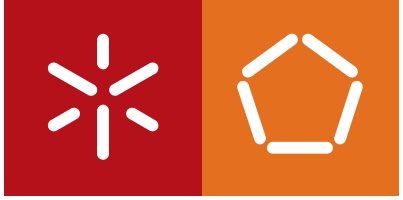
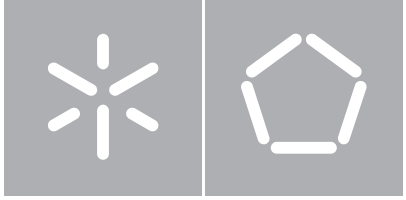


Universidade do Minho
Escola de Engenharia

André da Silva Nogueira

Profiling de aplicações Web : Estudo comparativo entre aplicações Java Web e aplicações RoR

Outubro de 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André da Silva Nogueira

Profiling de aplicações Web : Estudo comparativo entre aplicações Java Web e aplicações RoR

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor F. Mário Martins

Outubro de 2014

Agradecimentos

Gostaria de expressar os meus agradecimentos a todas as pessoas, que de forma directa ou indirecta, contribuíram para a realização e conclusão desta dissertação. No entanto, gostaria de realçar e deixar um agradecimento especial às seguintes pessoas ou entidades.

Ao meu orientador Professor F. Mário Martins, pela orientação, compreensão e disponibilidade despendida durante a realização desta dissertação. Agradeço, também, pela pertinentes questões colocadas, sugestões e ajuda fornecida durante o desenvolvimento do trabalho que possibilitaram a conclusão do mesmo.

Ao Engenheiro António Aragão do Departamento de Informática, pela disponibilização de duas máquinas virtuais que foram importantes para a criação das aplicações web e a realização dos testes necessários para a conclusão do trabalho.

Aos meus pais, Maria Cândida Silva e Emílio Nogueira pela oportunidade que me proporcionam, pela confiança depositada em mim durante a licenciatura e o mestrado, pelo incentivo e compreensão que sempre me deram.

Ao Daniel Silva e à Joana Sousa pelas brincadeiras, pela amizade, pela paciência e ajuda, pelo apoio e compreensão que sempre tiveram comigo.

Por fim, também gostaria de deixar os meus agradecimentos a todos os meus colegas e amigos que me acompanharam durante o percurso académico, em especial ao José Pinheiro, Paulo Maia e Diogo Barbosa.

Abstract

Profiling of Web applications is a recent concern of Software Engineering that already act profiling programs. Profiling of Web applications presents itself as much more complex because a correct Web application must meet the multi-tier model MVC, apart from the necessary persistence implementation in databases.

Therefore, the analysis of behavior and performance must be also separable by layers, by the implemented interactions between them and the implementation of persistence. Any poor implementation of a layer will cause inefficiency, and any bad interactions between layers may cause even greater inefficiency.

Nowadays, several tools exist for profiling Web applications, and there is no systematic study on how to effectively measure each layer and each execution context. This work aims to synthetize a crucial set of facets (characteristics or properties) covered by the various profiling tools, to catalog types of objective measurement and/or layer, specific evaluation grids.

As a case study and experience in the application of the synthesized grids, multifaceted performance tests will be conducted using two Web applications with different characteristics, both developed in Java Web and RoR. The results of the study are completely dependent on the results of the performed tests.

Resumo

O *profiling* de aplicações Web é uma preocupação da Engenharia de Software mais recente que o já conhecido *profiling* de programas, e que se apresenta como muito mais complexa dado que, necessariamente, uma correta aplicação Web deve satisfazer o modelo multicamada MVC, para além de garantidamente possuir uma implementação de persistência em bases de dados.

Assim, a análise de comportamento e performance é também separável por camadas, pelas interações implementadas entre elas e pela implementação de persistência. Qualquer má implementação de uma camada pode provocar ineficiência, e qualquer má interação entre camadas pode provocar ainda maior ineficiência.

Existindo atualmente diversas ferramentas de *profiling* de aplicações Web, e não havendo um estudo sistemático sobre o que efetivamente medem relativamente a que camada e em que contexto de execução, pretende-se neste trabalho, após uma análise destas ferramentas, sintetizar um conjunto crucial de facetas (características ou propriedades) abrangidas pelas várias ferramentas de *profiling*, realizar a sua catalogação por tipos de objectivo de medida e/ou camada, visando construir grelhas específicas de avaliação.

Como caso de estudo e experiência de aplicação das grelhas sintetizadas, serão realizados testes de performance multi-faceta a duas aplicações Web, com características diferentes, mas ambas desenvolvidas em Java Web e em RoR. Os resultados do estudo são completamente dependentes dos resultados dos testes realizados.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Acrónimos

CoC *Convention over Configuration*

EJB *Enterprise Java Beans*

GC *Garbage Collector*

IDE *Integrated Development Environment*

Java EE *Java Enterprise Edition*

JDBC *Java Database Connectivity*

JPA *Java Persistence API*

JSP *JavaServer Pages*

JTA *Java Transaction API*

JVM *Java Virtual Machine*

MVC *Model-View-Controller*

ORM *Object-Relational Mapping*

REST *Representational State Transfer*

RoR *Ruby on Rails*

RVM *Ruby Version Manager*

Capítulo 1

Introdução

O aumento exponencial das designadas aplicações para a Web (*web-based applications*), aplicações em geral multicamada e inerentemente complexas, e a necessidade de, tal como com qualquer outra aplicação, monitorizar o seu funcionamento para o compreender e, assim, optimizar a sua performance, conduziu ao aparecimento de uma subárea de *software profiling* designada por *web application profiling*.

Fazer *profiling* de aplicações Web é uma tarefa bastante mais complexa dado que, sendo estas aplicações multicamada, as razões para o seu eventual mau ou pobre desempenho podem ser encontradas no mau funcionamento, instalação ou configuração dos componentes usados em cada uma das várias camadas, ou, pior ainda, por estrangulamentos resultantes da má implementação das comunicações entre camadas.

Tal grau de complexidade de análise parece aumentar, ironicamente, quando o desenvolvimento das aplicações Web é feita seguindo princípios fundamentais em Engenharia de Software, tal como o princípio da separação de camadas que conduziu ao modelo MVC, actualmente não só suportado mas quase obrigatório na maioria dos mais conhecidos *frameworks* para desenvolvimento de aplicações Web, sejam as plataformas .Net, Java ou mesmo Ruby on Rails (RoR).

O número de *Web profilers* para cada uma das plataformas anteriormente indicadas é muito grande, desde os *open-source* até aos comerciais, desde os que vêm incluídos nas próprias plataformas (cf. *JVisualVM* para Java, *ANTS Profiler* para *.NET* ou *RubyProf* para *RoR*) bem como muitos outros entretanto desenvolvidos, alguns deles associados mesmo a *frameworks* de desenvolvimento como, por exemplo, *Spring Insight* para *Spring-MVC*, *dotTrace* para *VisualStudio .NET*, etc.

Pretendendo-se realizar um estudo comparativo multifacetado de questões associadas à performance final de implementação em aplicações *Web* desenvolvidas usando tecnologias Java Web e usando tecnologia RoR, torna-se imperioso que, numa primeira fase, se pesquisem e classifiquem todas as facetas que são cobertas pelos diversos *web profilers* existentes, muitos deles sendo vocacionados para a análise de performance de uma ou mais camadas específicas, procurando sintetizar numa grelha de facetas, etiquetadas por camada MVC, por parâmetros de servidor ou por parâmetros relacionados com a persistência, ou outros.

Esta grelha, estruturada e tipada, servirá de base, enquanto conjunto quase universal de facetas de comparação, para a criação de subconjuntos específicos que serão usados nas comparações finais de desempenho de duas aplicações *Web* desenvolvidas nas plataformas Java *Web* e RoR.

Sabe-se, à partida, que tais comparações não serão completamente independentes do contexto. Por exemplo, uma aplicação Java *Web* desenvolvida para *TomCat* ou para *GlassFish* (servidores pré-instalados em *NetBeans 7.4*, que até inclui um profiler), deverá ter performances diferentes por diferentes razões.

Assim, e em resumo, mais do que a própria comparação em si, que se apresenta apenas como desafio para todo o estudo, pretende-se encontrar as facetas de comparação que podem constituir uma grelha comparativa adequada em função do contexto de implementação e da perspectiva de comparação. Finalmente, todo o estudo é confinado a arquitecturas de implementação cliente-servidor-base de dados não distribuídas.

1.1 Objectivos

O principal objectivo deste trabalho é a comparação e o estudo de dois frameworks de desenvolvimento web através da técnica de *profiling*. Para tal comparação poder ser feita, é necessário antes de mais estudar ambas as linguagens de programação, conhecer os *profilers* existentes e recolher o que cada um deles tem para oferecer em termos de facetas/características.

Após a recolha das diversas propriedades analisáveis por cada ferramenta de profiling, será preciso organizá-las pelas camadas da arquitectura multi-camada. De seguida, é importante termos uma aplicação com alguma complexidade desenvolvida nos dois frameworks, efectuar a análise a cada uma das aplicações criadas e recolher todos os valores associados às facetas escolhidas.

Por fim e já com todos os dados recolhidos e tratados, analisar esses mesmos dados e efectuar uma análise acerca do desempenho de cada uma das plataformas. Durante essa análise, os dados recolhidos serão divididos nas diversas camadas e comparados. Finalmente, após a análise individual de cada camada, e se tal for possível, concluir qual dos dois *frameworks* de estudo possui um melhor desempenho global.

1.2 Estrutura do Documento

No capítulo 1, é apresentada a contextualização e motivação que levou ao desenvolvimento deste trabalho e os objectivos a atingir no fim do estudo comparativo de aplicações em Java Web e *Ruby on Rails* (RoR).

No capítulo 2 é dada uma visão sobre o que é o *profiling* e em que vertentes da Engenharia de Software se pode aplicar. De seguida, são expostas as categorias a serem tratadas ao longo deste trabalho, bem como uma descrição de cada uma delas. Por fim, enumera-se uma lista de ferramentas de *profiling* que encaixam nas categorias já expostas, independentemente de possuírem uma licença comercial ou de serem de livre uso (*open-source*).

No capítulo 3, é exibida uma visão geral sobre as plataformas que serão consideradas durante o estudo

comparativo. Acerca de cada uma, apresenta-se a linguagem que sustenta a plataforma em questão e algumas das características que as tornam únicas. Depois, é exposta a plataforma em si e explicado a forma como cada uma implementa o modelo *Model-View-Controller* (MVC).

No capítulo 4, é apresentada a cartografia das facetas de *profiling* que as ferramentas já estudadas anteriormente permitem averiguar acerca de uma aplicação nas plataformas de desenvolvimento *web*. Esta cartografia inicial não tem em consideração a designação utilizada por cada ferramenta nem a refinação usada pela faceta, ou seja, se a faceta analisa alguma característica global ou se permite saber algo específico. Depois, a partir da cartografia apresentada, é efectuada uma selecção daquelas que se encaixam no foco principal deste estudo e que também podem ser consideradas importantes de conhecer numa aplicação *web*. Após a selecção, descreve-se em que consistem as facetas, bem como as unidades métricas que elas usam.

No capítulo 5, é descrito o conceito e os requisitos que se encontram por detrás da aplicação *web* desenvolvida. Também são referidos alguns dos objectivos que se pretende atingir com a aplicação no âmbito de *Web Profiling*. Após essa descrição conceptual, são expostas as tecnologias utilizadas para implementar a aplicação *web* na plataforma Java e na plataforma *Ruby on Rails* (RoR), bem como o software necessário para colocar uma aplicação *web* disponível para qualquer utilizador.

No capítulo 6, apresentam-se os passos realizados para efectuar *profiling* sobre as aplicações *web* desenvolvidas e exibidos os resultados obtidos durante as sessões de *profiling*. Também se descreve a forma como foi desenhado o teste de desempenho a executar sobre as aplicações *web*, como forma de gerar utilização simultânea e aleatória. Após isto, é realizada uma análise dos resultados com vista a verificar qual das duas plataformas possui melhor desempenho nas camadas da arquitectura MVC. Por último, é feita uma análise de escalabilidade das duas plataformas usando para tal uma ferramenta que consegue averiguar essa informação através da contínua utilização de uma aplicação *web*.

No capítulo 7, são expostas as conclusões retiradas durante a realização do estudo das plataformas *web* e referida a importância de associar as facetas que actualmente as ferramentas de *profiling* oferecem às camadas da arquitectura MVC. Além disso, também serão efectuadas algumas sugestões para trabalhos futuros usando como base a categorização apresentada.

Capítulo 2

Estado da Arte

2.1 Profiling

As aplicações *web* são cada vez mais predominantes no mercado e com tal crescimento surgiu a necessidade de optimização e eficácia nas respostas apresentadas aos utilizadores para garantir uma boa experiência de utilização. Para tal poder ser alcançado, é necessário descobrir em que camada da aplicação *web* está o problema, isto é, saber qual a parte da arquitectura multicamada que está a causar o problema na eficaz apresentação da resposta ao utilizador.

Para atingirmos o objectivo anterior, é importante termos propriedades/características que possam ser medidas. Uma boa afirmação acerca da importância da medição dessas propriedades é feita por *Lord Kelvin's*[1] [1] :

> *"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science."*[2]

Em Engenharia de Software, *profiling* de programas ou simplesmente *profiling* é a investigação do comportamento de um programa através da recolha de dados durante a sua execução [3]. As ferramentas que utilizam esta técnica possibilitam a análise da execução de uma aplicação através da instrumentação do código fonte ou do código binário da aplicação ou através de *hooks* fornecidos pela linguagem de programação para os *profilers* se conectarem à execução do programa desejado. Com esta técnica, é possível averiguar entre outras coisas, o tempo de execução de um método ou o número de vezes que esse método foi invocado ou a memória usada por ele e todos os seus submétodos caso eles existam. Para além disso, também é possível descobrir a memória alocada para cada objecto durante a execução, o número de

[1] William Thomson (ou Lord Kelvin) foi um físico matemático e engenheiro britânico

classes carregadas e descarregadas da memória do computador e no caso de linguagens orientadas pelos objectos, descobrir a eficácia e o número de vezes que é executado o *Garbage Collector* (GC) embutido na linguagem em questão.

Em termos de aplicações *web* surgiu também a necessidade de descobrir quais os pontos mais críticos de forma a que se pudesse assim melhorar a performance geral e também garantir a satisfação de resposta da aplicação ao utilizador, que é conhecido na indústria como *Application Performance Index* (*Apdex*) [4]. Numa aplicação orientada para a Internet, é importante termos em atenção aspectos que numa aplicação mais direccionada para computadores pessoais seriam inexistentes ou irrelevantes. Devido à natureza da sua implementação, os usuais *frameworks* de desenvolvimento *web* sugerem ou forçam, através dos seus criadores, o uso do padrão de software multi-camada conhecido como MVC[5]. Uma breve explicação de como as plataformas *Java Enterprise Edition* (Java EE) e RoR implementam este padrão será apresentada nas secções 3.1.2 e 3.2.2 respectivamente.

O uso deste padrão de software traz muitos benefícios ao desenvolvimento de aplicações *web* mas em contrapartida traz desvantagens quando se trata da investigação do funcionamento de uma aplicação se usarmos as convencionais ferramentas de *profiling*. Isto acontece pois é preciso termos acesso a informação mais detalhada do que aquela disponibilizada pelos *profilers* usuais, como, por exemplo, o tempo de resposta de um pedido efectuado pelo utilizador e dentro deste pedido saber quais os métodos executados e a que camadas da arquitectura MVC pertencem, para assim chegarmos a uma conclusão de que camada poderá afectar o tempo de resposta.

2.2 Categorias de Profiling

Antes de analisarmos em profundidade quais são as facetas ou características recolhidas pelas ferramentas de *profiling* que serão expostas no capítulo 4, é preciso categorizá-las de forma a sabermos onde cada uma das ferramentas a estudar se encaixa no mundo da análise de software e aplicações *web*, e os objectivos que pretendem atingir durante a investigação da execução de um programa.

Como se pode observar na figura 2.1, a área de *profiling* pode ser dividida em três principais categorias que muitas vezes se complementam. Essas categorias são o *CPU*, *Memory* e *Web Profiling*. Normalmente, as duas primeiras categorias, *CPU* e *Memory*, são encontradas em aplicações a serem usadas em computadores pessoais, pois é necessário ter a preocupação acerca da memória usada pela aplicação e também pelo tempo de execução necessário no processador para o programa funcionar. Já em aplicações *web*, essas duas categorias encontram-se, usualmente, nas subcategorias do *Web Profiling*, pois no fim de contas uma aplicação *web* não passa de uma programa, com um propósito e arquitectura diferente, mas que também consome memória e precisa de tempo de execução no CPU.

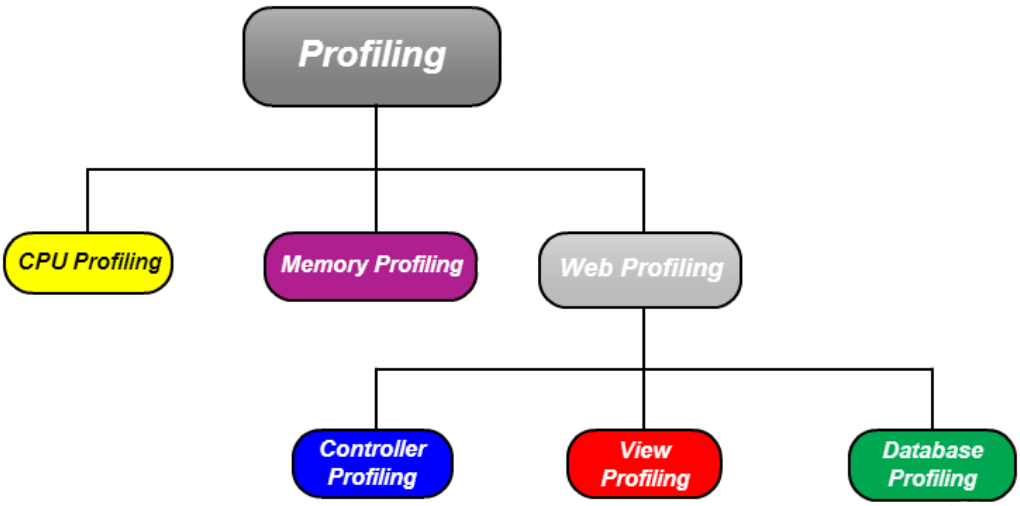


Figura 2.1: Categorias de Profiling

Antes de ser apresentada uma explicação para cada uma das categorias presentes na figura, há que ressaltar que a categoria de *Web Profiling* é muito recente e que a documentação acerca dela é escassa, bem como as ferramentas que permitem este tipo de análise e que, maioritariamente, são pagas. Também será importante referir que as cores presentes na figura 2.1 associadas às categorias serão usadas no capítulo 4, que vai incidir sobre as propriedades analisáveis por cada ferramenta de *profiling* referidas ao longo deste capítulo.

Análise de CPU (CPU Profiling)

A análise de *CPU* consiste principalmente em identificar os métodos com maior latência de uma aplicação, isto é, as funções que ocupam mais tempo de processamento no *CPU*. O cálculo da latência de um método pode ser feito de duas formas, tempo decorrido (*Elapsed Time*) e tempo de *CPU*[6].

O tempo decorrido é, como o próprio nome indica, o tempo gasto pela execução de um método. Neste tempo gasto pelo método, está incluído o tempo gasto dentro do mesmo, de todos os seus sub-métodos e qualquer tempo gasto em acções relacionadas com a rede (*Network I/O*) ou a escrita/leitura do disco (*Disk I/O*). Idealmente, o tempo decorrido seria apenas a duração entre a entrada e a saída do método como se tratasse de uma corrida cronometrada mas o tempo decorrido é calculado através da seguinte fórmula: $ElapsedTime = CPUTime + NetworkI/O + DiskI/O$.

O tempo de *CPU*, por outro lado, refere-se ao tempo gasto exclusivamente no *CPU* por um método ao executar a lógica contida nele. Neste tempo não está incluído nenhum tempo despendido nas execuções de pedidos à rede nem ao disco.

Análise de Memória (Memory Profiling)

O *profiling* de memória consiste principalmente na análise de como os objectos são criados na memória do computador onde está a ser executada a aplicação. Os dados acerca da criação e alocação de memória de cada objecto são obtidos durante a execução através de ferramentas de *profiling* (a discutir na secção 2.3).

Através destas ferramentas é possível encontrar falhas de memória ou objectos que estejam a ocupar demasiada memória ou em certos casos saber se o *Garbage Collector* está ou não a ser invocado demasiadas vezes, causando assim uma quebra no desempenho da aplicação.

Web Profiling

O *Web Profiling* pode ser considerado como sendo um categoria de *Profiling*, pois surgiu a necessidade de optimizar uma aplicação *web* e as técnicas já descritas de *profiling* não eram suficientes para descobrir os problemas inerentes a uma arquitectura multicamada.

Através de uma análise a uma qualquer aplicação desenvolvida usando a arquitectura MVC, é possível extrair três grandes áreas a avaliar e que podem levar uma aplicação a ficar pesada. Estas áreas podem provocar problemas, pois cada *framework* de desenvolvimento oferece uma abstracção sobre cada uma delas para facilitar o desenvolvimento, mas escondendo alguma da complexidade inerente e nessa mesma complexidade poderá estar o problema.

Se não for da abstracção criada pelo que é oferecido, então o problema pode derivar do mau uso ou conhecimento por parte do programador das funcionalidades oferecidas. Então, as três áreas do *Web Profiling* são as seguintes :

- **Controller** : Pretende avaliar todo o fluxo gerado nos pedidos do utilizador e averiguar qual das acções internas poderá causar o problema. Entre as acções internas inclui todos os métodos que no fim da sua execução fazem um pedido para apresentar uma página *Web*.
- **View** : Pretende avaliar o tempo que demora a carregar o conteúdo de uma página *HTML* que foi previamente escrita num *template* com *tags* específicas de cada *framework*.
- **Database** : Pretende avaliar a forma como é feita uma ligação à base de dados e de como trata cada *query* feita no modelo correspondente, pois existe uma grande abstracção no que toca às ferramentas que fazem a transformação das tabelas de uma base de dados em classes da linguagem.

2.3 Ferramentas de Profiling

Como já referido anteriormente, existe uma variedade de ferramentas que facilitam a análise de software em tempo de execução e com a informação fornecida, através de gráficos ou apenas números numa tabela, consegue-se descobrir onde se encontra o problema de performance com o programa.

Ao longo desta secção, será apresentada uma lista de ferramentas de *profiling* para cada um dos *fra-*

meworks em estudo com uma breve descrição. Será considerado qualquer tipo de *profiler*, quer *open-source* quer comercial, que permita conhecer algum tipo de informação acerca da execução de um programa e do seu funcionamento interno.

No capítulo 4, serão apresentadas, em tabelas organizadas, as facetas que cada uma das ferramentas apresentadas nesta secção analisam, para que seja feita uma comparação entre elas e posteriormente escolher quais as melhores ferramentas a usar para avaliar ambas as plataformas seleccionadas e efectuar a sua comparação.

Plataforma Ruby on Rails

Ruby-Prof/Rubinius/JRuby

Na linguagem de programação *Ruby*, existe a possibilidade de escolher um diferente interpretador para além daquele que se encontra pré-definido. Cada um destes interpretadores tem embutido um *profiler*. São capazes de analisar praticamente as mesmas facetas apenas com algumas diferenças.

Então, para o interpretador *Matz's Ruby Interpreter* (*MRI*) e *Ruby Enterprise Edition* (*REE*) é recomendado usar o *Ruby-Prof*, que é uma *gem*[2] desenvolvida pela comunidade de *Ruby*. Já nos restantes interpretadores, *Rubinius* e *JRuby*, ambos trazem consigo a sua própria ferramenta de *profiling*.

%self	cumulative	total	self	children	calls	self/call	total/call	name
46.34	4.06	8.72	4.06	4.66	501	0.01	0.02	Integer#upto
23.89	6.16	2.09	2.09	0.00	61	0.03	0.03	Kernel.sleep
15.12	7.48	1.33	1.33	0.00	250862	0.00	0.00	Fixnum#%
14.13	8.72	1.24	1.24	0.00	250862	0.00	0.00	Fixnum#==
0.18	8.74	0.02	0.02	0.00	1	0.02	0.02	Array#each_index
0.17	8.75	6.64	0.01	6.63	500	0.00	0.01	Object#is_prime
0.17	8.77	6.66	0.01	6.64	1	0.01	6.66	Array#select
0.00	8.77	0.00	0.00	0.00	501	0.00	0.00	Fixnum#-
0.00	8.77	0.00	0.00	0.00	1	0.00	0.00	Array#first
0.00	8.77	0.00	0.00	0.00	1	0.00	0.00	Array#length
0.00	8.77	0.00	0.00	0.00	1	0.00	0.00	Array#initialize
0.00	8.77	8.77	0.00	8.77	1	0.00	8.77	Object#run_primes
0.00	8.77	0.00	0.00	0.00	1	0.00	0.00	Integer#to_int
0.00	8.77	6.66	0.00	6.66	1	0.00	6.66	Object#find_primes
0.00	8.77	2.09	0.00	2.09	1	0.00	2.09	Object#find_largest
0.00	8.77	0.02	0.00	0.02	1	0.00	0.02	Object#make_random_array
0.00	8.77	0.00	0.00	0.00	1	0.00	0.00	Class#new
0.00	8.77	0.00	0.00	0.00	500	0.00	0.00	Array#[]=
0.00	8.77	0.00	0.00	0.00	61	0.00	0.00	Fixnum#>
0.00	8.77	0.00	0.00	0.00	61	0.00	0.00	Array#[]
0.00	8.77	8.77	0.00	8.77	1	0.00	8.77	#toplevel
0.00	8.77	0.00	0.00	0.00	500	0.00	0.00	Kernel.rand

Figura 2.2: Relatório *Flat* do *Ruby-Prof*

Nos *profilers* referidos, os relatórios com os dados da observação da execução da aplicação podem ser vistos em três formatos básicos, estando na figura 2.2 um exemplo deles. Estes formatos de relatórios são:

[2]Na forma mais básica, uma *Ruby gem* é um *package*, ou seja, pode-se comparar com os *packages* existentes na linguagem Java.

Flat, *Graph* e *Tree*.

Em relação ao que permitem avaliar, estes *profilers* encaixam-se nos usuais modos de *profiling* para aplicações já referidos na secção anterior, sendo eles *CPU* e Memória. É importante referir que são bastantes extensíveis e que existem *gems* no repositório que permitem obter ou organizar melhor a informação gerada por cada *profiler*.

Mini-Profiler

Mini-Profiler é uma ferramenta *open-source* de *profiling* simples desenvolvida pela equipa do *StackOverflow* orientada originalmente para aplicações *web* em *.NET* mas que já foi implementada para aplicações em RoR, **Go** e *Node.js*. Esta ferramenta oferece a capacidade de em ambiente de desenvolvimento descobrir os problemas de desempenho na aplicações através da inclusão de um botão num dos cantos superiores da página *web* carregada, como se pode observar na figura 2.3. Ao carregar nesse botão, é aberto um painel onde é possível saber o fluxo de execução que a página precisou para ser criada, isto é, os pedidos ao servidor necessários para que a página seja carregada correctamente, bem como a duração de cada um desses pedidos e das transacções feitas na base de dados.

Figura 2.3: *Mini-Profiler*

Plataforma Java EE

VisualVM

VisualVM é uma ferramenta visual de *profiling open-source* que vem incluída no *JDK* desde a versão 6.7. Esta ferramenta possui a capacidade de monitorizar o *CPU* usado e a memória usada por uma aplicação, quer seja *stand-alone* quer seja baseada em Internet. Consegue apresentar a informação referente a classes e métodos usados mas em contrapartida não é capaz de exibir o fluxo de uso de uma aplicação.

NetBeans Profiler

NetBeans Profiler é uma extensão ao *Integrated Development Environment* (IDE) *NetBeans* 7.4 que fornece as funcionalidades inerentes a uma ferramenta de *profiling*, isto é, capacidade para monitorizar o uso da *CPU* e da memória em tempo de execução. Também tem a capacidade de apresentar gráficos do uso da memória ao longo da aplicação, bem como mostrar em detalhe a informação acerca de um método ou o fluxo que possui internamente. Este *profiler* é, em aparência, praticamente idêntico ao *VisualVM* e também como este último é *open-source*.

YourKit

YourKit Profiler é mais uma ferramenta de *profiling* como as anteriores mas com um senão, ela apenas pode ser usada gratuitamente durante 15 dias e após esse período de avaliação é preciso possuir uma licença comercial paga para continuar a usar a ferramenta.

Esta ferramenta de *profiling* consegue analisar características relacionadas com o CPU e com a Memória gasta durante a execução das aplicações para além de possuir um sistema de detecção automática de falhas, uma sobrecarga (*overhead*) muito baixa na aplicação e a capacidade de se integrar com os seguintes IDE's : *Eclipse*, *JBuilder*, *IntelliJ IDEA*, *NetBeans* e *JDeveloper*. Também tem a capacidade de analisar aplicações *J2EE* através da ligação aos servidores de aplicação como *Geronimo*, *JBoss*, *Jetty*, *JRun*, *Orion*, *Resin*, *GlassFish*, *Tomcat*, *WebLogic* e *WebSphere*.

JProfiler

Como o *YourKit*, o JProfiler é uma ferramenta de *profiling* com licença comercial e com um período de avaliação de 15 dias. Tem praticamente as mesmas características que o *YourKit* e entre elas encontram-se a capacidade de analisar o *CPU* gasto e a Memória de uma aplicação, análise de vários servidores de aplicações para aplicações Java EE e de integração com IDE's.

O que distingue esta ferramenta da ferramenta *YourKit*, é a possibilidade de efectuar *profiling* sobre uma base de dados, onde se pode incluir *JDBC*, *JPA/Hibernate*, *MongoDB*, *Cassandra* e *HBase*, e o número de métricas que consegue analisar sobre uma aplicação *web* a funcionar num dos servidores. Para além do que já foi referido acerca da ferramenta, é imperativo evidenciar que o *JProfiler* é uma das ferramentas mais

usadas e reconhecidas na comunidade de Java, devido à sua qualidade de análise de uma aplicação e da sua interface gráfica simples e intuitiva que possibilita uma curva de aprendizagem rápida.

New Relic

Por último, uma ferramenta que oferece a possibilidade de analisar o fluxo de uma aplicação *web* e que se encaixa na categoria de *Web Profiling*. New Relic é uma aplicação *web* que oferece a capacidade de analisar e agregar a informação relativa a um *website* ou uma aplicação móvel. Inicialmente, estava disponível apenas para aplicações *Ruby on Rails* mas actualmente é capaz de analisar qualquer aplicação *web* escrita nas seguintes linguagens : *PHP*, *Java*, *Microsoft .NET*, *Python* e *Node.js*.

Na figura 2.4, é possível verificar uma das muitas funcionalidades que oferece. Na imagem em questão, tem-se acesso à performance no navegador de Internet, isto é, o tempo que demora a carregar essa página bem como todos os seus componentes. E caso o problema não se encontre na renderização da página em si mas no *back-end* da aplicação, também é possível ver essa informação ao carregar na aba *App performance*.

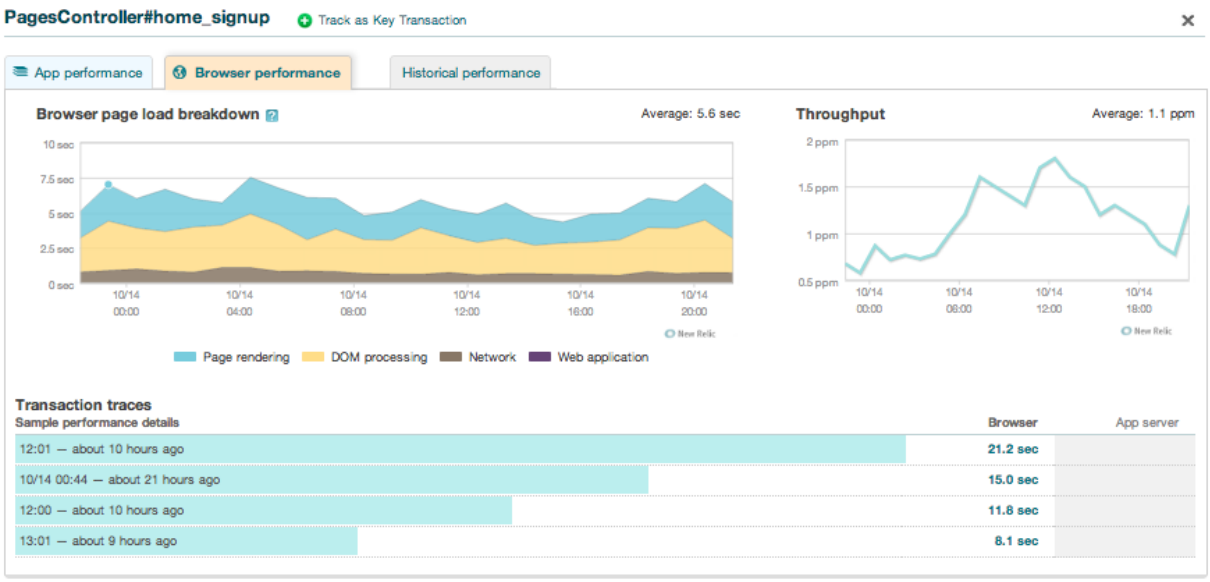


Figura 2.4: New Relic

A ferramenta possui a capacidade para monitorizar qualquer camada da arquitectura MVC, chegando ao ponto de organizar a informação recolhida por diferentes filtros. Esta informação pode ser relativa a *Queries SQL*, a tempos de resposta da base de dados, a chamada de métodos no servidor para executar alguma acção, a serviços externos que estejam acopulados à aplicação *web*, entre outros.

Para o ambiente de desenvolvimento *Rails*, existe a possibilidade de criar mais métricas de análise à aplicação, para que seja ainda mais fácil descobrir em que parte do código se encontra o problema. Em

modo de desenvolvimento também se consegue saber o fluxo de chamadas ao servidor feitas no *browser*, e, em cada chamada, pode saber-se quanto tempo demorou a carregar dada página e quais os métodos necessários, bem como os seus tempo de execução. No caso de *Queries SQL*, a própria *query* executada na base de dados é apresentada.

Por fim, o único senão acerca da aplicação é possuir um plano pago em que oferece todas as funcionalidades que não são oferecidas no plano normal e, neste caso, perdem-se algumas funcionalidades que podem ser actualmente consideradas importantes numa aplicação *web*.

Capítulo 3

Plataformas de Desenvolvimento Web

Na área de desenvolvimento de aplicações *web*, o modo de criação de um *website* é feito, normalmente, através do uso de uma plataforma capaz de abstrair as funcionalidades mais básicas de comunicação entre o servidor e o cliente para que assim seja retirado do programador a maioria das preocupações inerentes a essa comunicação. Actualmente, existe um grande conjunto de plataformas de desenvolvimento *web* que abstraem essas funcionalidades e, para além disso, oferecem a capacidade de um desenvolvimento rápido e estruturado, isto é, implementam a arquitectura MVC que permite uma separação simples entre as diferentes partes que constituem uma aplicação *web*.

Então, no contexto da comparação de duas diferentes plataformas *web*, é obrigatório estudar e conhecer as características e funcionalidades que cada uma dessas plataformas tem para oferecer. Assim ao longo deste capítulo, descrevem-se as duas plataformas, *Java Web* e *RoR*, que serão comparadas posteriormente através da técnica de *profiling*, bem como as linguagens que se encontram por detrás delas.

3.1 Plataforma Java EE

A plataforma Java EE é uma plataforma de desenvolvimento de aplicações empresariais desenvolvida inicialmente pela *Sun Microsystems* que posteriormente foi comprada pela *Oracle* na linguagem Java. Então ao longo desta secção, será descrita um pouco da história da linguagem Java, da plataforma Java EE e das características que tornaram o Java como uma das linguagens de eleição para desenvolvimento de aplicações *web*.

3.1.1 Linguagem de Programação Java

A linguagem Java começou a ser desenvolvida no início dos anos 90, numa pequena equipa de engenheiros de *software* da empresa *Sun Microsystems*, chefiada por James Gosling, com o objectivo inicial de desenvolver uma pequena linguagem para equipamentos electrónicos com *chips* programáveis. Essa pequena linguagem tinha como requisitos principais a robustez, a segurança, o baixo custo e a independência dos

chips [7].

Então, em 1992, a equipa desenvolveu a linguagem Java, designada na altura como *Oak*, que era simples, segura e independente de arquitecturas e possuía um compilador que gerava código intermédio, ou seja, independente que é designado por *byte-code*. Para além do compilador para *byte-code*, também criaram uma máquina virtual - a *Java Virtual Machine* (JVM) - que permite executar qualquer código gerado por esse compilador em qualquer plataforma computacional, isto é, a JVM é capaz de converter o *byte-code* em código-máquina para funcionar em qualquer processador independentemente da sua arquitectura [7].

No ano de 1995, a linguagem Java foi oficialmente anunciada, já contando com o apoio de empresas como a *Netscape*, a *Oracle*, a *Symantec* e outras que possibilitaram uma aceitação generalizada no ambiente empresarial e propulsionaram o seu crescimento, chegando ao ponto de ser actualmente a linguagem mais usada no mundo. Actualmente, Java encontra-se na versão 8 que foi lançada em Março de 2014.

Para além do que já foi referido, a importância da linguagem Java e das suas características para o universo das linguagens orientadas pelos objectos pode ser visto pelo surgimento, em 2001, da linguagem C# desenvolvida pela Microsoft. Esta linguagem é uma visível cópia do Java, visto que possui uma semântica muito idêntica e um sistema hierárquico de classes parecido, o que demonstra o impacto da linguagem Java no mundo da Engenharia de Software. E esse impacto foi atingido através das seguintes características :

- Possui um vasto conjunto de bibliotecas ou API's que concedem ao programador funcionalidades cruciais numa aplicação mas que muitas das vezes são consideradas repetitivas;
- Capacidade para ser executada em qualquer sistema operativo;
- Desalocação de memória automática usando para isso o *Garbage Collector* (GC);
- Possibilidade de reescrever um método declarado pela classe-mãe (*SuperClass*);
- Possuir poliformismo;
- Classes/Tipos Genéricos.

3.1.2 Java EE

A plataforma *Java Enterprise Edition* ou Java EE é uma plataforma de desenvolvimento *web* em Java. *Java EE* extende o *Java SE*, fornecendo uma *API* para mapeamento objecto-relacional, distribuídos e arquitecturas multicamada. A plataforma incorpora um design baseado em grande parte em componentes modulares visíveis na figura 3.1, que funcionam num servidor de aplicações, como o *Tomcat* ou *Glassfish*. Alguns desses componentes serão descritos mais abaixo, como forma de ressaltar a sua importância durante o desenvolvimento de uma aplicação *web* em Java. A plataforma Java EE foi lançado pela primeira vez em 1999 e actualmente encontra-se na versão 7 que foi lançada ao público em 2013.

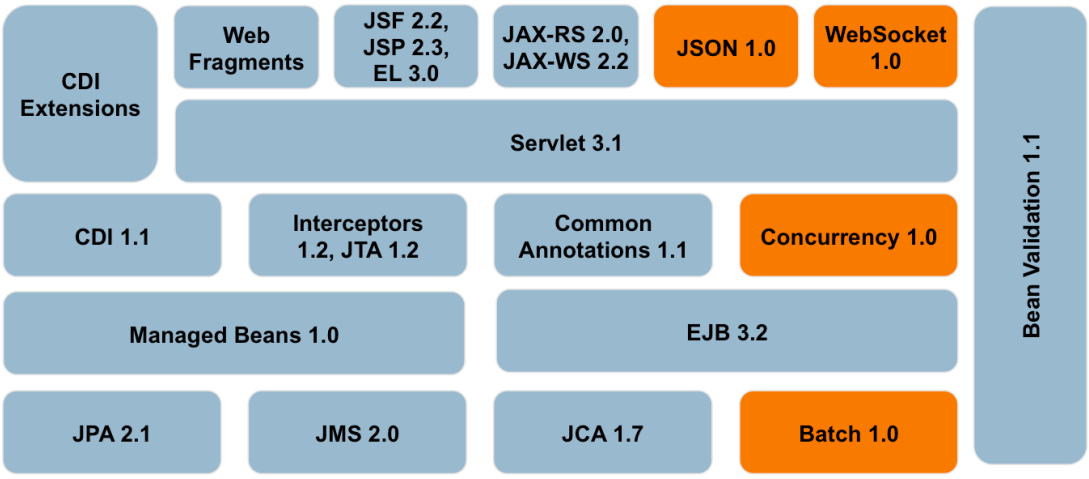


Figura 3.1: Arquitectura tecnológica da plataforma Java EE

Na figura 3.2, é exposta a forma como se consegue implementar o MVC na plataforma Java EE. Os componentes presentes na imagem serão brevemente descritos no fim desta secção. É de ressaltar que a implementação demonstrada na imagem é simples, pois é possível adicionar ainda mais componentes a cada uma das camadas do modelo refinando assim cada vez mais o modelo MVC. Entre estes componentes adicionais, encontram-se o *PrimeFaces*, o *Apache Struts*, o *Spring* e o *Hibernate* mas, por enquanto, não será explicado o que cada destes componentes tem para oferecer a uma aplicação Java, pois nem todos os componentes referidos serão utilizados posteriormente.

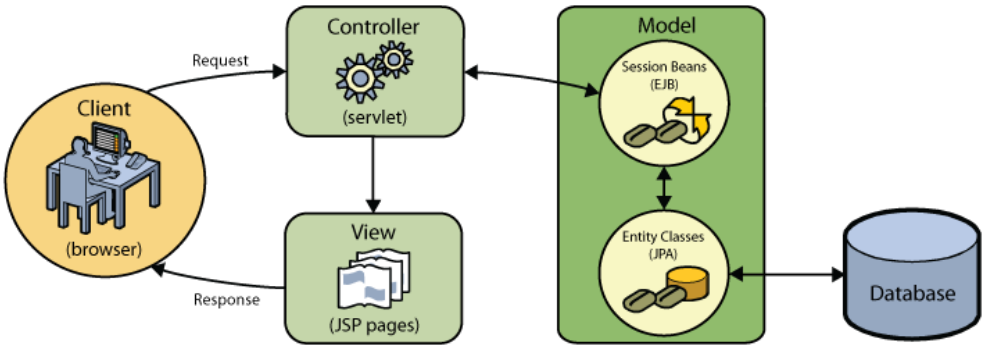


Figura 3.2: Arquitectura de uma aplicação em Java EE

Por fim e como já referido, a plataforma Java EE é baseado em módulos ou componentes, que se designam *Containers* ou *API's*. O número de *containers* existentes é grande e cada um deles possui a sua importância no desenvolvimento de aplicações Java. Na lista seguinte, descrevem-se apenas os mais comuns, que por vezes são a base tecnológica de outros componentes :

- **Java Database Connectivity (JDBC)** : Utilizada para implementar a conexão a uma base de dados;

- **Servlets** : Utilizados para o desenvolvimento de aplicações Web com conteúdo dinâmico. São uma *API* que abstrai os pedidos HTTP enviados pelo utilizador, permitindo assim uma maneira simplificada para o programador tratar de todo o fluxo inerente à resposta de um pedido;
- **JavaServer Pages (JSP)** : Uma especialização de *Servlet* que permite que conteúdo dinâmico seja facilmente desenvolvido;
- **Java Transaction API (JTA)** : Trata das transacções dentro de uma aplicação Java;
- **Enterprise Java Beans (EJB)** : Utilizados no desenvolvimento de componentes de software. O seu principal objectivo consiste em fornecer um desenvolvimento rápido e simplificado de aplicações Java, com base em componentes distribuídos, transaccionais, seguros e portáveis.
- **Java Persistence API (JPA)** : Padrão para persistência de dados que deve ser implementada por *frameworks* que desejem seguir tal padrão. Entre os *frameworks* que implementam *Java Persistence API* (JPA) encontram-se : *Hibernate*, *OpenXava*, *EclipseLink* e *Oracle Toplink*.

3.2 Plataforma Ruby on Rails (RoR)

A plataforma de aplicações *web Ruby on Rails*, muitas vezes referida como *Rails*, é um *framework open-source* que promete aumentar a produtividade e fácil criação de aplicações *web* por parte de programadores com menos experiência na criação de *websites*. Mas antes de discutirmos o *framework* em si, é preciso referirmos a linguagem na qual é desenvolvida, Ruby. É comum pensar que Ruby e Rails são a mesma coisa e apesar de ambas estarem intrinsecamente relacionadas, são diferentes pois possuíam propósitos diferentes quando foram criadas.

Ao longo desta secção, será descrita um pouco da história da linguagem Ruby, da plataforma RoR e das características únicas que ambas possuem e que levaram ao grande crescimento de ambas nos últimos anos.

3.2.1 Linguagem de Programação Ruby

A linguagem Ruby foi criada por Yukihiro "Matz" Matsumoto em 1995 e é uma linguagem livre, dinâmica e orientada pelos objectos, cujo propósito inicial seria ser uma linguagem de *scripting*. Esta linguagem une os princípios das linguagens favoritas de Matsumoto (*Python*, *Perl*, *Smalltalk*, *Eiffel*, *Ada* e *Lisp*) na tentativa de ser uma linguagem de *script* mais poderosa do que *Perl* e mais orientada pelos objectos do que *Python*, criando assim um equilíbrio entre a programação funcional e a programação imperativa[8]. O objectivo principal era criar uma linguagem natural mas isto não significa que Ruby seja uma linguagem de fácil aprendizagem. Matsumoto normalmente explica esta diferença dizendo que:

> *"Ruby is simple in appearance, but is very complex inside, just like our human body."*[9]

Actualmente, Ruby é a 13ª linguagem de programação mais popular do mundo, de acordo com o Índice

Tiobe[10]. Esta popularidade deve-se muito ao crescimento da plataforma RoR.

```ruby
class Pessoa
  # Getters and Setters
  attr_accessor :nome, :idade
  #Construtor vazio e com paramêtros
  def initialize(nome="Desconhecido",idade=0)
    self.nome = nome
    self.idade = idade
  end
  # Método para comparar duas Pessoas
  def >(pessoa)
    self.idade > pessoa.idade
  end
  # Re-definição do método to_s
  def to_s
    "#{self.nome} (#{self.idade})"
  end
  # Re-definição do método de comparação de objectos
  def ==(pessoa)
    if pessoa.equal?(self)
      true
    elsif (pessoa == nil || !pessoa.instance_of?(self.class))
      false
    else
      self.nome == pessoa.nome && self.idade == pessoa.idade
    end
  end
  # Re-definição do clone da classe Object
  def clone
    Pessoa.new(self.nome,self.idade)
  end
end
```

Código 3.1: Exemplo de Código Ruby

Nos dias de hoje, existem várias implementações de Ruby e cada uma tem o seu compilador. Cada implementação possui características únicas e estão direccionadas para diferentes ambientes. Entre as várias implementações, encontram-se as seguintes: **YARV**, **JRuby**, **Rubinius**, **IronRuby**, **MacRuby** e **HotRuby**.

Como já foi dito, Ruby é uma linguagem dinâmica e é interpretada durante o tempo de execução usando *Duck Typing*. Em linguagens orientadas pelo objectos, *Duck Typing* é um estilo de tipagem dinâmica na qual os métodos e as propriedades dos objectos determinam a correta semântica ao contrário da herança de uma classe ou implementação de uma interface que é usado por exemplo pela linguagem Java[11].

Em Ruby, não existem tipos primitivos, pois todos os tipos são objectos, onde a classe mãe é *Object*. Todas as classes são abertas, isto é, a qualquer momento é possível alterar um método ou propriedade de uma determinada classe. Isto também significa que qualquer programador pode alterar até as classes padrão do Ruby para satisfazer as suas necessidades durante o desenvolvimento de uma aplicação. Há que apontar que esta alteração às classes oferecidas pelo Ruby não é recomendada pois pode interferir com

alguma classe que tenha como sua base uma das classes que foi alterada pelo programador.

3.2.2 Ruby on Rails

Ruby on Rails é um framework *open-source* de desenvolvimento *web* desenvolvida sobre a linguagem Ruby. Foi desenvolvida inicialmente por David Heinemeier Hansson como uma extracção de um trabalho realizado no Basecamp, um gestor de projectos da empresa 37signals. Rails foi lançada ao público em Julho de 2004 e actualmente a última versão estável foi lançada em Abril de 2014.

É um framework *full-stack*, isto é, oferece todos os componentes necessários para uma aplicação *web*, como a biblioteca de ligação à base de dados (ActiveRecord), a arquitectura MVC, geração automática de código através de *scaffolding*, entre outros, como se pode observar na figura 3.3.

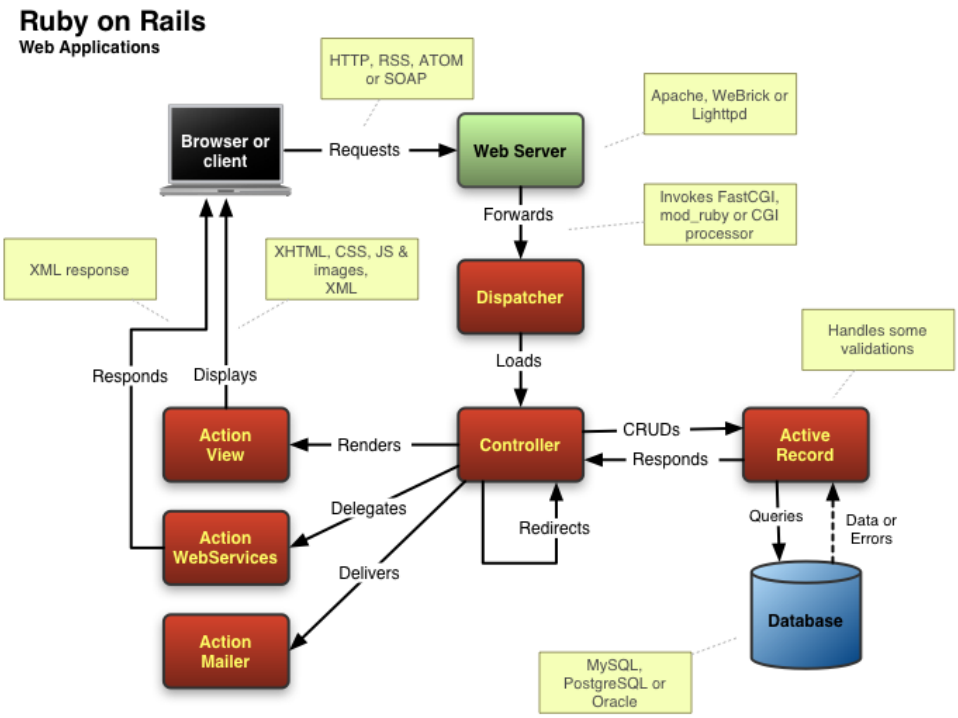


Figura 3.3: Arquitectura de uma aplicação em RoR

O *framework* em si salienta o uso dos conhecidos padrões e princípios de Engenharia de Software, como *Active Record* (usado nas classes que representam a base de dados), *Convention over Configuration* (*CoC*), *Don't Repeat Yourself* (*DRY*), *Representational State Transfer* (REST) e *Model-View-Controller* (MVC). Para além de seguir os padrões e os princípios já mencionados, o *framework* é conhecida por ter um desenvolvimento ágil e fácil. Em 2005, o seu criador, David Hansson, divulgou um vídeo em que demonstrava como se implementava um *Weblog* em 15 minutos, o que na altura espantou toda a comunidade.

Um dos grandes motivos para o rápido crescimento e aceitação no mundo empresarial do *framework*

deve-se muito a uma comunidade muito activa que fornece muitas funcionalidades consideradas muitas vezes importantes, mas que custam tempo de desenvolvimento, num simples pacote de software mais conhecido como *GEM*.

Então, *Ruby on Rails* é um *meta-framework*, isto é, um *framework* que é composto por mais pequenos *frameworks*, expostos na figura 3.3, sendo eles os seguintes:

- **Active Record** : É a camada que mapeia os objectos/relações existentes numa base de dados. Acompanha de perto os padrões do modelo *Object-Relational Mapping* (ORM), que são: tabelas mapeadas em classes; linhas mapeadas em objectos; e colunas mapeadas em atributos das classes. Cada classe que implementa-o tem os métodos *Create-Read-Update-Delete* (*CRUD*). Para além dos métodos CRUD, o *ActiveRecord* também oferece métodos de pesquisa na base de dados [12] [13].
- **Action Pack** : Divide a resposta a um pedido HTTP em duas partes, a parte do controlador (*ActionController*) que realiza a lógica necessária e a parte relativa à renderização da página *web* (*ActionView*). Esta abordagem em duas partes é chamada de acção ou *action*, que efectuará um método *CRUD* sobre um ou mais modelos da base de dados antes de renderizar o *template* correspondente ao método *CRUD* ou redireccionar para outra acção. O *ActionPack* implementa estas acções como métodos públicos nos *ActionControllers* e usa os *ActionViews* para apresentar os *templates* criados que contém Ruby embebido em *HTML* [12] [13].
- **Action Mailer** : Permite a criação de um serviço de *e-mails*. Com este *framework*, é possível criar *templates* para o envio de *e-mails* e também permite a recepção dos mesmos [12] [13].
- **Active Support** : É uma extensa colecção de classes úteis e de extensões à biblioteca da linguagem Ruby que suporta todos os componentes da arquitectura *Rails*. Algumas destas classes estão destinadas a uso interno pelo *framework*, contudo toda a biblioteca está disponível para ser utilizada em aplicações que não sejam RoR [12] [13].

Capítulo 4

Cartografia de Facetas de Profiling

Uma faceta é o que distingue particularmente um objecto de outro, possibilitando assim uma comparação exacta e objectiva. Neste caso, as facetas que uma ferramenta de *profiling* analisa serão características únicas que uma execução de um programa possui e que serão úteis para distinguir aplicações diferentes mas com o mesmo propósito.

Assim, ao longo deste capítulo, serão apresentadas as facetas que as ferramentas de *profiling*, referidas na secção 2.3, investigam e que se encontrarão separadas pela plataforma de implementação. Esta separação será feita devido ao facto de não haver um ferramenta transversal às linguagens de programação, isto é, que independentemente da linguagem usada na aplicação seja capaz de recolher sempre as mesmas facetas. E também irá possibilitar uma fácil comparação e escolha da ferramenta a usar dentro do mesmo ambiente de desenvolvimento.

Após a exposição das facetas recolhidas, serão escolhidas as facetas de *profiling* que posteriormente servirão como suporte de comparação entre as duas plataformas do estudo e apresentadas numa tabela única para proporcionar uma fácil leitura. Por fim, serão apresentados os motivos pelo quais se optou por uma faceta em vez de uma outra para efectuar a comparação, tendo em consideração que o estudo pretende focar-se, maioritariamente, na análise da arquitectura MVC e das possíveis facetas que se encontrem dentro desse espectro. Também serão expostas as definições para cada uma das facetas escolhidas, bem como a unidade métrica usada por cada uma delas.

4.1 Facetas de Profiling Analisáveis

Com o intuito de recolher todas as facetas que as ferramentas de *profiling* permitem observar, foi necessário investigar cada uma delas através sua da execução com uma aplicação simples para conseguir saber ao certo o que analisam. A recolha das facetas foi um processo bastante exaustivo, visto que é necessário configurar e perceber a forma como cada *profiler* funciona e também tentar perceber em que cada categoria de *profiling* se encaixam as facetas analisadas pela ferramenta, visto que nenhum *profiler* possui uma orga-

nização clara e que permita um mapeamento directo para a arquitectura MVC. Porém, as várias definições e as categorizações usadas pelos diferentes *profilers* conseguem-se aproximar do que é pretendido com o estudo multi-camada de uma aplicação *web*.

Antes de prosseguir com a cartografia das facetas recolhidas em cada uma das plataformas *web*, é importante referir que na figura 4.1, abaixo apresentada, encontra-se presente a coloração atribuída a cada categoria de *profiling*, descritas na secção 2.2, e que será usada ao longo do documento para permitir uma distinção simples a que categoria pertencem as facetas analisadas.

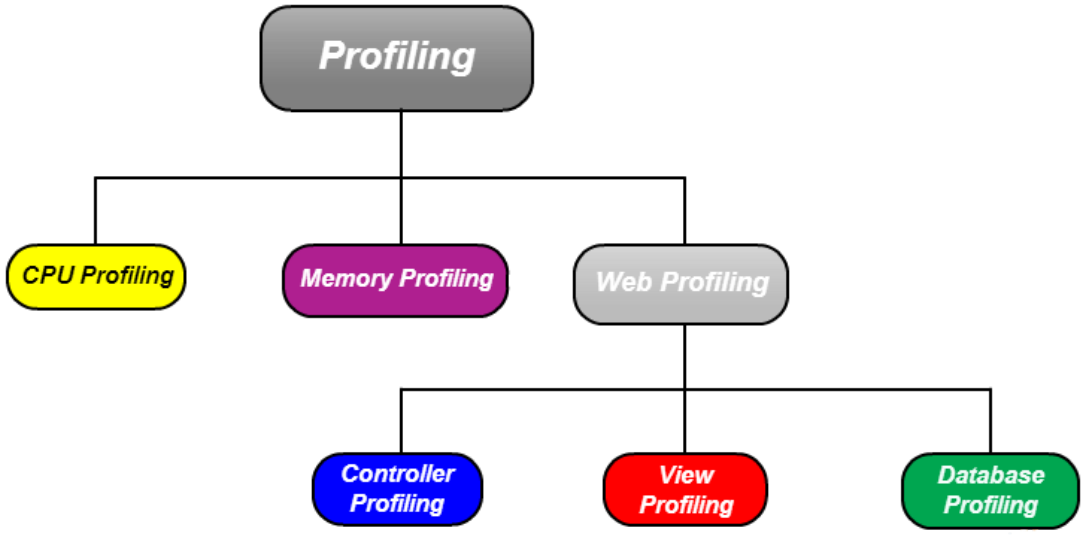


Figura 4.1: Categorias de Profiling

Portanto, nas tabelas 4.1 e 4.2 abaixo apresentadas, encontram-se presentes todas as facetas possíveis de observar através da execução e análise da ferramenta de *profiling* em questão ou então através da pesquisa na página *web* e na documentação referente a cada ferramenta. É importante esclarecer que qualquer propriedade recolhida e analisável será considerada independentemente se um dos *profilers* não analisar e outro possibilitar o reconhecimento da mesma. Não será explicado por enquanto o significado de cada uma das propriedades, devido ao facto de se tratarem de várias facetas e de cada ferramenta possuir uma designação diferente, o que se tornaria um processo demasiado exaustivo e dificultaria a percepção que se pretende pela recolha inicial das facetas que posteriormente serão seleccionadas para que se possua a mesma base de comparação entre as plataformas *web* usadas no estudo. Por fim, é decisivo também referir que os nomes usados para as facetas nas tabelas abaixo apresentadas se encontram todos em inglês, visto que as ferramentas de *profiling* se encontram em inglês e também para manter uma certa consistência de nomes entre as ferramentas e as facetas extraídas delas.

Profiler / Faceta	VisualVM	NetBeans Profiler	YourKit	JProfiler	New Relic
Wall Clock Time	●	●	●	●	●
CPU Load				●	
CPU Time	●	●	●	●	
Total Time				●	
Median Time				●	
Min Time				●	
Max Time				●	
Standard Deviation				●	
Outlier Coefficient				●	
GC Activity	●	●		●	
GC Time		●	●		●
GC Objects			●		
GC Size	●	●	●		
Classes Size	●	●		●	
Number of Instances	●	●		●	
Loaded Classes	●	●	●	●	●
Unloaded Classes	●		●		●
Heap Memory Usage	●	●	●	●	●
Non-Heap Memory Usage			●	●	
Average Response Time			●	●	●
Action Average Response Time				●	●
Action Average Throughput					●
Action Count				●	
Average Page Creation Time					●
Average Page Loading Time					●
Page Average Throughput					●
Page Loading Time			●	●	
Average Time in DB Operations					●

Continua na próxima página

Average Query Response Time			•	•	•
Query Time			•	•	•
Query Average Throughput					•

Tabela 4.1: Facetas das Ferramentas da plataforma Java EE

Faceta \ Profiler	MRI/REE	JRuby	Rubinius	MiniProfiler	New Relic
Wall Clock Time	•	•	•		•
Process Time	•				•
CPU Time					
User Time					
CPU Burn					•
GC Runs	•				•
GC Time	•				•
Memory	•				•
Objects	•				
Average Response Time					•
Action Average Response Time					•
Action Average Throughput					•
Average Page Creation Time					•
Average Page Loading Time					•
Page Average Throughput					•
Page Loading Time				•	
Page Rendering Duration				•	
Average Time in DB Operations					•
Average Query Response Time					•
Query Time				•	•
Query Average Throughput					•

Tabela 4.2: Facetas das Ferramentas da plataforma Ruby on Rails

4.2 Facetas de Profiling Seleccionadas

Após a exposição das facetas recolhidas, é necessário efectuar uma selecção de quais facetas serão usadas posteriormente na comparação das duas plataformas *web*. Durante esse processo de selecção das facetas,

foi tido em consideração o objectivo principal por detrás do estudo, a comparação de duas plataformas que implementam a arquitectura MVC usando para tal propriedades inerentes a cada uma das camadas dessa arquitectura. Além disso, também foi levado em consideração qualquer outra faceta que não se encontrasse dentro das características do MVC e pudesse ajudar à comparação entre duas plataformas *web*, mesmo que essa faceta, por exemplo, representasse algo relacionado com o *CPU* ou a Memória.

Facetas	Unidade Métrica
CPU Time	Milissegundos (**ms**)
Wall Clock Time	Milissegundos (**ms**)
Average Response Time	Milissegundos (**ms**)
Action Average Response Time	Milissegundos (**ms**)
Action Average Throughput	*Requests Per Minute* (**RPM**)
Average Page Creation Time	Milissegundos (**ms**)
Average Page Loading Time	Segundos (**s**)
Page Average Throughput	*Pages Per Minute* (**PPM**)
Average Time in DB Operations	Milissegundos (**ms**)
Average Query Response Time	Milissegundos (**ms**)
Query Average Throughput	*Calls Per Minute* (**CPM**)

Tabela 4.3: Facetas de Profiling

Então, o resultado desse minucioso processo de selecção das facetas a serem usadas na comparação das plataformas é exposto na tabela 4.3, acima apresentada. Nessa tabela, é possível verificar que o número de facetas diminui consideravelmente em relação à cartografia apresentada na secção anterior e que as facetas que foram escolhidas encontram-se ligadas sobretudo à categoria de *Web Profiling*, o foco principal do estudo, com as únicas excepções sendo o *CPU Time* e o *Wall Clock Time* que se referem à categoria de *CPU Profiling*. Na tabela anterior, também é possível verificar a unidade métrica usada por cada faceta durante a execução dos testes de *profiling*. E assim, apenas fica apenas a faltar a descrição individual de cada faceta para uma melhor compreensão do que cada uma delas pretende observar acerca da execução de uma aplicação *web*. Na seguinte lista, é apresentada as definições das facetas a serem utilizadas posteriormente na comparação das duas plataformas *web*.

- **CPU Time** : O tempo realmente gasto pelo CPU na execução do código de um método ou função;
- **Wall Clock Time** : O tempo do mundo real decorrido entre a entrada e a saída de um método. Se houver outros processos a executar simultaneamente no sistema, estes podem afectar os resultados obtidos por esta faceta;

• **Average Response Time** : O tempo médio que demora a executar todos os métodos relativos a um pedido do utilizador, sendo que esses métodos apenas se referem a execuções que decorrem na camada relativa ao *Controller*, sem contar com o tempo passado na execução de *queries* à base de do dados ou na criação da página *web* com os resultados da execução da acção no *Controller*;

• **Action Average Response Time** : O tempo médio que demora a executar uma acção requisitada pelo utilizador à aplicação. Dentro deste tempo médio, é considerado todas as execuções de métodos necessários para resposta ao utilizador, ou seja, a execução de métodos relativos ao MVC;

• **Action Average Throughput** : O número médio de pedidos ao servidor num minuto de uma dada acção. Esta faceta é, ao contrários das demais, uma taxa que mede o número de pedidos recebidos num minuto pelo utilizadores;

• **Average Page Creation Time** : O tempo médio que é necessário para que uma página *web* seja criada no servidor, isto é, durante a execução de um pedido do utilizador, o tempo gasto na camada relativa à *View*;

• **Average Page Loading Time** : O tempo gasto em média para que uma página *web* seja carregada pelo *browser web*. Normalmente, esta faceta pode sofrer grandes variações devido à velocidade de *download* da Internet do utilizador e também à má criação de uma página *web* por parte do programador;

• **Page Average Throughput** : O número médio de páginas servidas por minuto aos utilizadores da aplicação *web*.

• **Average Time in DB Operations** : O tempo médio gasto por uma acção do *Controller* em operações relativas ao *Model* da aplicação. Nesta faceta, é contabilizado todo o tempo usado em *queries* à base de dados num determinado pedido do utilizador;

• **Average Query Response Time** : O tempo médio de resposta de uma *query* à base à dados, independentemente do conteúdo da *query* e dos dados enviados na resposta;

• **Query Average Throughput** : O número médio de chamadas à base de dados por minuto de uma dada *query*.

Capítulo 5

A Aplicação Web

Após a introdução, no capítulo 3, de cada uma das linguagens e respectivas plataformas *web* a usar durante o estudo comparativo e da recolha e elaboração da cartografia das facetas de *profiling* do capítulo 4, é agora importante desenvolver uma aplicação em cada uma das respectivas plataformas *web* com o intuito de possuir uma base de comparação entre elas.

Portanto, durante este capítulo, descreve-se a aplicação desenvolvida e o propósito que se pretende atingir durante a realização dos testes de *profiling*. Após esta descrição, apontam-se algumas decisões tomadas durante a concretização de cada umas aplicações, bem como as alterações ou adições de software necessárias para colocar a aplicação *web* no estado de produção, ou seja, uma aplicação pronta a ser usada pelo utilizador final e que possua a capacidade para responder correctamente a muitos pedidos simultaneamente. Por fim, será explicado qual foi a técnica de Engenharia de Software necessária para atingir essa capacidade de resposta simultânea e das tecnologias usadas em cada uma das plataformas que ajudaram a atingir esse objectivo.

5.1 Conceito da Aplicação

Com o intuito de fazer uma comparação justa entre duas plataformas web, é necessário ter uma aplicação, que independente das tecnologias que se encontrem por detrás do seu desenvolvimento, responda aos mesmos requisitos funcionais e que o resultado final seja similar. Então, para que o desenvolvimento da aplicação decorresse suavemente, foi importante saber quais eram os requisitos funcionais que ambas as aplicações deveriam respeitar, bem como os *websites* de onde foram retirados alguns desses requisitos e os dados a serem usados por cada uma das aplicações. Assim, para descrever os requisitos funcionais de uma forma mais simples e perceptível, há que começar pelos lugares na *Internet* de onde foram retirados os requisitos que serão expostos posteriormente.

Assim sendo, nas figuras 5.1a e 5.1b, é possível observar os lugares de onde foram retirados, maioritariamente, os requisitos funcionais que as aplicações web a desenvolver deverão cumprir. Na figura 5.1a,

encontra-se uma aplicação web com o objectivo de permitir ao utilizador reservar um táxi. Para tal, é feita uma pesquisa de táxis livres usando uma lista de países disponíveis e das cidades associadas, o número de passageiros desejados e a data em que se pretende o serviço. Também é dada a hipótese de reservar um táxi com uma data de retorno, ou seja, o mesmo táxi irá fazer o percurso inverso na data escolhida. Há que apontar que os táxis existentes em cada cidade podem variar, isto é, algumas cidades poderão não conseguir responder às necessidades do utilizador, quer em termos de número de passageiros ou preço dos táxis desocupados.

(a) Airport Taxi Transfers (http://airportstaxitransfers.com) (b) Rental Cars (http://www.rentalcars.com)

Figura 5.1: *Websites* usados como base para a aplicação web

Na figura 5.1b, encontra-se outra aplicação web com um objectivo semelhante à anterior. Neste *website*, é dado ao utilizador a hipótese de alugar carros, como normalmente acontece nos aeroportos onde existem agências que fornecem este serviço. O aluguer do carro é feito, como anteriormente, através de uma pesquisa de carros disponíveis usando para tal uma lista de países e de cidades associadas a cada país como filtro. A única diferença para a pesquisa, referida acima, é que os carros pertencem sempre a uma ou mais agências de aluguer e que cada agência nem sempre se encontra em todos as cidades disponíveis no sistema. Há que referir também a possibilidade de devolver o carro alugado numa cidade diferente daquela onde foi alugado.

Então, após as descrições das fontes para os requisitos, é essencial listá-los para que fique explícito o que a aplicação desenvolvida permite fazer e os objectivos pretendidos para a comparação das plataformas web. A lista abaixo pretende, de uma forma genérica, apresentar esses requisitos com uma breve e simples descrição. Há a denotar que para o utilizador efectuar qualquer acção no website deverá encontrar-se registado e com a sessão iniciada na aplicação.

- O utilizador deve poder efectuar uma pesquisa de carros/táxis usando para tal um formulário de pesquisa com os filtros necessários;

- O utilizador deve poder reservar/alugar um carro/táxi após a confirmação dos pessoais e o devido pagamento do montante;
- O utilizador deve poder refazer a pesquisa a qualquer momento;
- O sistema deve possuir uma lista com os países e de cidades associadas em que é possível reservar/alugar um veículo;
- O sistema deve calcular os custos de reserva de um táxi com base da distância percorrida;
- O sistema deve possuir uma lista de agências de automóveis com as devidas subsidiárias;
- O sistema deve verificar os dados do pagamento do utilizador;
- O sistema deve manter as reservas e alugueres efectuados pelos utilizadores.

E, depois da listagem dos requisitos acima apresentada, falta mencionar qual é o impacto que estes requisitos pretendem aplicar na comparação das plataformas web. Então, o principal impacto e objectivo por detrás dos requisitos mencionados é possuir uma pesquisa mais demorada, isto é, que não seja uma simples procura na base de dados para possibilitar assim uma base para a comparação mais plausível do que uma simples aplicação web que trate apenas da gestão de registos.

5.2 Implementação em Java

Numa aplicação desenvolvida na plataforma Java Web e sem o uso de um *framework full-stack* específico, como o *Spring*, o *Grails* ou a *Play Framework* que oferecem todos os componentes que tratam da implementação da arquitectura MVC, tornou-se crucial estudar as diversas possibilidades existentes no mundo Java que tratassem individualmente de cada uma das camadas e que também em conjunção não gerassem nenhuma incompatibilidade entre elas.

Antes de apresentar as tecnologias escolhidas para cada camada, é imperativo referir que algumas das tecnologias que serão apresentadas oferecem e tratam da implementação de várias *API's* da plataforma Java *Web*, já enunciadas na secção 3.1.2, e que foram seleccionadas preferencialmente por serem as mais usadas na comunidade de Java quando se trata de criar uma aplicação web através da junção de várias tecnologias. Assim, as principais tecnologias da plataforma Java *Web* escolhidas foram o *Apache Struts 2* e o *Hibernate*.

Portanto, o *Struts 2*, como é normalmente referenciado e que se encontra na versão *2.3.16*, usa e extende a *Java Servlet API* para encorajar os programadores a implementarem a arquitectura MVC nas suas aplicações web. Entre as características principais para a sua escolha pode-se enumerar as seguintes:

- Uso de *Annotations* ao invés de ficheiros de configuração em XML;
- Uso do paradigma *Convention over Configuration*, tornado famoso pelo framework RoR;
- Suporte a pedidos *AJAX* através do *plugin* para *JQuery*;
- Suporte a diferentes tipos de resposta, como *JSON*, *XML* e outros;

- Suporte a pedidos *HTTP* do utilizador usando a técnica REST e;
- Facilidade de extender o *framework* através de *plugins*.

E com este *framework*, as camadas referentes ao *Controller* e à *View* ficaram asseguradas. Assim, a camada em falta da arquitectura MVC é o *Model*. Para colmatar esse camada, optou-se pelo *Hibernate*, que se encontra na versão *4.3.6*, como ORM para a aplicação desenvolvida. O *Hibernate* oferece uma implementação da *Java Persistence API* que permite criar *queries* dinâmicas através da utilização da *Criteria API*. Para além de permitir a criação e execução de *queries*, também o mapeamento das tabelas da base de dados em classes Java é efectuado usando para isso ou *Annotations* ou ficheiros *XML*.

Após a introdução das tecnologias usadas durante o desenvolvimento da aplicação e que foram configuradas para funcionarem em conjunto, basta somente expor o software necessário para libertar a aplicação web para *Internet*. Assim, foi essencial estudar o software que permitisse executar aplicações *web* na plataforma Java e assim optar pela melhor opção para o servidor. Durante esse estudo, foi perceptível a necessidade de separar o servidor em duas partes mais simples mas mesmo assim capazes de fornecer a aplicação ao utilizador com segurança e rapidez.

Então, uma dessas partes, à qual normalmente se refere como *reverse proxy*, tratará de encaminhar os pedidos dos utilizadores para um conjunto de servidores internos e assim garantir uma distribuição eficaz do tráfego gerado pelos utilizadores. Enquanto que a outra parte, que irá corresponder aos servidores internos já referidos e é normalmente chamado de servidor aplicacional, trataria de executar a aplicação e responder a todos os pedidos provenientes do *reverse proxy*.

Assim sendo, as tecnologias escolhidas que satisfarão estas duas partes e que irão criar um servidor *web* com a aplicação desenvolvida são o *Nginx* que servirá como *reverse proxy* e o *Apache Tomcat* que fará o papel de servidor aplicacional. É importante referir que o processo de instalação e configuração de todo o ambiente que permite executar aplicações web desenvolvidas na plataforma Java, encontra-se detalhado no apêndice B.

Começando pelo software que tratará de executar a aplicação *web*, o *Apache Tomcat* é um servidor *web* em Java, mais especificamente, um *container* de *servlets*. Este servidor implementa as tecnologias *Java Servlet* e *JavaServer Pages* e outras de menor relevância mas não implementa a tecnologia EJB, o que não permite executar toda a *API* referente às *Enterprise Java Beans*. Para além do *Tomcat*, também foram tido em conta os seguintes servidores de Java: *Glassfish*, *Apache TomEE*, *WildFly* e *IBM WebSphere*. Nenhum dos servidores referidos foi seleccionado pois, na maioria, implementem funcionalidades da plataforma Java EE que não serão usadas pela aplicação e que acrescentam um peso desnecessário aquando do *deploy* do *website*.

Por fim, o software escolhido para fazer de *reverse proxy* foi o *Nginx*, visto que, para além ser o escolhido para a mesma função na plataforma RoR, é o mais eficaz, robusto e de simples configuração em comparação

com o seu rival, o *Apache HTTP Server*. A escolha não recaiu sob o servidor *Apache*, porque optou-se por manter uma homogeneidade no servidor que trataria do *reverse proxy* em cada uma das plataforma em que foram desenvolvidas as aplicações.

5.3 Implementação em RoR

Na aplicação desenvolvida em RoR, não existiu a necessidade de adicionar ou configurar nenhuma tecnologia para tratar qualquer uma das camadas da arquitectura MVC pois, como mencionado na secção 3.2.2, a plataforma RoR é um *framework full-stack*, o que significa que a própria plataforma fornece as tecnologias que cuidam das camadas MVC de uma aplicação *web*. Assim, resta descrever o software necessário para criar um servidor que fará o *deploy* da aplicação para que possa ser acedido por qualquer utilizador.

Então, para poder ter uma aplicação *web* disponível na *Internet*, é preciso montar um servidor *web* capaz de fornecer essa aplicação para que o utilizador final a use. E para isso, foi necessário escolher tecnologias que possibilitassem um ambiente concorrente, fossem de fácil uso, oferecessem um reduzido consumo de memória do computador, uma boa *performance* e fossem o padrão usado na comunidade de RoR para o *deploy* de aplicações *web*.

Assim, segundo o *site* oficial do *Ruby on Rails* (http://rubyonrails.org), o padrão usado pela comunidade é ter o *Phusion Passenger* como servidor aplicacional e ou *Apache HTTP Server* ou *Nginx* como servidor *web* com *reverse proxy*, visto que o *Passenger* possui um módulo para ambos os servidores *web* que possibilita uma fácil configuração, gestão e *deploy* de aplicações.[14] É importante denotar que existe uma grande variedade de servidores aplicacionais para RoR, tais como *Puma*, *Unicorn*, *Mongrel* e *Thin*, mas a opção escolhido foi o *Phusion Passenger*[15][16].

Após a escolha do servidor aplicacional, falta apenas referir a opção escolhida para o servidor *web* que podia variar entre o *Apache HTTP Server* e o *Nginx*. Então, a escolha final recaiu sob o servidor web *Nginx*, visto que, na comunidade de RoR, é uma das soluções mais usadas para obter boa *performance* das aplicações desenvolvidas em RoR. Por fim, é importante referir que o processo de instalação e configuração de todas as tecnologias que permitem criar o servidor *web* e expor a aplicação final para a *Internet*, se encontra detalhado no apêndice A.

5.4 Connection Pooling

Em Engenharia de Software, *connection pooling* é uma técnica que permite que múltiplos clientes usem um conjunto de conexões reutilizáveis para executar *queries* a uma base de dados. Com esta técnica, consegue-se obter uma melhoria no desempenho da execução de comandos a uma base de dados, visto que já não é necessário que para cada comando executado seja aberta e fechada uma conexão, o que representa um grande custo em termos de recursos utilizados e tempo desperdiçado. Normalmente, é usada em

aplicações *web* com grande afluência de utilizadores para garantir uma boa experiência de utilização e um aumento substancial do tempo de resposta. Também pode ser usada em aplicações locais ou *stand-alone* que precisam de efectuar constantes pedidos à base de dados [17].

Então, para que as aplicações desenvolvidas nas plataformas já mencionadas anteriormente fossem capazes de aguentar com uma grande afluência de utilizadores e pedidos, foi necessário usar a técnica de *connection pooling* em conjunção com uma boa configuração dos servidores onde se encontram a funcionar para que, durante os testes de *profiling*, se obtivesse um desempenho significativo. De seguida, será exposto e descrito as tecnologias escolhidas em cada uma das plataformas que implementam esta técnica como forma de retirar a complexidade inerente à sua implementação.

Java Connection Pool

Na plataforma Java, para implementar a camada referente ao *Model* e à persistência foi escolhido o *framework Hibernate*. No contexto de *connection pooling*, este *framework* possui uma *connection pool* interna mas, como é referido na sua documentação, essa *connection pool* só deve ser usada em ambiente de desenvolvimento, devido a ser uma implementação muito rudimentar da técnica de *connection pooling* e de não assegurar uma boa gestão do conjunto da conexões à base de dados.

Assim, foi preciso encontrar uma solução tecnológica que oferecesse todos os benefícios de uma *connection pool*, se encontrasse pronta para ser usada em ambiente de produção e que também fosse facilmente integrada no *Hibernate*. Então, após a pesquisa de uma solução tecnológica em Java que implementasse a *connection pool* e ao mesmo tempo conseguisse ser rápida, eficaz e se mantivesse actualizada, surgiram as seguintes opções:

- *C3P0* ;
- *BoneCP* ;
- *HikariCP* ;
- *Apache Commons DBCP* ;
- *Tomcat-JDBC* ;
- *Vibur-DBCP* .

Portanto da lista anterior, a escolha tecnológica para implementar a *connection pool* na aplicação desenvolvida na plataforma Java foi o *C3P0*. A escolha recaiu sob esta *pool*, devido à sua facilidade de integração com o *Hibernate*, pois este já lhe oferece suporte imediato, bem como recomenda o seu uso no momento em que a aplicação passa para o ambiente de produção. Para além disso, o *C3P0* é uma das *connection pools* mais antigas, estáveis, usadas na comunidade Java, e também apresenta um bom desempenho em termos de gestão da *pool*. A sua configuração é feita através da adição de um ficheiro à aplicação, bem como de alguns atributos que se deve colocar na configuração do *Hibernate* que indicam-lhe o que utilizar para criar e gerir uma *connection pool*. Esses ficheiros de configuração podem ser encontrados no apêndice B, na secção B.6.

Em relação às demais *connection pools*, é de referir que a *HikariCP*, a *Vibur-DBCP* e a *BoneCP* foram

tomadas em consideração para efectuarem a gestão da *pool*, visto que, nos *benchmarks* apresentados em cada uma das suas páginas, elas possuíam melhor desempenho e melhores características/funcionalidades em relação à *C3P0* [18][19]. Mas, por fim, acabaram por não serem as escolhidas devido a um problema de integração com o *Hibernate*. Este problema surgiu devido às versões usadas pelas *connection pools* para se integrarem com o *Hibernate* não serem compatíveis, o que significa que não se encontram suficientemente actualizadas com a última versão do *Hibernate* usada. Já as duas restantes *pools* não foram alvo de experimentação devido à falta de documentação que ajudasse no processo de integração e também por ser necessário alterar a configuração inicial do servidor *Tomcat* para que se conseguissem funcionar com a aplicação.

Ruby on Rails Connection Pooling

Na plataforma *Ruby on Rails*, a técnica de *connection pooling* já se encontra implementada por padrão através do *framework ActiveRecord*, que para além do que foi descrito na secção 3.2.2, também consegue efectuar uma gestão eficaz das conexões e assim libertar o programador das preocupações inerentes à procura de uma implementação desta técnica. E para configurar e aumentar a *pool* de conexões em *Rails*, que possui como padrão o tamanho máximo de 5 conexões, basta adicionar ao ficheiro de configuração da base de dados (*database.yml*) o atributo "pool" e o tamanho desejado para ela [20].

Capítulo 6

Análise dos Resultados

Para se poder efectuar uma comparação entre as duas plataformas *web* do estudo, existem passos importantes a realizar antes de analisar os resultados e se retirarem conclusões. No desenrolar deste capítulo, serão apresentados os passos e as decisões tomadas para que fosse possível efectuar essa comparação das plataformas. Estas decisões e passos tomados, referem-se às ferramentas usadas para efectuar o *web profiling*, e à ferramenta escolhida para a criação do plano de teste que será executado para emular uma utilização aleatória das aplicações *web* por parte de vários utilizadores simultaneamente e alguns dos condicionantes utilizados durante a execução desses testes. Para além do que já foi mencionado, será apresentado um mapa das acções que as aplicações executam e que serão importantes posteriormente para a apresentação dos resultados obtidos durante o *web profiling*.

Por último, fazem-se algumas considerações a ter conta em relação ao processo de recolha dos resultados dos testes efectuados e também serão expostos os resultados obtidos em cada acção do mapa já referido, dividindo-os pelas categorias de *profiling* e com uma análise dos mesmos. Por fim, será feita uma análise à escalabilidade de ambas as aplicações usando como base a ferramenta *NewRelic* que fornece alguma informação acerca deste importante tema.

6.1 Criação de Cenários de Testes

Para que seja possível efectuar uma recolha e análise rigorosa sobre o desempenho de uma aplicação *web* antes de ela se encontrar exposta à Internet e ao utilizador final, é determinante possuir uma forma de gerar dados constantes capazes de imitar a interacção de um utilizador com a aplicação desejada. Uma forma de atingir esse objectivo é usar a área de Engenharia de Software chamada de *Performance Testing*, que permite criar cenários de testes de uma aplicação *web* e com esses cenários, que se encontram preenchidos com dados aleatórios, simular a interacção do utilizador com o *website*.

Assim, durante esta secção, será descrita a área de *Performance Testing* e algumas das suas principais subcategorias. Após a descrição da área, serão enunciadas as ferramentas existentes que dão a capacidade

ao programador de efectuar e executar cenários de utilização da aplicação. E por fim, será explicada a ferramenta usada para efectuar os testes sob as aplicações *web* desenvolvidas nas plataformas Java e *Rails*.

6.1.1 Performance Testing

A área de Engenharia de Software que se preocupa com a velocidade, escalabilidade e/ou estabilidade de uma aplicação em fase de testes é denominada de *Performance Testing*. Esta área tende a utilizar características como os tempos de resposta, a capacidade de transferência de recursos (*throughput*), os níveis de utilização do sistema onde se encontra a aplicação e outras para determinar se uma aplicação corresponde aos objectivos para os quais os testes de desempenho foram criados[21].

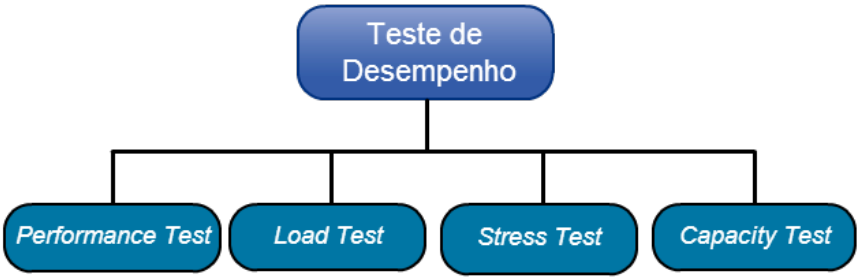


Figura 6.1: Categorias de Performance Testing para Aplicações Web

Dentro desta área, é possível enumerar quatro grandes categorias de tipos de testes a aplicar sob uma aplicação *web* que se encontram expostos na figura 6.1: *Performance Test*, *Load Test*, *Stress Test* e *Capacity Test*. Há que referir que, apesar de todas as categorias efectuarem testes de desempenho, cada uma delas assume um papel importante e diferente aquando do lançamento de uma aplicação *web*. Essas diferenças entre as categorias serão explicadas sucintamente abaixo como forma de contextualizar o uso desta área aquando da recolha e análise dos dados resultantes do *profiling* das aplicações e da ferramenta escolhida para a criação e execução de cenários de utilização. Para além disso, é relevante referir que esta área foi usada, pois seria necessário possuir uma forma de gerar informação para que, durante a execução das ferramentas de *profiling*, fosse possível obter dados mais fiáveis e controlados de utilização e também para atingir um certo número de utilizadores simultâneos na aplicação sem requerer interacção humana, o que poderia ser difícil de atingir.

Assim sendo, e tendo como base a figura 6.1, que é uma abstracção retirada de um guia publicado pela Microsoft acerca de testes de desempenho para aplicações *web* [21], na seguinte lista é descrito muito concisamente em que consiste cada uma das categorias já referidas anteriormente.

- **Performance Test** : Tem como propósito determinar ou validar a velocidade, escalabilidade e/ou a estabilidade de uma aplicação *web* durante a fase de testes. Foca-se sobretudo em determinar se o

utilizador ficará satisfeito com o desempenho da aplicação. Durante este teste, poderá não ser possível encontrar defeitos funcionais na aplicação que normalmente tendem a aparecer quando o sistema está sobre grande carga;

- **Load Test** : Serve para verificar o comportamento da aplicação quando ela se encontra em condições normais ou em grandes picos de carga. Com este tipo de teste, é possível averiguar os tempos de resposta, o *throughput* e os níveis de utilização do hardware onde se encontra a aplicação. Também se consegue identificar em que altura a aplicação começa a falhar, isto é, quantos utilizadores consegue simultaneamente responder sem apresentar nenhum erro. Para além disso, também se detecta problemas de concorrência e defeitos funcionais da aplicação;
- **Stress Test** : Serve para determinar o comportamento de uma aplicação quando ela é levada ao extremo, ou seja, sabendo que a aplicação aguenta com um determinado número de utilizadores e apresenta um comportamento aceitável e estável, ultrapassar esse limite e verificar como se comporta nesse estado. O principal objectivo deste teste é descobrir erros que normalmente só aparecem quando a aplicação se encontra com um grande afluência de dados e estes erros podem variar entre problemas de sincronização, *race conditions* e falhas de memória;
- **Capacity Test** : Serve para averiguar quantos utilizadores e/ou transacções a aplicação consegue suportar e mesmo assim atingir os objectivos de desempenho definidos antes da realização do teste. Também serve como forma de planear o crescimento da aplicação, quer em números de utilizadores quer em volume de dados a serem tratados.

6.1.2 Ferramentas de Performance Testing

Para poder aplicar a área de Engenharia de Software referida anteriormente, foi preciso procurar ferramentas que fornecessem a capacidade de criar uma bateria de testes com as várias possibilidades de utilização por parte do utilizador, visto que o objectivo final será sempre emular a acção executada por ele na aplicação. Para além da criação de cenários de testes, estas ferramentas também deveriam ter a capacidade de executar esses mesmos cenários e conceder ao programador métricas importantes, algumas delas já referidas anteriormente, resultantes da execução da bateria de testes.

Assim, na seguinte lista serão apresentadas as ferramentas encontradas que permitem efectuar testes de desempenho, independentemente da categoria a que pertençam esses testes. Não será feita nenhuma descrição individual de cada ferramenta nem o que oferecem ao programador, devido a serem ainda algumas e possuírem características muito únicas que na maioria delas nem sequer serão levadas em consideração aquando da escolha da ferramenta a usar. Para além disto, também será levado em consideração a licença de uso que possuem, isto é, se são ferramentas gratuitas ou se é preciso possuir uma licença paga para usufruir delas.

- *Apache JMeter*;
- *Curl-loader*;
- *Selenium*;
- *Watir*;
- *The Grinder*;
- *Pylot*;
- *Siege*;
- *OpenSTA*;
- *WCAT(Web Capacity Analysis Tool)*;
- *IBM Rational Functional Tester*;
- *LoadUIWeb*;
- *HP LoadRunner*.

6.1.3 Ferramenta Escolhida - Apache JMeter

Após a pesquisa e enumeração das ferramentas que permitem criar cenários de testes de execução de uma aplicação *web* anteriormente, foi preciso escolher a ferramenta que garantisse uma curva de aprendizagem rápida, fosse simples de manusear, conseguisse emular o comportamento de um *browser web* e enviar pedidos ao servidor.

Então, para tal decisão ser tomada, foi importante analisar algumas das funcionalidades oferecidas pelas ferramentas enunciadas anteriormente, a documentação e exemplos de uso existentes para cada uma delas, bem como efectuar a instalação das ferramentas como forma de garantir o seu perfeito funcionamento, visto que algumas delas já não sofrem actualizações com a regularidade necessária. Assim sendo, a ferramenta testada e escolhida entre as demais foi o *Apache JMeter*, ou simplesmente *JMeter*, que se encontra na versão *2.11*.

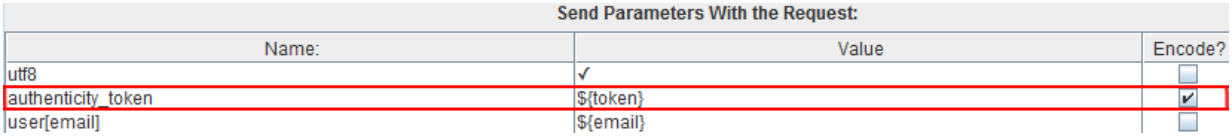
O *JMeter* é uma ferramenta *open-source* da *Apache Software Foundation* que permite efectuar testes de desempenho sobre uma variedade de serviços. É uma das principais e mais usadas ferramentas *open-source* para criar e efectuar cenários de teste de desempenho sob uma aplicação *web*, devido à capacidade de enviar pedidos *HTTP* ou *HTTPS*, analisar e construir a resposta do servidor, efectuar o descarregamento dos ficheiros necessários de uma página, tais como imagens, ficheiros *CSS*, *Javscript* e outros. Há que apontar que também possibilita o descarregamento paralelo de vários ficheiros para assim garantir uma emulação quase verdadeira de um *browser*. Para além disso, é possível recorrer as funções que analisam a resposta do servidor, normalmente em *HTML* ou *JSON/XML*, e efectuam o *parsing* sob essa resposta, usando para tal uma expressão regular, como forma de recolher alguma informação relevante para a submissão de formulários através do armazenamento da informação em variáveis.

Apesar das funcionalidades já mencionadas e de possuir uma documentação razoável no seu próprio *website*, a comunidade e os exemplos que demonstram o seu uso nas tarefas mais variadas é que são os motivos principais pela sua escolha e que atenuam consideravelmente a curva de aprendizagem inerente à utilização da ferramenta.

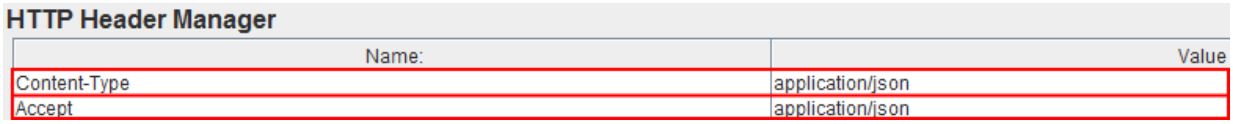
Mas como é habitual durante a aprendizagem de uma nova ferramenta, as dificuldades tendem a surgir sobretudo quando é preciso fazer algo diferente do que existe nos exemplos que se encontram dispersos pela Internet. Assim, em seguida, serão explicadas as duas das maiores dificuldades encontradas durante

a criação dos cenários de testes para a aplicação desenvolvida na plataforma RoR e que são raramente referidas nos exemplos encontrados na Internet. Não será mencionada nenhuma grande dificuldade para a plataforma Java, pois as que foram encontradas ao longo do desenvolvimento dos cenários foram superadas com relativa facilidade.

- Durante a submissão de formulários na aplicação, é necessário enviar no corpo da mensagem uma variável que representava o *token* de autenticidade. Este *token* é usado pelo *Rails* para garantir a segurança da aplicação e dos utilizadores que a usam contra ataques *Cross Site Request Forgery* (CSRF). E o problema não se prendia por recolher o *token* que se encontra na página *HTML* que contém o formulário mas sim no envio dele no pedido, pois o conteúdo do *token* é composto por letras, números e símbolos. O *JMeter* interpretava os símbolo *+* e *?* como fizessem parte de uma expressão regular e assim no seu lugar encontrava-se um espaço em branco, o que resultava num erro no servidor. Para resolver este problema, bastou apenas efectuar a verificação da opção de*encode* da variável que guardava esse *token*, como se pode observar na figura 6.2a, para que o servidor de *Rails* aceitasse todos os pedidos que exigiam esse *token* de autenticidade;
- Durante o envio de pedidos *HTTP* usando *AJAX*, o servidor interpretava os pedidos HTTP como sendo pedidos para construir uma página *web* normal e não como para responder apenas com a informação necessária codificada em *JSON*. Então, para resolver este problema, foi preciso adicionar ao cabeçalho do pedido *HTTP* os parâmetros *Content-Type* e *Accept* preenchidos com o conteúdo *application/json*, como se pode observar na figura 6.2b.



(a) Verificação da opção *encode* na variável referente ao *token* de autenticidade



(b) Adição de parâmetros ao cabeçalho de um pedido *HTTP*

Figura 6.2: Resolução das dificuldades encontradas no Apache JMeter

6.1.4 Testes de Performance

Para poder obter dados durante a execução de uma aplicação *web* e conhecer o seu comportamento antes desta estar disponível ao utilizador, é preciso criar e emular a sua utilização por parte de um cliente. E para conseguir conceber essa utilização foi utilizada a ferramenta *open-source Apache JMeter*.

Assim, foi necessário idealizar um plano de teste que cobrisse todas as páginas visitáveis pelo utilizador e inclui-se, nesse plano de teste, o tempo que o utilizador demora a ler e pensar no que fazer a seguir, normalmente designado como *user think time*. Usualmente, para conhecer uma aproximação ao valor a atribuir ao *user think time*, são utilizadas ferramentas que analisam uma aplicação *web* com foco na experiência de utilização do *website* e nas opções que os utilizadores tomam quando o visitam, conhecidas por *web analytics tools*. Entre as ferramentas que se podiam usar para conhecer o comportamento de um utilizador num *website*, encontram-se o *Google Analytics*, o *Yahoo Web Analytics*, o *Spring Metrics* e o *Woopra*. Mas como não seria possível garantir que as aplicações desenvolvidas iriam possuir uma grande afluência e variedade cultural de utilizadores, então o uso de qualquer das ferramentas mencionadas foi descartado. Assim, foi preciso fazer uma estimativa desse tempo com base no que as empresas que efectuam *Load Testing* recomendam para tal efeito e do que seria espectável ao navegar nas aplicações *web* desenvolvidas. Tendo isto em mente, o *think time* atribuído durante a criação do plano de teste foi um valor gerado aleatoriamente entre 3 e 7 segundos de espera entre cada pedido do utilizador ao servidor, visto que não existia um consenso entre as empresas estudadas acerca da aproximação a usar, por se tratar de uma métrica que não é facilmente mensurável e que é alvo de alguma subjectividade, normalmente associada às pessoas que usufruem das aplicações *web* [22].

Agora, fica apenas a faltar descrever a forma como se comporta o plano de teste e a duração utilizada durante a sua execução, bem como o motivo pelo qual foi escolhida essa duração. O plano de teste foi desenhado usando para tal, como já referido, a ferramenta *JMeter*. Nesse plano de teste, que é igual quer para a aplicação desenvolvida em Java quer em *Rails*, foram criadas as várias acções/páginas que o utilizador poderá chamar/visitar ao navegar no *website* e que serão apresentadas posteriormente. Os dados a serem introduzidos para as pesquisas e submissão de formulários são gerados aleatoriamente através do uso dos métodos *Random Variable* e *BeanShell PreProcessor*, oferecidos pelo *JMeter*. No caso específico das pesquisas, os dados são escolhidos através da inspecção da página *HTML* resultante de um acção que antecede a pesquisa e da utilização sobre essa página de funções que permitem executar expressões regulares para retirar a informação com um padrão pré-definido, através do *Regular Expression Extractor* oferecido pelo *JMeter*.

Para além disso, é importante referir que cada utilizador apenas consegue efectuar um caminho de acções de cada vez que acede à aplicação *web*, sendo que esse caminho percorrido é escolhido aleatoriamente pelo *JMeter* através da utilização do *Random Controller* que garante a unicidade do caminho realizado. E para poder garantir que, de facto, ao longo de 3 horas encontra-se sempre um número constante de utilizadores a navegar nas aplicações *web* foi utilizado um *plugin* oferecido pela comunidade oficial do *JMeter* chamado *Stepping Thread Group*, que permite indicar o número máximo de utilizadores e por quanto tempo, eles devem estar a efectuar pedidos ao servidor.

Por último, é de mencionar que o plano de teste foi executado durante 3 horas para cada uma das aplicações, como forma de investigar se as respectivas plataformas *web* e servidores usados sofriam alguma variação inesperada ao longo desse período de tempo com um número constante de utilizadores a fazerem pedidos e também para garantir um grande número de execuções de cada acção para posteriormente possuir um melhor conjunto de resultados para análise.

6.2 Selecção das Ferramentas de Profiling

Após a explicação de como o plano de teste se comporta genericamente e que servirá sobretudo para gerar tráfego constante nas aplicações *web* e da escolha das facetas de *profiling*, na secção 4.2, que serão alvo de análise posteriormente, é imperativo saber quais das facetas escolhidas cobrem um maior número de ferramentas de *profiling* em cada uma das plataformas de desenvolvimento *web*. Também serão apresentados os motivos para não utilização de algumas das ferramentas que poderiam cobrir o espectro das facetas escolhidas e assim ajudar na obtenção de dados das aplicações, mas que por alguma limitação tecnológica ou falta de documentação por parte da empresa que desenvolve a ferramenta para o seu correcto funcionamento não tornaram possível a sua utilização na obtenção de dados. E antes demais, é importante deixar como nota que as ferramentas estudadas estão em constante desenvolvimento e que as versões usadas durante a realização do estudo estão limitadas ao que conseguem efectuar dentro dessa versão e que numa futura análise, elas poderão ser capazes de analisar ainda mais facetas, resolverem qualquer limitação que possa ter ocorrido e apresentarem dados mais refinados e com uma divisão mais aproximada da arquitectura MVC.

Profiler / Faceta	VisualVM	NetBeans Profiler	YourKit	JProfiler	New Relic
CPU Time	●	●	●	●	
Wall Clock Time	●	●	●	●	●
Average Response Time			●	●	●
Action Average Response Time				●	●
Action Average Throughput					●
Average Page Creation Time					●
Average Page Loading Time					●
Page Average Throughput					●
Average Time in DB Operations					●

Continua na próxima página

Average Query Response Time			•	•	•
Query Average Throughput					•

Tabela 6.1: Selecção da Ferramenta de Profiling a usar na plataforma Java

Assim, na tabela 6.1 acima apresentada, é possível identificar as facetas escolhidas e as ferramentas que permitem efectuar *profiling* sobre aplicações na plataforma Java. Através de uma simples observação da tabela, pode-se perceber que as ferramentas que conseguem cobrir mais o espectro das facetas pretendidas são o *YourKit*, o *JProfiler* e o *NewRelic*. As outras duas ferramentas, *VisualVM* e *NetBeans Profiler*, não foram sequer consideradas, visto que elas apenas se centram em analisar o tempo gasto no CPU pelos métodos executados no servidor. Das três ferramentas que cobrem o espectro, apenas foi possível efectuar a instalação e a utilização de duas delas com sucesso. O *YourKit* foi a ferramenta que apresentou problemas na altura de se conectar quer com o servidor *Tomcat* quer com a JVM, tornando assim impossível a sua utilização para recolher dados acerca à execução da aplicação *web*. Em relação ao *JProfiler*, como é possível observar na tabela, não oferece a capacidade para analisar as facetas que se encontram dentro da categoria de *View Profiling* mas em compensação, permite saber em detalhe quantas *queries* foram executadas, como estavam definidas e o tempo médio de execução de cada um, bem como o tempo demorado em cada acção da aplicação. A única contrapartida encontrada com a utilização desta ferramenta é o *overhead* exercido sobre o servidor, o que poderá adulterar os valores a serem expostos mais à frente. Em relação à ferramenta *NewRelic*, as facetas que oferece em termos de análise do funcionamento de uma aplicação *web* enquadram-se na arquitectura MVC e o *overhead* exercido sobre o servidor é praticamente nulo, tornando-a assim na ferramenta de eleição. Para além disso, a sua instalação é praticamente linear, bastando para isso seguir o guião fornecido pela empresa.

Profiler / Faceta	MRI/REE	JRuby	Rubinius	MiniProfiler	New Relic
CPU Time					
Wall Clock Time	•	•	•		•
Average Response Time					•
Action Average Response Time					•
Action Average Throughput					•
Average Page Creation Time					•
Average Page Loading Time					•

Continua na próxima página

Page Average Throughput					•
Average Time in DB Operations					•
Average Query Response Time					•
Query Average Throughput					•

Tabela 6.2: Selecção da Ferramenta de Profiling a usar na plataforma RoR

Já, na tabela 6.2 acima exposta, consegue-se identificar o mesmo conteúdo que na tabela anterior mas com as ferramentas que são capazes de realizar *profiling* sobre aplicações na plataforma RoR ao invés da plataforma Java. Ao contrário do que aconteceu para a escolha da ferramenta em Java, as hipóteses em Rails ficaram reduzidas apenas a uma ferramenta capaz de cobrir o espectro já falado, o *NewRelic*. Antes de expor algumas das características únicas da ferramenta escolhida, há que denotar que as ferramentas *MRI/REE*, *JRuby* e *Rubinius* se tratam de diferentes interpretadores da linguagem Ruby, como referido na secção 2.3, e que cada interpretador possui um *profiler* incorporado nele mas que apenas conseguem analisar características relacionadas com a memória e o CPU. Enquanto que o *MiniProfiler* não cobre nenhuma faceta daquelas que foram escolhidas e não possibilitar a recolha de qualquer informação aquando da passagem da aplicação para produção, é de mencionar que é uma excelente ferramenta para ser usada durante o desenvolvimento de uma aplicação, pois fornece ao programador um grande conjunto de dados relevantes sobre a execução da aplicação e que permitem descobrir qual parte da aplicação está a causar a lentidão da criação de uma página específica. No entanto, a ferramenta *NewRelic* apresenta-se como sendo uma ferramenta de monitorização de aplicações *web* e que melhor consegue distinguir as diferentes camadas da arquitectura MVC de uma aplicação, tornando-a assim como em Java a ferramenta a utilizar para averiguar os dados de execução da aplicação. Além disso, a sua configuração na plataforma *Rails* é feita através da adição de uma *gem* e um ficheiro fornecido pela empresa à aplicação, tornando assim a sua utilização simples e intuitiva.

6.3 Mapa das Acções

Uma comparação entre duas aplicações *web* que servem o mesmo propósito e possuem a mesma organização em termos das páginas *web* que o utilizador consegue visualizar só pode acontecer se existir um mapa das acções que a aplicação é capaz de realizar. Este mapa é necessário, pois não seria justo nem possível comparar duas plataformas de desenvolvimento *web* com base apenas em facetas que apenas pretendem averiguar globalmente o comportamento e o funcionamento quer da plataforma em si quer do servidor usado para a execução das aplicações.

Mas antes de ser explicado a construção e a nomeação das acções presentes no mapa, é preciso identificar e definir o que é entendido por uma acção numa aplicação *web*. Uma acção é, basicamente, um pedido *HTTP* ao servidor com o intuito de receber uma resposta, que pode variar entre uma simples página

HTML com ou sem dados específicos, submissão de formulários de pesquisa, de registo ou de início de de sessão e da recepção de dados em formato *JSON* ou *XML* através do uso de *AJAX* para efectuar esses pedidos. Também convém referir que, num contexto de modulação de um sistema informático, cada acção da aplicação *web* pode ser tratada como sendo um *Use Case*.

Acções	Java	Ruby on Rails
Página Inicial	/index.jsp	HomeController#index
Página Login	/users/login-page	Devise::SessionsController#new
Login	/users/login	Devise::SessionsController#create
Logout	/users/logout	Devise::SessionsController#destroy
Página do Registo	/users/register-page	RegistrationsController#new
Registo	/users/register	RegistrationsController#create
Perfil do Utilizador	/users/profile-page	RegistrationsController#edit
Update do Perfil	/users/update-profile	RegistrationsController#update
Página das Reservas	/users/reservations	HomeController#reservations
Página dos Táxis	/taxis/index-taxi	TaxisController#index
Pesq. de Cidades	/taxis/country	TaxisController#city
Cidades de Partida	/taxis/city	TaxisController#pick_up
Cidades de Chegada	/taxis/pick-up-locations	TaxisController#drop_off
Pesq. de Táxis	/taxis/search-taxis	TaxisController#search_results
Selecção do Táxi	/taxis/select-taxi	TaxisController#taxi_selected
Reserva do Táxi	/taxis/reserve-taxis	TaxisController#taxi_reservation
Página dos Carros	/cars/index-car	CarController#index
Cidades de Partida	/cars/pick-city	CarController#pick_city
Subsidiárias da Partida	/cars/pick-subsidiary	CarController#pick_subsidiary
Pesq. de Agências	/cars/get-agency	CarController#get_agency
Cidades de Chegada	/cars/drop-city	CarController#drop_city
Subsidiárias da Chegada	/cars/drop-subsidiary	CarController#drop_subsidiary
Pesq. de Carros	/cars/search-results	CarController#search_results
Escolha dos Extras	/cars/service-extras	CarController#service_extras
Pagamento do Carro	/cars/payment	CarController#service_payment

Continua na próxima página

Reserva do Carro	/cars/car-reservation	CarController#service_reservation

Tabela 6.3: Mapa das Acções da Aplicação em Java e RoR

Assim sendo, na tabela 6.3, é apresentado o mapa da acções presentes em ambas as aplicações, na qual a coluna *Acções* pretende ser uma designação intuitiva e única para o pedido *HTTP* realizado quer na aplicação em Java quer em *Rails*. Essa designação será usada mais à frente nas tabelas com os resultados obtidos durante a execução do plano de teste. Enquanto que o conteúdo nas demais colunas não representa o pedido *HTTP* que normalmente aparece no *browser* mas sim uma abstracção utilizada pela ferramenta de *profiling* escolhida para ambas as plataformas de desenvolvimento *web* na secção anterior como forma de identificar unicamente todos os pedidos efectuados ao servidor, já que a exposição de cada pedido *HTTP* como ele é realmente feito, poderia dificultar a leitura e a compreensão da tabela. E para facilitar uma melhor leitura e compreensão de alguns dos resultados a serem expostos posteriormente, as acções *Pesq. de Cidades*, *Cidades de Partida*, *Cidades de Chegada*, *Subsidiárias da Partida*, *Pesq. de Agências* e *Subsidiárias da Chegada* são efectuadas através de *AJAX*. Isto implica que o tempo que estas demoram a criar a página *HTML* resultante do pedido é nulo, pois a resposta a esse pedido será enviada em formato *JSON*, removendo assim a necessidade de compilar e transformar o resultado em *HTML*.

6.4 Web Profiling

Após a exposição de como serão realizados os testes de desempenho sobre as aplicações *web* e de alguns condicionantes impostos, da ferramenta de *profiling* que será utilizada para efectuar a recolha dos resultados e do mapeamento das acções existentes nas aplicações que servirão para realizar uma comparação minuciosa e correcta das duas plataformas *web*, é preciso demonstrar a forma como foram sintetizados e encaixados os resultados apresentados pelo *New Relic* nas categorias de *Profiling* existentes e também referir o motivo pelo qual o número de facetas analisadas diminui em relação à tabela apresentada na secção 4.2, bem como apresentar as facetas que realmente foram possíveis analisar com a ferramenta escolhida. Para além disso, serão expostos finalmente os resultados dos testes realizados, já divididos nas suas subcategorias, juntamente com uma análise aos diversos valores apresentados ao longo da variação do número de utilizadores em simultâneo nas duas aplicações.

Então, antes de se olhar para as tabelas com os resultados finais dos testes de desempenho, é preciso compreender o processo de recolha e divisão dos valores apresentados pela ferramenta de *profiling*. Nas aplicações desenvolvidas, a designação usada para os diversos componentes que compõem uma aplicação *web* variam e como seria de esperar, para poder efectuar uma separação correcta pelas diversas subcategorias de *Web Profiling*, é necessário apresentar alguns exemplos de como a ferramenta expõem os dados e que serviram para ajudar a compreender o processo já referido.

Assim, no apêndice C, é possível visualizar dois exemplos de como os dados dos diversos componentes

que constituem uma execução de uma acção na aplicação desenvolvida em RoR são expostos pelo *New Relic*. Dessas figuras, é importante reter que os componentes que pertençam à "*Database*" ou ao "*ActiveRecord*" irão constituir os dados relativos à categoria *Database Profiling*, enquanto que os componentes que se encaixem no "*Controller*" ou no "*Beans*" ou no "*BusinessLogic*" irão pertencer à categoria *Controller Profiling*. E como seria espectável, todos os componentes que se encaixem na "*View*" irão pertencer à categoria *View Profiling*. E por último, todos os componentes com a designação de "*Middleware*" não serão considerados, pois eles não pertencem a uma camada especifica de uma aplicação em *Rails* apesar da sua utilidade na execução correcta de uma acção.

Já, no apêndice D, é apresentado a forma como os componentes relativos à plataforma Java são analisados e expostos pela ferramenta. Ao contrário do que aconteceu com os componentes de *Rails*, em Java é necessário investigar alguns dos métodos executados durante uma acção para averiguar a qual das categorias de *profiling* pertence, possibilitando assim uma divisão mais coerente. Assim, qualquer componente que contenha a designação de "*JSP*" será encaixado na categoria *View Profiling*, enquanto que os componentes que possuem na designação ou no método executado alguns dos seguintes nomes: "*ORM*", "*Database*", "*C3P0*" e "*Hibernate*" irão pertencer à categoria *Database Profiling*. Faltando apenas a categoria *Controller Profiling*, os componentes com as seguintes designações ou nomes de métodos executados irão fazer parte dela: "*StrutsAction*", "*Servlet*", "*StrutsResult*", "*Filter*", "*WebTransaction*" e "*ServletRequestListener*". E como aconteceu também em RoR, os restantes componentes que não encaixarem nas condições apresentadas serão descartados.

Antes de serem expostos os resultados dos testes, que permitiram fazer uma análise ao desempenho das aplicações *web* nas diferentes plataformas do estudo, é preciso referir que, apesar de a ferramenta escolhida conseguir analisar um maior número de facetas do que as demais ferramentas estudadas, nem todas as facetas serão utilizadas na análise. Isto acontece, pois existem facetas como *Page Average Throughput* e *Average Page Loading Time* que se referem ao tempo passado pelos diferentes *browsers* a construírem o *HTML* recebido do servidor ou ao número de páginas servidas por minuto pelo *browser*, não sendo o foco deste estudo não serão consideradas. Outro motivo pelo qual estas facetas não serão consideradas é a impossibilidade de gerar dados através do *JMeter*, visto que para o *New Relic* recolher informação acerca delas precisa de executar um *script* de *javascript* e como o *JMeter* não oferece essa possibilidade, elas serão postas de parte. Apesar das facetas *Average Query Response Time* e *Query Average Throughput* serem passíveis de análise, também não serão utilizadas, visto que apenas se focam nas *querys* efectuadas à base de dados, levando assim a uma granularidade indesejada. As duas facetas relativas ao *CPU Profiling* foram descartadas, porque o *New Relic* apenas analisa o *Wall Clock Time* e esta é apenas analisada no contexto do servidor sem que se saiba nada em relação às acções de uma aplicação *web*.

Facetas	Unidade Métrica
Average Response Time	Milissegundos (ms)
Action Average Response Time	Milissegundos (ms)
Average Page Creation Time	Milissegundos (ms)
Average Time in DB Operations	Milissegundos (ms)

Tabela 6.4: Facetas de Profiling Utilizadas

Portanto, após a apresentação dos motivos pelo quais a tabela de facetas escolhidas sofreu uma diminuição, existe a necessidade de apresentar a tabela final das facetas a serem utilizadas na comparação. Assim, na tabela 6.4, são expostas as facetas que serão o alvo de recolha e categorização dos resultados dos testes de desempenho, bem como de uma posterior análise das mesmas a serem apresentados nas subsecções seguintes. E antes de passarmos às análises individuais das categorias de *profiling*, é imperioso referir que os resultados apresentados ao longo das subsecções seguintes poderão variar, já que como foi referido anteriormente os testes de desempenho são aleatórios e a probabilidade de duas execuções durante três horas do mesmo teste serem iguais é muito pequena, levando assim à possibilidade de pequenas variação nos resultados aquando de uma nova execução.

6.4.1 View Profiling

A criação da página *HTML* a ser apresentada ao utilizador e o tempo que o servidor demora a criá-la é um factor importante num aplicação *web* que usa como padrão a arquitectura MVC. Este tempo passado a compilar as várias partes que constituem uma página *web* será o principal foco desta secção. E antes de ser feita uma análise mais global aos resultados obtidos para 50, 100, 150 e 200 utilizadores nesta categoria, é preciso mencionar algumas discrepâncias nos valores apresentados em algumas das acções. As principais discrepâncias encontram-se sobretudo na aplicação desenvolvida em RoR que trata algumas das acções como não tendo nenhuma página *web* resultante do pedido e que reencaminham o esperado resultado da acção inicial para a uma outra acção que irá concluir o processo normal, originando assim valores iguais a 0. Então, entre estas acções encontram-se o "*Login*", o "*Logout*", o "*Registo*" e o "*Update do Perfil*" e que não serão consideradas quando for efectuado a análise mais global ao desempenho das aplicações no que toca a esta categoria de *profiling*. As restantes acções que têm valores iguais a 0 já foram explicadas o motivo desse valor na secção 6.3.

Assim, através de uma observação cuidada das tabelas E.1, E.2, E.3 e E.4 presentes no apêndice E.1, verifica-se que a plataforma Java obtém um melhor desempenho global em termos do tempo médio passado a compilar as diversas páginas associadas às acções ao longo dos diferentes número de utilizadores em simultâneo na aplicação. Além disso, o desempenho da aplicação desenvolvida em RoR começa a deteriorar quando o número de utilizadores é de 200, enquanto que a aplicação Java, apesar de os valo-

res aumentarem, mantém um tempo médio de resposta linear sem que alguma acção executada consiga afectar o servidor de forma a ele sofrer algum atraso noutra acção. Para garantir que esta análise não seja só feita de uma forma global, é necessário escolher algumas acções que possuam algum significado e que representem os pontos críticos da aplicação. Sendo assim, das acções presentes na aplicação, foram escolhidas as seguintes para uma análise mais objectiva : "*Página de Reservas*", "*Pesq. de Táxis*" e "*Pesq. de Carros*". Esta capacidade de escolher e analisar individualmente as acções de uma aplicação *web* é um facto a ter em conta aquando da comparação de plataformas *web* diferentes. Para além disso, também permite descobrir quais são as partes mais problemáticas de uma aplicação.

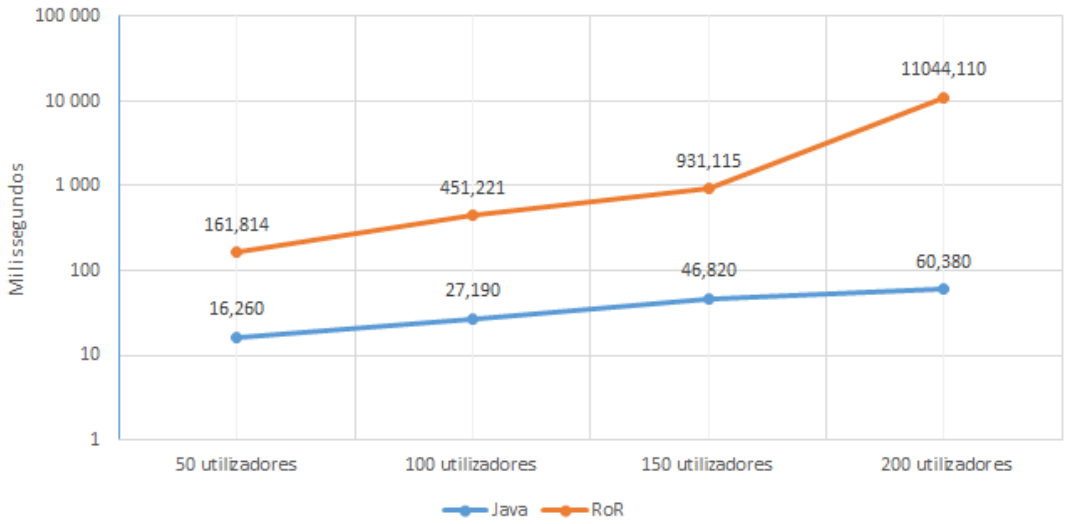


Figura 6.3: Average Page Creation Time - Páginas de Reservas

Então, no caso da acção "*Página de Reservas*", ela contém todas as reservas feitas por um utilizador, quer sejam de carros ou de táxis. Esta listagem foi limitada a apenas a 200 reservas de carros e táxis, já que seria inconvencional listar mais de mil reservas de um utilizador numa só tabela. Ao verificar os valores nas duas plataformas, é perceptível que Java consegue criar essa página em menos tempo do que *Rails* ao longo do aumento gradual de utilizadores a efectuarem pedidos ao servidor, visível na figura 6.3. E para demonstrar isso de uma forma simples, basta observar na figura acima os pontos referente a 50 utilizadores e verificar que a diferença de valores entre as duas plataformas é de praticamente 146 milissegundos. A diferença é ainda mais gritante nos pontos que dizem respeito a 200 utilizadores. O valor da diferença é perto dos 10984 milissegundos, o que significa que a aplicação em RoR possui um problema na construção de páginas *web* que tenham uma listagem de um grande número de dados.

Em relação à acção "*Pesq. de Táxis*", na figura 6.4 é visível o seu comportamento ao longo dos diversos utilizadores utilizados nas duas plataformas. Inicialmente, a plataforma RoR consegue obter um melhor tempo médio de criação da página *web* de resposta em relação à plataforma Java mas que aumenta dras-

ticamente quando a carga é de 200 utilizadores. Em contrapartida, a plataforma Java consegue obter um tempo médio de criação mais linear e sem aumentos drásticos como aconteceu na aplicação em *Rails*.

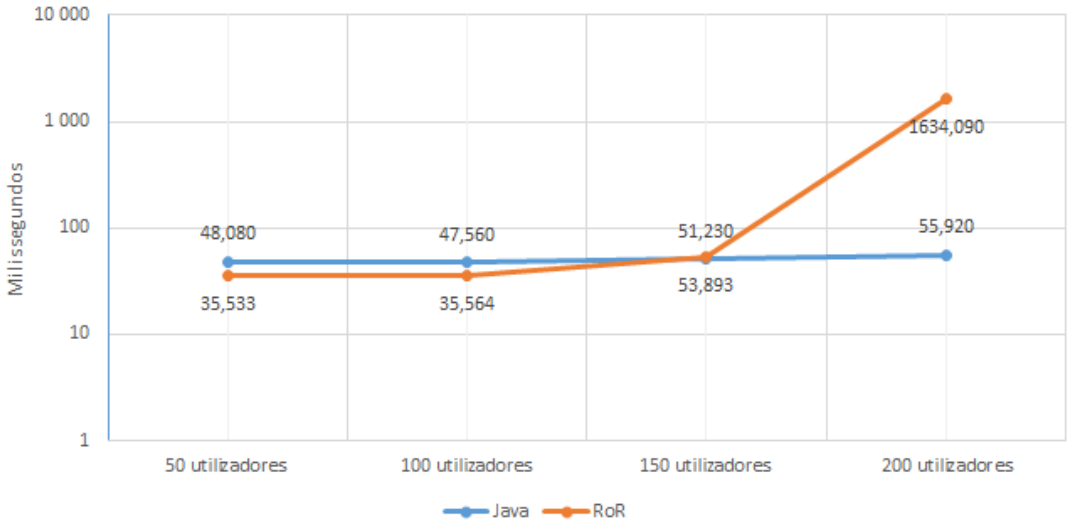


Figura 6.4: Average Page Creation Time - Pesq. de Táxis

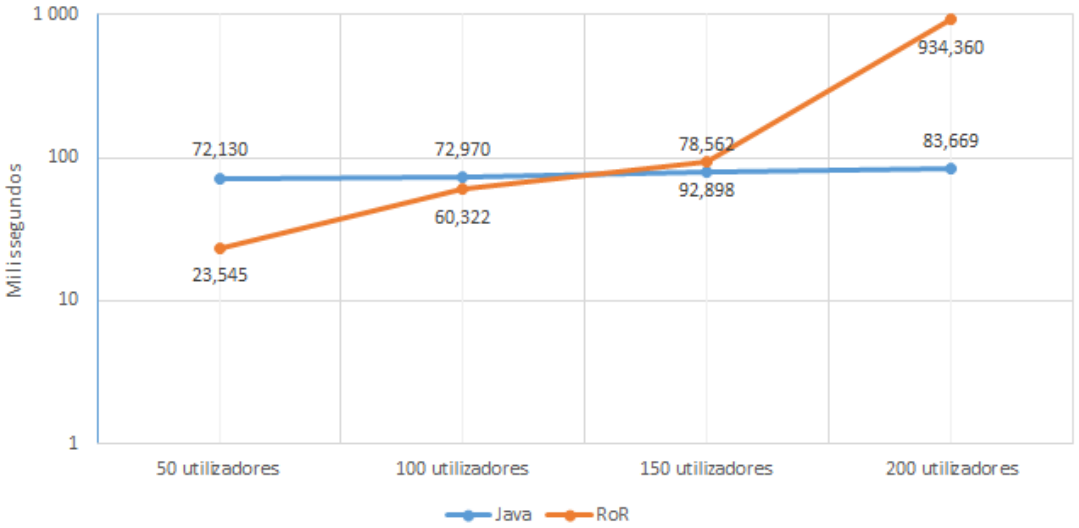


Figura 6.5: Average Page Creation Time - Pesq. de Carros

Por fim, em relação à acção "*Pesq. de Carros*, na figura 6.5 percebe-se que o tempo médio de criação da página *web* para essa acção na plataforma RoR aumenta gradualmente sem nunca estabilizar em oposição com a aplicação em Java que consegue manter um valor estável sem grande oscilações. Inicialmente, para 50 utilizadores, a aplicação Rails consegue obter um melhor desempenho, sendo que a diferença entre as duas plataformas é de aproximadamente 48 milissegundos. Porém, convém denotar que a diferença

entre os pontos do gráfico relativos a 200 utilizadores é de aproximadamente 851 milissegundos, relevando assim, como anteriormente, um problema grave na plataforma RoR em relação à camada que trata da *View* de uma aplicação *web*.

6.4.2 Controller Profiling

Para que uma aplicação *web* que seja desenvolvida usando a arquitectura MVC como base, é importante que a camada que lida com os todos pedidos realizados pelo utilizador ao servidor consiga processar tais pedidos com um bom tempo de resposta e possua um bom fluxo de comunicação entre as demais camadas da arquitectura. Assim, no decorrer desta secção, será analisado a camada referente ao *Controller* das aplicações desenvolvidas usando para tal duas facetas diferentes. Primeiramente, será examinada a faceta *Average Response Time* que pretende apenas se refere ao tempo passado na camada em cada uma das plataformas. Depois, no fim da secção 6.4, será dedicada uma subsecção que irá examinar a faceta *Action Average Response Time* que, ao contrário da anterior, expressa o tempo despendido numa determinada acção de uma forma mais global, ou seja, nessa faceta é contabilizado todo o tempo gasto pela aplicação na construção da resposta ao utilizador que pode envolver a passagem pelas três camadas da arquitectura MVC. Esta análise final com base na faceta anterior irá rematar o estudo das diversas camadas estudadas e na qual será tecida alguns comentários aos resultados obtidos a durante o *Web Profiling* sobre duas plataformas *web* diferentes.

Então com isto tudo em mente, nas tabelas E.5, E.6, E.7 e E.8 presentes no apêndice E.2, são expostos os resultados obtidos durante a execução do plano de testes já referido anteriormente para 50, 100, 150 e 200 utilizadores. Ao observar as tabelas referidas na faceta *Average Response Time* de uma forma objectiva, consegue-se perceber com clareza que a plataforma Java despende menos tempo a tratar do pedido recebido e efectuar qualquer algoritmo ou tratamento de dados necessário para enviar uma resposta ao utilizador. Porém, esta observação não deve ser considerada como final, visto que nem todos os pedidos possuem algoritmos ou construções da resposta mais elaboradas e levando assim à necessidade de examinar em detalhe algumas das acções das aplicações onde exista uma complexidade extra na construção da resposta. Deste modo, foram seleccionadas as seguintes acções para examinadas em maior detalhe através de gráficos com os seus resultados para 50, 100, 150 e 200 utilizadores : "*Registo*", "*Update do Perfil*", "*Pesq. de Táxis*", "*Pesq. de Carros*", "*Reserva de Táxis*" e "*Reserva do Carro*". Estas acções foram seleccionadas por se tratarem genericamente de *Use Cases* que envolvem um certo nível de complexidade ou que são usualmente utilizados em grande parte das aplicações *web* existentes actualmente e que, futuramente, esta facilidade de análise individual de acções/ *use cases* poderá configurar uma metodologia a utilizada para investigação de zonas críticas de uma aplicação, bem como servir de base para a comparação de plataformas de desenvolvimento *web*.

Assim, na figura 6.6, é apresentado o gráfico de desempenho da acção "*Registo*" que, como o próprio

nome indica, lida com a adição de novos utilizadores na aplicação. A partir do gráfico, é verificável que a aplicação em *Rails*, inicialmente, consegue responder em menos tempo do que a de Java, todavia com o aumento de utilizadores essa capacidade de resposta tende a deteriorar. Por outro lado, a aplicação em Java faz o percurso inverso, ou seja, no início possui uma fraca capacidade de resposta mas conforme o número de utilizadores cresce, essa capacidade melhora e mantém-se dentro do mesmo conjunto de valores sem nenhum oscilação.

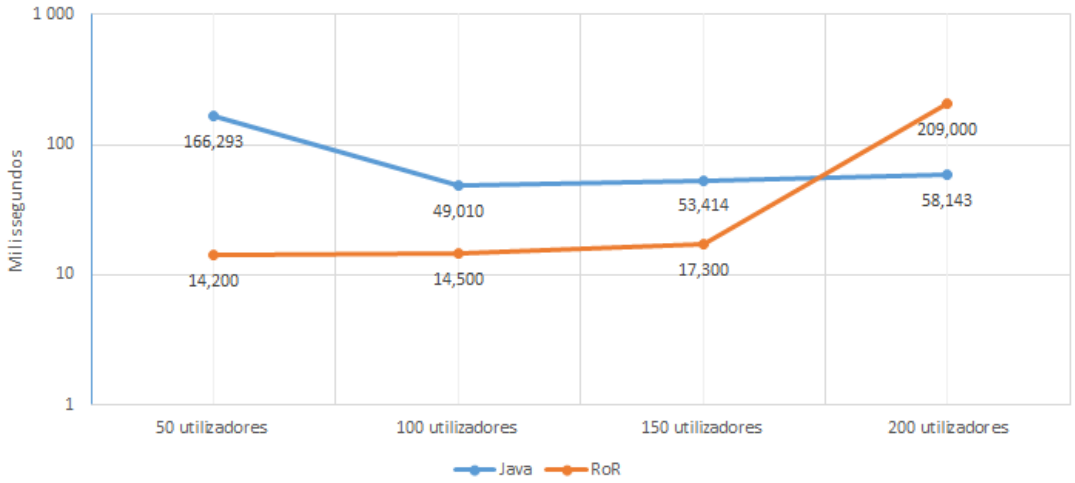


Figura 6.6: Average Response Time - Registo

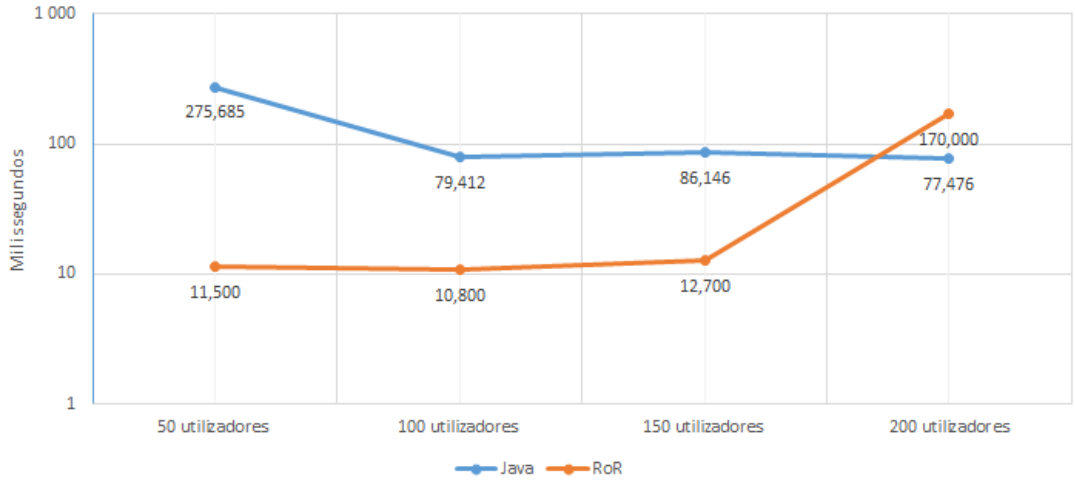


Figura 6.7: Average Response Time - Update do Perfil

Na figura 6.7, é exibido o gráfico de desempenho da acção "*Update do Perfil*". Nesta acção, o utilizador realiza apenas a alteração dos seus dados pessoais sem que seja efectuado nenhum algoritmo complexo. Como aconteceu na acção anterior, a plataforma *Rails* exibe inicialmente um melhor tempo de resposta do

que a plataforma Java, porém com o crescimento gradual de utilizadores esse tempo de resposta piora. Enquanto que a plataforma Java efectua o percurso inverso, levando a acreditar que conforme o número de utilizadores simultâneos cresce, o tempo necessário para responder a essa demanda estabiliza.

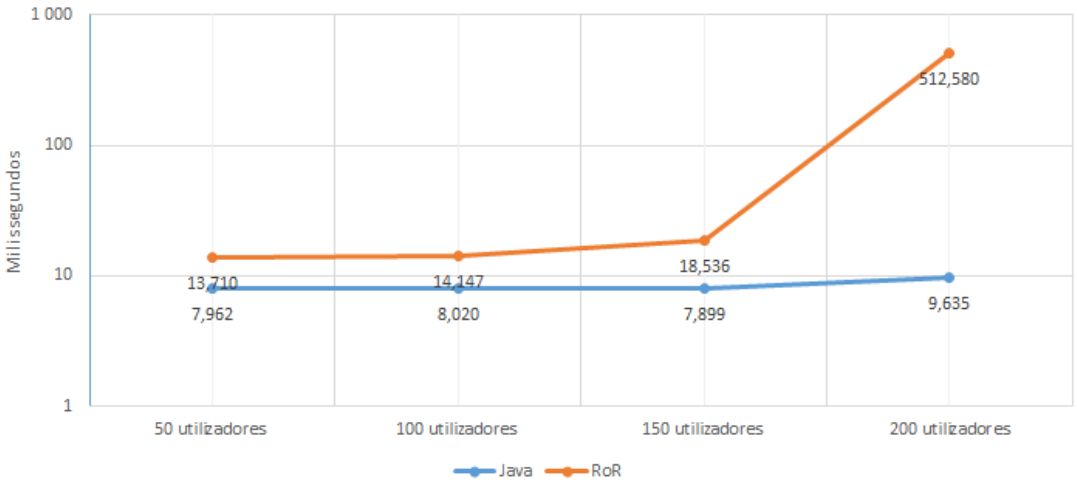


Figura 6.8: Average Response Time - Pesq. de Táxis

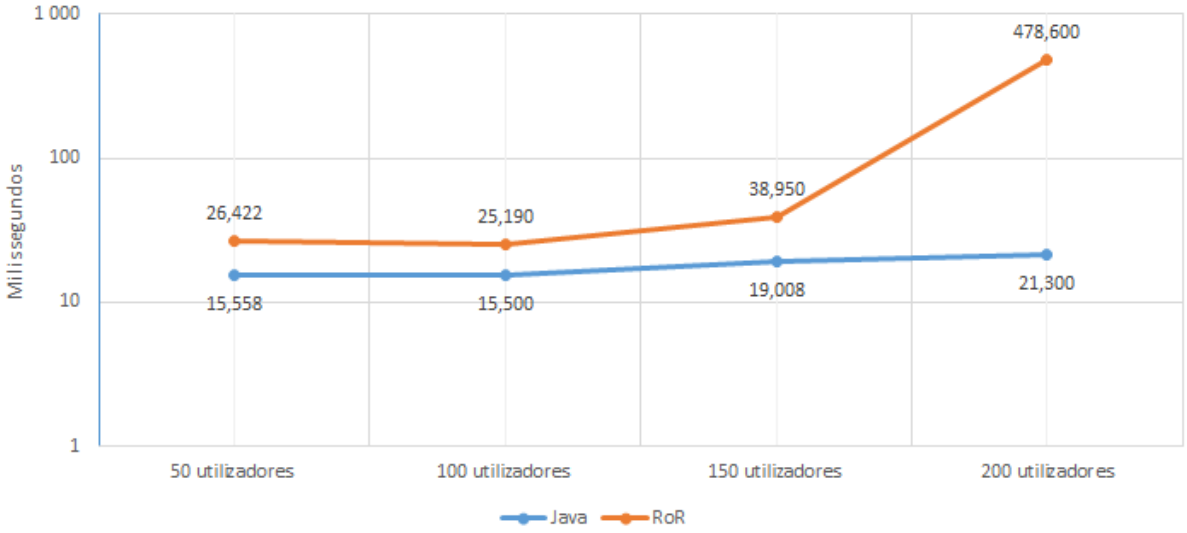


Figura 6.9: Average Response Time - Pesq. de Carros

Nas figuras 6.8 e 6.9, são expostos os gráficos de desempenho das acções "*Pesq. de Táxis*" e "*Pesq. de Carros*, respectivamente. Estas acções em comparação com as que já foram apresentadas, possuem alguma complexidade na construção da resposta que será mostrada ao utilizador, ou seja, o processo de pesquisa envolve mais do que apenas uma simples *query* à base de dados. Ao contrário do que se pode assumir, é necessário restringir os táxis ou carros que serão apresentados para escolha, visto que ao efectuar

a reserva de um deles ele fica indisponível no mesmo período de tempo que foi reservado. Para além disso, é necessário associar o custo que cada carro ou táxi tem, com base no percurso escolhido pelo utilizador. Assim, através da observação dos dois gráficos, consegue-se concluir que a plataforma Java executa os algoritmos necessários e toda a panóplia de métodos associados ao *Controller* em menos tempo do que o RoR. É de ressaltar a diferença de tempos existente para 200 utilizadores entre as duas plataformas, que é de 503 milissegundos para a pesquisa de táxis e de 457 milissegundos para a pesquisa de carros.

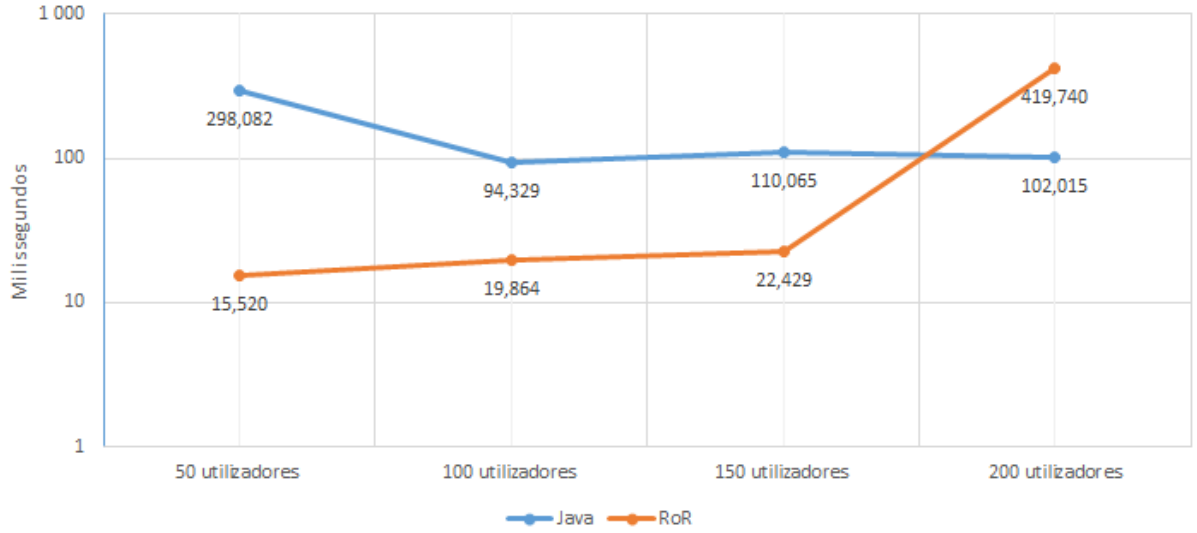


Figura 6.10: Average Response Time - Reserva de Táxis

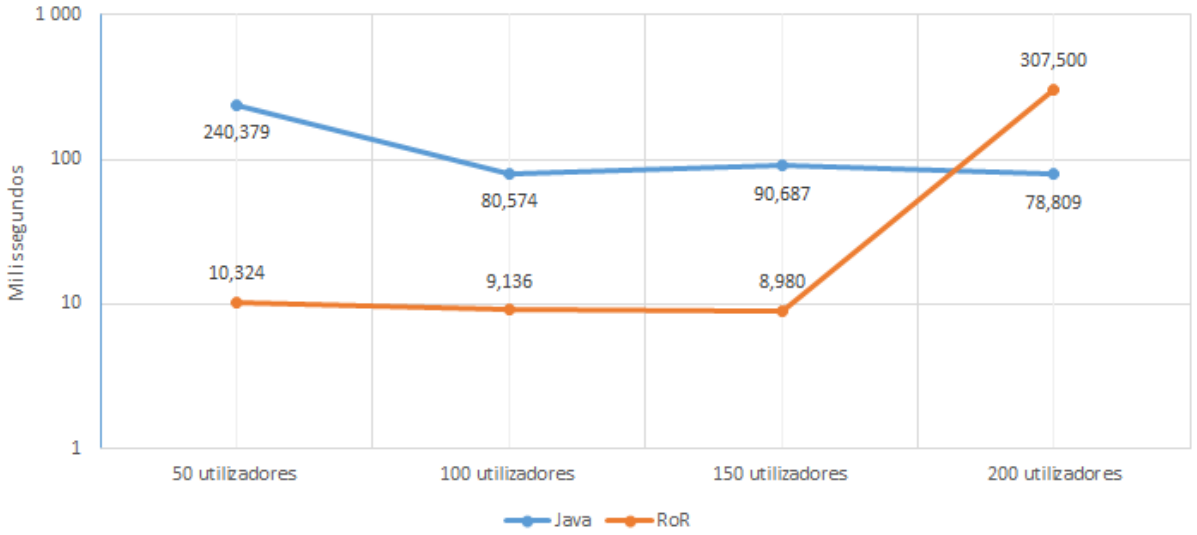


Figura 6.11: Average Response Time - Reserva do Carro

Por último, nas figuras 6.10 e 6.11, são exibidos os desempenhos das acções "*Reserva de Táxis*" e

"*Reserva do Carro*" nos diversos números de utilizadores testados, respectivamente. No que toca ao que cada uma das acções executa no servidor, pode-se referir que em ambas são verificadas as credenciais do cartão de crédito inserido e são adicionados novos registos às tabelas na base de dados. Apesar de não ter a mesma complexidade que as acções acima expostas, estas precisam de transformar os dados para verificação e conseguente inserção, caso passem nessa verificação. Mas passando à análise dos gráficos expostos, é possível verificar que a plataforma RoR consegue efectuar o que já foi descrito em menos tempo do que a plataforma Java. Contudo, a aplicação em Java mantém semelhante desempenho para diversos utilizadores em simultâneos, enquanto que a aplicação em *Rails* começa a demonstrar incapacidade de resposta aceitável quando se encontram 200 utilizadores sobre a mesma. E olhando para os valores em concreto, quando se encontram 50 utilizadores nas aplicações, a diferença entre as duas plataformas é de 282 milissegundos para a reserva de táxis e de 230 milissegundos para a reserva do carro, levando inicialmente a crer que *Rails* consegue responder mais rápido do que Java. Apesar desta tendência na diferença gritante entre as plataformas manter-se até que existam 200 utilizadores em simultâneo nas aplicações. Já, nesse caso, percebe-se que a diferença é de 318 milissegundos para a reserva do táxi e de 229 milissegundos para a do carro, levando a supor que Java, apesar de inicialmente apresentar pior desempenho, conforme o aumento de utilizadores na aplicação a capacidade de resposta e execução de métodos tende a estabilizar sem que apresente um crescimento desproporcional.

6.4.3 Database Profiling

Uma base de dados é uma parte importante de qualquer aplicação *web* dinâmica e como tal, o tempo gasto pelo *framework* que trata de efectuar a ligação e construir a *query* a executar é um aspecto que deve ser tomado em conta. Assim, nesta secção é feita uma análise mais genérica dos resultados obtidos durante os diversos testes realizados sobre as aplicações. Após essa análise, serão escolhidas algumas acções que tenham algum significado dentro do contexto de *Database Profiling*, como forma de demonstrar o comportamento das duas ORM que poderá não ser perceptível numa análise mais genérica. É importante referir que todas as construções de *queries* realizadas, quer no *Hibernate* quer no *ActiveRecord*, foram feitas sempre usando para tal os métodos que cada um dos *frameworks* oferece e sem nunca recorrer à escrita manual da *query* a executar, como forma de garantir que a ORM em questão era explorada ao máximo. Antes de passar à análise em concreto, convém mencionar que algumas acções apresentam valores nulos na plataforma Java em relação aos valores na plataforma RoR, pois em *Rails* foi utilizado uma *gem* que implementa todas as operações usuais relacionadas com a gestão de utilizadores, retirando assim o controlo ao programador sobre o que é realmente efectuado em cada acção da aplicação e da implementação da gestão.

Nas tabelas E.9, E.10, E.11 e E.12 presentes no apêndice E.3, são exibidos os resultados dos testes efectuados para 50, 100, 150 e 200 utilizadores em relação à faceta *Average Time in DB Operations*. Esta

faceta, como já foi mencionado anteriormente, analisa tudo o que foi executado pela ORM em relação a operações de base de dados, sendo que as operações recolhidas também foram referidas na secção 6.4. Então, através de uma observação minuciosa das diversas tabelas, consegue-se concluir que globalmente o *framework ActiveRecord* possui um melhor desempenho em relação ao *Hibernate*. Relativamente às acções que respondem em formato *JSON* e que envolvem *queries* de selecção através de uma chave primária, consegue-se perceber que o *ActiveRecord* despende menos tempo em operações relacionadas com a base de dados. Contudo, na tabela que refere a 200 utilizadores simultâneos, o *Hibernate* é o *framework* que globalmente gasta menos tempo nesse tipo de operações, revelando a sua capacidade para tratar de um número maior de pedidos simultâneos e de uma gestão mais eficaz da *connection pool* usada para obter as ligações à base de dados.

Depois de uma análise mais global aos resultados obtidos, é importante observar em pormenor e escolher algumas acções das aplicações que contenham diferentes *queries* para uma análise mais cuidada e objectiva. Assim, as acções escolhidas para sofrerem uma observação mais minuciosa são as seguintes : "*Registo*", "*Update do Perfil*", "*Página das Reservas*", "*Pesq. de Táxis*", "*Reserva de Táxis*", "*Pesq. de Carros*" e "*Reserva do Carro*".

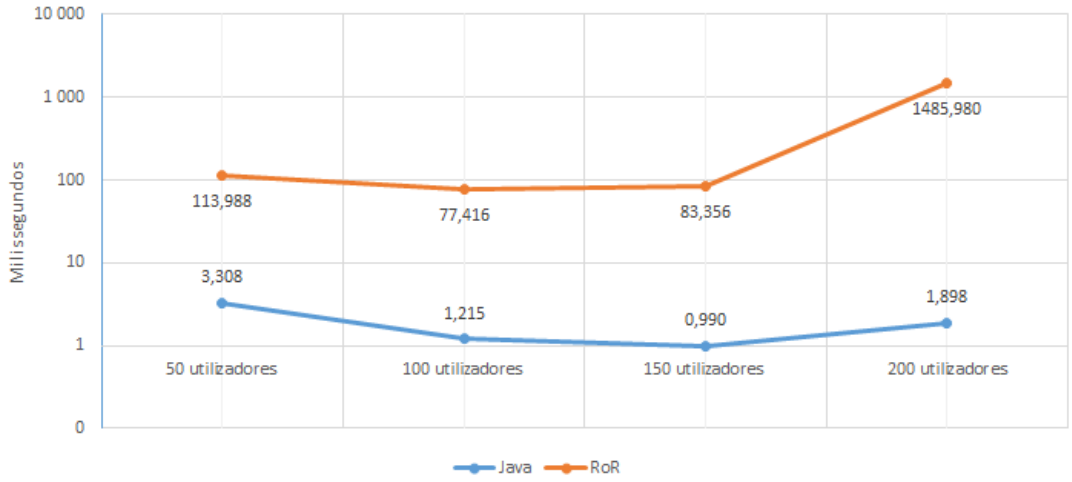


Figura 6.12: Average Time in DB Operations - Registo

Então, começando pela acção "*Registo*", que representa a criação de um novo utilizador na aplicação, o comportamento dela no que toca à faceta *Average Time in DB Operations* nas duas plataformas é apresentado na figura 6.12. Desta figura, é possível concluir que a plataforma Java possui em média um melhor desempenho no que toca à criação de registos, chegando a possuir um diferença entre as duas plataformas de 1484 milissegundos em termos de tempo passado em operações relacionadas com a base de dados. É de apontar que o *Rails*, quando contém 200 utilizadores a efectuarem pedidos, começa a não conseguir suportar os pedidos e a causar atraso nos demais, enquanto que o Java nesse aspecto é bastante estável.

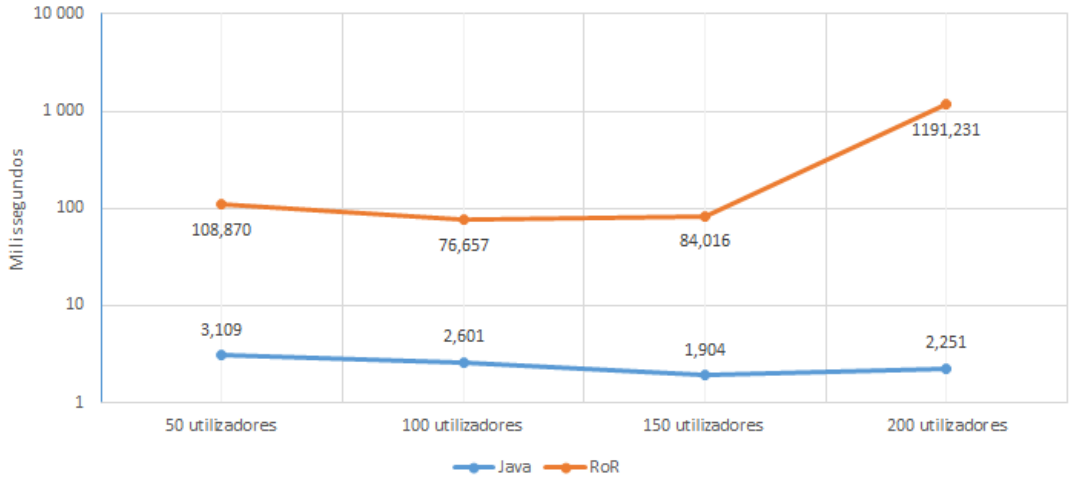


Figura 6.13: Average Time in DB Operations - Update do Perfil

Para a acção "*Update do Perfil*", que representa a actualização dos dados de um utilizador, a plataforma Java demonstra um melhor desempenho ao longo dos diversos números de utilizadores usados nos testes, visível através da figura 6.13. E como aconteceu com a acção apresentada acima, consegue-se concluir que em termos de operações que envolvem actualizações de dados, a plataforma Java despende menos tempo nesse tipo de operações do que a plataforma RoR, chegando, inicialmente para 50 utilizadores, a ter uma diferença de 106 milissegundos. Enquanto que para quando se encontram 200 utilizadores nas aplicações, essa diferença crescer astronomicamente para 1189 milissegundos.

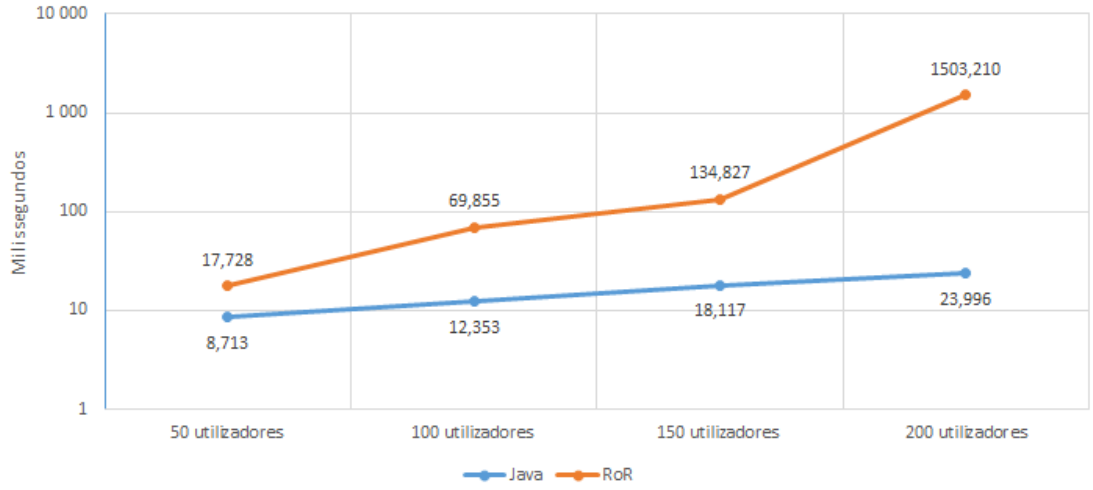


Figura 6.14: Average Time in DB Operations - Página das Reservas

Já em relação à acção "*Página das Reservas*", onde é listado até um máximo de duzentos registos de reservas de carros e de táxis de apenas um utilizador, a plataforma Java apresenta um aumento linear do tempo passado em operações relacionadas com a base de dados mas que em comparação com *Rails*,

ainda é capaz de obter um melhor desempenho nesse aspecto. Isto pode-se verificar facilmente através da observação da figura 6.14, que expõe o tempo despendido pelas duas plataformas nesta acção com números de utilizadores simultâneos.

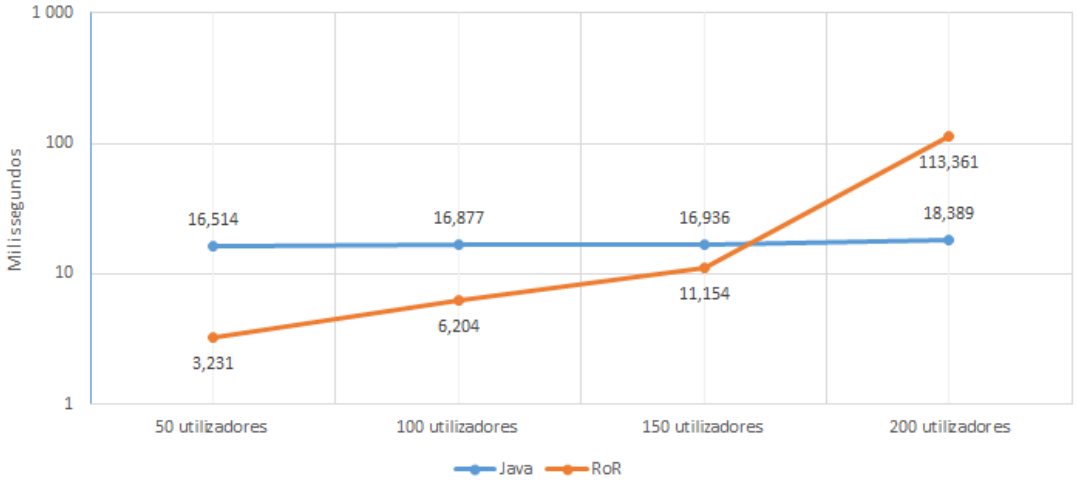


Figura 6.15: Average Time in DB Operations - Pesq. de Táxis

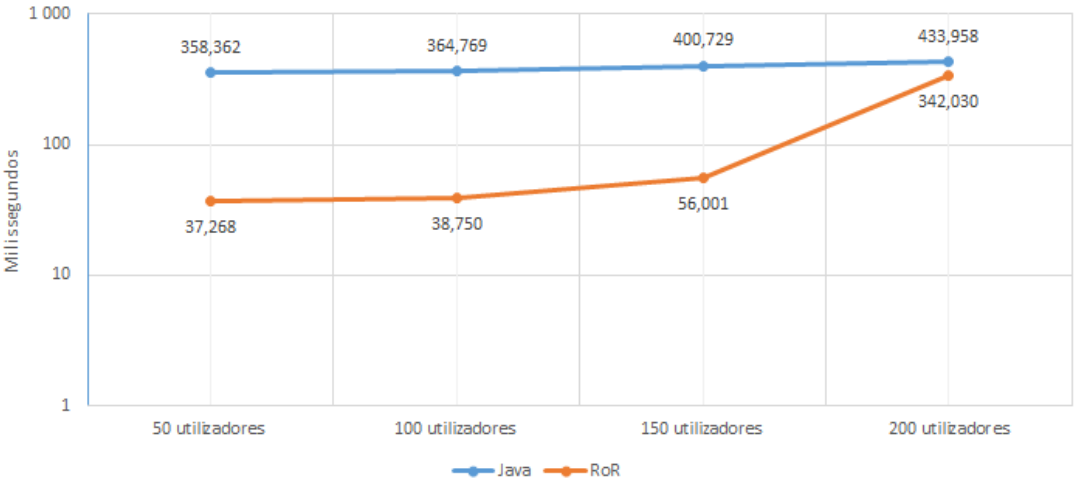


Figura 6.16: Average Time in DB Operations - Pesq. de Carros

Nas figuras 6.15 e 6.16, é apresentado o comportamento das acções de Pesquisa de Táxis e de Carros, respectivamente, com vários utilizadores simultâneos nas aplicações. Verifica-se que até 150 utilizadores, a plataforma RoR executa em menos tempo os métodos necessários para responder a esses pedidos mas que ao chegar aos 200 utilizadores, esse tempo gasto cresce bastante. Contudo, o desempenho demonstrado pela aplicação em Java nas duas acções é constante e sem apresentar grandes oscilações, levando à conclusão que essa plataforma conseguiria lidar com um maior número de pedidos. Porém, é necessário

olhar para as diferenças apresentadas entre as duas plataformas nas diversas situações escolhidas. Assim, quando se encontram 50 utilizadores nas aplicações, a diferença de tempo de execução entre elas é de 13 milissegundos para a pesquisa de táxis e de 321 milissegundos para a pesquisa de carros. Enquanto que, aos 200 utilizadores, essa diferença é de 95 milissegundos para a pesquisa de táxis e de 92 milissegundos para a de carros.

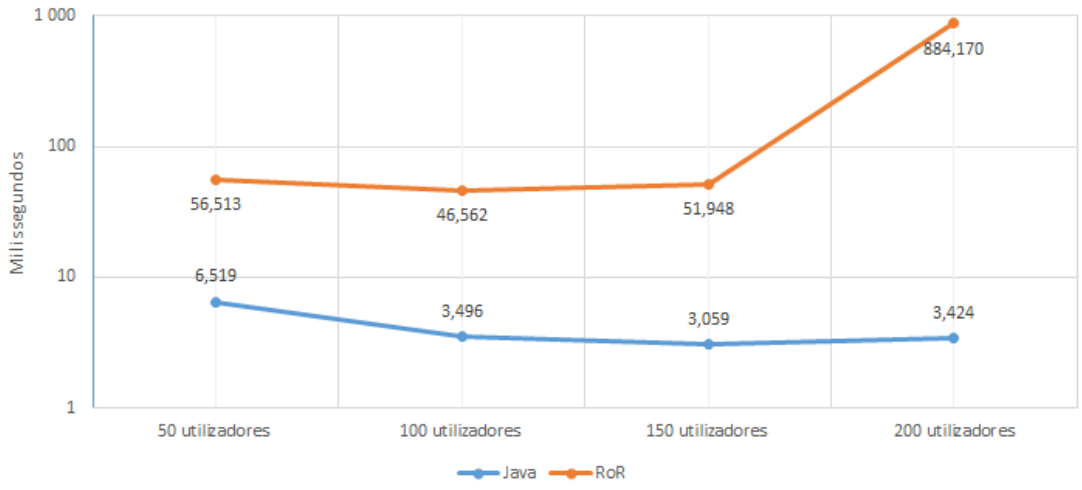


Figura 6.17: Average Time in DB Operations - Reserva de Táxis

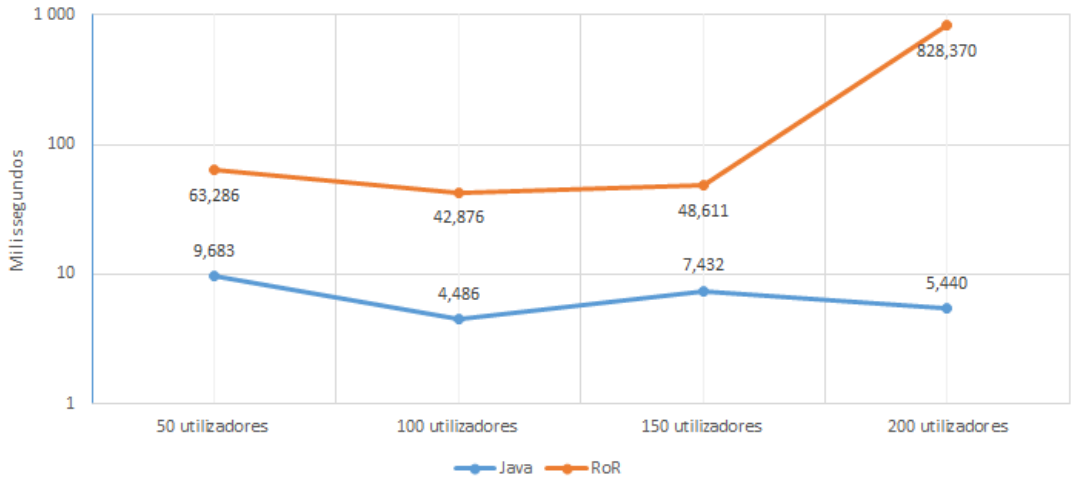


Figura 6.18: Average Time in DB Operations - Reserva do Carro

Por fim, nas figuras 6.17 e 6.18, confirma-se o que já foi exposto anteriormente no que toca a inserção de novos dados numa base de dados pelas duas plataformas de desenvolvimento *web*, a plataforma Java obtém um melhor desempenho com diferentes números de utilizadores simultâneos. Em termos de valores mais específicos e como forma de mostrar as diferenças existentes entre as aplicações, pode-se afirmar que a

diferença entre elas quando tem 200 utilizadores a efectuarem pedidos ao servidor é de 881 milissegundos no caso da acção "*Reserva de Táxis*" e no caso da acção "*Reserva do Carro*" é de 823 milissegundos, o que a longo prazo influencia drasticamente o desempenho global da aplicação desenvolvida em RoR. Contudo, há que apontar o comportamento da linha da aplicação Java que, nos dois casos, se mantém estável ao longo do crescimento do número de utilizadores presentes na aplicações, ao contrário da linha da aplicação *Rails* que dispara para valores muito grandes quando a aplicação possui 200 utilizadores.

6.4.4 Apreciações Finais

Após uma análise exaustiva a cada categoria de *Web Profiling*, é importante retirar algumas conclusões e tecer alguns comentários que possam ser considerados relevantes dentro do contexto da aplicação desenvolvida. Para tal, como já foi referido numa das categorias anteriores expostas, será utilizado uma das facetas que melhor expressa o desempenho de cada acção da aplicação para averiguar qual plataforma se encontra actualmente com melhor desempenho e capacidade de resposta ao utilizador. A faceta em questão é a *Action Average Response Time*, que aglomera nela o tempo gasto por cada camada estudada da aplicação e ainda outros tempos que não pertencem a uma camada especifica da arquitectura MVC mas que são importantes para o correcto funcionamento da aplicação.

Mas antes de falarmos dos resultados referentes à faceta mencionada, convém tecer alguns comentários mais genéricos do que foi apresentado nas subsecções anteriores e que poderão ser confirmados posteriormente pela observação da dita faceta. Assim de uma forma global, pode-se dizer que a plataforma Java despende menos tempo nas camadas referentes à *View* e ao *Controller*, levando a que consiga satisfazer mais pedidos do utilizador do que RoR. Por outro lado, o *Rails* consegue passar menos tempo em pedidos à base de dados do que Java. Contudo, esta pequena vitória global é eclipsada quando se começa a verificar em detalhe algumas das acções mais complexas e se verifica afinal que, apesar do Java ser mais lento nessa camada e conforme o número de utilizadores aumenta, essa desvantagem inicial torna-se numa vantagem, possibilitando assim numa capacidade de resposta por parte da plataforma Java mais estável em comparação com o súbito aumento desse tempo de resposta por parte do *Rails*.

Expondo agora os resultados referentes à faceta mencionada que poderão ser visíveis nas tabelas E.5, E.6, E.7 e E.8 do apêndice E.2 e que irão confirmar o que foi dito anteriormente acerca das duas plataformas. Assim, de um modo geral, a observar essas tabelas na coluna referente à faceta, é possível verificar que a plataforma Java consegue obter um melhor desempenho global conforme o número de utilizadores simultâneos na aplicação cresce gradualmente. Isto pode ser confirmado através da figura 6.19, que representa a mediana de todas as acções presentes em cada uma das aplicações e onde se consegue observar que a plataforma Java responde em menos tempo ao utilizador do que a plataforma RoR. A partir do gráfico, também se consegue perceber uma tendência recorrente que já apareceu nas categorias do *Web Profiling* estudadas anteriormente e que agora pode-ser confirmada. Esta tendência diz respeito à incapacidade da

aplicação desenvolvida em *Rails* de conseguir manter um bom tempo de resposta quando se encontram 200 utilizadores sobre ela.

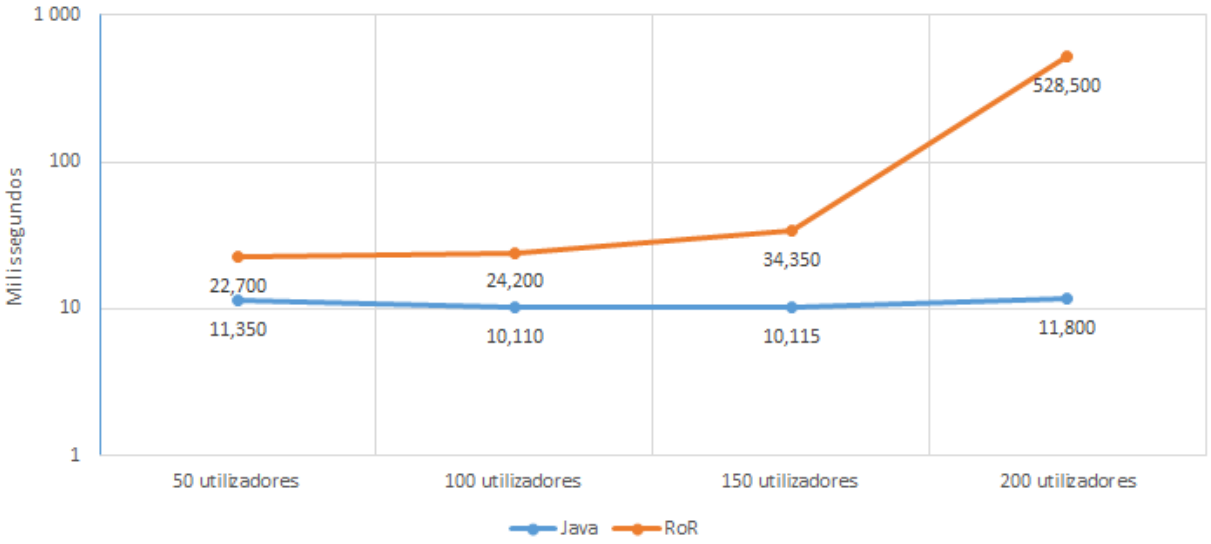


Figura 6.19: Mediana de todas as acções das aplicações na faceta Action Average Response Time

6.5 Análise de Escalabilidade

A escalabilidade é a capacidade que um sistema informático possui para crescer face à demanda de utilizadores sem que este perca os seus atributos ímpares e mantenha a qualidade de serviço previamente oferecido ao cliente final. Esta característica é considerada importante para o sucesso e crescimento de uma empresa no ramo informático e existem duas formas de escalar um sistema informática mas que não serão abordadas, pois não se encontram no âmbito deste estudo. Durante esta secção, será tentada uma análise superficial à escalabilidade das aplicações *web* desenvolvidas, utilizando para tal a ferramenta *New Relic* que, para além do que foi falado acerca dela, oferece alguns gráficos a respeito desta questão muito importante actualmente. Estes gráficos não são categóricos nem devem ser lidos de uma forma linear, pois eles oferecem informação relativa a um período de tempo em que aplicação teve utilizadores a usufruírem dela e devem ser usados para compreender o comportamento da aplicação quando sofre picos de visitas. Os gráficos que serão abordados posteriormente nesta secção apenas se referem ao tempo de resposta (*response time*) das aplicações, visto que os outros dois tipos de gráficos disponibilizados pela ferramenta representam o tempo de resposta da base de dados e do *CPU* do servidor e estes podem sempre ser melhorados já que se encaixa numa das formas de escalar uma aplicação *web*.

Sendo assim, antes de serem apresentados os gráficos que permitem efectuar uma análise à escalabilidade de cada uma das aplicações *web*, é necessário referir a forma como foi implementado o aumento gradual de utilizadores em cada uma das aplicações. Este aumento gradual foi implementado devido sobre-

tudo à necessidade de emular um crescimento substancial de uso da aplicação ao longo de um determinado período de tempo sem que houvesse alguma pausa temporal entre esse crescimento para imitar ao máximo o que podia acontecer realmente. Para tal ser realizado, foi utilizado mais uma vez a ferramenta *JMeter*, anteriormente discutida, através do *plugin Stepping Thread Group* que garante que o número de utilizadores presentes na aplicação é constante ao longo de um período de tempo definido e foram definidas várias *Thread Groups* com os utilizadores desejados.

Então, na figura 6.20, é possível observar o esperado plano para emular um aumento gradual do número de utilizadores. Dessa figura, também se sabe que, passada uma hora, o número de utilizadores cresce em 50, sendo que o número máximo de utilizadores a efectuarem pedidos ao servidor é de 200.

Figura 6.20: Esperado Plano de utilizadores sobre as aplicações

Assim, após a execução do plano de testes exibido na figura acima apresentada, é necessário visitar a ferramenta *New Relic* e retirar o gráfico referente ao tempo de resposta de ambas as aplicações para realizar uma análise para retirar alguma conclusão se possível. Nos gráficos que serão apresentados, há que referir que a cor dos pontos presentes neles representam as horas do dia em que a aplicação tratou de pedidos do utilizador e que a linha horizontal representa a mediana local. Em relação aos eixos do gráfico, o eixo X simboliza os pedidos recebidos por minuto (**R**equests **P**er **M**inute), normalmente referido como *throughput*, enquanto que o eixo Y simboliza o tempo de resposta da aplicação em milissegundos.

Então, nas figuras 6.21 e 6.22 são expostos os gráficos para a plataforma Java e *Rails*, respectivamente.

Para poder conseguir efectuar um análise correcta desses gráficos, é preciso conhecer o que pretendem representar no contexto de uma aplicação. Para tal ser possível, é necessário consultar a documentação da ferramenta *New Relic* e segundo o que ela afirma acerca deste gráfico, para uma aplicação que esteja a escalar bem, o tempo de resposta deve ser igual ou próximo a horizontal. Isto é, a linha da mediana deve ser o mais aproximadamente possível da horizontal.

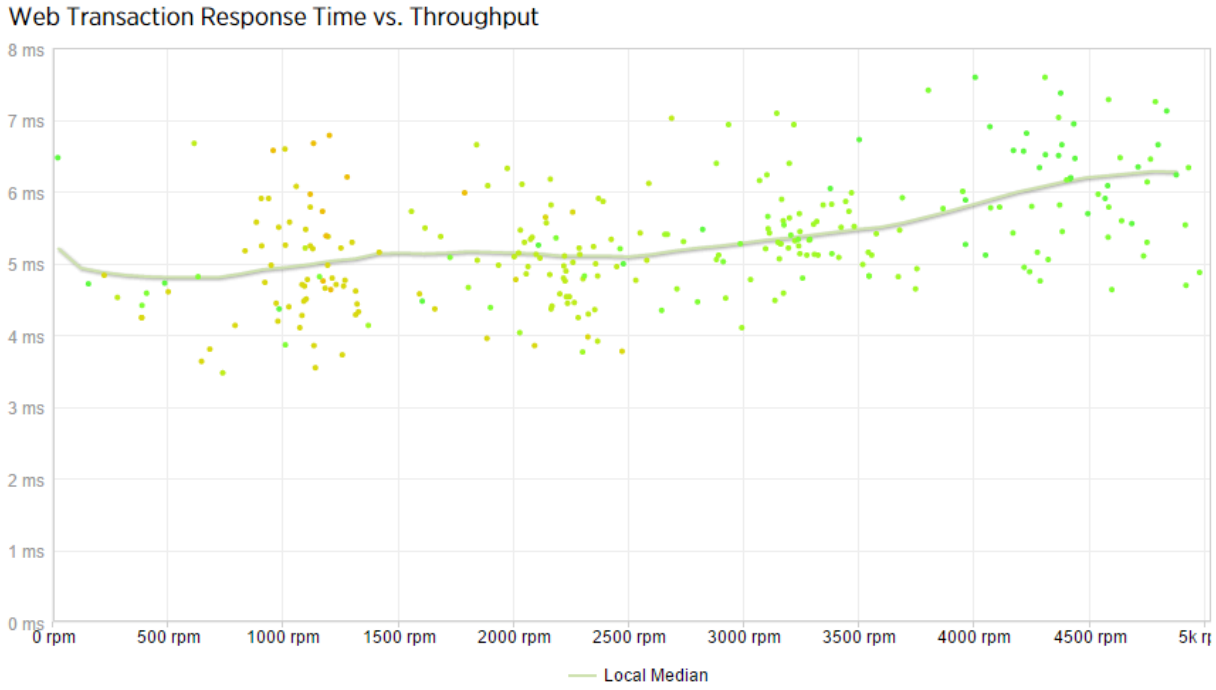


Figura 6.21: Análise de Escalabilidade - Response Time na plataforma Java

Assim sendo, a observar a figura 6.21 que se refere ao tempo de resposta da aplicação em Java, pode-se afirmar que está a escalar bem a aplicação dentro dos testes realizados, visto que a linha da mediana é aproximadamente horizontal, sendo que apenas no fim dessa linha é que acontece um crescimento mas que não é o suficiente para garantir que a aplicação não está a escalar bem.

Já, na figura 6.22, é apresentado o gráfico da aplicação em RoR referente ao tempo de resposta da mesma. Através da sua observação, é visível que a linha da mediana é horizontal para maior parte dos pedidos recebidos mas quando atinge os 300 *RPM*, a aplicação deixa de conseguir manter o tempo de resposta dentro do esperado. A partir disto, é possível afirmar que seria necessário adicionar mais capacidade processamento ao servidor na forma de mais cores ou de um sistema distribuídos de servidores, visto que existe uma degradação do tempo de resposta e que seria visível e inaceitável para o utilizador final a demora de uma resposta.

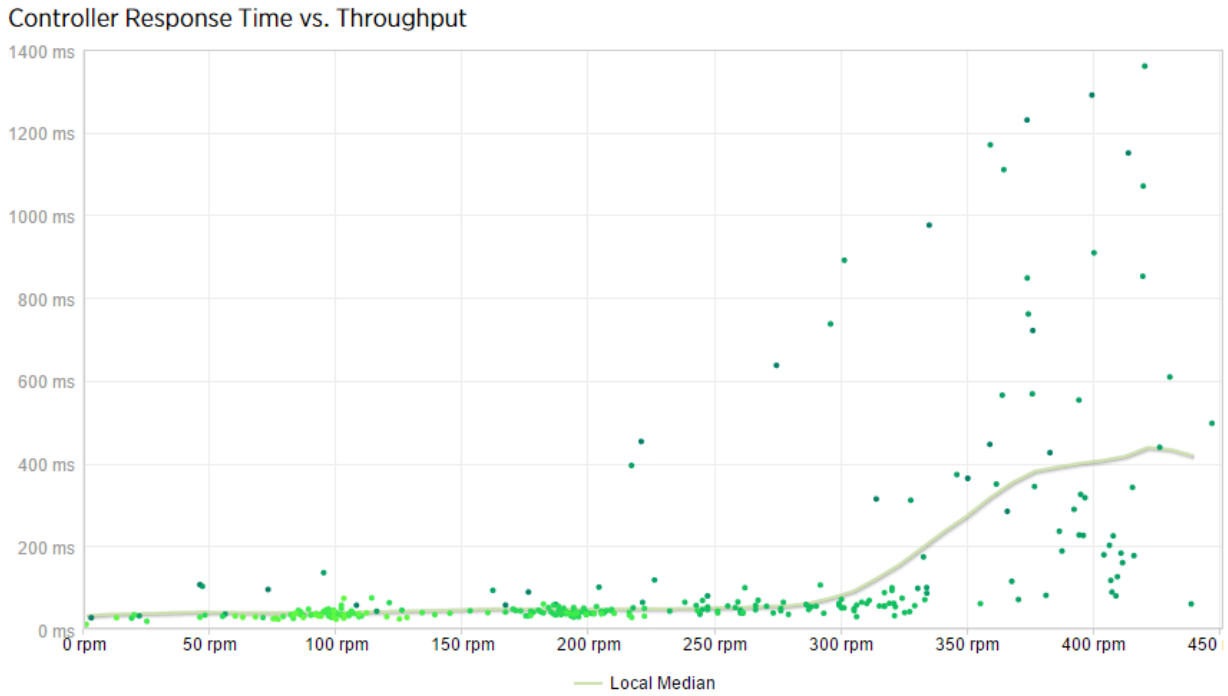


Figura 6.22: Análise de Escalabilidade - Response Time na plataforma RoR

Por fim, convém mencionar que estes resultados podem sofrer alterações mediante outras configurações do servidor que se encontra a servir a aplicação ao utilizador, bem como das configurações usadas nas *connection pools* das duas aplicações. As configurações e as características do computador utilizado para cada aplicação podem ser encontrados nos apêndices B e A. Também é importante deixar como nota que a documentação encontrada para a configuração de cada servidor utilizado e das suas *pools* não é muito específica no que conta a optimização do mesmo através da sua configuração, sem recorrer à mais computadores ou melhores características no servidor.

Capítulo 7

Conclusão

A crescente popularização e utilização de aplicações *web* no quotidiano do ser humano para a realização de uma variada gama de actividades levou à necessidade de conhecer em detalhe o funcionamento interno das linguagens usadas para a sua criação por parte das empresas. Esta preocupação surgiu devido, em alguns casos, à uma grande demanda de utilização do serviço fornecido. Em certos casos, como o da aplicação *web* *Citius* do Governo Português, que quando passaram de um ambiente de desenvolvimento para um ambiente de utilização diária e constante, simplesmente não aguentaram a quantidade de informação recebida e pedida, colocando assim em dúvida todo o processamento de desenvolvimento de uma aplicação *web* para que esta seja lançada de uma forma eficaz. Para colmatar estes casos de fracasso inicial, é importante saber detalhadamente o funcionamento da aplicação desenvolvida e para atingir esse nível de conhecimento, deve-se utilizar a área de Engenharia de Software designada de *Profiling*, sendo que no caso mais específico de aplicações *web*, deve-se estudar a área de *Web Profiling*.

Inicialmente, foi estudado a área de *Profiling* e de *Web Profiling* e quais vertentes elas poderiam avaliar na execução de uma aplicação. A área de *Profiling* consegue avaliar características únicas ou facetas relativas ao tempo despendido pelo *CPU* na execução de métodos, pela memória utilizada durante a execução, bem como, no caso de linguagens orientadas pelos objectos, o funcionamento do *Gargabe Colllector* e entre outras. Enquanto que a área de *Web Profiling* preocupa-se maioritariamente com o tempo despendido em cada uma das camadas da arquitectura MVC. Esta variada de facetas descobertas através das ferramentas já faladas conduziu a uma necessidade de associar-las às diferentes categorias para uma leitura e procura do que se pretende saber mais simples e objectiva, visto não existir uma clara categorização delas. E com esta categorização finalizada, um dos principais objectivos deste trabalho foi cumprido, relevando um avanço considerável numa área que se encontra pouco estudada e documentada apesar da sua crescente importância no mundo empresarial.

Outro ponto importante antes de passar à apresentação das facetas em categorias de *Profiling*, foi estudar as duas plataformas escolhidas para o estudo comparativo e as ferramentas associadas a elas que

permitissem efectuar *profiling*. Inicialmente, foi estudado a linguagem Java e Ruby para que depois fosse compreendido a forma como a plataforma Java e a plataforma RoR implementam a arquitectura MVC e que tecnologias oferecem para essa implementação. Durante o estudo das plataformas, chegou-se à conclusão que na plataforma Java para implementar a arquitectura desejada seria necessário seleccionar dentro das várias hipóteses possíveis as tecnologias que posteriormente seriam usadas para implementar cada camada da arquitectura MVC. Enquanto que na plataforma RoR, essa selecção não foi feita, já que a plataforma oferece tudo o que é necessário para criar uma aplicação *web* sem ter que tomar qualquer decisão.

Depois de compreender melhor o funcionamento interno das duas plataformas, foi feita uma recolha e separação nas diversas categorias de *Profiling* das facetas que as ferramentas associadas às plataformas conseguem analisar. Durante este processo, foi denotado uma certa quantidade de ferramentas pagas e com apenas alguns dias para experimentar e também a falta de ferramentas capazes de analisar aplicações *web* seguindo uma aproximação à arquitectura multi-camada. Mas apesar desta falha, elas conseguiam compensar através da oferta de uma boa análise da camada referente à *Model*. Após essa recolha e separação, foi efectuada a escolha das facetas que seriam usadas para comparar e caracterizar um *website*.

Para conseguir efectuar uma comparação entre as duas plataformas, seria necessário desenvolver uma aplicação em ambas como forma de possuir a mesma base de comparação entre elas. Durante o desenvolvimento delas, foi aprofundado o estudo das plataformas e restringida as tecnologias a utilizar na implementação da arquitectura multi-camada, bem como os servidores usualmente utilizados para executarem as aplicações dentro do contexto da plataforma. Para além disso, foi estudada uma forma de aumentar o desempenho de uma aplicação que possuí uma base de dados como fonte de informação, conhecido como *connection pooling*. Dentro dela, foi analisado o que era e a forma de implementar em cada uma das plataformas essa técnica para que as aplicações se aproximassem o máximo possível de uma aplicação lançada ao público para uso geral e contínuo. Nesta parte, foi encontrada uma estranha ocorrência, *Rails* não necessita de nenhum mecanismo ou tecnologia externa para implementar esta técnica, visto que a própria plataforma trata disso sozinha. Já na plataforma Java, foi preciso procurar as soluções existentes para este mecanismo e seleccionar aquela que oferecesse garantias de funcionamento e configuração simples.

Após o desenvolvimento concluído e com as aplicações colocadas a funcionar em servidores, seria necessário arranjar uma forma de gerar o uso delas por parte de utilizadores reais. Mas como o número de utilizadores desejados para ter dados credíveis acerca da execução delas não seria fácil de alcançar, foi estudada mais uma área de Engenharia de Software que também deve ser levada em consideração quando se pretende conhecer o comportamento de um *website*, sendo essa área a de *Performance Testing*. Após este estudo inicial para compreender onde se poderia aplicar a área e quais as ferramentas que estão disponíveis no mercado que fornecem a capacidade de gerar dados de utilização aleatórios, foi então escolhida apenas uma ferramenta, o *Apache JMeter*, que iria ser a responsável pela geração de *load* sobre as aplicações. Da

utilização desta ferramenta, conclui-se que, para uma ferramenta *open-sourece*, ela permite efectuar um variado número de tarefas consideradas importantes dentro desta área.

Então, foram efectuados os testes desenhados às duas aplicações e recolhido os diversos dados das suas execuções. Nos testes efectuados, foram utilizados 50, 100, 150 e 200 utilizadores em simultâneo nas aplicações durante três horas a efectuarem pedidos ao servidor. Após a conclusão dos vários testes e da distribuição dos diversos resultados pelas facetas escolhidas das categorias de *Web Profiling*, foi efectuado uma análise crítica e objectiva aos resultados obtidos que eram um dos principais objectivos e condicionantes deste estudo. Dessa análise, pode-se afirmar que a plataforma Java apresenta globalmente um melhor desempenho do que a plataforma RoR, isto é, consegue responder a um pedido do utilizador em menos tempo do que a outra plataforma. Mais ainda, em termos das categorias de *Web Profiling*, Java consegue obter um melhor desempenho na categoria de *View* e *Controller Profiling*, enquanto que *Rails* obtém uma melhor performance na *Database Profiling*. Para além disto, foi feita uma análise de escalabilidade às duas aplicações e aí como foi visto durante a análise dos testes sobre as aplicações, chegou-se à conclusão que a aplicação Java não iria precisar de escalar por enquanto. Mas por outro lado, a aplicação *Rails* iria necessitar escalar imediatamente se quisesse suportar 200 ou mais utilizadores.

A tarefa de comparar duas plataformas de desenvolvimento *web* revelou-se um desafio, devido à não existência de uma forma sistemática de efectuar essa comparação objectivamente e também à complexidade inerente a uma aplicação *web* multi-camada. Esta complexidade surge do facto que uma aplicação *web* é constituída por um conjunto de acções ou *use cases* e que nem sempre é simples distinguir o que pertence ao *Model*, ou *View* ou *Controller* quando uma acção é executada. Então, foi concebida uma metodologia que permite efectuar uma análise objectiva às plataformas através da clarificação do que é cada componente executado na aplicação e a qual das categorias de *Web Profiling* pertence. Para além disso, também foi utilizada uma abordagem ao nível da acções que o utilizador pode executar, sendo que estas podem ser vistas, genericamente, como sendo um *Use Case* do processo de modelação do sistema informático. Disto tudo, conclui-se que, ao efectuar *profiling* sobre uma aplicação *web* e seguindo a metodologia implementada neste estudo, se estaria a efectuar uma análise ao desempenho de um Use Case nas diversas camadas da arquitectura MVC. Com isto, poderia-se descobrir, em maior detalhe, qual dos *Use Cases* desenhados aquando da modulação da aplicação estaria a afectar o desempenho global e também averiguar se não estaria-se a colocar demasiado peso num *Use Case*, tornando assim, este estudo, um avanço importante para a análise minuciosa do comportamento de qualquer aplicação *web* e da evolução da área de *Web Profiling*.

Em jeito de conclusão e possíveis utilizações deste estudo comparativo para trabalhos futuros, é importante referir que as tecnologias e as ferramentas utilizadas tendem a evoluir constantemente e que os resultados obtidos estão totalmente dependentes das versões utilizadas durante o desenvolvimento das apli-

cações e dos servidores. No que toca à categorização das facetas recolhidas, seria interessante futuramente verificar a evolução das ferramentas que fornecem os dados de *profiling* e perceber se existiu algum alteração das facetas analisáveis. Para além disso, também se poderia utilizar essa mesma categorização para comparar outras plataformas de desenvolvimentos *web* populares com aquelas mais estáveis e usadas no mercado actualmente. Por fim, convém referir que apesar do *Ruby on Rails* se ter tornado uma tendência crescente no desenvolvimento de aplicações, Java, com a sua estabilidade e suporte fornecido pela empresa *Oracle*, ainda consegue oferecer uma boa qualidade de opções para desenvolvimento em conjunção com bons tempos de resposta ao utilizador final.

Bibliografia

[1] JOC/EFR, "William thomson (lord kelvin)," http://turnbull.mcs.st-and.ac.uk/history/Mathematicians/Thomson.html/, October 2003. [Online]. Available: http://turnbull.mcs.st-and.ac.uk/history/Mathematicians/Thomson.html

[2] D. Dhyani, W. K. Ng, and S. S. Bhowmick, "A survey of web metrics," *ACM Comput. Surv.*, vol. 34, no. 4, pp. 469–503, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/592642.592645

[3] H. M. Aguiar, "Profiling of real-world web applications," Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologias, Novembro 2010. [Online]. Available: http://run.unl.pt/handle/10362/7608

[4] N. Relic, "Apdex: Measuring user satisfaction," https://docs.newrelic.com/docs/site/apdex-measuring-user-satisfaction, 2012. [Online]. Available: https://docs.newrelic.com/docs/site/apdex-measuring-user-satisfaction

[5] D. H. Hansson, "Rails is omakase," December 2012. [Online]. Available: http://david.heinemeierhansson.com/2012/rails-is-omakase.html

[6] M. Tanikella, "Practical challenges of profiler integration with java/j2ee applications," http://www.theserverside.com/feature/Practical-Challenges-of-Profiler-Integration-with-Java-J2EE-Applications, Outubro 2011.

[7] F. Mário Martins, *Java6 e Programação Orientada pelos Objectos*, 2nd ed., F. E. Informática, Ed. FCA, 2009.

[8] Y. Matsumoto, "Ruby : A programmers' best friend," https://www.ruby-lang.org, 2014. [Online]. Available: https://www.ruby-lang.org

[9] Matz, "Speaking on the ruby-talk mailing list," http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773, May 2000. [Online]. Available: http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2773

[10] Tiobe, "Tiobe index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, Setembro 2014. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[11] S. Metz, *Practical Object-Oriented Design in Ruby: An Agile Primer*, 1st ed. Addison-Wesley Professional, 2012.

[12] M. Hartl, *Ruby on Rails 4 Tutorial: Learn Rails by Example*, ser. Addison-Wesley Professional Ruby Series. Pearson Education, 2013. [Online]. Available: http://ruby.railstutorial.org/ruby-on-rails-tutorial-book

[13] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz, *Agile Web Development with Rails 4*. Pragmatic Bookshelf, 2013.

[14] H. Lai, "Ruby on rails server options," Fevereiro 2014. [Online]. Available: http://stackoverflow.com/questions/4113299/ruby-on-rails-server-options/4113570#4113570

[15] ——, "Puma versus phusion passenger," Fevereiro 2014. [Online]. Available: https://github.com/phusion/passenger/wiki/Puma-vs-Phusion-Passenger

[16] ——, "Unicorn versus phusion passenger," Fevereiro 2014. [Online]. Available: https://github.com/phusion/passenger/wiki/Unicorn-vs-Phusion-Passenger

[17] S. Visveswaran, "Dive into connection pooling with j2ee," Outubro 2000. [Online]. Available: http://www.javaworld.com/article/2076221/jndi/dive-into-connection-pooling-with-j2ee.html

[18] B. Wooldridge, "Pool analysis," Março 2014. [Online]. Available: https://github.com/brettwooldridge/HikariCP/wiki/Pool-Analysis

[19] ——, "Jmh benchmarks," 2014. [Online]. Available: https://github.com/brettwooldridge/HikariCP

[20] J. Rochkind, "Activerecord concurrency in rails4: Avoid leaked connections!" Julho 2014. [Online]. Available: http://bibwild.wordpress.com/2014/07/17/activerecord-concurrency-in-rails4-avoid-leaked-connections/

[21] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, "Performance testing guidance for web applications," Microsoft, Tech. Rep., 2007. [Online]. Available: http://perftestingguide.codeplex.com/

[22] Neustar, "Load testing best practices: Do try this at home," 2014. [Online]. Available: http://www.neustar.biz/enterprise/docs/whitepapers/web-performance/best-practices-for-load-testing.pdf

Apêndice A

Instalação e configuração do servidor de Ruby on Rails

Durante este capítulo, será apresentado o processo necessário para criar um servidor *web* para executar aplicações desenvolvidas em *Ruby on Rails*, bem como os passos efectuados para o *deploy* de uma aplicação desenvolvida em *Rails*. Para além disso, serão expostos os ficheiros de configuração usados no *Nginx* e na própria aplicação. Convém também referir que a máquina utilizada para funcionamento do servidor, possui *Ubuntu Server 14.04 64-bit* como sistema operativo, tem 4GB de memória *RAM*, 20GB de disco rígido e um processador de 2.6 GHz.

A.1 Actualizações do Sistema Operativo

O primeiro passo depois da instalação do sistema operativo é efectuar as actualizações disponibilizadas oficialmente. Estas actualizações são fundamentais para corrigir *bugs* e potenciais problemas de segurança. Como se trata de um sistema operativo desenvolvido para correr com o máximo de *up-time* possível, por defeito o *Ubuntu Server* não faz actualizações automáticas. Essa tarefa estará a cargo do administrador do sistema. Para evitar *down-times* ou possíveis problemas no futuro é aconselhável fazer as actualizações antes da instalação das ferramentas para fazer o *deploy* da aplicação. Assim sendo, é necessário actualizar a lista de *packages* disponíveis nos repositórios do *Ubuntu*, utilizando para tal o seguinte comando :

```
$ sudo apt-get update
```

De seguida, deve-se efectuar as actualizações dos *packages* se existir alguma disponível:

```
$ sudo apt-get upgrade
```

A.2 Ferramentas Úteis

Antes de se começar a instalação dos principais componentes, é importante instalar algumas ferramentas úteis que irão facilitar a configuração dos restantes *softwares*. A primeira destas ferramentas chama-se **curl**

e permite efectuar o *download* de ficheiros a partir de um *URL*. Para instalar a ferramenta, basta executar o seguinte comando no terminal do servidor:

```
$ sudo apt-get install curl
```

E como será necessário efectuar o *download* de ficheiros em servidores *HTTPS*, é também preciso instalar a seguinte biblioteca :

```
$ sudo apt-get install libcurl4-openssl-dev
```

A.3 Ambiente Ruby

Para desenvolver uma aplicação *web* em *Rails*, é necessário instalar todo o *boilerplate* necessário para a aplicação funcionar. O primeiro passo passa por instalar a linguagem e o compilador de *Ruby*. Visto que o *Ruby* está em constante desenvolvimento sendo possível possuir várias versões do Ruby no mesmo computador e desenvolver aplicações em *Rails* para diferentes versões do Ruby, é recomendado pela comunidade de Ruby que se instale o *Ruby Version Manager* (RVM). Esta ferramenta permite instalar e gerir facilmente múltiplos ambientes Ruby no mesmo computador. Para descarregar e instalar esta ferramenta, basta executar a seguinte linha :

```
$ curl -L get.rvm.io | bash -s stable
```

Durante o processo de instalação, será adicionado à *bash* do sistema operativo algumas directivas para que a ferramenta possa ser executada a partir de qualquer directório do sistema operativo sem indicar o caminho completo do RVM. No fim da instalação, é necessário executar o seguinte comando para que a ferramenta possa ser utilizada de imediato.

```
$ source ~/.rvm/scripts/rvm
```

Depois é importante verificar e garantir que as dependências desta ferramenta estão asseguradas, e para tal esta ferramenta disponibiliza um comando que verifica as suas dependências e instala-as de seguida no sistema:

```
$ rvm requirements
```

Depois de concluída a instalação do *RVM*, passamos à instalação do compilador e interpretador *Ruby*. Para tal, basta usar o seguinte comando que permite escolher a versão a instalar.

```
$ rvm install 2.1.2
```

O comando anterior irá demorar alguns minutos. Assim que tiver concluído, estará instalado no servidor a última versão estável do Ruby. Para começar a usar efectivamente esta versão do Ruby em todo o sistema, basta executar o comando :

```
$ rvm --default use 2.1.2
```

E agora a versão do Ruby usada em todo o sistema será a 2.1.2.

A.4 Framework Ruby on Rails

Após a instalação e configuração do ambiente de desenvolvimento Ruby, é preciso instalar o *framework Ruby on Rails* para que se possa desenvolver um aplicação *web*. Como o *framework* RoR trata-se de uma *gem* da linguagem Ruby, então para efectuar a sua instalação basta apenas executar o seguinte comando, visto que a ferramenta RVM fornece-nos alguns comandos úteis para a instalação e gestão de pacotes (*gems*) feitos pela comunidade :

```
$ gem install rails
```

E após a instalação estar finalizada, o sistema possuirá toda a *stack* necessária para que seja desenvolvida uma aplicação *web* padrão em *Ruby on Rails*.

A.5 Phusion Passenger

De modo a fazer *deploy* da aplicação *web* de forma simples e segura, é preciso um servidor aplicacional, sendo o *Phusion Passenger* a solução mais robusta e usada na comunidade de *Rails*. Também seria possível usar outros servidores aplicacionais como o *Puma* ou o *Unicorn*. Este servidor aplicacional escolhido é responsável por gerir dinamicamente os processos associados a cada aplicação consoante o tráfego que o servidor regista. Em caso de *crash*, o *Passenger* também se encarrega de reiniciar automaticamente o processo em causa.

Para podermos instalar este servidor, basta executar na consola o seguinte comando :

```
$ gem install passenger
```

O *Passenger* apenas irá tratar da lógica relacionada com o *Ruby on Rails*. Toda a stack *HTTP* necessária para um servidor *web* funcionar é delegada para o servidor *Apache* ou *Nginx*.

A.6 Nginx

Como foi referido anteriormente, é preciso ter instalado o servidor *Apache* ou o *Nginx* para que o servidor aplicacional funcione correctamente. A escolha recaiu sobre o *Nginx*. Este serve como um *reverse proxy*, é mais rápido e simples que o *Apache*, caracteriza-se por possuir um mecanismo de *cache* HTTP, usar poucos recursos no servidor e sendo ideal para cenários de alta concorrência.

Mas antes de instalarmos o servidor *web* que tratará de todos os pedidos HTTP, é preciso instalar no sistema a linguagem *Node.js* para que o *Ubuntu* e o próprio ambiente *Rails* possam executar em tempo real *Javascript*. Para tal, basta executar os seguintes comandos na consola do servidor:

```
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install nodejs
```

E para que a compilação e instalação do servidor *Nginx* decorra normalmente, é necessário instalar duas dependência que irá facilitar o processo de compilação e instalação.

```
$ sudo apt-get install libpcre3 libpcre3-dev
```

Por defeito, a instalação do *Nginx* seria feita através dos repositórios do *Ubuntu*. Porém, como será usado no contexto do *Phusion Passenger*, é necessário instalar uma versão especialmente desenvolvida para esse fim :

```
$ rvmsudo passenger-install-nginx-module
```

Se forem detectadas dependências em falta, deve-se instalar através das instruções disponibilizadas e reiniciar a instalação executando o comando anterior novamente. No primeiro passo da instalação do *Nginx*, apenas é necessário confirmar carregando na tecla **ENTER** para tal :

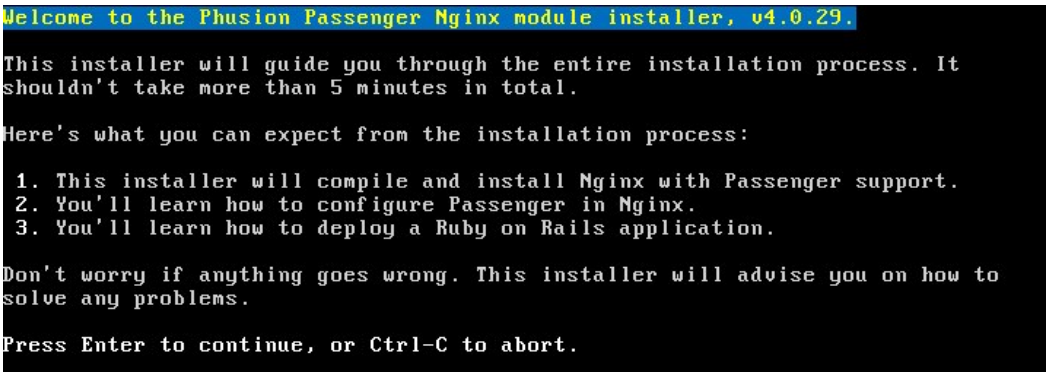


Figura A.1: Primeiro passo da instalação do Nginx

De seguida, são apresentadas duas opções para escolha : 1) podermos instalar automaticamente o *Nginx* ou 2) escolher manualmente as configurações do *Nginx*. É recomendado optar pela opção **1**, pois apenas é preciso a configuração usual do *Nginx* para executar qualquer aplicação *web* em *Ruby on Rails*.

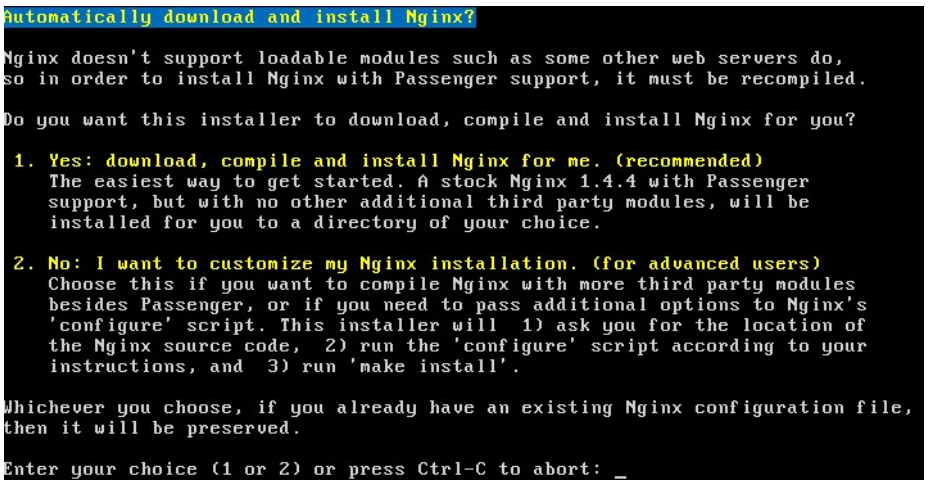


Figura A.2: Segundo passo da instalação do Nginx

Depois de concluída a instalação, o *Nginx* já se encontra a correr. Para confirmar o correcto funcionamento do mesmo, basta visitar o endereço do servidor no *browser*.

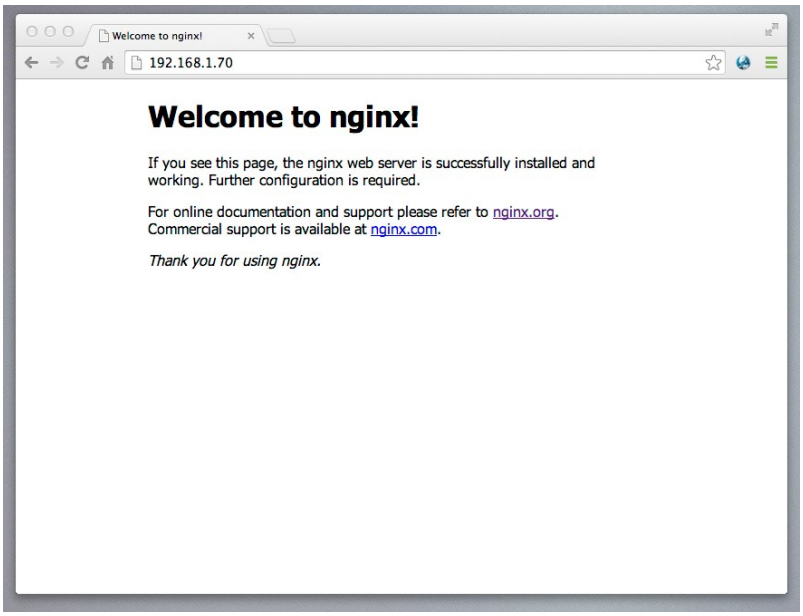


Figura A.3: Página inicial do Nginx

Por último, devemos adicionar o *Nginx* ao conjunto de processos que arrancam com o sistema permitindo assim que, em caso de falha do servidor, este seja automaticamente iniciado sempre que o servidor é iniciado. Para tal, devem ser executados os seguintes comandos que transferem um *script* que fornece já alguns comandos para reiniciar o servidor e adicionam o processo referente ao *Ngnix* ao sistema operativo:

```
$ wget -O init-deb.sh http://library.linode.com/assets/1139-init-deb.sh
$ mv init-deb.sh /etc/init.d/nginx
$ chmod +x /etc/init.d/nginx
$ /usr/sbin/update-rc.d -f nginx defaults
```

A.7 MySQL Server

A base de dados escolhida para a aplicação *web* foi o *MySQL Server*. Para se poder instalar a última versão disponível nos repositórios do *Ubuntu*, basta executar na consola o seguinte comando :

```
$ sudo apt-get install mysql-server
```

Durante o processo de instalação, vai ser pedido para colocar uma password para o utilizador **root**. No fim da instalação, o servidor *MySQL* estará a correr automaticamente.

E antes de instalarmos a *gem* que trata da ligação entre o *Rails* e o *MySQL*, é importante instalarmos algumas bibliotecas necessárias para que essa ligação funcione correctamente:

```
$ sudo apt-get install libmysql-ruby libmysqlclient-dev
```

E por fim, é necessário instalar a *gem* que irá permitir criar ligações entre o *Rails* e o *MySQL*, bastando para tal executar o seguinte comando :

```
$ gem install mysql2
```

A.8 Deploy

Depois de todo o sistema estar configurado, é necessário efectuar o *deploy* da aplicação *web*. O primeiro passo é fazer o *upload* da aplicação para uma directoria à escolha. De seguida, navegar até à directoria escolhida. Neste ponto, deve-se executar uma série de comandos que vão instalar automaticamente todas as dependências, criar e povoar a base de dados a usar pela aplicação *web*.

```
source 'https://rubygems.org'

gem 'rails', '4.1.4'
gem 'mysql2', '~> 0.3.16'
gem 'sass-rails', '~> 4.0.3'
gem 'uglifier', '>= 1.3.0'
gem 'jquery-rails'
gem 'jquery-ui-rails'
gem 'jquery-ui-themes'
gem 'turbolinks'
gem 'jquery-turbolinks'
gem 'jbuilder', '~> 2.1.1'
gem 'foundation-rails'
gem 'foundation-icons-sass-rails'
gem 'foreigner'
gem 'immigrant'
gem 'active_attr'
gem 'activerecord-import', '~> 0.5.0'
gem 'google_directions', '~> 0.1.6.2'
gem 'devise'
gem 'activemerchant'
```

Código A.1: Ficheiro Gemfile

Começando por instalar as dependências da aplicação, visíveis na listagem A.1 que se encontra na raíz da aplicação, basta executar para tal o seguinte comando:

```
$ bundle install
```

Os seguintes comandos servirão para criar e povoar a base de dados no *MySQL* com os dados presentes no ficheiro de *seeds*. Há que referir que os dados que irão preencher a base de dados apenas representam a informação que não será inserida pelo utilizador que usará a aplicação *web*.

```
$ RAILS_ENV=production rake db:create
$ RAILS_ENV=production rake db:migrate
```

```
$ RAILS_ENV=production rake db:seed
```

Após a conclusão dos comandos anteriores, é preciso configurar o servidor *Nginx* para que ele saiba onde se encontra a aplicação que irá executar, bem como algumas configurações extras que servirão como forma de optimizar a gestão feita pelo *Ngnix* da aplicação *web*. Para alterar a configuração base do *Ngnix*, é preciso editar o ficheiro de configuração (*nginx.conf*) do servidor que se encontra na pasta */opt/nginx/conf*. Na listagem seguinte, é possível visualizar o ficheiro de configuração final usado no *deploy* do servidor.

```
worker_processes  2;

events{
  use epoll;
  worker_connections 2048;
  multi_accept on;
}

http{
  # Phusion Passenger Configurações Gerais
  passenger_root  /home/admin2root/.rvm/gems/ruby-2.1.2/gems/passenger-4.0.46;
  passenger_ruby  /home/admin2root/.rvm/gems/ruby-2.1.2/wrappers/ruby;
  passenger_pool_idle_time          300;
  passenger_max_pool_size           100;
  passenger_max_instances_per_app   0;
  passenger_pre_start               http://setare.com;

  include             mime.types;
  default_type        application/octet-stream;
  sendfile            on;
  tcp_nopush          on;
  tcp_nodelay         on;
  keppalive_timeout   75;
  send_timeout        10;
  access_log          off;
  gzip                on;
  gzip_comp_level     2;
  gzip_min_length     1000;
  gzip_proxied        expired no-cache no-store private auth;
  gzip_types          text/plain application/x-javascript text/xml text/css application/xml;

  # Configuração da Aplicação
  server{
    listen        80 default_server;
    server_name   setare.com;
    charset       utf-8;
    rails_env     production;
    root          /home/admin2root/Setare/public;

    # Configuração da aplicação através do Passenger
    passenger_enabled                 on;
    passenger_spawn_method            smart;
    passenger_min_instances           5;
```

```
    passenger_max_preloader_idle_time    0;
  }
}
```

Código A.2: Configuração do Nginx em Rails

Estes valores significam que o *Nginx* vai servir a aplicação através do *Phusion Passenger*, a porta onde aceitará os pedidos será a 80 e o ambiente do *Ruby on Rails* é o de produção.

E antes de reiniciarmos o *Nginx*, é preciso compilar todos os ficheiros *CSS* e *Javascripts* para que o tempo de resposta da aplicação *Rails* aumente. Este aumento do tempo de resposta é conseguido através da unificação e minimização dos ficheiros de *CSS* e *Javascripts* usando o *uglifier* para ficheiros de *Javascript* e *Sass* para os ficheiros de *CSS*.

```
$ RAILS_ENV=production rake assets:clean assets:precompile
```

Para finalizar, deve-se reiniciar o *Nginx* para que a aplicação *web* desejada se encontre a funcionar.

```
$ sudo service nginx restart
```

Apêndice B

Instalação e configuração do servidor de Java

Durante este capítulo, será apresentado o processo necessário para criar um servidor *web* para executar aplicações desenvolvidas em Java, bem como os passos efectuados para *deploy* de uma aplicação desenvolvida em Java. Para além disso, serão expostos os ficheiros de configuração usados no *Nginx*, no *Apache Tomcat* e na gestão das ligações à base de dados *MySQL* usada pela aplicação. É importante referir que a máquina utilizada para o funcionamento do servidor possui o *Ubuntu Server 14.04 64-bit* como sistema operativo, tem 4 GB de memória *RAM*, 20 GB de disco rígido e um processador de 2.6 GHz.

B.1 Actualizações do Sistema Operativo

O primeiro passo depois da instalação do sistema operativo é efectuar as actualizações disponibilizadas oficialmente. Estas actualizações são fundamentais para corrigir *bugs* e potenciais problemas de segurança. Como se trata de um sistema operativo desenvolvido para correr com o máximo de *up-time* possível, por defeito o *Ubuntu Server* não faz actualizações automáticas. Essa tarefa estará a cargo do administrador do sistema. Para evitar *down-times* ou possíveis problemas no futuro é aconselhável fazer as actualizações antes da instalação das ferramentas para fazer o *deploy* da aplicação. Assim sendo, é necessário actualizar a lista de *packages* disponíveis nos repositórios do *Ubuntu*, utilizando para tal o seguinte comando :

```
$ sudo apt-get update
```

De seguida, deve-se efectuar as actualizações dos *packages* se existir alguma disponível:

```
$ sudo apt-get upgrade
```

B.2 Ferramentas Úteis

Antes de se começar a instalação dos principais componentes, é importante instalar algumas ferramentas úteis que irão facilitar a configuração dos restantes *softwares*. A primeira destas ferramentas chama-se **curl** e permite efectuar o *download* de ficheiros a partir de um *URL*. Para instalar a ferramenta, basta executar o seguinte comando no terminal do servidor:

```
$ sudo apt-get install curl
```

E como será necessário efectuar o *download* de ficheiros em servidores *HTTPS*, é também preciso instalar a seguinte biblioteca :

```
$ sudo apt-get install libcurl4-openssl-dev
```

B.3 Ambiente Java

Para desenvolver uma aplicação *web* em Java, é preciso instalar o *JDK*, que é a base da linguagem de desenvolvimento e o *JRE* que permite desenvolver aplicações para a Internet. E como nos encontramos num ambiente Linux, mais propriamente *Ubuntu*, para instalar o ambiente Java é preciso adicionar um repositório aos já existentes no *Ubuntu*. A necessidade de adicionar um novo repositório ao sistema operativo deve-se sobretudo à falta de manutenção do antigo repositório e da empresa responsável pelo *Ubuntu* não suportar mais esse mesmo repositório.

Então, para conseguirmos instalar todo o *boilerplate* relativo ao Java, basta executar os seguintes comandos na linha de comandos :

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Depois para verificar qual a versão do Java instalado, basta executar :

```
$ java -version
java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)
```

Por fim, é necessário configurar as variáveis de ambiente do Java que irão permitir usar o compilador da linguagem em qualquer directório do sistema. Para tal, basta apenas instalar um pacote que está incluído no novo repositório e que indica qual das versões instaladas do Java o sistema terá que usar.

```
$ sudo apt-get install oracle-java8-set-default
```

B.4 Apache Tomcat

De modo a efectuar um *deploy* de uma aplicação *web* facilmente, é preciso um servidor aplicacional, sendo que o *Apache Tomcat* é a solução mais robusta e fácil no mundo das tecnologias Java. Também seria possível usar outros servidores que possuem mais funcionalidades como *Glassfish*, *Apache TomEE* e *WildFly* mas convém referir que nenhum destes servidores aplicacionais foi escolhido devido ao facto de todos eles implementarem todas as tecnologias referentes ao Java EE e que não será necessário o uso de algumas

dessas tecnologias para a aplicação *web* possa funcionar correctamente.

Para podermos efectuar a instalação do servidor, é preciso efectuar o *download* do ficheiro que contém o necessário para a sua execução directamente do *website* do *Apache Tomcat* e para tal, basta executar o seguinte comando :

```
$ wget http://mirrors.fe.up.pt/pub/apache/tomcat/tomcat-8/v8.0.12/bin/apache-tomcat
  -8.0.12.tar.gz
```

E após o *download* estar efectuado, é preciso descompactar o ficheiro e copiar a pasta criada pela descompactação para outro directório do sistema operativo :

```
$ tar -xvf apache-tomcat-8.0.12.tar.gz
$ sudo mv apache-tomcat-8.0.12 /usr/local
```

Depois para facilitar o uso do executável fornecido pelo *Tomcat* e também criar um serviço no sistema operativo, é preciso criar um ficheiro de *bash* com o nome de *tomcat8* na pasta */etc/init.d* com o seguinte código :

```
#!/bin/bash
export CATALINA_HOME=/usr/local/apache-tomcat-8.0.12
PATH=/sbin:/bin:/usr/sbin:/usr/bin
start() {
 sh $CATALINA_HOME/bin/startup.sh
}
stop() {
 sh $CATALINA_HOME/bin/shutdown.sh
}
case $1 in
  start|stop) $1;;
  restart) stop; start;;
  *) echo "Run as $0 &lt;start|stop|restart&gt;"; exit 1;;
esac
```

Código B.1: Script para iniciar o Apache Tomcat

Depois é necessário dar as permissões ao ficheiro criado para executar bastando executar para tal :

```
$ sudo chmod 755 /etc/init.d/tomcat8
```

De seguida, é preciso adicionar um utilizador ao servidor *Tomcat*, que irá servir como administrador e tratará da gestão do mesmo. Então, para adicionar um utilizador, basta abrir o ficheiro *tomcat-users.xml* que se encontra na pasta do *Tomcat* sob a pasta *conf* e adicionar as seguintes as linhas ao ficheiro :

```
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<user username="setare" password="admin" roles="manager-gui,admin-gui"/>
```

Após o passo anterior, para verificar que o *Tomcat* se encontra correctamente instalado, é preciso iniciar o servidor, bastando para tal executar o seguinte comando :

```
$ sudo /etc/init.d/tomcat8 start
```

E para garantir o servidor é inicializado sempre que o sistema operativo seja inicializado, basta executar :

```
$ sudo update-rc.d tomcat8 defaults
```

E para verificar que de facto o servidor está a funcionar correctamente, basta abrir a seguinte página *web* *http://193.136.19.146:8082/*.

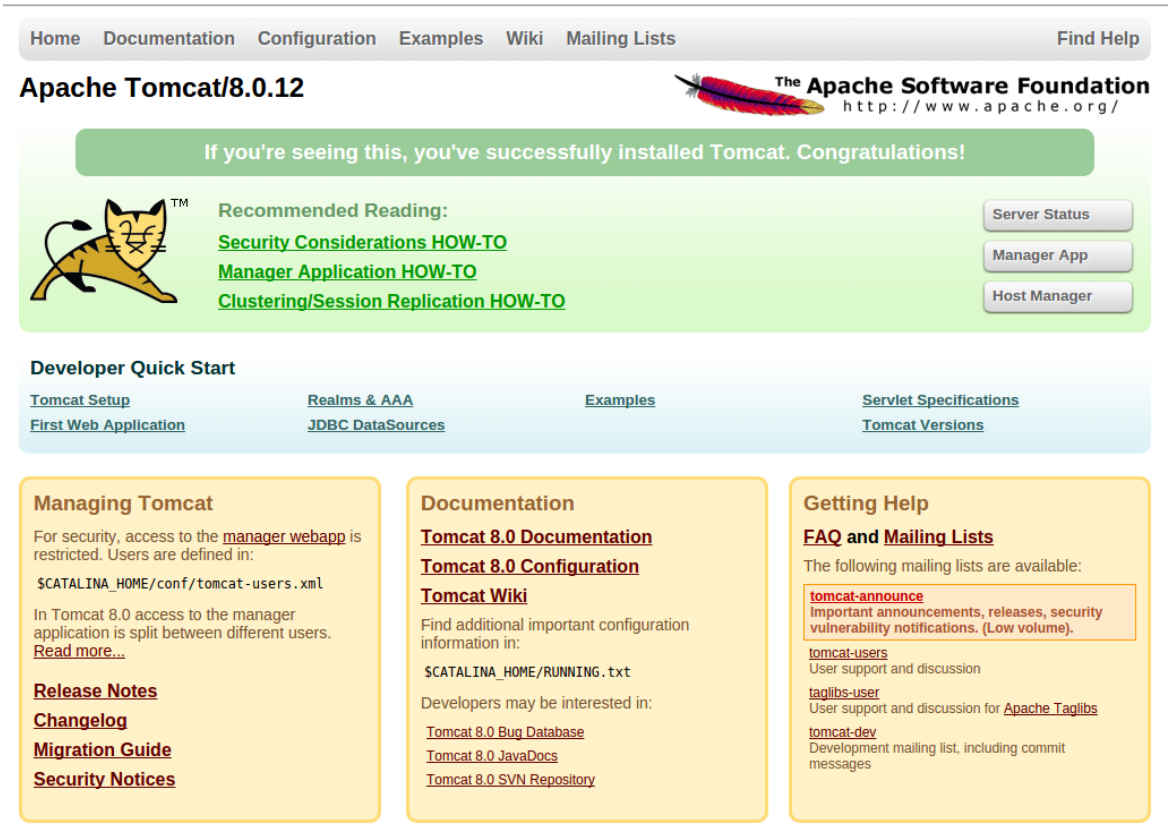


Figura B.1: Página inicial do Apache Tomcat

B.5 MySQL Server

A base de dados escolhida para a aplicação *web* foi o *MySQL Server*. Para instalar a última versão disponível nos repositórios do *Ubuntu*, basta executar na consola o seguinte comando :

```
$ sudo apt-get install mysql-server
```

Durante o processo de instalação vai ser pedido para colocar uma password para o utilizador **root**. No fim da instalação, o servidor *MySQL* estará a correr automaticamente.

B.6 Hibernate e C3P0

Para garantir que a aplicação desenvolvida em Java é capaz de efectuar pedidos à base de dados, foi escolhida o *framework Hibernate*, como já referido na secção 5.2, para implementar toda a panóplia referente ao *Model* da aplicação. Na listagem seguinte, é apresentado a configuração usada no *Hibernate* para efectuar a conexão à base de dados e de algumas variáveis importantes para controlar o seu funcionamento interno. Para além disso, também é apresentado alguns parâmetros de configuração da *connection pool* (*C3P0*) necessários para o *Hibernate* reconhecer e utilizá-la.

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/Setare?autoReconnect=true&
            amp;autoReconnetForPools=true</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">admin</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.max_fetch_depth">3</property>
        <property name="hibernate.query.factory_class">org.hibernate.hql.internal.ast.
            ASTQueryTranslatorFactory</property>
        <property name="hibernate.cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="hibernate.cache.use_query_cache">false</property>
        <property name="hibernate.cache.use_minimal_puts">false</property>
        <property name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.connection.provider_class">org.hibernate.service.jdbc.connections.
            internal.C3P0ConnectionProvider</property>
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">100</property>
        <property name="hibernate.c3p0.idle_test_period">150</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.acquire_increment">1</property>
        <property name="hibernate.c3p0.unreturned_connection_timeout">1</property>
    </session-factory>
</hibernate-configuration>
```

Código B.2: Configuração do Hibernate

Já na listagem seguinte, são apresentados mais variáveis de configuração relativas apenas ao *C3P0* e que servem sobretudo para extrair dela o melhor desempenho possível e garantir a fiabilidade das conexões usadas pela aplicação em Java.

```
/* Basic Pool Configuration */
c3p0.initialPoolSize=5
c3p0.minPoolSize=5
c3p0.maxPoolSize=100
c3p0.acquireIncrement=5
/* Managing Pool Size and Connection Age */
c3p0.maxConnectionAge=0
c3p0.maxIdleTime=18000
```

```
c3p0.maxIdleTimeExcessConnections=150
/* Connection Testing */
/* Control when Connections will be tested */
c3p0.idleConnectionTestPeriod=30
c3p0.testConnectionOnCheckin=true
c3p0.testConnectionOnCheckout=false
/* Control how they will be tested */
c3p0.connectionTesterClassName=com.mysql.jdbc.integration.c3p0.MysqlConnectionTester
c3p0.preferredTestQuery=SELECT 1
/* Statement Pooling */
c3p0.maxStatements=100
c3p0.maxStatementsPerConnection=200
c3p0.statementCacheNumDeferredCloseThreads=1
/* Recovery From Database Outages */
c3p0.acquireRetryAttempts=30
c3p0.acquireRetryDelay=6000
c3p0.breakAfterAcquireFailure=false
/* Unresolved Transaction Handling */
c3p0.autoCommitOnClose=false
c3p0.forceIgnoreUnresolvedTransactions=false
c3p0.numHelperThreads=10
```

Código B.3: Configuração do C3P0

B.7 Nginx

Como já referido na secção A.6 do capítulo anterior, o *Nginx* é usado com o *front-end* para o servidor aplicacional, neste caso o *Apache Tomcat*. A escolha recaiu sob o *Nginx*, pois como já foi explicado anteriormente, é um servidor mais rápido e simples do que o *Apache HTTP Server*.

Por defeito, a instalação seria feita através dos repositórios do *Ubuntu* mas a versão presente nos repositórios é **1.4.6** e como é importante ter a versão mais recente e estável (1.7.2) a funcionar, é necessário efectuar o descarregamento a partir do *website* oficial (*http://nginx.org*).

```
$ wget http://nginx.org/download/nginx-1.7.2.tar.gz
```

Depois do *download* efectuado, é preciso descompactar o ficheiro e compilar os ficheiros contidos na pasta descompactada para que o servidor seja executável e funcione correctamente. Para isso, basta executar os seguintes comandos dentro do directório criado após a descompactação do ficheiro :

```
$ ./configure --prefix=/usr/local/nginx --with-http_ssl_module
$ make
$ sudo make install
```

Após a instalação e configuração do servidor *Nginx*, é importante inicializá-lo e para isso, basta executar o seguinte comando dentro da pasta */usr/local/nginx/sbin*:

```
$ sudo ./nginx
```

E para verificar que de facto o servidor se encontra a funcionar, basta visitar o endereço associado ao servidor num *browser web*, que se pode observar na imagem A.3 presente na secção A.6.

B.8 Deploy

Após a configuração e instalação de todos os componentes necessários para executar uma aplicação *web* desenvolvida em Java, fica a faltar apenas efectuar o *deploy* da aplicação. Para isto ser realizado, é importante que a base de dados a usar seja povoada com dados, como já foi feito no servidor de *Rails*. Como em Java, não existe um mecanismo que através de comandos efectue a povoação da base de dados facilmente, é preciso recorrer à consola do *MySQL* para conseguir criar e carregar a base de dados que será usada pela aplicação. Para aceder à consola do *MySQL*, basta executar o seguinte comando que irá nos pedir a password de acesso ao *MySQL* :

```
$ mysql -u root -p
```

Depois de realizada a autenticação à consola do *MySQL*, é preciso criar uma base de dados e preencher-la com dados. Para isso foi utilizado um *script* em *SQL* que contém as instruções de SQL necessários para que a criação e povoação seja realizada. Para executar esse *script* na consola do *MySQL*, apenas é preciso executar o seguinte comando :

```
$ source SetareCreation.sql
```

Após a conclusão da execução do *script*, o *MySQL* possuirá a base de dados que será usada pela aplicação *web*. O próximo passo será configurar o *Nginx* para servir a aplicação a correr no *Apache Tomcat* para qualquer utilizador ter acesso sem ter que usar a porta em que corre o *Tomcat*.

Então, a seguir é apresentado a configuração usada para servir a aplicação a correr no *Tomcat* através do *Nginx*. Há que referir que a maioria dos parâmetros apresentados foram configurados com base na configuração do servidor *Nginx* para *Rails*, apresentada na secção A.8. Como é óbvio, o bloco com o nome "*server*" é ligeiramente diferente do que foi usado no *Rails*, pois neste caso trata-se de uma aplicação de Java e a configuração necessita ser diferente.

```
worker_processes  2;

events{
  use epoll;
  worker_connections 2048;
  multi_accept on;
}

http{
  include             mime.types;
  default_type        application/octet-stream;
  sendfile            on;
  tcp_nopush          on;
```

```
  tcp_nodelay         on;
  keppalive_timeout   75;
  send_timeout        10;
  access_log          off;
  gzip                on;
  gzip_comp_level     2;
  gzip_min_length     1000;
  gzip_proxied        expired no-cache no-store private auth;
  gzip_types          text/plain application/x-javascript text/xml text/css application/xml;

  # Configuração da Aplicação
  server{
    listen        80 default_server;
    server_name   setareJava;
    charset       utf-8;
    root          /home/admin2root/apache-tomcat-8.0.12/webapps/ROOT;

    location / {
      proxy_set_header X-Forwarded-Host     $host;
      proxy_set_header X-Forwarded-Server   $host;
      proxy_set_header X-Forwarded-For      $proxy_aad_x_forwarded_for;
      proxy_pass                            http://127.0.0.1:8080;
    }
  }
}
```

Código B.4: Configuração do Nginx em Java

Assim, após a configuração do *Nginx* apresentada na listagem anterior, é preciso reiniciar o servidor para ele assumir os novos valores escolhidos. Para isso, executa-se o seguinte comando :

```
$ sudo service nginx restart
```

Após os passos anteriores, apenas falta colocar a aplicação no *Tomcat* para que ela se encontre pronta a ser usada. E para isso acontecer, será efectuar mais dois passos que nos levarão ao *deploy* de uma aplicação de Java.

Uma aplicação para funcionar correctamente no servidor *Tomcat* precisa de ser compilada e transformada num ficheiro com a extensão *WAR* (*Web application ARchive*). Existem muitas formas de transformar um projecto de *Java Web* num ficheiro *WAR* pronto a ser usado pelo servidor e a forma usada para efectuar essa operação foi através do IDE *NetBeans* que foi utilizado durante o desenvolvimento da aplicação. Dentro das funcionalidades que oferece, é possível compilar o projecto de Java de forma a transformá-lo num ficheiro *WAR* através de apenas um clique num botão que o software possui. Após essa compilação, o ficheiro resultante pode ser encontrado na pasta *dist* do projecto e será esse ficheiro que facilitará o *deploy* no servidor *Tomcat*.

Então, antes de acedermos ao gestor de aplicações do *Tomcat*, é preciso renomear o ficheiro *WAR* já referido anteriormente para que seja a aplicação padrão a ser servida pelo servidor. E para termos a

aplicação desenvolvida como padrão, basta mudar o nome do ficheiro *WAR* compilado para *ROOT*.

E após o passo anterior, falta agora efectuar o *upload* desse ficheiro *WAR* para o *Tomcat* e fazer o *deploy* dele. Como o *Tomcat* é um dos servidores mais usados no ambiente Java, ele já possui uma forma para facilitar o processo de *deploy*. Para tal, basta aceder ao Gestor de Aplicações do *Tomcat* e na secção com o nome "*WAR file to deploy*", carregar no botão com o nome "*Escolher ficheiro*" e depois seleccionar o ficheiro *ROOT.WAR* presente no computador utilizado para o desenvolvimento da aplicação. E por fim, carregar no botão com o nome "*deploy*" e esperar que seja feito o *upload* do ficheiro. Após a conclusão do *upload*, o *Tomcat* irá tratar de fazer o *deploy* e também executar a aplicação automaticamente. Na figura B.2, é possível observar a secção usada, a vermelho, para efectuar os passos já descritos.

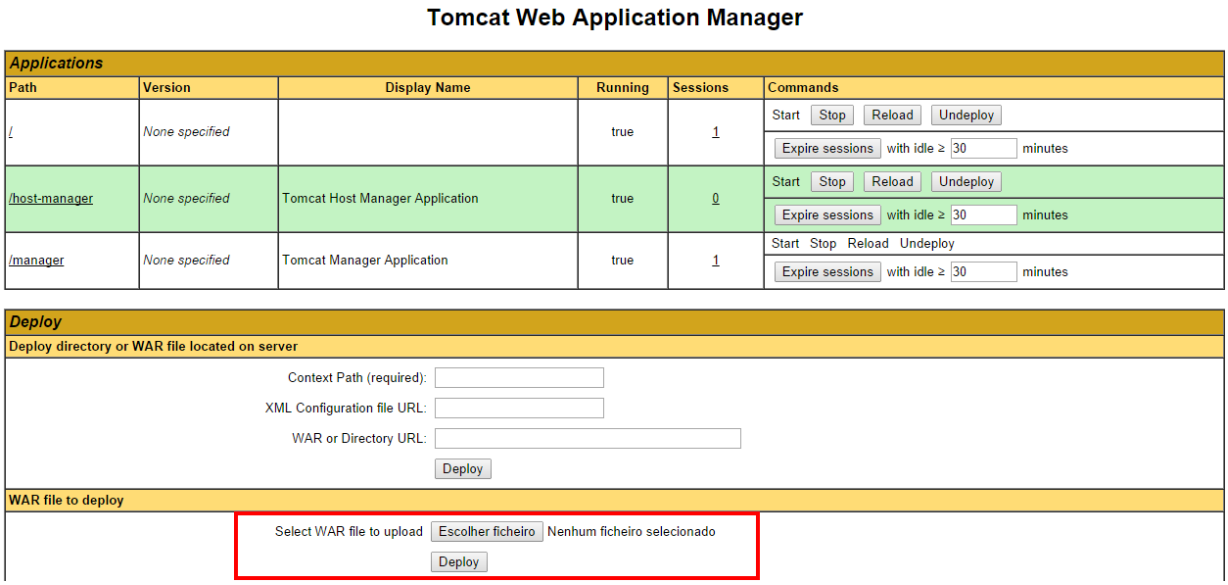


Figura B.2: Gestor de Aplicações Web do Apache Tomcat

Apêndice C

Componentes expostos pelo NewRelic em RoR

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
View	partials/_search_box_results.html.erb Partial	28.3	1.0	59.6
ActiveRecord	Car#find	14.2	1.54	29.9
View	partials/_result_list_grid.html.erb Partial	12.2	7.79	25.7
Beans	CarSearch/cars_list	6.9	1.0	14.6
Beans	CarLocations/initialize	4.9	1.0	10.4
ActiveRecord	CarService#find	4.3	9.21	9.11
ActiveRecord	Subsidiary#find	3.4	5.17	7.07
Beans	CarSearch/other_cars_list	2.7	1.0	5.68
View	partials/_results_list.html.erb Partial	2.5	1.0	5.2
View	partials/_header.html.erb Partial	2.4	1.0	5.05
Beans	CarSearch/remaining_subsidiaries	2.3	1.0	4.8
Controller	CarController#search_results	2.0	1.0	4.19
GC	GC Execution	1.7	0.116	3.51
Middleware	NewRelic::Rack::BrowserMonitoring#call	1.6	1.0	3.26
View	layouts/application Template	1.5	1.0	3.05
ActiveRecord	City#find	1.3	2.0	2.69

Figura C.1: Exemplo 1 dos componentes na plataforma RoR

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
View	partials/_form_after_search.html.erb Partial	26.3	1.0	26.4
View	partials/_taxi.html.erb Partial	18.8	7.38	18.9
Controller	TaxisController#search_results	7.2	1.0	7.19
Beans	TaxiLocations/initialize	5.9	1.0	5.9
View	partials/_list_taxis.html.erb Partial	5.6	1.0	5.58
ActiveRecord	DropOffLocation#find	3.4	2.0	3.39
View	layouts/application Template	3.0	1.0	3.04
ActiveRecord	PickUpLocation#find	2.6	2.0	2.57
Middleware	ActionDispatch::Routing::RouteSet#call	2.5	1.0	2.52
Middleware	NewRelic::Rack::BrowserMonitoring#call	2.4	1.0	2.44
ActiveRecord	User#find	2.0	1.0	2.02
GC	GC Execution	2.0	0.0705	1.98
View	partials/_header.html.erb Partial	1.8	1.0	1.8
BussinessLogic	TaxiSearch/create_taxis_response	1.8	1.0	1.76
BussinessLogic	TaxiSearch/search	1.8	1.0	1.83
ActiveRecord	Country#find	1.5	1.0	1.54

Figura C.2: Exemplo 2 dos componentes na plataforma RoR

Apêndice D

Componentes expostos pelo NewRelic em Java

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
ORM	model.Country.list()	24.0	1.0	125
Database	car - SELECT	23.8	1.61	124
Jsp	WEB-INF/jspf/partials/cars/search_005fbox_005fresults.jsp	14.0	1.0	72.8
Database	RESULTSET	11.0	51400	57.2
Database	country - SELECT	8.5	2.0	44
Hibernate	org.hibernate.internal.SessionImpl.flush()	8.5	18.3	44.2
Java	other	4.6	89.9	23.9
StrutsAction	search-results.execute()	3.4	1.0	17.7
Jsp	WEB-INF/jspf/partials/cars/result_005flist_005fgrid.jsp	1.1	4.6	5.82
Database	subsidiary - SELECT	0.4	7.0	1.95
Database	city - SELECT	0.3	4.0	1.49
Database	car_service - SELECT	0.2	3.67	1.28
Servlet	org.apache.jasper.servlet.JspServlet.service()	0.1	0.336	0.271
WebTransaction	/search-results	0.1	1.0	0.271
Jsp	WEB-INF/view/cars/search_002dresults.jsp	0.1	0.0186	0.329
Jsp	WEB-INF/jspf/partials/cars/results_005flist.jsp	0.0	0.0041	0.0556
Filter	org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter.doFilter()	0.0	0.0052	0.0671

Figura D.1: Exemplo 1 dos componentes na plataforma Java

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Jsp	WEB-INF/view/taxis/search_002dtaxis.jsp	52.1	1.0	41.9
Database	country - SELECT	12.5	1.0	10
Jsp	WEB-INF/jspf/partials/taxis/list_005ftaxis.jsp	10.7	1.0	8.6
StrutsAction	search-taxis.execute()	8.1	1.0	6.49
Java	other	6.3	27.2	5.04
ORM	model.Country.list()	3.7	0.968	2.97
Database	RESULTSET	1.8	1490	1.44
Servlet	org.apache.jasper.servlet.JspServlet.service()	1.7	1.12	1.33
Database	pick_up_location - SELECT	0.7	2.0	0.595
Database	drop_off_location - SELECT	0.7	2.0	0.552
Database	taxi - SELECT	0.6	1.0	0.471
Database	city - SELECT	0.6	1.0	0.492
Java	com.mchange.v2.c3p0.PoolBackedDataSource.getConnection()	0.2	0.26	0.144
Hibernate	org.hibernate.internal.SessionImpl.flush()	0.1	0.161	0.0499
WebTransaction	/search-taxis	0.1	1.0	0.101
ORM	model.PickUpLocation.list()	0.1	0.0547	0.098
ORM	model.City.list()	0.0	0.0077	0.017
ORM	model.Taxi.list()	0.0	0.0044	0.0098
Filter	org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter.doFilter()	0.0	0.0131	0.0297
ORM	model.DropOffLocation.list()	0.0	0.0055	0.0106

Figura D.2: Exemplo 2 dos componentes na plataforma Java

Apêndice E

Resultados do Web Profiling

E.1 View Profiling

Acções	Average Page Creation Time	
	Java	RoR
Página Inicial	2,700 ms	4,563 ms
Página Login	6,429 ms	7,603 ms
Login	3,290 ms	0 ms
Logout	3,180 ms	0 ms
Página do Registo	8,666 ms	10,871 ms
Registo	4,030 ms	0 ms
Perfil do Utilizador	8,640 ms	12,302 ms
Update do Perfil	7,700 ms	0 ms
Página das Reservas	16,260 ms	161,814 ms
Página dos Táxis	19,500 ms	12,683 ms
Pesq. de Cidades	0 ms	0 ms
Cidades de Partida	0 ms	0 ms
Cidade de Chegada	0 ms	0 ms
Pesq. de Táxis	48,080 ms	35,533 ms
Selecção de Táxis	49,020 ms	27,690 ms
Reserva de Táxis	2,860 ms	13,174 ms
Página dos Carros	58,800 ms	32,046 ms

Continua na próxima página

Cidades de Partida	0 ms	0 ms
Subsidiárias de Partida	0 ms	0 ms
Pesq. de Agências	0 ms	0 ms
Cidades de Chegada	0 ms	0 ms
Subsidiárias de Chegada	0 ms	0 ms
Pesq. de Carros	72,130 ms	23,545 ms
Escolha dos Extras	64,345 ms	55,560 ms
Pagamento do Carro	16,812 ms	24,511 ms
Reserva do Carro	2,740 ms	9,684 ms

Tabela E.1: Resultados na Categoria de View Profiling para 50 Utilizadores

Acções	Average Page Creation Time	
	Java	RoR
Página Inicial	2,420 ms	4,843 ms
Página Login	5,542 ms	10,044 ms
Login	3,090 ms	0 ms
Logout	2,530 ms	0 ms
Página do Registo	7,839 ms	12,250 ms
Registo	3,735 ms	0 ms
Perfil do Utilizador	7,860 ms	13,846 ms
Update do Perfil	8,340 ms	0 ms
Página das Reservas	27,190 ms	451,221 ms
Página dos Táxis	19,400 ms	14,387 ms
Pesq. de Cidades	0 ms	0 ms
Cidades de Partida	0 ms	0 ms
Cidade de Chegada	0 ms	0 ms
Pesq. de Táxis	47,560 ms	35,564 ms
Selecção de Táxis	49,510 ms	30,169 ms
Reserva de Táxis	3,410 ms	14,236 ms
Página dos Carros	57,700 ms	35,971 ms
Cidades de Partida	0 ms	0 ms

Continua na próxima página

Subsidiárias de Partida	0 ms	0 ms
Pesq. de Agências	0 ms	0 ms
Cidades de Chegada	0 ms	0 ms
Subsidiárias de Chegada	0 ms	0 ms
Pesq. de Carros	72,970 ms	60,322 ms
Escolha dos Extras	67,924 ms	60,129 ms
Pagamento do Carro	16,311 ms	25,068 ms
Reserva do Carro	3,050 ms	9,986 ms

Tabela E.2: Resultados na Categoria de View Profiling para 100 Utilizadores

Acções	Average Page Creation Time	
	Java	RoR
Página Inicial	2,360 ms	6,602 ms
Página Login	5,274 ms	11,310 ms
Login	3,330 ms	0 ms
Logout	2,540 ms	0 ms
Página do Registo	7,917 ms	16,590 ms
Registo	3,878 ms	0 ms
Perfil do Utilizador	7,970 ms	20,172 ms
Update do Perfil	8,260 ms	0 ms
Página das Reservas	46,820 ms	931,115 ms
Página dos Táxis	21,100 ms	20,060 ms
Pesq. de Cidades	0 ms	0 ms
Cidades de Partida	0 ms	0 ms
Cidade de Chegada	0 ms	0 ms
Pesq. de Táxis	51,230 ms	53,893 ms
Selecção de Táxis	56,110 ms	43,121 ms
Reserva de Táxis	3,170 ms	16,768 ms
Página dos Carros	65,000 ms	52,317 ms
Cidades de Partida	0 ms	0 ms
Subsidiárias de Partida	0 ms	0 ms

Continua na próxima página

Pesq. de Agências	0 ms	0 ms
Cidades de Chegada	0 ms	0 ms
Subsidiárias de Chegada	0 ms	0 ms
Pesq. de Carros	78,562 ms	92,898 ms
Escolha dos Extras	70,444 ms	82,041 ms
Pagamento do Carro	17,207 ms	35,864 ms
Reserva do Carro	3,900 ms	14,736 ms

Tabela E.3: Resultados na Categoria de View Profiling para 150 Utilizadores

Acções	Average Page Creation Time	
	Java	RoR
Página Inicial	2,570 ms	62,750 ms
Página Login	5,555 ms	130,140 ms
Login	3,360 ms	0 ms
Logout	2,680 ms	0 ms
Página do Registo	10,033 ms	147,300 ms
Registo	4,260 ms	0 ms
Perfil do Utilizador	9,110 ms	197,600 ms
Update do Perfil	8,680 ms	0,000 ms
Página das Reservas	60,380 ms	11044,110 ms
Página dos Táxis	21,600 ms	241,730 ms
Pesq. de Cidades	0 ms	0 ms
Cidades de Partida	0 ms	0 ms
Cidade de Chegada	0 ms	0 ms
Pesq. de Táxis	55,920 ms	1634,090 ms
Selecção de Táxis	57,780 ms	484,660 ms
Reserva de Táxis	3,780 ms	202,127 ms
Página dos Carros	60,100 ms	460,290 ms
Cidades de Partida	0 ms	0 ms
Subsidiárias de Partida	0 ms	0 ms
Pesq. de Agências	0 ms	0 ms

Continua na próxima página

Cidades de Chegada	0 ms	0 ms
Subsidiárias de Chegada	0 ms	0 ms
Pesq. de Carros	83,669 ms	934,360 ms
Escolha dos Extras	67,845 ms	1224,230 ms
Pagamento do Carro	19,467 ms	521,940 ms
Reserva do Carro	3,340 ms	194,182 ms

Tabela E.4: Resultados na Categoria de View Profiling para 200 Utilizadores

E.2 Controller Profiling

Acções	Average Response Time		Action Average Response Time	
	Java	RoR	Java	RoR
Página Inicial	0,970 ms	2,780 ms	3,740 ms	13,600 ms
Página Login	2,538 ms	4,090 ms	9,160 ms	18,000 ms
Login	3,239 ms	8,130 ms	8,500 ms	15,800 ms
Logout	1,896 ms	13,800 ms	5,180 ms	21,400 ms
Página do Registo	2,377 ms	3,760 ms	11,300 ms	20,800 ms
Registo	166,293 ms	14,200 ms	177,000 ms	137,000 ms
Perfil do Utilizador	2,583 ms	4,870 ms	11,400 ms	24,000 ms
Update do Perfil	275,685 ms	11,500 ms	290,000 ms	128,000 ms
Página das Reservas	6,178 ms	6,150 ms	32,400 ms	192,000 ms
Página dos Táxis	2,993 ms	5,380 ms	38,200 ms	25,800 ms
Pesq. de Cidades	2,668 ms	7,300 ms	4,460 ms	14,700 ms
Cidades de Partida	2,275 ms	13,900 ms	5,210 ms	21,300 ms
Cidade de Chegada	2,516 ms	9,490 ms	4,570 ms	17,400 ms
Pesq. de Táxis	7,962 ms	13,710 ms	77,000 ms	61,200 ms
Selecção de Táxis	7,414 ms	8,580 ms	76,800 ms	46,200 ms
Reserva de Táxis	298,082 ms	15,520 ms	312,000 ms	93,800 ms
Página dos Carros	0,592 ms	3,120 ms	308,000 ms	42,700 ms
Cidades de Partida	2,786 ms	10,100 ms	7,780 ms	17,200 ms
Subsidiárias de Partida	2,447 ms	3,920 ms	4,040 ms	10,900 ms

Continua na próxima página

Pesq. de Agências	2,461 ms	4,080 ms	4,280 ms	10,900 ms
Cidades de Chegada	2,489 ms	5,510 ms	4,960 ms	12,800 ms
Subsidiárias de Chegada	2,844 ms	4,320 ms	4,550 ms	11,300 ms
Pesq. de Carros	15,558 ms	26,422 ms	467,000 ms	130,000 ms
Escolha dos Extras	10,802 ms	9,370 ms	327,000 ms	76,600 ms
Pagamento do Carro	4,782 ms	5,224 ms	24,200 ms	40,500 ms
Reserva do Carro	240,379 ms	10,324 ms	258,000 ms	92,100 ms

Tabela E.5: Resultados na Categoria de Controller Profiling para 50 Utilizadores

Acções	Average Response Time		Action Average Response Time	
	Java	RoR	Java	RoR
Página Inicial	0,642 ms	2,820 ms	3,130 ms	14,300 ms
Página Login	2,501 ms	4,120 ms	8,220 ms	20,500 ms
Login	2,998 ms	7,820 ms	8,290 ms	15,500 ms
Logout	1,586 ms	14,300 ms	4,190 ms	22,600 ms
Página do Registo	1,892 ms	3,950 ms	9,920 ms	22,400 ms
Registo	49,010 ms	14,500 ms	55,900 ms	101,000 ms
Perfil do Utilizador	2,223 ms	4,750 ms	10,300 ms	25,800 ms
Update do Perfil	79,412 ms	10,800 ms	93,400 ms	95,300 ms
Página das Reservas	5,374 ms	6,030 ms	46,600 ms	533,000 ms
Página dos Táxis	2,691 ms	5,310 ms	38,100 ms	28,100 ms
Pesq. de Cidades	2,398 ms	7,420 ms	4,700 ms	15,200 ms
Cidades de Partida	2,861 ms	13,600 ms	6,150 ms	22,300 ms
Cidade de Chegada	2,345 ms	10,600 ms	4,530 ms	19,400 ms
Pesq. de Táxis	8,020 ms	14,147 ms	77,000 ms	69,100 ms
Selecção de Táxis	7,847 ms	8,470 ms	77,800 ms	50,200 ms
Reserva de Táxis	94,329 ms	19,864 ms	104,000 ms	90,400 ms
Página dos Carros	1,212 ms	3,270 ms	321,000 ms	49,500 ms
Cidades de Partida	2,325 ms	10,300 ms	6,120 ms	17,100 ms
Subsidiárias de Partida	2,417 ms	4,240 ms	4,080 ms	11,200 ms
Pesq. de Agências	2,336 ms	3,900 ms	4,260 ms	11,100 ms

Continua na próxima página

Cidades de Chegada	2,458 ms	5,770 ms	5,400 ms	13,200 ms
Subsidiárias de Chegada	2,274 ms	4,500 ms	4,150 ms	11,700 ms
Pesq. de Carros	15,500 ms	25,190 ms	475,000 ms	134,000 ms
Escolha dos Extras	12,156 ms	11,720 ms	347,000 ms	87,600 ms
Pagamento do Carro	3,884 ms	5,436 ms	22,800 ms	43,500 ms
Reserva do Carro	80,574 ms	9,136 ms	91,500 ms	70,000 ms

Tabela E.6: Resultados na Categoria de Controller Profiling para 100 Utilizadores

Acções	Average Response Time		Action Average Response Time	
	Java	RoR	Java	RoR
Página Inicial	0,575 ms	3,550 ms	2,990 ms	18,300 ms
Página Login	2,246 ms	5,270 ms	7,700 ms	23,900 ms
Login	2,986 ms	10,700 ms	8,430 ms	21,000 ms
Logout	1,557 ms	20,300 ms	4,170 ms	32,000 ms
Página do Registo	1,747 ms	5,190 ms	9,830 ms	29,300 ms
Registo	53,414 ms	17,300 ms	60,300 ms	109,000 ms
Perfil do Utilizador	2,263 ms	6,560 ms	10,400 ms	36,700 ms
Update do Perfil	86,146 ms	12,700 ms	99,500 ms	105,000 ms
Página das Reservas	5,509 ms	7,290 ms	73,300 ms	1080,000 ms
Página dos Táxis	2,938 ms	7,630 ms	40,600 ms	38,900 ms
Pesq. de Cidades	2,393 ms	8,670 ms	4,480 ms	19,800 ms
Cidades de Partida	2,602 ms	17,300 ms	5,950 ms	26,900 ms
Cidade de Chegada	2,437 ms	12,800 ms	4,710 ms	24,800 ms
Pesq. de Táxis	7,899 ms	18,536 ms	81,300 ms	94,800 ms
Selecção de Táxis	7,789 ms	12,270 ms	84,800 ms	74,400 ms
Reserva de Táxis	110,065 ms	22,429 ms	120,000 ms	101,000 ms
Página dos Carros	1,444 ms	3,970 ms	348,000 ms	67,600 ms
Cidades de Partida	2,473 ms	12,100 ms	6,130 ms	21,000 ms
Subsidiárias de Partida	2,416 ms	4,760 ms	4,130 ms	12,900 ms
Pesq. de Agências	2,571 ms	4,840 ms	4,710 ms	13,300 ms
Cidades de Chegada	2,505 ms	7,320 ms	5,160 ms	16,100 ms

Continua na próxima página

Subsidiárias de Chegada	2,483 ms	5,130 ms	4,280 ms	14,600 ms
Pesq. de Carros	19,008 ms	38,950 ms	522,000 ms	199,000 ms
Escolha dos Extras	12,669 ms	13,240 ms	365,000 ms	117,000 ms
Pagamento do Carro	3,884 ms	8,748 ms	23,800 ms	65,600 ms
Reserva do Carro	90,687 ms	8,980 ms	106,000 ms	84,000 ms

Tabela E.7: Resultados na Categoria de Controller Profiling para 150 Utilizadores

Acções	Average Response Time		Action Average Response Time	
	Java	RoR	Java	RoR
Página Inicial	0,572 ms	41,400 ms	3,200 ms	196,000 ms
Página Login	2,275 ms	59,500 ms	8,020 ms	242,000 ms
Login	3,089 ms	131,000 ms	8,710 ms	222,000 ms
Logout	1,488 ms	524,000 ms	4,240 ms	629,000 ms
Página do Registo	1,868 ms	50,100 ms	12,100 ms	258,000 ms
Registo	58,143 ms	209,000 ms	66,700 ms	1830,000 ms
Perfil do Utilizador	2,130 ms	55,700 ms	11,500 ms	401,000 ms
Update do Perfil	77,476 ms	170,000 ms	91,800 ms	1400,000 ms
Página das Reservas	5,048 ms	113,000 ms	92,500 ms	12800,00 ms
Página dos Táxis	3,039 ms	73,200 ms	41,500 ms	428,000 ms
Pesq. de Cidades	2,720 ms	389,000 ms	5,170 ms	791,000 ms
Cidades de Partida	2,813 ms	196,000 ms	6,290 ms	316,000 ms
Cidade de Chegada	2,817 ms	173,000 ms	6,500 ms	336,000 ms
Pesq. de Táxis	9,635 ms	512,580 ms	89,900 ms	2810,000 ms
Selecção de Táxis	8,596 ms	174,700 ms	88,000 ms	872,000 ms
Reserva de Táxis	102,015 ms	419,740 ms	112,000 ms	1540,000 ms
Página dos Carros	0,821 ms	52,100 ms	347,000 ms	669,000 ms
Cidades de Partida	2,722 ms	183,000 ms	7,170 ms	294,000 ms
Subsidiárias de Partida	2,697 ms	43,200 ms	4,670 ms	126,000 ms
Pesq. de Agências	2,788 ms	54,000 ms	5,380 ms	115,000 ms
Cidades de Chegada	2,670 ms	106,000 ms	5,560 ms	207,000 ms
Subsidiárias de Chegada	2,863 ms	72,800 ms	4,930 ms	153,000 ms

Continua na próxima página

Pesq. de Carros	21,300 ms	478,600 ms	566,000 ms	1900,000 ms
Escolha dos Extras	14,544 ms	188,200 ms	393,000 ms	1790,000 ms
Pagamento do Carro	4,171 ms	161,800 ms	26,600 ms	911,000 ms
Reserva do Carro	78,809 ms	307,500 ms	90,900 ms	1400,000 ms

Tabela E.8: Resultados na Categoria de Controller Profiling para 200 Utilizadores

E.3 Database Profiling

Acções	Average Time in DB Operations	
	Java	RoR
Página Inicial	0 ms	0,249 ms
Página Login	0 ms	0,001 ms
Login	1,656 ms	0,701 ms
Logout	0 ms	1,591 ms
Página do Registo	0 ms	0,001 ms
Registo	3,308 ms	113,988 ms
Perfil do Utilizador	0 ms	0,734 ms
Update do Perfil	3,109 ms	108,870 ms
Página das Reservas	8,713 ms	17,728 ms
Página dos Táxis	14,528 ms	1,169 ms
Pesq. de Cidades	1,641 ms	1,230 ms
Cidades de Partida	2,765 ms	1,288 ms
Cidade de Chegada	1,879 ms	1,986 ms
Pesq. de Táxis	16,514 ms	3,231 ms
Selecção de Táxis	16,283 ms	2,580 ms
Reserva de Táxis	6,519 ms	56,513 ms
Página dos Carros	242,300 ms	0,998 ms
Cidades de Partida	4,793 ms	0,693 ms
Subsidiárias de Partida	1,445 ms	0,832 ms
Pesq. de Agências	1,672 ms	0,677 ms
Cidades de Chegada	2,310 ms	1,020 ms

Continua na próxima página

Subsidiárias de Chegada	1,555 ms	0,843 ms
Pesq. de Carros	358,362 ms	37,268 ms
Escolha dos Extras	238,665 ms	4,271 ms
Pagamento do Carro	1,789 ms	3,450 ms
Reserva do Carro	9,683 ms	63,286 ms

Tabela E.9: Resultados na Categoria de Database Profiling para 50 Utilizadores

Acções	Average Time in DB Operations	
	Java	RoR
Página Inicial	0 ms	0,345 ms
Página Login	0 ms	0,001 ms
Login	1,913 ms	1,030 ms
Logout	0 ms	2,282 ms
Página do Registo	0 ms	0,018 ms
Registo	1,215 ms	77,416 ms
Perfil do Utilizador	0 ms	1,080 ms
Update do Perfil	2,601 ms	76,657 ms
Página das Reservas	12,353 ms	69,855 ms
Página dos Táxis	14,648 ms	1,972 ms
Pesq. de Cidades	2,145 ms	1,775 ms
Cidades de Partida	3,118 ms	2,544 ms
Cidade de Chegada	1,995 ms	2,657 ms
Pesq. de Táxis	16,877 ms	6,204 ms
Selecção de Táxis	16,489 ms	3,859 ms
Reserva de Táxis	3,496 ms	46,562 ms
Página dos Carros	255,800 ms	3,503 ms
Cidades de Partida	3,635 ms	0,889 ms
Subsidiárias de Partida	1,507 ms	1,020 ms
Pesq. de Agências	1,773 ms	1,061 ms
Cidades de Chegada	2,757 ms	1,303 ms
Subsidiárias de Chegada	1,737 ms	1,040 ms

Continua na próxima página

Pesq. de Carros	364,769 ms	38,750 ms
Escolha dos Extras	252,797 ms	6,399 ms
Pagamento do Carro	1,880 ms	4,639 ms
Reserva do Carro	4,486 ms	42,876 ms

Tabela E.10: Resultados na Categoria de Database Profiling para 100 Utilizadores

Acções	Average Time in DB Operations	
	Java	RoR
Página Inicial	0 ms	0,584 ms
Página Login	0 ms	0,004 ms
Login	2,097 ms	1,769 ms
Logout	0 ms	4,376 ms
Página do Registo	0 ms	0,007 ms
Registo	0,990 ms	83,356 ms
Perfil do Utilizador	0 ms	1,640 ms
Update do Perfil	1,904 ms	84,016 ms
Página das Reservas	18,117 ms	134,827 ms
Página dos Táxis	15,058 ms	3,080 ms
Pesq. de Cidades	1,938 ms	3,008 ms
Cidades de Partida	3,193 ms	2,808 ms
Cidade de Chegada	2,102 ms	5,070 ms
Pesq. de Táxis	16,936 ms	11,154 ms
Selecção de Táxis	16,370 ms	8,790 ms
Reserva de Táxis	3,059 ms	51,948 ms
Página dos Carros	275,600 ms	3,227 ms
Cidades de Partida	3,508 ms	1,659 ms
Subsidiárias de Partida	1,562 ms	1,701 ms
Pesq. de Agências	1,977 ms	1,827 ms
Cidades de Chegada	2,472 ms	2,032 ms
Subsidiárias de Chegada	1,647 ms	1,728 ms
Pesq. de Carros	400,729 ms	56,001 ms

Continua na próxima página

Escolha dos Extras	267,366 ms	12,763 ms
Pagamento do Carro	1,942 ms	9,657 ms
Reserva do Carro	7,432 ms	48,611 ms

Tabela E.11: Resultados na Categoria de Database Profiling para 150 Utilizadores

Acções	Average Time in DB Operations	
	Java	RoR
Página Inicial	0,000 ms	18,505 ms
Página Login	0,000 ms	0,103 ms
Login	1,919 ms	16,587 ms
Logout	0,000 ms	54,180 ms
Página do Registo	0,000 ms	0,335 ms
Registo	1,898 ms	1485,980 ms
Perfil do Utilizador	0,000 ms	69,355 ms
Update do Perfil	2,251 ms	1191,231 ms
Página das Reservas	23,996 ms	1503,210 ms
Página dos Táxis	11,998 ms	36,890 ms
Pesq. de Cidades	2,269 ms	64,954 ms
Cidades de Partida	3,282 ms	68,530 ms
Cidade de Chegada	3,474 ms	82,400 ms
Pesq. de Táxis	18,389 ms	113,361 ms
Selecção de Táxis	16,484 ms	117,940 ms
Reserva de Táxis	3,424 ms	884,170 ms
Página dos Carros	279,900 ms	75,726 ms
Cidades de Partida	4,252 ms	13,200 ms
Subsidiárias de Partida	1,804 ms	15,450 ms
Pesq. de Agências	2,401 ms	15,357 ms
Cidades de Chegada	2,690 ms	29,880 ms
Subsidiárias de Chegada	1,905 ms	18,610 ms
Pesq. de Carros	433,958 ms	342,030 ms
Escolha dos Extras	294,442 ms	267,700 ms

Continua na próxima página

Pagamento do Carro	2,110 ms	100,220 ms
Reserva do Carro	5,440 ms	828,370 ms

Tabela E.12: Resultado na Categoria de Database Profiling para 200 Utilizadores