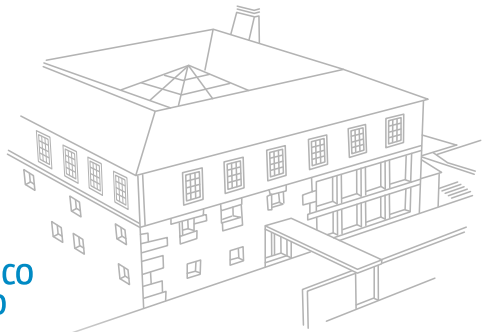

ESTGF | POLITÉCNICO DO PORTO

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Stellar - Uma Web Framework Baseada em Ações

DESIGNAÇÃO DO MESTRADO: Mestrado em Engenharia Informática

AUTOR: Gil Manuel Carvalho Mendes

ORIENTADOR(ES): Doutor Ricardo Jorge Santos

ANO: 15/16

Agradecimentos

A realização deste projeto de mestrado contou com importantes apoios e incentivos que sem os quais todo este longo caminho teria sido mais difícil.

Ao Professor Doutor Ricardo Santos, pela sua orientação, apoio, disponibilidade, pela sua visão na estrutura deste documento e pela revisão do mesmo.

Um grande obrigado ao meu amigo António Magalhães, pela sua prontidão e disponibilidade em ajudar e apoiar nos momentos mais complicados assim como pelo seu fantástico favor em rever toda a documentação.

Aos meus amigos e colegas, José Silva, Vasco Carvalho, entre outros que não menciono o nome, obrigado por terem estado ao meu lado durante esta fase, pelo companheirismo, força e apoio.

Por último, tendo consciência que sozinho nada disto teria sido possível dirijo um agradecimento especial à minha família (em particular à minha mãe), por serem modelos de coragem, pelo apoio incondicional, incentivo, amizade, paciência e ajuda na superação dos obstáculos surgidos ao longo desta longa caminhada. A eles dedico este trabalho!

Abstract

Stellar is a web framework that allows the separation between the visual and logic components of a web application. The system consists of a highly flexible and extensible engine that has a set of basic functions that will be used by developers to create modules. The modules are a set of features created by developers and grouped by area that will be loaded by the engine to provide those features to the clients. Stellar supports different protocols (HTTP, TCP, and WebSocket) all in the same execution instance. Throughout the document are presented some actual frameworks, their main gaps and how the developed solution pretends to fix them.

Key words: web, framework, APIs, actions, modules, NodeJS

Resumo

O Stellar é uma Framework web que permite fazer a separação entre as componentes visual e lógica de uma aplicação web. O sistema consiste num motor extremamente flexível e extensível que contém, à partida, um conjunto de funcionalidades base que serão usadas pelos programadores para a criação de módulos. Os módulos são um conjunto de funcionalidades criadas por qualquer programador agrupadas por área, essas funcionalidades serão carregadas pelo motor de forma a disponibilizar as funcionalidades aos clientes. O Stellar suporta diferentes protocolos (HTTP, TCP e WebSocket) tudo na mesma instância de execução. Ao longo do documento são apresentadas algumas *frameworks* atuais, as suas principais falhas e como a solução desenvolvida pretende resolvê-las.

Palavras Chave: web, Framework, APIs, ações, módulos, NodeJS

Epígrafe

“A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo”
- Albert Einstein

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos

API Application Programming Interface

BDD Behavior-Driven Development

CMS Content Management System

CRUD Create Read Update Delete

DOM Document Object Model

DRY Do not Repeat Yourself

GUI Graphic User Interface

JSON JavaScript Object Notation

MVC Model-View-Controller

ORM Object-Relation Mapping

REST Representational State Transfer

SAP Single Page Application

TCP Transmission Control Protocol

XML Extensible Markup Language

WWW World Wide Web

Glossário

WebSocket: é um protocolo de comunicação de computador, que fornece um canal de comunicação full-duplex através de uma única ligação TCP. O protocolo WebSocket é um *standard* definido pela IETF no RFC 6455 em 2011.

Hook: é essencialmente um lugar no código que permite a execução de lógica de outro módulo a fim de fornecer um comportamento diferente ou reagir quando alguma coisa acontecer.

Framework Full-Stack: é uma *framework* que apoia no desenvolvimento tanto do *back-end* como do *front-end* de uma aplicação.

Proxy: em redes de computadores, um servidor proxy é um servidor (sistema computacional ou uma aplicação) que age como um intermediário para as requisições de clientes a requisitar recursos de outros servidores.

Promise: objeto para computação assíncrona. Uma Promise representa um valor que pode estar disponível agora, no futuro ou nunca.

1 Introdução

1.1 *Contextualização*

Há poucos anos atrás era impensável prever a evolução e a dependência que iria existir de tecnologias e da *WWW* (World Wide Web). A evolução foi vertiginosa, mas de tal forma aceite pela sociedade que hoje em dia as pessoas não se vêm sem estas ferramentas, tanto para uso profissional como pessoal. Esta evolução levou a uma grande oferta de opções a nível tecnológico, de tal forma, que quase todas as pessoas, não só aderiram como são incapazes de fazer o seu dia a dia normalmente. Atualmente, no quotidiano é quase impossível para as pessoas não utilizar alguma destas tecnologias e suas ferramentas.

O crescimento da Internet e Intranet teve um grande impacto no comércio, indústria, banca e finanças, educação, gestão, no setor do entretenimento e em todo o nosso quotidiano. Muitos sistemas legados e sistemas de base de dados foram migrados para a *WWW*, assim como informatizadas inúmeras operações manuais. O comércio eletrónico continua em rápido crescimento, ultrapassando os territórios de cada país de origem e chegando a todo o mundo. Mais recentemente, estamos a assistir ao surgimento de aplicações bastante complexas, distribuídas devido à popularidade e ubiquidade da própria *web* e das suas funcionalidades [1].

Nos primórdios da internet, as soluções desenvolvidas eram simples e objetivas não sendo necessário grandes trabalhos de organização. Hoje em dia existe a necessidade de se estudar e abordar a melhor perspetiva de organização e desenvolvimento para não se correr o risco de chegarmos a um determinado ponto do projeto e não haver saída para o que foi inicialmente proposto.

Com a aceitação das massas na utilização da internet para acesso à informação, suporte de decisão e até mesmo comparação de produtos, as empresas naturalmente ajustaram o seu paradigma de marketing e começaram a considerar cada vez mais a internet como um meio de comunicação e divulgação dos seus produtos. Esta revolução comercial levou a que surgissem soluções complexas de exposição e divulgação de marcas e produtos.

Por exemplo o *e-commerce* é uma das soluções que surgiu com a imensa adesão à internet, pois através deste tipo de solução é permitido utilizar uma estrutura para executar transações completas a nível global, ou seja, conseguimos chegar a qualquer ponto do globo facilmente

e proceder a transações sem que haja necessidade de deslocações ou mesmo custos. Estes tipos de soluções requerem acesso a bases de dados, organização de informação e mesmo segurança da informação, tornando toda a sua envolvência bastante sensível, confusa e complexa logo qualquer tipo de *framework* que permita orientar e ajudar nessa mesma organização é bem-vinda.

Hoje em dia as aplicações web são cada vez mais complexas, já não existem aquelas páginas com conteúdo que se altera de semana a semana. Hoje, não só queremos que a cada *refresh* da página o conteúdo tenha sofrido alterações, mas também queremos que sempre que se atualize determinado conteúdo numa página não seja necessário carregar a mesma na sua totalidade permitindo assim rapidez na atualização e até mesmo poupança de tráfego. As pessoas querem a experiência nativa que as aplicações móveis proporcionam, desde o aparecimento dos *smartphones* a complexidade necessária para atingir este estado de experiência de utilização é elevada, sendo um desafio para quem tem que desenvolver tais projetos. Atualmente, existe a necessidade de desenvolver aplicações móveis em que a fonte de dados seja comum entre si, ou seja, para que as aplicações tenham informação mais abrangente e consequentemente sejam mais úteis. Para isso é necessário alargar a quantidade de informação a disponibilizar e é aqui que surge a necessidade de separar o projeto em diferentes partes, neste caso a aplicação *web*, móvel e a API.

Atualmente, se observamos mais atentamente podemos constatar que estamos basicamente a desenvolver sempre as mesmas aplicações, mas com um visual diferente. Já foram desenvolvidas inúmeras aplicações **CRUD**, possivelmente o problema que a equipa de trabalho vai enfrentar já foi resolvido, mas por uma razão qualquer, continuam-se a desenvolver as mesmas aplicações e muitas vezes a partir do zero. Uma vez que um problema esteja resolvido, a solução deveria ser aplicada a outros problemas com características semelhantes, como uma espécie de modelo. Se pensarmos bem, por exemplo na industria automóvel, quando um modelo entra em produção são dadas instruções a todos os trabalhadores para replicarem o design em todas as unidades produzidas. Não dizemos a cada trabalhador para desenhar o seu próprio carro a partir do zero. Em vez disso, os trabalhadores devem seguir o modelo dado e se assim fizerem as coisas irão correr como planeado, uma vez que não há marguem para acontecer problemas inesperados. Quando as instruções são tão claras que qualquer um as pode seguir, torna-se um trabalho braçal, ou mesmo mais que isso, torna-se num trabalho de um robô [2].

Engenheiros devem estar a resolver problemas novos e interessantes e não a reconstruir as mesmas aplicações vezes e vezes sem conta. Isso é trabalho para robôs.

1.2 *Framework Web*

Uma *framework* web é um conjunto de ferramentas, componentes e metodologias de desenvolvimento mais ou menos definidas que facilita o desenvolvimento de aplicações *web*. [3] Muitas das novas *frameworks* fornecem diferentes paradigmas para abordar o desenvolvimento de aplicações. As *frameworks* conseguem facilmente usar e manipular os dados diretamente da base de dados através de uma camada de abstração concebida para reduzir o tempo de desenvolvimento. Muitas das *frameworks* tentam seguir o principio "*Do not Repeat Yourself*" (**DRY**) [4] [5], este principio permite aplicar mudanças a um elemento do sistema sem que seja necessária a alteração da lógica de outros elementos não relacionados.

De forma a melhor perceber como uma *framework* pode ajudar os programadores no desenvolvimento de aplicações para a **WWW**, é importante perceber alguns conceitos relacionados com este tipo de aplicações. Uma aplicação *web* pode ser dividida em, pelo menos, três camadas, normalmente, têm uma camada de dados, uma camada aplicacional e uma camada visual [6].

A camada de dados é onde a informação é guardada [6]. Esta refere-se aos dados que são guardados de forma persistente. Os dados podem ser guardados em bases de dados relacionais, através de documentos, como JavaScript Object Notation (**JSON**) ou Extensible Markup Language (**XML**), ou então através de dados em bruto.

A camada aplicacional é onde a lógica de negócio reside [6]. Tipicamente esta camada é usada para processar os pedidos do cliente e preparar a *Graphical User Interface* (**GUI**) para ser apresentada ao utilizador.

A camada visual é onde o utilizador interage com a aplicação, nas aplicações *web*, isto é feito através de um *browser*. Esta camada não é apenas usada para apresentar informação ao utilizador, mas também tem que tratar da inserção de nova informação na aplicação.

Dada a existência destas diferentes camadas existe a necessidade que estas comuniquem entre elas. É no meio deste processo de comunicação que é necessário código extra, como por exemplo, para abrir uma ligação com a base de dados e formatar dados de *output* para a página *web*. É aqui que as *frameworks* atuais tentam facilitar a vida aos programadores criando estes pontos de conexão.

Com o crescimento do uso das aplicações *web* começou-se a sentir a necessidade de um conjunto de ferramentas que suportassem o desenvolvimento de aplicações *web* e

reduzissem a necessidade de realizar tarefas repetitivas, tais como, autenticação, autorização, envio de e-mails, etc [7].

Como um resultado desta necessidade, nos anos recentes, surgiram diversas *frameworks* escritas em diferentes linguagens, essencialmente seguindo o padrão Model-View-Controller (**MVC**). Este padrão divide o *software* em três partes independentes, começando pela representação da informação, passando pela forma como a informação é apresentada e terminando no como o utilizador interage com a informação [8].

1.3 *Motivação e Objetivos*

Como foi referido acima estamos constantemente a repetir as mesmas tarefas para todas as aplicações desenvolvidas. Essa é a motivação para o desenvolvimento deste projeto, o objetivo é desenvolver uma *framework* que permita a reutilização das funcionalidades já desenvolvidas em projetos anteriores e incentive os programadores e engenheiros (quem desenha as soluções) a adotar a mesma no desenvolvimento dos seus novos projetos. Pretende-se, no entanto, que esta reutilização seja conseguida de forma muito mais fácil.

Existem diversos cenários de utilização para a esta solução, como por exemplo, na área dos Sistemas de Gestão de Conteúdos (**CMS**). Os **CMS's** atuais são rígidos em termos de aparência, aumentando o esforço necessário para o desenvolvimento de uma interface gráfica que resolva uma determinada necessidade de um dado cliente. Essas necessidades diferem entre clientes e uma mesma abordagem não é a solução para todos os problemas.

Um outro cenário seria uma empresa que desenvolveu um projeto X que possui as funcionalidades F1, F2 e F3, e agora tem que construir um projeto Y com as funcionalidades F1, F2 e F4. Poderíamos dizer que apenas têm que utilizar as funcionalidades já desenvolvidas no projeto X (F1 e F2) e copiar para o novo projeto, o problema é que as *frameworks* atuais não foram desenvolvidas para maximizar a reutilização das funcionalidades já desenvolvidas. Com o atual estado do mercado, as empresas têm que ter ferramentas verdadeiramente flexíveis e fáceis que permitam acompanhar as rápidas alterações de requisitos dos projetos e elevada reutilização dos componentes já desenvolvidos em projetos anteriores.

Sendo assim, este projeto tem como objetivo o desenvolvimento de uma *framework* moderna que recorra ao uso das últimas tecnologias e técnicas, seja multiplataforma, que seja capaz de se adaptar a diferentes cenários de utilização e que dê resposta aos novos desafios tecnológicos que se avizinham, como funcionalidades em tempo-real necessárias para o

desenvolvimento da *Internet of Things* e de funcionalidades que potenciem a experiência de utilização das aplicações desenvolvidas. Por fim, também se ambiciona que seja capaz de ajudar as empresas e programadores a dar uma melhor resposta à constante e rápida alteração de requisitos, na maximização de lucros e na minimização do tempo de desenvolvimento.

1.4 *Organização*

Este relatório encontra-se estruturado em 8 capítulos. O presente capítulo corresponde ao primeiro de um conjunto de 8, este contém uma introdução do trabalho aqui apresentado, introduz os objetivos, motivação e organização do documento. O próximo capítulo descreve a forma como os desafios são resolvidos nos dias de hoje, os problemas das abordagens atuais e a solução introduzida pelo Stellar. O capítulo 3 descreve em maior detalhe a abordagem baseada em ações e o porquê da adoção desta abordagem. No capítulo 4 é apresentada a arquitetura da solução desenvolvida. O capítulo 5 descreve e detalha a implementação da solução. No capítulo 6 é descrito o processo de validação da solução e os grandes grupos de testes. Já no capítulo 7 é apresentado um caso de estudo e todos os passos para a implementação do mesmo. Por último, no capítulo 8 são feitas algumas considerações gerais, caracterizando o resultado obtido e são também apresentadas pistas acerca do trabalho futuro.

2 Plataformas Semelhantes

O iniciar de um novo projeto a equipa de desenvolvimento depara-se com uma série de questões que necessitam resposta de forma a determinar qual o caminho a seguir no desenvolvimento do projeto. A primeira questão é: “Existe a necessidade de utilizar uma *framework* para o desenvolvimento do novo projeto?”. Esta pergunta está depende da existência de soluções no mercado para o que está a ser pedido ou de soluções que sejam facilmente adaptáveis aos requisitos pedidos em vez de se desenvolver tudo a partir do zero. A segunda questão é: “Que linguagem deve ser usada para o desenvolvimento?”. A resposta a esta pergunta depende não só das capacidades da equipa, mas também do tempo disponível até à entrega do projeto, pois pode não haver tempo para aprender uma nova linguagem. Outro aspeto a ter em atenção é se a linguagem consegue dar resposta a todos os requisitos do projeto, como por exemplo o desenvolvimento de funcionalidades em tempo real, pois grande parte das linguagens usadas na *web* são *stateless*, ou seja, o estado da ligação não persiste entre pedidos. Por último, “Qual é a *framework* a usar?”. Esta é sem dúvida a questão mais pertinente, porque vai ditar como todo o projeto será organizado, que tipo de servidores irá suportar, a que padrões responde e os programadores terão que seguir as *guidelines* rígidas da *Framework*.

Existe uma grande variedade de *frameworks* que podem ser usadas para o desenvolvimento de aplicações *web*, em diferentes linguagens, com diferentes objetivos e com diferentes padrões. Das inúmeras *frameworks* existentes foram selecionadas três que se diferenciam no seu propósito, linguagem de programação e paradigma. Escolheram-se as *frameworks* Lumen, Meteor e ActionHero como sendo as mais relevantes e descritas neste capítulo. Atualmente são também algumas das mais usadas, tendo em conta o Google Trends [9] e o número de *stars* que os projetos possuem no GitHub [10], já o ActionHero foi escolhido devido à sua semelhança com o sistema desenvolvido. Um dos requisitos de escolha é que todas as plataformas sejam *open source* e gratuitas. O propósito deste capítulo é mostrar os seus pontos fortes e limitações, por fim mostrar a sua mais valia e onde o Stellar mais se destaca [11] [12] [13].

2.1 *Laravel & Lumen*

A *framework* Laravel [14] é uma das *frameworks* mais usadas da atualidade. Apesar de ser muito conhecida pela comunidade PHP é reconhecida e usada por muito outros

programadores sendo muito poderosa e concebida para desenvolver aplicações *web* em PHP. Segue o padrão **MVC**, de forma a fazer uma divisão mais clara e percetível entre o código dos *models*, dos *controllers* e das *views*. Sendo um projeto *open source* licenciado sobre a licença MIT, é possível modificar e utilizar o *software* livremente sem qualquer limitação. [15]

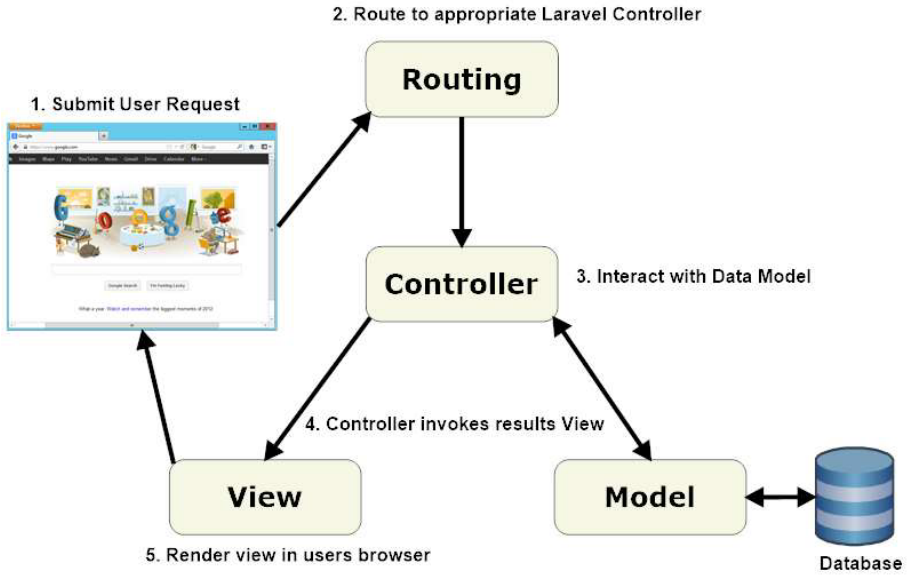

Figura 1- Arquitetura do Laravel [16]

Na imagem acima (Figura 1- Arquitetura do Laravel), é apresentada a arquitetura do Laravel, pode-se constatar que corresponde à arquitetura **MVC** [17]. Sempre que um pedido do utilizador chega ao servidor é necessário iniciar toda a *framework* e redirecionar o pedido para o sistema de rotas. O sistema de rotas permite associar um determinado URI a uma ação, isto é, assim que é encontrada a rota correspondente ao pedido, o *controller* em questão é instanciado e o método associado à rota é executado. Durante a execução do *controller* é possível aceder a dados persistentes, guardados numa base dados, através de um *model* e mais tarde construir uma *view* com os dados obtidos enviando essa *view* como resposta ao pedido do cliente.

2.1.1 Funcionalidades

De seguida são apresentadas algumas funcionalidades do Laravel de uma forma mais técnica assim como algumas das suas limitações mais relevantes. As tecnologias que serão apresentadas são as seguintes: bundles, Eloquent ORM, RESTfull controllers e inversion of control.

Bundles

Uma vez que o Laravel utiliza o composer [18] para carregar as suas dependências, existe a possibilidade de criar pacotes que permitem adicionar mais funcionalidades à aplicação *web* que está a ser desenvolvida. Este conceito tem o nome de *packages* [19], um *package* contém pequenas funcionalidades de uma aplicação que podem ser reutilizadas em outras, mas estes necessitam de configurações adicionais, não conseguem registar rotas, *controllers* e *models* diretamente, tudo isso tem que ser feito manualmente pelo programador.

Eloquent ORM

O Eloquent ORM é a implementação mais avançada do ActiveRecord [20] [21] em PHP. Esta implementação permite facilmente criar um *model* que represente uma dada entidade na base de dados e desde logo poder manipular os dados da mesma. Todo o processo de ligação fica a cargo do Laravel. Também é possível definir relações e restrições entre as diferentes entidades. Esta é a terceira camada de abstração para realizar a ligação à base de dados, sendo possível usar diferentes tipos de SGBDs, como por exemplo: MySQL, PostgresSQL, SQL Server, SQL Lite. A imagem (Figura 2) abaixo apresenta os diferentes níveis de abstração no sistema de ligação à base de dados do Laravel.

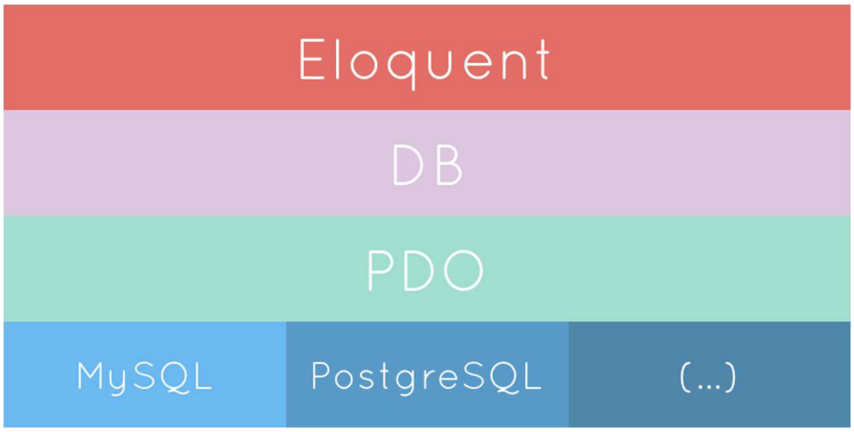

Figura 2 - Níveis de Abstração da Base de Dados do Laravel

Uma vez que é usado o PHP Document Object (PDO) [22] para proceder à ligação aos *drivers* nativos, é possível alternar entre sistemas de base de dados sem a necessidade de aplicar alterações no código já desenvolvido, a não ser em certas *raw queries* que utilizem sintaxe específica de um sistema de gestão de base de dados.

RESTfull Controllers

De forma opcional, os programadores podem criar aplicações completas seguindo o padrão Representational State Transfer (REST) [23] que permite fazer uma melhor separação entre as diferentes secções da aplicação *web*. Este padrão não é só usado no desenvolvimento *full-stack* [24], mas também na criação de APIs *web*. Com o elevado crescimento na procura de ferramentas que apoiem o desenvolvimento de APIs, por parte dos programadores, os criadores do Laravel resolveram conceber uma nova framework focada apenas no desenvolvimento de APIs, o Lumen [25]. O Lumen contém todas as funcionalidades que se espera de uma *framework* de última geração, como por exemplo: *database schemas*, Object-Relational Mapping (ORM), *authentication*, *authorization*, gestão de rotas e *queues* para executar tarefas de forma assíncrona.

Inversion of Control (IoC)

O IoC é um principio que permite delegar o controlo da execução a uma infraestrutura, normalmente designada de *container*. Ou seja, o programador em vez de manter o controlo em todo o fluxo de execução, delega algumas funcionalidades a um terceiro, como por exemplo a injeção de dependências [26].

No Laravel é muito comum pedir-se à framework para instanciar um determinado objeto em vez de realizar a tarefa repetitiva de instanciar uma nova classe e ter a necessidade de conhecer as suas dependências.

2.1.2 Pontos Fracos

O Lumen nasceu para tentar resolver as novas necessidades. Sendo uma micro-framework, permite consumir muito menos recursos à maquina, permitindo responder a um maior número de pedidos dos clientes e contém apenas as funcionalidades necessárias para o desenvolvimento de APIs, mas não é o suficiente. O reaproveitamento de lógica e funcionalidades entre projetos é limitada, copiar *controllers, schemas* e *models* manualmente não é prático, necessitam de adaptação e configurações para funcionarem corretamente e nem é solução. Sendo uma *framework* escrita em PHP não permite de forma nativa desenvolver funcionalidades em tempo real que tirem partido de novos protocolos como WebSocket.

O Eloquent, é uma classe extremamente poderosa e muito fácil de utilizar, isto graças a toda a magia que acontece "*debaixo de capô*", embora traga algumas complicações. Quando aplicamos um nível tão elevado de abstração pode ficar complicado fazer *debug*, é aqui onde o Eloquent falha, quando acontece algum problema é complicado saber onde aconteceu [13].

Por fim, um outro problema que é muito inconveniente é a quantidade de alterações necessárias entre *minor releases*. Como esperado de uma *release* mais pequena, esta deveria trazer algumas alterações e a adição de funcionalidades, mas não deve quebrar funcionalidades já desenvolvidas. Geralmente não é o que acontece, não só no código, mas também na estrutura de ficheiros da aplicação, isso acontece sempre que é lançada uma nova versão, esta trás algumas quebras em funcionalidades da versão anterior. A atualização das aplicações, para que os programadores tirem partido das funcionalidades mais recentes, é custosa e demorada.

2.2 *Meteor*

O Meteor é uma *framework full stack* desenvolvida inteiramente em JavaScript, esta foi desenhada para o desenvolvimento de aplicações *web*, aplicações móveis e mais recentemente *desktop* graças à integração com o Electron [27]. Este projeto é *open source* e licenciado sobre a licença MIT, mas os seus pacotes têm licenças próprias que por vezes não são livres como o *core*.

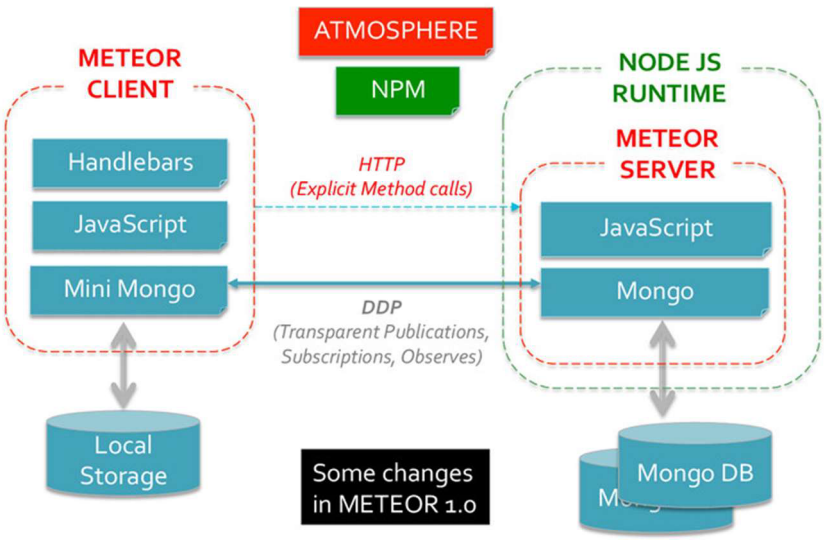

Figura 3 - Arquitetura do Meteor [28]

Como podemos ver na imagem da arquitetura de uma aplicação desenvolvida em Meteor (Figura 3 - Arquitetura do Meteor), este está presente tanto no lado do servidor como no lado do cliente. No lado do servidor, o Meteor é desenvolvido sobre a plataforma Node.JS [29] e para armazenar dados de forma persistente é usado o MongoDB [30]. No lado do cliente, é usado o Mini Mongo para criar a experiência de utilização de uma base de dados não relacional no *browser* e é usado o Handlebars [31] para construir as *views* para o cliente. A

forma de comunicar entre os dois lados da aplicação é através de um protocolo chamado **DDP**, que permite publicar e subscrever alterações em coleções de dados, ou através de uma chamada normal em HTTP. Quanto à gestão de pacotes é usado o Node Package Manager (**NPM**) [32] e para pacotes específicos para o Meteor pode-se usar o Atmosphere [33].

2.2.1 Funcionalidades

Nos subcapítulos seguintes serão apresentadas algumas das funcionalidades mais relevantes do Meteor. As funcionalidades em analise serão: tempo real, prototipagem, *packages* e blaze.

Tempo Real

O Meteor é completamente estruturado para que todas as funcionalidades desenvolvidas sejam inteiramente em tempo real. É possível manter dados sincronizados entre o cliente e servidor e utilizar funcionalidades de publicação e subscrição para partilhar informação entre todos os clientes e servidor. O Meteor recorre a WebSockets para que as comunicações sejam realizadas em tempo real.

Prototipagem

O Meteor possui um mecanismo que desabilita todos os mecanismos de segurança, ou seja, a aplicação cliente tem acesso direto e total à base de dados, sem restrições, pode chamar diretamente métodos do servidor e todos os clientes possuem uma cópia integral da informação guardada. Esse mecanismo pode ser usado para que rapidamente se crie um protótipo de uma aplicação *web*, móvel ou *desktop*, sem que tenha de se preocupar com aspetos “secundários” (autenticação, autorização, ...) numa fase inicial, tudo para que a equipa de desenvolvimento se possa concentrar ao máximo na produção de valor para o negócio.

Packages

A arquitetura do *core* do Meteor foi concebida de forma a permitir que os programadores possam instalar pacotes que adicionem pequenas funcionalidades à aplicação em desenvolvimento. Mais recentemente foi criado um repositório com todos os pacotes do Meteor, o Atmosphere [33].

Blaze

O Blaze [34] é o sistema de *templante* do Meteor que permite desenvolver *views* reativas sem a instalação de pacotes adicionais para o efeito. A *syntax* é baseada no Handlebars [31] e o

motor recorre a um motor **DOM** reativo para proceder à rápida atualização das *views* e apenas atualizar as secções que sofreram alterações [34].

2.2.2 Pontos Fracos

O Meteor tem funcionalidades interessantes, é uma *Framework* bem preparada para alguns cenários de utilização como prototipagem de aplicações e desenvolvimento de pequenas aplicações móveis, mas contém muitos problemas. Um dos problemas é não haver uma forma padrão de o programador organizar o seu código no projeto, isto faz com que se tenha que pensar no "Como vou organizar isto? Porque vou organizar assim?". Esta falta de um modelo de desenvolvimento resulta numa desorganização tremenda que leva a que seja difícil manter projetos grandes e dificulta a entrada a novo membros à equipa de desenvolvimento. Um outro problema, é que não existe uma separação lógica entre o código do lado do cliente e do servidor, por vezes fica difícil orientar-se no projeto. No que toca a testes unitários, estes nem sequer estão pensados, surgem alguns pacotes de tempos a tempos, criados pela comunidade, mas estão sempre em constantes alterações. Para concluir, usa apenas uma única tecnologia no *frontend* e no *backend* o que impede/dificulta que no futuro se possa facilmente alterar de tecnologia. Colocando a organização do projeto de parte, ao longo do desenvolvimento em Meteor o programador ver-se-á obrigado a fazer "maroscas" no código de forma a contornar alguns problemas de *design* da plataforma.

A falta de *hooks,* que deveria permitir a execução de lógica de outros módulos durante a execução normal do programa, leva a que seja necessário reescrever os módulos a fim de introduzir novos comportamentos e assim dar resposta às necessidades. Sendo assim, pode-se dizer que a reutilização é deitada por terra [35].

2.3 *ActionHero*

Por fim, a ActionHero é uma *framework* apenas focada na criação de APIs *web*. Esta *Framework* segue um conceito não muito usado, trata-se de uma *framework* baseada em ações, todas as funcionalidades são ações, isto permite um melhor isolamento das funcionalidades do projeto de forma a torná-las independentes. A ActionHero é uma *framework* bastante extensível, mas com algumas falhas, no que toca a funcionalidades é muito mais pobre do que outras *frameworks* atuais, como tem menos funcionalidades é mais limitada. Um dos pontos a seu favor é ser *open source,* estando licenciada sob a licença Apache 2.0 [36].

2.3.1 Funcionalidades

Nos próximos subcapítulos serão apresentadas algumas das funcionalidades mais interessantes da *framework* ActionHero. Desse conjunto fazem parte os múltiplos protocolos, sistema de chat, cluster e localização.

Múltiplos Protocolos

O ActionHero permite o uso de múltiplos protocolos em simultâneo sem a necessidade de criar múltiplas instâncias de execução. Atualmente os protocolos suportados são: HTTP, TCP e WebSocket. Isto é possível por causa do ActionHero ter sido implementado em cima de Node.JS.

Sistema de Chat

O ActionHero facilita as comunicações em tempo real, não só entre o servidor e o cliente, mas também entre clientes. Este sistema de *chat* permite enviar mensagens públicas ou privadas entre clientes.

Cluster

Através do uso de um servidor Redis [37], o ActionHero, partilha os dados entre as diferentes instâncias da *framework*. O sistema de *cache*, comunicações entre os nós do *cluster* e o sistema de tarefas assíncronas, estão desenvolvidas em volta do Redis, assim, de forma fácil é possível escalar a aplicação.

Localização

O ActionHero segue um padrão *i18n* [38] de forma permitir personalizar qualquer *string* no servidor, nas respostas das ações ou mensagens de *log*. Com a utilização deste padrão consegue-se melhorar a resposta às necessidades dos programadores, uma vez que apenas é preciso criar um novo ficheiro com as frases traduzidas para os programadores adaptarem as suas aplicações para diferentes idiomas.

2.3.2 Pontos Fracos

O Stellar foi muito inspirado na *framework* ActionHero, esta também segue um modelo baseado em ações, mas contém alguns problemas/limitações. A principal desvantagem é a duplicação de ficheiros da *framework*, ou seja, têm que estar duplicados em todos os projetos e o código que é efetivamente do projeto fica misturado com o código base da *framework*, isto torna difícil de atualizar a versão da *framework* ou aplicar *patchs* críticos de segurança de

forma fácil e rápida. Outro ponto negativo, tal como nas *frameworks* anteriores, é que existem limitações no que toca à modularização das diferentes funcionalidades desenvolvidas. Por fim, outro problema é a forma como a documentação é apresentada, é de difícil compreensão e por vezes é difícil compreender alguns conceitos ou mecanismos apresentados. A documentação de qualquer plataforma/tecnologia é de extrema importância, hoje em dia as pessoas querem rapidamente compreender o funcionamento de um sistema/tecnologia sem que tenham que perder muito tempo a estudar a mesma.

2.4 *Comparação*

A tabela abaixo apresenta uma comparação entre o Stellar e as *frameworks* faladas anteriormente.

Característica	Laravel	Meteor	ActionHero	Stellar
Baseada em Ações			✓	✓
MVC	✓			
Full Stack	✓	✓		
Ligação a BD	✓	✓		✓
Validadores de *Inputs*	✓			✓
Múltiplos Protocolos			✓	✓
Comunicação em Tempo Real		✓	✓	✓
Modular				✓
Processamento de Tarefas Integrado			✓	✓
Chat			✓	✓
Sistema de Eventos	✓			✓
Executa Apenas com Ferramenta Global				✓
Cluster			✓	✓

Tabela 1 - Caracteristicas das Frameworks Comparadas

Existem realmente boas *frameworks* no mercado que permitem resolver de uma forma relativamente fácil e rápida os problemas diários, mas sempre numa perspetiva de curto prazo e com grandes limitações no que toca à reutilização, porque reutilizar não é fazer *Copy&Paste*. Estas *frameworks* são uma solução para empresas que trabalhem apenas num único projeto onde conseguem resultados satisfatórios, mas para organizações que estão constantemente a trabalhar em novos projetos, similares em termos de funcionalidades, não só é necessário andar a copiar os ficheiros necessários como também a aplicar modificações. Por vezes, existe mesmo a necessidade de reescrever a funcionalidade por inteiro, porque esta não funciona como esperado.

O Stellar permite aos programadores criar funcionalidades apenas uma vez e aproveitá-las para projetos futuros. Realmente, esta *framework* foi desenhada para colmatar os problemas das *frameworks* atuais, nascendo da fusão dos melhores mecanismos destas *frameworks*. Ao identificar as falhas das mesmas, foi criada uma ferramenta de trabalho capaz de responder aos desafios do dia a dia e suportar tecnologias futuras, oferecendo múltiplos tipos de protocolos na mesma instância de execução e a capacidade de comunicar em tempo real de forma simples sem adicionar complexidade desnecessária ao código das aplicações. O seu simples, mas poderoso sistema de eventos permite estender e modificar o comportamento de funcionalidades de outros módulos ou da própria plataforma sem a necessidade de reescrever módulos, tudo para atingir um grau de reutilização não antes possível com outras soluções, tornando assim os componentes mais genéricos e adaptáveis para diferentes cenários de utilização. Ao promover uma solução apenas para o desenvolvimento de **API's**, está a permitir criar a mesma lógica de sempre, mas apenas na primeira vez, promovendo a manutenção e a evolução dos módulos, já que esta lógica fica completamente independente da *view*. Assim, é possível criar produtos desenhados com o intuito de colmatar diferentes necessidades de diferentes clientes, mantendo assim o foco na experiência de utilização e não na lógica. Por outras palavras, é pretendido que a mesma API e/ou os mesmos módulos, possam alimentar diferentes aplicações.

3 Abordagem Baseada em Ações

No decorrer do capítulo é feita uma pequena introdução sobre um dos problemas que as organizações estão a enfrentar hoje em dia, seguindo-se uma explicação do porquê da adoção de uma abordagem baseada em ações, quais as suas vantagens e o que isso possibilita.

Este capítulo encontra-se dividido em três partes distintas. A primeira parte faz um breve enquadramento do problema atual. O porquê da escolha de uma abordagem por ações é descrito na segunda parte. Por fim, é apresentada uma conclusão para finalizar e fechar algumas ideias.

3.1 *Problema Atual*

Como já referido anteriormente, atualmente não faz muito sentido desenvolver uma aplicação usando *frameworks full-stack,* visto que mais tarde poderá ser necessário criar uma aplicação móvel, *desktop*, alimentar qualquer outro serviço. No final todas se irão alimentam da mesma fonte de dados e todas elas têm camada visual especifica para a sua plataforma. Ao desenvolver apenas duas camadas (lógica e dados) é possível, de forma fácil, desenvolver novas aplicações que consumam a mesma fonte de dados, pois apenas necessitam de fazer chamadas à **API**. Com estas duas camadas desenvolvidas, podemos persistir ou apresentar os dados nelas envolvidos com apenas uma simples chamada não interessando o destino dos dados, esse destino será definido por qualquer aplicação que faça um pedido à **API**. Da mesma forma, não interessa a aplicação que acede à **API**, desde que siga os requisitos necessários que foram definidos na **API** para fazer as chamadas.

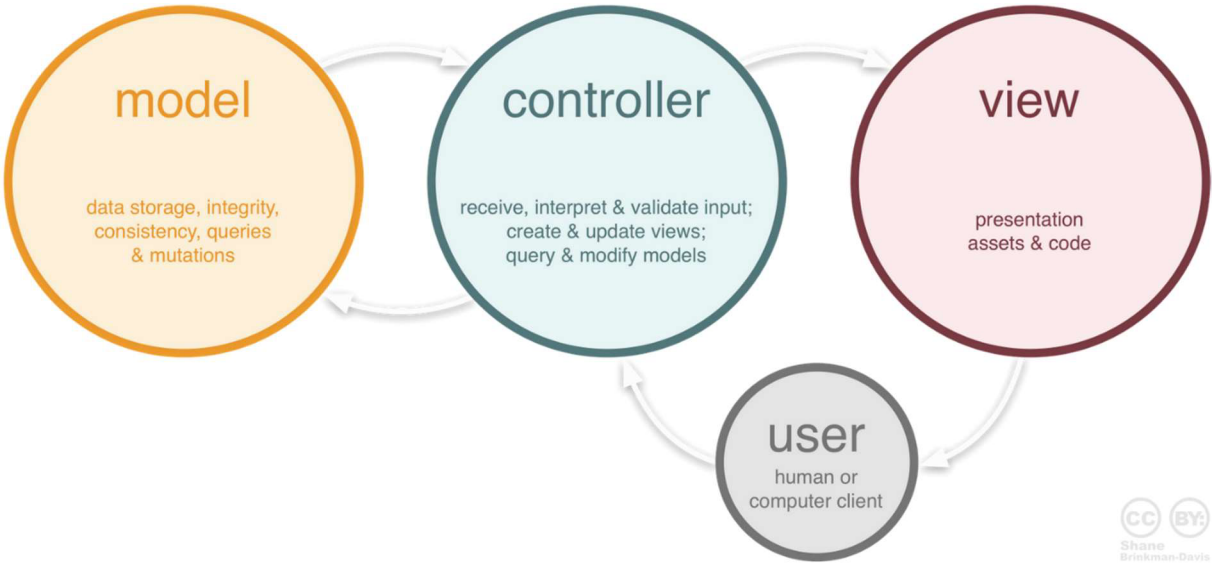

Figura 4 - Model-View-Controller [39]

A imagem acima (Figura 4) mostra o modelo **MVC** e a sua interação com o utilizador. Na imagem podemos ver as diferentes camadas do modelo e as funções associadas a cada camada.

Ao juntar as três camadas (visual, lógica e dados) apenas num projeto, leva a uma estrutura de pastas e a uma implementação mais complexa, tornado a manutenção do projeto mais complicada e demorada, além de tornar mais difícil a adaptação de novos programadores no projeto. Mais ainda, seguindo esta abordagem haverá mais espaço para os programadores cometerem erros de *design* que vão contra o padrão adotado. Isto é muito comum quando falamos no modelo **MVC**, pois seguir padrões pode ser frustrante e alguns programadores começam a misturar diferentes camadas, desvanecendo assim a clara distinção entre elas.

Desta forma, a arquitetura **MVC** deixa de fazer sentido e é deixada de parte, uma vez que não existe a necessidade de desenvolver a camada da *view* (representada por V na arquitetura **MVC**).

3.2 *Abordagem a Seguir*

A *framework* aqui apresentada segue uma abordagem em ações, isso quer dizer que todas as funcionalidades são ações. Mas, porque não continuar com o MC (*Model-Controller*) e as rotas **REST**? Simples, a chamada de uma rota não é nada mais nada menos do que um pedido do cliente para a tomada de uma determinada ação, daí ser chamada de ação e esta

é algo muito humano. Ao pesquisar no dicionário pela palavra “ação”, pode-se ler a seguinte definição: - “Movimento ou atividade para obter um determinado resultado; influência ou efeito sobre algo ou alguém”. Uma ação é isso mesmo, é um pequeno bloco de lógica que irá aplicar alterações à camada de dados para devolver um resultado.

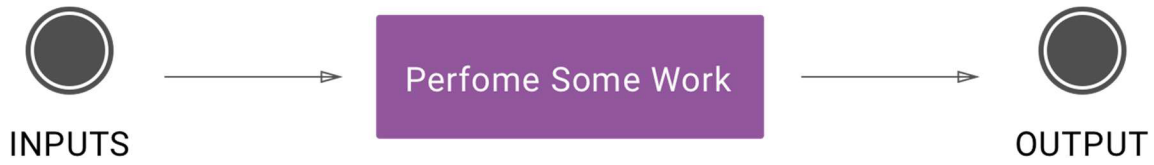

Figura 5 - Representação Simples de uma Ação

Como se pode ver na Figura 5, uma ação recebe um conjunto de *inputs* que influenciam a execução. Após uma série de passos (execução) é recebido um *output*, um resultado, o qual é devolvido ao cliente. Chamar uma ação é muito mais natural do que fazer um pedido a uma rota **REST**, por exemplo, se a intenção é apagar um utilizador, em vez de fazer uma chamada **HTTP** do tipo **DELETE** à rota **/users/12** pode-se indicar ao servidor para executar a ação **deleteUser?id=12**. Os humanos são isso, humanos, não maquinas, por isso devem proceder como tal, usar uma linguagem mais natural de forma a facilitar a comunicação. Desta forma executar operações é mais simples e muito mais lógico. Atenção, o Stellar continua a disponibilizar um sistema de rotas para quem quiser continuar a usar o sistema de forma *RESTfull* ou queira dar suporte a algum sistema legado que possua.

Além disso, uma abordagem por ações incentiva a reutilização de código, permitindo isolar cada funcionalidade da plataforma. Os programadores são motivados a desenvolver pequenos componentes altamente testáveis e com o menor número de dependências entre si. Portanto, além de facilitar a manutenção do código desenvolvido (permitindo uma melhoria incremental) é possível reutilizar pequenas partes de lógicas e combiná-las com outras ações de forma a criar uma ação mais complexa.

Implementando as funcionalidades usando ações, é possível desenvolver uma **API** capaz de comunicar através de diferentes protocolos sem a necessidade de adaptar o código para os mesmos. Esta abordagem é de tal forma flexível que é capaz de escalar desde um simples *website* até à *Internet of Things* ou então de servir como **API** de uma empresa que tenha que alimentar múltiplas aplicações em simultâneo.

A abordagem aqui apresentada é muito semelhante a uma *framework* já existente e que foi introduzida no capitulo anterior (na secção 2.3), o ActionHero [36]. O ActionHero possui um sistema de *plugins* [40] cujo objetivo é permitir desenvolver módulos que a *framework* conheça, mas a este sistema acresce uma complicação desnecessária, tanto aos criadores dos módulos, como aos seus utilizadores, já que este não é o objetivo padrão. Mesmo depois de anos da primeira versão para produção ter sido lançada, praticamente não existe módulo algum para ser usado por outros programadores. O Stellar é fortemente influenciado por esta *framework*, mas a solução aqui apresentada leva o conceito da reutilização a outro patamar, permitindo e incentivando a reutilização de todos os componentes desenvolvidos através de um sistema de módulos. Mais ainda, devido à anatomia das ações e à sua estrutura de elevada interoperabilidade, é possível, de forma fácil, escrever testes unitários sem dores de cabeça. O Stellar fornece um sistema e uma ferramenta de linha de comandos que permite escrever e executar testes de forma rápida, o sistema é tão simples, que mesmo alguém que nunca tenha tido contacto com testes unitários em JavaScript vai-se sentir completamente à vontade. Facilita e incentiva a escrever testes unitários, a seguir uma metodologia de desenvolvimento **BDD** transmitindo mais confiança a outros programadores que queiram utilizar um módulo já desenvolvido. Existem duas coisas importantes quando se partilha uma ferramenta na internet, uma é a sua documentação, que deve ser clara e de leitura simples, a outra é a validação dessa ferramenta, sobretudo através de testes unitários.

Como se pode ver uma abordagem por ações faz todo o sentido a fim de criar uma *framework* capaz de responder a diferentes desafios, em diferentes cenários. Seguindo esta abordagem e deixando de lado o **MVC** (que é largamente usado nos dias de hoje), consegue-se obter uma maior independência entre as camadas de dados e lógica e a camada da *view*. Por fim, esperasse que esta opção de *design* consiga incentivar e levar a reutilização de componentes já desenvolvidos a um outro nível não antes possível em outras *frameworks* e de uma forma tão facilitada.

4 Arquitetura da Solução

O Stellar é uma *Framework web* para a criação de **API's** *web de forma fácil*. Esta foi concebida para corrigir as principais falhas das *frameworks* atuais, tendo em vista um novo paradigma de fácil manutenção, escalabilidade e performance. A *Framework* pretende suportar múltiplos protocolos em simultâneo, permitindo partilhar o mesmo código base entre funcionalidades que necessitem de uma abordagem diferente em tempo-real (usando *websockets* e/ou o clássico Transmission Control Protocol [**TCP**]) ou então recorrendo ao protocolo **HTTP** seguindo (ou não) o standard RESTfull. A *Framework* foi desenvolvida em JavaScript ES6, correndo por cima do Node.JS, permitindo assim que novos utilizadores se consigam envolver na *framework* mais facilmente, graças ao elevado aumento de programadores que usam a plataforma Node.JS.

Ao desenvolver um sistema que processe apenas a lógica, podemos criar independência entre as vistas e o *back-end*. Isto torna a **API** reutilizável, independente e possível de ser portada para outros projetos, mas isso não é o suficiente. Dentro da **API** teremos que ser capazes de fazer uma separação das funcionalidades por área, para isso recorremos a um sistema de módulos que irá permitir fazer essa mesma separação e mais uma vez aumentar a modularidade do sistema desenvolvido.

Toda a plataforma foi pensada para aumentar a reutilização do código desenvolvido e permitir que seja possível migrar, remover ou adicionar funcionalidades de forma simples, sem envolver muito esforço por parte do programador. Esta possibilidade irá permitir diminuir o tempo de desenvolvimento/configuração, reduzir custos e tornar as soluções desenvolvidas mais eficazes e eficientes. Seguindo esta abordagem conseguimos com que os módulos se tornem mais maduros, possam evoluir ao longo do tempo e não tenham necessidade de passar por validações novamente, poupando assim tempo de desenvolvimento.

Ao longo deste capitulo será apresentada a arquitetura da solução desenvolvida, algumas decisões de design e o funcionamento, mais teórico, de alguns componentes da *framework*.

4.1 *Arquitetura do Core*

Neste subcapítulo é introduzida a arquitetura do **Core** do Stellar, também é explicado e definido o **Engine** e os **Satellites** assim como todos os seus comportamentos.

O *Core* é comporto por um **Engine**, uma série de **Satellites** e os *servers*. O **Engine** por si só não faz absolutamente nada, é apenas responsável por procurar os **Satellites** que compõem

o *Core* do Stellar. Estes **Satellites** são carregados de forma ordenada, de acordo com os seus níveis de prioridade, adicionando funcionalidades ao **Engine**.

A inicialização do Stellar é composta por três fases. Na primeira fase são carregados os **Satellites** críticos, necessários para o carregamento do resto do sistema, um exemplo é o **Satellite** das configurações. Na segunda etapa todos os **Satellites** do *Core* e dos módulos são carregados em memória e inicializados de forma ordenada de acordo com a suas prioridades. Por fim, na terceira e última etapa do processo de inicialização, os **Satellites** carregados são inicializados.

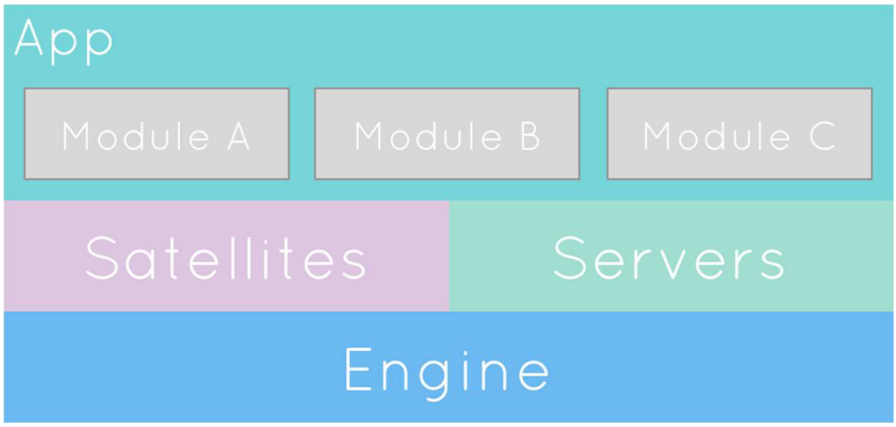

Figura 6 - Arquitetura do Stellar

Na Figura 6 está representada a arquitetura do Stellar, pode se ver os três principais componentes do *core* (**Engine**, **Satellites** e **servers**), assim como a constituição das aplicações serem compostas por módulos. O **Engine** é responsável por carregar toda a lógica da aplicação. Os **Satellites** são pedaços de lógica do Stellar, sendo os responsáveis por atribuir ações e funcionalidades à instância de execução. Os *Servers* podem ser de três tipos: HTTP, TCP e WebSocket. Mais adiante, neste documento, cada um destes componentes será abordado individualmente.

O *Core* está desenvolvido de forma a que seja possível ser executado em múltiplas instâncias do Stellar na mesma máquina, para isso apenas é necessário alterar as portas em que os *Servers* ativos estão à escuta. Também é possível usar diferentes instâncias na mesma porta, mas para isso é necessária uma configuração adicional. As instâncias podem também ser distinguidas através do nome de domínio, mas para isso é necessário recorrer a um servidor aplicacional (como por exemplo, o Nginx [41]) de forma a fazer de proxy dos diferentes domínios/URIs para portas diferentes.

Uma vez que o *Core* é uma ferramenta global (instalada no sistema) não existe necessidade de ter uma nova cópia do *Stellar* a cada projeto, desta forma termina a duplicação de código e os programadores podem facilmente atualizar os seus projetos para a uma nova versão da *Framework*.

4.2 *Arquitetura das Aplicações*

As aplicações são compostas por um ou mais módulos configurados e carregados da forma indicada no processo de descrição da aplicação.

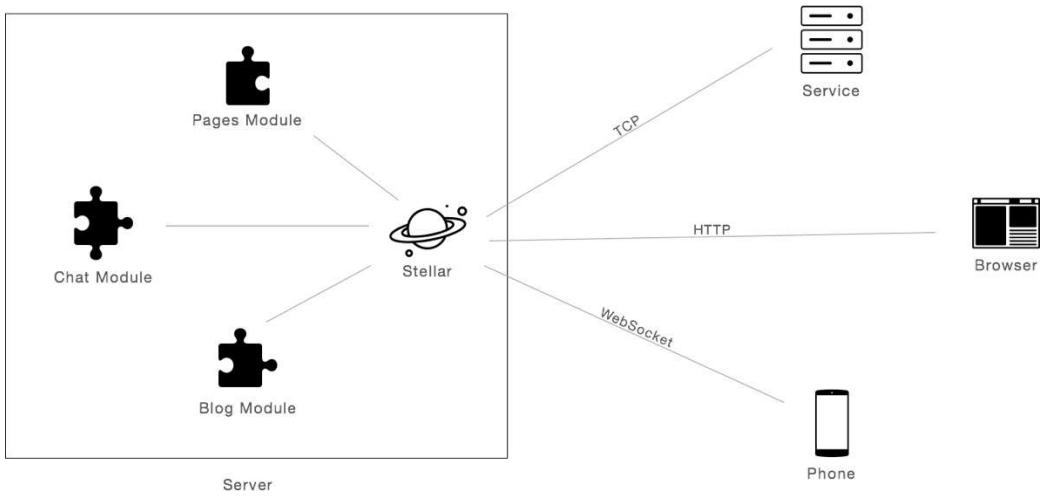

Figura 7 - Arquitetura de uma Aplicação

No esquema anterior (Figura 7) é presentada uma possível arquitetura de aplicação usando o Stellar. Neste caso, está a ser executada uma aplicação composta por três módulos (*page*, *chat* e *blog*), cada módulo carrega um bloco de lógica para a instância de execução do Stellar. Também se pode verificar, que as comunicações dos clientes com o servidor estão a ser efetuadas através de três tipos de protocolos diferentes em simultâneo, neste caso (HTTP, TCP e WebSocket).

Ao fazer a divisão da aplicação por módulos é ainda possível fazer a divisão das funcionalidades por áreas de atuação, permitindo que estas sejam mais facilmente portadas para outros projetos ou mesmo para facilitar a manutenção das mesmas. A *framework*, foi desenvolvida desta forma com vista a que novos intervenientes no projeto se possam adaptar mais facilmente, sem ter grandes dificuldades em perceber a estrutura da aplicação e as funcionalidades oferecidas. Isto é excelente para empresas com um grande número de colaboradores ou para empresas em rápido crescimento, em que estão constantemente a entrar novos colaboradores.

4.3 *Satellites*

Como descrito no subcapítulo anterior, o **Engine** por si só não faz absolutamente nada, a única lógica que este possui logo que a instância do Stellar é iniciada, é a de procurar pelos **Satellites** que compõem o *core* e a aplicação, estes é que irão ditar os próximos passos a serem dados.

Os **Satellites** são o nome dado aos componentes que permitem estender e subscrever as funcionalidades do Stellar. Através deste mecanismo é possível isolar as funcionalidades do *core* por áreas, facilitando a manutenção do *core*, tornando a *framework* extremamente extensível e permitindo aos programadores criarem módulos que permitam estender as funcionalidades base do Stellar.

Todo o *core* do Stellar é criado por **Satellites**, estes carregam as funcionalidades básicas da *framework*, mas o *core* não é o único local onde estes componentes podem existir, os módulos também podem fazer uso deles para carregar novas funcionalidades, subscrever existentes e até mesmo realizar tarefas assim que a *framework* inicie ou termine a sua execução.

4.3.1 Lifecycle de um Satellite

Todos os **Satellites** passam por uma série de fases de inicialização durante a execução de uma instância do Stellar. De seguida é explicado esse processo assim como o que deve ser realizado em cada uma das três etapas.

Figura 8 - Fases de um Satellite

Como pode ser visto na Figura 8 , as três fases de carregamento de um **Satellite**, são: **load**, **start** e **stop**. A etapa de carregamento é obrigatória, enquanto a inicialização e a paragem são opcionais. No caso de ser iniciada uma operação sem previsão de paragem, na etapa de inicialização, é recomendado efetuar a sua paragem na terceira etapa (stop), isto porque é possível reiniciar o servidor, sem que o processo de execução do Stellar tenha que ser terminado.

Na fase de carregamento do **Satellite** deve ser carregada toda a lógica no objeto da API de forma a tornar as funcionalidades públicas. Nenhum tipo de operação complexa deve ser realizada nesta fase, o carregamento deve ser feito o mais rapidamente possível. Na fase de inicialização devem ser iniciadas todas as tarefas continuas, como por exemplo *servers* ou algum outro tipo de *listener*. Por fim, na etapa de paragem todas as tarefas pendentes não concluídas devem ser terminadas, assim como todos os *servers*.

A arquitetura foi cuidadosamente concebida de forma a permitir uma enorme flexibilidade e reutilização. O **Core** é extremamente extensível e modificável, permitindo adicionar e editar as suas funcionalidades. Desta forma o Stellar está preparado para o futuro, pois é possível facilmente se adaptar para novos desafios.

5 Implementação

No decorrer deste capitulo vai ser abordada a implementação da *framework*. Irá começar por se falar da metodologia de desenvolvimento usada e depois será descrita a implementação dos diferentes componentes do Stellar.

5.1 *Metodologia de Desenvolvimento*

O Behavior-Driven Development (ou apenas **BDD**) foi a metodologia de desenvolvimento usada em todo o processo de implementação do Stellar. Esta metodologia combina as técnicas e princípios do Test Driven Development (**TDD**) [42] com as ideias do Domain-Driven Design (**DDD**) [43] e Object-Oriented Analysis and Design [44] para fornecer às equipas de desenvolvimento de *Software* e de gestão uma forma simples de partilharem ferramentas e processos a fim de colaborarem.

No caso aqui apresentado, e como em quase todos os projetos de desenvolvimento, foram adotadas quatro fases fundamentais de projeto de desenvolvimento de *software* que são: a fase de análise, a de *design*, a de implementação e a de testes.

Na fase de análise foram considerados alguns casos de uso acerca da forma como algumas das *frameworks* existentes se comportariam nesses mesmos cenários. Assim, foi possível encontrar os pontos fracos (descritos no capitulo 2) das *frameworks* atuais e tentar colmatá-los na fase de *design*.

Na fase de *design* foram obtidos os casos usados na fase anterior e adaptados para a *framework* aqui criada. Usando as conclusões obtidas na fase anterior foram criadas as estruturas das ações, tarefas e *listeners*. De seguida, foram escritos alguns blocos de código que representam a implementação das funcionalidades dos casos de uso, na nova *framework*. Isto permitiu perceber a naturalidade e a facilidade de implementação para o programador. Após essa etapa foi também criada uma estrutura (ao nível da pasta) e escolhida uma linguagem de programação para a implementação da solução. Ainda foi considerado o uso da linguagem RUST [45] para o desenvolvimento do *core*, mas o número de pessoas que domina esta linguagem é muito inferior ao número de pessoas que têm conhecimentos avançados em Node.js e JavaScript.

A fase de testes e implementação foram realizadas em conjunto de forma a seguir a metodologia BDD. Antes de implementar as funcionalidades foram escritos alguns testes que descrevessem o comportamento dessas mesmas funcionalidades (dai o **B** na sigla **BDD**). Esses testes foram realizados para as falhas, uma vez que não havia código para implementar os comportamentos descritos e só depois é que foi feita a implementação. Isto permite escrever testes completamente independentes da implementação, não caindo no comum erro de testar apenas os cenários que o programador tem em mente.

Mesmo assim, pode-se considerar que a fase de testes também teve uma etapa desacoplada da fase de implementação, onde foi desenvolvida uma aplicação exemplo para demostrar o uso da solução e validar a usabilidade de implementação para o programador.

5.2 *Implementação do Core*

A nível estrutural, o projeto encontra-se dividido de forma a que seja facilmente interpretado por novos programadores sem que tenham conhecimento prévio da estrutura do projeto. A raiz é composta por cinco pastas: `bin`, `exemple`, `src`, `staticFiles` e `test`.

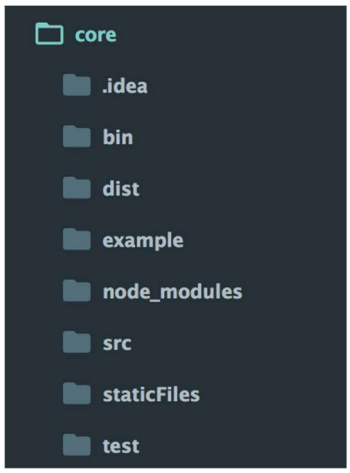

Figura 9 – Estura de Pastas do Core

A pasta *bin* contém o ponto de entrada para a ferramenta de linha de comandos assim como toda a sua lógica. A pasta *example*, contém um exemplo de uma aplicação, esta não é só útil para os programadores entrarem em contacto com uma aplicação funcional criada em Stellar, já que também contém ações que são usadas em alguns testes unitários. A pasta *src* é onde reside toda a lógica do *Core*, sendo composta pelo **Engine**, e é uma pasta que contém um conjunto de **Satellites** *Core* e três servidores prontos para produção. Existe também uma pasta com o nome *staticFiles* que contém alguns ficheiros estáticos que são usados em

algumas funcionalidades, como por exemplo, no processo de geração de documentação automática. Por fim, a pasta *test* que contém todos os testes unitários da plataforma.

5.3 *Carregamento de uma Aplicação*

Quando o **Engine** é iniciado começa por ler o ficheiro `manifest.json` (representado na Figura 10) que se encontra localizado na raiz do projeto. Este ficheiro descreve a aplicação e os módulos pela qual é composta. De seguida, todos os **Satellites** do **Core** são carregados assim como os módulos ativos. O carregamento dos módulos é feito de acordo com a ordem de declaração no ficheiro JSON.

```
{
  "name": "example",
  "version": "1.0.0",
  "description": "This is just an example API to show the integration with a SPA",
  "modules": [ "identify", "email" ]
}
```

Figura 10 - Ficheiro manifest.json que Descreve um Projeto

A fim de tornar os projetos e/ou módulos ainda mais adaptáveis a diferentes cenários de utilização é possível recorrer a um sistema de configurações para ditar o comportamento desejado para certas operações. Todo o sistema é desenvolvido à volta deste conceito de customização, sendo assim, o próprio *core* está desenvolvido de forma a que seja extremamente configurável. Os módulos podem alterar as configurações do *core* e de módulos de menos prioridade, assim como o programador/administrador de sistemas que irá configurar um projeto pode alterar essas mesmas configurações na pasta *config*, que se encontra na raiz do projeto. Assim, é possível alterar o comportamento de uma aplicação sem recorrer a codificação e sem afetar o comportamento de outras aplicações já disponibilizadas ou futuras variantes.

Na pasta *modules* (onde se encontram guardados os módulos da aplicação), pode ser criada uma nova pasta com o nome de *private*. Esta pode ser usada para adicionar lógica apenas àquela aplicação, se o programador achar que não existe necessidade da criação de funcionalidades especificas para a aplicação. Esta pasta tem a estrutura normal de um módulo e é carregada no final de todos os outros, desta forma é possível alterar o comportamento dos módulos já carregados.

5.4 *Ações*

As ações são os *building blocks* do Stellar, esta é a unidade básica da *framework*. Sendo o Stellar uma *framework* baseada em ações, significa que existe um repositório onde estão disponíveis todas as ações registadas no projeto. Uma ação representa uma pequena funcionalidade do projeto, as ações podem ser chamadas diretamente pelo cliente ou então por outro qualquer componente que tenha acesso à referência da **API**. Estas podem receber um conjunto de *inputs* que depois de processados devolvem um conjunto de *outputs*. Podem ser privadas, podendo apenas ser chamadas internamente e não pelo cliente, também podem ser subscritas por outros módulos, a não ser que se encontrem protegidas para que isso não aconteça.

Os programadores podem criar as suas próprias ações manualmente, criando um novo ficheiro na pasta *actions* do módulo em questão ou então, recorrer à ferramenta de linha de comandos que irá gerar o ficheiro e uma estrutura exemplo para a ação, de forma automática. Para isso basta correr o comando `stellar makeAction <action_name> --module=<module_slug>`.

As ações são carregadas para o **Engine** quando este é iniciado, seguindo a ordem de carregamento dos módulos.

```
exports.randomNumber = {
  name: 'randomNumber',
  description: 'Generates a random number',
  outputExample: {
    number: 0.40420848364010453
  },

  run (api, action, next) {
    // generates a random number
    var number = Math.random()

    // save the generated number on the response object
    action.response.number = number

    // also adds a formatted string
    action.response.formatedNumber = 'Your random number is ' + number

    // finish the action execution
    next()
  }
}
```

Figura 11 - Declaração de uma Ação

Na figura anterior (Figura 11) pode-se ver a estrutura de uma ação, esta ação é responsável por gerar um número aleatório e por fornecer ao cliente uma mensagem formatada com esse mesmo número.

As ações são compostas por duas propriedades obrigatórias, uma é a identificação da ação (`name`), a outra é a lógica da ação (`run`). Esta pode conter muita mais informação adicional, tal como, uma descrição (`description`), restrições aos valores de *input* (`inputs`), *middleware* (`middleware`) e um exemplo de *output* (`outputExample`). Com esta meta informação o Stellar é capaz de gerar documentação de forma totalmente automática sem intervenção humana (abordado em maior detalhe na secção 0). Isto é excelente para grandes equipas de forma a que todos os elementos possam facilmente conhecer todas as ações disponíveis no projeto, assim como os seus *inputs* e estrutura dos *outputs*, sem terem que andar constantemente a perguntar a outros elementos da equipa.

As ações são assíncronas e recebem uma referência para a API (funções partilhadas do **Engine** [**api**]), o objeto com o estado da conexão (`action`) no momento que a ação foi chamada e a função de `callback (next)`. Para completar a execução de uma ação basta chamar a função `next(error)`. Se existir um erro, tem que se assegurar que se passa uma instância de Error e não uma `String` como argumento da função `next`, isto fará com que seja gerada uma mensagem de erro e esta será depois devolvida ao cliente.

Devido à anatomia das ações, estas podem ser chamadas internamente pelo cliente ou através de outras ações sem a necessidade de alterações ou de escrever código específico para cada cenário de utilização. Isto é uma propriedade impossível da maioria, se não todas, as *frameworks* atuais.

5.4.1 Opções

Existe um conjunto de opções que podem ser adicionadas às ações, na tabela seguinte (Tabela 2) encontram-se todas as opções disponíveis.

Opção	Descrição
`name`	Identifica unicamente a ação. É recomendado que se use um *namespace* no nome das ações a fim de evitar conflitos, por exemplo: `auth.login`.
`description`	Descreve de forma extensa o comportamento e o objetivo principal da ação. Esta informação é importante para gerar a documentação automática.
`inputs`	Enumera os parâmetros de entrada da ação. É também através desta propriedade que é possível aplicar restrições aos valores de *input*.
`run(api, action, next)`	Lógica da ação, trata-se de uma função composta por três parâmetros de `input (api, action, next)`
`middleware`	Indica os *middleware* que devem ser executados na ação. Os *middleware* globais são automaticamente aplicados.
`outputExample`	Contém um objeto com um exemplo de uma resposta da ação. Este exemplo será anexado automaticamente à documentação gerada pelo Stellar.
`blockedConnecitionTypes`	Permite definir os tipos de conexões a serem bloqueadas para esta ação.
`logLevel`	Permite definir como a ação deve ser registada no sistema de *logs*.
`toDocument`	Por defeito esta opção está definida para `true`, caso contrario não será gerada documentação para esta ação.
`private`	Permite definir uma ação como privada. As ações privadas apenas podem ser chamadas internamente.

`protected`	Permite impedir que a ação seja subscrita por um módulo de maior prioridade.
`version`	Define a versão da ação.

Tabela 2 - Propriedades de uma Ação

Alguns dos metadados, como o caso do `outputExample` e o `description`, são usados para alimentar a documentação automática.

5.4.2 Versões

O Stellar suporta múltiplas versões da mesma ação permitindo manter ações com o mesmo nome, mas com funcionalidades diferentes/melhoradas. Esta funcionalidade é bastante útil quando existem muitas aplicações cliente a alimentarem-se da mesma **API**, podendo assim atualizar cada aplicação individualmente para a nova **API** sem interrupção do serviço nas demais aplicações.

As ações podem conter opcionalmente o parâmetro `version` para definir a versão da mesma. Na altura do pedido do cliente pode-se usar o parâmetro `apiVersion` para pedir uma versão especifica da ação, quando este parâmetro não existe é o Stellar que irá responder com a última versão da ação.

5.4.3 Declaração de Inputs

Na declaração das ações, opcionalmente, podem-se declarar os campos de *input*, esta declaração é feita usando a propriedade `inputs`. A declaração dos *inputs* não só permite perceber mais facilmente quais os parâmetros de entrada da ação, mas permite aplicar restrições a esses dados. Essas restrições podem ser `validators` já existentes no sistema (serão abordados num subcapítulo mais à frente, 5.13), uma expressão regular ou uma função que devolva um `Boolean` (em que `true` indica que o valor de `input` é valido). Por fim, também é possível converter o valor de *input* para um tipo específico (`Number`, `Decimal` e `String`) ou então usar uma função para formatar o valor.

A tabela de seguida (Tabela 3- Opções para os Valores de Input) apresenta as opções disponíveis para a declaração dos *inputs*:

Opção	Descrição
`required`	Este campo informa se o parâmetro é obrigatório.
`default`	Valor por defeito a ser usado no caso de o parâmetro não estar presente no conjunto de *inputs*.
`validator`	Valida o parâmetro contra um conjunto de restrições.
`convert`	Permite converter o valor de *input* para um outro tipo de dados ou formato.

Tabela 3- Opções para os Valores de Input

5.4.4 Objeto Action

O objeto `action`, passado como segundo parâmetro para a função `run` da ação, guarda o estado da conexão no momento em que a ação é chamada, nesse momento os *midlewares* de pré processamento já foram executados e os valores de *input* validados. A imagem abaixo (Figura 12) mostra algumas propriedades do objeto `action`.

```
action = {
    connection: connection,
    action: 'sum',
    toProcess: true,
    toRender: true,
    params: { action: 'sum', expression: '23;4' },
    actionStartTime: 123,
    response: {}
}
```

Figura 12 – Principais Propriedades do Objeto action

O objetivo da maioria das ações é realizar uma série de operações e alterar os dados da resposta `data.response`, que posteriormente serão enviados para o cliente.

Os programadores podem ter acesso a informações mais avançadas sobre a ligação ao cliente, também podem modificar as propriedades da conexão como por exemplo os *headers* do HTTP a serem enviados na resposta. Esta alteração pode ser realizada acedendo à propriedade `connection` do segundo parâmetro (`action`) que é passado para a função que contem a lógica da ação.

Caso o programador não queira que o *Engine* envie uma resposta para o cliente (por exemplo, já foi enviado um erro), apenas terá que alterar a propriedade `data.toRender` para `false`.

5.4.5 Chamadas Internas

Com vista a melhorar o reaproveitamento de código e fazer uma melhor separação das ações que partilham parte da mesma lógica, o Stellar implementa um mecanismo que permite fazer chamadas internas a ações. Isto quer dizer que se pode extrair parte da lógica de uma ou mais ações para ações mais simples, podendo essa mesma lógica ser usada por outras ações. Desta forma, a partir da composição de ações simples podem-se criar ações mais complexas sem tornar a leitura do código difícil ou mesmo dificultar a gestão das aplicações e módulos. Afim de tornar o código ainda mais simples foi usado o padrão Promise [46] para contornar o problema do *callbackhell* [47] do JavaScript.

Ações Privadas

Por vezes serão criadas ações que os programadores não querem que sejam invocadas pelos clientes, ou porque são apenas para uso interno, ou não realizam nenhuma operação relevante para o cliente ou são ações mais simples ou críticas que não devem ter exposição pública. Para isso o Stellar permite definir uma ação como privada, fazendo com que esta apenas possa ser chamada internamente. Para tornar uma ação privada apenas é necessário definir a propriedade `private`, da ação, para `true`.

5.5 *Modelos*

Os modelos (*models*) são uma forma de implementar a estrutura de dados de um determinado recurso guardado na base de dados. O sistema de *models* do Stellar permite especificar um modelo segundo o formato usado pelo Mongose [48], esse modelo pode depois ser acedido durante a execução de uma ação/tarefa de forma a permitir um acesso facilitado à base de dados.

Como referido anteriormente, o Stellar faz uso do Mongoose para o seu sistema de modelos. O Mongoose fornece uma forma fácil baseada em *schemas* para modelar os dados da aplicação. Isto inclui *type-casting*, validação, construção de *queries* e outras funcionalidades diretamente *out-of-the-box*. Este foi escolhido devido à necessidade do tempo reduzido para a sua implementação no sistema, algo crucial para a alcançar todos os outros objetivos a tempo da entrega deste projeto.

Os *models* devem ser criados na pasta *models* que se encontra na raiz de cada módulo. Existem um comando para facilitar a criação destes modelos, será discutido em maior detalhe na secção 5.16.4.

5.6 *Middleware*

Os programadores podem criar *middlewares* que podem ser aplicados antes e depois da execução das ações. Existem dois tipos de *middlewares*, os globais, que são aplicados a todas as ações, ou então aplicados de forma individual a cada ação utilizando a propriedade `action.middleware`. Cada *middleware* tem um nome e opcionalmente pode conter uma prioridade que irá definir a ordem de execução dos *middlewares*.

Existem três tipos de *middleware*: para ações, conexões e *chat*. Cada um é distinto dos outros e operam em diferentes partes do *ciclo de vida* do pedido do cliente.

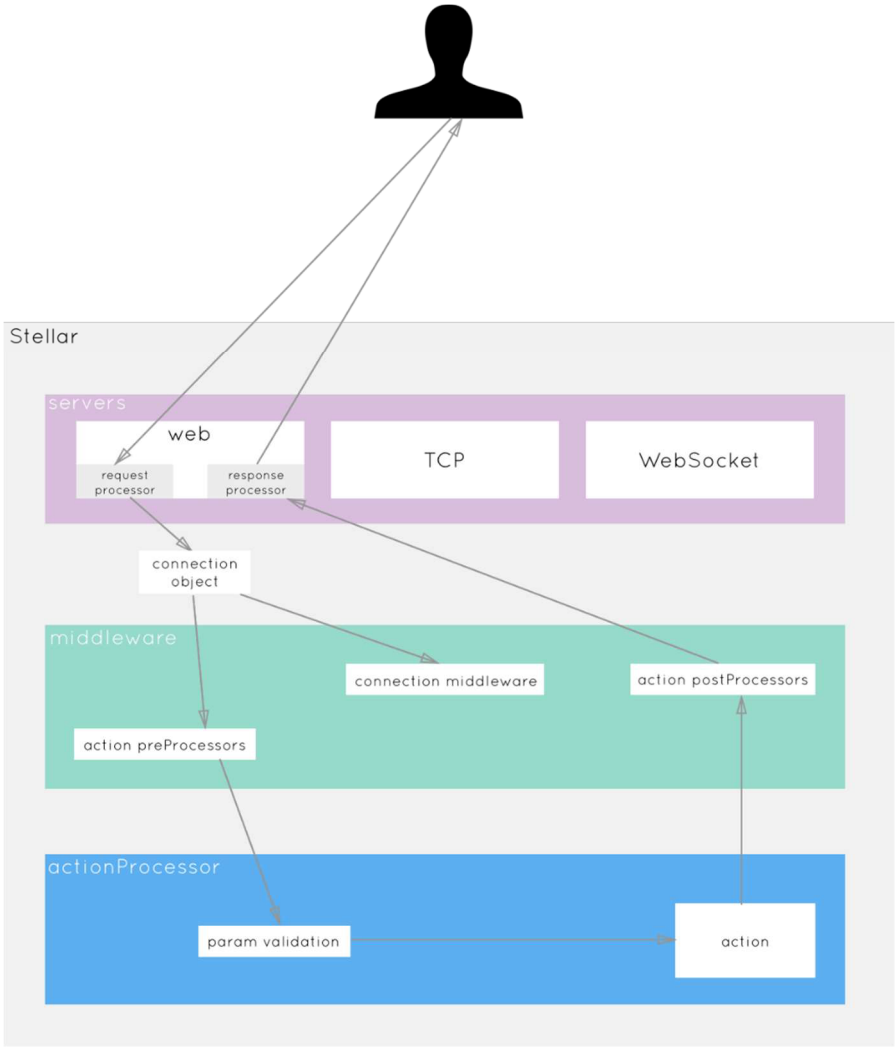

Figura 13 - Request Flow

Como pode ser visto na Figura 13, existem diferentes locais onde pode ser executado um *middleware*. A tabela (Tabela 4 - Middleware Disponíveis) apresentada de seguida mostra os diferentes *middlewares* disponíveis no Stellar:

Area de Atuação	Nome do Middleware
Quando um cliente se liga	
Middleware de conexão	`create`
O cliente faz o pedido de uma ação	
Middleware de ação	`preProcessor`
Middleware de ação	`postProcessor`
Um cliente junta-se a uma sala de chat	
Middleware de chat	`join`
O cliente envia uma mensagem para a sala de chat	
Middleware de chat	`say`
Middleware de chat	`onSayReceive`
O cliente faz um pedido de desconexão (quit)	
Middleware de chat	`leave`
Middleware de conexão	`destroy`

Tabela 4 - Middleware Disponíveis

5.6.1 Middleware de Ação

O Stellar oferece *hooks* para se executar código antes e depois de algumas ações, este é o local apropriado para adicionar lógica relacionada com a autenticação ou para validação do estado de um determinado recurso.

5.6.2 Middleware de Conexão

É possível criar *middlewares* para reagir à criação e destruição de todas as conexões. Ao contrário dos *middlewares* de ações, estes não bloqueiam o pedido até que a execução seja finalizada, sendo assíncronos.

É preciso ter em atenção que algumas conexões persistem (TCP e WebSocket) enquanto que outras apenas existem durante um único pedido (HTTP). Pode-se no entanto, inspecionar o valor `connection.type` de forma a determinar qual o tipo de ligação que está a ser criada ou destruída.

5.6.3 Middleware de Chat

Por último, existe o *middleware* para o *chat*. Este tipo de *middleware* é ativado quando um cliente se junta, quando sai de uma sala, ou quando comunica dentro de uma sala de *chat*. Existem quatro tipos de *middleware* para cada etapa: `say`, `onSayReceive`, `join` e `leave`.

5.7 *Tarefas*

As tarefas são operações executadas em *background,* independentes do pedido do cliente sendo iniciadas através de uma ação ou pelo próprio servidor. Com o Stellar não existe a necessidade de correr um *deamon* separadamente para processar os trabalhos. O Stellar faz uso do pacote **node-resque** [49], para armazenar e processar tarefas usando um servidor Redis [37].

As tarefas podem ser processadas de três formas: normal, com atraso ou periodicamente. As tarefas normais são inseridas na *queue* e processadas uma a uma pelo `TaskProcessor`. As tarefas com atraso são inseridas numa *queue* especial em que as tarefas são processadas num certo momento no futuro (definido por um *timestamp* em milissegundos ou *miliseconds-from-now*). Por fim, as periódicas são tarefas com atraso, mas que são executadas numa certa frequência (por exemplo, a cada 24 horas), este tipo não recebe parâmetros de *input*.

Por vezes os *workers* podem falhar de uma forma tão severa que não seja possível notificar o servidor Redis que este vai sair da *poll* (isto acontece inúmeras vezes em PAAS [*Platform As A Service*]). Quando isto acontece é necessário extrair a tarefa do *worker* que morreu, inseri-la numa *queue* especial para as tarefas falhadas, para ser reprocessada mais tarde e por fim remover o *worker*.

As tarefas podem ser criadas manualmente criando um ficheiro na pasta `tasks` dos módulos ou recorrer ao comando `stellar makeTask <taskName> --module=<moduleName>`. A tabela abaixo descreve as propriedades que compõem uma tarefa:

Propriedade	Descrição
`name`	Nome único da tarefa. Recomenda-se usar um *namespace* para prevenir colisões, por exemplo: **blog.sendNewsletter**
`description`	Descrição detalhada da finalidade da tarefa
`queue`	Por defeito a tarefa é atribuída à *queue* **default** (este valor pode ser subscrito quando a tarefa é executa manualmente)
`frequency`	Este valor define a frequência com que a tarefa é executada, em milissegundos. Quando este valor não existe a tarefa não é executada periodicamente.
`plugins`	Esta propriedade permite definir *plugins* do pacote node-resque.
`pluginsOptions`	Trata-se de uma *hash* com opções que são passadas para os plugins.
`run`	Função que contém as operações a serem realizadas pela tarefa.

Tabela 5 - Propriedades das Tarefas

5.8 *Eventos*

O Stellar tem um sistema de eventos que permite subscrever e ficar à escuta de eventos na aplicação. Isto é útil para manipular dados durante a execução ou estender funcionalidades, adicionando novos comportamentos à lógica existente. Os *listeners* são guardados na pasta `listeners` dos módulos. Alguns componentes do sistema fazem uso deste sistema para permitir aos módulos estender as suas funcionalidades, um exemplo disso é o sistema de

modelos, que permite adicionar novos campos a *schemas* de outros módulos. Este sistema também pode ser utilizado pelos criadores de módulos.

Os ficheiros podem ser criados manualmente ou através da ferramenta de linha de comandos usando o comando: `stellar generateEvent <eventName> --module=<moduleName>`

5.9 *Servers*

Os **Servers** são responsáveis por receber os pedidos feitos pelos clientes e juntar todos os valores de *inputs*. Após isto, estes dados são expedidos para o `ActionProcessor` que irá interpretar o pedido e executar a ação. Logo que terminado, é pedido ao **Server** que recebeu a ação para enviar a resposta de volta ao cliente. O Stellar, por defeito, vem equipado com três tipos de servidores, HTTP, WebSocket e TCP, isto sem a necessidade de instalação de módulos adicionais. Uma outra vantagem é que estes **Servers** podem ser executados em simultâneo sem a necessidade de executar instâncias adicionais do Stellar por cada um deles. Todos os três tipos de **Server** têm suporte a ligações seguras através de TLS, apenas será necessário ativar a funcionalidade nas configurações e indicar os certificados a serem usados para cifrar as comunicações.

5.9.1 HTTP

O servidor HTTP permite ao cliente fazer diferentes tipos de chamadas à **API** para pedir a execução de uma ação. Os pedidos podem ser feitos por GET, POST ou através de rotas RESTfull de onde podem retirar todo o poder do protocolo HTTP recorrendo a diferentes tipos de pedidos.

A passagem de parâmetros não se encontra restrita apenas a uma única forma. Esta passagem pode acontecer através de *query strings*, *form-data* ou através de dados *raw,* utilizando o formato JSON. No caso de se tratar de um pedido POST, é possível usar múltiplas formas em simultâneo, como por exemplo *query strings* e *raw data*. As *query strings* têm sempre prevalência em relação às demais.

5.9.2 WebSocket

Através do servidor WebSocket é possível criar ligações em tempo real bidirecionais. Este tipo de servidor torna-se interessante quando existe a necessidade de criar funcionalidades críticas que necessitem que os dados sejam o mais atualizados possível. Para recorrer ao uso deste servidor é necessário que o servidor de HTTP esteja ativo uma vez que é este o

responsável por responder aos pedidos de conexão dos clientes. Após a receção do pedido é feita uma troca de protocolos.

5.9.3 TCP

A possibilidade de usar um servidor TCP torna-se interessante em cenários onde os pedidos estão a ser feitos por outro serviço ou através de uma APP nativa, facilitando assim a implementação da mesma reduzindo ao número e tamanho do trafego necessário para o funcionamento além de permitir ainda a criação de uma ligação bidirecional em tempo real.

5.10 *Módulos*

Uma das grandes vantagens do Stellar é a sua capacidade de modularização, capaz de aumentar a independência dos vários componentes que fazem parte de um projeto e aplicar uma divisão lógica entre eles. Os módulos têm uma estrutura bem definida fornecendo uma boa linha de orientação para os programadores, além disso, podem ser facilmente adicionados e removidos de um projeto. Um módulo é um conjunto de funcionalidades de uma determinada área, tornando-as independentes das restantes funcionalidades do projeto. Os módulos são constituídos pelo seguinte conjunto de pastas e ficheiros: `actions`, `tasks`, `middleware`, `models`, `satellites`, `lib` e `manifest.json`.

A pasta `actions` contêm os ficheiros que implementam as ações, esta é a principal fonte de lógica dos módulos. A pasta `tasks` contém as tarefas do módulo, estas podem ser periódicas caso tenham uma frequência, ou então podem apenas ser chamadas durante a execução do projeto, durante uma ação ou de uma outra tarefa. Para declarar um *middleware* a ser aplicado nas ações pode ser guardado na pasta `middleware`. A pasta `models` contém a declaração dos *schemas* para guardar dados de forma persistente, estes *schemas* devem seguir o formato do Mongoose [48] (como já referido na secção 5.5). Na pasta `satellites` podem ser guardados os **Satellites** pertencentes ao módulo, estes serão carregados de acordo com a prioridade dos mesmos e não pela ordem de execução do módulo. Na pasta `lib` estão presentes todos os ficheiros com a lógica que não façam parte das pastas descritas anteriormente, como por exemplo funções utilitárias ou *wraps* de bibliotecas externas. Finalmente, o ficheiro `manifest.json` que descreve o módulo com um identificador único, um nome mais fácil para a leitura humana, uma breve descrição do seu objetivo principal, uma ordem de carregamento e as suas dependências (Node Package Manager (npm) e/ou outros módulos do Stellar).

Toda a plataforma foi concebida para ser extremamente extensível e personalizável. Uma vez que os módulos têm uma ordem de execução isso pode ser usado para subscrever ações já carregadas por outros módulos, ou então fornecer ações para serem usadas por módulos de maior prioridade.

5.10.1 Descrição de um Módulo

A descrição dos módulos é feita através do ficheiro `manifest.json`. Este ficheiro é composto pelas seguintes propriedades abaixo descritas:

Nome da Propriedade	Descrição
`Id`	Identificador único do módulo
`name`	Nome do módulo
`version`	Versão do módulo, deve seguir o padrão **Semantic Versioning** [50]
`description`	Descrição detalhada do principal objetivo do módulo
`dependencies`	`Array` com o identificador dos módulos aos quais o módulo depende.
`npmDependencies`	Identificador dos pacotes npm que o módulo depende
`author`	Identificação do autor do módulo, por exemplo Gil Mendes <gil00mendes@gmail.com>
`website`	Endereço *web* relacionado com o módulo

Tabela 6 - Propriedades da Descrição dos Módulos

5.11 *Cluster*

O Stellar já vem equipado de raiz com um mecanismo de *cluster*. O objetivo do cluster é ajudar a criar um grupo de servidores, de forma fácil, que partilhem o mesmo estado e lógica, a fim de serem capazes de processar um maior número de pedidos e de executar tarefas em simultâneo. O uso desta funcionalidade é extremamente simples, sendo apenas necessário executar o comando *stellar startCluster* na raiz do projeto e o Stellar encarrega-se de tudo o

resto. Também é possível definir o número de *workers* usando o parâmetro *--workers*, sendo no entanto recomendado que não seja superior ao número de *threads* lógicas do CPU.

Ao usar a funcionalidade de *cluster* deixa de existir a necessidade de recorrer ao PM2 [51] para gerir os processos do Node.JS. A implementação é capaz de recuperar uma instância após um *crash* sem que o serviço seja interrompido e a informação perdida, já que uma nova instância é automaticamente criada assim que alguma em execução deixe de funcionar. Uma vez que a informação é partilhada por todas as instâncias que compõem o *cluster*, não existe o risco de perder dados ou duplicar tarefas.

Os subsistemas que operam em *cluster*, tal como *cache* e *chat rooms*, recorrem a um servidor Redis para centralizarem a informação e através da implementação de um sistema de semáforos é possível aceder a um recurso sem correr o risco de uma outra instância o alterar ou eliminar.

5.12 *Chat*

O Stellar está equipado com uma solução de salas de *chat*, que pode ser usada com todas as conexões persistentes (TCP e/ou WebSocket). Existem métodos para criar e gerir as salas de *chat* e os utilizadores dessas salas. Este sistema pode ser usado para diferentes finalidades, como por exemplo: atualização de dados em tempo real, propagação de informação de forma rápida entre clientes que estejam ligados e até mesmo para criação de jogos *multiplayer* (uma vez que estes necessitam de constante partilha de informação entre todos os jogadores).

Os clientes comunicam com as salas através de *verbs*. Os *verbs* são comandos curtos que permitem alterar o estado da conexão, como juntar-se ou sair de uma sala. Os clientes podem ainda estar em diferentes salas ao mesmo tempo.

Os *verbs* mais relevantes são:

- `roomAdd`
- `roomLeave`
- `roomView`
- `say`

Esta funcionalidade pode ser usada *out-of-the-box* sem que seja necessária qualquer instalação de pacotes adicionais, configuração ou programação. Por defeito, é criada uma sala com o nome “defaultRoom”. Quando o servidor de WebSocket está ativo é gerado um

script de cliente, este pode ser usado em aplicações *web* para facilitar a chamada de ações e de comunicação com as salas de *chat*, sem que seja necessário o desenvolvimento de uma interface de ligação entre o cliente e o servidor.

Não existem limites ao número de salas que podem ser criadas, mas é necessário ter em mente que cada sala guarda informação no Redis, assim existe carga por cada ligação criada.

5.13 *Validators*

Os **Validators** são um conjunto de "validadores" pré implementados no *core* do Stellar, que permite aos programadores validar os parâmetros de *input* das suas ações com enorme facilidade. Todo o trabalho de validação e resposta de erro fica ao cargo do Stellar.

Alguns "validadores" podem receber parâmetros que serão usados para dar indicações ao mesmo de como deve ser aplicado ao valor de *input*. Os parâmetros seguem a seguinte *syntax*: "`validatorName:param1,param2,...`" Os dois pontos (`:`) declaram o inicio da especificação dos parâmetros e as virgulas (`,`) informam que irá existir outro parâmetro a seguir.

Os "validadores" podem ser aplicados em conjunto a um único parâmetro, seguindo a *syntax*: "`valiador1|validator2:param1`". Os *pipes* (`|`) informam a existência de outro validador declarado após este caracter para ser aplicado ao *input*.

Pode encontrar no Anexo I uma tabela com todos os "validadores" implementados no sistema. Esta tabela contem os seus nomes, parâmetros e uma descrição resumida do seu funcionamento.

5.14 *Documentação Automática*

O Stellar permite gerar documentação das ações de forma completamente automática. A informação necessária é extraída através da declaração das propriedades das ações. Para fazer com que não seja gerada uma página de documentação para uma dada ação adiciona-se a propriedade `toDocument` e define-se a mesma para `false`, na ação em questão. Caso se queira desativar para todas as ações define-se a configuração `api.config.general.generateDocumentation` para *false*. Para aceder à documentação basta visitar a página `docs/index.html` no endereço do servidor HTTP.

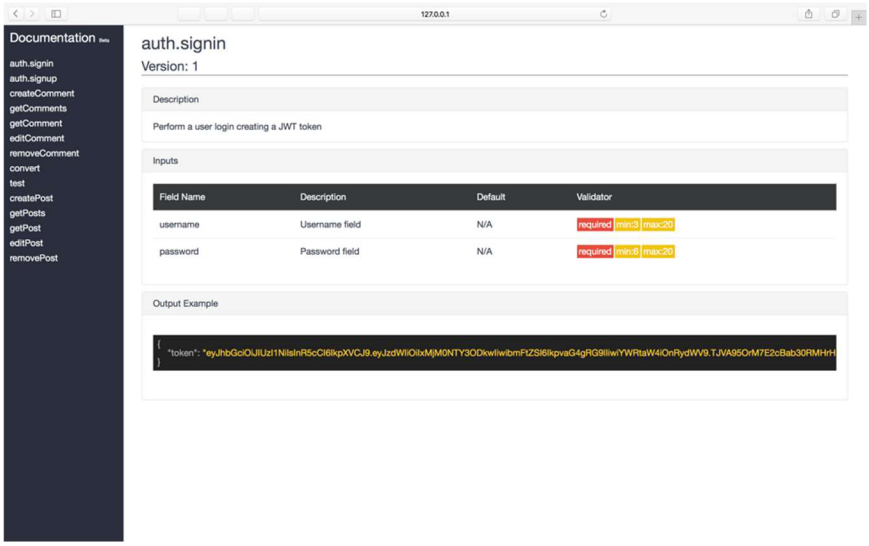

Figura 14 - Documentação Automática

A figura anterior (Figura 14 - Documentação Automática) apresenta um ecrã com o exemplo de uma página de documentação gerada automaticamente. Na barra lateral encontram-se todas as ações existentes na plataforma, inclusivamente as ações privadas. Já na secção à direita podemos ver os detalhes da ação selecionada, como por exemplo: nome, descrição, campos de *input* e as suas restrições, bem como um exemplo de *output*. Quando as ações têm múltiplas versões todas elas são apresentadas.

5.15 *Modo de Desenvolvimento*

O modo de desenvolvimento, tal como o próprio nome indica, é um modo especial para facilitar o desenvolvimento dos módulos e aplicações no Stellar. Ao alterar os ficheiros de rotas, tarefas, ações e modelos, o servidor consegue substituir essa lógica em memória assim que seja detetada uma alteração no sistema de ficheiros. Deste modo não terá que estar

constantemente a parar e a reexecutar o servidor a cada alteração realizada. Alterações mais severas, como configurações e **Satellites** obrigam a que o servidor reinicie por completo, mas tudo é executado de forma automática.

Quando o modo de desenvolvimento está ativo o Stellar recarrega as ações, tarefas, modelos, configurações e **Satellites** assim que eles sejam modificados, tudo *on the fly*. Uma vez que o Stellar faz uso do método `fs.watchFile()` o recarregar pode não funcionar em todos os sistemas operativos / sistemas de ficheiros. Novos ficheiros não serão carregados, apenas os ficheiros com que a instância foi iniciada é que são monitorizados. Apagar um ficheiro pode causar num *crash* da aplicação, o Stellar não tem mecanismos para detetar esse tipo de eventos. Se o valor da frequência em que uma tarefa periódica é executada (`task.frequency`) for alterada, será usado o valor antigo até que a tarefa seja de novo "dispultada". Ao alterar configurações e ou `http://localhost:4000/guide/satellites.html`, será feito um *reboot* total ao servidor e não apenas dos ficheiros alterados.

5.16 *Ferramenta de Linha de Comandos*

De seguida são introduzidos os comandos que compõem o Stellar, estes foram concebidos com vista à otimização do *workflow* de desenvolvimento e facilitar a criação das estruturas com o objetivo de não haver necessidade de reler a documentação para criar cada novo componente ou para rever as melhores práticas.

Os subcapítulos seguintes apresentam cada comando existente no Stellar através da descrição do funcionamento de cada um, as suas opções e mostrando a forma como estes devem ser invocados.

5.16.1 Inicialização

O comando `init` permite ao programador inicializar uma pasta vazia com a estrutura de pastas e ficheiros de um projeto *Stellar*. Para a execução deste comando apenas é necessário especificar o nome do projeto e estar dentro da pasta onde pretende inicializar um novo projeto. Opcionalmente também é possível especificar a versão do projeto (por defeito é 1.0.0). A partir do momento em que é executado o comando já é possível iniciar uma instância do Stellar no diretório inicializado, começar a adicionar os módulos desejados e desenvolver toda a lógica das funcionalidades necessárias para implementar o projeto.

Estrutura

O comando `init` segue a seguinte estrutura: `stellar init –name=<projectName> [--options]`

Opções

A tabela seguinte apresenta as opções disponíveis para o comando `init`:

Parâmetro	Descrição
`name`	Parâmetro obrigatório que permite especificar o nome do projeto a ser criado.
`version`	Parâmetro opcional que permite indicar a versão do projeto a ser criado. Por defeito assume o valor de 1.0.0

Tabela 7 - Parâmetros para o Comando Init

Execução

O comando `run` permite criar uma nova instância de execução do Stellar. Para tal apenas é necessário que o comando seja executado na pasta que contém o projeto.

Estrutura

O comando `run` segue a seguinte estrutura: `stellar run [--options]`

Opções

A tabela seguinte apresenta todas as opções disponíveis para o comando `run`:

Parâmetro	Descrição
`clean`	Quando este parâmetro está presente, todos os ficheiros temporários de dependências do NPM são apagadas.
`port`	Define a porta onde o servidor HTTP irá ficar à escuta por pedidos dos clientes.

`prod`	Ativa o modo de produção. Isto quer dizer que desliga todos os mecanismos de ajuda ao desenvolvimento, como a escuta por alterações nos ficheiros das ações e configurações para reiniciar o servidor.

Tabela 8 - Parâmetros para o Comando run

5.16.2 Cluster

O comando `cluster` permite iniciar o Stellar como um *cluster*. Isto quer dizer que serão iniciados tantos nós do Stellar, quanto o número de *cores* do CPU. Por agora este comando não possui nenhum tipo de opções.

Estrutura

O comando *cluster* segue a seguinte estrutura: `stellar cluster`

5.16.3 Criar uma Ação

O comando `makeAction` permite criar uma nova ação. Para que este comando funcione o programador deve executar o comando na raiz do projeto, indicando o nome da ação e o módulo onde esta deverá ser criada. Este comando gera um ficheiro com o nome passado como parâmetro e inicializa-o com uma estrutura padrão de uma ação. Caso a ação a ser criada já exista o comando irá terminar com um erro a informar isso mesmo. Caso o parâmetro **force** esteja presente esta ação é substituída.

Estrutura

O comando `makeAction` segue a seguinte estrutura: `stellar makeAction <actionName> --module=<moduleName> [--options]`

Opções

A tabela seguinte apresenta as opções que podem ser usadas com o comando `makeAction`:

Parâmetro	Descrição
`module`	Id do módulo onde é pretendido que a ação seja criada.
`force`	Caso esta opção esteja presente e exista uma ação com o mesmo nome da que está a ser criada, esta será substituída por uma nova.

Tabela 9 - Parâmetros para o Comando makeAction

5.16.4 Criar um Model

O comando `makeModel` permite criar um novo *model***.** Este comando deve ser executado na raiz do projeto e deverá ser indicado o nome do *model* e o módulo onde este deve ser criado. Junto com este comando podem ser usados uma série de parâmetros que permitem gerar rotas e ações para o *model* criado.

Estrutura

O comando `makeModel` segue a seguinte estrutura: `stellar makeModel <modelName> --module=<moduleName> [--options]`

Opções

A tabela seguinte apresenta todas as opções disponíveis para usar com o comando `makeModel`.

Parâmetro	Descrição
`module`	Este parâmetro é obrigatório, indica o id do módulo onde os ficheiros devem ser gerados
`crud`	Esta opção faz com que sejam geradas ações para a manipulação dos dados do *model* gerado
`actionName`	Permite subscrever o nome do ficheiro de ação gerado
`rest`	Gera rotas segundo o padrão *restfull* para as ações geradas

Tabela 10 - Parâmetros para o Comando makeModel

5.16.5 Criar uma Tarefa

O comando `makeTask` permite gerar uma nova tarefa no módulo indicado. Este comando não possui nenhuma opção opcional, apenas é obrigatório passar o nome do modulo onde deve ser gerada a nova tarefa. O ficheiro gerado contém a estrutura padrão de uma tarefa.

Estrutura

O comando `makeTask` segue a seguinte estrutura: `stellar makeTask <taskName> --module=<moduleName>`

6 Validação da Solução

A validação da solução desenvolvida é extremamente importante, é através de testes unitários e aplicações exemplo, que se consegue validar se as funcionalidades desenvolvidas desempenham as funções para as quais foram concebidas. Desta forma, assegura-se que as versões de produção disponibilizadas para uso contêm o menor número de erros possível e que não se comportam de forma inesperada para não frustrar os programadores que os poderia levar a abandonar a *framework*.

Com o objetivo de atingir esta meta de qualidade, foram definidas um conjunto de etapas a serem executadas no processo de validação. Primeiro, foram definidas um conjunto de diretivas que devem ser estritamente seguidas quando o programador do *core* está a escrever novo código. Este deverá respeitar estas regras de forma a manter um estilo conciso e uma estrutura constante, mantendo o código o mais legível e simples possível. Na segunda etapa são executados um conjunto de quase 200 testes unitários para verificar se as funcionalidades desenvolvidas cumprem com os seus objetivos. Todas estas etapas de teste assim como a medição da percentagem de cobertura (através da ferramenta coveralls [52]) são efetuadas cada vez que é feito um *push* para o repositório Git. Quando é efetuado um *push*, o Travis [53] é despoletado para executar todo o processo de testes. A percentagem de cobertura deve ser sempre superior a 70% para que seja considerada aceitável.

Neste capítulo, são apresentadas as formas como a solução desenvolvida foi validada. Para verificar que tudo funciona como pretendido foram implementados um grande conjunto de testes unitários segundo o processo de desenvolvimento de *software Behaviour Driven Development* (ou mais largamente conhecido como BDD). Além desse conjunto de testes, foi também desenvolvida uma aplicação web que interage com a API implementada recorrendo ao Stellar.

6.1 *Testes*

Para a validação da solução desenvolvida foram implementados um conjunto de testes unitários que permitem validar o comportamento das funcionalidades. Para isso, foi adotado o processo de desenvolvimento de *software* BDD. O BDD foi escolhido em alternativa ao TDD para tornar os testes mais expressivos, levar o foco para o comportamento que estes têm e não tanto para a parte técnica de como tudo foi desenvolvido. Além disso, é facilitada a leitura dos testes e o envolvimento de novas pessoas no projeto.

Os testes encontram-se organizados em 10 grandes grupos, sendo eles: API, cache, erros, eventos, *hash*, *middleware*, "validadores", HTTP, *WebSocket* e TCP. No anexo II é possível encontrar a listagem completa com todos os testes unitários, nomes, descrição e resultados esperados.

6.1.1 API

Neste grupo são testados alguns comportamentos gerais do *core*. A API é um ponto importante da *framework* uma vez que todos os Satellites vão disponibilizar aqui as suas funcionalidades, além de que será o objeto acedido por todos os componentes que pretendam usar essas mesmas funcionalidades.

6.1.2 Cache

Neste grupo é testado o comportamento do sistema de *cache*. A *cache* é um sistema crítico, principalmente quando o Stellar é executado num ambiente de cluster uma vez que é através deste que os diferentes nós comunicam.

6.1.3 Error

Neste grupo testam-se as funções de erro que são chamadas e executadas quando ocorrem erros específicos e verifica-se também se são capazes de fazer a correta serialização dos erros. O incorreto funcionamento destas funcionalidades leva à dificuldade da identificação de erros de programação, por parte dos programadores.

6.1.4 Events

No grupo *events* é validado o comportamento das funções que permitem adicionar e despoletar um evento durante a execução, de forma a manipular a informação que dele faz parte ou de executar uma determinada operação. Este é outro módulo fundamental para atingir o objetivo da máxima modularização, uma vez que é através deste sistema de eventos que é possível estender e modificar o comportamento das funcionalidades presentes em outros módulos.

6.1.5 Hash

Neste grupo são validadas as funcionalidades de *hash* sendo de extrema importância que este sistema funcione em pleno e de acordo com os padrões já que está intrinsecamente ligado com os sistemas de segurança da plataforma.

6.1.6 Middleware

Este grupo valida se os *middlewares* são executados nos momentos corretos e se estes aplicam as alterações necessárias às ações. É um componente de elevada importância, o seu incorreto funcionamento pode levar a comportamentos inesperados. O nível de prioridade dos *middlewares* e a sua correta execução é validada a fim de tudo acontecer como descrito na documentação do Stellar. Não devem acontecer situações inesperadas para os programadores.

6.1.7 Validators

Os testes deste grupo têm como objetivo validar se os *validators* implementados têm o comportamento esperado. Assim como outros subsistemas descritos anteriormente, os *validators* são uma parte importante do ciclo de pedidos do cliente já que os programadores confiam nos mesmos para aplicar restrições aos valores de *input* nas ações.

6.1.8 HTTP

Os testes que compõem este grupo permitem validar o funcionamento do servidor de HTTP.

6.1.9 TCP

Este tipo de testes validam o comportamento do servidor TCP.

6.1.10 WebSocket

Neste grupo estão presentes os testes que permitem verificar se o servidor de WebSocket tem o funcionamento esperado.

7 Caso de Estudo

Como uma diferente forma de validação e a fim de disponibilizar um exemplo funcional que se assemelhe a um caso real, foi desenvolvida uma aplicação *web* muito simples. Esta aplicação permite aos utilizadores iniciarem sessão através das redes sociais, utilizando o sistema de autenticação Auth0 [54]. Quando tiverem sessão iniciada, os utilizadores podem criar perguntas e responder a outras perguntas, sejam estas do próprio ou de outros utilizadores.

A aplicação *web* foi desenvolvida utilizando a biblioteca Vue.js [55] e claro está, a *framework* aqui desenvolvida, o Stellar. O Vue foi usado para a criação da aplicação cliente (*web app*) e, obviamente, como esperado, o Stellar foi usado para o desenvolvimento da API de onde a *web app* se irá alimentar.

De forma a fazer um pequeno enquadramento, Vue.js é uma biblioteca para a construção de interfaces *web*. A primeira *release* a público aconteceu em fevereiro de 2014 e desde aí a comunidade tem crescido quase exponencialmente existindo já centenas de empresas a utilizar esta biblioteca em produção sendo dezenas deles gigantes da informática, tais como o Alibaba [56] e a Xiaomi [57] [58].

Para a ligação ao Stellar foi usada a biblioteca cliente gerada pelo próprio *core* sempre que este é iniciado. Trata-se de uma ligação em tempo real sob o protocolo WebSocket. Todas as comunicações são efetuadas em tempo real e novas perguntas são apresentadas automaticamente a todos os utilizadores sem a necessidade de fazer *refresh* à aplicação *web*.

A imagem apresentada de seguida (Figura 15), mostra a interface principal onde constam todas as perguntas existentes na plataforma. No topo da página existe também um formulário que permite aos utilizadores com sessão criarem uma nova pergunta. Logo que este formulário seja submetido a uma nova pergunta irá aparecer automaticamente em todos os clientes ligados.

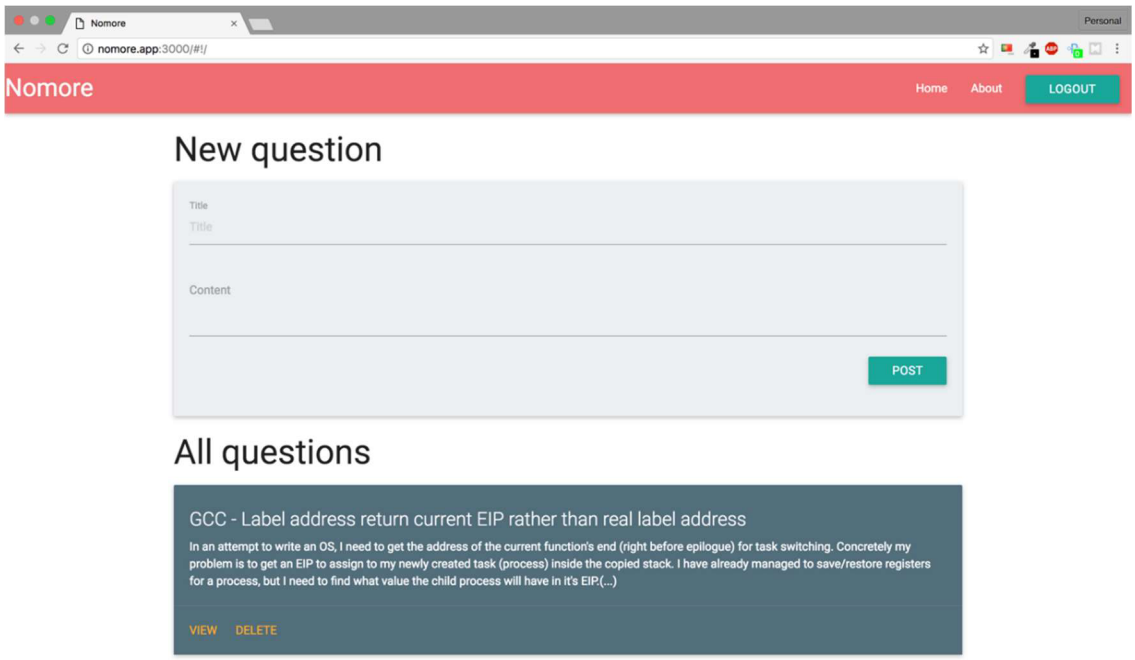

Figura 15 - Aplicação Exemplo

O projeto encontra-se disponível no GitHub (*https://github.com/stellarFw/example*) para que facilmente fique acessível a toda a comunidade e para que possam contribuir para a evolução deste exemplo, a fim de mostrar o máximo de funcionalidades possível do Stellar mantendo sempre a maior simplicidade possível. A ideia é ser um exemplo introdutório para os recém-chegados à comunidade e ser uma outra forma de aprender e compreender o funcionamento do Stellar, sem ser através da documentação [59].

7.1.1 Processo de Criação da API

Neste subcapítulo são demostrados todos os passos tomados, desde a criação da API até à sua integração com a aplicação cliente. No texto abaixo é prossuposto que a aplicação cliente já se encontre desenvolvida, uma vez que o objetivo é mostrar na prática o uso do Stellar e não do desenvolvimento de aplicações através do Vue.

Criação da Estrutura API

Para a criação da estrutura da API foi usada a ferramenta de linha de comandos. Dentro de uma pasta vazia foi executado o comando `stellar init –name=example`. Este comando criou toda a estrutura necessária para se poder desenvolver a nova API. Tal como o seu nome indica, o argumento `name` é usado para definir o nome do projeto. Existe também um

argumento opcional, `version` que permite definir o numero da versão do projeto, mas neste caso não há interesse. Por defeito o numero de versão é definido como 1.0.0.

Criação de Modelos e Suas Ações

Nesta aplicação temos dois modelos, as perguntas (*questions*) e os comentários (*comments*). Mais uma vez, é usada a ferramenta de linha de comandos para gerar a estrutura necessária. Ao executar os comandos `stellar makeModel question -module=private -crud` e `stellar makeModel comment -module=private -crud` são automaticamente criados os ficheiros para os dois modelos e as ações correspondentes para a manipulação dos dois recursos na base de dados. O argumento *module* permite indicar o módulo onde os ficheiros devem ser gerados, o argumento *crud* informa o Stellar que deve gerar um ficheiro com um conjunto de ações (obter todos, obter por id, criar, editar e apagar) que permitam manipular o recurso na base de dados.

```
exports.default = (api, mongoose) => {
  // get Schema type
  let Schema = mongoose.Schema

  // return the schema
  return new Schema({
    _creator: { type: Schema.Types.ObjectId, ref: 'question' },
    content: String,
    user: Schema.Types.Mixed
  }, {
    timestamps: true
  })
}
```

Figura 16 - Declaração do Model para os Comentarios

O próximo passo define as propriedades dos dois modelos que pode ser visto na Figura 15 e Figura 16. Os modelos podem ser descritos de duas formas distintas, através de uma *hash,* que descreve apenas as propriedades do modelo, para modelos mais complexos pode ser usada uma função que recebe a instancia da API e do Mongoose. Neste caso foi feito uso das funções para se poder relacionar os dois modelos, necessitando estes do objeto **Schema** (do Mongoose) para fazer essa relação.

```
exports.default = (api, mongoose) => {
  // get Schema type
  let Schema = mongoose.Schema

  // return the model schema
  const questionS = new Schema({
    title: String,
    content: String,
    user: [Schema.Types.Mixed],
    comments: [{ type: Schema.Types.ObjectId, ref: 'comment' }]
  }, {
    timestamps: true,
    toJSON: { virtuals: true },
    toObject: { virtuals: true },
  })

  // add a dynamic fields
  questionS.virtual('excerpt').get(function () {
    return this.content.substring(0, 400) + '(...)'
  })

  // return the schema
  return questionS
}
```

Figura 17 - Declaração do Model para as Perguntas

Ligação com a Aplicação Cliente

Neste momento a API tem toda a lógica necessária para a manipulação de perguntas e de comentários. Na próxima etapa segue-se a inclusão da biblioteca cliente, esta é gerada a cada inicio da instância de execução. Esta biblioteca permite a simples interação entre a API e a aplicação cliente disponibilizando funções que permitem criar uma ligação persistente de forma automática, fazer chamadas a ações, seja por HTTP ou WebSocket ou mesmo enviar e responder a mensagens no sistema de *chat*. Depois de incluir a biblioteca é altura de criar uma nova instância da mesma:

```
// create a new client instance
self.stellar = new StellarClient({
  // server url
  url: 'http://0.0.0.0:8080/'
})
```

Figura 18 – Criação de uma Nova Instância do Cliente do Stellar

A instância anteriormente criada é partilhada por todos os componentes da aplicação *web* sendo assim possível obter os dados que estão armazenados na base de dados. A imagem apresentada de seguida (Figura 19) demostra uma chamada à API para obter a lista de perguntas.

```
self.$root.stellar.action('getQuestions', response => {
  self.questions = response.questions
})
```

Figura 19 – Chamada a uma Ação

Como se pode ver, usando a biblioteca gerada, é extremamente fácil ligar-se e fazer chamadas à API para obter ou manipular informação.

Comunicações em Tempo Real

Para demostrar o funcionamento do sistema que permite comunicação em tempo real (*chat*) quando novas perguntas são criadas por outros utilizadores e aparecem automaticamente para todos os clientes ligados.

Inicialmente é necessário adicionar uma linha (Figura 20) na ação que trata da criação das perguntas, isto para que a nova pergunta seja partilhada com todos os clientes ligados ao servidor nesse momento.

```
// spreed the new question by all WebSocket users (we don't need wait
api.chatRoom.broadcast({}, 'defaultRoom', { newQuestion: newModel })
```

Figura 20 – Envio de Mensagens em Broadcast na API

A imagem seguinte (Figura 21) mostra a subscrição das mensagens enviadas pelo servidor. Neste caso, apenas são tratadas as mensagens que indicam a criação e remoção de perguntas, essas mensagens são depois convertidas em eventos que depois são propagadas para os outros componentes de forma a atualizar os dados da *view*.

```
// handle the 'newQuestion' event
self.stellar.on('say', packet => {
  // get only the message property
  let message = packet.message

  // handle the different events
  if (message.newQuestion !== undefined) { // new questions
    self.$broadcast('new-question', message.newQuestion)
  } else if (message.delQuestion !== undefined) { // deleted question
    self.$broadcast('del-question', message.delQuestion)
  }
})
```

Figura 21 – Subscrição de Mensagens no Sistema de Chat

Como se pode confirmar, é extremamente simples criar uma API com todas as funcionalidades necessárias para inserir e manipular os dados armazenados, assim como consumir essa mesma API em sistemas já existentes. Apenas através da linha de comandos é possível criar uma API cem por cento funcional em apenas alguns minutos e sem que tenha que ser programada uma única linha de código.

8 Conclusões

O principal objetivo deste trabalho era conceber e implementar uma *framework* que fosse capaz de colmatar as principais falhas das que existem e são usadas atualmente. Esta solução tem o propósito de maximizar a reutilização do código desenvolvido em outros projetos. Procura também incentivar os engenheiros e programadores a modularizar as suas aplicações e a partilhar parte da lógica aplicacional com a comunidade de forma a que todos possam evoluir ajudando-se mutuamente. É promovida a simplicidade, a divisão das diferentes funcionalidades, em componentes pequenos e altamente testáveis, que facilitem a manutenção dos projetos, a curto ou a longo prazo não distinguindo grandes ou pequenos projetos.

Foram tomados em conta diferentes cenários de utilização da solução desenvolvida. O primeiro cenário de referência usado consiste no desenvolvimento de aplicações *web* de última geração, que se divide em duas partes, o componente responsável pelo processamento dos pedidos do cliente, tratamento e disponibilização de informação, outra parte é componente visual que reside no lado do cliente a SPA. Outro cenário de utilização é uma aplicação móvel que divide a mesma API com uma aplicação *web*, onde a sua manutenção é separada e há mecanismos que suportam esta manutenção. Por fim, um cenário onde existe a necessidade de alimentar outro serviço de *back-end* e onde comunicação em tempo real é uma necessidade.

8.1 *Revisão do Trabalho*

O Stellar pode representar um grande passo em frente para os programadores e organizações que desenvolvem APIs *web*. A solução apresentada oferece um conjunto de vantagens em relação a outras *frameworks, as quais foram* referidas no capítulo 2, possibilitando a utilização de apenas uma linguagem desde o servidor até ao *front-end*. Sendo o JavaScript uma linguagem já conhecida pelos programadores *web* e *designers* em geral, faz aumentar o número de pessoas que podem fazer uso do Stellar sem a necessidade de aprender uma nova linguagem. Sendo o Stellar uma *framework* construída por cima de uma poderosa plataforma, o Node.js, este oferece a performance e escalabilidade tais que cobrem as necessidades das grandes organizações.

O Stellar também introduz mecanismos que permitem uma migração incremental das funcionalidades, sem haver quebras de serviços nas aplicações já desenvolvidas. Ao possibilitar atribuir uma versão às ações desenvolvidas, é possível facilitar a manutenção de

projetos críticos que necessitem de uma migração incremental, dada as sua grande dimensão. Não existe necessidade de criar e configurar um novo servidor para manter o sistema antigo, tudo é tratado numa única instância de execução. Tudo o que é necessário é especificar uma versão na ação desenvolvida e qual a aplicação cliente, se estiver a consumir a API deve indicar a versão que pretende.

No que diz respeito ao suporte de diferentes cenários de utilização, pode-se dizer que o Stellar se adapta aos cenários mais simples, mas também aos mais complexos, onde existe a necessidade de suportar diferentes clientes que comunicam usando diferentes protocolos ou então que necessitam de comunicação em tempo real. O Stellar suporta, nativamente, até três protocolos em simultâneo, tudo numa única instância de execução. Isto dá-lhe uma flexibilidade e uma adaptabilidade imensa, tornando-o ideal para desenvolver uma "simples" API *web* ou então para ser o *core* de um sistema de última geração de IOT que necessite de comunicação intensa e em tempo real, sob o protocolo TCP.

Juntamente ao desenvolvimento da *framework* foi criado um conjunto de documentação em português[1] e em inglês[2] para explicar o funcionamento do Stellar aos programadores, através de exemplos práticos e através de uma linguagem simples e envolvente. O sucesso de uma ferramenta/tecnologia não depende apenas da técnica, mas também do suporte e da confiança transmitida a quem a irá utilizar. A documentação e os testes são uma parte extremamente importante, que mostram a maturidade e transmitem a segurança desejada na hora de adotar a solução. A documentação está disponível, na íntegra, no GitHub de forma a que a comunidade possa contribuir com correções e melhorias, de forma a tornar a documentação de maior utilidade.

8.2 *Trabalho Futuro*

Ao longo do desenvolvimento do Stellar foram identificadas algumas áreas onde podem ser feitas algumas melhorias e serviços adicionais que podem ser criados em torno deste trabalho. Nos pontos abaixo são apresentadas algumas ideias e melhorias que se pretende implementar no futuro, para dar continuidade ao trabalho aqui desenvolvido.

[1] https://pt.stellar-framework.com/
[2] https://stellar-framework.com/

Repositório de Módulos

Pretende-se desenvolver um repositório, onde a comunidade possa submeter os seus módulos e aplicações desenvolvidas. Este sub-projecto pode trazer múltiplas vantagens para a comunidade. Uma das vantagens é ser possível encontrar num único lugar todos os módulos da plataforma, outra é o facto dos criadores dos módulos poderem encontrar ajuda para amadurecer as suas criações ou outros programadores para contribuírem com novas ideias para os seus projetos. Por último pode-se encontrar os melhores programadores do Stellar, todos num único local. Assim, quando alguém necessitar de pessoas para trabalhar num projeto, facilmente se pode encontrar alguém com as *skills* adequadas.

Sistemas de Testes

Atualmente existe um sistema de testes que possibilita a sua criação e execução nos módulos, permitindo os programadores testarem as suas ações. Este sistema ainda tem espaço para evoluir, ao sistema atual, pode-se juntar um sistema de *coverage* para medir a cobertura dos testes desenvolvidos. Desta forma, será possível dar um *score* de confiabilidade a um determinado módulo que pode depois ser anexado à página dedicada ao módulo, no *website* do repositório.

Este sistema teria por propósito aumentar a confiabilidade de outros programadores nos módulos de terceiros, isto é muito importante porque deve haver total confiança nos módulos que são adicionados às nossas aplicações, pois estas podem não fazer exatamente aquilo que dizem fazer, levando a aplicação a ter um comportamento não esperado.

Comunidade

O trabalho apenas agora começou, uma vez que foi atingida uma versão estável, é necessário criar uma comunidade em torno do Stellar. Agora, começa todo um processo de promoção e divulgação do Stellar, é necessário mostrar as vantagens aos programadores *web* levando-os a experimentar, envolver-se e contribuir para que assim o projeto possa evoluir como um todo em direção a uma ferramenta mais poderosa e muito mais madura.

9 Referências

[1] S. C. Ho, “Electronic Commerce Research and Applications,” *Elsevier,* vol. 6, nº 3, pp. 237-259, 2007.

[2] L. Mendoza, “Coding Is Over,” 21 Junho 2016. [Online]. Available: https://medium.com/@loorinm/coding-is-over-6d653abe8da8#.b0xpzaj8i. [Acedido em 10 Outubro 2016].

[3] M. Elliston, “Evaluation of Web Frameworks,” 2006.

[4] B. Venners, “Orthogonality and the DRY Principle,” 10 Março 2003. [Online]. Available: http://www.artima.com/intv/dry.html. [Acedido em 25 Julho 2016].

[5] “Dont Repeat Yourself,” 11 Novembro 2014. [Online]. Available: http://c2.com/cgi/wiki?DontRepeatYourself. [Acedido em 25 Julho 2016].

[6] Microsoft, “Deployment Patterns,” [Online]. Available: https://msdn.microsoft.com/en-us/library/ms998478.aspx. [Acedido em 20 09 2016].

[7] Internet World Stats, “INTERNET GROWTH STATISTICS,” [Online]. Available: http://internetworldstats.com/emarketing.htm. [Acedido em 20 09 2016].

[8] T. Reenskaug e J. Coplien, “The DCI Architecture: A New Vision of Object-Oriented Programming,” 20 03 2009. [Online]. Available: http://www.artima.com/articles/dci_vision.html. [Acedido em 20 09 2016].

[9] Google, “Google Trends,” Google, [Online]. Available: https://trends.google.com/trends/. [Acedido em 22 03 2017].

[10] GitHub, “GitHub,” GitHub, [Online]. Available: https://github.com. [Acedido em 22 03 2017].

[11] "Why I won't recommend Meteor anymore," 2 06 2015. [Online]. Available: https://medium.com/@pierrebaz/why-i-won-t-recommend-meteor-anymore-fcec4e478d78#.nl2iyz8rp. [Acedido em 12 09 2016].

[12] M. Schoebel, "Meteor.js - The Perfect Match For Lean Startups," 17 01 2014. [Online]. Available: http://www.manuel-schoebel.com/blog/meteorjs-the-perfect-match-for-lean-startups. [Acedido em 12 09 2016].

[13] n. "Why experienced developers consider Laravel as a poorly designed framework?," 2015. [Online]. Available: https://www.reddit.com/r/PHP/comments/3bmclk/why_experienced_developers_consider_laravel_as_a/. [Acedido em 12 09 2016].

[14] "The PHP Framework For Web Artisans," [Online]. Available: https://laravel.com. [Acedido em 10 09 2011].

[15] B. Skvorc, "The Best PHP Framework for 2015: SitePoint Survey Result," 05 04 2015. [Online]. Available: https://www.sitepoint.com/best-php-framework-2015-sitepoint-survey-results/. [Acedido em 10 09 2016].

[16] "Architecture of Laravel Applications," [Online]. Available: http://laravelbook.com/laravel-architecture/. [Acedido em 25 Julho 2016].

[17] M. Model, "Model View Controller History," 26 12 2014. [Online]. Available: http://c2.com/cgi/wiki?ModelViewControllerHistory. [Acedido em 03 10 2016].

[18] . N. Adermann e J. Boggiano, "Composer," [Online]. Available: https://getcomposer.org/. [Acedido em 03 10 2016].

[19] "Package Development," [Online]. Available: https://laravel.com/docs/5.3/packages. [Acedido em 04 10 2016].

[20] "Eloquent: Getting Started," [Online]. Available: https://laravel.com/docs/5.3/eloquent. [Acedido em '4 10 2016].

[21] K. Marshall, C. Pytel e J. Yurek, "Introducing Active Record," em *Pro Active Record*, Apress, 2007, pp. 1-24.

[22] The PHP Group, “PHP Data Objects,” [Online]. Available: http://php.net/manual/en/intro.pdo.php. [Acedido em 04 10 2016].

[23] R. Fielding, “Principled design of the modern Web architecture,” *ACM Transactions on Internet Technology (TOIT),* vol. 2, nº 2, pp. 115-150, May 2002.

[24] G. Fekete, “Being a Full Stack Developer,” 22 09 2214. [Online]. Available: https://www.sitepoint.com/full-stack-developer/. [Acedido em 09 09 2016].

[25] “Lumen - PHP Micro-Framework By Laravel,” [Online]. Available: http://lumen.laravel.com. [Acedido em 25 Julho 2016].

[26] M. Fowler, “InversionOfControl,” 26 06 2005. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html. [Acedido em 21 07 2016].

[27] GitHub, “Electron,” [Online]. Available: http://electron.atom.io/. [Acedido em 09 09 2016].

[28] C. Quinn, “Meteor: bringing the awesome back to web dev,” 13 Agosto 2015. [Online]. Available: https://www.gravitywell.co.uk/latest/how-to/posts/meteor-bring-the-awesome-part-0/. [Acedido em 25 Julho 2016].

[29] Node.js Foundation, “Node.js,” [Online]. Available: https://nodejs.org/en/. [Acedido em 09 09 2016].

[30] MongoDB, Inc, “MongoDB,” [Online]. Available: https://www.mongodb.com. [Acedido em 09 09 2016].

[31] Handlebars, “Minimal Templating on Steroids,” [Online]. Available: http://handlebarsjs.com/. [Acedido em 09 09 2016].

[32] npm, Inc, “NPM,” [Online]. Available: http://npmjs.com/. [Acedido em 09 09 2016].

[33] Percolate Studio, “The trusted source for JavaScript packages, Meteor resources and tools,” [Online]. Available: https://atmospherejs.com/. [Acedido em 09 09 2016].

[34] Meteor Development Group Inc., “Blaze - Overview,” [Online]. Available: http://blazejs.org/. [Acedido em 04 10 2016].

[35] C. Froedge, “Why I’m not staking my future on MeteorJS,” 20 10 2015. [Online]. Available: https://medium.com/@calvinfroedge/why-i-m-not-staking-my-future-on-meteorjs-52e55fbf5332#.hjalxafhs. [Acedido em 12 09 2016].

[36] E. Tahler. [Online]. Available: http://www.actionherojs.com. [Acedido em 10 09 2016].

[37] redislabs, “Redis,” [Online]. Available: http://redis.io/. [Acedido em 04 10 2016].

[38] T. Texin, “Origin Of The Abbreviation I18n,” 2010. [Online]. Available: http://www.i18nguy.com/origini18n.html. [Acedido em 09 09 2016].

[39] W. Azeem, “MVC (Model View Controller),” Waqar, 16 11 2015. [Online]. Available: http://www.waqar.me/learn/mvc-model-view-controller/. [Acedido em 15 10 2016].

[40] E. Tahler, “ActionHero - Plugins,” [Online]. Available: http://www.actionherojs.com/docs/#plugins. [Acedido em 05 Outubro 2016].

[41] NGINX Inc, “Nginx,” [Online]. Available: https://www.nginx.com/. [Acedido em 05 Outubro 2016].

[42] D. Astels, Test Driven development: A Practical Guide, Prentice Hall Professional Technical Reference, 2003.

[43] E. Evans, “Domain Driven Design,” PLoP, 2002.

[44] A. Haigh, Object-Oriented Analysis and Design, McGraw-Hill Professional, 2001.

[45] The Rust Project Developers, “RUST,” The Rust Project Developers, [Online]. Available: https://www.rust-lang.org/en-US/. [Acedido em 08 Outubro 2016].

[46] Mozilla, “Promise,” Mozilla, 15 11 2016. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Acedido em 15 11 2016].

[47] M. Ogden, “Callback Hell,” [Online]. Available: http://callbackhell.com. [Acedido em 15 11 2016].

[48] “Mongoose,” [Online]. Available: http://mongoosejs.com/. [Acedido em 12 09 2016].

[49] E. Tahler, “node-resque,” [Online]. Available: https://github.com/taskrabbit/node-resque. [Acedido em 12 09 2016].

[50] “Semantic Versioning,” [Online]. Available: http://semver.org/. [Acedido em 12 09 2016].

[51] [Online]. Available: http://pm2.keymetrics.io/. [Acedido em 12 09 2016].

[52] “Coveralls,” [Online]. Available: https://coveralls.io/github/StellarFw/stellar. [Acedido em 13 09 2016].

[53] “Tavis-CI,” [Online]. Available: https://travis-ci.org/StellarFw/stellar. [Acedido em 13 09 2016].

[54] Auth0, “Auth0,” Auth0, [Online]. Available: https://auth0.com/. [Acedido em 15 10 2016].

[55] E. You, “Vue.js,” [Online]. Available: https://vuejs.org. [Acedido em 15 11 2016].

[56] Alibaba.com, “Alibaba,” Alibaba.com, [Online]. Available: https://www.alibaba.com/. [Acedido em 27 10 2016].

[57] Xiaomi, “Xiaomi,” Xiaomi, [Online]. Available: http://www.mi.com/en/. [Acedido em 27 10 2016].

[58] E. You, “Vue 2.0 is Here!,” 30 09 2016. [Online]. Available: https://medium.com/the-vue-point/vue-2-0-is-here-ef1f26acf4b8#.rsajstfqn. [Acedido em 27 10 2016].

[59] “Stellar Example,” [Online]. Available: http://github.com/stellarFw/example. [Acedido em 12 09 2916].

Anexos

Anexo I

A tabela seguinte contém a listagem de todos os "validadores" implementados pelo sistema, os seus parâmetros e uma pequena descrição do seu funcionamento.

Nome	Parâmetros	Descrição
`alpha`	Sem parâmetros	O valor a ser validado apenas pode conter letras.
`alpha_num`	Sem parâmetros	O valor a ser validado apenas pode conter letras e/ou números.
`alpha_dash`	Sem parâmetros	O valor a ser validado apenas pode conter letras, números e/ou hífens e *underscores*.
`array`	Sem parâmetros	O valor a ser validado tem que ser um `array`.
`before`	1. Limite superior da data	O valor a ser validado tem que ser uma data antes do dia especificado no primeiro parâmetro.
`between`	1. Limite inferior 2. Limite superior	O valor a ser validado deve ser um número entre os limites definidos no parâmetro.
`boolean`	Sem parâmetros	O valor a ser validado deve ser do tipo `boolean`

`confirmed`	Sem parâmetros	O valor sobre validação deve ser confirmado com outro campo. O campo de confirmação deve ter o nome do campo sobre validação com o sufixo '_confirmation'
`date`	Sem parâmetros	O campo sob validação deve ser uma data
`diferente`	1. Nome do campo a usar na comparação	O campo sob validação deve ser diferente do campo especificado como primeiro parâmetro
`email`	Sem parâmetros	O campo sob validação deve corresponder a um endereço de email valido
`filled`	Sem parâmetros	O campo sob validação deve-se encontrar preenchido
`in`	1. Um valor valido 2. Segundo valor valido 3. (...)	O campo sob validação deve estar presente num dos parâmetros. Este validador não tem limites de parâmetros.
`not_in`	1. primeiro valor invalido 2. segundo valor invalido 3. (...)	O valor do campo sob validação não pode estar presente nos parâmetros.
`integer`	Sem parâmetros	O campo sob validação deve ser um número inteiro

`ip`	Sem parâmetros	O campo sob validação deve ser um endereço IP válido
`json`	Sem parâmetros	O campo sob validação deve ser uma `string` com um JSON válido
`max`	1. limite superior	O campo sob validação deve ser um número ou `string`. No caso de ser um número, este não pode ultrapassar o limite superior definido no primeiro parâmetro. No caso de ser uma `string`, o comprimento da mesma não deve ultrapassar o limite superior definido no primeiro parâmetro.
`min`	1. limite inferior	O campo sob validação deve ser um número ou `string`. No caso de ser um número, este não pode ser menor ao limite inferior definido no primeiro parâmetro. No caso de ser uma `string`, o comprimento da mesma não deve ser menor do que o limite inferior definido no primeiro parâmetro.
`required`	Sem parâmetros	O campo a validar deve estar presente no conjunto de `inputs`
`numeric`	Sem parâmetros	O campo sob validação deve ser numérico

required_if	1. campo para usar na validação 2. valor 1 3. valor 2 4. (...)	O campo a validar fica a ser obrigatório se o campo especificado no primeiro parâmetro assumir um dos valores presentes nos restantes parâmetros.
required_unless	1. campo para usar na validação 2. valor 1 3. valor 2 4. (...)	O campo a validar é obrigatório a não ser que o campo especificado no primeiro parâmetro assuma um dos valores declarados nos restantes parâmetros
required_with	1. campo 1 2. campo 2 3. (...)	O campo a validar torna-se obrigatório se um dos campos declarados nos parâmetros estiver presente
required_with_all	1. campo 1 2. campo 2 3. (...)	O campo a validar torna-se obrigatório com a presença de todos os campos especificados nos parâmetros
require_without	1. campo 1 2. campo 2 3. (...)	O campo a validar torna-se obrigatório quando um dos campos especificado nos parâmetros não está presente no conjunto de *inputs*
required_without_all	1. campo 1 2. campo 2 3. (...)	O campo a validar torna-se obrigatório quando todos os campos declarados nos parâmetros não estão

		presentes no conjunto de *inputs*
`same`	1. valor	O valor do campo a validar deve ser o mesmo que o primeiro parâmetro
`size`	1. limite de tamanho	O valor a validar deve ser igual ao primeiro parâmetro, caso seja um número, ou então ao seu comprimento, no caso de uma `string` ou `array`
`url`	2. sem parâmetros	O campo a validar deve ser um URL válido

Anexo II

Documento de Testes

Este documento contém alguns dos testes realizados para validar as funcionalidades do Stellar. Foram realizados 170 testes no total, mas apenas alguns serão aqui apresentados, pois contribuiria para que este documento ficasse ainda mais extenso. Os testes foram divididos por grupos e são apresentados aqueles que foram considerados os testes de maior "importância" para cada um desses mesmos grupos.

Actions

Neste grupo é testado o comportamento do *ActionProcessor*. O teste apresentado abaixo valida as ações, se estas podem ser executadas internamente e se respeitam o padrão Promise [44].

```
describe('can execute internally', () => {

  it('without params', done => {
    api.actions.call('formattedSum').catch(_ => { done() })
  })

  it('reject works', done => {
    api.actions.call('formattedSum').should.be.rejected()

    done()
  })

  it('normally', done => {
    api.actions.call('formattedSum', { a: 3, b: 3 })
      .should.be.fulfilledWith({ formatted: '3 + 3 = 6' })
      .then(_ => { done() })
  })

})
```

API

Nos testes apresentados de seguida são verificados alguns comportamentos genéricos do **Engine** do Stellar. O primeiro verifica se é possível pedir uma versão especifica de uma ação, o segundo, testa os *validatores, se estes* são aplicados corretamente no momento de execução de uma ação.

```
it('can specify an apiVersion', function (done) {
  api.helpers.runAction('versionedAction', {apiVersion: 1}, function (response) {
    response.requesterInformation.receivedParams.apiVersion.should.equal(1)

    api.helpers.runAction('versionedAction', {apiVersion: 2}, function (response) {
      response.requesterInformation.receivedParams.apiVersion.should.equal(2)
      done()
    })
  })
})
```

```
it('will use validator if provided', function (done) {
  api.helpers.runAction('testAction', {requiredParam: true, fancyParam: 123}, response => {
    response.error.fancyParam.should.be.equal(`fancyParam should be 'test123'. so says test-server`)
    done()
  })
})
```

Cache

Neste grupo é testado o sistema de *cache*. Existem muitos testes de grande relevância, que poderiam ser apresentados, mas por serem bastante extensos fica apenas um exemplo. O primeiro teste verifica se o sistema de *cache* está bem implementado e se impede que o valor seja lido depois de expirado. O segundo verifica se é possível alterar uma chave protegida com um bloqueio, caso o utilizador seja o detentor do bloqueio.

```
it('cache.load with expired items should not return them', function (done) {
  api.cache.save('testKeyWait', 'test123', 10, (error, saveRes) => {
    saveRes.should.equal(true)

    setTimeout(() => {
      api.cache.load('testKeyWait', (error, loadRes) => {
        String(error).should.equal('Error: Object expired')
        should.equal(null, loadRes)
        done()
      })
    }, 20)
  })
})
```

```
  it('you can save an item if you do hold the lock', function (done) {
    api.cache.lock(key, null, (error, lock) => {
      lock.should.equal(true)

      api.cache.save(key, 'value', (error, success) => {
        success.should.equal(true)
        done()
      })
    })
  })
```

Error

Neste grupo é testada a correta serialização dos erros. O primeiro teste verifica se uma *string* é apresentada de forma correta, o segundo teste valida a serialização de um objeto.

```
it('returns string errors properly', function (done) {
  api.helpers.runAction('aNotExistingAction', {}, response => {
    response.error.should.equal('Error: unknown action or invalid apiVersion')
    done()
  })
})
```

```
it('returns Error object properly', function (done) {
  api.config.errors.unknownAction = () => new Error('error test')

  api.helpers.runAction('aNotExistingAction', {}, response => {
    response.error.should.be.equal('Error: error test')
    done()
  })
})
```

Events

Estes testes asseguram o correto funcionamento do sistema de eventos. Os dois apresentados verificam se o "disparo" dos eventos acontece de forma esperada e se os *listeners* são executados na ordem correta.

```
it('event.fire', done => {
  api.events.fire('example', { value: '' })
    .then(response => {
      response.value.should.be.equal('thisIsATest')
      done()
    })
})
```

```
it('listeners are executed in order', done => {
  api.events.listener('prog', (api, params, next) => {
    params.value += '1'
    next()
  }, 10)

  api.events.listener('prog', (api, params, next) => {
    params.value += '0'
    next()
  }, 5)

  api.events.fire('prog', { value: 'test' })
    .then(response => {
      response.value.should.be.equal('test01')
      done()
    })
})
```

Hash

Neste grupo é validado o sistema de *hashing*. O primeiro teste verifica se é possível gerar uma *hash* através da *salt* predefinida no sistema de configuração, o segundo verifica se é possível determinar se uma *hash* corresponde a um determinado "texto às claras" e se esta devolve resultados corretos.

```
it('hash data with predefined salt', done => {
  api.config.general.salt = SALT
  api.hash.hash(TEST_PASSWORD).then(result => {
    result.should.be.equal(TEST_PASSWORD_HASHED)
    done()
  })
})
```

```
it('compare plain data with hash in sync', done => {
  should(api.hash.compareSync(TEST_PASSWORD, TEST_PASSWORD_HASHED)).be.true()
  should(api.hash.compareSync('wrong_password', TEST_PASSWORD_HASHED)).be.false()
  done()
})
```

HTTP

Os testes que figuram abaixo valida o comportamento do *server* HTTP. O primeiro teste verifica se o *server* é capaz de responder aos pedidos do cliente de forma correta, o outro verifica se é possível chamar uma ação que chame outra ação privada.

```
it('server basic response should be JSON and have basic data', done => {
  request.get(`${url}/api/`, (error, response, body) => {
    body = JSON.parse(body)
    body.should.be.an.instanceOf(Object)
    body.requesterInformation.should.be.an.instanceOf(Object)
    done()
  })
})
```

```
it('can call actions who call private actions', done => {
  request(`${url}/api/formattedSum?a=3&b=4`, (error, response, body) => {
    body = JSON.parse(body)
    should.not.exist(body.error)
    body.formatted.should.equal('3 + 4 = 7')
    done()
  })
})
```

Middleware

Os testes deste grupo asseguram o correto funcionamento do sistema de *middleware*. O primeiro teste verifica se o *middleware* é capaz de bloquear uma conexão, o segundo verifica se o *middleware* pode alterar a conexão.

```
it('preProcessors can block actions', function (done) {
  api.actions.addMiddleware({
    name: 'test middleware',
    global: true,
    preProcessor: (data, next) => {
      next(new Error('BLOCKED'))
    }
  })

  api.helpers.runAction('randomNumber', response => {
    response.error.should.be.equal('Error: BLOCKED')
    should.not.exist(response.randomNumber)
    done()
  })
})
```

```
it('postProcessors can append the connection', function (done) {
  api.actions.addMiddleware({
    name: 'test middleware',
    global: true,
    postProcessor: (data, next) => {
      data.response._postProcessorNote = 'note'
      next()
    }
  })

  api.helpers.runAction('randomNumber', response => {
    response._postProcessorNote.should.be.equal('note')
    done()
  })
})
```

TCP

Os testes TCP verificam se o *server* TCP funciona de forma apropriada. O primeiro teste valida se é possível trocar mensagens realmente grandes, o segundo verifica se o bloqueio do numero de conexões em simultâneo funciona corretamente.

```
it('really long messages are OK', done => {
  // build a long message using v4 UUIDs
  let msg = {
    action: 'cacheTest',
    params: {
      key: uuid.v4(),
      value: uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4()
    }
  }

  makeSocketRequest(client1, JSON.stringify(msg), response => {
    response.cacheTestResults.loadResp.key.should.eql(`cache_test_${msg.params.key}`)
    response.cacheTestResults.loadResp.value.should.eql(msg.params.value)
    done()
  })
})
```

```
it('will error if received data length is bigger then maxDataLength', done => {
  // define a new data length value
  api.config.servers.tcp.maxDataLength = 64

  // build a long message using v4 UUIDs
  let msg = {
    action: 'cacheTest',
    params: {
      key: uuid.v4(),
      value: uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4() + uuid.v4()
    }
  }

  makeSocketRequest(client1, JSON.stringify(msg), response => {
    response.should.containEql({status: 'error', error: 'data length is too big (64 received/449 max)'})

    // return maxDataLength back to normal
    api.config.servers.tcp.maxDataLength = 0

    done()
  })
})
```

Validators

Estes testes validam o correto funcionamento dos *validators* existentes no sistema. Os testes abaixo verificam o funcionamento do *validator* max e required_with, respetivamente.

```
it('max', done => {
  (() => { simplesValidator('max', '') }).should.throw('Validation rule max requires at least 1 parameters.')
  should(simplesValidator('numeric|max:10', 9)).be.equal(true)
  should(simplesValidator('numeric|max:10', 10)).be.equal(true)
  should(simplesValidator('numeric|max:10', 11)).have.value('key', 'The key may not be greater than 10.')
  should(simplesValidator('max:3', 'as')).be.equal(true)
  should(simplesValidator('max:3', 'asd')).be.equal(true)
  should(simplesValidator('max:3', 'asdf')).have.value('key', 'The key may not be greater than 3 characters.')
  should(simplesValidator('array|max:3', [ 1, 2 ])).be.equal(true)
  should(simplesValidator('array|max:3', [ 1, 2, 3 ])).be.equal(true)
  should(simplesValidator('array|max:3', [ 1, 2, 3, 4 ])).have.value('key', 'The key may not have more than 3 items.')

  done()
})
```

```
it('required_with', done => {
  (() => {
    api.validator.validate({ key: '' }, { key: 'required_with' })
  }).should.throw('Validation rule required_with requires at least 1 parameters.')
  should(api.validator.validate({ key: '' }, { key: 'required_with:name,surname' })).be.equal(true)
  should(api.validator.validate({
    key: '',
    name: 'Alec'
  }, {
    key: 'required_with:name,surname'
  })).have.value('key', 'The key field is required when at least one of name, surname is present.')
  should(api.validator.validate({
    key: 'someValue',
    name: 'Alec'
  }, { key: 'required_with:name,surname' })).be.equal(true)

  done()
})
```

Websocket

Este grupo testa o correto funcionamento do *server* WebSocket. O primeiro teste verifica se os *interceptors* podem alterar a resposta do servidor, já o segundo verifica se é possível obter os detalhes de uma *room*.

```
it('can change the response', done => {
  client1.interceptors.push((params, next) => {
    next(response => {
      response.additionalField = 'awesomeCall'
    })
  })

  client1.action('formattedSum', { a: 3, b: 4 })
    .then(response => {
      response.additionalField.should.be.equal('awesomeCall')
    })
    .then(_ => { done() })
})
```

```
it('can change rooms and get room details', done => {
  client1.roomAdd('otherRoom')
    .then((res) => client1.detailsView())
    .then(response => {
      should.not.exist(response.error)
      response.data.rooms[ 0 ].should.equal('defaultRoom')
      response.data.rooms[ 1 ].should.equal('otherRoom')

      return client1.roomView('otherRoom')
    })
    .then(response => {
      response.data.membersCount.should.equal(1)
      done()
    })
})
```