Rodrigo de Almeida Graças

Measuring Performance in Network-Intensive Web Applications

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Carlos Antunes Leitão,
Assistant Professor, NOVA University of Lisbon

Co-advisers: João Ricardo Viegas da Costa Seco,
Assistant Professor, NOVA University of Lisbon

Nuno Alexandre Neves Cruz,
Front-End Engineer, Feedzai

Luís Filipe Alves Cardoso,
Front-End Engineer, Feedzai

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2018

Abstract

Web applications performance is a key aspect nowadays. In order to be successful, and productive, users expect to have a system highly responsive. Moreover, network-Intensive web applications performance is sensitive to many factors.

Currently, companies are building their products using a well established web design paradigm, known as Single-Page Applications. These applications introduce a whole new set of challenges and reliably measuring the performance of these applications is not a simple task. Both the application code and data are loaded asynchronously, thus making the classic load time metrics unrepresentative.

It is important for developers to understand the effects that impact the performance of this kind of applications. In this work, we address the problem of providing feedback to developers regarding modifications performed on these applications. We will focus on using metrics related with network time. We instantiate in two typical network-intensive applications developed by Feedzai.

The goal of this thesis is to develop a framework and methodology to reliably measure performance of single-page applications, describing the necessary infrastructure to track the evolution of loading time metrics, detect regressions, and measure improvements objectively.

Keywords: network-intensive, web applications, single-page web applications, web performance

Resumo

O desempenho das aplicações Web é um aspecto fundamental hoje em dia. Os utilizadores esperam ter um sistema altamente responsivo. O desempenho das aplicações de uso intensivo da rede é sensível a muitos fatores.

Atualmente, as empresas estão a construir os seus produtos usando um paradigma emergente de design para a internet, conhecido como aplicações de página-única. Estas aplicações apresentam um novo conjunto de desafios e medir de forma confiável o desempenho destas aplicações não é uma tarefa simples. Tanto o código da aplicação como os dados são carregados de forma assíncrona, tornando as métricas de tempo de carga clássicas não representativas.

Consequentemente, é importante para os programadores entenderem os efeitos que afetam o desempenho destas aplicações. Neste trabalho, abordamos o problema de fornecer respostas aos programadores sobre modificações realizadas nestas aplicações. Vamonos focar no uso de métricas relacionadas com o tempo na rede. Iremos contextualizar este cenário em duas aplicações típicas de uso intensivo de rede desenvolvidas pela Feedzai.

O objetivo desta tese é desenvolver uma estrutura e metodologia para medir de forma confiável o desempenho de aplicações de página-única, descrevendo a infra-estrutura necessária para identificar a evolução de métricas de tempo de carregamento, detectar regressões e medir melhorias objetivamente.

Palavras-chave: rede-intensiva, aplicações internet página-única, desempenho

Contents

List of Figures **ix**

Glossary **xi**

Acronyms **xiii**

1 Introduction **1**

1.1 Single Page Applications 2

1.2 Differences from Classical Web Applications 3

1.3 Challenges in measuring the performance of SPAs 5

1.4 Problem statement 6

1.5 Document structure 6

2 Thesis context **7**

2.1 Feedzai overview 7

2.2 Pulse 8

2.3 Case Manager 9

2.4 Summary 11

3 State of the art **13**

3.1 Web Applications architecture 13

3.2 SPAs frameworks 13

3.3 Performance measurements 14

3.3.1 Performance measurements for web applications 14

3.3.2 SPAs performance measurements 15

3.4 Discussion 19

4 Approach **21**

4.1 Framework proposed 21

4.2 Framework requirements 21

Bibliography **23**

Webography **25**

List of Figures

1.1 "Pages" of a SPA - *Views*, adapted from [EAS15] 3
1.2 Classical Lifecycle vs. SPA Lifecycle 4

2.1 *Pulse*'s components overview 9
2.2 Communication between *Case Manager*'s components 10

3.1 Waterfall overview from resources in a SPA 16
3.2 First page load filmstrip of a SPA 16
3.3 New Relic Browser's UI 18

4.1 Architecture for the solution proposed 22

Glossary

AJAX	Method that allows to exchange data from a server asynchronously, allowing to update a web application without refreshing the page.
div	Tag that defines a section in an HTML document.

Acronyms

AI	Artificial Intelligence.
API	Application Programming Interface.
CLI	Command-Line Interface.
CSS	Cascading Style Sheets.
DOM	Document Object Model.
HTML	Hyper Text Markup Language.
HTTP	Hypertext Transfer Protocol.
JS	JavaScript.
JSON	JavaScript Object Notation.
REST	Representational State Transfer.
SPA	Single Page Application.
UI	User Interface.
URL	Uniform Resource Locator.
UX	User Experience.
XML	Extensible Markup Language.

Chapter 1

Introduction

Nowadays, everything is accessible through the web. We live in an online world and it is important for companies to maintain their online presence. Web applications have become an essential component for companies to develop their products, helping them to target users and achieve business objectives faster. Consequently, companies want to retain their customers and engage them with new strategies and technologies. Web applications play an important role to meet customer expectations, enhance user interaction and provide better engagement. Being able to access these applications from nearly everywhere, it captures a larger audience faster and so, it creates the need for companies to build their products as web applications.

In today's world, a significant number of web applications deployed on cloud infrastructures are network-intensive. These applications tend to drive more traffic over the network, with resources such as Cascading Style Sheets (CSS), JavaScript (JS) and Hyper Text Markup Language (HTML) files. It is critical for applications to access data over the network and to optimize overall performance. Due to web applications growing importance, companies rely their products on these applications. Performance has to be consistently measured and monitored, in order to avoid critical consequences and lost customers.

As web development practices evolve, a modern web design paradigm is emerging, known as Single Page Applications (SPAs). Nowadays, companies such as Facebook and Twitter built their web applications using this new approach, because it can deliver more dynamic content and richer user experiences. These applications are not usually network-friendly since they heavily depend on JS, CSS, HTML, and AJAX requests but, in comparison to traditional web applications, these applications are faster, since they can redraw, asynchronously, the User Interface (UI) without requiring a full roundtrip to the server in order to retrieve new content. By breaking the traditional web design approach, these applications introduce a whole new set of challenges, when it comes to measure performance.

In this work, we present a solution that aims to overcome these challenges. By providing network-related metrics and measure overall performance on SPAs, we intend to help *Feedzai* and their developers, to assess their applications performance and the sources of bottlenecks as to allow them to easily meet their business demands.

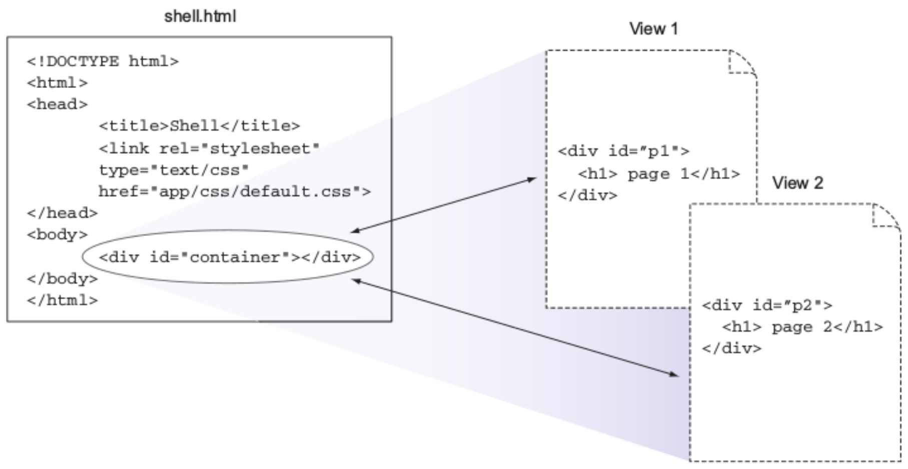
1.1 Single Page Applications

SPAs are web applications that run in a single static HTML page and whose content is dynamically updated in response to user interactions. All the application code - HTML, CSS, and JS - is retrieved within a single page load and other resources are loaded from the server, asynchronously, via AJAX, and inserted (i.e, displayed) to the page as necessary.[MD06]

SPAs design focus on providing a user experience of a native desktop application within the browser environment can be achieved with the proper level of abstraction thanks to advanced JS frameworks like AngularJS[Gooa], Ember.js[Emb], Backbone.js[Bac], and ReactJS[Rea]. These frameworks are initially loaded by the browser when we first visit a SPA, after the first page load.[EAS15] They are responsible for creating and managing independent sections of the application, called *views*, asynchronously loaded from the server. These *views* are not entire HTML pages, but portions of HTML that the user sees and interacts with. [EAS15] They are dynamically attached to the Document Object Model (DOM), usually inside a *div* container, as illustrated in Figure 4.1, or another document area, via JS. As the user navigates and interacts with the page, the framework smoothly swaps the content of one *view* for another *view* and, if necessary, communicates with the server to get additional data to populate *views*. These transactions are done asynchronously using AJAX and, as the prefered data format exchange, JavaScript Object Notation (JSON). This design allows to have more responsive web applications which contribute to their usability by end users.

This approach is fundamentally different from classical web applications. In the next section, we present the differences between this two types of web applications. For now it is suffice to say that, with classical web applications, a full-page reload can be highly disruptive for the user interaction, resulting in the user losing context of his activity and being unable to (temporarily) interact with the web application. In comparison, SPAs are faster and usually provide a better User Experience (UX) since navigation on the application does not require full page reloads, resulting in a much more fluid and responsive experience without the user having to wait for several seconds for a full page to re-render.

With this approach, the traditional browser's design of history navigations is broken. In order to keep track of user's location as they navigate, JS frameworks provide a component called *router*, that controls and manages the browser's navigation history.



```
<!DOCTYPE html>
<html>
<head>
        <title>Shell</title>
        <link rel="stylesheet"
        type="text/css"
        href="app/css/default.css">
</head>
<body>
        <div id="container"></div>
</body>
</html>
```

```
<div id="p1">
  <h1> page 1</h1>
</div>
```

```
<div id="p2">
  <h1> page 2</h1>
</div>
```

Figure 1.1: "Pages" of a SPA - *Views*, adapted from [EAS15]

Routers can solve this problem in two ways. The first one is via *hashbang* - a technique that provides access to the actual Uniform Resource Locator (URL) of the browser, via JS, and can change the browser URLs hash fragment identifier according to the current state, without causing a full page reload.[MP13] This approach forced the creation of navigation paths with the hash symbol (e.g, *http://index.html/#contacts*).

Further ahead, the HTML5 specification introduced the *history.pushState()* and *history.replaceState()* methods in the HTML5 History API, allowing *routers* to access browser's history and manipulating history entries, in a much cleaner way without relying on the fragment identifier. With these solutions, users seem to be navigating through separate pages in the application. This whole process of controlling the UI of a SPA is performed, mostly, on the client side, in the browser, introducing to a new set of problems[MP13] that will be further discussed in Section 1.2.

Currently, millions of users are using these applications every day, such as *Facebook* and *Gmail*, as SPAs become a popular approach to a modern development practice for responsive web applications.

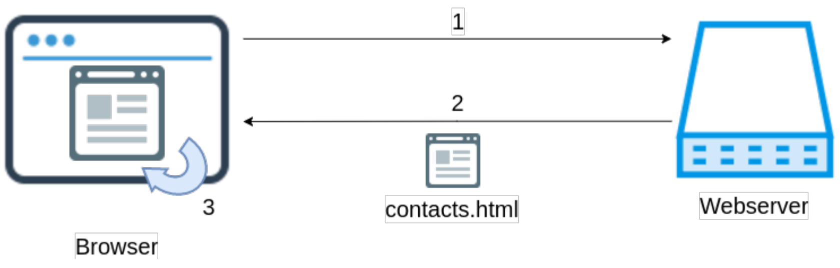
1.2 Differences from Classical Web Applications

In this section, we will focus on the main differences between SPAs and classical web applications. Since these two types of web applications have some differences, it is interesting to explore this topic in order to understand why we need to take a different approach when measuring the performance of SPAs in comparison to known methodologies and techniques for performance assessment in classical web pages.

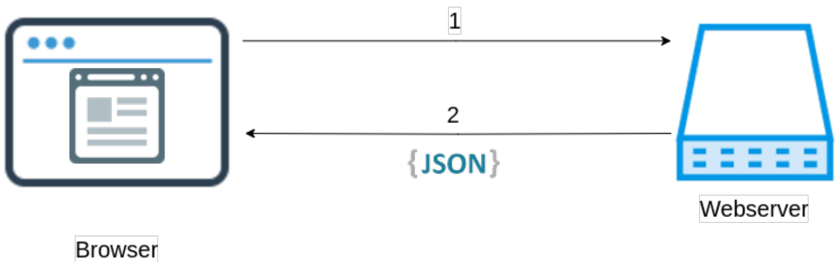
In classical web applications, each page is defined in one single HTML file. Each time a user enters a new URL on the browser location bar, a new request to the server is done in order to get the next page. The server then responds by sending back an entire HTML file causing (potentially containing processed data fetched from a database in the server side) the browser to redraw everything on the page at the same time he loads the necessary resources and assets included on that page (e.g, CSS, JS files and images). This process results in heavier payloads and constant page reloads.

In terms of UX, SPAs have definitely an advantage. These applications feel more fluid and responsive since they run on a single page, dynamically updating the views that need to change only as necessary, resulting in much lighter payloads as most resources are loaded once throughout the lifespan of the application.

To better illustrate this, Figure 1.2 shows the lifecycles of this two types of web applications.



(a) Classical web applications lifecycle



(b) Single page applications lifecycle

Figure 1.2: Classical Lifecycle vs. SPA Lifecycle

Let's assume the initial page on the browser is *main.html*. With classical web applications (Figure 1.2a), as the user interacts with the page, he clicks on a link that leads to the contacts page. The browser will need to make a new request to the server in order to get this new page content (1). The server then responds with the entire *contacts.html* file (2) causing a full page reload in order to display the new content (3).

SPAs (Figure 1.2b) take a different approach. When a user clicks on a link that requests the contacts page, the JS framework is responsible to route and display the new contacts *view*. This content is usually much smaller than an actual *view* content for the full contacts page would be. First, the framework requests the *view* content from the server, asynchronously (1). Then, the server will respond with the contacts content (2), in JSON format, and the new *view* is rendered dynamically without a page reload since the client-side framework is running in the browser.

1.3 Challenges in measuring the performance of SPAs

Measuring the performance of traditional applications is pretty straightforward with the current performance monitoring solutions. These solutions allow for each new page request, to track how long it takes new content to load and the time the browser took to display the new page. Tracking page load times it's accomplished, for example, by listening to the onload [1] event, used to determine when a page finished loading.

With SPAs, all the application code and data are loaded asynchronously thus, making this classic load time metrics unrepresentative of the user perception of the responsiveness of the page load. We can no longer fully rely on the currently available performance monitoring solutions available in browsers that follow a more traditional, or multiple pages design approach. Since all interactions are done on a single page, this leads to a whole new set of challenges to measure performance. In this domain, it is important to mention the work from SOASTA [2], a company that provides web performance optimization solutions, and we now present the three major challenges, presented in their article [Rum]:

- **The *onload* event no longer matters.** Each time a user enters a new URL on the browser location bar, the browser starts to parse the HTML and, assuming the cache is empty, it downloads all the components statically included on the page requested (e.g, images, script files, CSS files, etc.). In the case of SPAs, the JS framework code is also downloaded. When the browser completes downloading these components, it will fire a *window.onload* event, that notifies the page finished loading. The challenge here is that, on a SPA, the framework only starts to run in the browser after this event, dynamically updating *views* as necessary, by fetching new content asynchronously. Since SPA load times will be generally longer than the traditional ones, measuring the page load performance until the triggering of the *onload* event is not an optimal solution because it will not contain the load times of the initial resources that are dynamically loaded.

- **Soft navigations are not real navigations.** At this point, we know that navigation on SPAs is based on switching content of views, by changing the route dynamically as the user interacts within a single page. This route changes, or "internal" navigations, are also called *soft navigations*. They can force the URL to change, without causing a browser refresh as traditional link-based navigations. This is a problem since the current performance monitor solutions rely on traditional browser navigation design, assuming that each browser navigation causes a page to load. Unfortunately, there is no standard mechanism to track *soft navigations* event nor the time required to fetch content while performing these navigations.

[1] https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onload
[2] https://www.soasta.com/

- **The browser will no longer fire the *onload* event, again.** Since the browser only fires the *onload* event after a full page load, we can not track *soft navigations*. The browser will no longer fire the *onload* event again since these navigations do not cause a page to reload. Consequently, we can not track the load timings from resources that have been fetched, asynchronously by the *SPA* framework, leaving us without a notion of when these resources have finish download. The challenge here will be to measure these times and to listen for changes after the *onload* event.

1.4 Problem statement

In this document, we describe a conceptual framework and methodology to reliably measure performance of SPAs and the necessary infrastructure. This framework will be developed at *Feedzai*'s Lisbon office, integrated in the Front End Engineering team. This work aims to build a generic approach to track the evolution of loading time metrics, detect regressions and measure improvements objectively, of both *Pulse* and *Case Manager* products, but ideally being possible to generalize for other SPAs also.

We plan to build a framework that monitors and measures overall performance metrics in these applications and to build a data visualization tool that report results to *Feedzai*, in order for developers easily identify bottlenecks on their applications.

1.5 Document structure

In this chapter, we explain the concept of network-intensive web applications, giving focus on client-side applications or more specifically, single-page web applications. We introduce the main concepts of these applications and made a comparison between SPAs and traditional web applications, to introduce some of the challenges associated with measuring the performance of SPAs.

Chapter 2 presents two network-intensive applications developed by *Feedzai*, *Pulse* and *Case Manager*, which will be the main use cases for applying the solution to be developed in the context of this thesis.

Chapter 3, covers the state-of-the-art of monitoring performance in client-side applications, current techniques that are already monitoring these applications and discusses metrics that can be measured and their relevance.

Chapter 4 presents how we will tackle the problem, providing an overview of the approaches we plan to employ. Furthermore, we present the main architecture and logic of the framework proposed and the architecture of the framework and the high-level schedule of the thesis work.

Chapter 2

Thesis context

The work to be conducted in this thesis elaboration will be integrated in an industrial context, in the particular case, in the *Feedzai* company. We will be focusing on two use cases provided by the company that will guide our work. For completeness, we now briefly discuss a bit the background of the company and the two use case applications.

2.1 Feedzai overview

Feedzai is a data science and machine-learning company with the mission to make commerce safe for business customers. Founded in Portugal, Coimbra by Nuno Sebastião, Pedro Bizarro, and Paulo Marques as the first start-up to be created as a result of Carnegie Mellon University program to create new knowledge in the areas of ICT (Information and Communications Technology) from Portugal through research and education. As a global company, they support its customers from offices in Silicon Valley, London, New York City, Lisbon, Porto, and Coimbra.

Their leading platform powered by Artificial Intelligence (AI) and big data can detect fraud in real time in omnichannel commerce. Companies located across Europe, United States, South America and Africa are currently using *Feedzai*'s anti-fraud solution to keep commerce moving safely.

They have built an intelligent platform with foresight into next-generation fraud technology that provides real utility to data feeds that come from different sources. Instead of using the current rules-based approach to fraud detection, their software solution combines machine learning with behavioral analysis. By using real-time processing, machine-based learning to analyze big data, they make possible to identify fraudulent payment transactions and minimize risk in the financial industry. Banks, payment providers, and retailers are using this technology in order to manage risks, in the way consumers behave when they make purchases related to banking or shopping, across physical and online transactions.

Feedzai handles, per day, $1B in total payments volume, with fraud detection rates varying between 80% and 90% and an associated very low rate of false positives.

In the next section, we will present the two main products that are responsible to ensure detection and preventing fraud in real-time, developed by *Feedzai*. These products will be the main use cases to apply in this thesis work.

2.2 Pulse

Pulse is a platform that integrates big data analytics and machine learning techniques in order to detect and prevent fraud in real-time. It processes transactions, physical and online purchases, cash withdrawals, and many more related events. *Pulse* automatically analyzes imported datasets, creates field mappings and histograms in order to identify patterns and provide additional insights into the data, allowing data scientists and engineers to validate data schemas, clean data and analyze datasets quickly. Combining these built-in data analysis tools, all of this is accessible in one ultimate unified toolset called *Data Science Design Studio*, and in here data scientists can create projects, train machine learning models, collaborate with teams together on projects, share insights and improve outcomes.

The main end goal is to efficiently iterate and train machine learning models, evaluating visually and easily compare them. These models are continuously retrained with new data as events are processed and alerts decided. A *Rules Engine* component adds an extra layer of depth to machine learning models, as it allows data science teams to detect and develop decision rules to optimize the performance of their projects.

All of this follows a flexible data architecture design, composed of different and isolated components. Figure 2.1 illustrates an overview of *Pulse* and how its parts are fully combined.

Pulse is a state-of-the-art and complex platform. It is designed to fully support tasks for different user profiles and built around data science and machine learning concepts. It supports a schema-agnostic architecture, making it easy to customize data schemas for each business use case. Its architecture is composed of several different parts that communicate and support complex scenarios. The main core component is the *Pulse Server*. This component has the responsibility of managing multiple applications, simultaneously, and their respective metadata. In here lies a Jetty Hypertext Transfer Protocol (HTTP) server responsible to deploy web applications and serve a Representational State Transfer (REST) Application Programming Interface (API). Jersey and Jackson are the services that handle data exchange, in JSON format. In the UI perspective, the main components are:

- **Predictive Analytics & Dashboards** - establish expected baseline behavior for performance measurements in order to evaluate success based on historical data, how

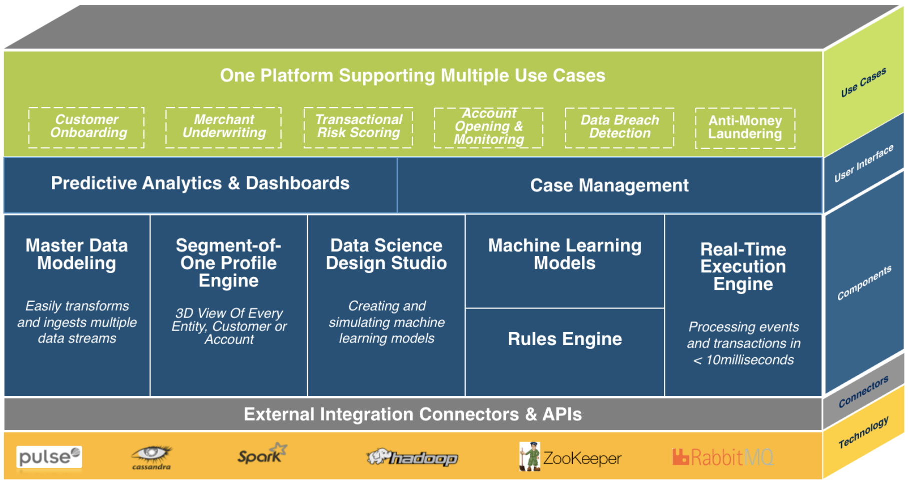

One Platform Supporting Multiple Use Cases
Customer Onboarding
Merchant Underwriting
Transactional Risk Scoring
Account Opening & Monitoring
Data Breach Detection
Anti-Money Laundering
Predictive Analytics & Dashboards
Case Management
Master Data Modeling
Easily transforms and ingests multiple data streams
Segment-of-One Profile Engine
3D View Of Every Entity, Customer or Account
Data Science Design Studio
Creating and simulating machine learning models
Machine Learning Models
Rules Engine
Real-Time Execution Engine
Processing events and transactions in < 10milliseconds
External Integration Connectors & APIs
pulse
cassandra
Spark
hadoop
ZooKeeper
RabbitMQ
Use Cases
User Interface
Components
Connectors
Technology

Figure 2.1: *Pulse*'s components overview

the system is evolving and its current state. Data visualization tools enable users to create custom dashboards that provide real-time tracking business metrics.

- **Case Management** - This component was designed for analysts to have access to a more user-friendly system where teams could act upon alerts that require manual review. Thanks to data visualization techniques, analysts can visually evaluate alerts, improving feedback and final decisions. The interface is named *Case Manager* and, in the next section, we describe the application architecture and main concepts.

2.3 Case Manager

Case Manager aims to bring a state-of-the-art interface that help analysts to discover new fraud patterns by search, navigate and correlate data in the system using an analytical approach supported by rich visualization of data. This platform is targeted for fraud analysts working in companies that manage risk internally, manually handle events that fulfill companies criteria, performing actions on alerts, organize workloads of teams and automatize processes regarding events. This product aims at high customization since each client has his own data scheme, by reducing the effort of integration when delivering solutions and support fully different scenarios.

It is integrated with *Pulse*, working together to confront actions and improve fraud-detection by manually handling them. It is able to set up rules to decide which transactions must need for human review. The information received and processed is then sent back to *Pulse*, via HTTP, and notified through its REST API.

Figure 2.2 illustrates a general overview of the architecture and communication between the different components of *Case Manager* and how it works with *Pulse*. These components are:

- **Load Balancer** - improves the distribution of workloads and connects browsers to *Case Manager* instances.
- **RabbitMQ** - messaging platform used by *Pulse* to support communication between machines and processes across datacenters.
- **Pulse** - sends events to *Case Manager* and receives feedback.
- **RDBMS** - data is stored in a relational database.

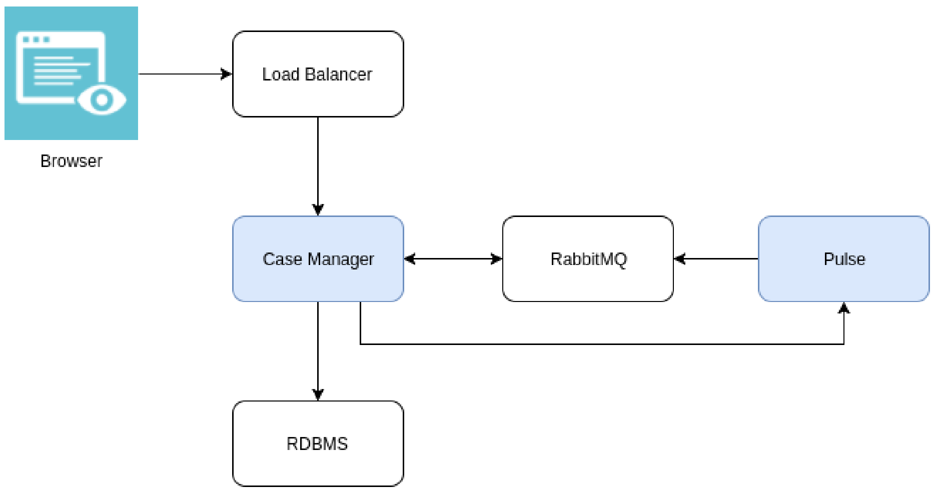


Figure 2.2: Communication between *Case Manager*'s components

From a front-end architectural perspective, many libraries and tools are used in *Case Manager*. React and Redux is the framework used to build the UI, and tools such as bundler Webpack and npm as the JS package manager.

Case Manager's application entry point is a JS file - *index.js*, responsible for initializing the redux store and render the *root* react *App* component in the DOM. This component initializes the application (e.g, fetch the UI configuration) and when everything is loaded, it renders the *Router* component. This component is responsible to render a react component based on the current application path.

The *MainContainer* component represents the general structure of a page. It contains common sections (i.e, header and sidebar) and also contains dynamic subsections. The content of these subsections changes accordingly to the current application path and are defined in separate components.

Together, these components play an important role to make *Case Manager* working in sync along with some other components that are necessary for each application page (e.g, login page, details page, and so on).

2.4 Summary

This chapter described the underlying concepts and architecture for both *Pulse* and *Case Manager*. We first described *Pulse*, a platform powered by AI that transforms multiple data streams and fraud insights, creating risk profiles looking towards better fraud prevention and improved customer experience. Then, we described *Case Manager*, a human-readable semantic layer to the underlying machine logic, with rich data visualization for analysts better discover new fraud patterns.

Chapter 3

State of the art

This chapter presents an overview of the state-of-the-art methodologies and techniques used to measure performance in SPAs. In Section 3.1, we first describe the architecture of web applications in order to understand how their different components interact and the overall structure of these applications. In section 3.2, we present some frameworks that help building SPAs. Section 3.3 introduce some current tools that monitor and measure performance of web applications in general, but also for SPAs. We also present the challenges of measuring performance in these applications and what are the most relevant metrics to consider. Lastly, Section 3.4, presents a short discussion about how these techniques and solutions relate but do not completely solve the problem we are trying to address.

3.1 Web Applications architecture

Web applications are distributed applications, generally using the web browser as its client. These client-server applications go beyond the two-layered client-server paradigm, using a multi-tier architecture, where each tier represents an application layer. Usually, a standard three-tier architecture is used for web applications:

3.2 SPAs frameworks

Currently, a wide collection of frameworks adopted the principles of SPAs. Frameworks such as AngularJS[Gooa], Ember.js[Emb], and ReactJS[Rea] are available as open-source projects that simplify the task of design and build well-architected SPAs. We know that these applications heavily depend on JS, resulting in more interactive interfaces on the client-side. It is essential to have a clean and well structured code, allowing SPAs developers to focus more on improving efficiency, code quality and maintainability.

These frameworks are all implemented in JS. This high-level programming language is usually used to make web pages more interactive and, with SPAs frameworks, allows web browsers to dynamically transfer data to the web server and retrieve content dynamically without a full page reload. By helping to build more secure and efficient SPAs faster, JS frameworks are used by developers to build their applications. Also, a combination of frameworks with libraries (e.g, React[Rea] and Redux[Red]) can help building more complex web applications. This strategy is usually adopt by companies such as *Feedzai*, to build their products.

When it comes down to performance, these JS frameworks cause extra overhead. When we first visit a SPA, the framework's code is initially loaded by the web browser and will be responsible not only to build views, but also to download the necessary resources, asynchronously, resulting in slower first page loads. Hence, it is essential to address the impact of this overhead when it comes to measure performance on SPAs.

In summary, this section presented the frameworks that are currently used by developers and companies to build their SPAs, in order to understand how these applications are built. The next section presents an overview of the measurements and techniques we can use to monitor and measure performance in today's wide range of web applications.

3.3 Performance measurements

3.3.1 Performance measurements for web applications

When it comes down to measure performance in web applications, there are alot of tools, each one with their utility and functionality. Generally, these tools aim to measure and benchmark performance metrics. Metrics such as the total number of requests and bytes transferred over the network and response time. This last, is the time it takes for a request to be processed (e.g, in the server) and a response is returned and, thus making probably the most important metric in web performance. We now present some of the widely used tools and techniques that helps developers and companies to easily identify bottlenecks in their applications.

3.3.1.1 Apache JMeter

Apache JMeter[Apa] load testing tool is considered to be a web performance testing tool. It is a Java application and it is designed to measure performance in web applications. It is designed to load tests and functional behavior. This tool can test the performance of static resources but also web dynamic languages such as ASP.NET[Asp]. It can simulate heavy loads, individual servers or networks. It also provides a graphical interface, with rich data visualization and analysis plugins enabling faster debugging and test plan creation.

3.3.1.2 WebLOAD

WebLOAD[Web] is a load and web performance testing tool built for web application that enables load testing using technologies such as AJAX and HTML5. This tool can generate load from on-premises machines, but also from the clout. Some of the features include DOM-based recording and playback, HTTP functionality and load methods. Based on JS, this tool can supports large-scale testing in complex user load conditions. It is ease to use, by generating clear analysis of a web application's performance and functionality.

3.3.1.3 LoadUI

In comparison to the previous tools, this web performance testing tool is probably the most interactive and flexible one, allowing to create and update tests while a web application is tested. *LoadUI*[Loa] offers advanced report and analysis features, to easily examine the actual performance. In comparison to previous tools, a great advantage is that we do not need to constantly restart the tool as the application changes.

Nowadays, a wide range of these type of web performance testing tools are available for free, as open-source projects, making it available to developers and companies that want to measure their web applications performance. As web development evolve, new modern practices such as SPAs, are introducing to a whole new set of challenges when it comes down to measure their performance. This web applications break the traditional web design approach and consequently, this current performance monitoring solutions can not be apply when it comes to measure performance. In the next section, we present the state-of-the-art performance measurements in these web applications.

3.3.2 SPAs performance measurements

With SPAs, we can not fully rely on the current web applications performance monitoring solutions based on traditional techniques, since the design approach is fundamentally different from classical web applications. This section presents and discuss briefly, some tools and techniques used to monitor and measure SPA overall performance. Before we dive into how performance is measured in these web applications, we introduce the metrics that can be measured and explaining some measurement challenges.

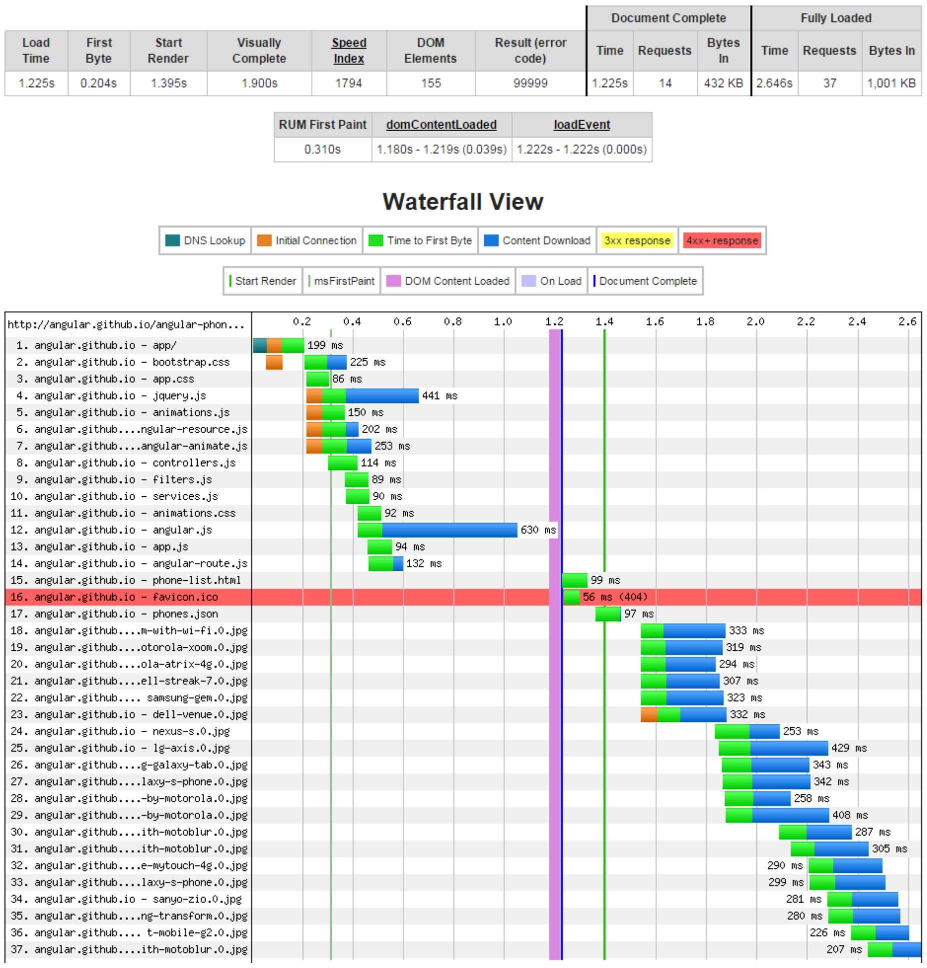
Metrics such as SPA framework load time, first page load and time to interactive[1] are essential when it comes down to measure performance of SPAs and, by using the current performance monitoring solutions, we can not be accurately measure them.

For example, the window.onload event, is trigger when all static resources finish loading, thus not waiting for asynchronous activity that is happening behind the scenes, by the SPA framework (i.e, loading necessary resources for views). Figure 3.1 illustrates a waterfall overview of these resources load times and Figure 3.2 represents a first page

[1]Time when the layout is visually stable and capable of handle user input

load timeline, on a SPA built with AngularJS[Gooa]. In this case, the browser fires the onload event at 1.2 seconds but the application is only visually complete in 1.9 seconds.

In this work, we intend to use these load times metrics as our main performance measurements for SPAs.



							Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
1.225s	0.204s	1.395s	1.900s	1794	155	99999	1.225s	14	432 KB	2.646s	37	1,001 KB

RUM First Paint	domContentLoaded	loadEvent
0.310s	1.180s - 1.219s (0.039s)	1.222s - 1.222s (0.000s)

Waterfall View

```
http://angular.github.io/angular-phon...
 1. angular.github.io - app/                      199 ms
 2. angular.github.io - bootstrap.css             225 ms
 3. angular.github.io - app.css                   86 ms
 4. angular.github.io - jquery.js                 441 ms
 5. angular.github.io - animations.js             150 ms
 6. angular.github....ngular-resource.js          202 ms
 7. angular.github....angular-animate.js          253 ms
 8. angular.github.io - controllers.js            114 ms
 9. angular.github.io - filters.js                89 ms
10. angular.github.io - services.js               90 ms
11. angular.github.io - animations.css            92 ms
12. angular.github.io - angular.js                630 ms
13. angular.github.io - app.js                    94 ms
14. angular.github.io - angular-route.js          132 ms
15. angular.github.io - phone-list.html           99 ms
16. angular.github.io - favicon.ico               56 ms (404)
17. angular.github.io - phones.json               97 ms
18. angular.github....m-with-wi-fi.0.jpg          333 ms
19. angular.github....otorola-xoom.0.jpg          319 ms
20. angular.github....ola-atrix-4g.0.jpg          294 ms
21. angular.github....ell-streak-7.0.jpg          307 ms
22. angular.github.... samsung-gem.0.jpg          323 ms
23. angular.github.io - dell-venue.0.jpg          332 ms
24. angular.github.io - nexus-s.0.jpg             253 ms
25. angular.github.io - lg-axis.0.jpg             429 ms
26. angular.github....g-galaxy-tab.0.jpg          343 ms
27. angular.github....laxy-s-phone.0.jpg          342 ms
28. angular.github....-by-motorola.0.jpg          258 ms
29. angular.github....-by-motorola.0.jpg          408 ms
30. angular.github....ith-motoblur.0.jpg          287 ms
31. angular.github....ith-motoblur.0.jpg          305 ms
32. angular.github....e-mytouch-4g.0.jpg          290 ms
33. angular.github....laxy-s-phone.0.jpg          299 ms
34. angular.github.io - sanyo-zio.0.jpg           281 ms
35. angular.github....ng-transform.0.jpg          280 ms
36. angular.github.... t-mobile-g2.0.jpg          226 ms
37. angular.github....ith-motoblur.0.jpg          207 ms
```

Figure 3.1: Waterfall overview from resources in a SPA

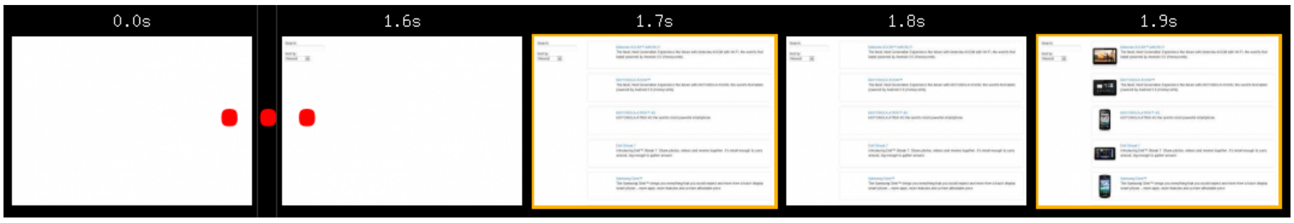


Figure 3.2: First page load filmstrip of a SPA

3.3.2.1 Google Analytics

Google Analytics[Goob] is a service provided by *Google* that can track and report all kind of website traffic. It works by simply adding a snippet of JS code - *analytics.js* - to every page of the website. For each page the user visits, the code collects data and reports to *Google*'s data collection servers. This snippet collects data such as the total time a user spends on a website, the time a user spends on a page, geographic location and among other relevant information.

This implementation works well for classical web applications since the snippet runs every single time the users load a new page. With SPAs, the site just loads once - the first page load - and other subsequent content is dynamically brought into the page without a full page load. Consequently, the snippet code runs only once and thus, is not able to track the different views over which the user navigates, therefore providing imprecise information.

In order to overcome this issue, the code of this tracker needs to be updated. For SPAs, when content is dynamically loaded, the URL address bar updates and the tracker needs to update the data about this new content. A limitation of this solution is that it requires the developer to manually write code in order to capture the information that should be tracked during the user interaction with the page.

To solve this, we can integrate a JS library named *autotrack*[Aut], combining with the *analytics.js* snippet. It provides tracking for interactions that we care about without the constant effort of manually update the *analytics.js* tracking code. This library includes a plugin called *urlChangeTracker*[Url] that detects changes to the URL, via the History API, and updates the tracker by sending additional page views, automatically.

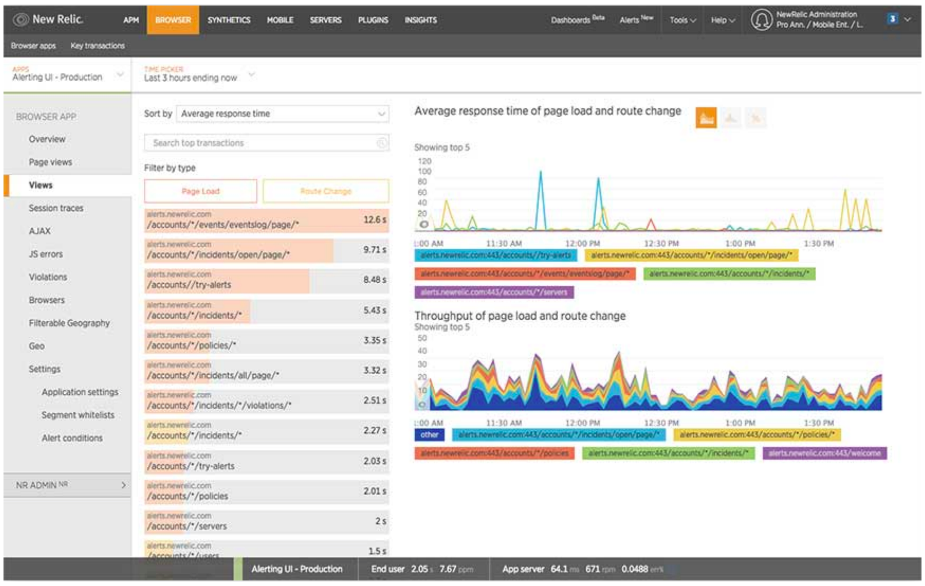
Thus, with this solution, we allow SPAs to be tracked, by adapting the original *analytics.js* tracking code. However, the *urlChangerTracker* plugin does not support tracking hash changes, which means that will not work properly for all SPAs and, if we aim to build a generalized tool to work properly for all SPAs, this solution does not seem total ideal for our goal.

3.3.2.2 New Relic Browser

New Relic is a company that provides software analytics and application performance monitoring as a service. One of their products is New Relic Browser[Rel] that provides full visibility into web applications lifecycles and supports monitoring for SPAs, independently of the framework used to build the application.

This service also provides more detailed information about all the activity that produces events, such as AJAX requests, synchronous and asynchronous JS operations, among others. To achieve this, it uses a JS snippet responsible for collecting data and monitoring interactions that can potentially lead to a page load or route changes. If these interactions lead to a route change, the interaction is saved along with additional information about the activity that lead to that event.

In comparison to *autotrack*[Aut], this solution is framework-agnostic, which means that it is compatible with most of SPA frameworks and, thanks to a browser monitoring UI (illustrated in Figure 3.3), more detailed information can be easily displayed to clients, helping them to analyze pitfalls of their applications. This solution is not suitable for our solution, because we aim to build a tool that must run on premise, without relying the contact for external services.



alerts.newrelic.com /accounts/*/events/eventslog/page/*	12.6 s
alerts.newrelic.com /accounts/*/incidents/open/page/*	9.71 s
alerts.newrelic.com /accounts//try-alerts	8.48 s
alerts.newrelic.com /accounts/*/incidents/*	5.43 s
alerts.newrelic.com /accounts/*/policies/*	3.35 s
alerts.newrelic.com /accounts/*/incidents/all/page/*	3.32 s
alerts.newrelic.com /accounts/*/incidents/*/violations/*	2.51 s
alerts.newrelic.com /accounts/*/incidents/*	2.27 s
alerts.newrelic.com /accounts/*/try-alerts	2.03 s
alerts.newrelic.com /accounts/*/policies	2.01 s
alerts.newrelic.com /accounts/*/servers	2 s
alerts.newrelic.com /accounts/*/users	1.5 s

Alerting UI - Production End user 2.05 s 7.67 ppm App server 64.1 ms 671 rpm 0.0488 err%

Figure 3.3: New Relic Browser's UI

3.3.2.3 Boomerang

Boomerang[Boo] is a JS library that measures the performance of page loads and interactions of a website from the end user's point of view. It was created by Philip Tellis, from *Yahoo*[Yah], and today it is an open-source project maintained by developers from companies such as *SOASTA*[Soa]. *SOASTA*'s product, *mPulse*, is a performance monitoring solution that loads *Boomerang*[Boo] in order to monitor and measure web applications performance.

It was first designed for classical web applications, but with the constant increase in popularity of SPAs in modern applications, this library was updated in order to support these applications. Currently, *Boomerang*[Boo] supports several frameworks and is able to track all SPA navigations. To achieve this, it will start by listen for several events from the SPA framework (e.g, *$routeChangeStart* in AngularJS) and, between the start and end of route changes, it will start monitoring for changes in the DOM (by listening to any MutationObserver[2] events), tracking any downloads of CSS, JS, images and XMLHttpRequests. If some changes occur, *Boomerang* automatically assumes that they were relevant to the route change event, considering a navigation to be complete when these new resources have been fetched.

According to *Boomerang*[Boo] documentation, it supports SPA frameworks such as AngularJS[Gooa], Ember.js[Emb] and Backbone.js[Bac] by enabling the right plugin.

[2]https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

In the purpose of this thesis, these frameworks do not apply for our main use cases (e.g, *Case Manager* built with ReactJS[Rea]) and thus, some extra configuration is necessary in order to work. This solution is a case to further study and to probably apply, to our solution.

3.4 Discussion

In this chapter, we introduce the architecture of web applications, in order to briefly understand how these applications are designed. Then, we introduce the concept of SPA frameworks to fully understand how these applications are built. We also present some tools and techniques that are currently measuring performance in both traditional web applications and SPAs. We can now understand that measuring performance in SPAs is not straightforward with current performance monitoring solutions.

The next chapter introduces a conceptual framework and methodology to reliable measure performance in SPAs.

CHAPTER 4

Approach

In this section, we present the main architecture and logic of the framework proposed, along with the task schedule and requirements to build and validate the given solution. First, Section 4.1 presents how we will tackle the problem, providing an overview of the conceptual framework and methodology. Section 4.2, presents the list of requirements proposed by *Feedzai*, for the given solution.

4.1 Framework proposed

This work aims to build a generic approach to track the evolution of loading time metrics, detect regressions and measure improvements objectively, of both *Pulse* and *Case Manager* products developed by *Feedzai*, but ideally being possible to generalize for other SPAs also.

We plan to build a framework that monitors and measures overall SPA performance metrics combining with a data visualization tool. The framework will work as a Command-Line Interface (CLI) that provides load time metrics, as JSON the output format, reporting these results back to *Feedzai*. The data visualization tool will parse the different outputs and manage to build a more user-friendly interface, with the goal for developers analyse the metrics results, and to easily identify bottlenecks on their applications. This tool will implement data visualization techniques, converting JSON format outputs such as statistics, graphics and timelines.

4.2 Framework requirements

The following requirements were proposed by *Feedzai*:

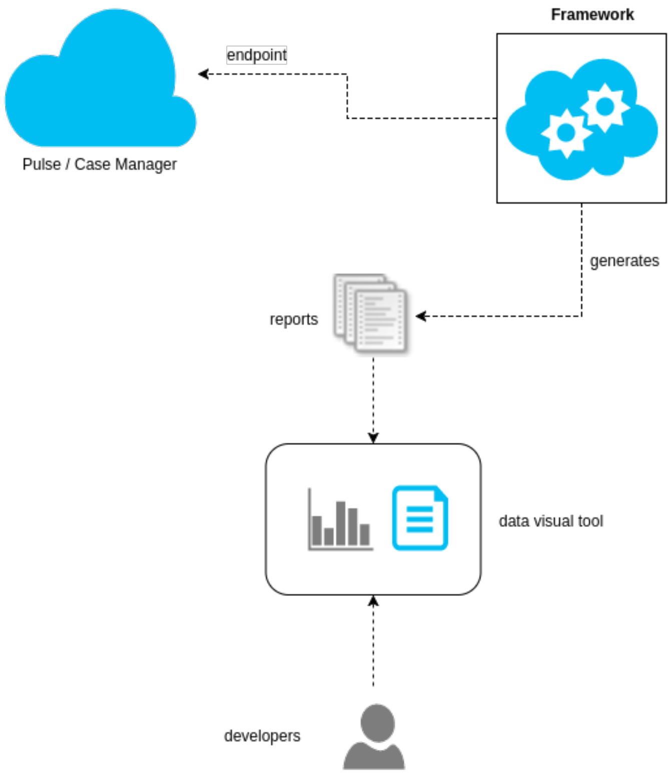


Figure 4.1: Architecture for the solution proposed

- **Must run on premise.** That is, the tool should be able to run on Feedzai machines without contacting external services (cloud services obviously fall on this category)
- **Must be able to integrate with Feedzai's continuous integration infrastructure.** This entails being a self contained application with an output in a parseable format.
- **Must be generic enough to be used in different SPAs.** Notability in the two SPAs developed at *Feedzai*, but ideally also in others.
- **Must produce replicable results.** Being a performance measuring tool it is very important that variables are reduced to a minimum and we have replicable results. Ideally the application should be able produce replicable results even in different machines.
- **Must measure simple and relevant metrics.** The measured metrics should be easy to explain to external stakeholders and should have a big impact on the user experience.
- **Must be well documented, simple to operate and maintain.** All the configurations must be documented, it must be able to operate without human intervention and mustn't require a lot of maintenance.
- **Should execute in an efficient way.** The application should not consume "large" amounts of resources.

Bibliography

[EAS15] J. Emmit A. Scott. *SPA Design and Architecture: Understanding Single Page Web Applications*. Manning, 2015. ISBN: 978-1-61729-243-9.

[MD06] A. Mesbah and A. van Deursen. "Migrating Multi-page Web Applications to Single-page AJAX Interfaces." In: *CoRR* abs/cs/0610094 (2006). arXiv: cs/0610094. URL: http://arxiv.org/abs/cs/0610094.

[MP13] M. S. Mikowski and J. C. Powell. *Single-Page Web Applications: JavaScript End-to-End*. Manning Publications, 2013. ISBN: 1617290750,9781617290756.

Webography

[Rum] *AngularJS Real User Monitoring Single Page Applications.* URL: https://www.soasta.com/blog/angularjs-real-user-monitoring-single-page-applications/.

[Apa] *Apache JMeter.* URL: http://jmeter.apache.org/.

[Asp] *ASP.NET.* URL: https://www.asp.net/.

[Aut] *autotrack.* URL: https://github.com/googleanalytics/autotrack.

[Bac] *Backbone.js.* URL: http://backbonejs.org/.

[Boo] *Boomerang.* URL: https://github.com/SOASTA/boomerang.

[Emb] *Ember.js.* URL: https://www.emberjs.com/.

[Gooa] Google. *AngularJS Developer Guide.* URL: https://angularjs.org/.

[Goob] *GoogleAnalytics.* URL: https://www.google.com/analytics/#?modal_active=none.

[Loa] *LoadUI.* URL: https://www.loadui.org/.

[Rel] *New Relic Browser.* URL: https://newrelic.com/browser-monitoring.

[Rea] *React.* URL: https://reactjs.org/.

[Red] *Redux.* URL: https://redux.js.org/.

[Soa] *SOASTA.* URL: https://www.soasta.com/.

[Url] *UrlChangeTracker.* URL: https://github.com/googleanalytics/autotrack/blob/master/docs/plugins/url-change-tracker.md.

[Web] *WebLOAD.* URL: https://en.wikipedia.org/wiki/WebLOAD.

[Yah] *Yahoo.* URL: https://www.yahoo.com/.