indústria para representar afirmações entre dois intervenientes [40]. Esta informação pode ser verificada e confiada porque é assinada digitalmente. Estes podem ser assinados usando uma palavra secreta, ou um par de chaves pública/privada [41].

2.2.18. DEBUGGING

Debugging é uma actividade relacionada com o desenvolvimento e teste de software, que consiste em encontrar e resolver problemas.

2.2.19. XML

XML, ou *Extensible Markup Language* é uma linguagem de marcação que define um conjunto de regras para estruturar um documento num formato que seja legível por seres humanos e por máquinas [42].

```
<aluno>
  <nome>Nuno</nome>
  <apelido>Barros</apelido>
  <escola>ESTG</escola>
  <curso>Mestrado em Engenharia Informática</curso>
</aluno>
```

Acima temos um exemplo de XML que descreve um objecto. Este objecto descreve um aluno, com quatro propriedades, cujo conteúdo está representado entre tags, por exemplo `<nome>` e `</nome>`. A utilização do caractere "/" na tag define que se fechou a introdução dos valores para esta propriedade.

2.2.20. CONTEXTS AND DEPENDENCY INJECTION (CDI)

CDI [43] é uma framework standard de injeção de dependências, incluída no JavaEE 6 e superior. Os serviços fundamentais fornecidos pelo CDI são os contextos, onde temos a capacidade de associar o ciclo de vida e as interacções de componentes

providos de estado, bem como contextos bem definidos, mas extensíveis, e a injecção de dependências (inversão de controlo) [44]. Este é um padrão Java cujo objectivo final é a simplificação do desenvolvimento de aplicações web [45].

2.2.21. WELD

O Weld é a implementação de referência do CDI [46].

2.2.22. GUICE

O Guice é a implementação da google do CDI, que permite suporte para injecção de dependências usando anotações. Esta implementação está disponível sob a licença Apache [47].

2.2.23. ROI

ROI, *Return on investment*, é o rácio entre o lucro obtido e o custo de investimento, resultante de investimento de algum tipo de recurso. Um ROI elevado significa que o investimento foi largamente eficiente, sendo o ganho bastante superior ao custo. Como medida de desempenho, é utilizado para medir a eficiência de um investimento [48].

2.2.24. BOOTSTRAP

Bootstraping é um termo utilizado para um processo que é autónomo e inicia sem ser necessária intervenção externa para funcionar. Um exemplo mais frequente é um sistema operativo, mas no caso aqui descrito refere-se o *Spring Boot*, que é um assistente do Spring. Ao utilizar *Spring Boot*, com muito pouco código consegue-se uma framework de *Spring* funcional, a qual pode ser estendida à posteriori consoante as necessidades e requisitos de cada projecto.

2.2.25. MAJOR (VERSÕES DE SOFTWARE)

Major, no que diz respeito a versionamento de software é o primeiro número a contar da esquerda, numa versão. Por exemplo, um software cuja versão seja 5.1.8, 5 é a Major, 1 é a minor e a 8 é a *patch* (correcção) [49].

2.2.26. ORM

ORM, ou *Object-relational mapping*, é uma técnica utilizada para fazer o mapeamento, ou conversão entre tipos incompatíveis ao longo de vários sistemas. Isto cria um conjunto de objectos virtuais que podem ser utilizados dentro do âmbito da linguagem de programação escolhida, sendo da responsabilidade do ORM a conversão para o tipo de destino [50]. De forma mais prática, um ORM permite-nos manusear dados na base de dados, sendo apenas necessário do lado do código da aplicação a criação de objectos e chamada de métodos para atingir a persistência dos dados.

2.2.27. JAVA PERSISTENCE API (JSR-220)

JPA, *Java Persistance API*, é uma especificação de interface utilizada na programação aplicacional que descreve a gestão de dados relacionais utilizando Java SE e Java EE. A persistência, neste contexto cobre três áreas, nomeadamente A API em si representada no package javax.persistence, a linguagem de consulta JPQL (*Java Persistence Query Language*) e os meta-dados que relacionam objectos com a base de dados [51].

2.2.28. CRUD REPOSITORY

CRUD Repository é um interface para operações básicas sobre entidades. CRUD, significando *Create*, *Read*, *Update*, *Delete*, indica as operações básicas a efectuar sobre objectos de base de dados. Este interface é oferecido pela *framework Spring* [52], com o objectivo de acelerar o desenvolvimento de entidades. Com este interface, todas as operações mais comuns relacionadas com persistência de dados estão disponíveis.

```
public interface ArticleRepository extends
CrudRepository<Article, Long>  {
    List<Article> findByTitle(String title);
    List<Article> findDistinctByCategory(String
category);
    List<Article> findByTitleAndCategory(String title,
String category);

    @Query("SELECT a FROM Article a WHERE a.title=:title
and a.category=:category")
    List<Article> fetchArticles(@Param("title") String
title, @Param("category") String category);
}
```

No exemplo acima temos vários exemplos da utilização de *CRUD Repository*. A criação dos métodos da interface deve respeitar o nome dos campos da entidade. Nos

exemplos acima, temos a utilização dos campos `title` e `category` para efectuar consultas de dados. Também é permitida a utilização de SQL nativo, com a anotação `@Query`, caso a query a efectuar seja mais complexa e não esteja dentro do âmbito das funcionalidades do *CRUD Repository*.

2.2.29. QUERIES

Queries, ou consultas, são formas de obter informação dentro de sistemas de base de dados e de armazenamento de informação. SQL por exemplo, significa Structured Query Language [53], ou Linguagem de Consultas Estruturadas.

2.2.30. PACKAGE

Na programação em Java, um package é uma estrutura de organização de código. A divisão do código por packages permite ao programador manter uma estrutura hierárquica de classes e recursos.

2.2.31. TRANSACTION MANAGER

O objectivo do *Transaction Manager* é a gestão das transacções de base de dados. Uma transação tem como objectivo manter a integridade de uma operação, podendo uma transacção ser composta de uma série de operações. Caso uma dessas operações falhe, toda a transacção falha, por outro lado, se todas as operações forem bem sucedidas, a mesma será bem sucedida. Neste âmbito, o gestor de transacções tem como responsabilidade a verificação e execução destas operações, revertendo as alterações efectuadas à base de dados em caso de falha ou persistindo as alterações em caso de sucesso.

2.2.32. TOKEN

Um token, no campo da segurança informática é o que contém a informação que diz respeito ao utilizador, para ser utilizado para efectuar *login* e identificando o utilizador, os seus grupos e privilégios, e informação adicional que seja necessária para o âmbito do token em questão.

2.2.33. JSC

JSC é um token criado pelo grupo Jumia para autenticar o utilizador em todos os sites pertencentes ao grupo, similar em funcionalidade a um token JWT.

2.2.34. ROLE (DE UTILIZADOR)

Role, podendo ser traduzido para papel (de um ponto de vista representativo) é o papel que o utilizador desempenha perante o sistema em questão. Estes papéis tem o propósito de dar ou negar acesso do utilizador ao sistema, podendo barrá-lo de aceder a uma área reservada ou restringi-lo a um determinado número de localizações.

2.2.35. SSO (SINGLE SIGN-ON)

O conceito de *single sign-on* é, de forma sucinta, a autenticação em um único lugar, gerando esta autenticação um token que possa ser utilizado em várias localizações. Um exemplo actual de um sistema de *single sign-on* é o *Google Login*, que podemos utilizar para iniciar sessão num número variado de sistemas com a nossa conta Google.

2.2.36. CAS (CENTRAL AUTHENTICATION SERVER)

O *CAS* é o sistema da *Jumia* que efectua a autenticação dos utilizadores. Este sistema central armazena informação relativa aos utilizadores, e perante a introdução de um email e password válida, devolve um token que é válido perante todas as plataformas *Jumia*.

2.2.37. HTTP

HTTP, ou *Hyper-Text Transfer Protocol*, é um protocolo aplicacional para os sistemas de informação hiper-media, distribuídos e colaborativos [54]. Este protocolo, apresentado em 1991, é uma das pedras basilares da internet tal como a conhecemos. Funciona na base de pedido-resposta, sendo um pedido efectuado a um servidor e sendo devolvida uma resposta ao cliente original com os conteúdos requisitados, podendo estes ser hiper-texto, multimédia ou outros.

2.2.38. FACTORY

Uma factory, em programação orientada a objectos, é um objecto que cria outros objectos. Isto é um padrão de desenvolvimento, permitindo centralizar o comportamento de criação de objectos e facilitando a manutenção do código.

2.2.39. PROXY

Proxy segundo o dicionário Merriam-Webster é a autoridade ou poder por agir em nome de outrém [55]. Neste âmbito a representação é adequada, pois a criação do módulo *paybuddy-proxy* é precisamente para agir em nome do paybuddy, não importando directamente para a estrutura do projecto o seu conteúdo.

2.2.40. 2FA (TWO-FACTOR AUTHENTICATION)

A autenticação a dois factores é uma variante da autenticação a vários factores (*Multi-factor authentication*). Este método permite confirmar uma operação apresentando dois ou mais factores de autenticação. Um exemplo comum da utilização de autenticação com múltiplos factores é o levantamento de dinheiro numa caixa multibanco. A combinação do cartão bancário (algo que o utilizador tem) com o código PIN (algo que o utilizador sabe) permite levar a cabo a operação pretendida [56].

2.2.41. QUARTZ

Quartz é uma biblioteca open-source de agendamento de tarefas que pode ser integrado com virtualmente qualquer aplicação Java, da aplicação mais pequena ao maior sistema de e-commerce. Pode ser utilizado para criar agendamentos simples ou complexos para executar dezenas, centenas ou até dezenas de milhares de tarefas. Estas tarefas são definidas como componentes Java normais que podem executar virtualmente qualquer tarefa desejada [57].

2.2.42. OTP

OTP, ou *One Time Password*, é uma palavra chave gerada no momento, para uma operação específica. A criação destas passwords requer um input do utilizador, que pode ser o despoletar de uma operação ou a utilização de um hard token para obter um código deste género. Um caso de uso habitual para a utilização de *OTP* é o envio de SMS por parte das entidades bancárias para confirmar uma operação que tenha sido despoletada pelo cliente [58].

2.2.43. HARD TOKEN

Um *hard-token* é um dispositivo físico com uma forma de obter um código que é temporário, e expira após um curto espaço de tempo. Este código é gerado com base num algoritmo que varia entre dispositivo e fornecedor. Existe uma chave partilhada

entre o dispositivo e a plataforma, fazendo com que os códigos gerados respeitem o algoritmo comum e garantindo a autenticidade do pedido para o qual o código foi necessário [59].

Figura 4 - Exemplo de um hard-token.

2.2.44. PSP, GATEWAY, PROVEDOR DE SERVIÇOS DE PAGAMENTO

Um provedor de serviços de pagamento (PSP) é uma entidade cuja responsabilidade é efectuar a ponte entre integradores, que é o caso da *Jumia*, e as respectivas entidades financeiras. Estas entidades poderão ser bancos, num caso mais tradicional, ou entidades como a Mastercard que fazem a ponte entre o número de cartão do cliente e as suas contas bancárias.

2.2.45. KAFKA

O Apache Kafka é uma plataforma de transmissão de dados. Este tipo de plataformas tem três características chave. A publicação e recepção de dados transmitidos, similarmente a um sistema de publicação de mensagens, como o JMS. Outra das características é o armazenamento dos dados transmitidos com tolerância a falhas, e a terceira característica é o processamento dos dados transmitidos á medida que vão sendo colocados na corrente de transmissão. O Kafka é executado como um aglomerado de um ou mais servidores, que podem conter os dados em várias localizações. O aglomerado armazena conjuntos de dados transmitidos, armazenados sob categorias denominadas de tópicos, e cada registo consiste numa chave, num valor e num registo horário [60].

2.2.46. KOOPA

Koopa foi o nome dado às aplicações responsáveis por converter pedidos efectuados pela plataforma JumiaPay em pedidos de acordo com os contratos fornecidos pelos provedores de serviços de pagamento. Agem como um tradutor, por forma a garantir a compatibilidade em ambos os pontos de vista de sistema.

2.2.47. ENDEREÇO IP

Um endereço IP é uma identificação numérica atribuída a cada dispositivo que está ligado a uma rede informática que utilize o protocolo IP para comunicação [61]. Estes endereços podem ser atribuídos manualmente em cada máquina, ou automaticamente através de um servidor dedicado para o efeito (Servidor DHCP).

2.2.48. MINDSET

Na teoria da decisão, um mindset é um conjunto de assumpções, métodos ou afirmações que sejam defendidas por uma ou mais pessoas. Um mindset também pode ser visto com uma forma de pensar. Segundo Stark M., quando numa organização, perante criticismo, uma equipa se defende das afirmações com "A culpa não é nossa" isto é causado pelo mindset negativo existente na mesma. Neste exemplo, o mindset nega pontos de vista novos ou diferentes [62]. Segundo o autor, existem também dois tipos de mindset, um produtivo, e um defensivo. A característica principal do mindset produtivo é a receptibilidade a ideias novas, e à busca por conhecimento novo e renovado. O mindset defensivo apenas prevê a aceitação de ideias que não sejam disruptivas ao que está estabelecido actualmente, de forma a defender a organização ou o departamento existente.

2.2.49. GRAPHQL

GraphQL é uma linguagem *open-source* de consulta e manipulação de dados [63]. A vantagem da utilização de GraphQL em comparação com uma API REST é a simplicidade no desenvolvimento. Habitualmente quando é necessário obter dados novos de uma aplicação, usando uma API REST, desenvolve-se um *endpoint* novo por forma a obter os dados e efectuar operações sobre eles. Utilizando GraphQL, apenas um endpoint é necessário, sendo da responsabilidade da equipa de desenvolvimento a ligação entre a API GraphQL e as fontes de dados. Outra das vantagens é o tamanho dos objectos transportados, sendo estes ajustados às necessidades de cada consulta. É habitual o envio e resposta com recurso a objectos que muitas vezes tem campos nulos ou vazios, sendo estes objectos necessários para garantir a compatibilidade, podendo até devolver dados que são desnecessários naquele ponto. Com GraphQL, apenas se envia o necessário, e apenas se recebe o necessário, com base na consulta que foi efectuada [64].

2.2.50. I/O

I/O, abreviatura para Input/Output ou em português Entrada/Saída. Referente ao fluxo de dados transmitidos por um determinado meio.

3. ANÁLISE DO PROBLEMA

3.1. INTRODUÇÃO

O JumiaPay tem por base uma solução desenvolvida por uma empresa cuja especialidade são plataformas de e-money [65]. Esta empresa fornece uma solução chave na mão, oferecendo consultoria quando necessário. Esta solução foi adquirida para ser utilizada na Ásia por um grupo também nascido na Rocket Internet, e acabou por ser utilizado como base para o JumiaPay.

Neste projecto, que foi herdado do continente asiático, encontramos inúmeras referências a esse mesmo continente, como por exemplo personalizações necessárias com base nas moedas utilizadas, integrações com provedores de serviços como envio de e-mails e sms, e provedores de serviços de pagamento. Ou seja, não se considera, de todo, que se partiu de uma base limpa, como é o caso da solução base oferecida pela empresa consultora mencionada no início deste tópico. Isto deve-se ao facto das pessoas que estavam à frente do projecto serem as mesmas, bem como os consultores alocados pela empresa de consultadoria estarem familiarizados com aquele projecto.

A solução monolítica de base desenvolvida por essa empresa, que doravante chamaremos de *core*, ou *core wallet* é uma aplicação Java, que utiliza Spring MVC e uma base de dados relacional MySQL. Numa visão de alto nível, a aplicação está estruturada conforme indicado na Figura 5.

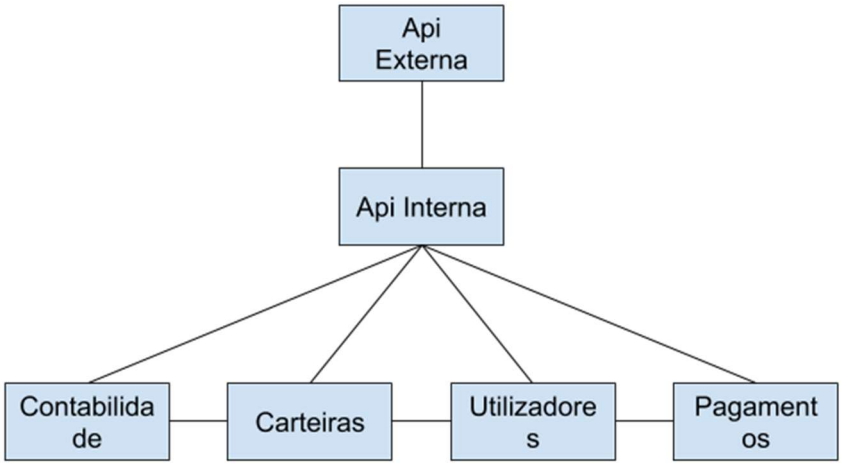


Figura 5 – Visão de alto nível da estrutura da solução core.

Na Figura 5 podemos ver que existe uma API externa, que liga a uma API interna com recurso a injecção de dependências. Esta API interna serve como ponte de ligação entre a API externa e os componentes, tendo o propósito de reduzir a visibilidade para o exterior dos módulos da aplicação, expondo para controladores REST o que é realmente necessário. Os módulos são utilizáveis via injecção de dependências,

disponível na framework Spring. Estes módulos estão divididos por áreas de responsabilidade, nomeadamente contabilidade (accounting), user (utilizador), wallet (carteira) e pagamentos (payments). Nestes módulos encontramos a base de código que consiste na funcionalidade deste sistema.

Na extensão mencionada acima, optou-se por criar um único módulo para estender os módulos de código, um novo módulo de API Interna que estende a API interna do *core*, e um novo módulo de API externa, que estende também da API externa do *core*.

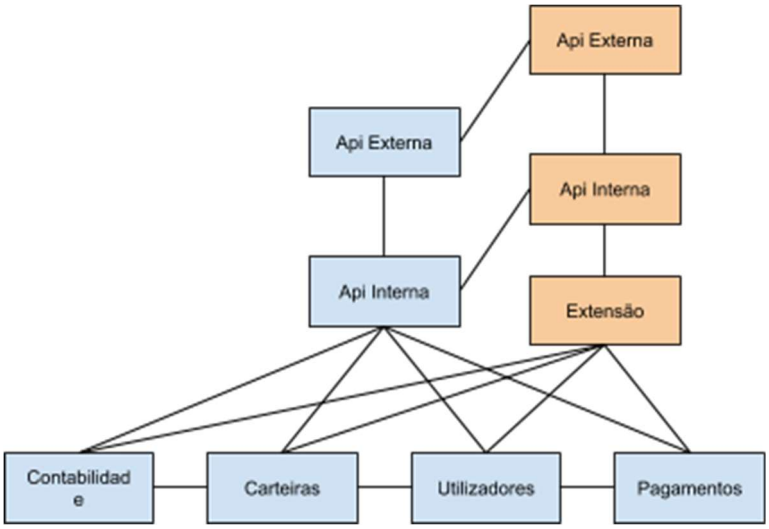


Figura 6 -Visão de alto nível da estrutura core com a extensão.

Verifica-se, na Figura 6, a utilização de um único módulo para a extensibilidade dos módulos existentes do core wallet, representado a laranja com o nome “Extensão”. A estes módulos, referir-nos-emos por *paybuddy* doravante, durante o decorrer deste relatório de projecto.

Dentro do *paybuddy*, encontram-se as integrações com os provedores de serviços de pagamento. Estas integrações, devido às diferenças que caracterizam os países integrantes do continente africano são bastante díspares entre elas, tendo poucos pontos de convergência, o que dificulta as tarefas de análise, levantamento de requisitos, planeamento, desenvolvimento e testes.

A nível de arquitectura, identifica-se que um dos problemas-chave do *paybuddy* é o facto de, contrariamente ao projecto do qual origina, o *core*, o *paybuddy* tem apenas um módulo que contém toda a lógica relativa às extensões de todos os sub-módulos do *core*. Adoptar esta metodologia de desenvolvimento faz com que se ganhe tempo inicialmente, mas, à medida que o projecto cresce, torna-se complexo adicionar funcionalidades e corrigir erros.

3.2. REQUISITOS

Os requisitos para o sistema em questão prendiam-se com escalabilidade, extensibilidade, manutenibilidade e fiabilidade. No início, de forma a obter uma prova de conceito de forma mais célere, tomam-se decisões a nível arquitectural que mais tarde, no decorrer do crescimento normal de uma prova de conceito confirmada, serão revistas por forma a melhorar o produto que foi alvo dessas mesmas decisões.

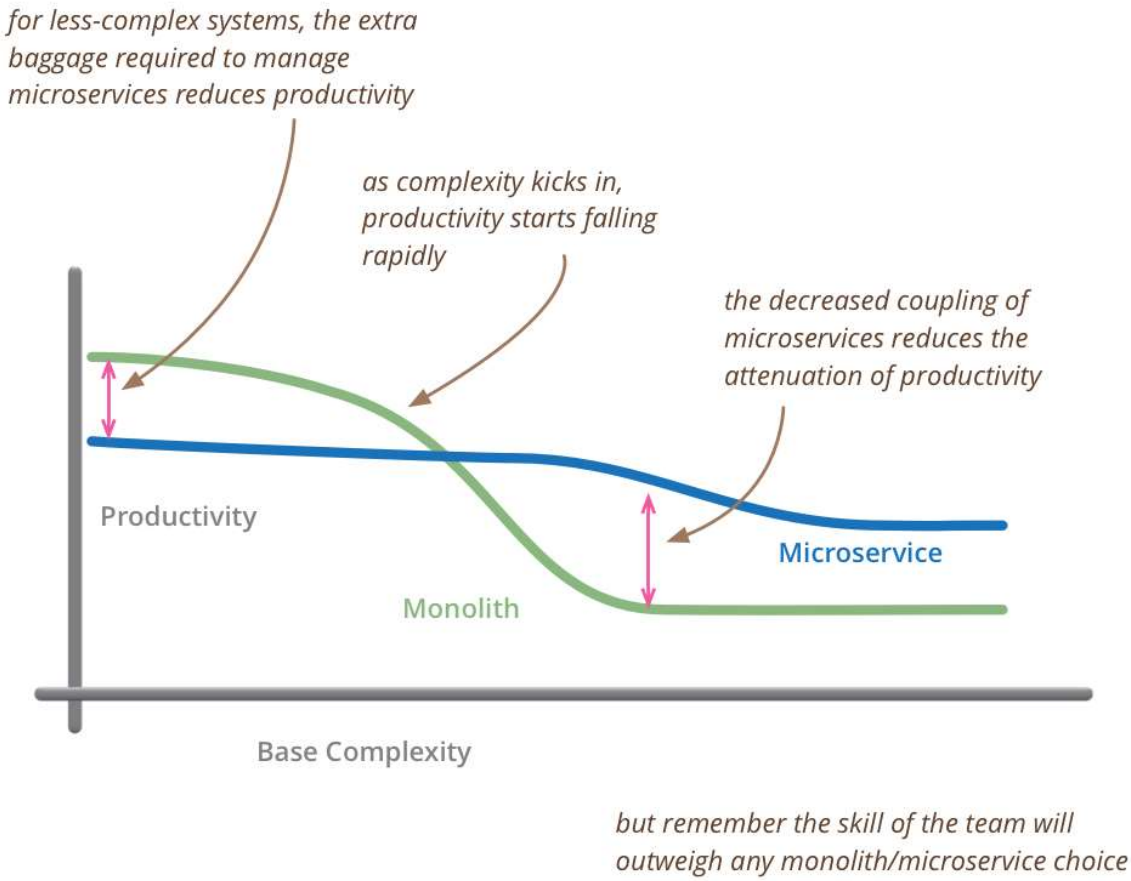


Figura 7 - Gráfico comparativo da produtividade e complexidade (Retirado de [66])

O gráfico acima (ver Figura 7), de Fowler M. [66] demonstra precisamente este problema, comparando monólitos com software desenvolvido com micro-serviços. No início, quando o projecto se encontra numa fase mais embrionária onde o número de funcionalidades e requisitos é pequeno, o autor demonstra que a necessidade de uma maior de complexidade associada ao desenvolvimento de software modular faz com que a produtividade seja mais reduzida, ao invés do desenvolvimento em monólito, onde esta camada não é necessária. No entanto, à medida que a complexidade aumenta, torna-se mais eficaz o desenvolvimento em micro-serviços precisamente devido ao desenvolvimento de forma desacoplada, não sendo necessário fazer alterações noutras partes do código-fonte e tendo esta abordagem a vantagem de tornar a

manutenibilidade muito mais elevada. Num monólito, à medida que a complexidade aumenta, aumenta também a dificuldade em manter o código fonte organizado, fazendo com que à medida que o tempo passa se cometam mais erros, caindo numa crise de software (*Software crisis, 1972*) [67]. Existe também um ligeiro decréscimo na produtividade ao desenvolver com os micro-serviços em mente, devido precisamente ao desacoplamento que se verifica neste caso.

Foram identificadas pela equipa de negócio e a equipa técnica da empresa, como necessidades principais para o JumiaPay, de um ponto de vista tecnológico, a simplificação das suas APIs para integrações com parceiros, a integração fácil com mais provedores de serviços de pagamento e a fiabilidade do produto.

3.3. NOTAS FINAIS

As decisões técnicas que foram tomadas, foram-no sempre com a consideração do melhor de um ponto de vista do produto incluindo sempre a equipa responsável por esta parte no momento de tomar decisões. O objectivo desta plataforma é facilitar a vida aos seus utilizadores, providenciando-lhes um serviço que de outra forma teriam dificuldade em aceder, dadas as características do continente. Com isto em mente, existem compromissos que foram necessários assumir, quer do ponto de vista técnico, quer do ponto de vista do produto.

Estes compromissos originaram decisões, que podem ou não afectar o desenvolvimento técnico. Originam, por vezes, decisões mais precipitadas com o objectivo de cumprir o *roadmap*, sendo depois necessário melhorar os desenvolvimentos que foram encurtados com vista a garantir a qualidade da plataforma.

4. DESENVOLVIMENTO DO PROJETO

4.1. INTRODUÇÃO

Á data deste relatório de projecto, esta solução encontra-se em utilização pelo seu público alvo. Portanto, a modernização deste sistema é um processo que está em curso e como qualquer projecto de software próprio, ou seja, que não seja para entrega a um cliente e que esteja a ser desenvolvido com recurso a metodologias ágeis, é um constante processo de melhoria. O processo de desenvolvimento deste projecto passou por várias fases de estudo e planeamento, com provas de conceito que foram utilizadas, e outras que não evoluíram de uma fase embrionária.

Este processo teve início com uma prova de conceito, a nível técnico. Seria imperativo ter a possibilidade de expandir o projecto existente, ramificando-o internamente por forma a ser possível utilizar apenas o necessário, efectuando assim uma muito necessária limpeza de código fonte considerado *legacy*, que veio da fase anterior deste projecto e também de desenvolvimentos que foram feitos pela equipa até então, fruto da inexperiência da equipa e de decisões da equipa de produto que levaram a este estado da arte do JumiaPay.

Como vimos anteriormente, a aplicação é executada através de um artefacto *war* que é colocado num servidor java aplicacional, que neste caso é o *Tomcat*, mas poderia ser outro como por exemplo *Glassfish* ou *Wildfly*. Este artefacto war contém todo o contexto aplicacional que é gerido pela framework Spring, que tem entre outras vantagens a implementação do JSR-330 que é a injecção de dependências Esta aplicação do padrão de desenho da inversão de controlo faz com que a utilização de memória de uma aplicação seja reduzida devido à utilização de singletons, que são instâncias únicas de *beans* que estão disponíveis em todo o contexto aplicacional. Isto facilita o desenvolvimento na medida que a extensibilidade ou alteração de um determinado serviço apenas é efectuado num único local.

Este conceito de injecção de dependências é extremamente importante neste projecto, pois toda a arquitectura foi desenhada com base na utilização de *beans*, como se verifica no *core*, e consequentemente no *paybuddy*.

4.2. EXTENSÃO DO CORE

Tendo isto em consideração, optou-se por criar um novo módulo, o *supermario*. Este módulo novo, criado de raíz com uma arquitectura modular em mente (mas sendo nesta fase um monólito), visa estender módulo a módulo os módulos existentes, ao invés do trabalho feito no *paybuddy*. Assim, para além de uma melhor estruturação do código, pode-se trabalhar com vista a, no futuro, ser efetuada uma divisão em micro-serviços destes sub-módulos da nossa arquitectura de software. Assim sendo, a estrutura planeada e concretizada é conforme a Figura 8.

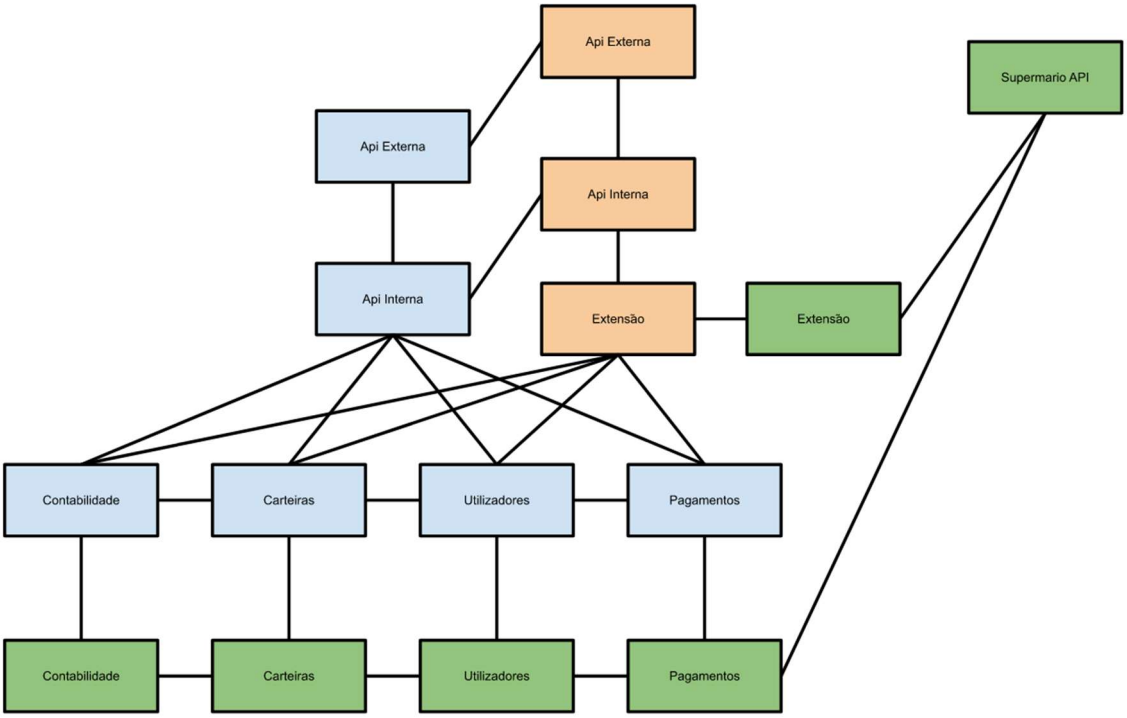


Figura 8 - Esquema de alto nível da solução core, extensão e novo módulo

Verifica-se a verde, na Figura 8, o novo módulo cujo nome de código é *supermario*. este módulo contém submódulos cuja função é única e exclusivamente estender o módulo que é hierarquicamente superior, respeitando assim a modularidade referida anteriormente.

Tomaram-se também algumas decisões no que diz respeito à acessibilidade da API, onde podemos ver que ao invés de uma API externa e de uma API interna, temos apenas uma API. Isto deve-se ao facto de não termos identificado qualquer tipo de vantagem, no nosso paradigma de negócio, na utilização da camada de abstracção oferecida pelo *core*. Outro dos factores desta reformulação deve-se à segurança oferecida pela infra-estrutura e pela forma como foi configurada. Para terminar, deve-se ainda acrescentar que o nível de acesso aos recursos oferecidos pela API será

controlado por um *JWT*, que contém a informação relativa ao utilizador que está a aceder ao recurso.

Quer o *core*, quer o *paybuddy* foram desenvolvidos com recurso á *framework Spring*, versão 4, e para a configuração dos seus *beans* foram utilizados ficheiros *XML*. Nestes ficheiros, encontramos os objectos que permitem a configuração dos *singletons* que compõem a nossa aplicação. Existe por cada módulo do core um ficheiro de configuração, que regista os singletons referidos acima para que possam ser carregados pelo motor do *spring*, que os coloca no seu contexto. Este contexto pode-se comparar a um registo, onde facilmente pelo nome se identificam os componentes que estamos a utilizar na nossa aplicação.

A primeira restrição encontrada foi a framework a utilizar. Migrar toda a estrutura de *Spring* para outra qualquer framework de *IoC*, como *CDI*, *Weld*, ou *Guice*, por exemplo seria algo moroso devido à experiência extensa da equipa em apenas uma framework. Outra questão importante aqui seria a ausência de suporte por parte da empresa consultora que desenvolveu o *core*, e por fim o *ROI*. Este seria demasiadamente longo, pelo que a decisão pendeu para a continuidade da utilização da framework Spring, com a qual a equipa já estava familiarizada.

A configuração dos beans que referimos acima pode fazer-se de duas formas, por ficheiro *XML* ou por anotação. Pode distinguir-se a vantagem da configuração em *XML* sendo a localização única desta configuração, podendo dividir-se por vários ficheiros se necessário. A vantagem da anotação em relação ao XML é a simplicidade com que se declara uma classe como sendo um bean, podendo ser um *@Component*, *@Service*, *@Repository* ou *@Controller*. O que distingue estes tipos diferentes de anotação é a localização onde são anotados. Um *@Component* é um tipo genérico de bean que é gerido pelo *Spring*, Um *@Service* anota na camada de serviço, um *@Repository* na camada de persistência e um *@Controller* é anotado na camada de controlador [68]. O fim atingido com uma ou outra forma de configurar é o mesmo, que é o registo dos nossos beans. Para o *core wallet* e para o *paybuddy* foi escolhida a configuração por XML, mas para o *supermario* preferiu-se a configuração por anotação. Tendo isto em consideração, será necessário apresentar os dois tipos de configuração para percebermos do que se trata. Abaixo temos um exemplo de configuração com XML:

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.exemplo.OlaMundo">
      <property name = "message" value = "Hello World!"/>
   </bean>
```

```
</beans>
```

De seguida, verificamos a classe que será injectada com estas configurações:

```
package com.exemplo;

public class OlaMundo {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

Quando for feita a chamada ao método `getMessage()` do nosso *bean*, será impressa para a consola a seguinte mensagem:

```
Your Message : Hello World!
```

Agora, veremos um exemplo onde não se utiliza configuração por *XML*. Teremos abaixo, a declaração do nosso *bean*:

```
package com.exemplo;

@Service
public class OlaMundo {

   private String message;

   public OlaMundo(@Value("${my.message}") String message) {
       this.message = message;
   }

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

Vemos que, de forma diferente do caso anterior, neste bean temos um construtor com uma anotação *@Value*. Isto faz com que no momento de criação do nosso bean, ele carregue a propriedade que está entre os caracteres `${}`. Esta propriedade deve estar num ficheiro com extensão ".properties" e localizável pelo Spring. O ficheiro de exemplo terá o seguinte conteúdo:

```
my.message = Hello World!
```

E num outro serviço que precise deste serviço, apenas é necessário injectá-lo, e passar os parâmetros conforme se verifica na configuração acima, por *XML* ou anotação:

```
@Autowired
OlaMundo olaMundoInstance;
```

De igual forma, quando se chamar o método `getMessage()` do nosso bean, o resultado será a impressão da mensagem para a consola.

Sendo estes métodos de criação e gestão de *beans* tão distintos, colocava-se o problema de haver uma harmonia entre o existente, que era em *XML* e o pretendido, com anotações. Migrar as configurações existentes para anotação seria algo indubitavelmente extenso, do qual não se retirava ganho suficiente para justificar o tempo investido. Portanto, a abordagem seguida foi criar uma configuração que pudesse ser lida pelo novo contexto da nova aplicação (o *supermario*), que funcionasse em simultâneo com os XML antigo, e as novas configurações por anotação.

Para conseguir este resultado, foram criados em cada um dos módulos *beans* de configuração, com a anotação *@Configuration* e visíveis ao contexto do *Spring*. Estes beans teriam a responsabilidade de carregar não só as classes e o contexto aplicacional próprio, como também fazer a gestão da herança a nível de *beans*, e propriedades. Abaixo, temos o exemplo de uma das classes de configuração criada:

```
@Configuration
@ComponentScan(basePackages = {"com.jumiapay.wallet",
"com.trimplement.wallet.server.wallet"})
@ImportResource({
	"classpath*:/META-
INF/spring/com.trimplement.wallet.server.wallet.*.xml"
})
@PropertySource({
	"classpath:/META-
INF/config/com.trimplement.wallet.server.wallet.api.properties",
	"classpath:/META-
INF/config/com.trimplement.wallet.server.wallet.impl.properties",
	"classpath:wallet.properties",
})
@Import({AccountingConfig.class, PaymentConfig.class,
CommonConfig.class})
@EnableTransactionManagement
public class WalletConfig {
}
```

De notar que neste exemplo foi retirada a informação do package, bem como os imports. Como podemos verificar acima, Este ficheiro de configuração diz respeito ao módulo *Wallet*. Não só esta configuração tem visibilidade dos beans sob o seu package (*com.jumiapay.wallet*), como também gere o que foi criado com recurso a ficheiros *XML* no módulo *wallet* do *core*. Assim, quando é necessário desenvolver algo no módulo de *wallet* do *supermario*, temos acesso a todo o código que já foi desenvolvido no *core*. Isto permite-nos ter acesso às funcionalidades desenvolvidas anteriormente, através dos respectivos componentes, serviços e repositórios. Deste modo, desenvolver uma funcionalidade torna-se mais célere.

Ao desenvolver um bean novo, apenas é necessário ter em atenção a anotação com a definição correcta. Esta classe de configuração encarrega-se de procurar por todos os *beans* que estejam sob o seu *package*, sendo depois possível a injecção deste mesmo *bean* em qualquer local que tenha visibilidade ao módulo, sendo neste caso o módulo *wallet* a fornecer os seus *beans* a quem efectuar a sua importação.

A próxima questão tem a ver com a versão da framework. Existe actualmente um *bootstrap* de *Spring*, chamado *Spring boot,* que permite aos programadores, de forma fácil e com recurso a anotações, configurar automaticamente funcionalidades como acesso a bases de dados e gestão de controladores *REST*. No entanto, a nossa aplicação utiliza uma versão cuja *major* é a versão 4, pelo que estamos restringidos à versão 1.5 do *spring boot*. Além desta limitação, temos também a limitação da versão do *Hibernate*, também numa *major* de versão 4. Isto significa que, para além de estarmos limitados a esta versão de *ORM*, não podemos utilizar uma das valências do *Spring boot*, que é o *Spring Data JPA*. Esta implementação da *Java Persistence API* (JSR-220) [51] tem uma sintaxe mais limpa nos serviços de acesso a dados, denominados habitualmente por repositórios. Abaixo temos um exemplo de um *CRUD Repository* que acede a dados apenas estendendo uma interface:

```
package com.exemplo;

(…)

public interface CustomerRepository extends CrudRepository<Customer,
Long> {

    List<Customer> findByLastName(String lastName);
}
```

Conforme podemos verificar acima, com a extensibilidade da interface CrudRepository consegue-se facilmente obter uma lista de objectos, neste caso

Customer. Com Hibernate, a criação de um repositório é um pouco diferente, conforme podemos ver no exemplo abaixo:

```
package com.exemplo;

(…)

@Service
public class UserServiceImpl {

	private UserRepository repository;

	@Autowired
	public UserServiceImpl(UserRepository repository) {
		super();
		this.repository = repository;
	}

	@Override
	public List getAllUsers() {
		List list = new ArrayList();
		repository.findAll().forEach(e -> list.add(e));
		return list;
	}
}
```

No exemplo acima verificamos que para obter uma lista com os dados temos uma complexidade acrescida. Precisamos de um serviço, que vai aceder a um repositório de dados (que não está representado aqui para simplificar este exemplo). Este serviço terá que receber este repositório como injecção, e sempre que necessário o consultar para obter estes dados. Dada esta complexidade, a utilização do *Spring Data JPA* seria uma vantagem a nível de facilidade de desenvolvimento em relação ao *Hibernate*, mas de novo o ROI seria longo demais para justificar esta mudança devido à quantidade de entidades registadas na aplicação e à sua conversão. Outra das razões que faz com que se prefira o Hibernate neste caso prende-se com a falta de funcionalidades do *Spring Data JPA*. A simplicidade associada tem um custo, e esse custo é a ausência de funcionalidade para a criação de *queries* elaboradas, que façam a ligação entre várias tabelas, ou que realizem a conversão dos dados resultantes numa lista de objectos de um tipo diferente, como *String* ou *Integer*.

Na interacção com a base de dados, utilizam-se transacções. Estas são unidades de trabalho atómica ou seja, todos os passos efectuados dentro desta unidade de trabalho terão de ser bem sucedidos, caso contrário a transacção não tem sucesso e é revertida. A Figura 9 demonstra o fluxo de uma transacção [69]:

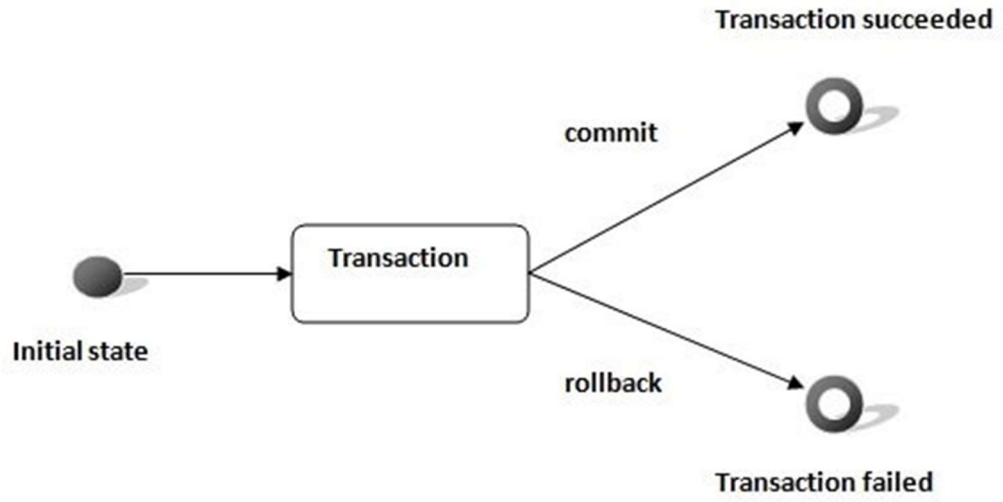


Figura 9 - Diagrama descritivo do fluxo transacional [69].

Conforme podemos ver na Figura 9, a transacção apenas pode ter um estado final, sendo este dependente do decorrer das operações efectuadas dentro da mesma. Caso exista alguma operação que cause uma excepção, é habitual fazer a reversão da transacção. Caso contrário, a transacção é enviada ao gestor de sessões que comunica com a base de dados.

Por forma a utilizar as transacções de uma forma correcta, o *Hibernate* tem um gestor de transacções, ou *Transaction Manager*. Este encarrega-se de fazer a gestão entre a nossa aplicação e o gestor de sessões, que finalmente comunicará com a base de dados. O gestor de transacções tem visibilidade sobre as entidades que pode persistir, sendo no caso do *core* e do *paybuddy* esta listagem de entidades feita através de mais um *XML* de configuração.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
Configuration DTD 3.0//EN" "http://www.hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
<hibernate-configuration>
      <session-factory>
      <!-- Package containing our Hibernate user types -->
      <mapping
package="com.trimplement.wallet.server.common.storage.hibernate"/>
      <mapping
package="com.trimplement.wallet.server.payment.impl.storage"/>
      <mapping
package="com.trimplement.wallet.server.paybuddy.impl.storage.hibernat
e"/>

      <mapping
package="com.trimplement.wallet.server.payment.impl.dom"/>
      <mapping
package="com.trimplement.wallet.server.user.impl.dom"/>
```

```
      <mapping
package="com.trimplement.wallet.server.paybuddy.impl.user.dom"/>
      <mapping
package="com.trimplement.wallet.server.wallet.impl.transaction.purcha
se.dom"/>

      <mapping
class="com.trimplement.wallet.server.wallet.impl.transaction.purchase
.dom.DbRefund"/>
       (...)

      </session-factory>
</hibernate-configuration>
```

Isto levantou uma questão pertinente aquando da criação do *supermario*, pois no momento em que surgia a necessidade da criação de uma nova entidade era necessário alterar este ficheiro *XML*, mas a entidade sendo criada no *supermario*, os módulos do *core* e *paybuddy* não teriam visibilidade sobre a mesma.

Para superar esta barreira, foi feita uma alteração nos *beans* que configuram os gestores de transacções do core e do paybuddy. Esta alteração permite a estes serviços de configuração carregar os *XML* já existentes e, após o carregamento dos XML, é efectuada uma pesquisa sobre os packages do respectivo módulo do *supermario* à procura de entidades.

Estas entidades devem estar anotadas com *@Entity*, o que faz com que aquela classe seja marcada para ser detectável pelo motor de persistência, fazendo com que possa então ser utilizada pelo *Hibernate* para manusear dados.

```
(...)

@Getter
@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Entity
@Table(name = PaymentConstants.DB_TABLE_EXTERNAL_PSP_REQUEST_LOGGING)
public class DbPspExternalRequestLog implements HasID {

    @Id
    @Column(name = COL_ID)
    @GeneratedValue(generator =
PaymentConstants.DB_GENERATOR_EXTERNAL_PSP_REQUEST_LOGGING)
    @SequenceGenerator(name =
PaymentConstants.DB_GENERATOR_EXTERNAL_PSP_REQUEST_LOGGING,
sequenceName =
PaymentConstants.DB_GENERATOR_EXTERNAL_PSP_REQUEST_LOGGING)
    @AccessType(CommonConstants.HIBERNATE_ACCESS_PROPERTY)
    private long id;

    @Column(name = COL_REQUEST_ID)
    private String requestId;
```

```
    (...)

}
```

Acima, temos o exemplo de uma entidade, anotada com **@Entity** para que seja visível pelas configurações de Hibernate que façam a procura por package, e que não dependam da indicação directa da entidade, seja por *XML* seja por classe de configuração. Abaixo temos a classe que nos vai permitir alterar o *bean* `payment.impl.sessionFactory`. Este *bean* é uma *factory*, ou seja, a sua responsabilidade é criar sessões:

```
(...)

@Component
@Slf4j
public class PaymentTransactionManager implements
BeanFactoryPostProcessor {

    @Resource
    ApplicationContext context;

    @Override
    public void postProcessBeanFactory(final
ConfigurableListableBeanFactory configurableListableBeanFactory)
throws BeansException {
       try {
             log.debug("Overriding payment.impl.sessionFactory");
             final LocalSessionFactoryBean sessionFactoryBean =
(LocalSessionFactoryBean)
configurableListableBeanFactory.getBean("&payment.impl.sessionFactory
");
             sessionFactoryBean.setPackagesToScan(
                          "com.jumiapay.payment"
             );
             sessionFactoryBean.afterPropertiesSet();
       } catch (IOException e) {
             log.error("Unable to override
payment.impl.sessionFactory: " + e.getMessage());
       }
    }
}
```

Do exemplo fornecido acima, retiramos o mais importante:

```
final LocalSessionFactoryBean sessionFactoryBean =
(LocalSessionFactoryBean) configurableListableBeanFactory.getBean
("&payment.impl.sessionFactory");
```

Esta linha permite-nos obter a *factory* que devolve as sessões. Note-se a utilização do "&" antes do nome do bean. Isto é necessário para dar ao *bean manager* a

instrução de que, de facto, queremos obter a factory. Caso contrário, por defeito, a instrução `getBean()`, quando executada numa factory, devolver-nos-ia uma instância.

O nosso objectivo é alterar todas as instâncias, ou seja, em vez de alterarmos uma única instância, alteramos a *factory* onde as instâncias são geradas de forma a obter estas mesmas instâncias já com as alterações pretendidas.

```
sessionFactoryBean.setPackagesToScan("com.jumiapay.payment");
```

Esta linha indica à nossa *factory* que é necessário fazer *scan* daquele *package* aquando da criação de uma instância de uma sessão transaccional, para que as entidades que estejam sob aquele package sejam localizadas.

Por fim, o método que efectiva as alterações, o método `afterPropertiesSet()`. Este método existe porque o bean em questão implementa o *interface* `InitializingBean`, o que nos providencia este método que será executado quando todas as propriedades forem carregadas correctamente.

```
sessionFactoryBean.afterPropertiesSet();
```

Com a execução deste método, a *factory* que nos devolve as instâncias de *SessionFactory* já fica preparada para, não só carregar as definições de entidade que foram estabelecidas no ficheiro de *XML* que existe no módulo do *core*, mas também está preparada para carregar as entidades que existem no supermario. Assim conseguimos garantir a retro-compatibilidade, não sendo necessária a criação de gestores de transacções novos.

Como cada módulo do core tem um gestor de transacções e uma fábrica de sessões, à medida que são adicionadas entidades nos diferentes módulos do *supermario* é necessário adicionar esta configuração em cada um dos módulos a que as entidades dizem respeito.

4.3. AUTENTICAÇÃO

Como prova de conceito, com o objectivo de demonstrar que seria possível a extensão do *core* e do *paybuddy* por forma a serem acessíveis por outros módulos, foi criado o módulo de autenticação. Este módulo tem como objectivo prático a autenticação de utilizadores, e como objectivo técnico transformar um *JSC*, que é um token de autenticação do grupo Jumia) num *JWT*. Neste *token* temos a informação do utilizador, como email, nível de permissão do utilizador (*role*), o emissor do *JWT*, e a sua validade. Abaixo temos um exemplo de um *token* descodificado:

```
{
  "alg": "HS256"
}
{
  "uid": {
      "type":"User",
      "id":"AAKLASMk7LOBBxPN02CE3l2G3RC-IFPd3ieQdUyIcRr63LXWnVsfr-4t"
   },
  "email": "hellotoken@maildrop.cc",
  "roles": [
      "CONSUMER"
  ],
  "exp": 1531927070,
  "iss": "supermario"
}
```

O propósito da utilização destes token é, primeiro, a utilização de um standard que é suportado por várias *frameworks* sem necessidade de adaptação e, em segundo lugar, a necessidade de obter a informação da autorização de determinados utilizadores para outros projectos satélite ao sistema de *wallets* do JumiaPay. A autenticação e a autorização diferem no propósito:

A autenticação tem como objectivo garantir a autenticidade das afirmações que o utilizador faz, sobre quem é, garantindo que é quem diz ser [70].

A autorização tem a ver com o nível de acesso, ou seja o que o utilizador pode fazer com a sua autenticação [71].

De um ponto de vista da segurança dos sistemas de informação, garante o não-repúdio. O diagrama na Figura 10 representa num nível mais alto todo o processo de **autenticação**.

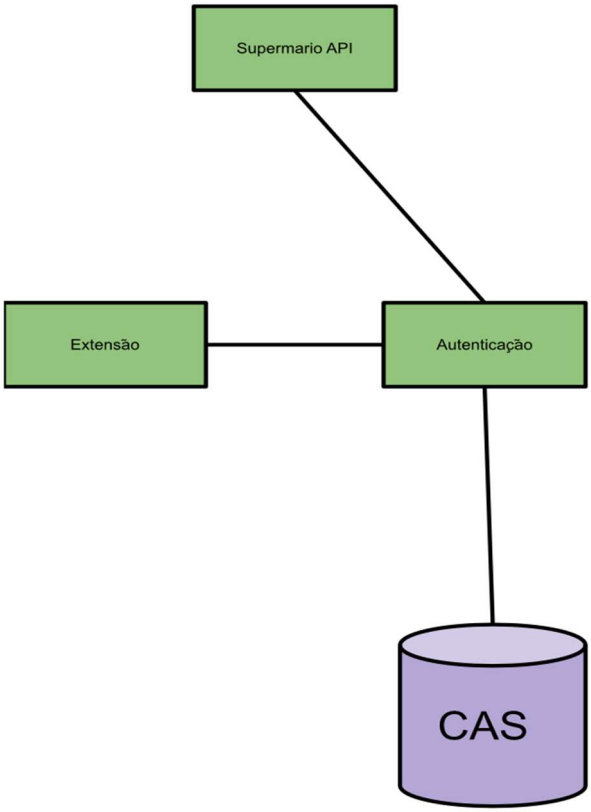


Figura 10 – Processo de autenticação

Os pedidos *REST* entram pela *Supermario API*. Estes pedidos contém o tipo de token, que neste caso é *JSC*, e o próprio token. Esta implementação foi assim decidida pois deixa-se em aberto a extensão com outros sistemas de *SSO (Single sign-on)*, como o disponibilizado pela *Google* ou o *OpenID.*

Após a recepção do token, é necessário determinar qual o email do utilizador. Para isso, efectua-se uma chamada REST ao CAS (Central Authentication Server) que é o serviço central da Jumia que guarda a informação sobre os utilizadores.

Com os dados de identificação do utilizador, acede-se aos serviços disponibilizados pela extensão (*paybuddy*) já via injecção de dependências, e com o email obtido valida-se a existência do utilizador. Caso este exista, devolve-se a sua informação necessária para a criação do token *JWT*, que é o número de utilizador, email e *role*. Caso não exista é criado no momento, com um *role* de *consumer* (cliente) e a mesma informação será devolvida para a criação do *JWT*.

No que diz respeito à autorização, esta é feita através de um filtro que é aplicado em qualquer ponto da aplicação, com recurso a uma anotação que foi criada com este fim. A anotação tem um processador, que verifica os métodos durante a sua execução, e aí podemos interceptar os métodos para permitir a sua execução, ou interrompê-los caso seja necessário. Abaixo temos o código que verifica a permissão do utilizador, que se encontra no processador da anotação:

```
(...)

@Aspect
@Component
@Slf4j
public class BouncerProcessor {

    @Resource
    JwtTokenService jwtTokenService;

    @Around(value = "@annotation(com.jumiapay.api.security.Bouncer)")
    public Object checkPermissions(ProceedingJoinPoint joinPoint)
throws CommonException {
       try {
             for (final Object parameters : joinPoint.getArgs()) {
                   if (parameters instanceof HttpServletRequest) {
                         HttpServletRequest httpServletRequest =
(HttpServletRequest) parameters;
                         final ApiRequestInfo apiRequestInfo =
jwtTokenService.getApiRequestInfoFrom(httpServletRequest);
                         final Collection<String> roleList =
apiRequestInfo.getRoles();
                         for (String role : roleList) {
                               if
(BouncerRolesList.checkPermission(role,
httpServletRequest.getMethod(), httpServletRequest.getRequestURI()))
{
                                     return joinPoint.proceed();
                               }
                         }
                   }
             }

       } catch (final Throwable throwable) {
             log.error("Error while obtaining ACL Information: " +
throwable.getMessage());
       }
       throw new CommonException(JumiaError.GENERAL_ERROR,
"Unauthorized Access", HttpStatus.UNAUTHORIZED);
    }
}
```

Conforme podemos verificar acima, a execução do método verifica todos os parâmetros, estando à espera de um parâmetro cujo tipo seja um `HttpServletRequest`. Este tipo de objecto guarda os dados relativos à execução do pedido *HTTP*, sendo os mais importantes para este caso o JWT (para a extracção dos *roles*), o método *HTTP* e o caminho do pedido. Estes dados serão enviados ao método que valida com base numa lista, se o método inicial que foi interceptado com a anotação pode ser executado ou não:

```
public static boolean checkPermission(final String role, final String
method, final String path) {
       return !roleToOperationList.stream().filter(
                   roleToOperation ->
roleToOperation.getRole().name().equals(role) &&
```

```
!theList.stream().filter(
                                    operationToPath ->
operationToPath.getOperationName().equals(roleToOperation.getOperatio
nName())
                                                    &&
role.equals(roleToOperation.getRole().name())
                                                    && (path != null &&
path.startsWith(operationToPath.getPath()))
                                                    && (method == null ||
operationToPath.getMethod().equals(method))
                    ).collect(Collectors.toList()).isEmpty()
       ).collect(Collectors.toList()).isEmpty();
    }
```

Este método verifica, com base na junção de 3 elementos, se aquele *role* pode executar o pedido no caminho que é enviado, com o recurso à acção definida pelo verbo *HTTP*. Existem elementos a reter, para que se consiga perceber este mecanismo de autorização:

O elemento abaixo contém um prefixo para um caminho, para validar junto do caminho que faz parte do pedido *HTTP*:

```
private static final String PATH_PREFIX_FOR_ADMIN_BUSINESS_CONTRACT =
PATH_PREFIX_FOR_ADMIN + "businesscontract/";
```

Este elemento é a acção que o utilizador quer tomar:

```
private static final String VIEW_GATEWAYS = "VIEW_GATEWAYS";
```

Este elemento coloca a acção, o caminho e o verbo sobre a mesma alçada, fazendo mais fácil a identificação da permissão durante a execução do método, e faz com que seja mais fácil também a manutenção deste código, onde é necessário adicionar informação aquando da criação de controladores ou métodos com restrições de acesso.

```
theList.add(new OperationToPath(VIEW_GATEWAYS,
PATH_PREFIX_FOR_ADMIN_BUSINESS_CONTRACT + "gateways",
HttpMethod.GET.name()));
```

Existe também um método, que é utilizado durante o *login* na plataforma de administração, que devolve todas as acções possíveis para aquele *role* de utilizador:

```
public static Set<String> getActionsForRoles(final
Collection<UserRole> roles) {
       final Set<String> operations = new HashSet<>();
       roles.forEach(userRole ->
operations.addAll(roleToOperationList.stream().filter(roleToOperation
```

```
->
roleToOperation.getRole().name().equals(userRole.getRoleName())).map(R
oleToOperation::getOperationName).collect(Collectors.toSet())));
        return operations;
    }
```

Este método revelou-se necessário pois no *paybuddy* existe um método similar, que é utilizado no backoffice (AdminUI) para configurações do sistema. Devido à complexidade do sistema, negócio, e das pessoas que utilizam a plataforma de configuração, um sistema de autorização granular é um recurso valioso, garantindo que os utilizadores com acessos de configuração apenas efectuam alterações no seu âmbito, isolando assim as responsabilidades de cada um dos utilizadores e optimizando o seu dia-a-dia.

4.4. PAYBUDDY (MÓDULO DE EXTENSÃO)

Como se tem verificado até ao momento, no módulo de extensão (*paybuddy*) existe lógica de negócio cuja dimensão actual não é possível migrar no imediato, optando-se pela migração gradual. Entretanto, enquanto este trabalho se desenvolve, é necessário ter disponível a lógica de negócio do *paybuddy* para que os fluxos existentes não sejam quebrados. Portanto, foi necessário importar este módulo para o *supermario*.

O *paybuddy* contém a extensão da solução base desenvolvida pela consultora externa, o *core*. Esta extensão contém alterações no que diz respeito a configurações de moeda, configurações relativas a fluxos contabilísticos, integrações com provedores de serviços de pagamento, sistemas de envio de email e sms.

Dada a heterogeneidade dos tipos de instrumento de pagamento e dos seus respectivos fornecedores, para além de extensões de funcionalidade e configurações, foram também efectuadas alterações ao *core* por forma a suportar tipos novos, e particularidades de cada um dos diferentes métodos de pagamento. O armazenamento da mais díspar informação revelava-se necessário, por forma a efectuar com precisão auditorias, *debugging* e fornecimento de informação aos stakeholders.

Ao longo do tempo, as modificações ao código foram sido feitas neste módulo, pois era o seu propósito. No entanto, dada a dimensão que o projecto atingiu e a necessidade de escalar rapidamente, torna-se imperativa a redução de funcionalidades de uma única aplicação para várias aplicações satélite, comumente designadas por micro-serviços.

4.4.1. PROBLEMAS COM DEPENDENCIAS

Aquando da importação do projecto para o *supermario*, constatou-se que no arranque da aplicação faltavam *beans*, e nos serviços que funcionavam existia lógica de negócio em falta, lógica de negócio existente no *paybuddy*. Isto colocou de imediato um problema, pois o propósito do desenvolvimento do *supermario* era a simplificação da modernização da solução passando por uma migração do desenvolvimento de forma gradual e faseada, Ao contrário do cenário que se colocava agora que era a cópia literal da declaração dos *beans* e o respectivo código para fazer com que a aplicação tivesse a funcionalidade original em pleno funcionamento.

Verificando o que se descreveu até agora, o *paybuddy* é uma extensão ao *core*, e uma das principais maneiras de alterar o comportamento da aplicação é com o recurso ao *override* de *beans*, alterando a maneira como a aplicação é executada. Ao executar o actual *paybuddy*, para a execução desta aplicação é carregado para o contexto aplicacional todo o *core* com as respectivas configurações e propriedades, e após este carregamento é carregado também o *paybuddy*, onde as configurações por defeito geradas pelo *core* serão substituídas.

Constatamos também que, da maneira como os módulos se importam, causam um problema de dependências circulares, onde A depende de B, B depende de C e C depende de A. Isto pode revelar-se de muitas formas, como por exemplo a aplicação não arrancar ou, devido ao funcionamento interno do *Spring* no que diz respeito ao contexto aplicacional, a aplicação pode carregar alguns beans mas não fazer a substituição respectiva com a lógica de negócio adequada, fazendo com que o funcionamento seja errático e / ou inesperado.

4.4.2. PAYBUDDY-PROXY

O *paybuddy-proxy* é o nome que foi dado ao módulo que importa o módulo de extensão, no supermario. A responsabilidade deste módulo é fazer a ponte entre o *paybuddy* e o *supermario*, fazendo com que este possa ser importado correctamente:

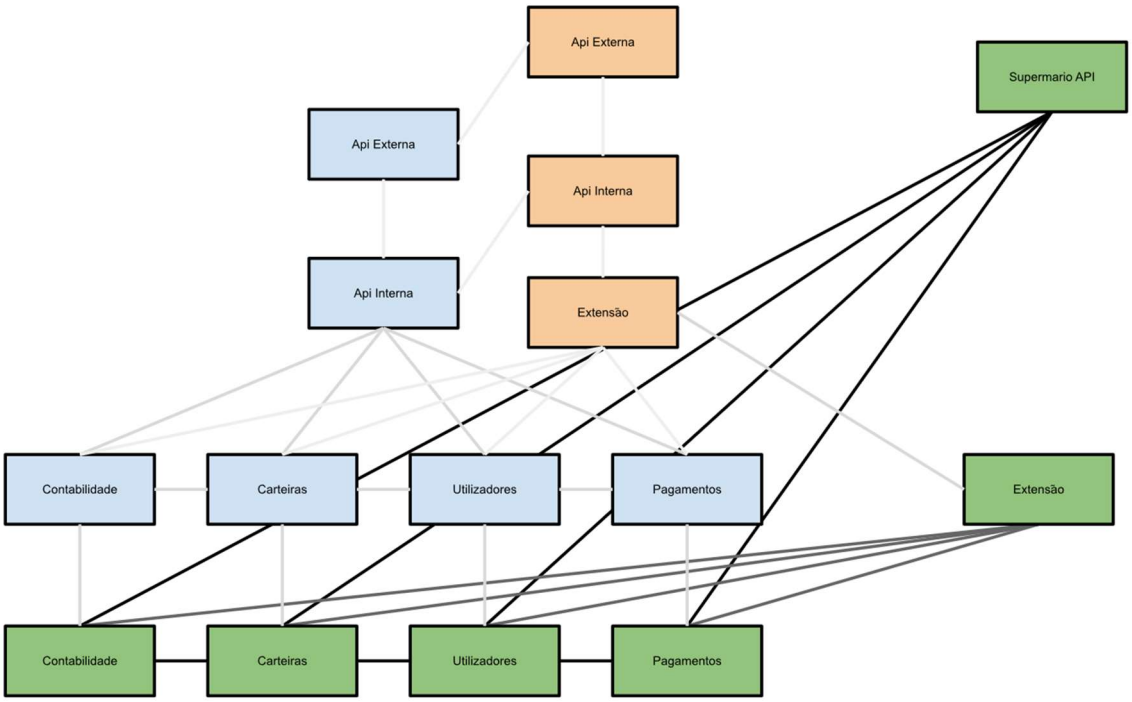


Figura 11 - Reestruturação das interligações entre os módulos.

Como podemos ver na Figura 11, a extensão a verde deixou de ter ligação à *API*. Isto deve-se ao objectivo da reformulação, que é eliminar todas as ligações externas ao módulo de ligação. Este continua activo, mas apenas em contexto aplicacional *core-paybuddy* (onde o paybuddy carrega o contexto aplicacional por forma a efectuar as funções que desempenhava até então). O *supermario* carrega o seu contexto e o contexto do core, carregando apenas o contexto do paybuddy nos módulos que são estritamente necessários, fazendo uso do seu código quando a reformulação se revela trabalhosa e/ou desnecessária naquele momento.

```
(...)
@Configuration
@ComponentScan(basePackages = {"com.jumiapay.paybuddy",
"com.trimplement.wallet.server.paybuddy"})
@ImportResource({
	"classpath*:/META-
INF/spring/com.trimplement.wallet.server.paybuddy.*.xml"
})
@PropertySource({
	"classpath:/META-
INF/config/com.trimplement.wallet.server.paybuddy.api.properties",
	"classpath:/META-
INF/config/com.trimplement.wallet.server.paybuddy.impl.properties",
	"classpath:paybuddy.properties"
})
@EnableTransactionManagement
public class PaybuddyConfig {
}
```

Acima, como podemos ver na classe de configuração existente no módulo que faz o *proxy* entre o *paybuddy* e o *supermario* (*paybuddy-proxy*), este carrega todo o contexto do paybuddy mas não importa outras configurações pois este é o seu único objectivo.

Esta classe de configuração é importada pela classe de configuração principal da aplicação que faz com que o conteúdo do *paybuddy* esteja disponível para importação no contexto aplicacional, mas os seus controladores REST estão indisponíveis.

4.4.3. UTILIZAÇÃO DE MESSAGE BROKERS (APACHE KAFKA)

Um dos desafios enfrentados com esta separação e modernização, tem a ver com os fluxos da aplicação. O caso de uso mais comum no JumiaPay é a sua utilização pelo consumidor final para o pagamento das suas compras, com recurso a um método de pagamento externo, como por exemplo o cartão de crédito. Neste caso, é comum a utilização de uma confirmação com dois factores (*two-factor authentication*, ou *2FA*)

Durante o pagamento, que é o acto de introduzir dinheiro no ecossistema do JumiaPay, é enviado pelo provedor de serviços de pagamento uma resposta ao acto de pagamento, que pode ser favorável ou não. Esta resposta é necessária para dar continuidade ao fluxo do pagamento e consequentemente da compra.

Existe, paralelamente ao *paybuddy*, um módulo de processamento assíncrono, o *jobs*. Este módulo contém várias tarefas controladas pelo *Quartz* sendo uma delas o processamento das instruções de pagamento associadas à *wallet* do utilizador. Esta tarefa é responsável por verificar numa tabela de base de dados se existe instrução de pagamento para uma determinada operação (neste caso compra) e em caso positivo, dar a compra como concluída.

No paybuddy, a acção de colocar na tabela a informação relativa ao pagamento para dar seguimento à compra era feita com um serviço no próprio *paybuddy,* já que devido à estrutura do projecto tudo era visível dentro deste módulo. Neste serviço, era colocada uma entrada na base de dados com a informação necessária. No entanto, dada a modernização do sistema em questão, esta chamada de serviço levantou imediatamente um problema aquando da reestruturação dos fluxos da compra devido à complexidade adicional que não tinha sido prevista relativamente à localização do módulo do *paybuddy-proxy*, estando este apenas visível do módulo de API, e esta operação de pagamento estar no módulo *payment*. Como se pode recordar do módulo acima, o módulo de *API* tem visibilidade para o módulo de *payment*, mas o contrário não é possível caso contrário teríamos um problema de dependências circulares.

A abordagem mais rápida e eficiente demonstrou-se na utilização de um *message broker*, neste caso o *Apache Kafka*. Dado que a base de dados é partilhada

pelas aplicações todas (*jobs*, *supermario*, *paybuddy*), a utilização do *kafka* permitiu-nos mitigar este problema. Já que a compra está, naquele momento, sob a responsabilidade do módulo *payment*, com o recurso a esta tecnologia podemos enviar uma mensagem com um objecto via *kafka* para o módulo *API*, conseguindo assim dar continuidade ao pagamento sem ser necessário mover no imediato esta lógica para o *supermario*.

A Figura 12, abaixo, descreve o fluxo que originou esta solução, que evita o dispêndio de tempo. O caso de uso tem início na compra, segue pela instrução de pagamento, seguida da comunicação com serviço externo, publicação e consumo da mensagem com o recurso ao *message broker*. O consumo da mensagem dá origem à criação da entrada na base de dados, e o job encarrega-se da leitura desses mesmos dados escritos, originando assim o prosseguimento do pagamento na máquina de estados.

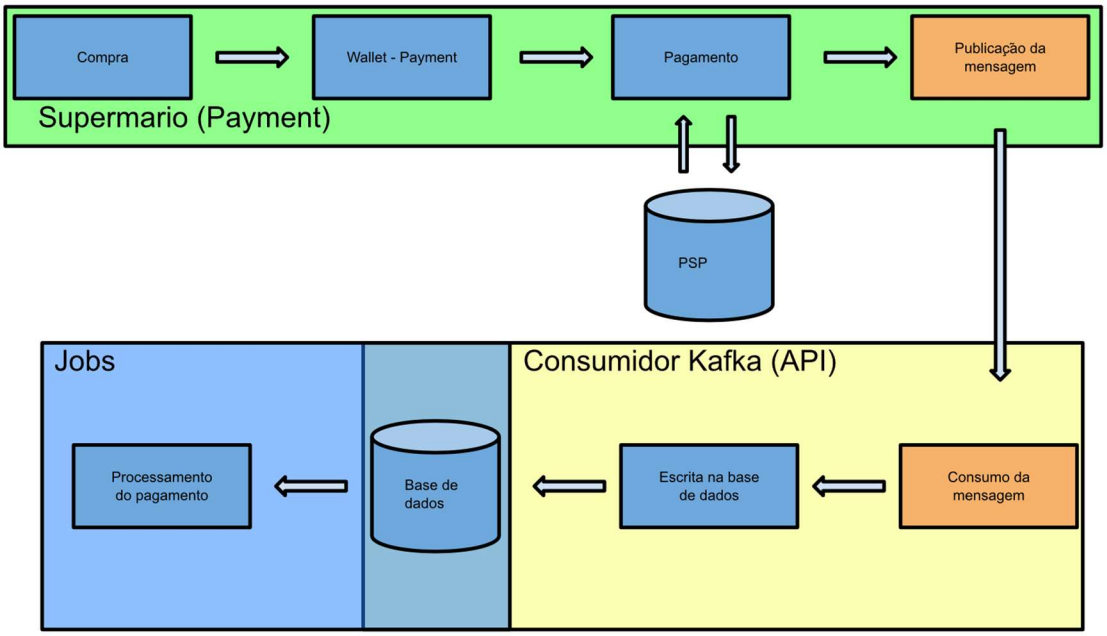


Figura 12 – Fluxo para processamento do pagamento.

4.5. INTEGRAÇÃO COM GATEWAYS

4.5.1. SISTEMA EXISTENTE

Para efectuar uma integração com um fornecedor de serviços de pagamento, no *paybuddy*, esta era feita juntamente com o restante código, separando por packages cada uma das integrações distintas. Para além destas integrações, era também necessário, devido à natureza de cada um dos fornecedores, efectuar alterações no fluxo das compras, pelas diferentes necessidades que cada um dos métodos de pagamento tinha no que diz respeito à validação, quer do instrumento de pagamento, quer do fluxo da compra. Um exemplo de validação durante a inserção do instrumento de pagamento seria a introdução da data de nascimento do titular aquando da associação da conta bancária com o JumiaPay. Por outro lado, um exemplo de validação aquando do pagamento é a introdução de um *OTP (One Time Password)*, que pode ser enviado por SMS, ou através da utilização de um *hard token*. Com estes mecanismos de segurança fornecidos pelos fornecedores de serviços de pagamento, torna-se mais difícil para um utilizador mal-intencionado obter acesso a instrumentos de pagamento.

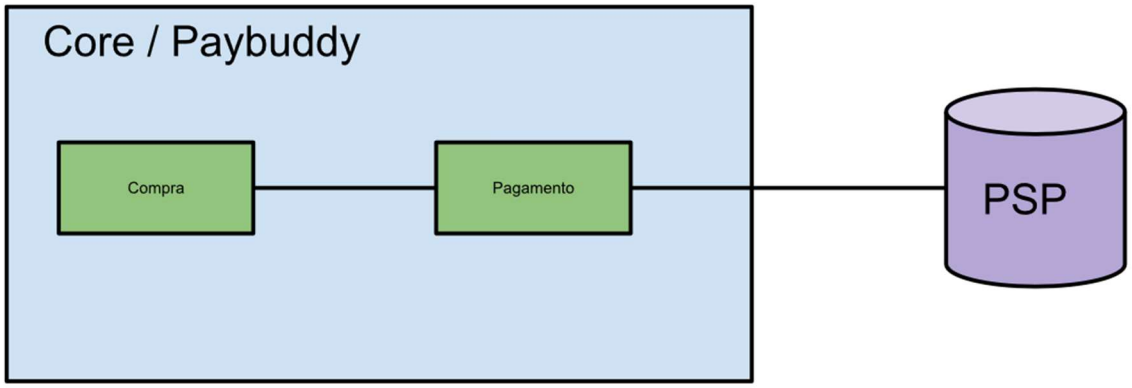


Figura 13 - Relação simplificada entre core, paybuddy, e provedor de serviços de pagamento aquando da comunicação efectuada pelo paybuddy

Na Figura 13, vemos descrito o fluxo que foi mencionado. É da responsabilidade do módulo de pagamento a integração com o provedor de serviços de pagamento, fazendo que, com o crescimento do JumiaPay, se torne complexo manter o código que é responsável por estas mesmas integrações.

Ao integrar com vários provedores de serviços de pagamento em simultâneo, aconteceu, inclusive, existirem programadores que tiveram a mesma necessidade de alterar uma particularidade no código. Mas, devido às restrições de cada um dos provedores de serviço de pagamento, efectuaram a alteração dessa mesma particularidade de forma distinta, tendo isto causado atrasos, pois foi necessário encontrar um ponto comum onde estas integrações conseguissem coexistir.

4.5.2. MIGRAÇÃO PARA KOOPA

Tendo em conta que as integrações com os *PSP* (Provedores de Serviços de Pagamentos) não fazem parte do código base do *core*, e o objectivo final destas integrações é efectuar pagamentos, a abordagem tomada foi a extracção gradual destas integrações para micro-serviços, sendo estes responsáveis por apenas e exclusivamente a comunicação entre o *supermario* e os *PSP*. A estas integrações foi dado o nome de *Koopa*.

Foi necessário definir então um contrato, que seja conhecido quer pelo *supermario*, quer por cada um dos *koopa*, de forma a existir uma base de comunicação comum, facilitando assim a integração de novos provedores de serviço de pagamento.

Foi efectuado um levantamento de todas as integrações existentes, de todos os tipos distintos de instrumentos de pagamento. Este levantamento foi necessário para que o contrato definido seja suficiente para integrar os actuais fornecedores, estando este contrato aberto a extensão apenas para garantir sempre a retro-compatibilidade.

Do lado do *supermario*, mais especificamente no que diz respeito ao *core* e *paybuddy*, este tipo de delegação de responsabilidade não estava previsto. A solução de base prevê que as integrações se encontrem fortemente ligadas com o restante do código, sendo necessário para isto a criação de uma estrutura vazia, povoável com a informação do gateway.

Como o propósito dos *koopa* é o de retirar ao *supermario* toda e qualquer responsabilidade relativa à comunicação com os *gateways*, o *supermario* também não sabe se existem koopa, ou quantos existem, naquele momento. Para isso, os koopa enviam para um tópico de *kafka* a sua informação, para depois ser consumida do lado do *supermario* e povoar uma tabela com a informação necessária, que é o endereço *IP,* configurações adicionais e os verbos suportados pelo koopa (os verbos têm a ver neste caso com a acção. Pré-autorizar, autorizar e, reembolsar são exemplos de verbos que dizem respeito ao âmbito dos pagamentos).

Após o envio da mensagem com os dados, a tabela é povoada e, no momento de pagar, existe um motor que decide, com base nas selecções do utilizador e com base também nas opções disponíveis naquele momento, qual o *gateway* com o qual deve comunicar. Após a determinação do gateway, o objecto é instanciado, e é utilizado para comunicar com o koopa. Este objecto fará utilização dos endereços e configurações enviados no início, configurações que podem conter por exemplo chaves de acesso à *API* do fornecedor.

A resposta que for obtida por parte do koopa, vinda do provedor de serviços de pagamento será transformada antes de ser devolvida ao *supermario*. Relembra-se que esta resposta tem de ser transformada para respeitar um contrato genérico, sendo este contrato reconhecível por qualquer tipo de método de pagamento. Após a recepção desta mensagem, será enviada para o *supermario* e processada em conformidade, de acordo com o tipo.

Existe também a possibilidade dos *gateway* responderem apenas de forma assíncrona, em determinados casos, podendo chegar às 48h. Nestes casos, é enviado pelo gateway um pedido *REST* que chegará ao *koopa* e, previamente ao envio da mensagem ao *supermario*, será transformado para respeitar o contrato mencionado anteriormente, fazendo avançar o pagamento na máquina de estados do supermario. A Figura 14, abaixo, apresenta um diagrama que demonstra o fluxo da compra.

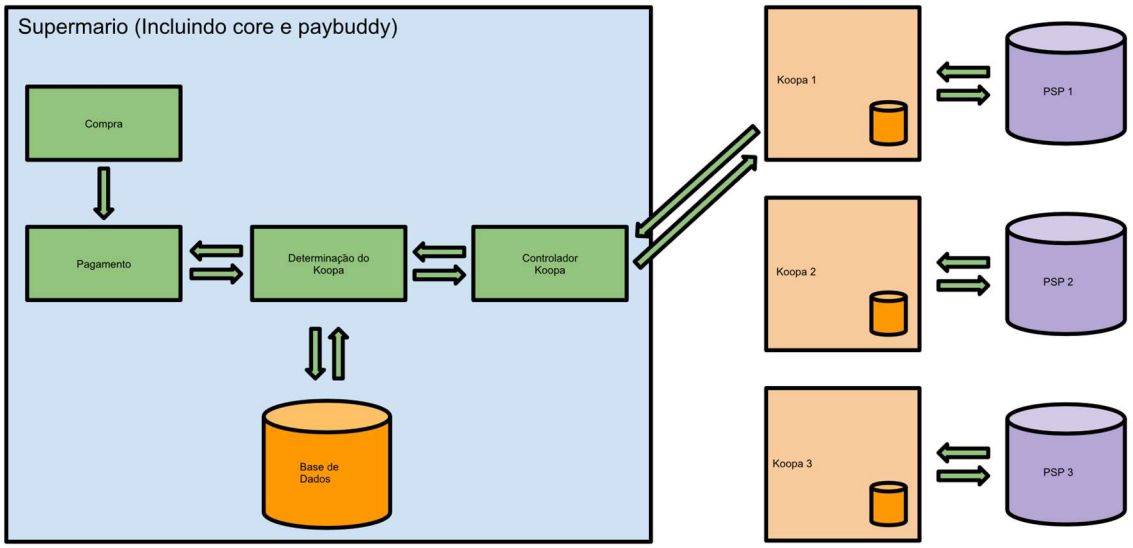


Figura 14 - Diagrama que demonstra o fluxo da compra.

Como se verifica na figura, resumidamente, a compra é iniciada, seguida do pedido de pagamento, determinação do *koopa* (que resultou no *Koopa 1*), comunicação ao *gateway*, devolução de resposta e conclusão do pagamento. Assume-se, que nesta fase todos os *koopa* já enviaram a sua informação para o tópico de *kafka*, e estão operacionais.

4.6. CONCLUSÕES DO DESENVOLVIMENTO

O desenvolvimento desta plataforma e a modernização é sem dúvida um processo que tem muito pela frente. A migração para um sistema de micro-serviços é uma tarefa que requer um planeamento cuidado, conhecimento alargado do negócio e da plataforma existente. Actualmente, a equipa encarregue do desenvolvimento deste

projecto está em crescimento, com vista precisamente à melhoria do serviço que presta aos seus clientes.

Uma das barreiras que é necessário ultrapassar do ponto de vista técnico prende-se com a formatação dos recursos existentes para a arquitectura monolítica. Tem vindo a ser feito um trabalho com vista a educar no sentido dos micro-serviços, ao isolamento de responsabilidades, e às boas práticas de desenvolvimento. Um caso claro desta melhoria no *mindset* é o desenvolvimento cada vez mais acelerado nos projectos com esta arquitectura, o que revela um conhecimento maior com cada projecto concluído.

Outra das limitações que é necessário reconhecer, e saber coexistir com ela, é a necessidade de coexistir com a realização do *roadmap*. A empresa não pode parar para efectuar estas melhorias, pois é algo que não é facilmente justificável junto dos investidores, a necessidade de parar por largos períodos de tempo para efectuar melhorias que nada trazem de visível ao produto. A corrida com os investidores deve ter o rumo traçado em entregar funcionalidade, e compete à equipa técnica saber negociar junto da equipa de produto que determinada melhoria num ponto A ou B trará melhorias noutros pontos. Compete também à equipa técnica garantir que os compromissos que são negociados não afectarão negativamente o produto ao ponto de ser necessário efectivamente parar para refazer determinada funcionalidade que foi atalhada anteriormente.

A migração para micro-serviços tem de ser algo que junto da equipa de produto seja identificável, de um ponto de vista de responsabilidades. Cada um destes micro-serviços tem de ser responsável pelo seu domínio, e neste aspecto a equipa responsável pelo produto pode ajudar a identificar esta questão. A identificação clara dos casos de uso por parte da equipa de produto, associado a uma boa análise técnica facilita o trabalho de desenvolvimento, garantindo que o foco da equipa será bem utilizado pois as tarefas foram bem identificadas, optimizando o custo geral de desenvolvimento.

5. DISCUSSÃO DOS RESULTADOS

As melhorias obtidas com este trabalho permitiram optimizar o tempo de desenvolvimento de novas integrações com *gateways*, que é a principal área de expansão do *JumiaPay*. Este trabalho permite inclusive a delegação destes desenvolvimentos para equipas que estejam geograficamente distantes, tendo apenas de seguir a documentação para integrar com o *supermario*, sendo esta documentação bastante mais reduzida do que a curva de aprendizagem necessária para integrar um *gateway* no *paybuddy*.

Com esta melhoria consegue-se também reduzir a dimensão do projecto, pois durante a migração das funcionalidades do *paybuddy* para o *supermario* conseguimos reduzir o número de linhas do código fonte.

A redução deveu-se a duas razões, podendo a respectiva funcionalidade já não estar a ser utilizada, tendo sido substituída por uma nova, ou simplesmente já não fazer sentido no contexto actual. Podia também ser código morto, como por exemplo configurações de moeda que não fazem parte do âmbito actual.

Esta redução no código fonte da aplicação central revela-se crucial num projecto com a complexidade deste. Associado à redução da quantidade de código está o tempo de arranque. Actualmente a quantidade de componentes carregados pela aplicação é de dimensão tal, que numa máquina de desenvolvimento é habitual a aplicação demorar mais de dois minutos a arrancar. Em casos de downtime, e com o volume de negócio que é processado pelo JumiaPay, dois minutos sem aplicação significam um grande volume de negócio perdido.

A utilização de um *message broker* também veio ajudar a reduzir a complexidade, e aumentar a escalabilidade. Quando as aplicações estão desenhadas para serem de responsabilidade única, e contém pouca lógica de negócio, a utilização de consumidores Kafka faz com que, quando seja necessário escalar para se efectuar maior processamento de dados, isto seja feito com mais facilidade. Em cada um dos message brokers existem configurações distintas que podem ou não ser melhores para o produto em questão. A utilização de um message broker simplifica a implementação de projectos complexos, também fazendo a ligação entre componentes que de outra forma não poderiam ser ligados, como foi o caso da ligação do componente *payment* com o *API*.

6. TRABALHO FUTURO

Durante o decorrer do desenvolvimento, são inúmeros os pontos que se revelam carentes de melhoria. A identificação de outras áreas da plataforma que sejam passíveis de uma migração para micro-serviços torna-se um dos objectivos a atingir no longo prazo, fazendo com que a modularidade seja de tal dimensão sendo assim possível criar várias instâncias de um determinado módulo, como por exemplo para ajudar num respectivo intervalo de tempo com a criação de contas de utilizador, que seja possível escalar apenas o módulo de utilizador.

A utilização de *GraphQL* [72] para consultar dados, ao invés de controladores *REST*, também se poderia revelar frutífera. Existem vários casos de serviços nas API que servem para devolver dados relativos ao mesmo domínio, mas com diferentes níveis de detalhe. A utilização de uma linguagem de consulta do lado dos componentes acima descritos facilitava por um lado a manutenção dos controladores, pois estes seriam limitados a um único ponto de entrada, e por outro lado facilitavam também a tarefa a quem consumia os dados, sendo apenas necessário conhecer os campos existentes para os solicitar na consulta.

A divisão da base de dados também se pode considerar um ponto a melhorar. Actualmente, a mesma base de dados relacional é partilhada por todas as aplicações (*Supermario*, *Paybuddy*, e *Jobs)*. Isto significa que em caso de falha da base de dados, existe uma quebra de serviço. Existem evidentemente mecanismos para recuperação e redundância da base de dados actualmente em curso, mas o ideal seria neste caso a divisão da base de dados também por cada um dos módulos. A utilização de uma base de dados não relacional também poderia melhorar a velocidade de *I/O*.

O recurso a programação assíncrona também se pode encaixar nalguns casos. A utilização de *Future* em Java permite-nos efectuar processamento paralelo [73]. Com a utilização desta interface, é possível a execução de tarefas de forma assíncrona, obtendo ganhos de performance. No entanto, isto requer uma mudança no mindset do desenvolvimento pois nem sempre é desejável ter processamento paralelo [74].

7. CONCLUSÕES FINAIS

A modernização de uma aplicação para a utilização de micro-serviços na sua arquitectura é algo para o qual não existe uma única forma para atingir o resultado desejado. Todos os factores em jogo têm de ser considerados, desde a tecnologia utilizada para a base de dados, as tecnologias nas quais o software foi desenvolvido, a infraestrutura, e a visão de produto. Todos estes componentes contribuem para o que é um bom ou mau produto de software.

É importante saber determinar a dimensão de uma tarefa, efectuar estimativas correctas para saber gerir expectativas, conhecer a velocidade da equipa de desenvolvimento, saber o que a equipa de desenvolvimento é capaz de entregar e saber aquilo que essa mesma equipa espera do produto. A equipa de desenvolvimento, por norma é sempre a que tem o contacto mais próximo junto da aplicação, pois é a que lida com ela no seu dia-a-dia. Se houver algo de errado com a aplicação, este actor será o primeiro a entrar em contacto para conseguir resolver essa mesma questão.

A palavra chave é a comunicação. A equipa de produto tem de ser capaz de fazer passar a sua visão para a equipa de desenvolvimento, e esta tem de ter a capacidade de perceber essa visão, efectuar uma análise adequada, levantar os requisitos e discutir as preocupações que daí possam advir. Não compete à equipa de produto reconhecer as limitações técnicas do software, mas compete-lhe percebê-las e aceitá-las. De igual modo, compete à equipa de desenvolvimento fazer o seu melhor para conseguir integrar esta mesma visão, sem recurso a atalhos ou a criação de dívida técnica. A criação de dívida técnica é habitualmente identificada pela utilização da frase “Depois arranja-se”, ou “Depois vemos”. No que diz respeito aos projectos de software, e em particular este projecto a criação da dívida técnica foi algo que se verificou ao longo do tempo, por fruto de uma série de factores, entre os quais a inexperiência da equipa técnica, a pressão de produto pela entrega de funcionalidades, a alteração de âmbito frequente e mesmo os poucos recursos disponíveis, fizeram com que o projecto atingisse os problemas referidos.

A identificação de factores problemáticos, como dependências ou pontos de falha, quando efectuada antecipadamente é um factor de sucesso para alcançar o objectivo pretendido. Quanto mais cedo os factores problemáticos forem identificados, mais depressa serão resolvidos. Um problema que seja reduzido numa fase inicial, quando descurado pode causar um problema de dimensões maiores numa fase mais avançada, podendo inclusive comprometer a entrega do produto.

Criar diagramas de fluxo com toda a comunicação que é efectuada dentro da nossa aplicação monolítica é uma ferramenta importante, quando chega a fase da divisão por serviços. A identificação das áreas comuns torna-se mais fácil quando existe recurso a um suporte visual.

Um levantamento correcto dos casos de uso permite também conhecer o propósito de uma tarefa ou funcionalidade com mais precisão. Determinar quais as tecnologias a aplicar num determinado caso de uso quando nos referimos a algo que foi extraído para um micro-serviço permite-nos optimizar os recursos para cada caso. Por vezes, incorre-se no erro de adoptar uma tecnologia complexa em demasia para resolver um problema pequeno, comparando isto a "matar uma mosca com uma bazuca". A análise das tecnologias a implementar é um dos passos mais importantes nesta abordagem, garantindo que a utilização de recursos é optimizada, tendo este factor particular importância nos dias de hoje, com a mudança de paradigma no que diz respeito à contratação de servidores para alojar os recursos que precisamos. Existem empresas que alugam servidores à hora e apenas quando necessário [73], permitindo isto uma redução de custos pois apenas escalamos as nossas aplicações quando necessário.

Não é de todo o propósito que se tenta atingir com este desenvolvimento, renegar a criação de aplicações monolíticas. Faz parte do desenvolvimento natural de uma aplicação, e várias empresas passaram pelo mesmo processo. O exemplo da Uber [74], REWE [75] ou Netflix [76] são exemplos de grandes empresas que começaram com aplicações monolíticas, atingiram a sua prova de conceito, e quando se revelou necessário escalar a aplicação, a divisão em microserviços foi a abordagem a seguir.

REFERÊNCIAS

[1] Jumia Group | Expand your Horizons, [Online]. Available: https://group.jumia.com/. [Acedido em 8 Junho 2019].

[2] Jumia Nigeria, [Online]. Available: https://www.jumia.com.ng/. [Acedido em 8 Junho 2019].

[3] Jumia One, [Online]. Available: https://one.jumia.com.ng/. [Acedido em 8 Junho 2019].

[4] Jumia Travel, [Online]. Available: https://travel.jumia.com/. [Acedido em 8 Junho 2019].

[5] Jumia Food, [Online]. Available: https://food.jumia.com.ng/. [Acedido em 8 Junho 2019].

[6] Killers Of Jumia Delivery Man Murdered In Port Harcourt, 28 Março 2017. [Online]. Available: https://naijagists.com/photos-killers-jumia-delivery-man-murdered-port-harcourt-rivers-state-saturday-arrested/. [Acedido em 8 Junho 2019].

[7] The Future Of E-commerce In Africa: A Mere Illusion? - Africa.com, 15 Abril 2019. [Online]. Available: https://www.africa.com/the-future-of-e-commerce-in-africa/. [Acedido em 8 Junho 2019].

[8] Africa - Practical Ecommerce, 13 Junho 2018. [Online]. Available: https://www.practicalecommerce.com/africa-emerging-ecommerce-market-many-challenges. [Acedido em 8 Junho 2019].

[9] J. Bowker, 10 Abril 2019. [Online]. Available: https://www.bloomberg.com/news/articles/2019-04-10/africa-s-amazon-set-for-new-york-ipo-as-online-retail-takes-off. [Acedido em 8 Junho 2019].

[10] Jumia – Africa's Alibaba and its IPO roller coaster | Daily Fintech, 7 Junho 2019. [Online]. Available: https://dailyfintech.com/2019/06/07/jumia-africas-amazon-and-its-ipo-roller-coaster/. [Acedido em 8 Junho 2019].

[11] What Is A Startup - Startup Commons, [Online]. Available: https://www.startupcommons.org/what-is-a-startup.html. [Acedido em 30 Junho 2019].

[12] Steve Blank What's A Startup? First Principles.., 25 Janeiro 2010. [Online]. Available: https://steveblank.com/2010/01/25/whats-a-startup-first-principles/. [Acedido em 30 Junho 2019].

[13] STANDALONE | meaning in the Cambridge English Dictionary, [Online]. Available: https://dictionary.cambridge.org/dictionary/english/standalone. [Acedido em 30 Junho 2019].

[14] Roadmap Basics: What Is a Roadmap? - ProductPlan., [Online]. Available: https://www.productplan.com/roadmap-basics/. [Acedido em 2019 Junho 2019].

[15] What is a technology or IT roadmap? | Aha!, [Online]. Available: https://www.aha.io/roadmapping/guide/product-roadmap/what-is-a-technology-or-it-roadmap. [Acedido em 30 Junho 2019].

[16] Digital Money, Liquidity, and Monetary Policy, [Online]. Available: https://ojphi.org/ojs/index.php/fm/article/view/1512/1427. [Acedido em 30 Junho 2019].

[17] Measuring API Usability | Dr Dobb's, 1 May 2004. [Online]. Available: http://www.drdobbs.com/windows/measuring-api-usability/184405654. [Acedido em 30 Junho 2019].

[18] R. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, Section 4," Junho 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231#section-4. [Acedido em 25 07 2019].

[19] Web Services Architecture - World Wide Web Consortium, 11 Fevereiro 2014. [Online]. Available: https://www.w3.org/TR/ws-arch/. [Acedido em 30 Junho 2019].

[20] D. Riehle, "Framework Design: A Role Modeling Approach," [Online]. Available: http://www.riehle.org/computer-science/research/dissertation/index.html. [Acedido em 30 Junho 2019].

[21] JCP.org, "JSR 330," [Online]. Available: https://www.jcp.org/en/jsr/detail?id=330. [Acedido em 9 Junho 2019].

[22] The DCI Architecture: A New Vision of Object-Oriented, 20 Março 2009. [Online]. Available: https://www.artima.com/articles/dci_vision.html. [Acedido em 30 Junho 2019].

[23] Pivotal, "Index of /spring/docs.," [Online]. Available: https://docs.spring.io/spring/docs/. [Acedido em 30 Junho 2019].

[24] Defining Legacy Code - DZone DevOps, 15 Maio 2018. [Online]. Available: https://dzone.com/articles/defining-legacy-code. [Acedido em 30 Junho 2019].

[25] Legacy | Definition of Legacy by Merriam-Webster, [Online]. Available: https://www.merriam-webster.com/dictionary/legacy. [Acedido em 30 Junho 2019].

[26] Microsoft, "Windows XP End of Support," Microsoft, [Online]. Available: https://www.microsoft.com/en-au/windowsforbusiness/end-of-xp-support. [Acedido em 30 Junho 2019].

[27] Marinha dos Estados Unidos da América, "File:US Navy 110129-N-7676W-152 Culinary Specialist 3rd ...," Marinha dos Estados Unidos da América, 10 Setembro 2018. [Online]. Available: https://commons.wikimedia.org/wiki/File:US_Navy_110129-N-7676W-152_Culinary_Specialist_3rd_Class_John_Smith_uses_the_existing_DOS-based_food_service_management_system_aboard_the_aircraft.jpg. [Acedido em 30 Junho 2019].

[28] "What is an App Server? - The Server Side," [Online]. Available: https://www.theserverside.com/news/1363671/What-is-an-App-Server. [Acedido em 30 Junho 2019].

[29] JCP.org, "JSR 914," JCP.org, [Online]. Available: https://jcp.org/en/jsr/detail?id=914. [Acedido em 30 Junho 2019].

[30] Apache Foundation, "Apache Tomcat® - Welcome!," Apache Foundation, [Online]. Available: http://tomcat.apache.org/. [Acedido em 30 Junho 2019].

[31] Oracle Corporation, “Glassfish Server,” Oracle Corporation, [Online]. Available: https://www.oracle.com/technetwork/middleware/glassfish/overview/index.html. [Acedido em 30 Junho 2019].

[32] “Wildfly,” [Online]. Available: https://wildfly.org/. [Acedido em 30 Junho 2019].

[33] “Java Singleton Pattern Explained - HowToDoInJava,” [Online]. Available: https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/. [Acedido em 1 Julho 2019].

[34] steveyegge2, “singleton-considered-stupid,” Google Sites, [Online]. Available: https://sites.google.com/site/steveyegge2/singleton-considered-stupid. [Acedido em 1 Julho 2019].

[35] “Clean Code Talks - Global State and ...,” Google Testing Blog, 21 Novembro 2008. [Online]. Available: https://testing.googleblog.com/2008/11/clean-code-talks-global-state-and.html. [Acedido em 1 Julho 2019].

[36] Oracle Corporation, “Enterprise JavaBeans Technology,” [Online]. Available: https://www.oracle.com/technetwork/java/index-jsp-140203.html. [Acedido em 30 Junho 2019].

[37] “A Singleton Session Bean Example: counter - The Java EE 6 Tutorial,” [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/gipvi.html. [Acedido em 30 Junho 2019].

[38] Spring Framework Guru, “Singleton Design Pattern in Java,” [Online]. Available: https://springframework.guru/gang-of-four-design-patterns/singleton-design-pattern/. [Acedido em 30 Junho 2019].

[39] Oracle Corporation, “BeanManager (Java(TM) EE 7 Specification APIs),” [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/enterprise/inject/spi/BeanManager.html. [Acedido em 1 Julho 2019].

[40] JWT.io, “JWT.io,” [Online]. Available: https://jwt.io/. [Acedido em 1 Julho 2019].

[41] JWT.io, “JSON Web Token Introduction,” [Online]. Available: https://jwt.io/introduction/. [Acedido em 1 Julho 2019].

[42] W3.org, “Extensible Markup Language (XML) 1.0,” 26 Novembro 2008. [Online]. Available: https://www.w3.org/TR/xml. [Acedido em 1 Julho 2019].

[43] Baeldung, “An Introduction to CDI (Contexts and Dependency Injection) in Java,” Novembro 2018. [Online]. Available: https://www.baeldung.com/java-ee-cdi. [Acedido em 6 Julho 2019].

[44] Oracle Corporation, “Overview of CDI - The Java EE 6 Tutorial,” [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/giwhl.html. [Acedido em 1 Julho 2019].

[45] JCP.org, “JSR 299,” [Online]. Available: https://jcp.org/en/jsr/detail?id=299. [Acedido em 1 Julho 2019].

[46] “Weld: Home - CDI,” [Online]. Available: https://weld.cdi-spec.org/. [Acedido em 1 Julho 2019].

[47] Google, “Google Guice - Github,” [Online]. Available: https://github.com/google/guice. [Acedido em 1 Julho 2019].

[48] Investopedia, “Return on Investment (ROI),” 22 Fevereiro 2019. [Online]. Available: https://www.investopedia.com/terms/r/returnoninvestment.asp. [Acedido em 1 Julho 2019].

[49] T. Preston-Werner, “SemVer,” [Online]. Available: https://semver.org/. [Acedido em 25 Julho 2019].

[50] Hibernate, “What is Object/Relational Mapping?,” [Online]. Available: https://hibernate.org/orm/what-is-an-orm/. [Acedido em 1 Julho 2019].

[51] JCP.org, “JSR 220,” [Online]. Available: https://jcp.org/aboutJava/communityprocess/final/jsr220/index.html. [Acedido em 10 Junho 2019].

[52] Pivotal, “CrudRepository (Spring Data Core 2.1.9.RELEASE API),” [Online]. Available: https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html. [Acedido em 4 Julho 2019].

[53] Oracle Corporation, “History of SQL - Oracle Docs,” [Online]. Available: https://docs.oracle.com/cd/B13789_01/server.101/b10759/intro001.htm. [Acedido em 4 Julho 2019].

[54] IETF, “RFC 2616 - Hypertext Transfer Protocol,” [Online]. Available: https://tools.ietf.org/html/rfc2616. [Acedido em 4 Julho 2019].

[55] “Proxy | Definition of Proxy by Merriam-Webster,” [Online]. Available: https://www.merriam-webster.com/dictionary/proxy. [Acedido em 4 Julho 2019].

[56] J. . Davis, “Two Factor Auth (2FA),” , . [Online]. Available: https://twofactorauth.org/. [Acedido em 27 11 2019].

[57] “Quartz Scheduler,” [Online]. Available: http://www.quartz-scheduler.org/. [Acedido em 4 Julho 2019].

[58] K. G. Paterson e D. . Stebila, “One-time-password-authenticated key exchange,” , 2010. [Online]. Available: https://eprint.iacr.org/2009/430.pdf. [Acedido em 27 11 2019].

[59] R. . Radhakrishnan, C. A. Frick, R. . Marian, A. O. Barbir e R. P. Badhwar, “Method and apparatus for token-based tamper detection,” , 2011. [Online]. Available: https://patents.google.com/patent/us9069943b2/en. [Acedido em 27 11 2019].

[60] “Apache Kafka Quick Intro,” [Online]. Available: https://kafka.apache.org/intro. [Acedido em 25 Julho 2019].

[61] IETF, “RFC 760 - DoD standard Internet Protocol - IETF Tools,” [Online]. Available: https://tools.ietf.org/html/rfc760. [Acedido em 4 Julho 2019].

[62] HBS Working, “Surfacing Your Underground Organization,” [Online]. Available: https://hbswk.hbs.edu/archive/surfacing-your-underground-organization. [Acedido em 4 Julho 2019].

[63] “GraphQL,” [Online]. Available: https://graphql.org/. [Acedido em 4 Julho 2019].

[64] “Why GraphQL? - Open GraphQL,” Medium, [Online]. Available: https://medium.com/open-graphql/graphql-1-140fab436942. [Acedido em 4 Julho 2019].

[65] Trimplement GmbH, “CoreWallet – the high-performance emoney ...,” [Online]. Available: https://trimplement.com/corewallet. [Acedido em 8 Junho 2019].

[66] M. Fowler, “MicroservicePremium,” [Online]. Available: https://martinfowler.com/bliki/MicroservicePremium.html. [Acedido em 8 Junho 2019].

[67] “E.W.Dijkstra Archive: The Humble Programmer (EWD 340),” [Online]. Available: https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html. [Acedido em 8 Junho 2019].

[68] “Spring @Component, @Service, @Repository, @Controller Difference ....,” 1 Agosto 2014. [Online]. Available: https://javapapers.com/spring/spring-component-service-repository-controller-difference/. [Acedido em 11 Junho 2019].

[69] “Hibernate Transaction Management Example - Javatpoint,” [Online]. Available: https://www.javatpoint.com/hibernate-transaction-management-example. [Acedido em 11 Junho 2019].

[70] T. D., 01 08 2016. [Online]. Available: https://www.cryptomathic.com/news-events/blog/digital-authentication-the-basics. [Acedido em 10 03 2020].

[71] B. Fraser, “IETF,” 09 1997. [Online]. Available: https://tools.ietf.org/html/rfc2196#page-29. [Acedido em 10 03 2020].

[72] “GraphQL,” [Online]. Available: https://graphql.github.io/graphql-spec/draft/. [Acedido em 23 Junho 2019].

[73] Amazon, “Preço Amazon EC2 - Quanto custa Amazon EC2? - AWS,” [Online]. Available: https://aws.amazon.com/pt/ec2/pricing/. [Acedido em 24 Junho 2019].

[74] Uber, “Service-Oriented Architecture: Scaling the Uber Engineering ....,” 8 Setembro 2015. [Online]. Available: https://eng.uber.com/soa/. [Acedido em 24 Junho 2019].

[75] “Migrating a Retail Monolith to Microservices: Sebastian Gauder ... - InfoQ,” 7 Abril 2019. [Online]. Available: https://www.infoq.com/news/2019/04/monolith-microservices-migration/. [Acedido em 24 Junho 2019].

[76] “Why You Can't Talk About Microservices Without ... - SmartBear,” [Online]. Available: https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/. [Acedido em 24 Junho 2019].

[77] “Apache Kafka Quick Intro,” [Online]. Available: https://kafka.apache.org/intro. [Acedido em 25 Julho 2019].

[78] D. . Florencio e C. . Herley, “One-time password access to password-protected accounts,” , 2007. [Online]. Available: https://patents.google.com/patent/us20080276098a1/en. [Acedido em 27 11 2019].