Rodrigo de Almeida Graças

Degree in Computer Science and Engineering

Measuring Performance in Network-Intensive Web Applications

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

Master of Science in
Computer Science and Engineering

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

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

Luís Filipe Alves Cardoso, Front-End Engineer, Feedzai

Nuno Alexandre Neves Cruz, Front-End Engineer, Feedzai

Examination Committee

Chair: Margarida Paula Neves Mamede
Rapporteur: Alcides Miguel C. Aguiar Fonseca
Member: João Carlos Antunes Leitão

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Measuring Performance in
Network-Intensive Web Applications

Dedicated to my father, Mário Graças, who always encouraged
me to go on every adventure, especially this one.
Even if you're not here at the end, I hope you're proud of me.
Love you, dad

Acknowledgements

This thesis would not be possible if it were not for the incredible people I had the opportunity to meet during this adventure. First and foremost, I have to thank my thesis advisors, João Leitão and João Costa Seco, for their guidance and support throughout this study and specially for ther confidence in me.

I would also like to sincerely thank to Luís Cardoso, Nuno Neves and the whole Case Manager team at Feedzai for helping me throughout my internship, and to help me be a better professional.

To all my friends, thank you for always being there on my darkest days and for your true friendship. To my mother and grandmother, for their unconditionally love and support. Thank you both for giving me strength and to help me reach my goals.

Finally, I am deeply thankful to Ana Rita Reis for being by my side in this final adventure. You deserve my wholehearted thanks as well. Thank you to inspire me and support me. I am truly grateful for having you in my life.

Abstract

Nowadays performance is a key aspect of web applications. In order to be successful, and productive, users expect a system to be 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. 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 because those metrics relied on synchronous page loading.

Regressions happen and product requirements change constantly. Moreover applications are worked on by multiple developers. All this contributes to the creation of technical debt and it turns affects the load time. And so, it is important for developers to understand the effects and impact on the performance of their applications and make an early call, shifting towards adopting performance budgets in the development stage, before deploying their applications into the wild.

In this thesis, we propose a framework and methodology to reliably measure performance of single-page applications, describing the necessary infrastructure to track the evolution of network usage and load time metrics, detect regressions, and measure improvements objectively. We address the problem of providing feedback to developers regarding the impact of modifications performed on their applications, by helping them to assess their applications performance and identify potential sources of bottlenecks. We further instantiate this work in two typical network-intensive applications developed by Feedzai.

Keywords: Network-Intensive, Web Applications, Single-Page Web Applications, Web Performance

Resumo

Hoje em dia, o desempenho é um aspecto fundamental nas aplicações web. Para ser bem sucedido e produtivo, os utilizadores esperam que o sistema seja altamente responsivo. Além disso, o desempenho de aplicações web com uso intensivo da rede é sensível a muitos fatores.

Atualmente, as empresas desenvolvem os seus produtos usando um paradigma de web design bem estabelecido, conhecido como Aplicações de Página Única. Estas aplicações apresentam um novo conjunto de desafios. Medir o desempenho destas aplicações de forma filedigna não é uma tarefa simples. Tanto o código da aplcação como os dados são carregados de forma assíncrona, tornando as métricas clássicas de tempo de carregamento não representativas porque essas métricas dependem do carregamento síncrono da página.

Regressões acontecem e os requisitos do produto mudam constantemente. Além disso, as aplicações são desenvolvidas por vários programadores. Tudo isso contribui para a criação de dívida técnica e afeta o tempo de carregamento. E, portanto, é importante que os programadores entendam os efeitos que afetam o desempenho das suas aplicações e que o percebam antecipadamente, mudando para a adoção de orçamentos de desempenho na fase de desenvolvimento, antes de colocar as suas aplicações em produção.

Nesta tese, propomos uma estrutura e uma metodologia para medir de forma viável o desempenho de aplicações de página-única, descrevendo a infraestrutura necessária para acompanhar a evolução do uso da rede e as métricas de tempo de carregamento, detectar regressões e medir melhorias de forma objetiva. Abordamos o problema de fornecer informação aos programadores sobre o impacto de modificações realizadas nas suas aplicações, ajudando-os a avaliar o desempenho das suas aplicações e a identificar possíveis fontes de bottlenecks. Instanciamos ainda mais este trabalho em duas aplicações típicas de rede intensiva desenvolvidas pela Feedzai.

Palavras-chave: Rede-Intensiva, Aplicações Web, Aplicações Web Página-Única, Desempenho Web

Contents

List of Figures

Listings

Glossary

AJAX	Method that allows to exchange data from a server asynchronously, allowing to update a web application without refreshing the page
beacon	A web beacon is a technique used to track who is visiting a web page.
div	Tag that defines a section in an HTML document
headless	A headless browser is a web browser without a graphical user interface.
hotfix	A hotfix is a single, cumulative package that includes information that is used to address a problem in a software product (i.e., a software bug). Typically, hotfixes are made to address a specific customer problem.
polling	Polling, in computer science, refers to actively sampling the status of an external device by a client program as a synchronous activity.

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CI	Continuous Integration
CLI	Command-Line Interface
CSS	Cascading Style Sheets
CSSOM	CSS Object Model
DOM	Document Object Model
DS	Data Science
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
ML	Machine Learning
NPM	Node Package Manager
QA	Quality Assurance
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 essential 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 motivates 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. JSON and images also contribute considerably to the traffic of web applications. It is critical for applications to access data over the network and to optimize overall performance. Due to web applications growing importance, companies rely on these applications. Performance has to be consistently measured and monitored, in order to avoid critical consequences and result in lost customers.

As web development practices evolve, a modern web design paradigm widely used, are 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 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 network 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 Motivation

Measuring 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 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. The three major challenges are:

- **The onload event no longer matters.** Each time a user enters a new Uniform Resource Locator (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 the onload event, that notifies the page finished loading. The challenge here is, on a SPA, the framework only starts to run in the browser after this event, dynamically updating *views* as necessary, 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 resources that are dynamically loaded.

- **Soft navigations are not real navigations.** Navigation on a SPA 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. In this thesis, we aim to measure network metrics in these type of navigations.

- **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.

We now understand some of the challenges of measuring performance in SPAs. As these applications tend to become more complex, they can easily become network-intensive, fetching a lot of resources between navigations. This can potentially lead to performance bottlenecks and it is crucial to measure all network activity. In this thesis, we aim to overcome these challenges and provide a solution that can reliably measure all these requests in soft navigations.

1.2 Problem statement

In this thesis, we tackle the problem of measuring performance in SPAs, by presenting a solution that provide network related metrics about these applications. This solution was developed at Feedzai's Lisbon office, integrated in Front-End Engineering team, at Case Manager's team. Our work will be applied, notably, in two real-world applications developed by Feedzai, Pulse and Case Manager, where in Section 1.3.1 and Section 1.3.2, we present these applications, respectively.

Furthermore, our solution implementation aims to build an extensible and generic approach that can be used with custom configurations, supporting not only Feedzai applications, but with other SPAs. Moreover, with this solution, developers will be able to receive instant feedback about the performance impact of new product features, in the development stage, and also monitor their applications performance over time and metrics variations.

We also design the solution to be simple, both to operate and maintain, and produce repeatable results with simple but relevant metrics. Hence, we aim to give an opportunity to all developers to empower them to make a call on the client-user experience and worry about performance before deploying their applications into the wild.

1.3 Feedzai

Feedzai [Feee] is a data science and machine-learning company with the mission to make commerce safe for business customers. Founded in Portugal, in the city of 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. Now, as a global company, they support its customers from offices in Silicon Valley, London, New York City, Lisbon, Porto, Coimbra, and Hong Kong.

Their leading platform powered by Artificial Intelligence (AI) and big data can detect fraud in real time, making possible to identify fraudulent payment transactions and minimize risk in the financial industry. Instead of using the current rules-based approach to fraud detection, their software solution combines machine learning with behavioral analysis.

Banks, payment providers, and retailers located across Europe, United States, South America, Africa, and Asia, 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, keeping commerce safe. According to Feedzai, it handles per day, $5B 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 sections, we will present the two main products developed by Feedzai, Pulse and Case Manager, that together are responsible to ensure detection and fraud prevention in real-time. These products will be the main use cases to apply in this thesis work.

1.3.1 Pulse

Pulse is a platform that integrates big data analytics and Machine Learning (ML) 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 and creates 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. [Feec]

Combining these built-in data analysis tools in one toolset, data scientists can create projects, train ML models, collaborate with teams together on projects, share insights and improve outcomes. It also allows data science teams to detect and develop decision rules to optimize the performance of their projects, adding an extra layer of depth to ML models. The main end goal is to efficiently iterate and train ML models, evaluating visually and easily compare them. These models are retrained manually with new data as events are processed and alerts decided.

It is designed to fully support tasks for different user profiles and built around data science and ML 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, Pulse Server, 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). The main component in the UI is the Data Science (DS) toolset where data scientists can build new DS projects, train ML models and share insights. By collaborating together on projects, DS teams can quickly create new data sources and transform data, to be used in models and runtime workflows, in order to improve outcomes faster.

Currently, a combination of two SPA frameworks, React [Rea] and Backbone.js [Bac], are part of Pulse's front-end stack to build the UI (represented in Figure 1.1), and a mix of React Router and Backbone.js Router are responsible for the application routing. Pulse also uses tools such as Webpack as modules bundler, and Node Package Manager (NPM) as the JS package manager.

Pulse is a *state-of-the-art* and complex web application platform. In 2017, Pulse had more than 220 React components and 1000 JS modules. By constantly developing new features, it is important for developers to shift towards adopting performance, as current business demands push complexity forwards.

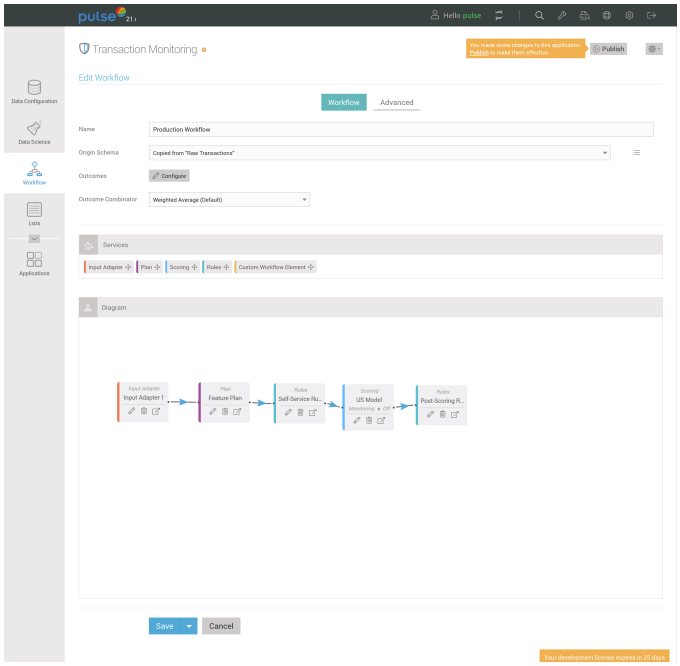


Figure 1.1: Pulse UI

1.3.2 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 automate 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. [Feea]

It allows to set up rules that decide which transactions need human review. The information received and processed is then sent back to Pulse, via HTTP, and notified through its REST API.

From a front-end architectural perspective, many libraries and tools are used in Case Manager. React [Rea] is the framework used to build Case Manager's UI, as shown in Figure. 1.2, Redux as the state management library and React Router responsible for the application router. Similar to Pulse, tools such as bundler Webpack and NPM are also used, and many more.

In 2016, the first version of Case Manager had 171 JS modules and, two years later, this number increased to almost 500 JS modules. As requirements constantly change, it is important for developers to shift towards measuring performance and carefully monitor their variations and understand its impact.

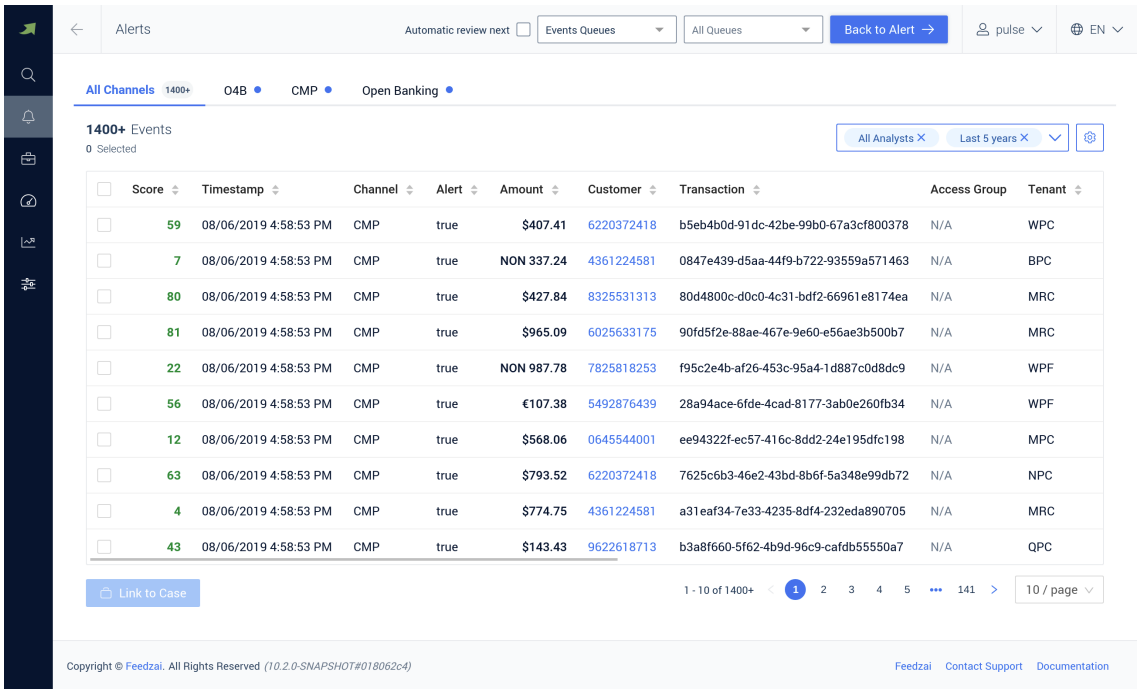


Figure 1.2: Case Manager UI

1.4 Contributions

This work presents two main contributions:

- A framework for measuring performance metrics for SPAs, with the ease of being integrated into a CI environment in order to automate the creation of performance reports, with the aim of constantly monitoring these applications.

- A study of the current performance of one of Feedzai's products, Case Manager, and validation of the solution against other types of SPAs.

1.5 Structure of the Document

In this chapter, we explained the concept of network-intensive web applications, giving focus on client-side applications or more specifically, single-page web applications. We then introduced Feedzai and the company vision and mission. Then, we dive deep into Feedzai's two main products, Pulse and Case Manager, both SPAs that will be the main use cases to apply this thesis work. The remainder of this document is organized as follows:

Chapter 2 covers the fundamentals of web applications and SPAs. In here, we describe the *state-of-the-art* solutions of monitoring and measuring performance in these two types of web applications, and describe the current measurement techniques and performance metrics.

Chapter 3 describes the framework responsible for measuring and monitoring performance in SPAs. We describe the concepts needed to the implementation and present the main architecture. This chapter also includes how the solution is integrated in Feedzai development environment, on a continuous integration level.

Chapter 4 details our implementation of the framework specified in Chapter 3. In this chapter we describe, in more detail, the different components and dependencies, how to extend the configuration and navigate through the codebase in order to understand how it gathers performance metrics of applications.

Chapter 5 presents the experimental evaluation, where we share the results for the measurements applied in the two main cases, Pulse and Case Manager. We then give a brief review about the performance variations and discuss the results.

In Chapter 6 we present some observations on our work, regarding the importance of these type of measuring tools comparing to our results. As future work, we share some paths on how this solution will help to tackle the next step of measuring performance, which is improving it.

Chapter 2

Related Work

In this chapter we present some relevant related work for the purpose of this dissertation and it is organized in two sections: Section 2.1 covers the topic of SPAs, where we define important concepts, how these applications work and the differences from classical web applications. In Section 2.2, we list some modern performance metrics and presents current measurement tools that are used to measure and monitor performance in web applications.

2.1 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 that can be achieved with the proper level of abstraction thanks to JS frameworks like AngularJS [Deva], Ember.js [Emb], Backbone.js [Bac], Vue.js and React [Rea]. These frameworks are initially loaded by the browser when we first visit a SPA, after the first page load. They are responsible for creating and managing independent sections of the application, called *views*, where the data rendered on these sections is 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 2.1, 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, JSON. This design allows to have more responsive web applications which contribute to their usability by end users.

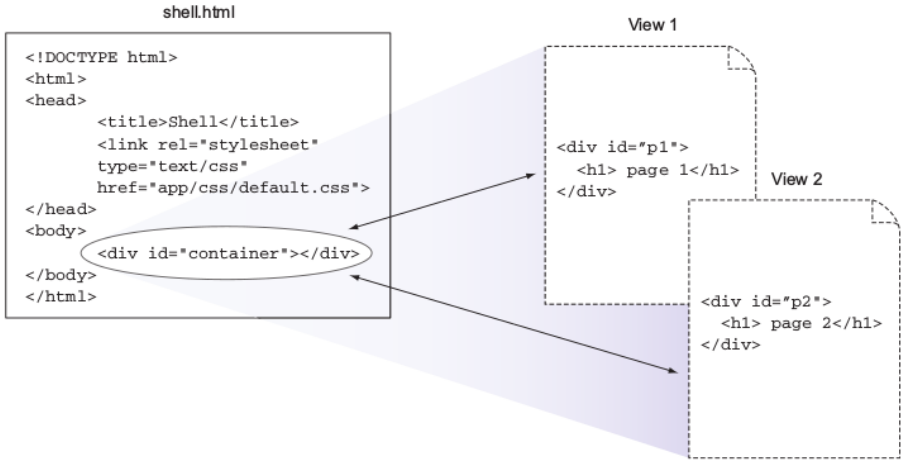


Figure 2.1: "Pages" of a Single-Page Application (SPA) - *Views*, adapted from [EAS15]

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 suffice it 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, SPA are faster and usually provide a better 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.

Routers can solve this problem in two ways. The first one is via *hashbang* - a technique that provides access to the actual 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, as discussed previously on Section 1.1.

Currently, millions of users are using these applications every day, such as Facebook,

Gmail, Google Docs or Twitter, as SPAs become a popular approach to a modern development practice for responsive web applications. In the next section, we present the differences of these applications and traditional web applications.

2.1.1 Differences from classical web applications vs. single-page 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.

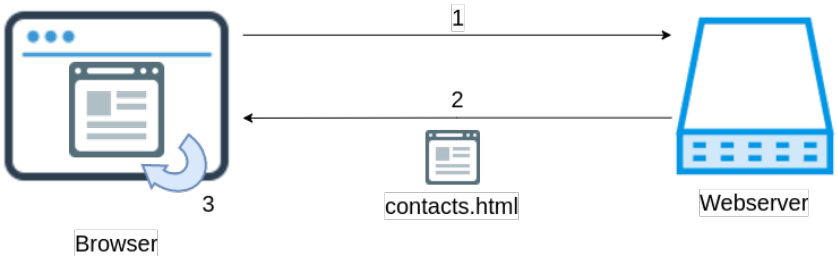
In classical web applications, each page is defined in one single HTML file. Each time a user enters a new URL on the browser location bar, a new request to the server is done in order to get the next page. The server then responds by sending back an entire HTML file (potentially containing processed data fetched from a database in the server side) causing 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). [SR03] This process results in heavier payloads and constant page reloads.

In terms of User Experience (UX), SPA 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 2.2 shows the lifecycles of this two types of web applications.

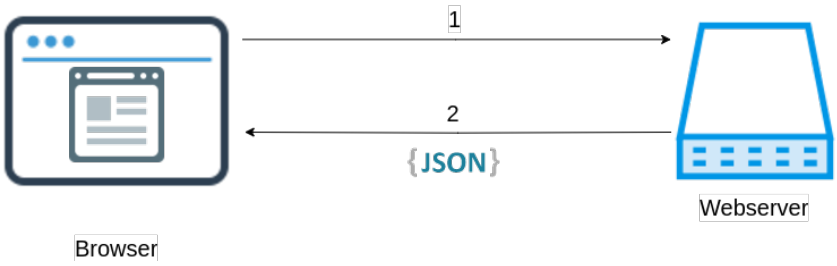
Let's assume the initial page on the browser is main.html. With classical web applications (Figure 2.2(a)), as the user interacts with the page, he clicks on a link that leads to the contacts page. The browser will need to make a new request to the server in order to get this new page content (1). The 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 2.2(b)) 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.

This different approach leads to a whole new set of challenges to measure performance in these applications. But, before we discuss how to measure performance in SPA, the next section describes the performance metrics measured in web applications and current tools that can help us monitor these metrics.



((a)) Classical web applications lifecycle



((b)) Single page applications lifecycle

Figure 2.2: Classical Lifecycle vs. Single-Page Application Lifecycle

2.2 Measuring Performance in Web Applications

As the web evolves, web applications are becoming more and more complex introducing to challenges when it boils down to keep track of metrics, since these can vary significantly depending on the users devices, network and latencies. It is critical to profile current performance to identify where we can achieve the greatest improvements, and in the context of this thesis work, the focus will be frontend performance. According to Steve Souders, there is more potential for improvement on the frontend [Sou07]. If we could reduce the frontend performance by half, overall response times would reduce by 40-45%, as opposed to backend performance, 5-10%. Luckily, frontend performance is no longer an afterthought and there are options to help measure and monitor web applications performance.

This section first introduces performance metrics, giving focus on network time and metrics. We first give a description for each metric and explain how this metric can impact application performance. After, we present some state-of-the-art tools that measure and monitor performance of web applications, and the measurement techniques behind these tools to give a complete picture on how they collect these performance metrics.

2.2.1 Performance Metrics

Back in the day, page load time was the metric of the entire web performance world. Nowadays, it is an insufficient performance benchmark since we are no longer building entire pages, but dynamic and interactive web applications. [Gri16] Nowadays, it is important to measure loading time metrics that can help us understand how and where our

application is spending so much time to load a specific navigation or user interaction. Not all metrics are equally important to measure and, in this section, we focus on listing performance metrics crucial to monitor, when it comes to network-intensive modern web applications.

2.2.1.1 Page load timings

To access timing-related information about a web page in our application, we can use the Navigation Timing API that lets us measure data such as domain lookups, DOM loading metrics and total time loading a page document. This API stores these performance metrics into a list of entries that is accessible using JS. When we load any web page, we can simply open the browser's console and type *window.performance.getEntriesByType("navigation")*. The result is an array with an object of type interface PerformanceNavigationTiming [W3Ca] containing performance timings for the current document.

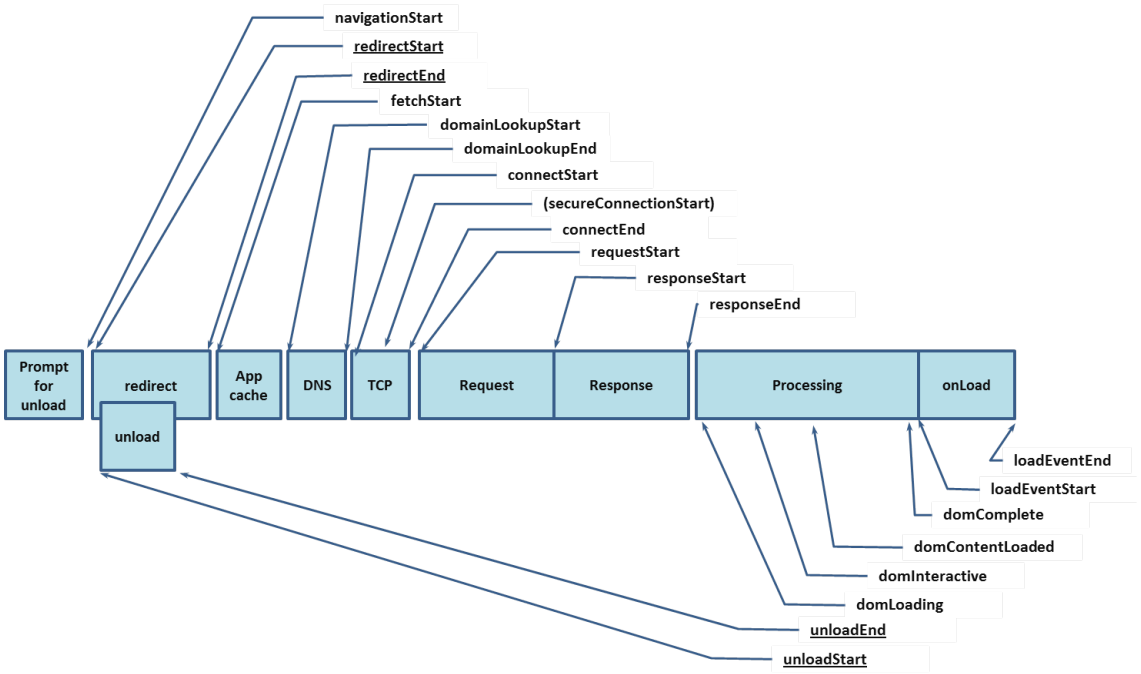


Figure 2.3: Representation of the *PerformanceNavigationTiming* timing attributes, adapted from https://medium.com/dailymotion/real-user-monitoring-1948375f8be5

These performance timing attributes are data points relative to the start of a page navigation (startTime equals 0). As illustrated in Figure 2.3, we can use the different timing attributes, defined by Navigation Timing API, to calculate several navigation load metrics. We now list the most important metrics and how to calculate them using the navigation timing attributes.

- **DNS time** - Total time the browser spent in resolving the request's IP address. This metric is the difference between domainLookupEnd and domainLookupStart, represented as the DNS block.

- **Connection negotiation** - The time spent on making the connection between client and server, prior to sending resources. This connection time metric is the difference between connectEnd and connectStart.

- **Response time** - Measures how long a response takes to finish. This metric represents the duration between receiving the first byte of the response from a server, until the very last byte of that response, i.e. responseEnd - responseStart.

- **Processing time** - As illustrated in Figure 2.3 as the Processing block, this stage is where the browser processes all data, by parsing and request additional subresources (e.g, images) requested by the current document. In this stage, we have these metrics:

  domInteractive - The moment when the browser has finished parsing all of the HTML and the DOM is complete.

  domContentLoaded - This event represents the exact point when both DOM and CSS Object Model (CSSOM) are ready and the render tree can be built.

  domComplete - The moment when all resources on the web page have finished downloading and the current document has been parsed and completed loading.

  The DOM could potentially be a source of bottleneck [Sou09], since browsers spend most of the time building this object model. These metrics are essential to understand the time spent by the browser throughout this processing.

- **onLoad event** - This timing corresponds to the actual moment when a web page is loading, i.e, being visually rendered to the user. The timing for this event is the loadEventEnd metric, but can be further measured as the difference between loadEventStart.

The Navigation Timing API has become a standard when measuring performance of web applications. This API provides a less-intrusive way to capture navigation timings information, giving more visibility into page load timing. For traditional web applications, this API describes a series of milestones during a web page load but, for SPAs, this API is only important for measuring the first-page (i.e, single-page) load. After the *onLoad* event, the browser will no longer trigger that event again, since SPAs navigations do not cause a page to reload, as discussed in Section 2.1.

2.2.1.2 Resources timings

If we want to have the complete picture about our applications performance, it is not enough to measure page load metrics. We should also measure performance for all other resources on a page, such as CSS, JS, AJAX requests, images and third-party assets. As these requests and the data accompanying them increase, so does the amount of time it takes a page to load [Wag17]. For these measurements, we can use the Resource Timing

API. This API allows us to retrieve and analyze a detailed profile of all the critical network timing information for each resource on the page.

Similar to Navigation Timing API, we can also access a list of entries with timing metrics using JS, with *window.performance.getEntriesByType("resource")*. The result is an array with as many objects as the resources loaded from within the current document. Each object is type of PerformanceResourceTiming [W3Cc] which contains a complete set of latency measurements that help us understand the load time for each resource, as illustrated in Figure 2.4. And so, the most important metrics that we can measure are:

- **StartTime** and **FetchStart** - Both of these metrics represent the moment when the resource was queued for fetching. If there are not HTTP redirects, these metrics are the same.

- **Time To First Byte** - This metric represents how long the browser waited for the initial response of the resource requested. And so, we can calculate this metric with the difference between responseStart and startTime. This timing includes both server response time and the time to transfer bytes (i.e, latency).

- **Download Time** - The time spent receiving the response data, responseEnd - responseStart.

- **Duration** - The total time spent on the entire resource lifecycle, i.e, request time plus response time. This metric is the difference between responseEnd and startTime.

- **Request size** - The total resource size in bytes, which is the size of response headers plus the response body.

- **Initiator Type** - This attribute represents the resource type that initiated the resource request. The possible values for this attribute are css, navigation, xmlhttprequest, fetch, beacon or other. For example, if the request is a result of processing a CSS *@import* or *background: url()*, the initiatorType metric would be css. The same logic applies for the other possible values.

With all these metrics provided by the Resource Timing API, we are able to track performance for each resource loaded in a page. This solution is suitable for all web applications, including SPAs, since the array is constantly being updated with all page requests. The only downside to use this solution in SPAs would be to handle a big array with a lot of entries over time. Since navigations do not cause a page to reload, the array will never refresh and so, all requests from the beginning of the first page load, would be in the array. This is something to consider when handling web applications that can be resource-intensive.

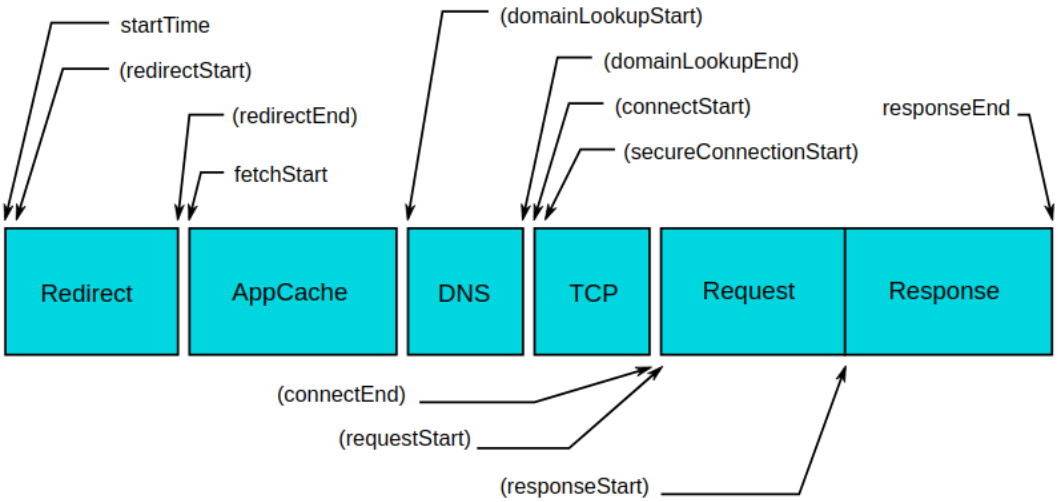


Figure 2.4: Representation of the *PerformanceResourceTiming* timing attributes, adapted from [W3Cc]

2.2.1.3 User-centric metrics

Nowadays, it is important for web applications to be fast. We use these applications everyday and we want fast response times with fluid user experiences. But, when developers test their applications load performance, they usually make misconceptions about the actual load performance for their users. [Wal] These load times can vary from user to user, with different devices and network conditions, and so it is important to measure different moments throughout the load experience. Luckily, there is a Paint Timing API that can help developers to better understand how our application is gradually being presented, i.e, painted, to the user.

The Paint Timing API can be used to measure different moments during a page load. Similar to the previous Performance APIs, presented in the last section, the performance metrics are also accessible using JS. To access these paint metrics we use *window.performance.getEntriesByType("paint")* and the result is two objects of type PerformancePaintTiming [W3Cb], that defines the metrics:

- **First Paint** - The point when the browser renders anything visually different from what was on the screen, prior to some navigation, meaning that the browser is starting to render the page.
- **First Contentful Paint** - The point when the browser first renders any content, such as text, image, canvas or SVG elements.

These paint metrics help developers to better measure performance on real users, and to better understand the different points of the load experience. And for SPAs, the first-page load can potentially result in a performance bottleneck, since it requires to load the entire application code, including the framework, initially. With code splitting, this

entire code can be split into multiple files and be loaded, asynchronously, known as lazy load. This approach have a major impact on first page load performance, resulting in a better UX to the end user. With this paint metrics in place, we should aim to measure these metrics to better understand how users are perceiving the first page load experience.

2.2.1.4 JS Processing Time

When building entire complex SPAs, we often send too much JS to our users. This process can be very expensive since there are a number of different steps a web browser has to take in order to process that JS. First, the JS is fetched over a network, then it needs to be parsed, compiled and executed by the browser. Parsing involves a CPU to process that JS and these times can vary significantly depending on the hardware of a device. During a page load, 10–30% of that time is spent on parse and compile using Chrome's JS engine - V8 [Devf]. Most performance issues, when developing SPAs, come from this initial process time to bootstrap the entire application [Osma]. If a long time is spent in the JS startup process, that delays how soon a user will interact with the application. Developers need to measure this JS processing time, parse and compile times, in order to guarantee faster page loads.

2.2.1.5 Bundle size

With the usage of modern front-end frameworks, one JS development process that changed the way developers code, was static file bundling. Tools such as Webpack achieve this by pulling all project dependencies into one single file, usually called bundle. When we scale this for complex web applications, we can end with a very large bundle but, luckily, techniques such as minification and compression can help decrease this size. Also, to achieve smaller bundles and prioritize resource load, a technique called code splitting allows to split the application code into various bundles which can then be loaded on demand or in parallel, which have a major impact on load time. When it comes to the bundle size, it is extremely important to monitor this metric over time. Every project dependency should be considered and well examined, since there might be lightweight alternatives to the current libraries imported, aiming to deploy less JS.

2.2.2 Measurement Tools

Nowadays, there are a lot of measurement tools that provide a complete picture of the performance in web applications. From open-source libraries to paid products, we can easily set up these tools to start monitoring our applications and how performance is changing over time. These tools offer an opportunity for developers to continuously measure and monitor their applications, ensuring that regressions do not happen and delight users with performant user experiences.

2.2.2.1 Chrome Developer Tools

Over the years, browser's developer tools have been improved to help developers troubleshoot performance issues in applications and one of the most known developer tools is Chrome's DevTools. [Devb] DevTools is a set of web developer and debugging tools built directly in Google Chrome, that can help track down problems quickly, with a whole set of features that are useful when analyzing application performance. Some of these features include inspecting network activity, analyzing runtime performance and even find memory issues that can affect page performance.

By default, DevTools records all network requests in the Network panel. [Basb]. In this panel we can also have access to a waterfall view of all requests activity by the browser, as illusted on Figure. 2.5. For each request, we can access the name, HTTP status, type (script, image, AJAX request), total size and the total duration. We can also view a timing breakdown for each request, where we have more information about the different request phases. These phases are important to know where the browser is spending time, helping developers to better understand the browser's network activity. In the context of this thesis, it is important to understand the different request timing phases:

- **Queueing** - Total time a request was queued by the browser
- **DNS lookup** - Total time the browser spent in resolving the request's IP address
- **Request sent** - The time spent for the browser to send a request
- **Time To First Byte** - Time waiting for the first byte of a response. This timing includes one round trip of latency, plus the time the server took to prepare the response.
- **Content Download** - Total time spent for the browser to download the server response

When it comes to measuring runtime performance, DevTools also includes a Performance panel that help developers to analyze performance runtime metrics to easily identify bottlenecks on their applications. In this panel, we can record different pages, interactions or specific application navigations. In the end, it will display the profiling results containing data such as scripting, rendering and painting metrics, screenshots of the application in different points in time and a flame chart of the browser's main thread activity. [Basa] These and many more metrics are available in the Performance panel, that are worth dive into more detail.

Luckily today, there are many ways to measure performance directly in our browser's developer tools. We have access to all sorts of metrics which can help us troubleshoot performance issues in the front end of our applications and encounter performance bottlenecks.

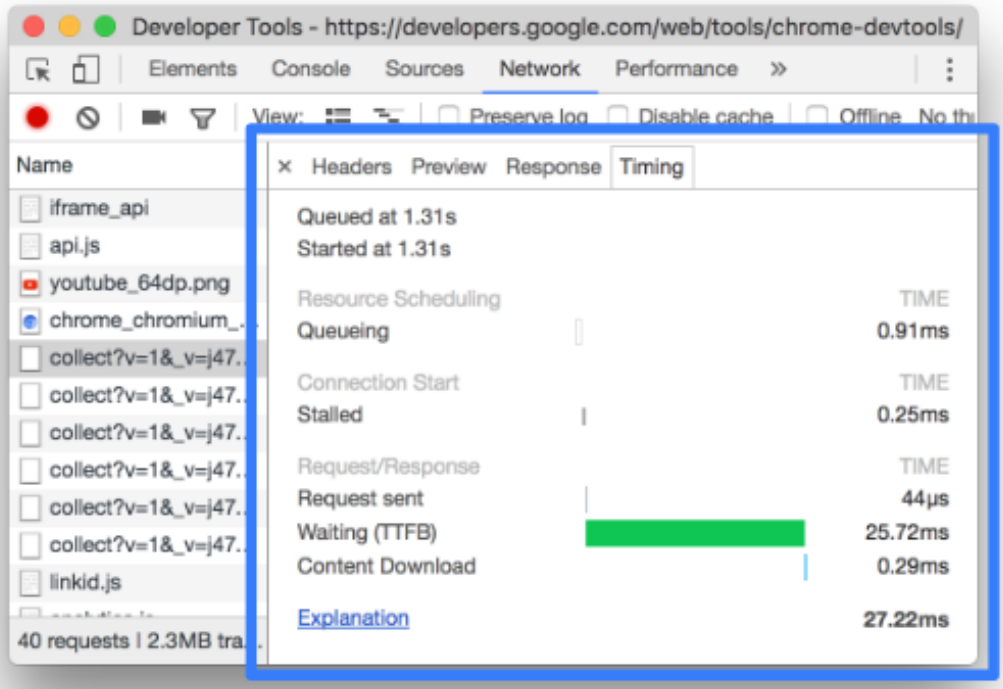


Figure 2.5: The different request timing phases in Timing tab, outlined in blue, available in Chrome DevTools Network panel

2.2.2.2 WebPagetest

WebPagetest [Webb] is probably the most popular tool for measuring performance of web applications. We can run web performance tests in our applications, from a number of different browsers, from a number of different locations across the world, with different connection speeds. There are a lot of options when it comes to configure the test suite, such as the number of runs, connection type and video capture. There is also specific settings for browser Chrome, that can enable capturing Developer Tools timeline and tracing metrics, such as browser engine metrics, CPU tasks, and much more. WebPagetest is available online and it is open-source. Also, it provides a REST API to submit performance tests and receive the corresponding test results. These results provide information including resource loading waterfall charts, page speed optimization checks and suggestions for improvements.

When it comes to performance metrics of a web page, the most important ones that WebPagetest measures are [Rep]:

- **Load Time** - the time from the start of the initial navigation until the *onLoad* event.
- **Fully Loaded** - the time between the start of the initial navigation until network

activity become idle for two seconds after document complete.

- **First Byte** - the time from the start of a navigation until the first byte is received by the browser.
- **Speed Index** - represents how fast the page rendered some visual page content, by measuring different points in time during a page load and calculate the average time.

When combining all these metrics, we can have the complete picture of our application's startup performance. This is important to understand how fast our application initially renders and becomes ready for user interaction but, if we want to measure other specific navigations inside our web application, prior to initial render, this solution is not flexible, since it only measures first page load performance metrics.

2.2.2.3 Lighthouse

Lighthouse [Deve] is a tool developed by Google, that analyzes a web application and collects all sort of performance metrics. It also provides feedback about developer practices and tips on how to fix them in order to achieve better performance. We can run it from a command-line, as a Node module or simply run it in Chrome DevTools. Similar to WebPagetest, presented in Section 2.2.2.2, it runs performance audits against an application URL, and generates a quality report on how good or bad the application is according to web standards and best practices. For each failing audit, Lighthouse provides not only solutions to fix it, but also explains why that audit was important to measure and why you should fix it. When it comes to performance audits, the most interesting metrics that this tool measure are:

- **Has enormous network payloads** - A score between 0 to 100, based on the sum of the total byte size of all resources that a web page requested.
- **JavaScript Bootup time** - Measures the total impact of JS on your page's load performance, e.g. parse, compile, execution and memory cost times.
- **Time to Interactive** - Measures how long it takes a page to become interactive, i.e. some useful content has been displayed, or the page responds to user input within 50ms.
- **Uses an Excessive DOM size** - Measures if the application is shipping a large DOM tree, leading to lots of unnecessary bytes which can result in slow page load times and slow rendering.
- **Keep Server Response Times Low** - This audit measures if the browser waits more than 600ms for the server to respond to a request. It also refers to the time that it

takes for a browser to receive the first byte of page content (also called, Time To First Byte).

These and many more audit metrics are measured when running Lighthouse against a web application. With this tool we can also throttle the network and CPU, and even emulate different mobile devices. This is extremely important if we want to understand how our application is performing in realistic constraints in real world scenarios, and not be limited to normal development tests.

2.2.2.4 SpeedCurve

SpeedCurve [Speb] provides performance insights of web applications by visually monitoring what users are experiencing, highlighted by dashboards that show visualizations of how fast your web applications renders in comparison to competitors. Their focus is to help developers, designers and managers to be aware of how fast their application's user experience is, by getting continuous feedback on the quality of their application code base and how new features development can affect performance.

SpeedCurve's provides a solution for monitoring real user metrics and synthetic metrics (which is built on top of WebPagetest). It measures performance metrics such as number of stylesheets and scripts load asynchronously, average DOM tree depth, bounce rate and session lengths, time from the start of a navigation until the first byte received by the browser (e.g. Time To First Byte metric), CPU long tasks and much more [Spea]. Their solution also enables to create performance budgets and receive alerts whenever these metrics cross some defined thresholds.

2.2.2.5 New Relic

New Relic [New] 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, including SPAs. Their solution for monitoring SPAs is framework-agnostic, which means that it is compatible with most of SPA frameworks, such as AngularJS [Deva], Ember.js [Emb], Backbone.js [Bac], and more.

With a UI that provides detailed information from monitoring real user experiences and interactions, we can easily identify ways to improve customer experience. With a complete view of an application environment, we can understand the performance impact to users and help to analyze entire pitfalls. This service also provides more detailed information about all the activity that produces events, such as AJAX requests and both synchronous and asynchronous JS operations. It also provides AJAX insights, such as group slowest-performing resources and resource-intensive AJAX requests by size, response time and data transfer time. To achieve this, New Relic provides a JS snippet responsible for collecting data and monitoring interactions that can potentially lead to application page loads or route changes, in case for SPAs.

2.2.2.6 Boomerang and mPulse

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, and send beacons back to a server. 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]. mPulse is a performance monitoring solution that loads Boomerang [Boo] in order to monitor and measure web applications performance. Boomerang [Boo] 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.

Boomerang [Boo] methodology is to start to listen for several events from a SPA framework (e.g, $routeChangeStart in AngularJS [Deva]) and, between the start and end of route changes, it will start monitoring for changes in the DOM, tracking any downloads of CSS, JS, images and AJAX requests. If some changes occur, Boomerang [Boo] 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's [Boo] documentation, there are several plugins available for different SPA frameworks, such as AngularJS [Deva], Ember.js [Emb] and Backbone.js [Bac]. When these plugins are enabled, Boomerang [Boo] is able to track all of the SPA navigations beyond the first, initial navigation. This technique uses APIs, such as NavigationTiming and ResourceTiming, to collect this type of performance metrics. Browser metrics are also collected such as screen size, orientation and memory usage, and DOM metrics such as the number of nodes, HTML length, number of images and scripts.

2.3 Summary

Luckily, there are great options for monitoring performance in web applications. These tools help to understand how web applications are performing over time, by measuring all types of performance metrics. All these tools are available to start monitoring our applications, but the options become limited if we want to control and automate measurements for different subsections (*views*) of single-page applications. For example, WebPagetest and Lighthouse just measure a specified URL, usually the main web application endpoint. Solutions such as NewRelic, SpeedCurve or mPulse are be able to measure specific application navigations, but they are paid external services that rely on adding a snippet (e.g., mPulse loads Boomerang [Boo]) into our applications. In the purpose of this thesis, these solutions are not suitable for our work, since we aim to build a tool that must run on premise, without relying on external services and without requiring changes to the source code of the application. In the next chapter, we present our performance measurement solution, which addresses the challenges of measuring different pages and navigations from single-page applications.

Chapter 3

Solution

In this chapter we present our performance measurement solution that addresses the challenges of measuring network-related performance and load metrics of SPAs. This chapter is organized in the following way: In Section 3.1 we describe the overview of the solution and how it is currently used as a tool, at Feedzai. In Section 3.2 we describe the tool architecture. In Section 3.3, we describe Puppeteer, the browser automation API tool used in the solution. Section 3.4 we list the main measurement goals required for measuring performance in SPAs and, in Section 3.5, we describe each performance metric that was implemented in our solution.

3.1 Overview

From the measurement solutions studied in Chapter 2, we know that these solutions are currently measuring performance in web applications, generating reports and visualizations in order to monitor performance metrics over time. These solutions are great options to help developers identify potentially sources of performance bottlenecks. Some solutions are available as paid services, and others rely on adding a JS snippet to our code base, in order to get the performance metrics. Moreover, these tools often receives the entry URL for the application, capturing only the first page load metrics. For SPAs, measuring the first page load is not enough and it is critical to also gather metrics for soft navigations, or user interactions within the page.

Our solution aims to tackle these challenges, by implementing a tool that measures several performance metrics of the application at different time points and combine all data in a single visualization view. This visualization allows developers to easily detect variations over time. The tool must run on premise and automate the process of continuously measure performance in an early stage of the application between developments.

This stage is when developers are constantly developing new features and it is the most important stage to measure performance. With this approach, developers can easily detect regressions and constantly be aware about their applications performance, prior to release them to clients.

Figure 3.1 illustrates how this approach can be implemented in the current development workflow of Feedzai. This workflow is similar to a web application development workflow. Everytime a new application release is planned, the product requirements are defined and the next stage is the features design. Then, the engineering team will be responsible to implement the design proposed. As you are able to deliver the developments of the features, an additional step of testing is necessary to guarantee the quality of the product. This step is critical because it gives some confidence that the new developments do not introduce regressions on existing features. This Testing step is carried out in parallel with the implementation, since it continuously runs all the application tests - unit and integration tests. Before the release, a team is responsible to test the features and to make sure everything is aligned and implemented as proposed.

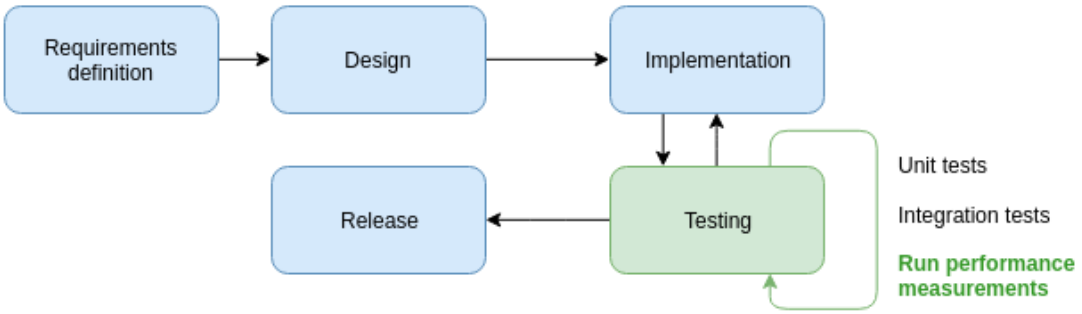


Figure 3.1: Feedzai development workflow and solution approach

In this workflow, it makes sense to add an additional step in the Testing phase, to perform performance tests. Here, the application will be the target of metrics measurement, and later their evolution will be evaluated in the visualization component. Everytime a developer finish developing code for a new feature, not only the existing tests will run but our solution tool will execute and gather new performance metrics. This can inform the development team of the impact on the new features performance to make a more informal decision about making such features (immediately) available in production.

3.2 Architecture

The solution that we've built should measure performance and capture metrics from a web application. In the context of this thesis, the focus are single-page web applications. The methodology design is divided into three components: a command-line interface, a browser automation tool and the visualization component. Figure 3.2, illustrates how these components interact with each other. We now describe briefly each component:

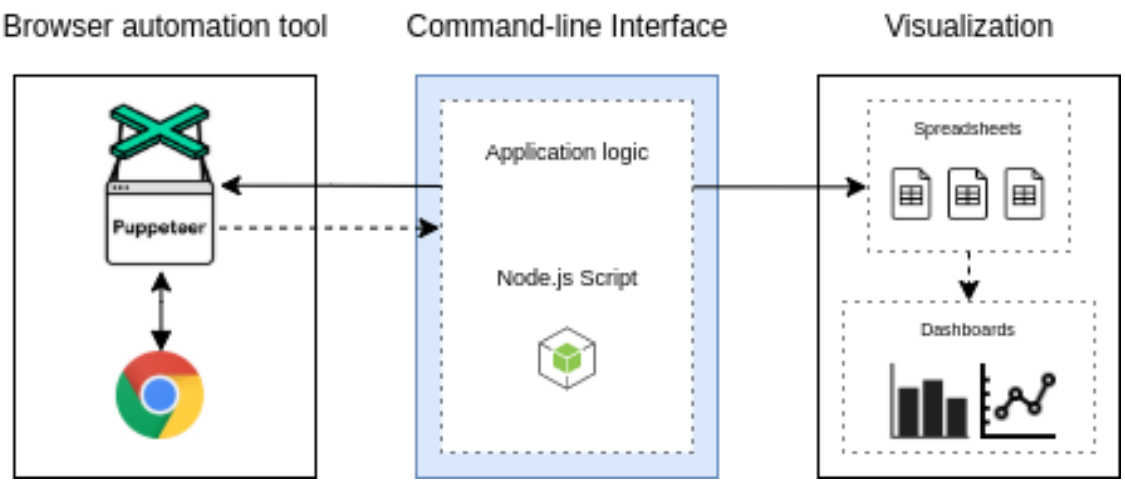


Figure 3.2: Solution architecture

- **Browser automation tool**: This interface is responsible for creating an instance of a web browser and automating browser tasks such as opening a web application, navigating to a certain page in the application or clicking a button. This automation tool will perform a set of operations as requested by the Command-Line Interface (CLI), against the web application for testing. The browser automation tool that we are using in this solution is Puppeteer, a API that controls Google Chrome browser, presented in Section 3.3.

- **Command-Line Interface**: This interface is the main component and the main logic lies on the execution of a script file, implemented in Node.js, that enable us to run code directly in a browser instance. This browser instance is created using the browser automation API described above. The script contains a set of commands that allows the automation of tasks that we can perform in the browser, thus controlling the application and interactions in it. This CLI supports a series of input commands that will be further described in Section 4.1 but the main ones are: running custom configurations, output reports in JSON format, and submit results to Google Sheets spreadsheets, using the Google Sheets API. These results are JSON files containing all performance metrics from the application measured during the CLI execution. These metrics are listed in Section 3.5.

- **Visualization**: This component results after the submission of the metrics reports generated by the CLI. This interface communicates with the Google Sheets API, to publish the metrics in spreadsheets. In other tabs of these sheets, dashboards are created using the submitted data. This process is therefore automated and allows the developer to have access to a user-friendly visualization interface with a data table and charts (line charts, histograms, etc). Thus, to quickly analyse performance variations over time and identify bottlenecks on their applications.

3.3 Puppeteer

Puppeteer is a library that provides a high-level API to control Chrome's browser, or Chromium (open-source version), over the DevTools protocol. This protocol is the API responsible to instrument, inspect and debugging Chrome. On top of all these components, we can create our own automation script, that can do all sort of browser tasks. Most of these tasks we can do manually in the browser, we can use Puppeteer to do that for us. Such as crawl a web application, automate form submission, keyboard input, and more.

In the context of this thesis, the most interesting feature of Puppeteer is the ability to capture timeline trace to help diagnose performance issues. If we enable tracing, we can have access to the entire Devtools trace file that contains all the application activity during execution. This activity is the information we see on the Performance panel, as presented in Section 2.2.2.1. With this, we can analyze all events that occured during page load, user interactions, JS execution and network requests. This is helpful when it comes to the challenges of measuring performance in SPAs, described in Section 1.1, as there is no standard mechanism to track navigations nor user interactions in these type of web applications. There are two Puppeteer package installation alternatives:

- *puppeteer* bundles the latest version of Chrome, with no setup configuration required. When we first run it, it downloads the most recent version of the browser available that is guaranteed to work with the API.
- By installing the *puppeteer-core* package, we can prevent this download. This package does not automatically download Chromium when installed and it is useful if the goal is to simply develop a library on top of the DevTools protocol.

However, another alternative to skip the browser download, using *puppeteer*, is to launch an existing browser in our machine, or even connecting to a remote one. By default, Puppeteer runs a browser instance in headless mode, but it can also be configured to run in headfull mode - where we would be able to actually see Chrome's browser UI.

```
1  const puppeteer = require('puppeteer');
2
3  (async () => {
4    const browser = await puppeteer.launch();
5    const page = await browser.newPage();
6    await page.goto('https://www.example.com');
7
8    await browser.close();
9  })();
```

Listing 3.1: Example to open a web page using Puppeteer API

The listing 3.1 is an example of using Puppeteer to automate opening a web page. When Puppeteer connects to a browser instance, a Browser object is created. We can then use this object to create a new Page object. The Page object provides methods to

interact with a single tab. In the example, we open the page on *'https://www.example.com'*. Moreover, a Browser instance can have multiple Page instances in order to test multiple pages.

Puppeteer is a very complete browser automation tool solution, and that is why we chose it to be implemented in our framework solution. Not only it offers a great high-level API to control Google Chrome, but has lots of documentation and examples throughout the web.

3.4 Measurement Goals

Measuring performance provides an important metric to help assess the success of a web application. The measurement goals for our solution were defined taking into account what was possible to obtain from current performance measurement solutions, but also solving the challenges of measuring SPA navigations. In the context of a company, with constant development of new features in its applications, we aim to determine how a web application performs across different product releases. If there is a baseline of performance metrics for each product release, the development team may be able to assess whether performance is improving, getting worse, and whether or not actions are needed to improve a certain navigation load time, etc. The metrics to measure should be relevant to understand the different aspects of the application, the team that develops it, and business goals. The metrics proposed by our solution are presented in the next section, Section 3.5. They should be collected and measured in a consistent manner and analyzed in a format that can be consumed and understood by non-technical stakeholders [Mozb]. The level of performance for a web application correlates to the end-user experience, with measurable effect on user engagement. This is why companies should measure and monitor their applications, because a happy customer, plays a major role in the success of products.

3.5 Performance Metrics

In this section, we describe and discuss the most important metrics when it comes down to measure performance in SPAs. Not only do we define the meaning of metrics and respective calculations according to the Performance APIs, defined in Section 2.2.1, but we also define a new methodology for measuring new metrics in SPAs.

3.5.1 Full page load time

In Section 2.2.1.2, we describe the different time points of a given resource loaded by a page, accessed by the Resource Timing API. The most important points were identified for performing metric calculations for each loaded resource. In the first page load, if we access the total set of resources loaded, we can have a complete picture of the network

activity in this instant - first page load. Since we cannot rely on the *onLoad* instant, for the measurement of the whole page load in SPAs, as described in Section 1.1, a new approach is necessary.

This approach considers analyzing not only loaded resources before the onLoad moment, but resources loaded after. This metric, which we have called full page load, considers all the synchronous and asynchronous loaded resources, and its value is equal to the moment from the last resource loaded by the first page. With this, we have the whole picture of the network activity and the actual bootup time of a single-page web application. To understand which resource was loaded last, we have to dive deep how resource times are calculated.

A resource total time to load, or duration, in milliseconds, is equal to the difference between the last load point of the resource, responseEnd, and the first load point, startTime, as described from the Resource Timing API, on Section 2.2.1.2. It is important to notice that, the last loaded resource is not always the last to finish loading. If resource 1 with a startTime equal to 1000 and responseEnd equal to 2000, and resource 2 with a startTime equal to 1200 but a responseEnd equal to 1800, this means that resource 1 was the last resource to finish load, altough it started to load before resource 2. With this in mind, the last loaded resource is the one with a maximum value for startTime + duration. As stated before in this section, this value will be the actual value for the full page load time metric, in milliseconds. Figure 3.3 represents this metric, compared to domContentLoaded and the onLoad event. As illustrated, the full page load metric captures the entire moment until the very last resource finish to load.

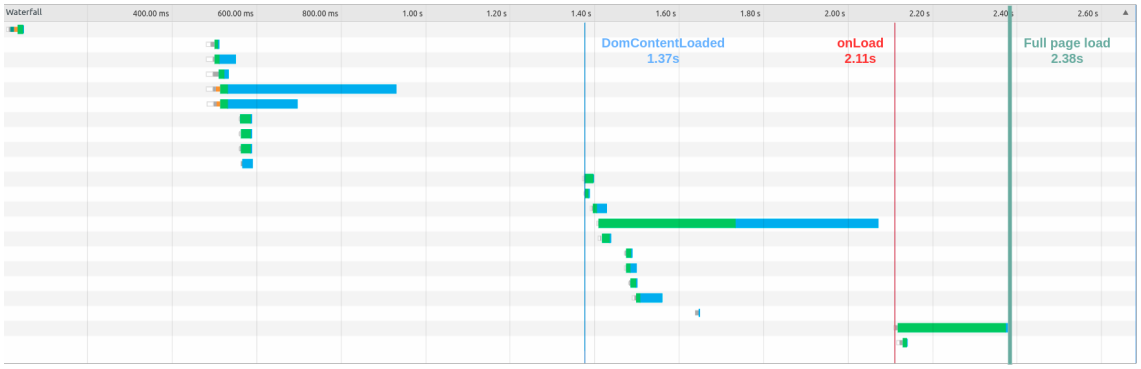


Figure 3.3: Full page load time representation, in green (on the right), compared to domContentLoaded and onLoad values

3.5.2 Bundle metrics

In Section 2.2.1.5, we discussed that the application bundle can be a possible source of performance bottleneck. Not only due to its size, but directly related to JS processing time. This JS file, contains all the logic of the application - the SPA framework and external dependencies, such as other JS libraries. We also observed in Section 2.2.1.5 that

the bundle can be split into multiple smaller files to improve the first page load. And as resources are needed in different navigation or application routes, files (also called, modules) with the necessary logic are loaded. Regardless of the improvements to the bundle, it is always important to monitor its size, but also its processing time. Therefore, our solution implements three metrics to measure the bundle - size, parse and evaluation times.

To retrieve these values, we access the trace file, generated from enabling DevTools tracing, from the Puppeteer API, as described in Section 3.3. After the CLI executes, a JSON file is generated, with all the information of the application activity. Through JS, we can filter and search values in a JSON object. And our approach to get the values from the bundle, is to filter exactly the entries corresponding to it. In the trace file, to get the parse value of any script, we can filter entries name - *v8.parseOnBackground*. Then we can filter by the URL of the script, which in this case is the URL of the application bundle. In this object there are different key-value entries, and one of them is the duration of the event. This duration is identified by the *dur* key and the value is given in microseconds. Since this time measurement is not perceptible for analysis, we divide the value by 1000, thus leaving the result in milliseconds. The same applies to the evaluation time of a script, but in this case, entries with name *EvaluateScript* are filtered. The bundle size is retrieved by filtering the *ResourceFinish* entries for the bundle request ID, and the value is equal to the *encodedDataLength*, in bytes.

3.5.3 Paint timings

In Section 2.2.1.3, we realize that there are some misconceptions when it comes to load performance for the application users. The load times can vary from user to user, with different devices and network conditions. Therefore, it was concluded that it is necessary to measure different points throughout the application first load experience.

With the Paint Timing API, it is possible to consult two values that help to understand how the application is being gradually presented, i.e., painted, to the user. These values are the first paint, and the first contentful paint. The first paint is the time when the browser starts to render anything that is visually different from what was on the screen prior to navigation [Moza]. The first contentful paint is the time it when the browser starts to render the first piece of DOM content prior to navigation. In this case, DOM content can be images, non-white <canvas> elements, or SVGs [Devd]. Figure 3.4 represents these two paint metrics, when loading a web page.

Additionally, and to further enrich the information about the user load experience, our solution also proposes to add the first meaningful paint metric, also represented in Figure 3.4. This metric is available through the Puppeteer API and like other performance metrics that can be obtained through this API, all are obtained from a certain point in time, the navigationStart, which defines the first navigation time on the page. Thus, calculating the difference between the *firstMeaningfulPaint* and *navigationStart* instants,

Figure 3.4: Representation of three paint metrics, First Paint, First Contentful Paint and First Meaningful Paint - adapted [Osmb]

we get the final metric value in milliseconds. This metric measures when the primary content of a page is visible to the user, with a big layout change. It represents when the users feel they can interact with the page.

With these three metrics, we can efficiently measure three different points throughout the entire load experience for one user. This allows us to determine target points for improvement, not only in the web application experience itself, but also in improving the first page load overall.

3.5.4 Browser Script and Task durations

Modern browsers today, under the hood, are an entire operating system with hundreds of components. These components such as parsing, layout, style calculation of HTML and CSS, JS execution, and more [Gri16], all play a critical role on the overall performance of the browser, and consequently on the web applications it runs.

The browser is responsible for some operations, when loading a web application, and divides this work into threads. On the main thread, operations such as JS execution, construction of DOM and processing user input, block browser responsiveness. This has implications for the user's load experience, because if the browser takes a long time in these operations, it will take longer for the application to be interactive. Other operations that can happen on other threads, allowing the browser to stay responsive are parsing HTML and CSS, parsing and compiling JS, JS garbage collection tasks, parsing images, and more. [Rus]

We now understand that the browser is responsible for the web application's own performance and that is why it is critical to measure its own performance, and the time it takes to execute its operations. With Puppeteer API, we were able to obtain the metrics that correspond to the combined duration of JS execution and the combined duration of all

tasks performed by the browser, on the first-page load. These performance metrics make it possible to understand how long the browser is taking to process the web application, and to understand target points for improvement.

3.5.5 Soft navigation metrics

From Section 1.1, we conclude that the traditional way of measuring navigations between pages does not apply to soft navigations. That there is no standard mechanism to measure the time required to fetch content while performing these navigations. As the user navigates and interacts with the page, the SPA 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, over the network. Our approach is to measure all these asynchronous resources, knowing the number of resources, how long it took to be loaded by the server and the total size of bytes transferred over the network.

The technique used to solve this problem was network interception. With Puppeteer API, we can intercept each request made by the network and perform a set of operations such as changing the headers, aborting the request, and more [Pup]. Listing 3.2 shows how to enable request interception with Puppeteer, in JS. For each intercepted request, ie when it is issued by the web application, it is possible to obtain some important data, listed in the next paragraph, but when the request ends, no data is returned, which makes it impossible to calculate the final load metrics for a request. The next paragraph explains how we solved this problem.

```
1  (async () => {
2      page._client.on("Network.requestWillBeSent", (requestData) => {
3          // Intercept all requests over network ...
4      });
5      page._client.on("Network.loadingFinished", () => {
6          // Intercept all finished requests ...
7      });
8  })();
```

Listing 3.2: Network requests interception with Puppeteer

After the first page load, each request made by our application is recorded in a data structure for each navigation performed. For each intercepted request, we record the data obtained: its timestamp, which corresponds to its first load time instant (such as startTime instant from the Resource Timing API), the request ID, the request type (fetch, image, font, ...) and the url itself. This request ID corresponds to the same ID that uniquely identifies each request in the DevTools trace file. With this, it is possible to map this data together with the data from the trace file and thus determine the final load metrics of a request, as identified in Section 2.2.2.1.

For each navigation, all intercepted requests will be recorded until the network becomes idle for 800 milliseconds If during this time, no more requests are made, we assume

the soft navigation finish, because all the necessary content for the new view has been loaded. Our solution also takes into account navigations whose content is constantly being updated, using polling. In this case, it is possible to configure a custom timeout that will define the time window in which requests are intercepted. This method is defined in pseudocode, on Listing 3.3.

```
/**
 * Method that intercepts all network requests and resolves when the network
 * becomes idle after a timeout (800 milliseconds)
 * @returns {Promise}
 */
waitForNetworkIdle() {
    const timeout = 800;
    const navigationRequests = new Set();
    let inflight = 0;
    let timeoutId = setTimeout(onTimeoutDone, timeout);
    const promise = new Promise((x) => resolve = x);

    page._client.on("Network.requestWillBeSent", onRequestStarted);
    page._client.on("Network.loadingFinished", onRequestFinished);
    return promise;

    function onRequestStarted(request) {
        // Save request metrics (first load time instant) on data structure
        // Increment inflight

        if (inflight > 0) // Network is not idle
            clearTimeout(timeoutId);
    }

    function onRequestFinished() {
        if (inflight === 0) return;

        // Decrement inflight

        if (inflight === 0) // Restart timeout since network is idle again
            timeoutId = setTimeout(onTimeoutDone, timeout);
    }

    function onTimeoutDone() {
        page._client.removeListener("Network.requestWillBeSent", onRequestStarted);
        page._client.removeListener("Network.loadingFinished", onRequestFinished);
        resolve(); // Resolve promise
    }

}
```

Listing 3.3: Pseudo code for waitForNetworkIdle function

Thus having access to the times of each resource loaded for each navigation, we were

able to determine the total time spent in loading the content for the creation of the new view. This loading time allows us to have a more complete perception of the navigation load time. Our solution considers a metric that corresponds to the time interval from the first resource to be loaded, until the last resource loaded until the network is idle during the previously defined timeout. Its value is calculated by the difference between the endTime of the last request and the startTime of the first request. This metric, which we call *navigationFullInsight*, determines the total time spent on network activity during navigation, in milliseconds. In our approach, we define this metric as the time of soft navigation itself. Figure 3.5 represents this metric.

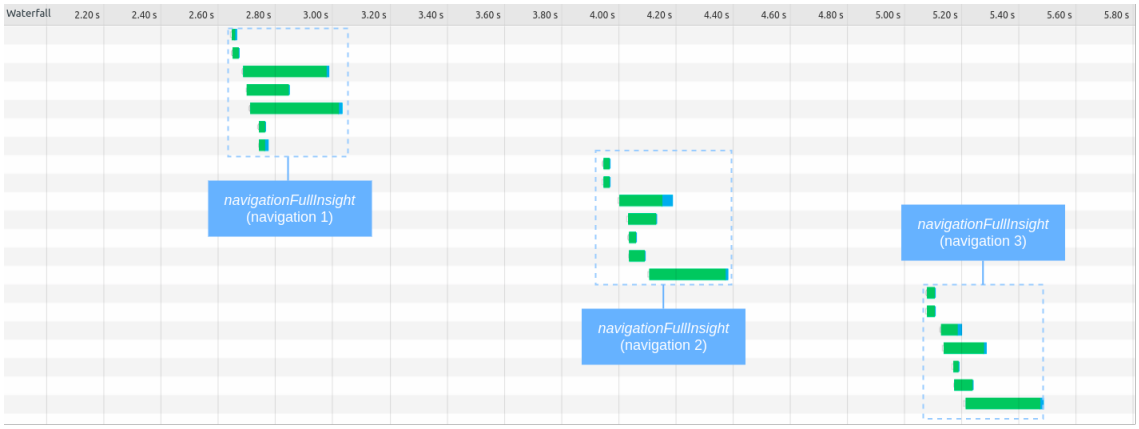


Figure 3.5: Representation of *navigationFullInsight* metric, in three different soft navigations

In order to complement the information about each navigation, we consider other metrics such as the total number of asynchronous requests - *numAjaxRequests* -, the average load time of these requests - *avgTimePerAjaxRequest* -, and the total number of bytes transferred over the network, of these requests - *totalAjaxBytesSize*. These metrics are possible to obtain, because we record the requests intercepted by the navigation, and we have the time load data, respectively.

3.5.6 Navigation resources metrics

In the previous section we described the metrics that can help to measure the load time of a soft navigation, in a high level way. But as we have access to each resource loaded by navigation, through network interception, we can try to better understand the different loading timeline for each resource. With the mapping between the network interception and the data in the application activity trace file, we were able to obtain three important metrics.

- **queueingTime** - Total time a request was queued by the browser
- **timeToFirstByte** - Time waiting for the first byte of a response.

- **contentDownloadTime** - Total time spent for the browser to download the server response

With these metrics it is possible to outline the different load points of a request and determine if a given resource is taking a long time to be processed by the browser, what the server's response time is and the time spent to process the response with the content of the view. As a performance metric, we also added the total load and processing time of the request - *requestDuration* -, as well as the total number of bytes transferred from the response - *requestSize*.

3.6 Summary

Although current solutions manage to return some performance metrics on web applications, not all implement those that apply in the new paradigm of these applications, single-page applications. We believe that we have developed a viable solution that manages, in a flexible and easy configuration setup, to return useful performance metrics over any SPA, in an agnostic way. The architecture of this tool consists of a library that connects to a browser automation API, Puppeteer, in order to perform a set of navigations or interactions, which are intended to be monitored. In addition to these navigations, metrics of first page load, application bundle, processing tasks performed by the browser are critical to understand application loading. To complement the results generated by the program, it is possible to connect to the Google Sheets API in order to save the tabulated values. From this data, it is possible to create in this application, dashboards with graphs that show the evolution of metrics over time. The next chapter, Chapter 4, contains all the information related to the implementation of this automation logic, and how it applies in the context of Feedzai and its current development workflow for single-page applications.

Chapter 4

Implementation

In this chapter we present our library implemented in Node.js. This library was made available as a NPM package in Feedzai's internal NPM repository. This library contains the logic needed to run the components described in Section 3.2, responsible for the connection with Puppeteer API, and in the creation and publication of results with the performance metrics, listed in Section 3.5, in spreadsheets, through the Google Sheets API.

Section 4.1 provides an overview of the main component, the command-line interface, describing the application logic in detail, configuration setup for a web application and listing the most important commands currently supported by this interface. Then, in Section 4.2, we describe our approach in the implementation of automated tests, in the integration with Feedzai's current Quality Assurance (QA) pipeline, and in the publication of metrics for monitoring. Finally in Section 4.3 we describe the approach in creating the respective spreadsheets for the publication of the results, and in creating the dashboards.

4.1 Command-Line Interface

The implementation is currently available as an NPM package, published internally in Feedzai's repository manager. This decision was mainly due to the need to test the library, on Feedzai's different products, before its launch as open-source library. The final goal, which would be to make this project open-source, has been postponed in order to take the necessary code and documentation improvements.

The main logic of this library is contained in an interface, a CLI, which supports different commands, given as input. These commands are listed in Section 4.1.4. The CLI is responsible for reading the commands and executing a script, implemented in Node.js, that instantiates a JS class that contains all the execution logic. The CLI is

Figure 4.1: Project directory tree

also responsible for reading a configuration file with data from the web application, as explained in Section 4.1.3.

The implementation of this interface, from reading the input to the script and helpers files, contains about 3400 lines of code. Of this total, the main script executed represents 700 lines of code. Figure 4.1 shows the directory tree seen from the root repository. This project was built as a NPM package, using *npm init*.

We now provide a high level description of the contents of each directory:

- **bin:** This folder contains a single file, cli.js, which is the script initially executed by the CLI, responsible for reading the input commands, data on the command line, and for instantiating the main script.
- **src:** This folder contains all the logic of the application. It contains helpers files, contains the scripts for connection and authentication with the Google API (Google Sheets and Google Drive), and contains the main script, responsible for instantiating an automated browser, using Puppeteer API.
- **node_modules:** This folder is part of the NPM installation that contains all the dependencies of the project.
- **results:** This directory contains the results with the performance metrics and DevTools trace file from the script output. This folder is only created at runtime, if we run the command to save the reports locally on our machine.

To run the interface, it is necessary to access a command line and have Node installed on the machine. Then to run the program, just type *node bin/cli.js –command*. It is also possible to create a symbolic link, from *cli.js* file in the *bin* folder, using *npm link*. When running, we can omit *node bin/cli.js –command* and write only *puppetzai –command*.

When running *npm init*, the command that sets up a new NPM package, creates a package.json file. This file contains the list of all project dependencies, along with the installed versions, and other information about the repository, such as the name, current version, description, registry. In our implementation we added a code quality validation script, eslint, which is a static code analysis tool for identifying problematic patterns in JS code. These quality standards follow those defined by Feedzai, with its

open-source repository [Feeb], where our file in the repository root - *.eslintrc* - extends this configuration.

4.1.1 puppetzai.js File

The main script executed by CLI, which contains the main logic of our solution, is a module named *puppetzai.js*, located in the *src* folder. After reading the input commands, to be processed by the CLI, a JS class is instantiated that receives these commands, and which also receives the application configuration, described in Section 4.1.3. Initially, the connection is made with the Puppeteer API, in creation of the browser object and page, where the instrumentation of operations will be performed. Before this instrumentation, page tracing is activated, so that, at the end of the execution, we have access to the data recorded by DevTools, with all the activity of the application.

The logic of the main execution method in this class, the *run* method, is responsible for opening the page at the endpoint of the application and instrumenting the different actions defined in the interface configuration. These actions are, for example, filling in the application's login form, if any, and performing the different navigations and/or interactions.

After the browser page loads the first (and single, because it's a SPA) application page, we calculate the main metrics for the first-page load - Listed in Sections 3.5.1, 3.5.2, 3.5.3 and 3.5.4. Then, for each navigation defined, we instrument the action that triggers the next navigation, and stores the network activity (ie, the requests loaded), indexed by the current navigation url. After performing all the configured operations, the metrics of these navigations and additional ones are calculated, as described below, for the creation of the final report.

Finally, the last method to be executed by this main script, is the *recordMetrics* method. This method is responsible for calculating the metrics from the DevTools trace file - Listed in Sections 3.5.5 and 3.5.6. In the end, when creating the JSON file with the final performance metrics results, we also added metadata about the current measurement, such as the timestamp, the current date and the browser version. This data is crucial in identifying the reports, over time, as well as in viewing the final dashboards.

This script is also responsible for integration with Google Sheets and Drive APIs. After calculating the final performance metrics and creating the final JSON object, these results are published in these tools. This integration is explained in more detail in Section 4.1.2.

4.1.2 Google APIs OAuth

Our approach allows results to be published on Google Sheets and Google Drive. This solution allows automation in the visualization of results and their reliability. Google Drive is used to store reports and Google Sheets to publish results in spreadsheets. The implementation of this visualization component is described in Section 4.3.

Figure 4.2: Folders for Google API integration

In our solution, the scripts responsible for integrating these Google APIs are distributed in three folders within the *src* directory, as shown in Figure 4.2.

In the *oauth* folder, lies the module responsible for the OAuth protocol, in the authentication and authorization of our tool to have data access by Google APIs, more specifically Drive and Sheets APIs. For this integration to happen, it is necessary to create a Google project [Devc], enable both APIs and download the credentials. These credentials are added to the configuration file, under the *clientSecret* property, as described in Section 4.1.3.

The *submit* and *upload* folder are responsible for the integration with the Sheets and Drive APIs, respectively. Each of these folders has a module - *index.js* - which is responsible for the API authentication and, in the case of the Google Sheets module, submitting the metrics results in a certain spreadsheet defined in the configuration file, under the property *sheets*, as described in Section 4.1.3. In the case of Google Drive, the respective module is responsible for submitting the JSON file and the DevTools trace file, within a folder defined in the configuration file, in the property *drive*, described in Section 4.1.3. These files are uploaded to a subfolder, whose folder name is the timestamp of the measurement date.

4.1.3 Configuration file

Our implementation aims to be an agnostic solution with regard to SPAs and their frameworks. This solution involves implementing a tool that allows it to be configured for each application. As each web application is different, with different routes and interactions, it was necessary to create a configuration file with application data. This file is a module that should be created from scratch for a new application and that exports an object in JSON, as shown in Listing 4.1, with the custom configuration.

We now proceed to explain the configuration supported in this file:

- **puppeteer:** Here you can find the browser automation configuration, using the Puppeteer API. All options supported by this API are supported within this configuration, as long as it complies with the Puppeteer documentation, more specifically in the supported options of the *puppeteer.launch(options)* method [Ppt]. Bear in mind to check the proper documentation for the version of the API used.

- **application:** All configuration related to the application is added within this property. It is supported:
  - **url**: A string that represents the endpoint of the application's main page. This field is required.
  - **bundle**: A string with the name of the main SPA bundle. This field is required.
  - **login**: If there is an authentication form, login, on the SPA main page, this field is necessary to automate the submission of this login form. Therefore, this field is optional. If used, this field is an object with the following fields, all mandatory:
    - **username:** A string that represents a valid username.
    - **password:** A string that represents the respective password for the above username, in plain text.
    - **fields:** An object that defines the selectors of the login form. Selectors are CSS selectors, which use CSS rules to find the element in the DOM.
      - **username:** A string that represents the selector for the username field.
      - **password:** A string that represents the selector for the password field.
      - **navigationSelector:** This element is normally the form's submit button, and contains a string for the button's selector. It is the trigger for the application's first soft navigation, that's why the name *navigationSelector*.
  - **pageUrls**: This field is a list, or array, of the different navigations and/or interactions of the application, which will be targeted in measuring performance in SPAs. Each navigation is represented by an object with the following fields:
    - **url:** This field represents the endpoint that identifies a navigation.
    - **navigationSelector:** Similar to the *login.fields* field, this is the selector of the element responsible for triggering the navigation. This element can be any element in the DOM, which is currently clickable in the current state of the application. It is represented as a string.
    - **hover:** If the *navigationSelector* described above is not clickable (eg. it is a sub-item hidden within a menu), we can use this field to activate an element up in the hierarchy, which is responsible for showing the navigation selector. This field is optional.
    - **navigationTimeout:** As mentioned in Section 3.5.5, our solution also takes into account navigations whose content is constantly being updated, using polling. This means that the promise for a navigation will never resolved, because the network will never be idle. To fill this particularity, this field has been added for navigations that use this functionality. Case Manager and Pulse implement polling in certain soft navigations, for example.

- **clientSecret:** This property corresponds to the configuration of a Google project, where our tool will make the connection to make requests for the supported Google APIs, Drive and Sheets. In here, lies the OAuth2 credentials such as the project id and client secret [Oau] used for authentication by the oauth protocol. This field is mandatory if we want to publish the results using one of the supported APIs.
- **sheets:** When running the CLI, if we choose to publish the results using the Google Sheets API, this property is mandatory. It is an object with the following fields:
  - **spreadsheetId:** ID of the spreadsheet where we want the results to be published. This field is mandatory to perform the API request.
  - **tableName:** Name of the sheet in the spreadsheet indicated above. This field is also mandatory.
- **drive:** When executing the CLI, with the command *upload* enabled, described in Section 4.1.4, this property is mandatory in the configuration file. It is an object currently with a single field, *folderId*. This field corresponds to the folder ID where we want to store the output reports generated by our tool. This ID is required for the Google Drive API request.

```
module.exports = {
    puppeteer: {
        headless: true,
        devtools: false,
        defaultViewport: {
            width: 1920,
            height: 1080
        },
        // More puppeteer options...
    },
    application: {
        url: "http://application-endpoint:port",
        bundle: "bundle.js",
        login: {
            // Login credentials
            username: "...",
            password: "..."

            // Login form
            fields: {
                username: "...",
                password: "..."
            },
            navigationSelector: "..."
        },
        pageUrls: [
            {
```

```
28              url: "navigation1",
29              navigationSelector: "..."
30          },
31          {
32              url: "navigation2",
33              navigationSelector: "...",
34              navigationTimeout: 1000
35          },
36          // more navigations ...
37      ]
38  },
39  clientSecret: {
40      installed: {
41          // Google project configuration
42      }
43  },
44  sheets: {
45      spreadsheetId: "...",
46      tableName: "master"
47  },
48  drive: {
49      folderId: "..."
50  }
51 }
```

Listing 4.1: Example of a configuration file

4.1.4 Supported Commands

As an interface that processes user input commands, the CLI currently supports the following commands to be executed:

- **–config <config-file-path>:** Sets the path to the application's configuration file.
- **–output <output-path>:** Outputs results in JSON format on a custom directory. This command can be abbreviated by simply using *-o*.
- **–login:** Fills and submits the login form, after *first-page* load. This command should be used for SPAs with a login form on the home page. When using this command, the *login* property in the configuration file is mandatory. This command can be shortened to *-l*.
- **–tag <tagName>:** Adds an additional tag property to the output file, on the report metadata. This tag is useful, say, for the commit ID of the target application's repository, or to identify a special measurement run. This command can be shortened to *-t*.

- **–screenshot:** Save screenshots of the current application page, in JPG format, after some navigation. If enabled, a new directory *screenshots* is created at the root of the repository. Can be shortened to *-p*.
- **–debug:** If used, the browser automation tool will run a headfull Chrome, with the UI, instead of the default headless mode. This is quite useful to see how the browser automation tool operates and check for application errors. Can be shortened to *-d*.
- **–upload:** Upload JSON files with output results, and DevTools trace file to a folder on Google Drive. Can be shortened to *-u*.
- **–submit:** Submits performance metrics results to a spreadsheet on Google Sheets. Can be shortened to *-s*.

4.2 Results automation on Feedzai's QA pipeline

At Feedzai, its products are SPAs with a constant development of new features, in order to support the needs of its customers and the evolution of the product itself. Because of this, there was not only the need to obtain data with performance metrics about the applications, but also in the automation of this whole process. In this section, we describe the integration of our solution in Feedzai's current test automation pipeline, and the process of visualizing the data obtained by our tool.

4.2.1 Feedzai QA pipeline overview

Feedzai currently performs all test automation on all of its products on one platform, called Jenkins. Jenkins is a CI server, which builds and tests software continuously, monitoring the execution and status of jobs. One of the advantages of using this tool, is to provide a faster feedback and a real time knowledge on the state of the work being developed across applications.

Both Pulse and Case Manager products run their automated tests on this platform, each with its own pipeline and different steps from checking the application repository, compiling and building dependencies, and executing the tests themselves. Figure 4.3 illustrates an overview of the current pipeline for Case Manager application. After analyzing the current pipeline, there are jobs that run in parallel, whose execution is independent of other steps, and also in order to improve the total execution time of the pipeline itself.

In order to support a wide range of clients, each with different needs in implementing solutions to combat financial fraud, Feedzai supports different versions for each of its products. Each version of the product has its own development branch, a hotfix branch, and developments for new features are launched on the master branch. In the CI server, different pipelines will be configured for different product versions, or repository branches. For example, in the Case Manager application, there is a pipeline for

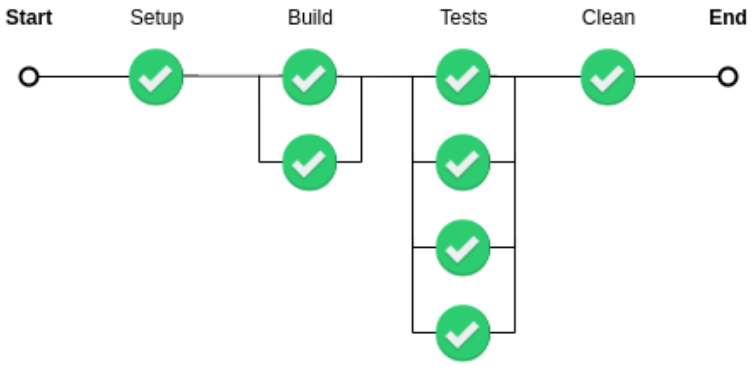


Figure 4.3: Overview of Case Manager CI pipeline steps

each version 10.1 and 10.0. In addition, there is also a pipeline for the master version. Each of these versions may result in a new product release, taking into account internal regulations and contracts with customers.

In order to monitor the performance metrics for each product version and build a comparison baseline between releases, integrating our solution into Feedzai's current QA pipeline involves adding an extra step to the pipeline for each product version. This step will be responsible for running our tool and automating the performance metrics obtained as a result. This process is described in the next section.

4.2.2 Results automation

In the previous section we gave an overview of the current QA pipeline at Feedzai. Now we describe how our solution was implemented in the CI pipeline, with the goal of automating the collection of performance metrics data, as a result of the tool's execution.

The solution then begins by adding an extra step in the execution of the pipeline for a given version of the product. For this it was necessary to create a *.groovy* file, inside the module responsible to setup everything that should run in CI, on the application repository. This file contains the stages for executing our solution step and, these build stages are illustrated, in pseudo-code, on Listing 4.2.

```
1   stage('Setup / Install application') { ... }
2   stage('Start application') { ... }
3
4   stage('Puppetzai') {
5       directory("path/to/puppetzai/folder") {
6           invokeNPMCommand("install")
7           invokeNPMCommand("run puppetzai")
8       }
9   }
10
11  stage('Stop application') { ... }
```

Listing 4.2: Different stage builds on *puppetzai.groovy*

One of these steps will be the execution of our solution, identified as stage *Puppetzai*. We enter the dedicated directory for this tool, where we can find our JS library as a NPM development dependency. After installation, the tool runs with the configuration provided in this same folder. The tool execution follows a NPM script added on the application *package.json* file. Listing 4.3 shows the current *puppetzai* script running on Case Manager application.

```
1  {
2      "scripts": {
3          "puppetzai": "puppetzai --config=puppetzai-config.js --outputPath \"./outputs
               ↪ /\" --login",
4          // This script is available by running 'npm run puppetzai'
5      }
6  },
```

Listing 4.3: Puppetzai NPM script on Case Manager application

To complete the execution of our CI solution, it is necessary to create, through the Jenkins UI of the product, a new pipeline step that will be in charge of executing the groovy file above. As described in Section 4.2.1, each product version has its development branch and a respective pipeline created on the CI server. During the development of features or in the resolution of bugs, each commit pushed to these branches will initialize the respective pipeline in CI. Additionally, each of these pipelines is executed automatically, during the night from Monday to Friday, called *daily jobs*. Here, other heavier end-to-end tests are carried out, in order to free up execution time during the day, when several developers are working and using the CI server. Nowadays, our solution is configured to run on CI, in these *daily jobs*, in order to have daily results and not several data dispersed per day. More specifically, in Case Manager *master*, *10.1* and *10.0* versions, in the repository branches, *master*, *hf-10.1.X* and *hf-10.0.X*, respectively. Currently, data with performance metrics obtained through our solution are being published, from Monday to Friday, in respective spreadsheets for each version, after each measurement run of the configured daily jobs. The publication of these results, through the Google Sheets API, is the last step in automating the process in obtaining the final results. In the next section, we describe the implementation of the visualization process. Finally, Figure 4.4 shows the current CI pipeline structure on Case Manager's *master-daily* job, with our solution step highlighted.

4.3 Visualization

In order to complete the flow in the automation process in obtaining the results of performance metrics, it is important to carry out continuous monitoring on these metrics. Development teams may be able to assess whether performance of their applications is improving or getting worse, and this information should be accessible in an easy and effective way. Thus, we decided to adopt Google Sheets as a visualization tool responsible

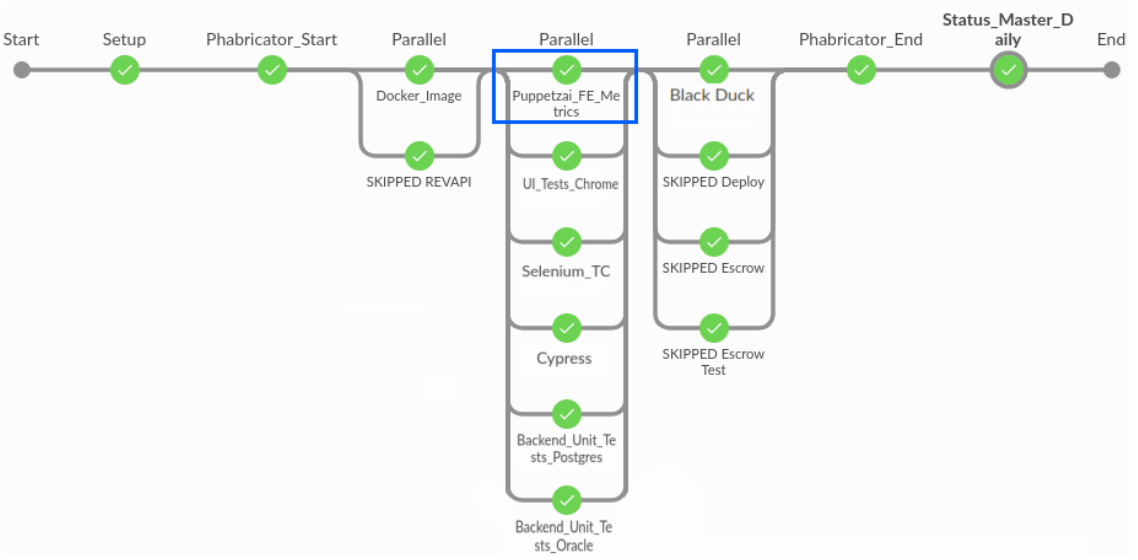


Figure 4.4: Current CI pipeline for *master-daily* on Case Manager application, with solution step highlighted in light blue. (Adapted - Some build steps are omitted)

for the results obtained, not only because it makes it possible to have access to original information in data tables, but also to easily create graphs from this data. We believe that this solution is practical and flexible in creating visualizations and dashboards. We now describe the content of these spreadsheets, and what type of dashboards we can create with Google Sheets.

For each version of the application to be monitored, a new spreadsheet must be created, according to a template [Feed] provided. In this template we find different sheets, among which the most important are: the main sheet whose name is the same as the application version, where the JSON raw results from the tool output are published. The table structure with this data is described on the next paragraph. The other most important sheet with the name *PERF_DASHBOARD*, where several charts were created with the obtained data. An example of these dashboards is illustrated in Figure 4.5. There are also individual sheets for each of these charts, in order to further analyze the data for a specific metric. This spreadsheet template is already responsible for the automatic creation of all these charts taking into account the data published.

Regarding the structure of the data tabulated in the spreadsheet that corresponds to the output results of the tool, each measurement run corresponds to several rows in the table, more specifically, equal to $\sum_i^N pageUrls[i].numRequests$ rows, where N is equal to the number of navigations configured within *pageUrls* property, on the tool configuration file, as described in Section 4.1.3. This is due to the fact that the data contains lists of requests with metrics, each one for a single navigation. In order to be correctly indexed in tables, it is necessary to have all these rows referencing the same measurement run.

With Google Sheets it is easy to share this data between the developers of the application team, and in the definition of certain read and write permissions in this document.

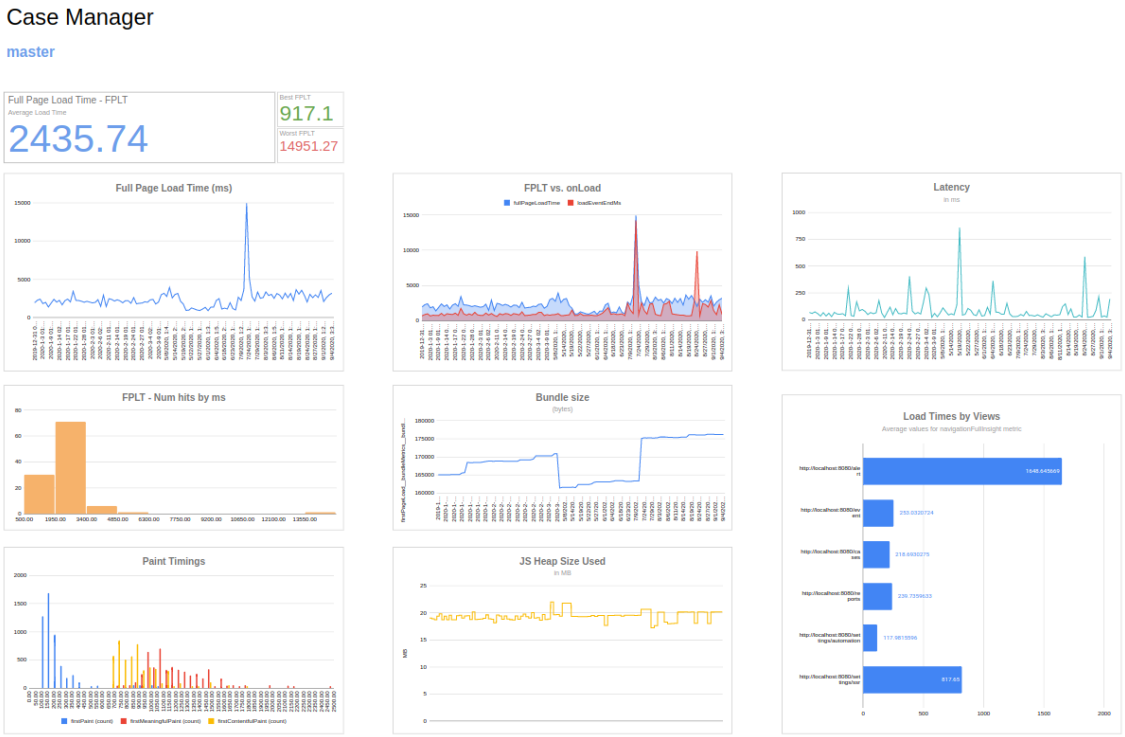


Figure 4.5: Performance dashboard for *master* version on Case Manager, with Google Sheets

Another advantage is the possibility to download the final dashboard and graphics, in pdf and jpg format, facilitating the sharing of information in other communication channels.

4.4 Summary

Our current implementation contains a CLI that connects to a browser automation tool, Puppeteer, to output results with performance metrics of a web application. Furthermore, these results are then submitted to Google Sheets, from it's API, to view the results in dashboards. In this section we not only described the implementation for each of these components, but how it is integrated into Feedzai's current QA pipeline. In the next chapter, we will use this implementation to perform an experimental study of the tool performance and a validation study to show that it is in line not only with Feedzai's requirements, but in measuring performance in different SPAs.

Chapter 5

Validation and Evaluation

In this chapter we present the experimental results using the implementation detailed in Chapter 4. In Section 5.1, we validate that our implementation meets not only Feedzai requirements but ensures that the solution is agnostic and flexible enough to be used in different SPAs. We present some of the performance results for Feedzai's product, Pulse, and also Airbnb [Air] and Web.dev [Weba]. We end this section by showing the performance results for Case Manager obtained from CI integration, and discuss briefly how these experimental results over time can help draw conclusions on the evolution of the performance of a SPA. Then, in Section 5.2 we evaluate the impact of our solution on the current CI integration in Case Manager QA pipeline and mention the current limitations of our solution. We end this chapter with a discussion on the usefulness of the results obtained for the automation in obtaining performance metrics in SPAs and their monitoring over time, in Section 5.3.

5.1 Validation

The methodology implemented for the development of our solution, took into account some requirements defined by Feedzai, presented in Section 5.1.1. The validation of our solution not only met the requirements defined, but the work resulted on a generic approach to measure performance metrics for different SPAs. A solution that tracks the evolution of loading time metrics, to measure improvements objectively and detect regressions of Feedzai products, Pulse and Case Manager, but generic enough to be applied on other SPAs. These results will be presented in Section 5.1.2.

5.1.1 Feedzai Requirements

In this section we present the initial requirements defined by Feedzai, to be applied in the context of this thesis work. For each requirement, we present a brief description and discussion of the implementation applied.

- **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. This requirement was one of the most important to be fulfilled. Our solution is easy to integrate in a CI environment, since its execution allows running the browser automation tool in a supported mode for this purpose, which allows running a browser in headless mode.

- **Must produce repeatable results.** Being a performance measuring tool it is very important that external factors are reduced to a minimum and we obtain results that are replicable. The only way for our implementation to be able to obtain replicable results, is to have access to a stable CI environment. The environment was not entirely stable, and sometimes external factors to our implementation, such as machine latency or instability on bootstraping the application, prevented us from having access to consistent results. Reliability and stability of our solution as a performance monitoring tool, depends only on the environment in which it executes.

- **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. The metrics resulting from our solution were primarily focused on the user experience. Metrics such as first-page load and paint timings easily show the user's first interaction with the application for the first time, and the navigation time metrics that justifies the time spent between different user interactions in the application.

- **Must be generic enough to be used in third party SPAs.** Notability in the two SPAs developed at Feedzai, Pulse and Case Manager, but ideally also in others. The heuristic used to implement our solution, as described in 3.5.5, allows it to be generic enough to be used across different SPAs. Additionally, with support for custom configurations, it allows a flexible and adapted solution for each application.

- **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. We made it available to generate documentation through the tool repository, in order to facilitate the process of setting and describing the metrics obtained. This is a step that is still in progress and we intend to finalize it until its open-source launch. It should be noted that, in terms of

maintenance, the only disadvantage identified will be in changing the selectors of the application itself, which must be manually updated in the configuration.

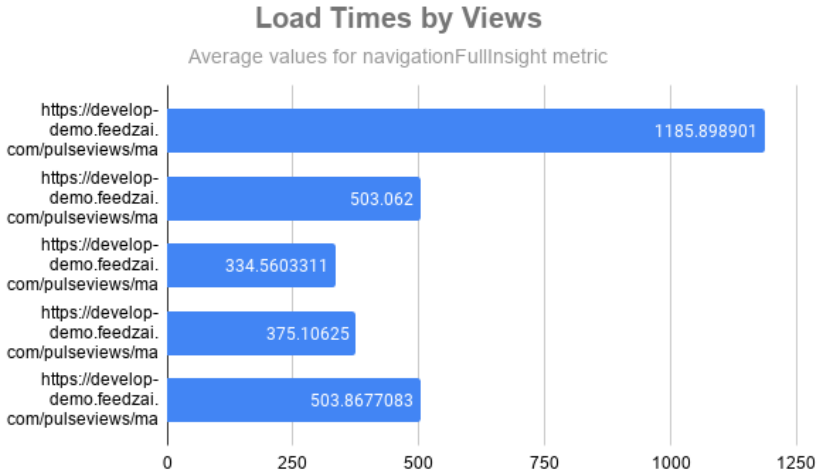
5.1.2 Pulse, web.dev, and Airbnb use cases

Our implementation is currently being applied on two Feedzai SPAs products, Pulse and Case Manager. With the current architecture of the tool, we were also able to apply our solution in different SPAs, thus resulting in a generic solution. This solution also comes with some limitations that will be further described in Section 5.2.2. Currently, the tool is deployed in Case Manager's CI infrastructure, which runs as an extra step in its test pipeline. Performance metrics results are published on spreadsheets and being monitored by the product team using the dashboards implemented, as described in Section 4.3. Regarding the Pulse application, it is not deployed in CI currently, but the solution has already been tested and evaluated, whose results of some of the metrics obtained are shown in Figure 5.1. In order to validate our solution in other SPAs, tests were also carried out in other applications, such as Airbnb [Air] and Web.dev [Weba]. For each application, a total of twenty measurement run tests were performed.

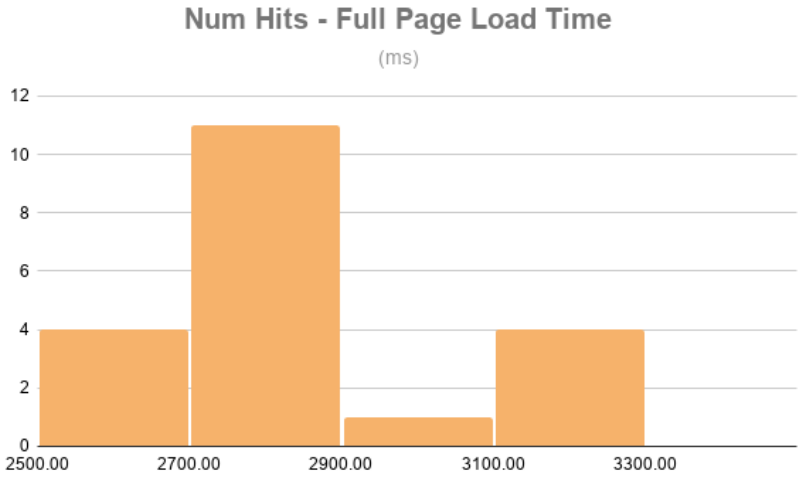
Regarding Pulse's application experimental results, the first graph in Figure 5.1(a), tells us that the navigation that takes the longest to load the necessary resources is the navigation */apps*. This navigation is the first interaction after authentication on the Pulse application login page. In the remaining routes, or configured navigations, the one considered faster compared to the others, is */data-science*. It is a page that takes about 334 milliseconds to load. Regarding the metric of *full-page* load time, which indicates the total time until the last resource needed to load the first page, we show a histogram on Figure 5.1(b). In this graph we can easily understand that the most common valus obtained by this metric are around 2.7 and 2.9 seconds. These and many other graphics will be available on the implemented dashboards, as shown in Section 4.3 for the case use of the Case Manager application. Thus, they complement the information on different metrics that help Pulse team developers, to gain some insights on the status of the application.

On the Airbnb [Air] website, the tests conducted were performed on the *Hosts* main page, with a configuration with three navigations between the */hosts/d/setup*, */hosts/d/safety* and */hosts/d/financials* pages, without a configured navigation timeout. The configuration used is shown on Appendix B, in Listing B.3. By the configuration used, it is possible to determine which of the navigations is considered the heaviest in terms of byte consumption by the network. Through the analysis of the results in the implemented dashboards, it is possible to generate a histogram with the *totalAjaxBytesSize* metric as a variable. Figure 5.2 shows this graph and we can easily determine that the route */setup* is the one that consumes the most resources loaded over the network, with a total of approximately 0.02 MB.

In the tests carried out for the Web.dev [Weba] website, a configuration with only three navigations was used, which identify the pages, */learn*, */measure* and */blog*, without



((a)) Representation of *navigationFullInsight* metric values for each navigation



((b)) Histogram with the number of hits for each *full-page* load time metric values

Figure 5.1: Pulse's experimental results for *navigation-FullInsight* and hit values for *full-page* load performance metrics

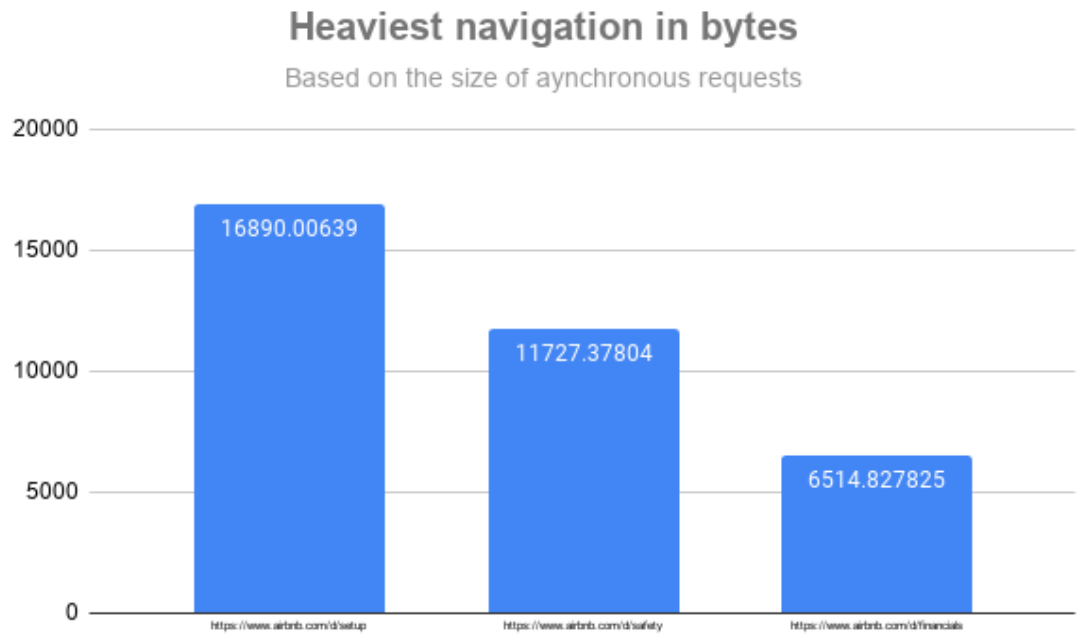


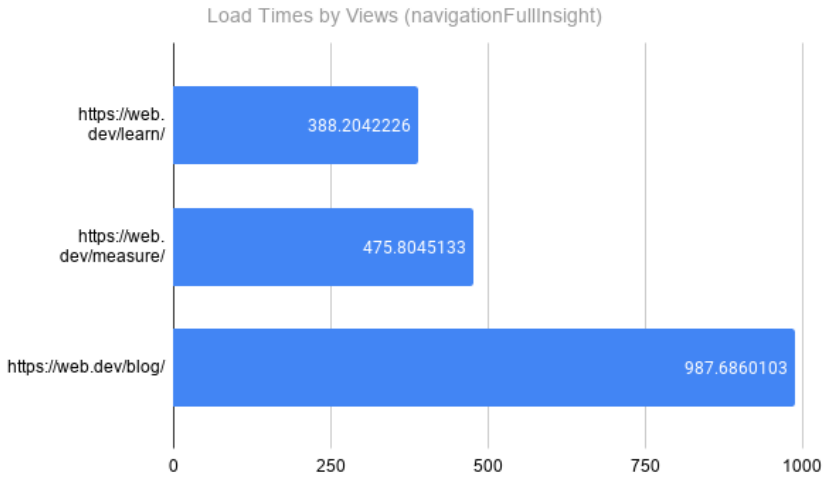
Figure 5.2: Histogram with *totalAjaxBytesSize* metric values for Airbnb configuration

any navigation timeout configured. The configuration used is shown on Appendix B, in Listing B.4. After analyzing the results, it is possible to check the total time spent on network activity, in each of these navigations, as illustrated in Figure 5.3(a). These data are obtained through the measurement of the *navigationFullInsight* performance metric. We can conclude that the route */blog* is the navigation that takes the longest, taking about 1 second to load all the necessary resources. In Figure 5.3(b), through the analysis of the metric *full-page* load time, we can see that the loading of the initial resources of the first page varies between 2 and 3.5 seconds.

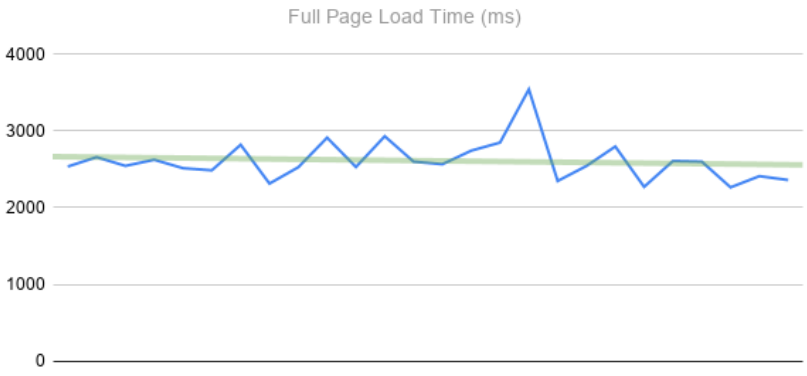
We conclude that the heuristic implemented to define the final time of soft navigations, as explained in Section 3.5.5, works for different navigations, or interactions, in a SPA. Together with configurations adapted for each application to be tested, it was possible to obtain in the end the essential metrics in this type of navigation. In the next Section 5.1.3, we go even further, and show the effectiveness and utility in obtaining these results in an automated environment, integrated and deployed in a test pipeline, of the Case Manager application.

5.1.3 Case Manager

In the context of the development of this dissertation work, its implementation and integration focused on a solution capable of being applied to the products developed by Feedzai. The main products were Pulse and Case Manager and, in Section 5.1.2, we showed some of the experimental results against the Pulse application. In a more final and refined phase of the implementation, the Case Manager was our main use case, in



((a)) Representation of *navigationFullInsight* metric values for each navigation



((b)) Evolution of the *full-page* load time metric

Figure 5.3: Web.dev's experimental results for *full-page* load time and *navigationFullInsight* performance metrics

order to validate that its integration in the current development pipeline in QA, we were able to generate results that would be in accordance with the requirements defined by Feedzai. We now show the experimental results obtained in this integration, whose data range from 31/12/2019 to 09/04/2020. This data represents the evolution of some of the metrics implemented by our solution, against the main development version, usually called the master version, of the Case Manager application repository.

The analysis of the evolution of the full-page load metric, whose implementation is explained in Section 3.5.1, gives us the whole picture of the total time spent loading the resources necessary for loading the first and only page of a SPA. The evolution of this metric is illustrated in Figure 5.4. With a variation of values between 0.9 and 4 seconds, with an average of 2.44 seconds whose trend points to an increase, as represented in a green line, if no action is taken to improve this time. For the previous calculations, we did not take into account the outliner result removed on 07/23/2020, with a value of approximately 15 seconds. This result, although outside the scope of the remaining values obtained, it is important to note that the instability of the test environment, as explained in Section 5.1.1, can also cause this type of results. Even considering the results presented in Figure 5.5, in the evolution of the bundle size, on the same day there was an increase. But this increase does not justify the value in the metric of full-page load, since the later results are lower. Additionally, the processing time of the bundle on this day was also evaluated, and there were no significant differences. These results are available for further analysis in Appendix C, in Figure C.2.

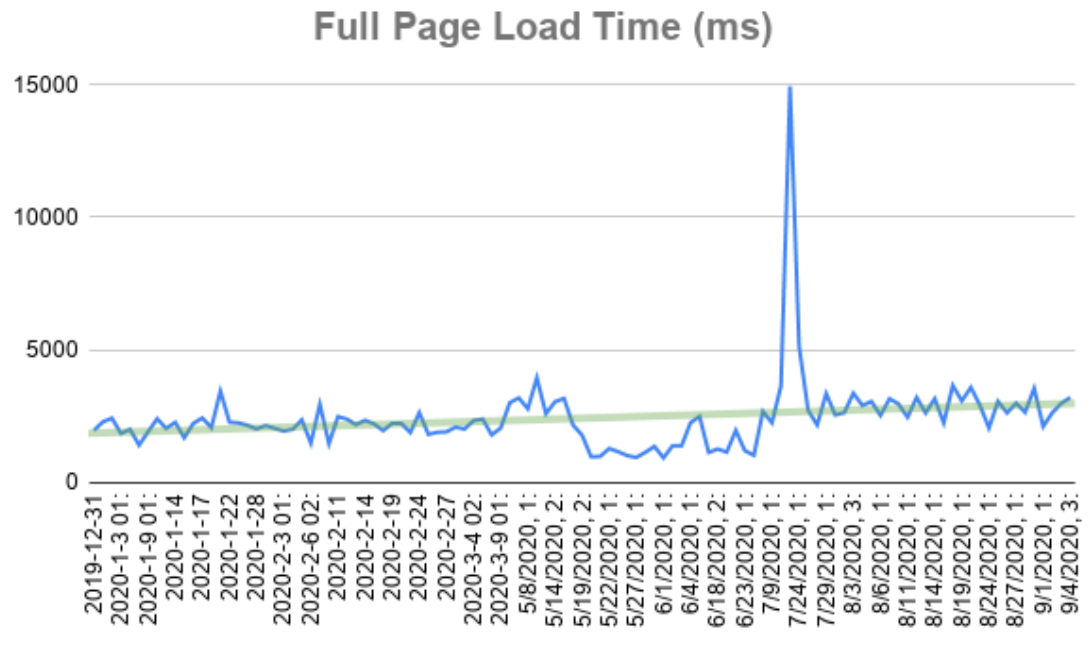


Figure 5.4: Evolution of the full-page load metric in Case Manager

The evolution of the bundle size of a SPA application is equally important in the analysis of the first-page load, since it initially loads the JS framework. Figure 5.5 shows an interesting evolution of this metric. As we can see, there was a decrease in the size of the bundle in early June. This decrease was justified by the merge of several optimizations performed in the bundle of one of the dependencies of the application itself, in the UI components library, in a feature that resulted in a total revamp of the application's UI. Two months later, with the launch of a new version of the application, several developments about the version were merged that impacted the size of the bundle, such as new dependencies added and more logic in the JS components.

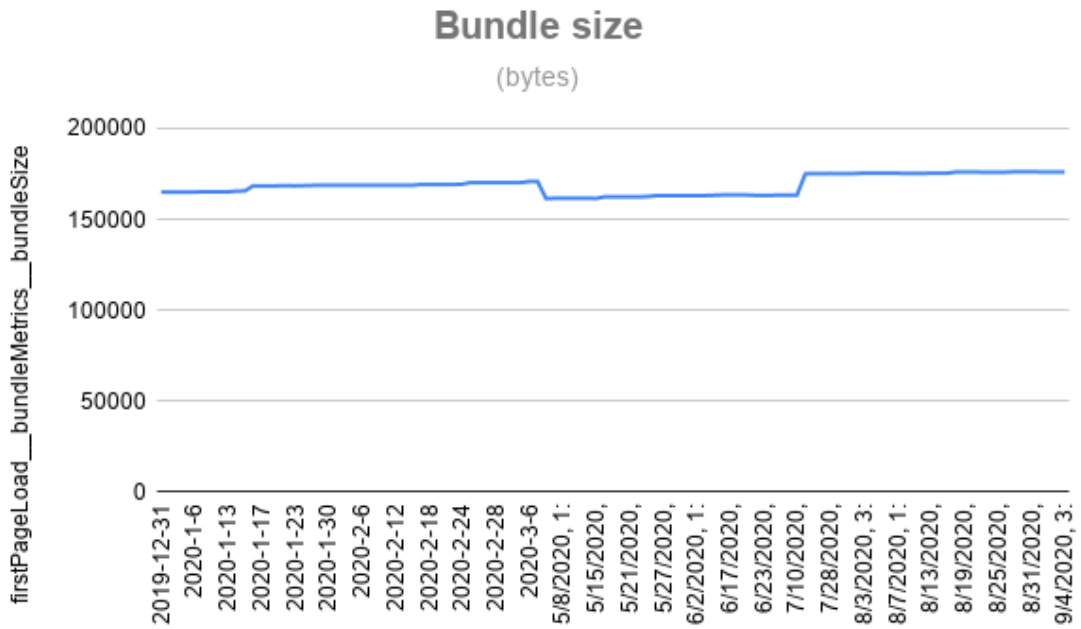


Figure 5.5: Evolution of the bundle size metric in Case Manager

One of the main objectives with these experimental results, would be to understand which page navigations were considered slower in Case Manager. Figure 5.6 represents a histogram with the average value of the *navigationFullInsight* metric, resulting from our solution and explained in Section 3.5.5, for each page navigation in the configuration. We can thus verify that the page navigation that consumes more network resources to complete the interaction, is the route */alert*. This is the navigation right after authentication in the application, which will be responsible for making several requests to the API, in order to build the main page.

In the end, these results show that it is possible to draw conclusions about the evolution of an application's performance. Starting by automating the process in obtaining the metrics, generating the results and their subsequent analysis, we can easily monitor their evolution. With this analysis, earlier it is possible to detect unwanted variations in an application version and revert changes if they really affect the application's performance.

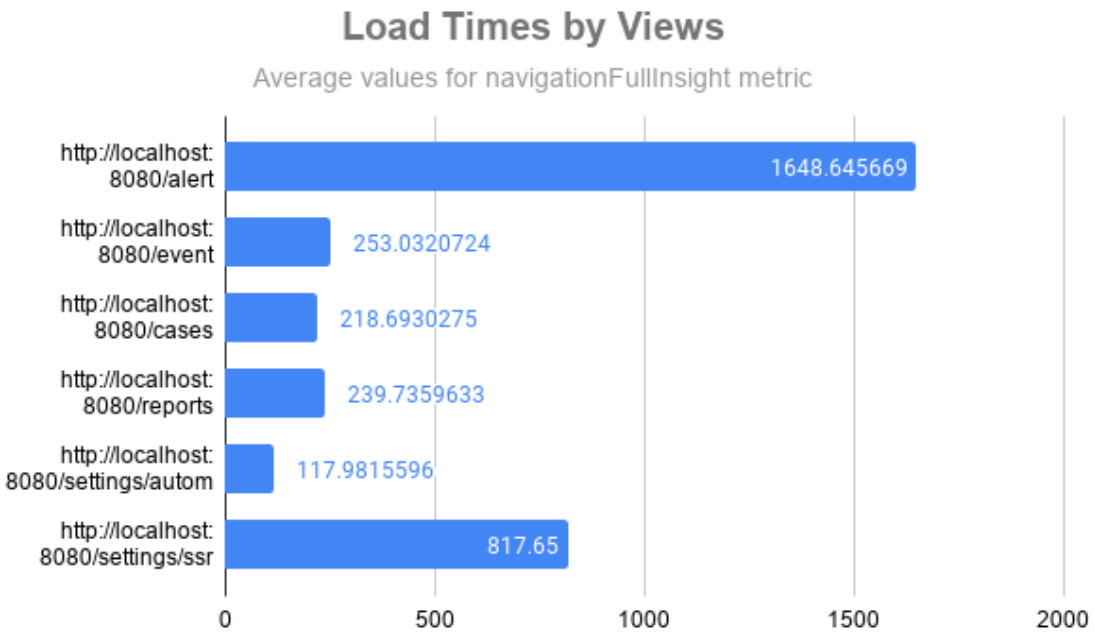


Figure 5.6: Average page navigations load times in Case Manager, representing the navigationFullInsight performance metric

As product requirements are constantly changing, and the more developers touch the codebase, the less likely it is that it's going to stay fast. We need to shift towards adopting this monitoring performance solutions in order to take care of product quality and performance.

5.2 Experimental Evaluation

This section documents the impact of our solution on a CI integration, in the current test pipeline of the Case Manager application, developed by Feedzai. The purpose of this assessment will be to demonstrate that the integration of our implementation as a performance monitoring solution is feasible and does not add a significant impact to a test pipeline. We end this section with a description of the limitations of our implementation, in order to raise awareness among developers who want to adopt our solution to monitor their applications.

5.2.1 Build impact results for a QA pipeline

Case Manager's CI test pipeline consists of several builds, which run in parallel to optimize the total execution of the so-called one-shot. These builds run a variety of tests, such as end-to-end tests, unit tests and integration tests. In Section 4.2.2, we describe how an extra step was added to this test pipeline. This new step is responsible for running a new build that runs our solution, called Puppetzai build. In order to accurately assess the

impact of this new build, Figure 5.7 shows the total percentage affected when compared to the remaining builds of a one-shot, in the last ten builds performed.

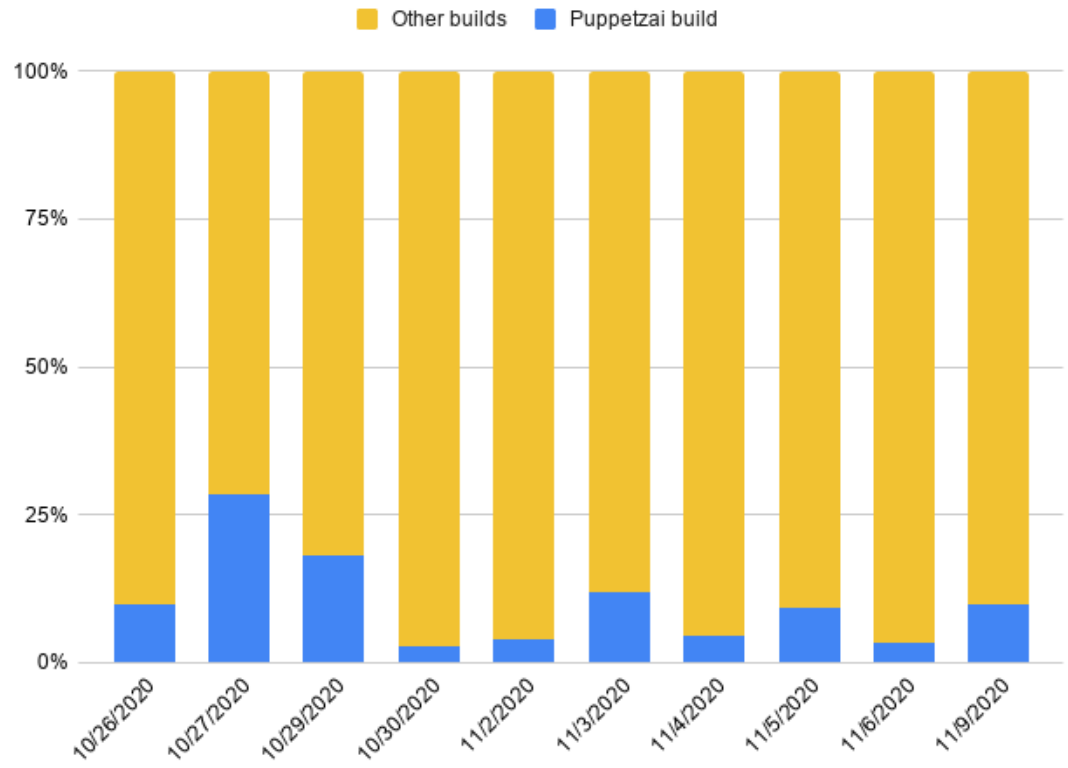


Figure 5.7: Puppetzai build impact (%) against total *one-shot* build time

As we can see, the maximum total percentage verified was slightly above 25%, with an actual value equal to 28.5% (on 10/27/2020) and the lowest equal to 2.6% (on 10/30/2020). The execution of our build not only depends on the execution of the *puppetzai.js* script, described in Section 4.1.1, but also depends on the installation and setup of the application. This bootstrap process is sometimes unstable and occupies more than 95% of the total Puppetzai build, as shown in the graph in Figure 5.8. The remaining 5% correspond to the total execution time of the *puppetzai.js* script.

We therefore believe that our solution adds minimal impact to a build running on CI, whose total time depends on variables such as the application's bootstrap and the test environment. These values go according to expectations in the requirements defined by Feedzai.

5.2.2 Limitations

The solution we propose in the context of the work carried out in this thesis has some limitations. The heuristic implemented in our solution, for measuring performance metrics in network activity, is not entirely reliable. It can trick us to think that the current

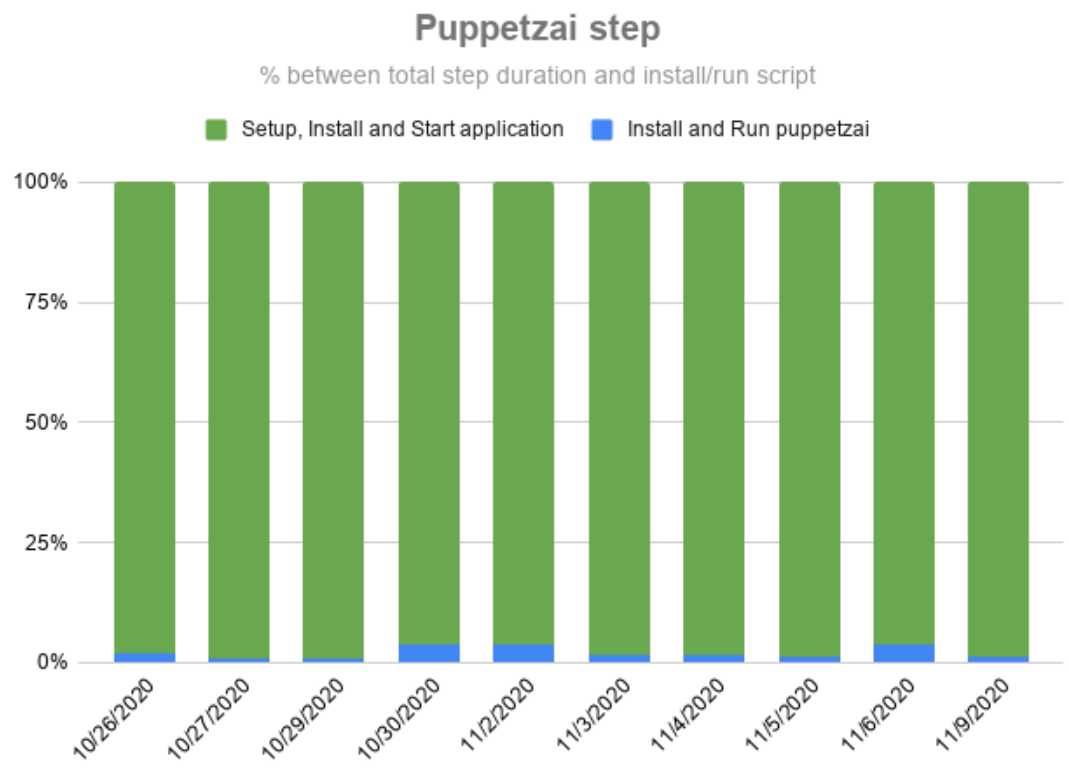


Figure 5.8: Breakdown in percentage from Puppetzai build step (Bootstrap application and puppetzai script execution)

navigation is taking x seconds to load, when actually the user perceived load time, ie when the user is able to interact with the application during this phase, can be before this load. We believe that the metric we proposed, *navigationFullInsight*, is useful but not entirely reliable on user's perceived navigation load time.

Another of the limitations of our solution is the fact that we only measure network activity between each navigation or interaction. And that, in fact, for the context of this thesis in the measurement of network-intensive SPAs, this measurement is trustworthy to give us some insight about the network activity between these navigations. On the other hand, does not give us the whole picture of background tasks that the browser is responsible for, such as JS processing, repaint time, etc. We believe that measuring these metrics is equally important and, therefore, we present this idea in future work, in Chapter 6, Section 6.2.

Finally, our solution is dependent on the implementation of the Chrome DevTools Protocol [Chr]. To obtain and calculate some metrics for requests, we need access to the DevTools tracing file, resulting from the execution of our solution. This means that measurements carried out by our solution will only be supported by the Chrome browser and not by others, such as Firefox.

5.3 Discussion

In the context of this thesis work, the final solution implemented proved to be capable of being used as a viable choice of performance monitoring solution. Its integration in CI allows to automate the entire flow from obtaining metrics, to its publication and in the creation of dashboards that allow a more rigorous analysis of the data. Currently integrated into the Case Manager pipeline, the data obtained served as source of truth to justify some of the developments. Not only do they prove to be useful for monitoring the application's performance, but it also intends to predict critical situations where there could be a significant performance regression.

Considering the results obtained in our experimental work in the context of Case Manager, it is possible to demonstrate that there is sufficient margin for performance optimizations, not only at the bundle level, but in the total time between navigations. This also applies to Pulse application. Both navigation times after the login authentication have the biggest times and should be considered to optimize. Subsequently, the team will have a source to verify that such optimizations are really having an impact on performance. We believe that even with the limitations presented in Section 5.2.2, the final results obtained justify the adoption of this framework. Furthermore, the impact shown in Section 5.2.1 does not prove to be significant in order not to be integrated into any other Feedzai product, for example Pulse, or any other SPA.

5.4 Summary

In this section, we conducted extensive experimental work focusing on measuring and capturing performance metrics for different SPA. From our results we were able to verify that our framework can provide insights and showcase the performance evolution over time for these type of applications, to capture relevant metrics and build dashboards with the goal of easily identify regressions and significant variatons.

In the following chapter we make some closing remarks regarding the contributions of this work and how it can impact the way we measure and monitor SPA performance. Finally, we end with a brief discussion on future work related to this thesis concerning further developments and new performance metrics to measure.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work we presented an approach to obtain metrics related to the performance of network activity in network-intensive SPAs. This approach was applied to Feedzai products, but also to other applications, such as Airbnb. We designed an easy and generic enough interface to be integrated with any SPA, and also in the automation of the process in obtaining performance metrics, with easy integration in a CI pipeline. We presented the main metrics, how they are obtained and calculated, which result in reports, later presented in visualization dashboard formats, such as Google Sheets, for the monitoring of these metrics. This study resulted in a means of carrying out this monitoring, in order to make it possible to detect regressions along different development points in a given application.

We have also detailed our implementation that became available internally from Feedzai, which currently supports the Case Manager product. We demonstrated that our solution can be easily configured to run other applications and generate reports with performance metrics. Taking into account the requirements presented, an additional effort was made to keep the interface simple to operate, maintain and document the processes and description of metrics. With our experiments we were able to demonstrate that the solution is reliable in measuring metrics in different SPAs. These experimental results not only demonstrate that the interface can be implemented in a SPA, but also validate our implementation as a reliable tool with regard to performance measurements in SPAs.

It should be noted that the current solution has a few limitations, as presented in Section 5.2.2, and thus do not demonstrate that our solution is effective in measuring certain metrics, such as the time between navigation. It is not trivial to cover all scenarios when

measuring performance of SPAs, since each one is designed and implemented differently. Our approach mainly focuses on measuring network activity, i.e., intercepting requests made during navigation. Therefore, we believe that the most important contribution in this work is the approach to obtain performance metrics regarding those navigations. We believe that from here it is possible to explore other types of metrics and complement information about the performance of SPAs.

6.2 Future Work

As future work, we plan to add more performance metrics between soft navigations in SPAs. In particular, we want to measure the total time spent on processing JS, between navigations. Thus, we complement our heuristic in deciding whether a navigation is finished, and not just relying on us when the network stays idle for 800 ms, as implemented in the navigationFullInsight metric. To improve the monitoring of performance metrics over time, and to avoid significant performance deviations between the development of new features in these applications, we propose to add a verification method taking into account thresholds, known as performance budgets, previously defined by the team itself. development. Thus, after measuring the results, they would be compared with the defined budgets and, if any value of any metric did not meet the performance requirements, the stage that runs in the CI deployed build of the application failed. Thus, the developer will have instant feedback on your changes and their impact on the application's performance.

Bibliography

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

[Gri16] I. Grigorik. *High-Performance Browser Networking*. OReilly, 2016.

[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.

[SR03] L. Shklar and R. Rosen. *Web Application Architecture: Principle, protocols and practices*. Wiley, 2003.

[Sou07] S. Souders. *High Performance Web Sites*. OReilly, 2007. ISBN: 0-596-52930-9.

[Sou09] S. Souders. *Even Faster Web Sites*. OReilly, 2009. ISBN: 978-0-596-52230-8.

[Wag17] J. L. Wagner. *Web Performance in Action: Building Faster Web Pages*. Manning Publications, 2017. ISBN: 9781617293771.

Webography

[Air] *Airbnb Hosts Website*. URL: https://www.airbnb.com/.

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

[Basa] K. Basques. *Get Started With Analyzing Runtime Performance*. URL: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/.

[Basb] K. Basques. *Network Analysis Reference*. URL: https://developers.google.com/web/tools/chrome-devtools/network/reference.

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

[Chr] *Chrome Devtools Protocol*. URL: https://chromedevtools.github.io/devtools-protocol/.

[Deva] G. Developers. *AngularJS Developer Guide*. URL: https://angularjs.org/.

[Devb] G. Developers. *Chrome Developer Tools*. URL: https://developers.google.com/web/tools/chrome-devtools/.

[Devc] G. Developers. *Create and managing projects*. URL: https://cloud.google.com/resource-manager/docs/creating-managing-projects.

[Devd] G. Developers. *First Contentful Paint*. URL: https://web.dev/first-contentful-paint/.

[Deve] G. Developers. *Lighthouse*. URL: https://developers.google.com/web/tools/lighthouse.

[Devf] G. Developers. *V8 website*. URL: https://v8.dev/.
V8 is Google's open source high-performance JavaScript and WebAssembly engine.

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

[Feea] Feedzai. *Case Manager documentation*. URL: https://docs.feedzai.com/display/ENG/C+-+Case+Manager.
[Feedzai internal]. This reference contains all the documentation regarding Feedzai's Case Manager web application, from the Engineering Team.

[Feeb] Feedzai. *Feedzai official ESlint configuration, open-source, on Github*. URL: https://github.com/feedzai/eslint-config-feedzai.

[Feec] Feedzai. *Pulse documentation*. URL: https://docs.feedzai.com/display/ENG/C+-+Pulse.
[Feedzai internal]. This reference contains all the documentation regarding Feedzai's Pulse web application, from the Engineering Team.

[Feed] Feedzai. *[TEMPLATE] App Version*. URL: https://docs.google.com/spreadsheets/d/1mrdJnxKobBrgyL0hJWyiuqf7zPuwncGbtkUSS9fr-Rg/edit?usp=sharing.
[Feedzai internal] This template is a Google Sheets spreadsheet ready to use to quick start visualization of performance metrics on a web application.

[Feee] *Feedzai website*. URL: https://feedzai.com/.

[Moza] Mozilla. *Mozilla Web Docs - First paint*. URL: https://developer.mozilla.org/en-US/docs/Glossary/First_paint.

[Mozb] Mozilla. *Mozilla Web Docs - Measuring performance*. URL: https://developer.mozilla.org/en-US/docs/Learn/Performance/Measuring_performance.

[New] *New Relic*. URL: https://newrelic.com/.

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

[Osma] A. Osmani. *JavaScript Start-up Optimization*. URL: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/javascript-startup-optimization.

[Osmb] A. Osmani. *Web Page Usability Matters*. URL: https://addyosmani.com/blog/usability/.

[Pup] Puppeteer. *Puppeteer official API documentation*. URL: https://pptr.dev/.

[Ppt] *Puppeteer API - Puppeteer Launch Options*. URL: https://pptr.dev/#?product=Puppeteer&version=v5.2.1&show=api-puppeteerlaunchoptions.

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

[Rep] W. G. Repository. *WebPageTest Metrics*. URL: https://github.com/WPO-Foundation/webpagetest-docs/blob/master/user/Metrics/README.md.

[Spea] *Representation of SpeedCurve metrics*. URL: https://support.speedcurve.com/understanding-metrics/what-do-the-different-speedcurve-metrics-represent.

[Rus] A. Russell. *Can You Afford It?: Real-world Web Performance Budgets*. URL: https://infrequently.org/2017/10/can-you-afford-it-real-world-web-performance-budgets/.

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

[Speb] *SpeedCurve*. URL: https://speedcurve.com/about/.

[Oau] *Using OAuth 2.0 to Access Google APIs*. URL: https://developers.google.com/identity/protocols/oauth2.

[W3Ca] W. W. W. C. W3C. *Performance Navigation Timing API*. URL: https://www.w3.org/TR/navigation-timing-2/#sec-PerformanceNavigationTiming.

[W3Cb] W. W. W. C. W3C. *Performance Paint Timing API*. URL: https://www.w3.org/TR/paint-timing/#sec-PerformancePaintTiming.

[W3Cc] W. W. W. C. W3C. *Performance Resource Timing API*. URL: https://www.w3.org/TR/resource-timing-2/#sec-performanceresourcetiming.

[Wal] P. Walton. *User-centric performance metrics*. URL: https://developers.google.com/web/fundamentals/performance/user-centric-performance-metrics.

[Weba] *Web.dev Website*. URL: https://web.dev/.

[Webb] WebPageTest. *WebPageTest Website*. URL: https://www.webpagetest.org/.

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

Appendix A

Performance metrics results example


```
{
  "timestamp": 1599697689,
  "date": "9/10/2020, 1:28:09 AM",
  "url": "http://localhost:8080",
  "firstPageLoad": {
    "bundleMetrics": {
      "bundleParseTimeMs": 23.737,
      "bundleEvaluationTimeMs": 408.552,
      "bundleSize": 176357
    },
    "fullPageLoadTime": 3018.15,
    "navigationTimings": {
      "responseEndMs": 53.26,
      "domInteractiveMs": 681.99,
      "domContentLoadedEventEndMs": 682.29,
      "domCompleteMs": 728.19,
      "loadEventEndMs": 728.24,
      "interactiveToCompleteMs": 46.2,
      "latencyMs": 51.59,
      "transferTimeMs": 1.42
    },
    "puppeteerTimings": {
      "jsEventListeners": 23,
      "nodes": 217,
      "layoutCount": 27,
      "recalcStyleCount": 44,
      "jsHeapUsedSize": 20118992,
      "layoutDurationMs": 60.65,
      "recalcStyleDurationMs": 30.77,
      "scriptDurationMs": 460,
      "taskDurationMs": 818.44
```

```
  },
  "paintTimings": {
    "firstPaint": 174.72,
    "firstContentfulPaint": 712.79,
    "firstMeaningfulPaint": 954.77
  },
  "resourceTimings": {
    "numRequests": 17,
    "totalDurationMs": 3090,
    "totalRequestsBytesSize": 928817,
    "resources": [
      {
        "name": "http://localhost:8080/ext/style/index.css",
        "initiator": "link",
        "startTime": 70.79,
        "transferSize": 5385,
        "timeToFirstByte": 9.37,
        "transferTime": 2.47,
        "totalTime": 12,
        "responseEnd": 82.64
      },
      {
        "name": "http://localhost:8080/vendor.ed9ee0e0.chunk.css",
        "initiator": "link",
        "startTime": 71.1,
        "transferSize": 54640,
        "timeToFirstByte": 38.62,
        "transferTime": 20.32,
        "totalTime": 59,
        "responseEnd": 130.04
      },
      {
        "name": "http://localhost:8080/app.aaa3c53b.js",
        "initiator": "script",
        "startTime": 71.72,
        "transferSize": 176357,
        "timeToFirstByte": 19.06,
        "transferTime": 31.68,
        "totalTime": 51,
        "responseEnd": 122.46
      },
      {
        "name": "http://localhost:8080/api/configuration/latest",
        "initiator": "fetch",
        "startTime": 680.77,
        "transferSize": 2020,
        "timeToFirstByte": 42.59,
        "transferTime": 7.85,
        "totalTime": 50,
        "responseEnd": 731.21
```

```
      },
      {
        "name": "http://localhost:8080/Roboto-Regular-4933e03a.woff2",
        "initiator": "css",
        "startTime": 684.07,
        "transferSize": 1104,
        "timeToFirstByte": 19.36,
        "transferTime": 0.79,
        "totalTime": 20,
        "responseEnd": 704.22
      },
      ...
    ]
  }
},
"softNavigationsMetrics": [
  {
    "navigationUrl": "http://localhost:8080/alert",
    "navigationFullInsight": 1454.91,
    "numAjaxRequests": 19,
    "avgTimePerAjaxRequest": 96.54,
    "totalAjaxBytesSize": 374412,
    "avgBytesSizePerAjaxRequest": 19705.89,
    "requests": [
      {
        "url": "http://localhost:8080/api/session",
        "requestId": "1000011940.44",
        "type": "Fetch",
        "startTime": 3054264.950255,
        "queueingTime": 0.4980000667273998,
        "timeToFirstByte": 92.21,
        "contentDownloadTime": 1,
        "duration": 94.08,
        "size": 2733,
        "endTime": 3054265.044334
      },
      {
        "url": "http://localhost:8080/api/version",
        "requestId": "1000011940.49",
        "type": "Fetch",
        "startTime": 3054265.326921,
        "queueingTime": 0.7650000043213367,
        "timeToFirstByte": 16.81,
        "contentDownloadTime": 0.83,
        "duration": 18.98,
        "size": 532,
        "endTime": 3054265.345898
      },
      {
        "url": "http://localhost:8080/api/status-reason?limit=-1",
```

```
        "requestId": "1000011940.51",
        "type": "Fetch",
        "startTime": 3054265.351114,
        "queueingTime": 0.7269997149705887,
        "timeToFirstByte": 71.25,
        "contentDownloadTime": 1.52,
        "duration": 74.16,
        "size": 6577,
        "endTime": 3054265.425274
      },
      ...
    ]
  },
  {
    "navigationUrl": "http://localhost:8080/event",
    "navigationFullInsight": 172.19,
    "numAjaxRequests": 6,
    "avgTimePerAjaxRequest": 58.36,
    "totalAjaxBytesSize": 36452,
    "avgBytesSizePerAjaxRequest": 6075.33,
    "requests": [
      {
        "url": "http://localhost:8080/api/queue-user/user-queues",
        "requestId": "1000011940.67",
        "type": "Fetch",
        "startTime": 3054267.245771,
        "queueingTime": 0.674000009894371,
        "timeToFirstByte": 10.33,
        "contentDownloadTime": 1.08,
        "duration": 12.94,
        "size": 389,
        "endTime": 3054267.258714
      },
      {
        "url": "http://localhost:8080/api/entity/customer?limit=100&offset=0",
        "requestId": "1000011940.68",
        "type": "Fetch",
        "startTime": 3054267.320767,
        "queueingTime": 0.421999953687191,
        "timeToFirstByte": 85.09,
        "contentDownloadTime": 0.87,
        "duration": 87.4,
        "size": 21407,
        "endTime": 3054267.408169
      },
      ...
    ]
  },
  {
    "navigationUrl": "http://localhost:8080/cases",
```

```
      "navigationFullInsight": 413.62,
      "numAjaxRequests": 5,
      "avgTimePerAjaxRequest": 93.2,
      "totalAjaxBytesSize": 5275,
      "avgBytesSizePerAjaxRequest": 1055,
      "requests": [
        {
          "url": "http://localhost:8080/api/tenant?limit=100&offset=0",
          "requestId": "1000011940.74",
          "type": "Fetch",
          "startTime": 3054268.333899,
          "queueingTime": 0.5179997533559799,
          "timeToFirstByte": 360.77,
          "contentDownloadTime": 1.42,
          "duration": 363.35,
          "size": 1445,
          "endTime": 3054268.697252
        },
        ...
      ]
    },
    {
      "navigationUrl": "http://localhost:8080/reports",
      "navigationFullInsight": 364.81,
      "numAjaxRequests": 5,
      "avgTimePerAjaxRequest": 90.2,
      "totalAjaxBytesSize": 5260,
      "avgBytesSizePerAjaxRequest": 1052,
      "requests": [
        {
          "url": "http://localhost:8080/api/queue-user/user-queues",
          "requestId": "1000011940.78",
          "type": "Fetch",
          "startTime": 3054269.549713,
          "queueingTime": 0.39799977093935013,
          "timeToFirstByte": 59.76,
          "contentDownloadTime": 0.66,
          "duration": 61.38,
          "size": 389,
          "endTime": 3054269.611096
        },
        ...
      ]
    },
    ...
  ]
}
```

Listing A.1: Example of a JSON report with performance metrics

Appendix B

Configuration examples

```
module.exports = {
    puppeteer: {
        headless: true,
        devtools: false,
        args: ["--no-sandbox", "--disable-setuid-sandbox"], // To run cleanly on \gls{
            ↪ CI} environment
        defaultViewport: {
            width: 1920,
            height: 1080,
        },
    },
    application: {
        url: "http://localhost:8080",
        bundle: "app",
        login: {
            // Login credentials
            username: "XXXXX",
            password: "**********",

            // Login form
            fields: {
                username: ".fdz-css-login__step-container > div > div > label:nth-child
                    ↪ (1) > input",
                password: ".fdz-css-login__step-container > div > div > label:nth-child
                    ↪ (2) > input",
            },
            navigationSelector: ".fdz-css-login__step-container > div > button",
        },
        pageUrls: [
            {
                // All Events page
```

```
29                url: "/event",
30                navigationSelector: "nav > ul > li:nth-child(1) > div > a",
31            },
32            {
33                // Reports page
34                url: "/cases",
35                navigationSelector: "nav > ul > li:nth-child(3) > div > a",
36            },
37            {
38                // Reports page
39                url: "/reports",
40                navigationSelector: "nav > ul > li:nth-child(5) > div > a",
41            },
42            {
43                // Automation rules
44                url: "/settings/automation",
45                hover: "nav > ul > li:nth-child(6)",
46                navigationSelector: ".rc-menu-submenu:not(.rc-menu-submenu-hidden) > ul.
                      ↪ rc-menu > li:nth-child(2) > a",
47            },
48            {
49                // Self-Service Rules page
50                url: "/settings/ssr",
51                hover: "nav > ul > li:nth-child(6)",
52                navigationSelector: ".rc-menu-submenu:not(.rc-menu-submenu-hidden) > ul.
                      ↪ rc-menu > li:nth-child(5) > a",
53                navigationTimeout: 2000,
54            },
55        ],
56    },
57 };
```

Listing B.1: Case Manager current configuration deployed on CI

```
1  module.exports = {
2      puppeteer: {
3          headless: true,
4          devtools: false,
5          ignoreHTTPSErrors: true,
6          defaultViewport: {
7              width: 1280,
8              height: 800
9          }
10     },
11     application: {
12         url: "https://develop-demo.feedzai.com/",
13         bundle: "commonDll",
14         login: {
15             username: "XXXX",
16             password: "*********",
```

```
            fields: {
                username: ".fdz-js-login-username",
                password: ".fdz-js-login-password"
            },
            navigationSelector: ".fdz-js-login-sign-in-btn",
            waitForElement: ".fdz-js-management__container",
            navigationTimeout: 2000
        },
        pageUrls: [
            {
                url: "#apps",
                navigationSelector: "body > div.fdz-js-management.fdz-css-management >
                    ↪ div.fdz-js-management__container.fdz-css-management__container.
                    ↪ interface > div > div.fdz-js-management__content.fdz-css-
                    ↪ management__content > div > div.fdz-js-pulse-center-column.fdz-
                    ↪ css-pulse-center-column > div.main-container > ul > li > div.fdz-
                    ↪ js-apps-list__item-body.fdz-css-apps-list__item-body > a",
                navigationTimeout: 2000
            },
            {
                url: "/data-science",
                navigationSelector: "body > div.fdz-js-management.fdz-css-management >
                    ↪ div.fdz-js-management__container.fdz-css-management__container.
                    ↪ interface > nav > div > div.fdz-js-react-menu-container > div >
                    ↪ ul.nav.builderSectionList.fdz-js-app-tabs > li:nth-child(2) > a",
                navigationTimeout: 2000
            },
            {
                url: "/workflows",
                navigationSelector: "body > div.fdz-js-management.fdz-css-management >
                    ↪ div.fdz-js-management__container.fdz-css-management__container.
                    ↪ interface > nav > div > div.fdz-js-react-menu-container > div >
                    ↪ ul.nav.builderSectionList.fdz-js-app-tabs > li:nth-child(3) > a"
            },
            {
                url: "/lists",
                navigationSelector: "body > div.fdz-js-management.fdz-css-management >
                    ↪ div.fdz-js-management__container.fdz-css-management__container.
                    ↪ interface > nav > div > div.fdz-js-react-menu-container > div >
                    ↪ ul.nav.builderSectionList.fdz-js-app-tabs > li:nth-child(4) > a"
            }
        ]
    }
};
```

Listing B.2: Pulse configuration

```
module.exports = {
    puppeteer: {
        headless: true
```

```
    },
    application: {
        url: "https://www.airbnb.com/host/homes/",
        pageUrls: [
            {
                url: "/setup",
                navigationSelector: "body > div:nth-child(7) > div > div > div > div.
                    ↪ _16grqhk > div._siy8gh > div > header > div > div._19h9w7f > nav
                    ↪ > div._w2q1b8 > a > div"
            },
            {
                url: "/safety",
                navigationSelector: "#section_id_242498 > div > div > div > div > div.
                    ↪ _nn6efy > div:nth-child(4) > a"
            },
            {
                url: "/financials",
                navigationSelector: "#section_id_242498 > div > div > div > div > div.
                    ↪ _nn6efy > div:nth-child(3) > a"
            }
        ]
    }
};
```

Listing B.3: Airbnb configuration

```
module.exports = {
    puppeteer: {
        headless: true
    },
    application: {
        url: "https://web.dev/",
        pageUrls: [
            {
                url: "/learn",
                navigationSelector: "body > web-header > div > div > a:nth-child(1)"
            },
            {
                url: "/measure",
                navigationSelector: "body > web-header > div > div > a:nth-child(2)"
            },
            {
                url: "/blog",
                navigationSelector: "body > web-header > div > div > a:nth-child(3)"
            }
        ]
    }
};
```

Listing B.4: Web.dev configuration

Appendix C

Case Manager Final Experimental Results

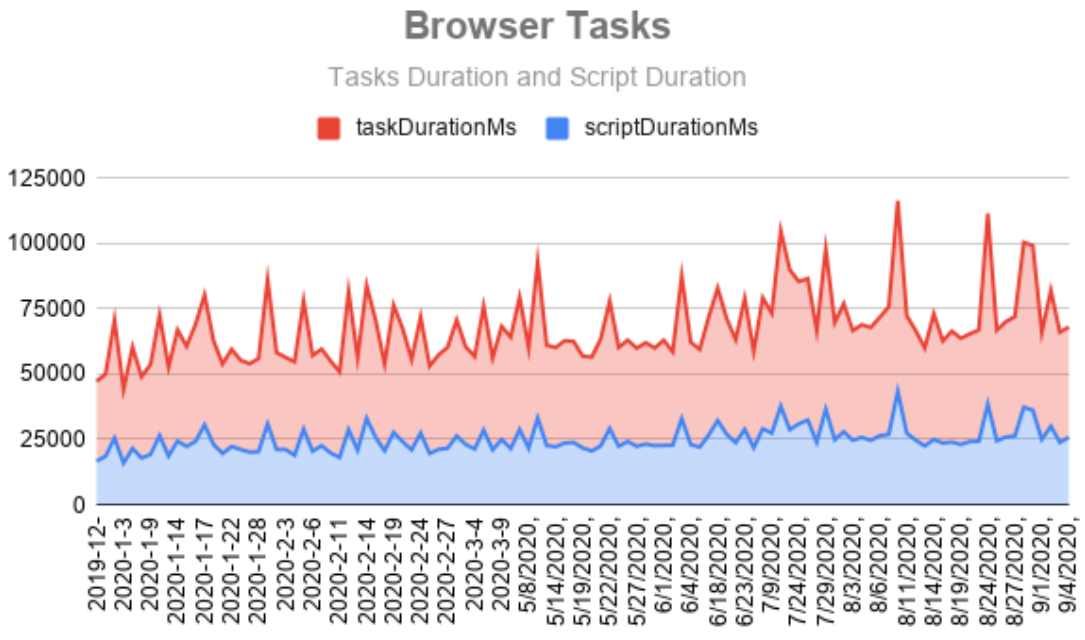


Figure C.1: Representation of browser metrics - Tasks and Script duration

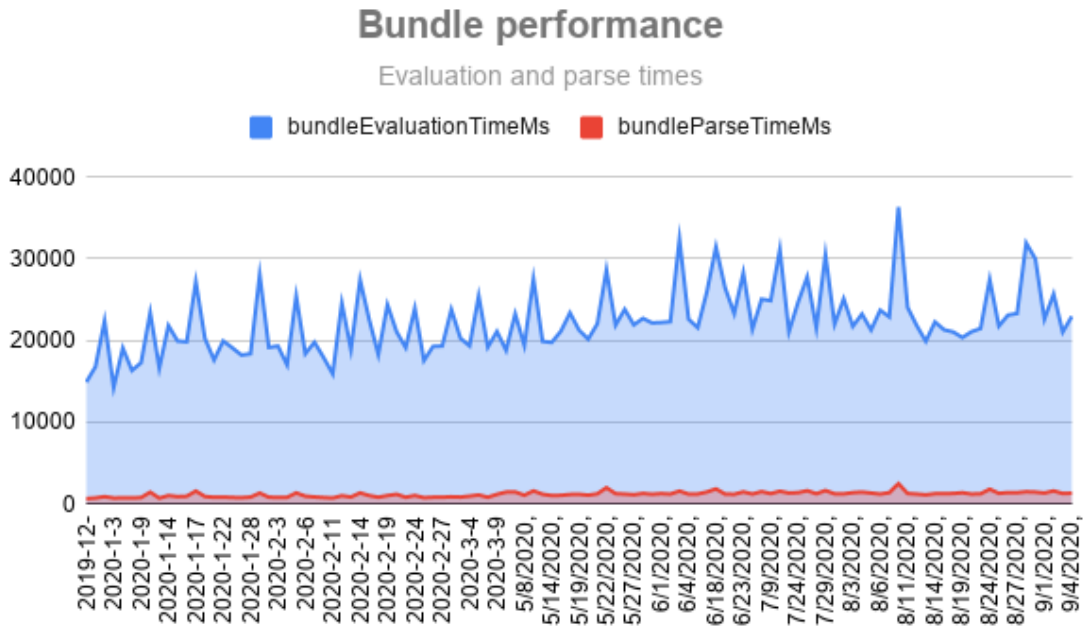


Figure C.2: Representation of bundle performance metrics - Evaluation and Parse times

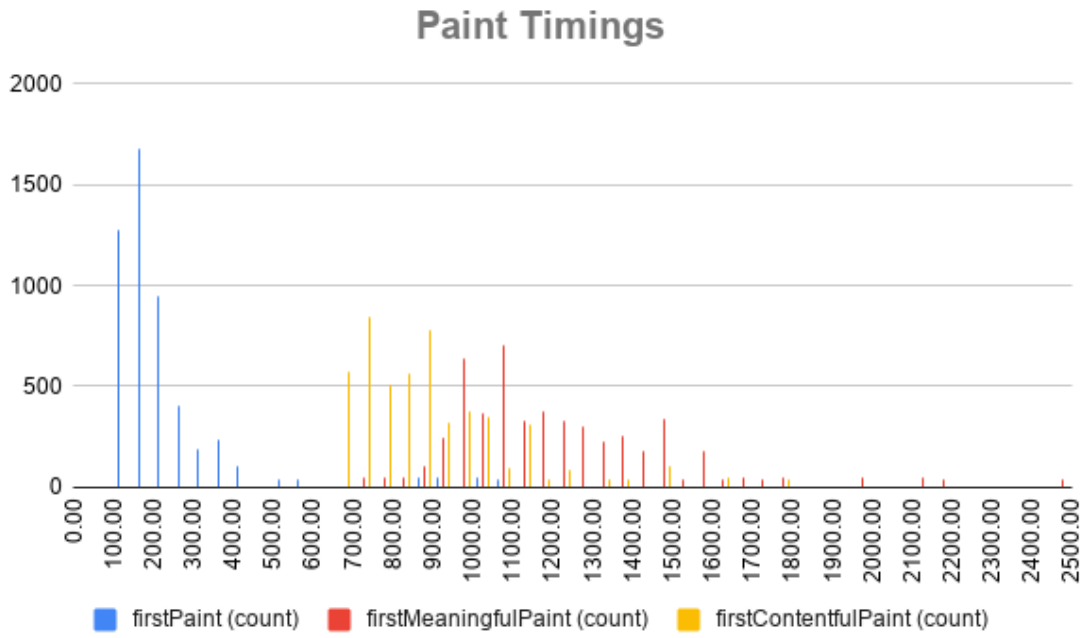


Figure C.3: Histogram that shows the number of hits for the different paint timing metrics - First Paint, First Meaningful Paint and First Contentful Paint

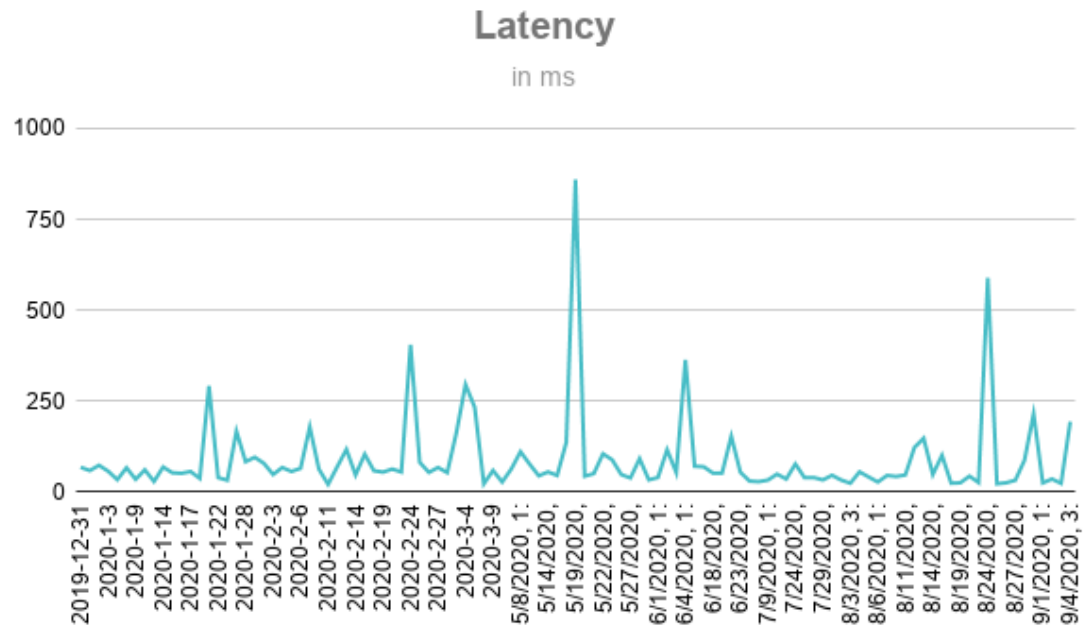


Figure C.4: Representation of the latency metric evolution.

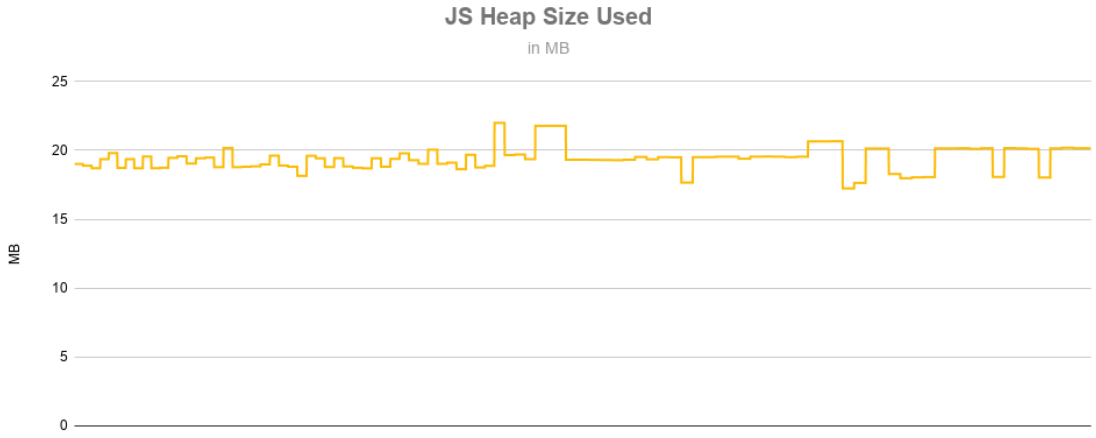


Figure C.5: Evolution of the total JavaScript heap size used, in MB.