Implementation of an Adaptable Rule-Based Dynamic Pricing Framework for E-Commerce Systems with Special Focus on AMP

Oliver Adam

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2019

Oliver Adam

Contents

Declaration iii

Abstract vii

Kurzfassung viii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Challenges . . . 2
- 1.3 Goal . . . 2
- 1.4 Outline . . . 3

2 Background 4
- 2.1 Mobile Web Applications . . . 4
  - 2.1.1 Types of Web Applications . . . 4
  - 2.1.2 Importance of Mobile Web Applications . . . 5
  - 2.1.3 Development of Mobile Web Applications . . . 6
  - 2.1.4 Challenges . . . 7
- 2.2 Accelerated Mobile Pages . . . 7
  - 2.2.1 AMP Concepts . . . 7
  - 2.2.2 Technical Aspects of an AMP . . . 8
  - 2.2.3 AMP Utilization . . . 10
  - 2.2.4 Challenges of AMPs . . . 11
- 2.3 Web Analytics in E-Commerce . . . 11
  - 2.3.1 Consumer Behavior . . . 12
  - 2.3.2 M-Commerce vs. E-Commerce . . . 13
- 2.4 Dynamic Pricing . . . 14
  - 2.4.1 Dynamic Pricing Strategies . . . 15
  - 2.4.2 Utilization . . . 15

3 State of the Art 16
- 3.1 Web Analytics . . . 16
  - 3.1.1 Tools . . . 16
  - 3.1.2 Utilization . . . 18
- 3.2 Dynamic Pricing Systems . . . 18
- 3.3 All-in-One Solution . . . 19

4 Concept 21
4.1 Requirements . . . 22
4.2 Use Case . . . 22
4.3 Framework Architecture . . . 23
4.3.1 Architecture . . . 23
4.3.2 Dynamic Pricing Client . . . 25
4.3.3 Dynamic Pricing Strategies . . . 25
4.3.4 Analytics-Server . . . 26
4.3.5 Rule-System . . . 26
4.3.6 WebSocket-Server . . . 28
4.4 AMP . . . 29
4.4.1 Extension . . . 30
4.4.2 Layout . . . 30
4.4.3 Analytics . . . 32

5 Implementation 33
5.1 Framework . . . 33
5.1.1 Dynamic Pricing Client . . . 33
5.1.2 Dynamic Pricing Strategies . . . 35
5.1.3 Analytics-Server . . . 35
5.1.4 Rule-System . . . 36
5.1.5 Websocket-Server . . . 37
5.1.6 Database - DAO and Models . . . 38
5.2 Usage of the Framework . . . 39
5.2.1 Server-Side Usage . . . 39
5.2.2 Client . . . 40
5.3 Build System . . . 44
5.4 Challenges . . . 45

6 Evaluation 47
6.1 Requirements . . . 47
6.1.1 Requirement 1: Generic Framework . . . 47
6.1.2 Requirement 2: Common Transfer Protocol . . . 47
6.1.3 Requirement 3: Server Push Capability . . . 48
6.1.4 Requirement 4: Analytics . . . 48
6.1.5 Requirement 5: Analytics Requests Processing . . . 49
6.1.6 Requirement 6: Alteration of Usability, Look and Feel of a Web Page . . . 49
6.1.7 Requirement 7: Dynamic Pricing Strategies . . . 49
6.2 Performance Analysis . . . 50
6.3 Possible Extensions . . . 51

7 Conclusion 52

A Content of the CD-ROM 54
A.1 PDF-Files . . . 54
A.2 Project Data . . . 54

A.3 Evaluation-Project Data 54
A.4 Online Literature 55
A.5 Miscellaneous 55

References 56
Literature 56
Audio-visual media 58
Software 58
Online sources 58

Abstract

Dynamic pricing is the art of changing prices of products to the right time for the right scenario to increase the profit of businesses. It is used since trading on earth exists. In the past, it was only used in the physical chain, but since the internet-era, it shifted more towards the electronic one. With the increase in performance and number of mobile devices and rise of mobile internet speed, many companies focus their effort on mobile commerce and therefore, dynamic pricing in the mobile sector.

Dynamic pricing strategies play a big role in dynamic pricing. These policies decide whenever a price of a product should change. Only with the right dynamic pricing strategies and tools, it is possible to increase the margin. By applying analytics tools, businesses are determining which policies can be lucrative and proper for a specific use case. The analytics data is also forwarded to dynamic pricing strategies that can analyze and work with it.

Currently, there are many third-party solutions for dynamic pricing available, which mostly offer APIs and standard policies like competitive pricing. But because of the complexity and scale of large e-commerce businesses and small customizability of that third-party software, many companies tend to develop their in-house solution.

This thesis gives an insight into dynamic pricing, user analytics and mobile web applications and describes state of the art in those areas. Also, it faces the problem of combining dynamic pricing and analytics to achieve a generalized real-time capable e-commerce framework which can alter prices based on environmental and user data. This framework is designed to be embedded in most of the current backend and frontend systems, especially in mobile web applications, since mobile web technology is the most used client technology in the e-commerce area.

Kurzfassung

Dynamic Pricing ist die Kunst, die Preise von Produkten zum richtigen Zeitpunkt im richtigen Szenario zu ändern, um den Gewinn von Unternehmen zu steigern. Dynamic Pricing wird schon seit dem der Handel auf der Erde existiert verwendet. In der Vergangenheit jedoch wurde es grundsätzlich nur in der physikalischen Kette verwendet, aber seit der Internet-Ära verlagert es sich mehr in Richtung der elektronischen. Mit der zunehmenden Leistung und Anzahl mobiler Endgeräte und der Steigerung der mobilen Internet-Geschwindigkeit konzentrieren sich viele Unternehmen auf den mobilen Handel und damit auf die dynamische Preisgestaltung in der Mobilbranche.

Dynamic Pricing Strategies spielen eine große Rolle in Dynamic Pricing. Diese Strategien entscheiden unter welchen Zuständen und zu welchem Zeitpunkt die Preise von Produkten geändert werden. Nur mit den richtigen Dynamic Pricing Strategies und Tools ist es möglich, die Gewinnmarge zu steigern. Mithilfe von Analysetools bestimmen Unternehmen, welche Strategien für einen bestimmten Anwendungsfall lukrativ und einsetzbar sein können. Die Analysedaten werden auch an die Dynamic Pricing Strategies weitergeleitet, die sie analysieren und damit arbeiten können.

Gegenwärtig gibt es viele Lösungen von Drittanbietern für Dynamic Pricing, die zumeist eine API und Standardstrategien wie wettbewerbsfähige Preisgestaltung bieten. Aufgrund der Komplexität und des Umfangs großer E-Commerce Unternehmen und der geringen Anpassbarkeit der Software von Drittanbietern tendieren jedoch viele Unternehmen dazu, ihre betriebseigene Lösung zu entwickeln.

Diese Arbeit gibt einen Einblick in Dynamic Pricing, User-Analytics und mobile Webanwendungen und beschreibt den Stand der Technik in diesen Bereichen. Darüber hinaus beschäftigt sich die Arbeit mit dem Problem Dynamic Pricing und User-Analytics zu kombinieren, um ein generalisiertes, echtzeitfähiges E-Commerce Framework zu erhalten, mit dem sich die Preise basierend auf Umwelt- und Benutzerdaten ändern lassen. Dieses Framework ist so konzipiert, dass es in die meisten aktuellen Backend- und Frontend-Systeme eingebettet werden kann, insbesondere in mobile Webanwendungen, da die mobile Webtechnologie, die am häufigsten verwendete Client-Technologie im E-Commerce Bereich ist.

Chapter 1

Introduction

1.1 Motivation

Before the internet was even a thing, and people and businesses could interact with each other in a matter of seconds, retailers had to put a lot of effort into getting attention and gaining customers. Early on, sellers understood that fixed prices can harm the business and can keep consumers away. Therefore, retailers used a technique called dynamic pricing where products were priced dynamically based on factors like demand, quality or competitor prices [10, 16].

In today's world, dynamic pricing hasn't changed a lot compared to centuries ago. The only thing that has altered is where it used and which factors change the price of a product. Since the internet and especially mobile internet has changed everybody's life and the way people interact with retailers, dynamic pricing is used more frequently and personalized to fulfill user expectations and drive the profit up. This change means that two humans can get different prices even if they are purchasing the same product from the same business. That is due to the improvements in technologies over the years, which can, for example, recognize behavioral patterns or nationality.

We live in a fast-moving world, where keeping up with technology and competitors is the key to success. Therefore, dynamic pricing plays a significant role for companies and especially e-commerce businesses, because e-commerce makes a considerable impact on the global market. The e-commerce area has been shifting more towards mobile commerce over the years because consumers order more products on the go. Thus, businesses take more factors into account, like GPS for location tracking.

In the past, some user analytics has been used when dealing with individuals, for example, based on the look of people, sellers determined how much they are likely to pay for a product. Today user analytics is commonly used on a technical level in web applications. For instance, businesses often track the click paths of customers and then analyze it to get a better understanding of how the people are interacting with the page. Based on the data, companies can establish a connection to their user base and can take actions or adjust their prices.

1.2 Challenges

Dynamic pricing evolved over the years, but still, a lot of challenges come with it. The timing of changing prices has to be wisely chosen because otherwise, it can drop the number of customers, for example, when they experience a price increase while in the purchasing process. Also, the frequency of changes should not be too high, like in the stock market. Customers get confused and eventually leave the product site. To change product prices, specific dynamic pricing strategies, have to be used, but only the right policies for certain application areas can increase profit. When a false strategy is used, money loss is inevitable. Critical to choosing the right strategy is getting to know the user base.

A lot of companies use dynamic pricing, but most of them utilize their in-house solutions because of the complexity of existing systems. Thus, creating software for dynamic pricing is a challenge itself, especially when its goal is to be used as a third-party application. The software has to support a lot of dynamic pricing strategies as well as offer customizability for company policies. It has to be integrable into a current running system. Frameworks can be even more challenging to create because they have to support the backend as well as the frontend and should not alter the system at all. That is why only a handful of frameworks are available.

Dynamic pricing yields plenty of challenges, but that's also true for user analytics. User analytics has to follow rules that customers comply with. It cannot collect Personal Identifiable Information (PII) but still has to provide user-specific data to the dynamic pricing system. When customer dependent pricing is used, this is only possible by omitting the identification so it cannot be traced back to a specific user.

Because the mobile segment is a big part of the e-commerce area, satisfying mobile users is essential. But a lot of companies have problems to bring the web experience of the desktop to mobile devices. Especially loading time is the number one problem of mobile web experience and can drop user engagement. One way to tackle this problem is with Accelerated Mobile Pages (AMPs). But as of today, AMPs don't support real-time web applications, which makes it hard to bring even better user experience to the end-user. For those who are running an e-commerce business, this can have an impact on conversion rates and therefore, on their margin.

1.3 Goal

The goal of this master theses is to give an understanding of how mobile web applications can be used in combination with dynamic pricing and analytics to achieve a generalized e-commerce system which is able to change prices based on user data in near real-time. Therefore, the importance of every part of this system and concept behind it is crucial to the thesis and will be covered by it. A framework has been developed, which combines analytics and dynamic pricing and is generic enough to be integrated into many existing systems independent of the database and client technology used, to underpin this thesis. It also aims to tackle the problem of developing a real-time Accelerated Mobile Page, which can track the user behavior in near real-time and react accordingly with price changes or offers. The framework consists of a rule system, which can be used to define various dynamic pricing strategies like pricing based on demand or user categorization.

1.4 Outline

Chapter 2 covers the explanation of the main foundations of this thesis, which are mobile web applications, in specific AMPs, web analytics in e-commerce and dynamic pricing. These sections give the concept behind every topic as well as the importance and utilization of the technology. Chapter 3 shows the state-of-the-art of web analytics, dynamic pricing and all-in-one solutions for both areas. Then the thesis describes the thesis project part, which begins with chapter 4. This chapter depicts the concept behind the thesis project. It covers the requirements of the framework, the use cases, the framework itself and the frontend part. After the concept the implementation part, which is chapter 5, follows. The created framework is then evaluated in chapter 6. Finally, chapter 7 describes the results found and summarizes this master thesis.

Chapter 2

Background

In this chapter, the focus is on providing a background of the four main topics of this master thesis which are mobile web applications and their impact on the global market, the concept behind Accelerated Mobile Pages, web analytics in e-commerce, precisely consumer behavior and mobile commerce, and the foundation of dynamic pricing.

2.1 Mobile Web Applications

A decade ago, it was unimaginable to browse the web via a handheld device the way we do today. With the introduction of the first iPhone and some other great inventions, smartphones became more and more powerful. But the mobile web didn't get that much attention, due to its low performance and bad user experience. Native applications, which are programmed for specific mobile devices, were just faster and had a better usability. But as of today, mobile web applications are as important as native ones. Browsers got optimized for mobile devices, and the bandwidth speed increased [4, 25].

Each day new libraries and commits by browser vendors like Google, Apple, Microsoft or Mozilla are improving the mobile web experience and pushing for a more fat-client approach. Therefore, it is necessary to know which frameworks, libraries and types of web applications are suitable for specific cases. One drawback of that is today's web apps tend to be bloated with JavaScript, which makes them quite slow on some devices [11].

2.1.1 Types of Web Applications

There are several types of web applications, but basically, there are four categories [25]:

- **Standard web apps**: They are applications which are designed for desktop but eventually work on smaller devices if they do not use browser features which are unavailable on mobile devices, like Adobe Flash.
- **Responsive web apps**: Responsive web applications consider various sizes of end-user-devices. The browser decides, based on the dimensions, which style to apply.
- **Mobile web apps**: They contain logic which is specific for mobile devices. The look and feel is typically better than standard web applications on mobile devices

because the user interface is similar to native application ones.

- **Hybrid apps**: They are written in the same language as web applications but are packed into a native application and have the same access to device-specific features. Businesses choose this option if the costs for developing native applications for multiple platforms is just too high because hybrid apps are platform independent.

2.1.2 Importance of Mobile Web Applications

The usage of mobile web applications is increasing every year and gets more important as figure 2.1 shows. The data is from SimilarWeb[1], which analyzes and collects internet traffic all across the world. The figure shows that nowadays more people browse websites with their handheld device than with a desktop computer. Therefore a mobile-first strategy is inevitable and advisable when creating a web application [40].

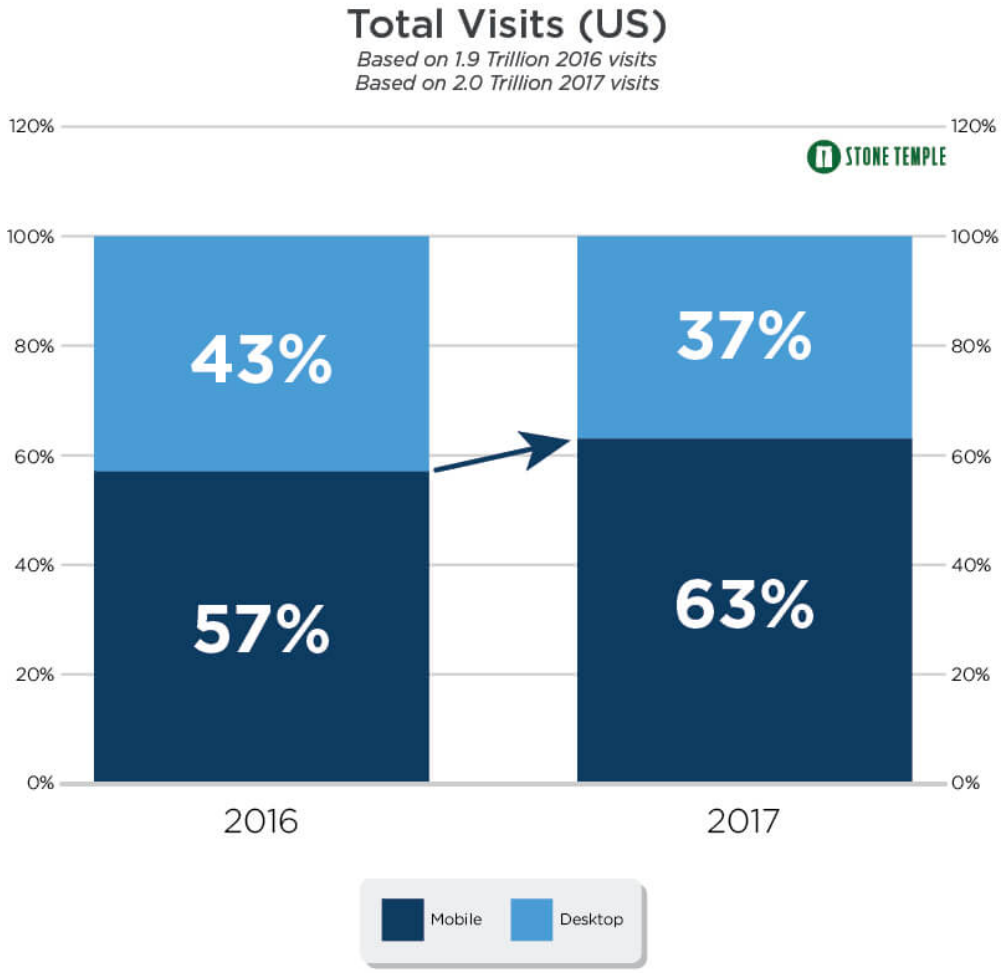

Figure 2.1: Mobile vs. desktop visits for 2016 and 2017 in the US [40].

[1] https://www.similarweb.com

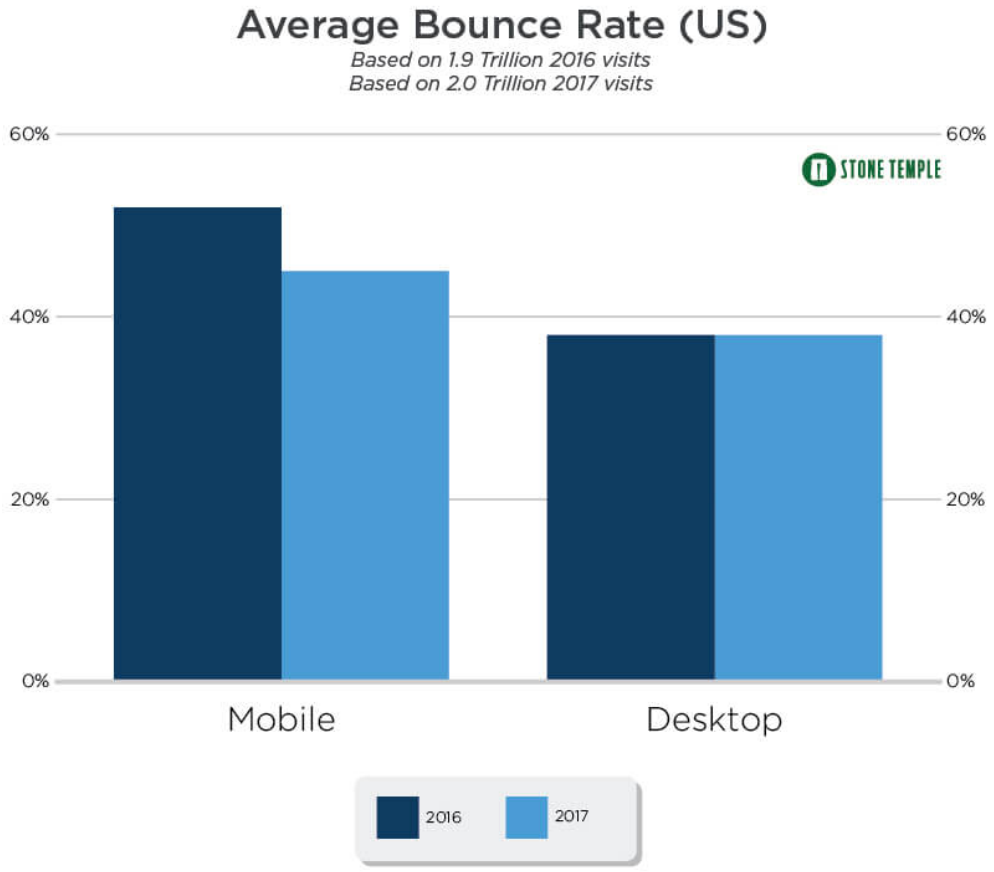

Figure 2.2: Mobile vs. desktop bounce rate for 2016 and 2017 in the US [40].

Another important fact is that the bounce rate, which indicates the percentage of visitors who leave the website after viewing only one page, on mobile is higher than on desktop, as displayed in figure 2.2. Therefore, it is necessary to provide excellent user experience because otherwise, the users will switch to a competitor site and probably never revisit the web application [40].

2.1.3 Development of Mobile Web Applications

It is necessary to understand the three essential ingredients when developing a mobile web application, as [8] states: context, information architecture and visual design. The context can be divided into two parts: the context/information a user can get when accessing the page and in which context a customer uses the app, like the location. Mobile web applications can access and use this information to give the viewer a better user experience .

The second ingredient is information architecture. It is the foundation of every web application. It defines how the information is structured and how people can access and interact with it. A web page with a good design can still fail with an awful information structure. In a desktop application, the information structure is often well handled,

but because of the different context in smartphones, most mobile pages have a lack of information structure. The smaller screen dimensions of the end-device, internet speed and performance make it difficult to show the user the information he is looking for.

The last step in developing mobile web apps is designing the user interface. Due to the small screen, users often have low expectations when it comes to the mobile web. But the visual representation is the first thing a user sees and can decide if a customer stays only for a few seconds or a longer span of time. Every device offers the capability to create a good user experience. It just depends on how well the specifics of each device is used.

2.1.4 Challenges

The most significant challenge when developing a mobile web application is bringing good usability onto mobile devices. The usability problem often occurs in navigation, representation and structure of those mobile web applications. These issues are caused by not taking characteristics of mobile devices into account like small displays, poor internet connection, other input devices and less power compared to desktop machines [13]. These problems ultimately result in inferior user experience, but seems to has improved over the years [8]. Still, supporting a whole lot of browsers and devices and making them fast and pretty enough so that customers don't bounce off is a big challenge that web developers have to face today.

2.2 Accelerated Mobile Pages

As of today, web users expect rich content, animations, smooth scrolling and high-quality pictures that load fast. On the browser level, Google and other companies are making significant progress serving those needs, but there are still websites which do not provide a good user experience. Be it low loading times, annoying ads or restructuring the web page when scrolling. That is why Accelerated Mobile Pages (AMPs) are used [32].

2.2.1 AMP Concepts

The concept of AMP is to offer a new way to implement a beautiful, fast and easy content web page without ever writing a single code of JavaScript. It is an open-source project where companies and developers can collaborate so that everyone benefits, especially users of the web [32].

As figure 2.3 shows, there are three main concepts in AMP. The base of an AMP is the AMP JavaScript library which handles all other libraries, such as social media or analytics, and has a built-in validator, so developers can see whether a page is a valid AMP or not. Libraries, which are AMP compliant, can be integrated into a web application by only using AMP HTML. AMP HTML is basically a stripped version of HTML. It has some restrictions in order to avoid bad user experience and a valid AMP at the same time. Lastly, there is the Google AMP cache. The Google AMP Cache serves AMP HTML pages, their JS files and all images included in the web page. It is a content delivery network for caching and delivering valid AMP documents to improve

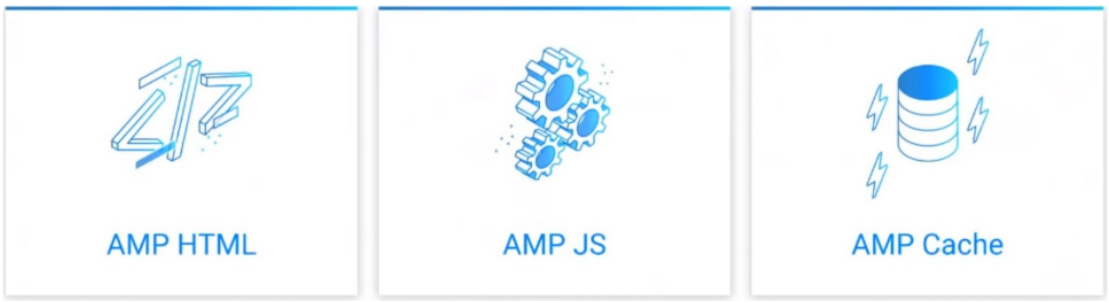

Figure 2.3: AMP core components [31].

loading time and page performance [31, 32].

Under the hood, AMP offers a "AMP.BaseElement" through the AMP library which serves as a starting point and a bone structure for every library (more details in section 2.2.2). Nearly everything can be integrated into these components, for example, a payment system or WebSockets for real-time capability. If the guidelines have been complied with, the component can be pushed to the ampproject[2] repository[3] and everyone that has an AMP can use it [31].

2.2.2 Technical Aspects of an AMP

AMP has a variety of concepts to achieve its performance and simplicity over regular web applications. These concepts consist of the AMP HTML and its components. The following sections give a more in-depth insight into these techniques.

AMP HTML Specification

AMP HTML is a subset of HTML and subsequently puts some restrictions on available HTML tags and functionalities, but it doesn't affect the way rendering engines like Chrome Engine renders its content. The AMP runtime offers custom HTML elements which primarily enables AMP to manage dynamically loading external resources, that can slow down web applications, and to extend the functionality of HTML, for instance with animations. The significant advantage of this is that the security and performance persists while allowing elements to use JavaScript, which is provided only by AMP itself [35]. These custom elements are called components. There are two types of components, the built-in ones that come with the base library of AMP and extensions to the base library that have to be explicitly included in the HTML.

The amp runtime or the base library is a JavaScript library which runs inside every AMP. It consists of the implementation of built-in components, the management system for resource loading and the validation runtime that checks if a document is a valid AMP.

[2] https://www.ampproject.org
[3] https://github.com/ampproject/amphtml

Extensions

The extensions, as previously stated, are custom HTML web components with background JavaScript running. These components always inherit from the base element of AMP, the *AMP.BaseElement*. The base element offers a lot of resource management and therefore controls when to load specific content, display it or remove it. The base element has a lifecycle which is shown in figure 2.4. This lifecycle is specific to the amp runtime, but it is approximated to the lifecycle of custom web components [33].

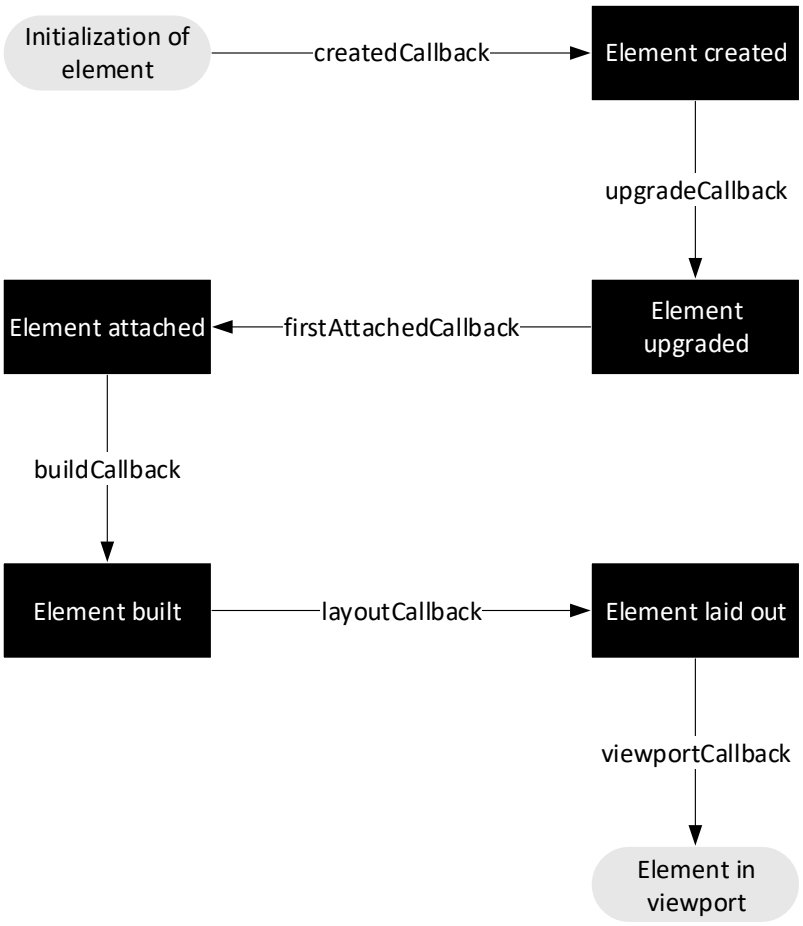

Figure 2.4: Lifecycle of an amp extension.

When the element is initialized, the constructor is called. Then the amp runtime calls the first lifecycle hook, which is called *createdCallback*. Then the *upgradeCallback* is fired, which gives the developer the chance to redirect its implementation to another *BaseElement*. If no object is returned, the component indicates that there is no redirection. When the element is added to the DOM, it can be adjusted by using the *firstAttachedCallback*. Then the most significant lifecycle hook *buildCallback* comes in play. This hook is called when the element is already attached to the DOM, but the

content has to be rebuilt. Until the content is not restored, the element is not visible to the client. This method allows manipulating and structuring the element. Next in line is the *layoutCallback* which is called when the element should perform layout, e.g. when the browser is resized. This method is used to load or reload resources. Primarily the callback will be called once when the element is created, but the behavior of the callback can be changed so that the method is called regularly when the layout needs adaption. The last callback is the *viewportCallback* which instructs the component that it has either entered or exited the visible viewport. After that, the element is in the viewport and rendered by the client.

2.2.3 AMP Utilization

As the mobile usage in the web increases, as figure 2.1 shows, also the need for a well performing and easy solution rises. Figure 2.5 tells exactly that. All across the internet as well as in the top one million web pages accessed every day, the usage of AMPs increased dramatically.

Ampproject say that AMPs are mostly used by publishers, like blogs or news pages [51]. But also e-commerce businesses are developing web apps with this technology because e-commerce makes up a large part of the mobile web consumption [30]. In fact,

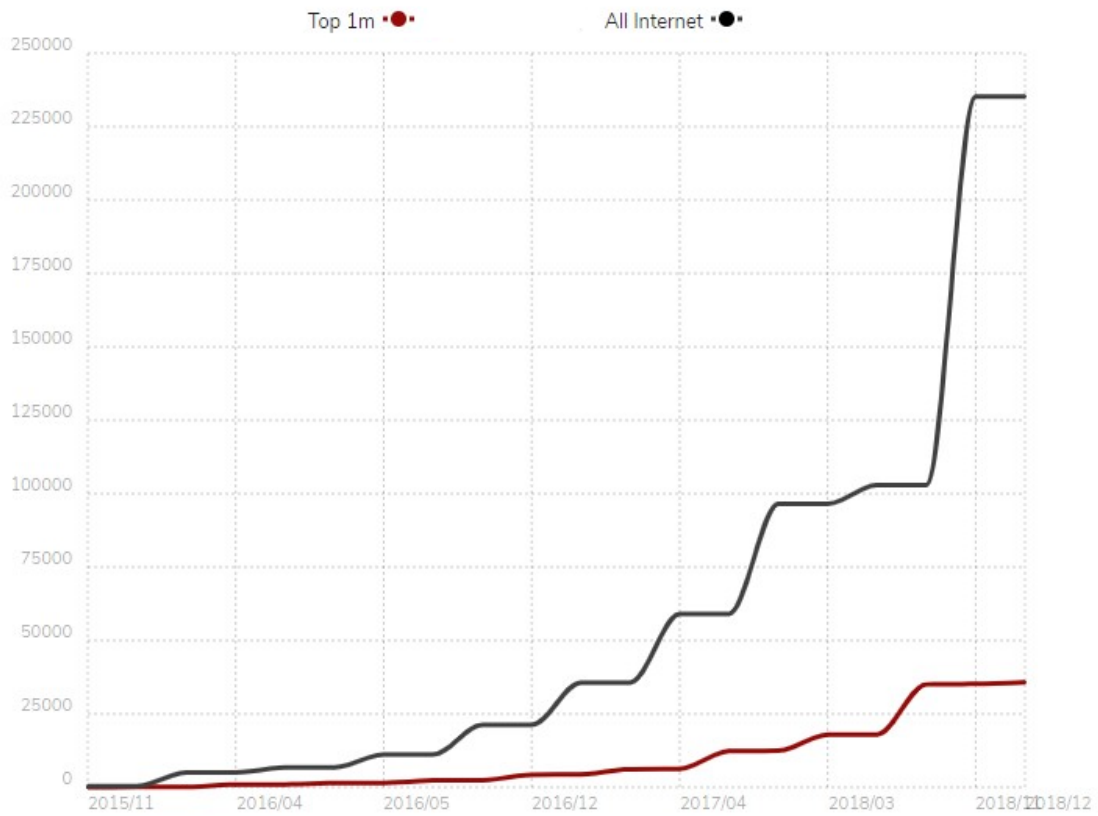

Figure 2.5: AMP Usage from 2015 till now (https://trends.builtwith.com/widgets/Accelerated-Mobile-Pages).

users spend more time on mobile e-commerce pages than in every other area. Good user experience is therefore inevitable which Ampproject claims to offer. An example of that is the case study of The Washington Post[4] [34]. Their data shows mobile users abandon their website after just three seconds when the content does not load quickly. So, The Washington Post has decided to create an AMP which has improved their loading speed by 88%. That has increased the number of mobile search users who return within seven days by 23%. Other studies of companies that use AMPs like, The New York Times[5], WIRED[6], eBay[7] or AliExpress[8] show that with making use of AMP-technology the user experience is improved which often translates into better engagement and conversions.

2.2.4 Challenges of AMPs

Despite the advantages of AMPs, there are also challenges waiting for the developers. As stated above and on the ampproject web page, AMPs are suited for static pages, e-commerce or publishers. For web app operators, who offer web games or other technologies which need third-party JavaScript code, an AMP is not ideal. Companies not only shy away from AMP because of the lack of support for third-party JavaScript, but they also have to maintain two web pages, the AMP and the canonical page [51].

Another drawback of AMP is that there are a limited number of libraries out there which are only supported by some browsers, like Chrome, Firefox, Edge, Safari, Opera and UC Browser [36]. Older customers, which are not so tech-savvy and use outdated browsers like Internet Explorer, will get a half-broken web page.

2.3 Web Analytics in E-Commerce

Web analytics is, as the Digital Analytics Association (DAA)[9], former Web Analytics Association (WAA), describes, "the measurement, collection, analysis, and reporting of internet data for the purposes of understanding and optimizing Web usage" [39].

> "If I have 3 million customers on the Web, I should have 3 million stores on the web."
>
> — Jeff Bezos, CEO of Amazon.com

Every customer has different needs, but it is just impossible to create a shop for every potential customer there is. That also implies that companies have to shift from a world of mass production of standardized products to a broad variety of products to meet multiple needs. This diversity of products results in a whole lot of choices, which can be a problem. Businesses are solving that by changing their representation respectively to the needs of their customers, which ultimately addresses the overflow of

[4] https://www.washingtonpost.com
[5] https://www.nytimes.com
[6] https://www.wired.com
[7] https://www.ebay.com
[8] https://www.aliexpress.com
[9] https://www.digitalanalyticsassociation.org

information a customer can get. To understand the needs of consumers, companies need Web analytics in order to gather data and analyze them.

With the rise of the internet, more people are buying products online than ever before. Customers can easily browse endless lists of products, compare prices and switch over to other online shops. The e-commerce area is highly competitive, and therefore, many companies are betting on web analytics to see consumer behavior as well as the reason why customers are visiting the page [12, 24].

2.3.1 Consumer Behavior

Discovering consumer behavior and the reason for buying specific products is not that trivial as it may look like, but it is vital to web analytics, as [12] states. In research, variables represent events that occurred where every variable can have a different value. In web analytics, that variable can be the duration of a session or number of clicks. That is why it is necessary to understand the research question driving the study and the context in which it takes place . As B. Jansen argues, a behavior is at its most basic "an observable activity of a person, animal, team, organization, or system" [15]. He also claims that behavior is an "overloaded term" because it addresses a big spectrum of actions. Therefore, it is difficult to characterize such behavior. But there are three categories in which behaviors can generally be classified, as described in [12]:

- Behaviors can be detected and recorded.
- Behaviors are tied to a specific goal other than specific observable actions.
- Behaviors are reactive responses to environmental stimuli.

From the consumer behavior standpoint, there are two stages in online shopping: Getting information about products and purchasing the products. These two stages combined, represent the major part of long-held consumer behavior models [22]. But because of the wide variety of products and immense navigational options and actions, it is a very complicated task to process these stages and derive behavioral patterns, as stated earlier [12]. Once patterns are found, they can be used to identify four browsing behavior categories [19]:

- **Directed buyers**: People that have a specific product in mind.
- **Search/Deliberation visitors**: People that have a general product in mind.
- **Hedonic browsers**: People who have not a specific product in mind but buy impulsively based on their visit experience.
- K**nowledge-building visitors**: People who are just looking for information about a product.

To capture those patterns and further process it, the recognition system needs user data. One way to achieve this is with click-stream data. Click-streams are basically web server logs which log every event occurring on the homepage [12]. So not only the purchase events are tracked, but also non-purchase data is gained [20], like the sequence of visited pages and the number of times pages are viewed [23, 26]. The click-stream data can help companies to determine how the user experience is, how users move from site to site and track users' exposure to advertising. Also, it is an authoritative source of information on customer behavior and is nearly the best predictors when it comes to online behavior [21]. With the data, existing studies have analyzed online behavior such as identifying

customer goals, information search and usage and customers with a high possibility of buying products [19].

2.3.2 M-Commerce vs. E-Commerce

Due to the increase in mobile activity e-commerce companies are aware of the possibilities that come with it. The term in which those users conduct commerce is called mobile commerce, or short m-commerce. As figure 2.6 shows, the m-commerce area has gained a lot of attention in the year of 2017 (figure . The area of shopping has increased by 54% in usage sessions over the year on mobile devices, and companies want to jump on the train.

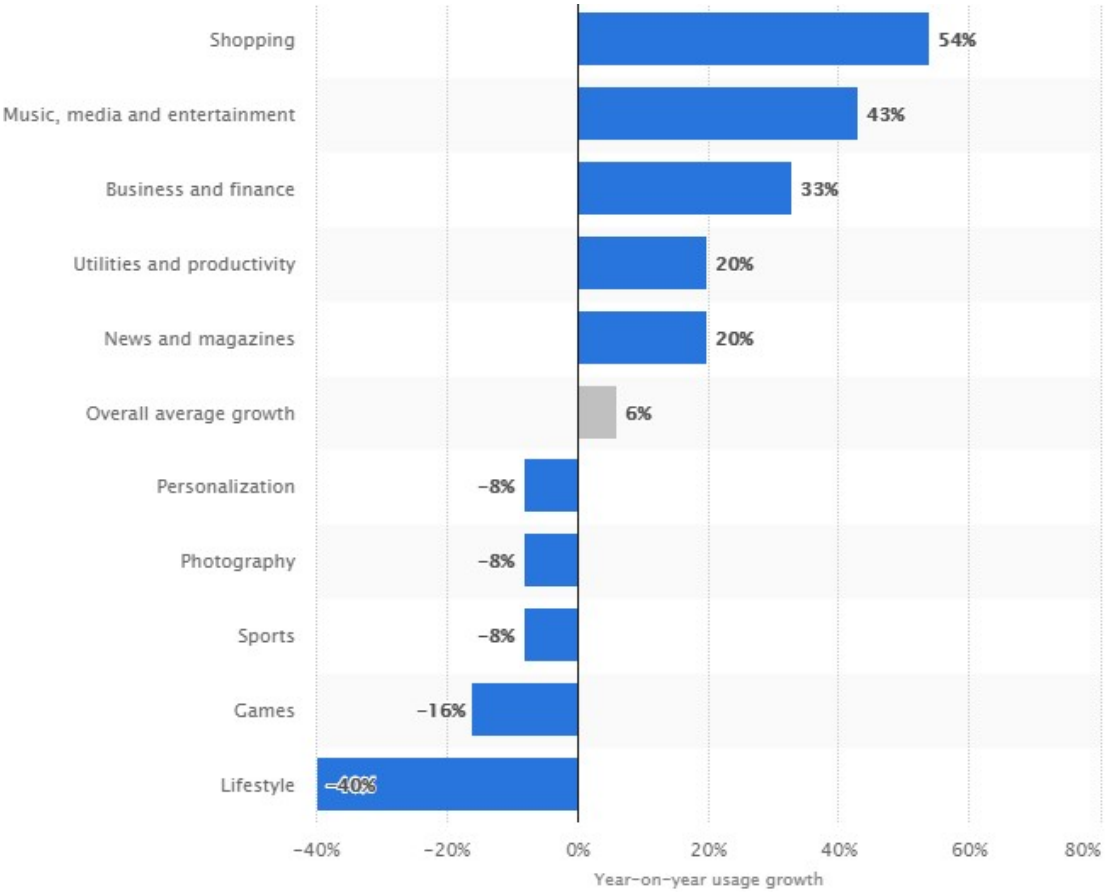

Figure 2.6: Annual growth in time spent per mobile app category in 2017 [48].

In the e-commerce area, PC and mobile devices have similarities because they have the same kind of information access. But the experience on mobile is different. M-commerce can be done everywhere and at any time. That means that for example, a customer in a local store can compare prices and products or even buy it from a competitor. This possibility brings new challenges for companies. Also, the smaller screen on handheld devices is restricting the information a customer can retrieve. Mobile commerce brings a lot of additional data, such as time- or location-related information.

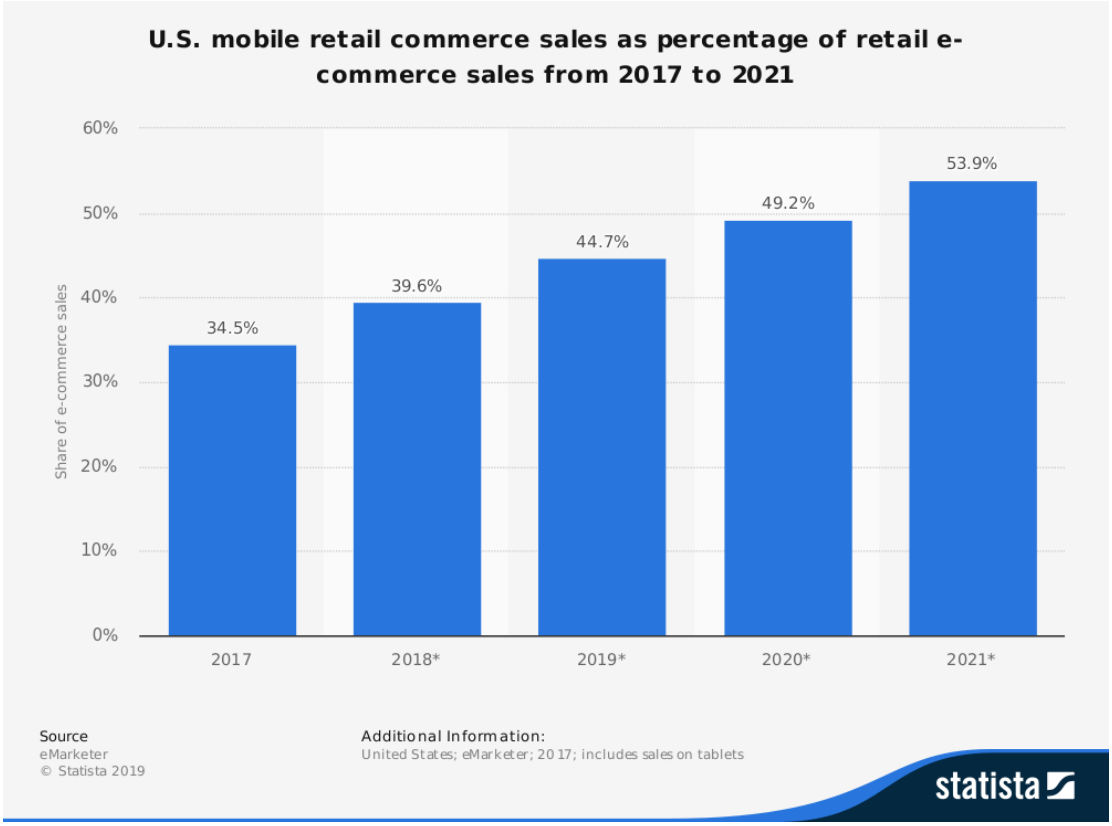

Figure 2.7: U.S. mobile retail commerce sales as percentage of retail e-commerce sales from 2017 to 2021 [50].

Although the companies can use the data to learn more about their customers, it also results in more privacy and security problems they have to deal with [23].

As figure 2.7 shows, the m-commerce market will be exceeding the e-commerce market in the next few years. Companies know that and are putting more effort into analyzing mobile user behavior, as the results in [23] show. The report indicates that mobile users often take a more task-oriented behavior, whereas desktop users are more exploration-oriented.

2.4 Dynamic Pricing

Because of the fast-increasing numbers of online shoppers, companies try to give customers a better e-commerce experience and maximize their margin by altering prices. Changes to the price are seen as a critical step which can be crucial for the success or failure of an e-commerce business. These variations of prices are also called dynamic pricing [10, 12, 18].

Dynamic pricing allows suppliers to reach more people, reduce costly intermediaries and increase revenue and decrease costs by eliminating inefficiencies, lowering overhead costs and increasing inventory turns. Companies not only profit from that but also

consumers who compare prices or buy products in high amounts can get a better deal [18].

2.4.1 Dynamic Pricing Strategies

The foundation of dynamic pricing are strategies which are applied onto certain products and markets that will result in changes of prices, for example when the departure time of a flight gets closer the willingness to pay for a flight ticket increases, and therefore the price will change. Dynamic pricing strategies take a lot of various factors into account, such as demand, competitor price, consumer behavior, time (e.g. flight tickets) or many others [6, 16, 29]. Usually, dynamic pricing strategies are based on the application area in which a company is operating in [10].

Consumers are aware of those strategies, and they time their purchases, like when waiting for the after-Christmas sales, tracking price mark-downs of fashion goods and electronic articles and many more, as [6] states. Although consumers are paying less for specific products, companies still can increase their profit by using dynamic pricing strategies to lure more customers in [29].

2.4.2 Utilization

Dynamic pricing is used in e-commerce, but before the internet era and the virtual chain (information-based environment), it has been in use in the physical chain (product-based environment). Based on demand and other factors prices have been changed, but not that frequently like on the internet. The many ways information can reach customers and the simplicity of changing the price tag of a product on the internet can help to lower menu costs, for example, the costs of setting and changing prices [16].

These price changes often occur in very different areas in which companies operate. The most used area for dynamic pricing is the retail sector. Alteration of prices are specially introduced for sport and fashion goods, which are seasonal. Companies want to sell their base inventory as fast as they can in the appropriate season because otherwise, these products are worthless. Also, other goods which are getting lower attraction over time, like tech-products or groceries, are perfect for dynamic pricing [10].

As most people know, airlines use dynamic pricing for their flight tickets. In contrast to retail stores, prices will not decrease as time passes, but increase. This strategy is also called Markup Pricing. This is mainly due to the fact that wealthy business travelers have considerable valuation risks and therefore tend to book them at short notice, as [10] states. Price sensitive people, on the other hand, buy their tickets long before the time of departure, because the risk of being booked out prevails. Dynamic pricing is also used in other areas, like the automobile, hospitality, oil industry and many more.

Chapter 3

State of the Art

Because of the increasingly competitive electronic market, customers can easily change to a different e-commerce business when their needs are not satisfied. As a consequence, companies have to fully understand why customers have or have not bought a product and what went wrong in acquiring an item. Knowing the behavioral pattern of those customers is the key to answering those questions. Customers can freely browse the page and take multiple steps until they commit to products or leave the site. That's where web server logs, data mining and analytics tools come in handy [12].

Today a lot of companies try to maximize their profits by understanding their user base and trying to fulfill their needs. Analytics tools and dynamic pricing play a significant role, and they are gaining more and more attention.

In the following sections, current analytics tools as well as their utilization, dynamic pricing and the combination of both will be described.

3.1 Web Analytics

3.1.1 Tools

One of the major players in web analytics is Google Analytics[1]. As of today, nearly 27.4 million websites use Google Analytics to track user behavior and monitor their customers. In the top one million sites, there are 21% who are using the tool from Google (figure 3.1). Google provides a real-time capable solution for tracking and analyzing data of a site which can be changed individually for a company's goal [5].

As figure 3.2 represents, Google has various components that interact with each other to provide the developer a rich user interface, client libraries and APIs for using analytics. Google Analytics is organized into four main components: collection, configuration, processing, and reporting [44].

Also, Facebook with its 2 billion user community is engaging this problem with its in-house solution called Facebook Pixel[2] which can be connected to the Facebook Analytics tool, as stated in [41]. Like Google, Facebook wants to give companies a better understanding of the type of customers who are visiting the web page and what kind of

[1] https://analytics.google.com/analytics/web

[2] https://www.facebook.com/business/learn/facebook-ads-pixel

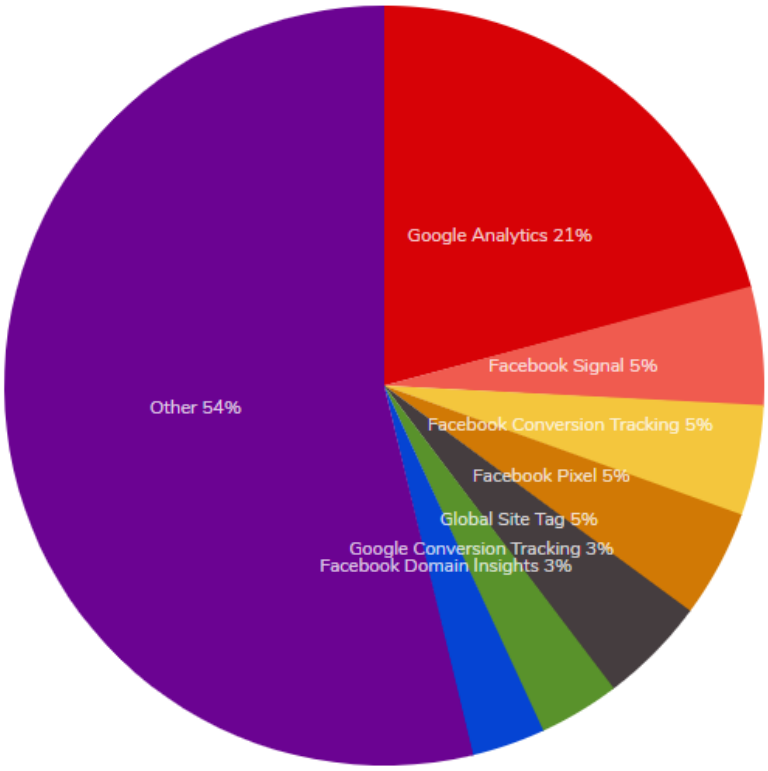

Figure 3.1: Analytics tools usage distribution in the top 1 million sites [38].

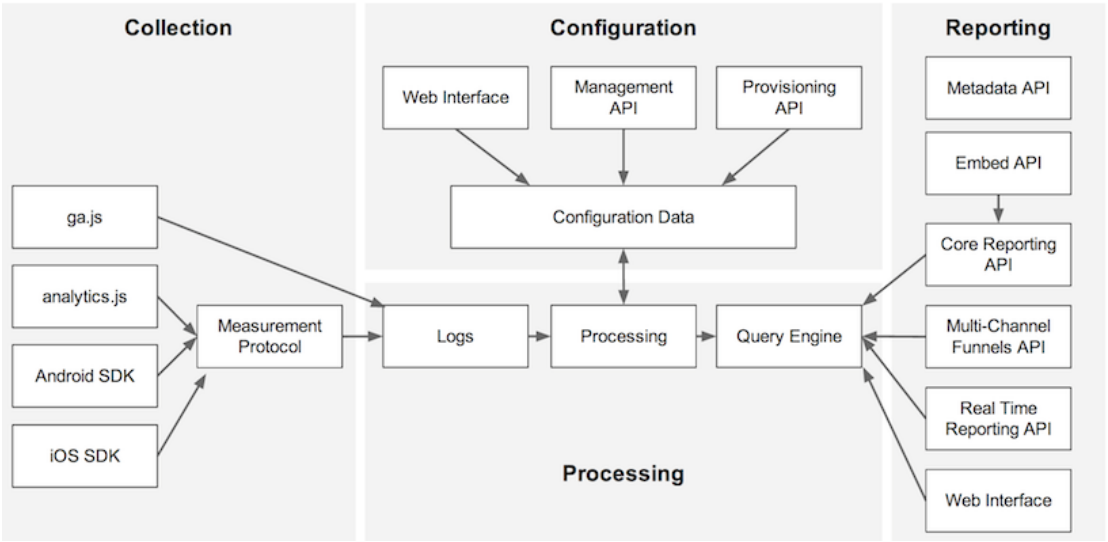

Figure 3.2: Google Analytics components [44].

devices they are using. Based on the data and visualization, it should be easier to make a smarter strategic decision for a company.

Facebook provides a full navigation hierarchy of the consumers, which shows how people interact with the page. Also, it can segment the customers into various groups, based on demographics or the behavior, so it easier to understand the different groups that interact with the page. Although it is mostly used on web pages, it can also be set up to monitor Facebook pages.

As figure 3.1 shows, there are a lot more tools, which have all nearly the same goal but try to achieve it in different ways.

3.1.2 Utilization

Web analytics is used in a variety of areas, but most commonly, it is applied in the e-commerce field. For instance, recommendation systems are widely used by e-commerce sites to show costumers products they might be interested in. One such an e-commerce site is Amazon.com[3]. Recommendation systems require a lot of user data to operate. For that reason, web analytics tools are inevitable [24].

Before analytics data can be analyzed, it has to be extracted from a data source. Some apps use web server logs and other applications data from analytics tools like Google Analytics, as shown in [3]. They use the browser information of the user, the pages users have been visiting before, and the keywords used to find the web page. With this information, it is possible to improve user experience and usability and thus increase user acquisition.

On the other hand, some projects specialize more on gathering information on products users have been looking at [26]. They are interested in browsing paths, frequency of page visits and the time spent on product categories. Based on that data, they derive the consumers' interests in certain products and categories. [14] describes how to use web server logs to improve understanding of elder self-care behavior and to classify it into three distinct cluster types. The usage of data cleaning, data processing, and assigning the data to a specific session is inevitable as many other papers like [12] show. Others go even further by using eye tracking to understand if there is a correlation between eye movement and acceptance and perception items [28]. Eye tracking could help to enhance the end-user performance and experience, by combining mouse and eye-tracking data to understand what happened in between clicks and reveal more about the users' cognitive processes.

3.2 Dynamic Pricing Systems

Dynamic pricing is a very active field where e-commerce businesses try to maximize their profits by analyzing customers and competitors. The definition of dynamic pricing varies a lot between key players in this segment, but today it is a business strategy in which prices are modified frequently by channel, product, customer and time [10]. The common areas where dynamic pricing is used are airlines, stock markets, electric utilities and gas stations but also more and more in e-commerce businesses like Amazon [6, 7].

[3] http://www.amazon.com

In the past, companies used basic models to determine the price of a product, like stochastic (e.g. Bernoulli Model) or deterministic models [10]. But as of today, the increase in artificial intelligence and machine learning in our everyday life led to an increase in AI-powered dynamic pricing. Transportation systems like Uber[4] or Lyft[5] are using real-time dynamic pricing which have their roots in the airline industry. Uber and Lyft use dynamic pricing in the following way: "Users see different prices in different areas of a city and at different times of day, dynamically adjusted based on real-time data on driver supply and predictions on customer location, traffic, weather and so on" [49]. This strategy is based on the well-known model of supply and demand. These policies are being applied, for instance, when some venue is happening in the area. There is an increased demand for cars, and so Uber and Lyft push the price.

In AI-powered dynamic pricing, heuristics often play a significant role. Heuristics aim to predict what the best price for a product is to increase the margin. Such an approach, which optimizes revenues for vehicle ferries, proposes [2]. They aim to "maximize the revenue obtained from the sale of vehicle tickets by varying the prices charged to different vehicle types, each occupying a different amount of deck space". The prices do not only change based on the size of the vehicle, but also by taking the season and time left until departure into account. The results show that on average, the revenues with existing approaches were 6% lower than with the proposed one. [27] have faced a similar problem. They introduced a new dynamic pricing model for parking reservations, also aiming to maximize the profits of the parking manager. The pricing strategy is to lower or increase the price based on demand. The demand is time-dependent as historical records show. The authors argue that when the demand is high, drivers care less about price and more about availability, which ultimately results in higher revenue when the price goes up. Simulation results even tell that not only the margin is maximized but also the use of the parking resources during peak periods. Therefore, it significantly saves customers' cruising costs.

3.3 All-in-One Solution

Dynamic pricing has to go hand in hand with analytics. Although dynamic pricing is used in quite a lot of companies, only a limited number of frameworks is available, due to the complexity of existing systems and various dynamic pricing strategies companies want to use. On the other hand, there are many standalone software solutions, which can be used in existing systems. But they often lack support for customization, which is usually required.

IBM is one of the vendors which offers an all-in-one solution for analytics and dynamic pricing. But they are focusing on competitor pricing instead of user analytics. Vendors and retailers can send their competitor prices to IBM. That data is then used for recommending prices for products. With a rule-based system, it is possible to set various rules which change the way how the system suggests prices. The prices will then be sent to a server. The information can be used for updating the prices on a mobile application or a website in real-time [42].

[4] https://www.uber.com
[5] https://www.lyft.com

A more sophisticated approach offers priceedge[6]. As IBM's solution, priceedge has a standalone cloud software solution for finding the optimal price for products. But instead of only taking competitor prices into account, an AI-engine analyzes the company's e-commerce pricing and then suggests prices based on product costs, stock rates and other data. It then creates automatic pricing rules which should fit the goal. With an API or tracking script, priceedge's software can be integrated into an existing system [46].

Omnia[7] has the same idea as priceedge but differ in the AI technology they provide. Omnia's AI incorporates data from internal systems (e.g. product and sales data) with data sources like Google Analytics to create optimal prices for the consumers. The core algorithm of the AI decides if the price needs to change based on the price elasticity of a product. It is a self-learning algorithm which evolves [45].

Spree Commerce[8] is an open source e-commerce framework which can directly be integrated into an existing system. It runs on Ruby on Rails and provides an e-commerce platform out of the box. Due to its large community, Spree Commerce offers a lot of third-party integrations and features like dynamic pricing. It does not involve any AI but instead is highly customizable, which is required by many companies [47].

[6] https://www.omniaretail.com
[7] https://priceedge.eu
[8] https://spreecommerce.org

Chapter 4

Concept

The concept of this master thesis project is to create a framework which combines analytics and dynamic pricing and is generic enough to be integrated into many existing systems independent of the database and client technology used. Figure 4.1 shows the architecture of the project, which consists of two environments. The first is an e-commerce web application which is built with the AMP technology and tracks the user behavior. The second environment consists of a server that communicates with the web pages to analyze the user behavior and send product information, real-time offers or price changes.

The framework consists of a rule system, which can be used to define various dynamic pricing strategies like pricing based on demand, conversion rates or user categorization. These strategies can all be connected in every possible way and are also extendable. That allows for the definition of customized rules.

The following sections describe the various use cases in which the framework can be

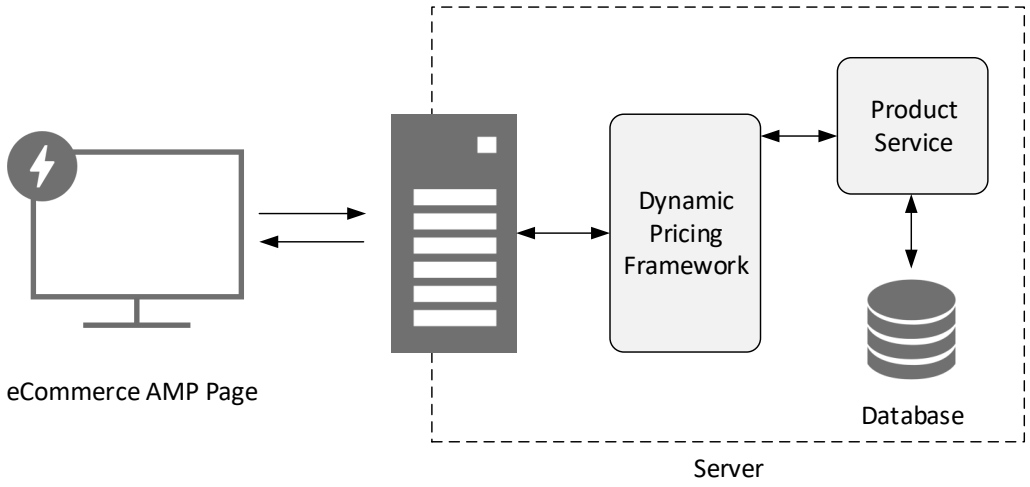

Figure 4.1: Project architecture.

used, as well as the structure of the framework itself. Other than that, there is a small insight into the technical aspects of AMPs and the concept behind the client-side AMP.

4.1 Requirements

The dynamic pricing framework should achieve specific goals and meet various requirements. The following list shows the requirements for the overall system:

1. The system has to be generic in such a way, that it is possible to integrate a variety of existing database technologies. The framework has to work in e-commerce systems while still being independent of the database architecture in use.
2. The system has to be able to send product information via a WebSocket so that the products can be displayed with any given client technology.
3. The framework has to send updated product information in web real-time so that newly priced products are displayed on the client instantly.
4. Analytics requests have to be accepted via REST and the amp-analytics[1] in-house protocol, which defines how the requests should be structured.
5. The framework has to process analytics requests immediately to shorten the time in which products can be updated.
6. The usability, look and feel of a web page are not to be altered by the integration of the dynamic pricing framework.
7. The system has to offer common dynamic pricing strategies (rules) and a basis for building customized simple and complex strategies that can also be concatenated.

4.2 Use Case

The workflow of the application is straightforward, as shown in figure 4.2. A customer visits the AMP and looks at different products. The behavior of the customer, like the click path and interactions on the page, will be sent to the analytics service for further analysis. Based on the user's product choice, the behavior of the user or other environmental data like the demand or conversion rate, the framework decides if specific rules in the framework apply. If one rule applies, then the price of products can be changed via the product service. These price changes are sent to the web application which updates the products in the view. In the ideal scenario, the customer is pleased with the price and purchases the product. If the customer buys the product, the system will recognize the purchase and stores the data in the database to use it for further analysis.

The steps that the customer takes before purchasing a product are looking for items and selecting one. These steps can happen in a short period. Therefore, it is necessary that the analytics service and the product service are connected in real-time so that price changes are occurring immediately after detecting behavioral patterns.

[1] https://amp.dev/documentation/components/amp-analytics

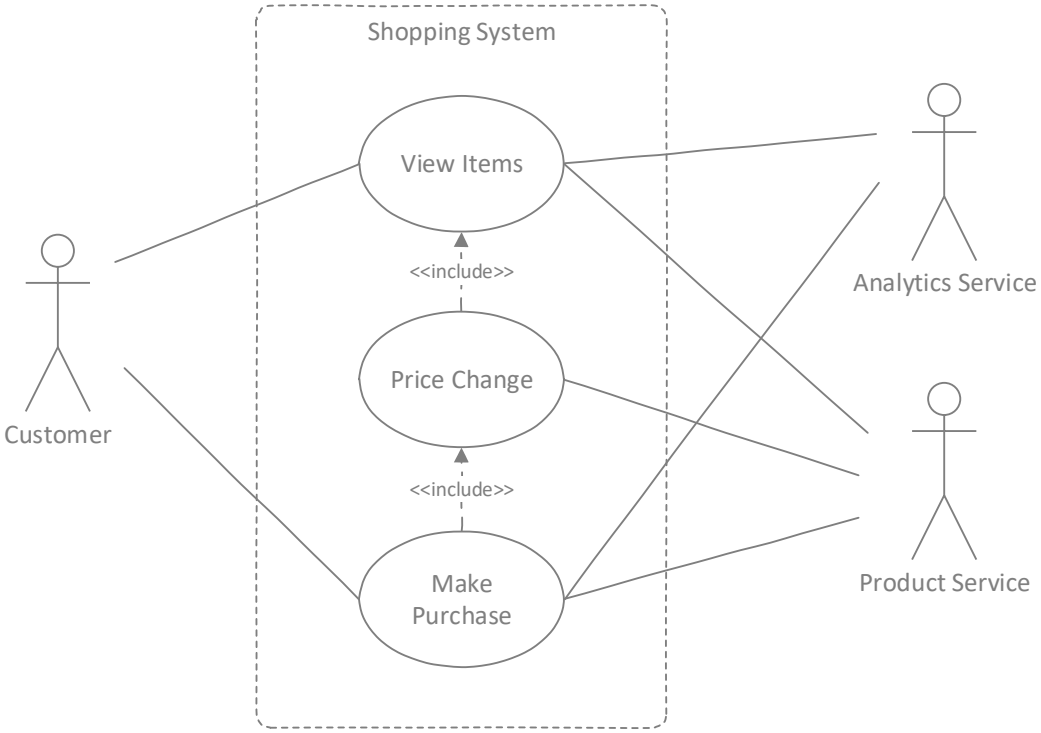

Figure 4.2: Use case diagram of the purchase process.

4.3 Framework Architecture

The framework is the base of the application. It handles all incoming requests for analytics and products, responses with suitable data, manages dynamic pricing strategies (rules) and retrieves data from the database if required. It can be used on any server, which supports JavaScript, with any database. Also, the client used is irrelevant, which makes the framework very generic.

The following sections give a more detailed description of the architecture and the parts the framework consists of.

4.3.1 Architecture

As figure 4.3 shows, the entry point of the framework is the *DynamicPricingClient* class. It is the first entity that will be initialized. The framework handles every part that is related to the framework (see section 4.3.2). It takes various configurations such as port numbers for the servers and a database object, which is used to access data. The rules for the dynamic pricing strategies are also located in the dynamic pricing client (see section 4.3.3). After the initialization, the *ServerManager* is created. Because the *AnalyticsServer* and the *WebsocketServer* have to interact with each other, the server manager handles the connection between these two and forwards calls to their methods (see section 4.3.4 and section 4.3.6).

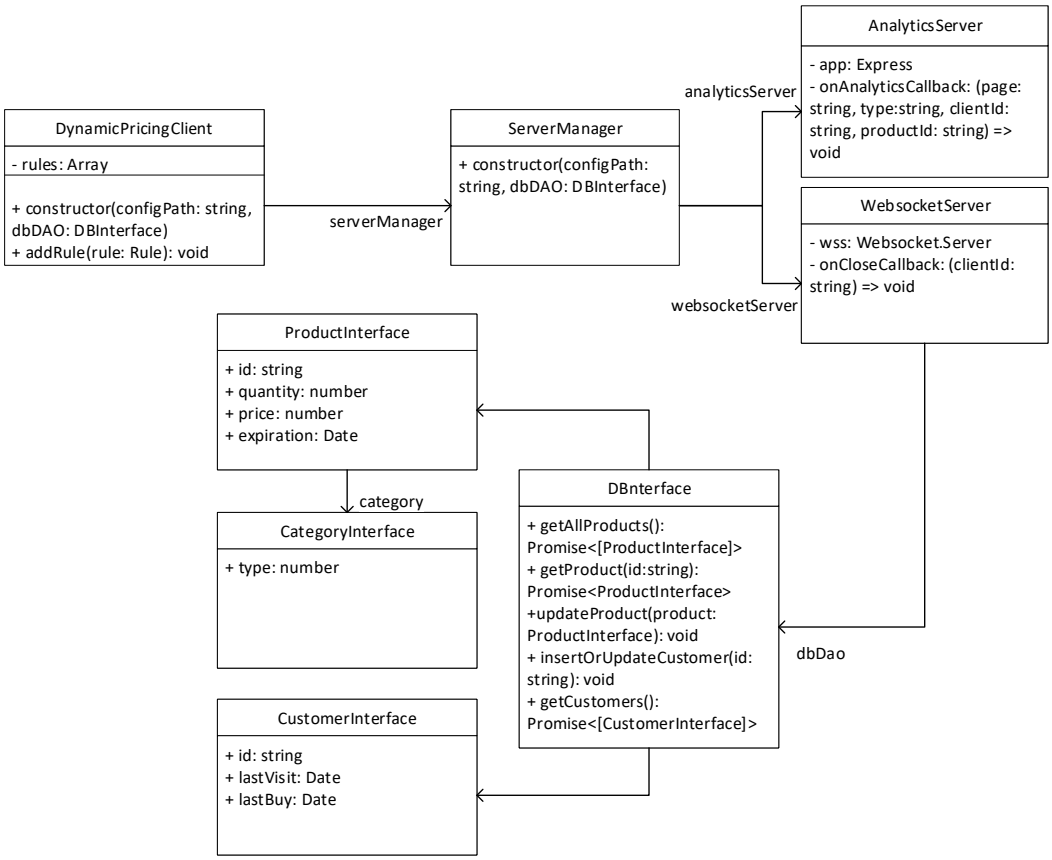

Figure 4.3: Simplified UML diagram of the system.

Database technologies usually differ from business to business. Thus, it is necessary to offer a way to generalize the connection to the database and retrieval of data. That is why the Data Access Object (DAO) pattern is introduced. The DAO pattern is a structural pattern which allows separating the application layer from the persistence layer (usually the database). That allows the two layers to evolve and change separately without knowing the internals of each other [1]. In the project, the DAO pattern is used for the database. The Data Access Object inherit properties and methods of the corresponding interface. That makes it possible to extend the database, so that the database technology can be changed. The database DAO uses interfaces of the domain objects (models), like *ProductInterface* or *CustomerInterface*. These interfaces are important, because various properties, which hold product data, have to be present in the database models for the validation of rules. Otherwise, it would be impossible to determine if, for example, the availability time of a product is below a certain threshold.

That still preserves the standard rules and functionality of the framework without altering it. The *WebsocketServer* class sends product information and updates to the client. It also uses the DAO for the database, which is an instance of a derived class of the *DBInterface*.

4.3.2 Dynamic Pricing Client

The *DynamicPricingClient* is the main entry point of the framework, as figure 4.3 shows. It provides configuration possibilities for the framework itself like setting port numbers for the analytics and WebSocket server, setting dynamic pricing strategies or sending updated product information to the user. This class is the main part the developer interacts with when using the framework.

The *DynamicPricingClient* class not only provides interfaces for the framework, but it also processes incoming analytics requests and sends them to rule processing unit for validation purposes. Also, the dynamic pricing client is interested in the closing state of a WebSocket, because it can affect the status of a rule, for example, a demand rule (see next section).

4.3.3 Dynamic Pricing Strategies

Dynamic pricing strategies are essential to companies and can differ across multiple application areas. That is why it is important to make rules very flexible in their usage as well as offer some basic ones. For the master thesis project, the following dynamic pricing strategies were chosen:

1. **Demand Rule**: In the demand rule, the cap for the demand for a product can be set. If the demand surpasses the threshold, the system will respond by increasing the cost of the product. That is done by detecting how many people are visiting the same product page.
2. **Time Rule**: The time rule detects if the availability time of a product drops below a certain threshold. This rule is independent of user behavior. Therefore, it is necessary to monitor the availability time of a product constantly.
3. **Conversion Rate Rule**: The conversion rate depends on the purchases of visitors. It is calculated by the number of purchases made by customers divided by the overall users visiting the web application. If the conversion rate is low, the system will effectively lower the price so that more visiting customers are buying products. If it's high, the price will increase.
4. **Category Rule**: When the system detects that a user is searching for a specific category, like cosmetic products, the system will respond by giving discounts on certain products in this category. That can increase profit because studies have shown that for example, women, who mainly buy cosmetic products, are more likely to buy products, which have some discount on it. In general, this means that the system can respond to various groups of customers and change prices based on their likely behavior to eventually increase the margin.
5. **Extendable Rule**: The given rule set may be not sufficient for big and complex companies. Therefore, it is possible to implement custom rules on custom conditions. When these are met, the developer can decide which actions to take next.

4.3.4 Analytics-Server

The analytics server uses the REST protocol and provides an in-house solution for analyzing user behavior. It is used to acquire data and transmit the data to the dynamic pricing client for further processing and rules validation. But to do that, analytics requests have to arrive in a certain format. This format is based on the amp-analytics component, which sends occurring events in a POST request. The data, which is used to analyze behavior, is stored in query parameters. These query parameters have to have a certain name so that the system can determine which actions the customer has made. There are 4 query parameters which are necessary to analyze customer behavior:

1. **Cid**: Cid is short for client identification. The system uses it to identify which customer is making which actions and to eventually change prices for that specific customer. To always be able to identify customers, it is inevitable to store the client id as a cookie on the client's browser.
2. **Page**: The page is used to determine which web page the customer is currently visiting.
3. **Type**: Because different actions can happen on several pages, like looking at an item or buying it, the type parameter has been introduced.
4. **Pid**: The product id is used to determine the product a customers is looking at or buying.

When requests arrive, the analytics server makes use of callbacks (figure 4.3). Therefore the server uses the command pattern. The command pattern is used when it is necessary to call objects without the knowledge of the receivers' intents or structure. That results in decoupling the sender object from the receiver object [9, pp. 233–237]. In some languages like TypeScript[2], this can be done with a callback function. This function is called, when needed, in this case, when an analytics request is received. The request is forwarded to the dynamic pricing client via the callback. This is necessary to avoid circular dependency because the dynamic pricing client has an analytics server object. The callback is also used to support asynchrony.

4.3.5 Rule-System

The rule system is designed to validate existing rules by analyzing user behavior or environment variables, like current weather. But before the system can validate rules, it has to gather analytics related data. Figure 4.4 shows a sequence diagram of this process. The customer, for instance, clicks on a product and the analytics server gets a request representing that action. By forwarding the information about this action to the dynamic pricing client, the server signals the rule system, that it has to reevaluate all rules. If the status of a rule changes, the developer, which uses the framework, can decide to update a specific product or not. If a product is being changed, the updated product will be sent to the client.

To create new rules and add them to the list of rules in the rule system, developers have to use the *RuleBuilder* (figure 4.5). The *RuleBuilder* is based on the Builder pattern [9, pp. 97–100]. As the Builder pattern intends, the *RuleBuilder* yields the complex

[2] https://www.typescriptlang.org

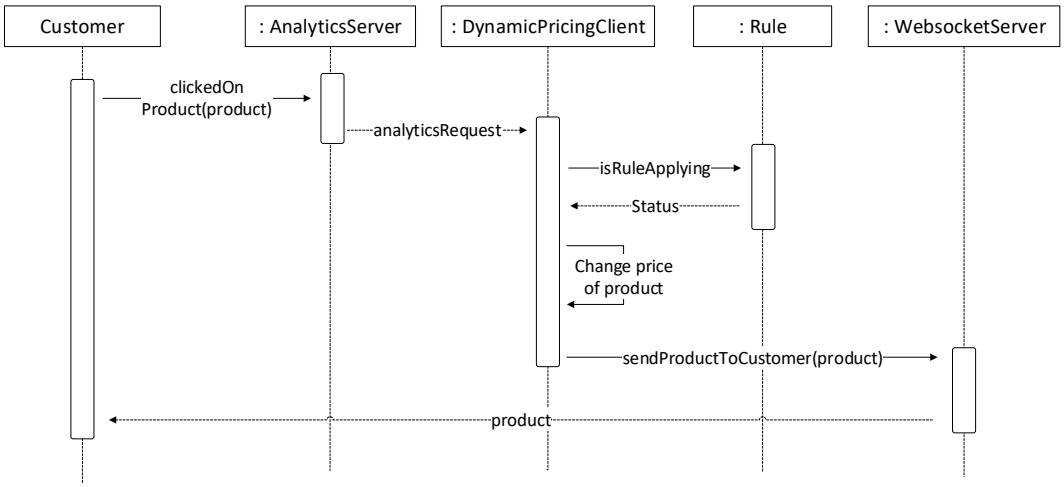

Figure 4.4: Sequence diagram of the rule system.

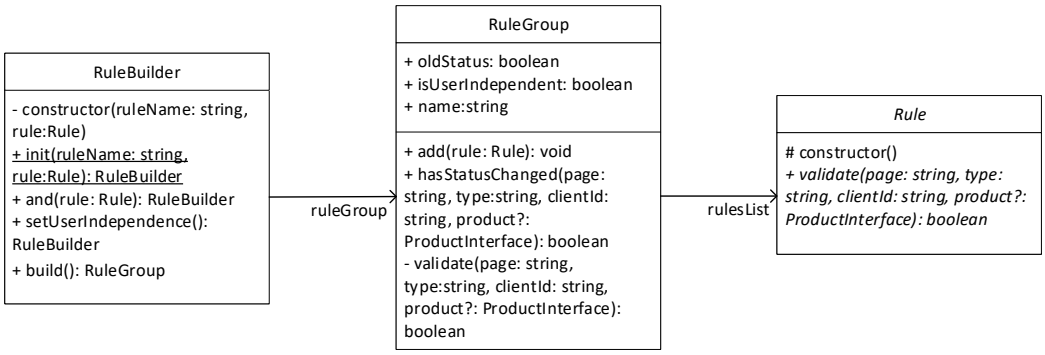

Figure 4.5: UML diagram of the rule-creation-system.

construction of a rule from its representation so that the same process can result in different representations. Also, it makes it easier to create rules and to change its entire construction, which creates a large number of possibilities. One other feature that the *RuleBuilder* offers is to concatenate rules. Therefore, it is necessary to have a holder class, which can add rules and decide whether the status of the concatenated rules has changed. Thus, a *RuleGroup* object is needed. The *RuleBuilder* initializes it upon its creation and uses it to store rules. The *RuleGroup* concatenates rules with the *and* operator, which means that all rules in a group have to apply, so that the whole group applies. For example, a taxi business wants to increase the price when the weather is sunny and an event is happening in the area. Therefore, a developer creates two rules, a weather and an event rule, and concatenates them. Whenever both apply, the price goes up and when one of them does not, the price stays the same.

Nearly any rule can be added to a group of rules because the *Rule* class itself is an

abstract base class which only needs two methods, one for the initialization and the other for validation purposes. But because customer behavior independent rules are handled differently by the dynamic pricing client than user behavior dependent rules, these have to be separated. Customer behavior independent rules are evaluated by environment variables and are not triggered by customer interactions but from the system itself. For that purpose, the *IsUserIndependent* property has been introduced, which lets the system know which rules to trigger.

When the rules are added, and the independence of the customer behavior is decided, a rule group can be built. This group of rules can be added to the rule system for further validation upon triggering. The validation process is shown in figure 4.6. When a customer clicks on a product or the rule system periodically triggers customer independent rules, the system selects one group of rules at a time. Then it selects one rule of the group and validates it. A rule can either be true or false, for example, when the demand of a product drops below the configured threshold of a demand rule, the rule is true and vice versa. If a rule applies, the rule system checks for more rules and validates them too. When these rules are also valid, the rule system reviews the old status of the group. When the status has changed, a callback will be triggered, and the developer can change something, like product prices. But if one rule fails, the whole group is considered to be false. For instance, a price change should occur when the weather is sunny, and the demand for cars is high, but when the weather is rainy, the whole strategy does not apply. Again, the rule system checks the old status and calls a callback if an alteration has occurred. After that, the rule system verifies whether more groups are available. If that is the case, the rule system picks another group of rules and starts the whole process again. At some point, there are no groups left, so the rule system stops validating rules until another analytics request arrives or the timer for customer independent rules triggers.

All rules can be used for any product. They only differ in the way products are assigned. Some rules have to be initialized with a product which makes them product dependent. For instance, the demand rule requires a product for its creation because it uses the demand for a product. Rules that do not need an item for the validation process are versatile and offer the possibility to assign products with the callback. That means that the developer does not have to decide which products to update until the status of a rule changes.

4.3.6 WebSocket-Server

The WebSocket server is responsible for pushing products and their updates to the clients. Therefore, the WebSocket server has a Data Access Object of the database to retrieve and store data (figure 4.3). The server has to know which WebSocket is mapped to which customer to push product updates to a specific client. The client id is used to identify WebSocket connections and map to the right customer. First, the behavior of customers is collected and evaluated and then the price changes are pushed to the correct client.

As mentioned in section 4.3.2, the dynamic pricing client needs to be informed when the connection to the client is closed. Therefore, the WebSocket server also uses callbacks.

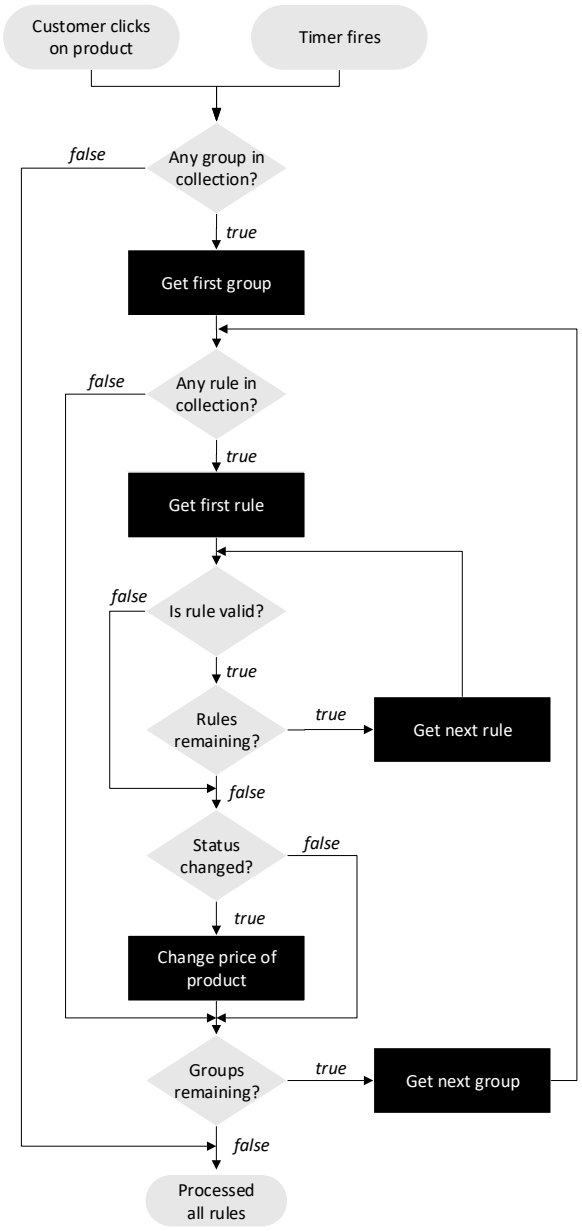

Figure 4.6: Flow digram of how the rules are processed and validated.

4.4 AMP

The AMP of the thesis project is demonstrating how such a framework can be used in an e-commerce application. AMPs are not known for their real-time capabilities and high customizability regarding programming. So, it is necessary to adapt the AMP with a custom solution for real-time functionality.

4.4.1 Extension

As discussed in section 2.2.2, an extension is necessary in order to bring real-time functionality to AMPs. That is important because dynamic pricing is often used in fast-changing markets like stock markets. Stocks change frequently, so the prices have to be updated instantly. When the prices of products are altered slowly, companies or customers can lose money. Neither case is suitable for a company, and so near real-time communication via WebSockets are introduced into the AMP. With a custom extension, a custom element, as well as custom JavaScript, can be implemented.

The lifecycle hooks (figure 2.4) are used to create a container view for the products and to initialize the WebSocket. The WebSocket is responsible for requesting and receiving four types of data:

1. **Client identification**: Upon initialization, the WebSocket opens a connection to the server and sends the client identification. This identification is used in order to map a client to an opened connection. That is necessary because customers can have individual product prices.
2. **Get all products**: When the customer is on the landing page of the AMP, the WebSocket requests all products from the server.
3. **Update product**: If the price of a product changes on the server, the WebSocket receives the updated information about the product.
4. **Get specific product**: Whenever the customer selects a product and lands on the product page, the WebSocket requests detailed information about that product.

These various requests are separated by the request type, which is provided by the data of a request. With these types, the extension knows, which actions it has to take. For example, the view of a product will be updated upon receiving new prices for it.

In order to know which page the customer is currently on and which type of request the extension has to send, HTML element attributes are used. That is common practice for AMP components. It helps to generalize the component and makes it easy to use. Not only can the request type be set in the attributes, but also the WebSocket URL which differs from company to company.

Receiving the data is simple, but displaying it is more complex. It has to be general enough so that the extension can be used in different web pages with different styles without altering the code of the extension. Therefore, HTML templates are used which can be customized. In combination with amp-mustache, which is a logic-less template syntax used to replace tags in the template with variables, properties of products can be displayed and arranged as the developer wishes. That works by parsing the template for tags and replacing them with properties found in products, like price or name [37] (see section 5.2.2).

4.4.2 Layout

The layout of the AMP is kept simple and is based on popular e-commerce pages. Therefore, the products, which the company offers, are displayed on the landing page, as shown in figure 4.7. The landing page shows the image as well as the name and the price of all available products. The customer can then browse through the products.

When the customer selects a product, he will be redirected to the product page

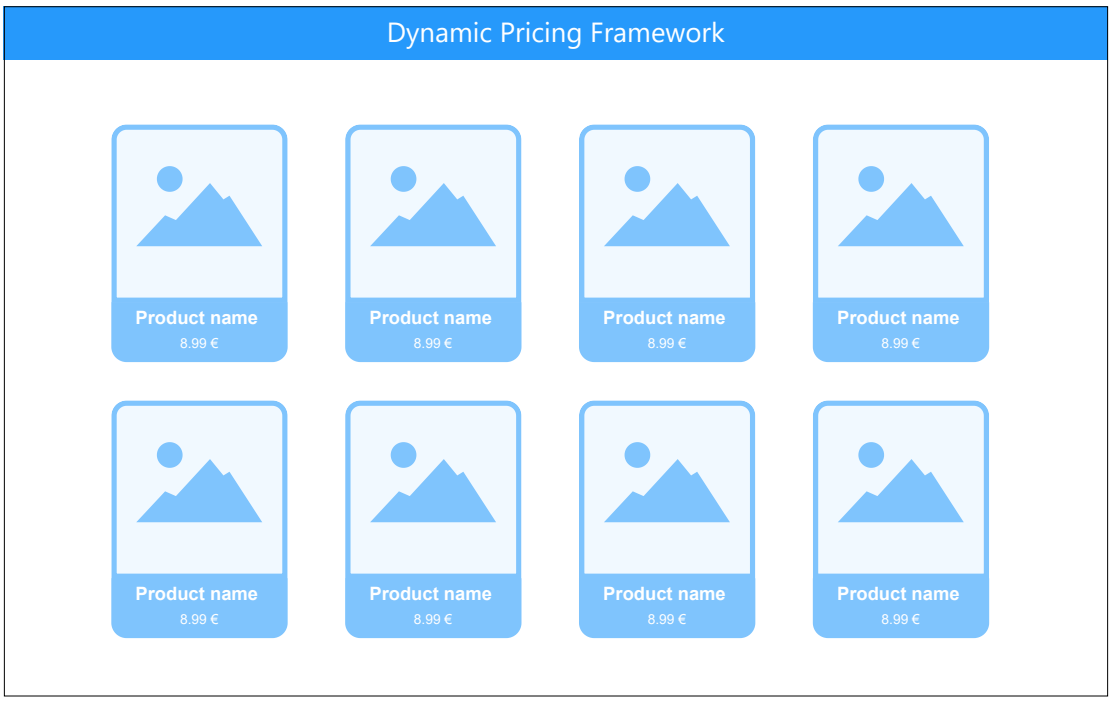

Figure 4.7: Wireframe of the landing page.

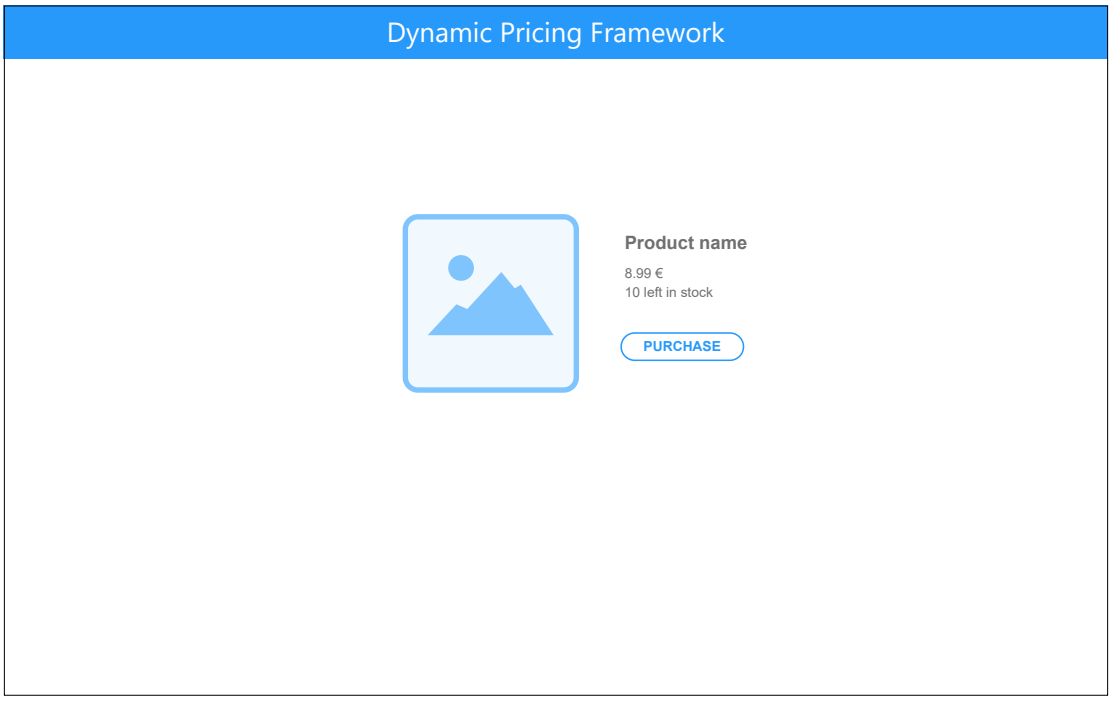

Figure 4.8: Wireframe of the product page.

(figure 4.8). The product page contains detailed information about the product. Of course, the company, who uses the framework, can decide what information to show, how the elements are arranged and how the page is designed. That is also the case for the landing page. Lastly, the customer can either purchase the product or go back to browsing the products.

The layout and design of the AMP are rather simple to demonstrate how the framework operates and to keep the focus on the technical aspects.

4.4.3 Analytics

Analytics in AMP is done via the amp-analytics component. It offers analytics for in-house solutions or most common third-party analytics tools, like Google Analytics or Facebook Pixel. In this project, amp-analytics is configured to interact with the analytics server of the project. Therefore, a configuration file is needed, which is served by the analytics server. The configuration file is used to define the endpoint to the analytics server as well as the analytics data that is tracked. The data is sent when some actions occur, like visiting a page. These triggers can also be set in the configuration file.

In order to gather data suitable for the framework, two triggers have to be registered. The first trigger fires when a customer visits a page like the landing or product page. The other trigger fires when a customer purchases a product. If some of these events occur, analytics data will be sent from the component to the analytics server. The data gives the framework the possibility to, e.g. determine which customer is currently looking at which product or calculate the conversion rate of the e-commerce application.

Because the configuration file is sent by the analytics server, the configuration always stays the same. That creates a problem when tracking the page customers are currently on. Therefore every page has to have its own setting. An inline configuration overrides the variables with the same name from the remote configuration and makes it possible to track and use page dependent data.

Chapter 5

Implementation

In this chapter, the focus is on giving a deeper understanding of how the concepts of the project were implemented and how specific goals have been achieved. The following sections describe the essential implementation parts of the framework, how the framework is used and applied and the implementation of the AMP.

The implementation of the project is mainly written in TypeScript. TypeScript is a typed superset of JavaScript, which means that JavaScript is valid TypeScript and can be used in TypeScript files. It adds types to the JavaScript language and support for static checking or code refactoring. It compiles/transpiles to clean JavaScript code which can be used on the client (AMPs) and the server. Therefore, there is no need to change the programming language when developing the client or the server. Also, TypeScript web services have a large community with lots of open source libraries.

5.1 Framework

The framework is the core of the project and offers the developer using it the possibility to define new rules, configure the servers and provide a way to retrieve product data from a database.

In the following sections, the implementation of the dynamic pricing client, its servers, database interfaces and rules are described.

5.1.1 Dynamic Pricing Client

The *DynamicPricingClient* class is the heart of the framework. It combines two major parts, the dynamic pricing part with the rule system, which manages rules and validates them, and the server part with the models and the DAO, which is the interface of the database. The class works with the rule system by holding the actual rules in an array:

```
export class DynamicPricingClient {
    private rules: Array<RulesCallback> = Array();

    addRule(rules: RuleGroup, callback: (clientId: string, isValid: boolean, product
    ?: ProductInterface) => void) {
        this.rules.push(new RulesCallback(rules, callback));
    } ...
}
```

It offers the possibility to add new ones by calling the *addRule* method The dynamic pricing client introduces a class called *RulesCallback*, which is a helper class to organize rules and their corresponding callback. With separating rules and callbacks from one another, it is possible to add the same rules, but with different callbacks. That can increase readability and adds more functionality to rules.

The *addRule* method takes a *RuleGroup* object and a callback as parameter, which are used to create a *RulesCallback* object. This object will be used later to validate the rules and call the appropriate method when the status of a rule has changed. The callback has three parameters defined of which one is optional. Upon status change of a rule, the clientId that has triggered the status change, as well as the status of the rule itself and the product, if available, will be sent via the callback to the system. The developer can then decide which actions to take.

Before any validation can occur, the analytics data has to be provided. The analytics server calls a callback upon receiving data. The *DynamicPricingClient* class accepts this data in the *onAnalyticsRequest* method and processes all rules before deciding which callbacks to call, as the following function shows:

```
private onAnalyticsRequest(page: string, type: string, clientId: string, productId:
    string) {
    if (productId) {
        this.dbDao.getProduct(productId).then(product => {
            for (let rulesCallback of this.rules) {
                if (rulesCallback.rules.hasStatusChanged(page, type, clientId,
    product)) {
                    rulesCallback.callback(clientId, rulesCallback.rules.oldStatus,
    product);
                }
            }
        })
    } else {
        for (let rulesCallback of this.rules) {
            if (rulesCallback.rules.hasStatusChanged(page, type, clientId)) {
                rulesCallback.callback(clientId, rulesCallback.rules.oldStatus);
            }
        }
    }
}
```

How the decision making is done, is shown figure 4.6.

In the case of user-independent rules, the dynamic pricing client uses a simple timeout loop with recursion in the *checkUserIndependentRules* method to validate them in a five-second interval:

```
private checkUserIndependentRules() {
    setTimeout(() => {
        this.rules.forEach(ruleCallback => {
            if (ruleCallback.rules.isUserIndependent && ruleCallback.rules.
    hasStatusChanged("", "", "")) {
                ruleCallback.callback("", ruleCallback.rules.oldStatus);
            }
        });
        this.checkUserIndependentRules();
    }, 5000)
}
```

Because user independent rules are dependent on environment variables and not customer behavior, it is not necessary to provide them with information about customers. Also, the dynamic pricing client offers features, like sending updated product information to customers.

5.1.2 Dynamic Pricing Strategies

Dynamic pricing strategies are rules, which have only one functionality, validating itself. Every rule has to inherit the following abstract base class *Rule*:

```
export default abstract class Rule {

    protected constructor() { }

    abstract validate(page: string, type: string, clientId: string, product?:
    ProductInterface): boolean;
}
```

It defines the validation method, which is used to acquire the status of a rule. The base class offers the possibility to develop custom rules and use them without altering the framework in any way.

As mentioned in section 4.3.3, there are four rules prepared in the system. One of them is the following *CategoryRule*:

```
export default class CategoryRule extends Rule {
    constructor(private categoryType: number) {
        super();
    }

    validate(page: string, type: string, clientId: string, product?:
    ProductInterface): boolean {
        return product != undefined && product.category.type == this.categoryType;
    }
}
```

The *CategoryRule* class checks if a client is currently looking at a specific product category. If that is the case, the method returns true and vice versa.

5.1.3 Analytics-Server

The analytics server uses the REST protocol. For that to work, the express library[1] is used. It is one of the most used web application frameworks for Node.js[2] and provides the functionality to open HTTP servers quite easily. Other than that, it supports parsers for request body data. They can be injected into the express server itself which makes them parse incoming request bodies in a middleware. These parsers are used for the analytics server, because it uses JSON as a data format.

When the setup of the analytics server is complete, requests can be handled. The server catches them by opening the following POST route with the help of the express library:

[1] https://expressjs.com
[2] https://nodejs.org

```
this.router.post('/', (req, res) => {
    this.onAnalyticsCallback!(req.query.page, req.query.type, req.query.cid, req.
    query.product_id);
    res.sendStatus(200);
});
```

It takes the request parameters, as shown in section 4.3.4, and forwards them to the dynamic pricing client via the *onAnalyticsCallback* property. After that, the server has to send a confirmation back to the client to make sure that the client knows about the successful request.

Finally, the analytics server implements a route to send analytics configs to the AMP in order to support page-dependent analytics data (see section 4.4.3).

5.1.4 Rule-System

Rules are created with a rule builder and the help of the builder pattern. The *RuleBuilder* class offers the possibility to concatenate rules, set user behavior independence and build very different rules. As the program 5.1 shows, the class makes use of a *RuleGroup* object. The *RuleGroup* object holds rules and calls their validation method. After adding all rules to the *RuleBuilder*, the *build* method can be called, which returns the *RuleGroup* object. This object can later be added to the rules list of the framework.

Program 5.1: RuleBuilder class.

```
1  export default class RuleBuilder {
2      private readonly ruleGroup: RuleGroup;
3
4      private constructor(ruleName: string, rule:Rule) {
5          this.ruleGroup = new RuleGroup(ruleName);
6          this.ruleGroup.add(rule);
7      }
8
9      static init(ruleName: string, rule:Rule): RuleBuilder {
10         return new RuleBuilder(ruleName, rule);
11     }
12
13     and(rule: Rule): RuleBuilder {
14         this.ruleGroup.add(rule);
15         return this;
16     }
17
18     setUserIndependence(): RuleBuilder{
19         this.ruleGroup.isUserIndependent = true;
20         return this;
21     }
22
23     build(): RuleGroup {
24         return this.ruleGroup;
25     }
26 }
```

As mentioned in section 5.1.1, the dynamic pricing client triggers the validation of

all *RuleGroup* objects. Because the dynamic pricing client is only interested in status changes of the rules, the *hasStatusChanged* function will be called from the dynamic pricing client. This method takes the analytics data given by the client and calls the helper validation function *validate*:

```
hasStatusChanged(page: string, type:string, clientId: string, product?:
    ProductInterface): boolean {
    let status = this.validate(page, type, clientId, product);
    if (this.oldStatus != status) {
        this.oldStatus = status;
        return true;
    }
    return false;
}

private validate(page: string, type:string, clientId: string, product?:
    ProductInterface): boolean {
    let result = true;
    for (let rule of this.rulesList) {
        result = result && rule.validate(page, type, clientId, product);
    }
    return result;
}
```

The purpose of this function is to check the status of all the rules in this group. Since more than one rule can be in a group, the result of the validation function of the rules is concatenated via an *and* operator and returned to the caller. The *hasStatusChanged* function checks if the status has changed and then returns the result of the check.

5.1.5 Websocket-Server

Like the WebSocket of the AMP extension, also the WebSocket on the server-side is responsible for receiving and sending various types of data. Therefore an enum is introduced representing the request types *GetAllProducts*, *UserId*, *GetProduct* and *UpdateProducts*. One of those request types is included in every request so that the receiver knows which actions to take. The following *onMessage* method in the *WebsocketServer* class filters all requests by their request type:

```
private onMessage(ws: WebSocket, message: any) {
    message = JSON.parse(message);
    if (message.request === RequestTypes.GetAllProducts) {
        this.dbDao.getAllProducts().then(products => ws.send(JSON.stringify({request
    : RequestTypes.GetAllProducts, data: products})));
        this.resetProductToUser(ws);
    } else if (message.request === RequestTypes.UserId) {
        this.wsToUser.push(new WsUserWrapper(ws, message.data));
        this.dbDao.insertOrUpdateCustomer(message.data);
    } else if (message.request === RequestTypes.GetProduct) {
        this.dbDao.getProduct(message.data).then(product => {
            ws.send(JSON.stringify({request: RequestTypes.GetProduct, data: product
    }));
            this.setProductToUser(ws, product);
        });
    }
}
```

The only request type that the server does not filter is the *UpdateProducts* type, because product updates are only sent from the server to the client. The method also makes use of the Data Access Object *dbDao* of the database to be able to send product information to the user (see section 5.1.6).

When a new user connects to the WebSocket, which is indicated by the request type *UserId*, the server creates a wrapper object in order to assign a WebSocket to the current customer. It is later used to send customer specific product updates. The WebSocket server not only maps the WebSocket to the customer, but it also adds the corresponding product, which is currently being looked at, to the client. That is used to identify customers with the same interest and to push specific product updates to them.

5.1.6 Database - DAO and Models

The Data Access Object is used to retrieve data from the database and to offer some sort of adapter in order to be independent of the used database. This DAO is a simple interface which offers a variety of methods that have to be overwritten in order to run the dynamic pricing framework. There is the *DBInterface* for retrieving the data from the database and the domain objects (models) for storing the data. The *DBInterface* offers the functionality to get all or one product, update them, create or update a customer and also retrieve a customer:

```
export interface DBInterface {
    getAllProducts(): Promise<[ProductInterface]>;

    getProduct(id:string): Promise<ProductInterface>;

    updateProduct(product: ProductInterface): void;

    insertOrUpdateCustomer(id: string): void;

    getCustomers(): Promise<[CustomerInterface]>;
}
```

Methods which return data always return a *Promise* because they are asynchronous and non-blocking.

The promises and also the method parameters always have an interface of the models as type. There are three types of models that are important to the framework, the *ProductInterface*, *CategoryInterface* and the *CustomerInterface*. These interfaces should be extended to support more data that exceeds the expectation of the framework (see section 5.2). The most important interface is the following *ProductInterface*:

```
export interface ProductInterface {
    id: string;
    quantity: number;
    price: number;
    category: CategoryInterface;
    expiration: Date;
}
```

It holds product relevant information. This class implements several properties that are used to display product information on the client and holds information needed by some of the basic rules, like the time rule which needs the availability time/expiration time.

5.2 Usage of the Framework

The *DynamicPricingClient* class is the heart and interface of the framework. It is the only entry point to the framework. The next sections cover this topic as well as how the usage of the framework looks like on the client side, the AMP.

5.2.1 Server-Side Usage

The framework implements a facade pattern [9, p. 185], so that the callable methods are very simplified and significant parts are hidden in the framework. That results in a few lines of code needed to operate the system, as the following program shows:

```
this.pricingClient =
    new DynamicPricingClient("./server.config.json", mongoDBClient);
```

A framework object is created by initiating a new instance of the *DynamicPricingClient* class. Parameters needed are the config file, which defines the port numbers for the analytics server and the WebSocket endpoint, and the database DAO, that serves as an access point for the data.

When the dynamic pricing client is initiated, the servers are up and running. The next step is to enable dynamic pricing on the server by using rules.

Rules

It is possible to create new rules with the abstract base class *Rule*. These rules do not differ from the basic rules that come with the framework. They extend the *Rule* class and can implement every validation algorithm needed.

If the rules have been created, they have to be added to the framework. That is done by adding them to the list of dynamic pricing strategies in the framework, as the program below shows:

```
let categoryRule = RuleBuilder.init("Category beauty discount", new CategoryRule(1))
    .build();
this.pricingClient.addRule(categoryRule, (clientId, isValid) =>
    this.userSearchingForCategory(clientId, isValid));
```

Also, a callback is passed on to the framework. The framework later calls this method if the rule changes its status. Parameters such as the client identification and the state of the rule are forwarded to the callback. The callback itself can then update products and push the updated information to the client via the *updateProducts* method of the framework.

To know which product to update, for example, when a customer is looking at a particular product category, the framework has to get the product information from the database.

Database

The database, as well as the products, customers and categories have to extend from the interfaces provided by the framework. This is necessary so that certain properties can always be used by the dynamic pricing framework even when a different database architecture is implemented. The following class *Product* shows this process:

```
export class Product implements ProductInterface {
    name: string = "";
    id: string = "";
    quantity: number = -1;
    price: number = -1;
    image: String = "";
    category: Category = new Category();
    expiration: Date = new Date();
}
```

This class is extended by the *ProductInterface* and adds additional information which is not in the *ProductInterface* class. Because the WebSocket server uses JSON serialization when sending products to the client, it doesn't matter how many properties are added to the models as long as the properties from the interface are kept. These properties can then be displayed and used on the client, which is shown in section 5.2.2.

Extending the model interfaces makes it possible to use the models in the database DAO because of the dynamic polymorphism. For more clarity, the section 5.1.6 shows that the database uses the model interfaces as a return type or parameters. Therefore, the framework can use the models for its methods. That makes it possible for the framework to obtain necessary data while still preserving the complete information of products, customers or categories.

The most used method in the database DAO is the *getProduct* method:

```
getProduct(id: string): Promise<Product> {
    return new Promise((resolve: Function, reject: Function) => {
        this.conn.then(conn => {
            this.productModel.findById(id, (err, doc) => {
                if (doc != null) {
                    let product = new Product();
                    product.id = (doc as any)._id;
                    product.name = (doc as any).name;
                    product.category = (doc as any).category;
                    product.expiration = (doc as any).expiration;
                    product.price = (doc as any).price;
                    product.quantity = (doc as any).quantity;
                    product.image = (doc as any).image;
                    resolve(product);
                } else {
                    resolve(undefined);
                }
            });
        })
    });
}
```

In general, the function looks up a product with the same identification as the parameter in a mongoDB[3] database. Upon finding it, the method creates a new product object. It then returns the product encapsulated in a promise.

5.2.2 Client

Receiving data via a WebSocket on an AMP is not supported as of today. So, there are several techniques necessary to enable server push capability on the client. One way

[3] https://www.mongodb.com

to achieve this is with extensions. Extensions are used in AMP because third-party JavaScript is not allowed. Therefore, the following sections give a more in-depth insight into how the extension has been implemented and how it is used by the AMP HTML.

AMP Extension

The extension is derived by the *AMP.BaseElement* and implements some lifecycle function of the AMP runtime. As figure 2.4 shows, one of the most important lifecycle methods is the *buildCallback*. The *buildCallback* is used to initialize elements on the DOM. In the extension, it creates the outer container for the products and helps to find an entry point into the DOM. As the following method *buildCallback* shows, the function creates a div element which is appended onto the element structure of the HTML element of the extension:

```
buildCallback() {
    this.container_ = this.element.ownerDocument.createElement('div');
    this.element.appendChild(this.container_);
    this.applyFillContent(this.container_, true);
}
```

To be able to support responsive layout that changes with the width of the browser window, it is necessary to implement the *isLayoutSupported* method. This method technically filters out all layout types and informs the AMP runtime, that only responsive layouts are supported.

If this setup is completed, the extension needs to get data in order to display products. Therefore, the WebSocket has to be initialized. The extension creates the WebSocket upon initialization in the following constructor:

```
constructor(element) {
    super(element);

    this.layoutReady_ = false;
    this.container_ = null;
    this.wsUrl_ = this.element.getAttribute('url');
    this.request_ = this.element.getAttribute('request');
    this.templates_ = Services.templatesFor(this.win);

    this.socket_ = new WebSocket(this.wsUrl_);
    this.socket_.onopen = () => this.onOpen_();
    this.socket_.onmessage = evt => this.onMessage_(evt.data);
    this.socket_.onerror = error => this.onError_(error);
}
```

The constructor not only generates the WebSocket, but it also tries to get attributes from the extension's HTML element in order to generalize the extension. It takes the endpoint URL of the server-side WebSocket as well as the request type, which is either indicating to get the list of all products or details for a specific one. The request type is important because otherwise the extension cannot differ between web pages. The distinction is necessary so that many products can be displayed on the landing page or only one on the product page.

When these attributes have been loaded from the HTML, the WebSocket is opened, and various methods are bound to the WebSocket events. These events trigger actions in the extension, for example, the *onopen* event triggers the user identification request.

This request is needed because the WebSocket server has to identify the connection of specific customers later on. Also, the opening state of the WebSocket indicates that the time to fetch data has come. Unfortunately, the open state of the WebSocket is not enough to retrieve data. Because the data transfer can be fast, the layout has to be ready in order to display the information. This race condition can be a real problem. Thus, a flag has been implemented to either fetch data in the open state of the WebSocket or when the layout is ready.

When the data has been fetched, the *onmessage* event of the WebSocket fires. This method has mainly the same functionality as the server-side WebSocket's function:

```
onMessage_(message) {
    message = JSON.parse(message);
    if (message.request === GETALLPRODUCTS) {
        this.templates_.findAndRenderTemplateArray(this.element, message.data)
            .then(elements => this.render_(elements));
    } else if (message.request === UPDATEPRODUCTS) {
        this.templates_.findAndRenderTemplateArray(this.element, message.data)
            .then(elements => this.render_(elements, true));
    } else if (message.request === GETPRODUCT) {
        this.templates_.findAndRenderTemplateArray(this.element, [message.data])
            .then(elements => this.render_(elements));
    }
}
```

The difference though is that the extension needs to render elements and does not retrieve data from the database. To render the elements, the extension makes use of the *findAndRenderTemplateArray* helper function of the AMP runtime. This method helps to render JSON data into an HTML template by replacing placeholders with the corresponding attributes.

When the elements have been rendered, they need to be added to the DOM. Therefore, they are passed on to the following *render_* function of the extension:

```
render_(elements, update = false) {
    if (!update) {
        elements.forEach(element => {
            this.container_.appendChild(element);
        });
    } else {
        elements.forEach(element => {
            try {
                this.container_
                .querySelector(`#${escapeCssSelectorIdent(element.id)}`)
                .replaceWith(element);
            }
            catch (e) { }
        });
    }
}
```

The rendering function iterates over all elements and adds them to the container, which the extension has created at the *buildCallback* lifecycle hook. If the server has sent updated product information, the extension looks up the corresponding HTML element of the product and updates it.

AMP HTML

The template for the products and the extension are defined in the AMP HTML. The extension is like any other AMP component defined at the header of the HTML file. Like every JavaScript file it is included via the *script* tag but uses the *custom-element* attribute to define its corresponding HTML element:

```
<script async custom-element="amp-realtime" src="https://cdn.ampproject.
    org/v0/amp-realtime-0.1.js"></script>
```

Because the AMP needs analytics and some template rendering on the HTML side, it also includes the amp-analytics as well as the amp-mustache extension.

Once the extension is loaded into the AMP and the custom element attribute defined, the extension can be used in the HTML, like the following:

```
<amp-realtime layout="responsive" width="150" height="80" url="ws://localhost:5800"
    request="all">
    <template type="amp-mustache">
      <a id="{{id}}" class="pointer product-view" href="amp-realtime-product.amp.
    html?id={{id}}">
        <amp-img alt="product image"
                 src="{{image}}"
                 width="150"
                 height="150"
                 layout="fixed">
        </amp-img>
        <p class="color-dark">{{name}}</p>
        <p class="color-dark">{{price}} €</p>
      </a>
    </template>
</amp-realtime>
```

The element consists of the attributes that are identifying the layout, the WebSocket URL, as well as the request type. Also, it contains the amp-mustache template used to render the products. The template has various placeholders that are marked with double curly brackets. These are later replaced by properties of the actual products given from the database of the server. This technique ensures that developers can use their structure and design for displaying products without altering the code of the extension.

As section 4.4.3 describes, amp-analytics is used to gather data and send them over to the analytics server. Therefore, a configuration file is needed. It is shown in program 5.2. The configuration is based on the JSON format and configures requests as well as triggers. The request variables are used to define how the request to the server endpoint will look like upon triggering an event. The configuration has three request types. The first one is the “base” request. It is the first part of every request that will be sent to the analytics server and contains basic data, like client identification or type of page. The base request also has “<baserurl>” in its string. This string is replaced with the correct endpoint of the analytics server before sending the configuration to the client. The next request, the “pageview” request, is used if the event “trackPageview” is fired. This event will be triggered whenever the client opens a page. Last is the “purchase” request, which is sent once a customer clicks on the purchase button.

Program 5.2: amp-analytics configuration file.

```
{
  "requests": {
    "base": "<baseurl>?cid=${clientId(dynamic-pricing-analytics)}&product_id=${
    queryParam(id)}&page=${page}",
    "pageview": {
      "baseUrl": "${base}&type=pageview"
    },
    "purchase": {
      "baseUrl": "${base}&type=purchase"
    }
  },
  "triggers": {
    "trackPageview": {
      "on": "visible",
      "request": "pageview"
    },
    "trackPurchaseClick": {
      "on": "click",
      "selector": "#purchase",
      "request": "purchase"
    }
  }
}
```

5.3 Build System

The project is based on the Node.js JavaScript runtime, which supports a lot of build systems. Widely used build systems are gulp[4] and Webpack[5]. Gulp is a task runner, and it supports a lot of libraries. For smaller projects, npm[6] with package.json is fine, but when it comes to flexibility and scalability gulp is typically a better choice. Because there are specific tasks to run and gulp is easy to set up and customize, gulp is used as the build system of this project.

Gulp uses one or more gulpfiles to execute tasks. A gulpfile is written in vanilla JavaScript or other languages like TypeScript. Within gulpfiles the gulp API can be used to execute various commands, but also any vanilla JavaScript is available. Gulp uses *tasks* to receive actions that the runtime has to make. These tasks are asynchronous JavaScript functions which have to be exported in order to be registered by gulp's task system. As soon as the setup is completed and tasks are registered, gulp can execute them by automatically loading the gulpfiles when running the gulp command [43].

Because the master thesis project consists of two separate systems, the AMP and the server, that interact with each other, many tasks have to be executed in series, as figure 5.1 shows. Every TypeScript file has to be compiled/transpiled to JavaScript since the Node.js runtime environment can only run JavaScript. First, the dynamic pricing framework is compiled and after that the server application, because it is dependent

[4] https://gulpjs.com
[5] https://webpack.js.org
[6] https://www.npmjs.com

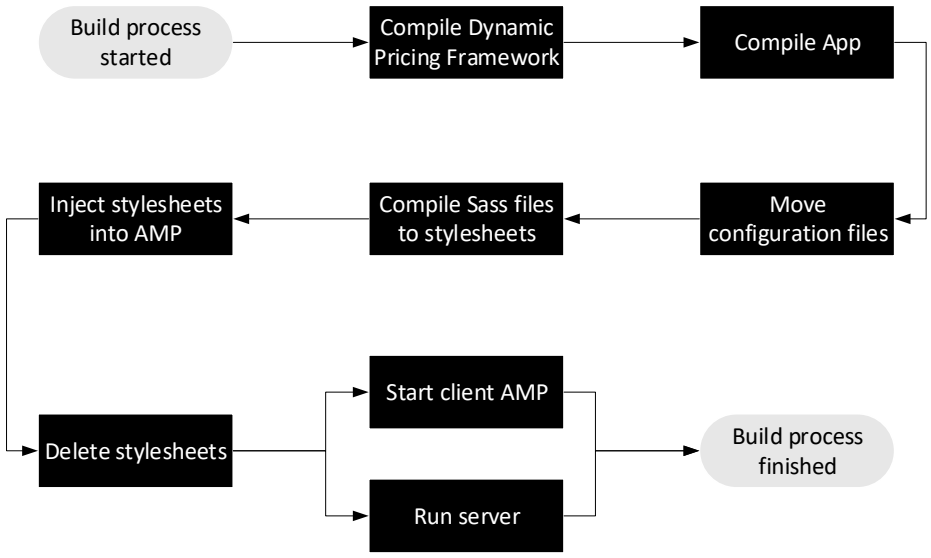

Figure 5.1: Gulp build system flow.

on the framework. Because the whole server files are deployed to a remote folder, the configuration files have to be moved as well. After the server part is done, the client has to be assembled. AMPs only support inline CSS, hence the Sass[7] files have to compiled to one CSS file and included into the AMP. After the inclusion, the CSS file is no longer needed. Therefore, it is being deleted. Once the build steps are completed, the server, as well as the AMP, can be started and used by the Node.js runtime.

5.4 Challenges

During the development of the system, some significant problems have been encountered. Most challenges are due to the generalization of the framework. The database integration has to be given some thought before actually starting to implement it. So first a configuration file has been created, to support different endpoints and to map database entries to properties of the system's models. But because of that, every database has to be implemented in order to support it. Therefore, the configuration file has been removed and replaced by the Data Access Object. The DAO yields more generalizability and leaves the database implementation to the user of the framework, which again offers more customizability.

The next step was to implement the rule system. Since the beginning it has been a requirement that rules have to be extendable and offer a great deal of customizability. That means that using polymorphism and therefore interfaces are the best choice for this kind of problem.

[7] https://sass-lang.com

Also, the client yields some significant challenges, like the customizability of AMPs. Because AMPs don't allow third-party JavaScript, extensions have to be used in order to support WebSockets. Furthermore, the analytics library of AMP has created problems. It is not possible to define the way analytics data is transmitted. Thus, the protocol of amp-analytics has to be used in order to support AMPs. That means that all other client technologies have to follow these rules, which can lead to problems in generalization.

The AMP project is using an older Node.js version, which leads to complications when using libraries that are built for a newer version. Using older versions of libraries increases vulnerability that can result in security risks as well as performance loss.

Chapter 6

Evaluation

This chapter focuses on the evaluation of the project. The assessment consists of the fulfillment of the requirements and the performance analysis, which indicates if the framework responses in a timely manner.

6.1 Requirements

The following sections give a more detailed description of the requirements defined in section 4.1 and if and how these have been met.

6.1.1 Requirement 1: Generic Framework

A second project has been set up to identify if the framework works under different conditions. Thus, the database software has been changed to a SQL database and the client to a straightforward single page with vanilla JavaScript in it. Because a SQL database has a different interface than a MongoDB one, the DAOs had to be changed. The program 6.1 shows the difference between the SQL client and the MongoDB client, which in general is only how the connection to the database is provided and how the data is received and processed. Due to the database interface, access to the database via the class will always be the same. That means that the *DynamicPricingClient*, as well as the framework, do not have to change.

On the other hand, the client had to be developed from scratch because AMP is missing from this evaluation project. Therefore, the analytics protocol has been implemented to support the analytics request and the way the WebSocket handles requests and sends responses has been taken into account.

Overall the system works well and does not encounter any problems. As long as the protocols are adhered to and the requests are sent in the right format, the framework can be used in any existing system which supports WebSockets.

6.1.2 Requirement 2: Common Transfer Protocol

As section 4.3.6 shows, products are sent via a WebSocket to the client. That opens a bidirectional communication channel which has the advantage to transmit information

Program 6.1: DAO of the sql client with the *getProduct* method.

```
1 private db = new sqlite3.Database("./master.db");
2
3 getProduct(id: string): Promise<Product> {
4     return new Promise((resolve: Function, reject: Function) => {
5         this.db.serialize(() => {
6             this.db.get("select product.rowid as id, product.name, price, quantity,
      expiration, description, image, type from product inner join category on
      category.categoryId = product.categoryId where product.rowid=" + id, (error, row
      ) => {
7                 if (error) {
8                     reject(error);
9                 } else {
10                    let tmpProduct = new Product();
11                    tmpProduct.id = row.id;
12                    tmpProduct.name = row.name;
13                    tmpProduct.price = row.price;
14                    tmpProduct.quantity = row.quantity;
15                    tmpProduct.expiration = row.expiration;
16                    tmpProduct.image = row.image;
17                    let tmpCat = new Category();
18                    tmpCat.type = row.type;
19                    tmpProduct.category = tmpCat;
20                    resolve(tmpProduct);
21                }
22            });
23        })
24    });
25 }
```

to the client from the server. Therefore, it is possible to send product updates at any given time.

Because WebSockets are standardized, nearly every client technology supports this API. Be it a Java application, a web page or a mobile app, WebSockets can be used in many different environments.

6.1.3 Requirement 3: Server Push Capability

WebSockets are known for their bidirectional communication technology. With a WebSocket, the server can send product information to the client at any given time independent of the state of the client. Thus, the WebSocket offers a quick (more details on that in section 6.2) and reliable solution.

6.1.4 Requirement 4: Analytics

Because the framework should support a lot of clients, the simple protocol of the amp-analytics library has been implemented on the analytics server, as shown in section 4.4.3. The analytics requests are accepted via REST and then processed by parsing the query parameters. This protocol can easily be implemented in any client technology because

it is a simple HTTP request.

6.1.5 Requirement 5: Analytics Requests Processing

The framework currently processes analytics requests immediately after receiving them. This process includes extracting the query parameters useful to the rule-system and forwarding them, so rules can be found which apply to the current state of the e-commerce system. Also, section 6.2 points out that the time between receiving the analytics request and sending the updated product information is minimal.

6.1.6 Requirement 6: Alteration of Usability, Look and Feel of a Web Page

As of the current state of the framework, the server does not influence the usability, look and feel of a web page at all. Because the products are sent via a WebSocket, literally any client technology, that supports WebSocket, can be used to exchange information with the server. This information can be displayed in any possible way with no exceptions. The framework has a minimal link to the web page itself.

6.1.7 Requirement 7: Dynamic Pricing Strategies

The framework currently supports pre-built rules, concatenation of rules to enable a more variety of combinations of dynamic pricing strategies and an interface to build custom rules. The interface offers flexibility of the rules system, which is the most crucial part because there are a lot of different application areas for dynamic pricing. One such application area is the taxi industry in which Uber is operating. As described in section 3.2, Uber uses dynamic pricing strategies to adjust their prices based on the number of drivers available, weather, current time and so on. A similar dynamic pricing strategy has been implemented in the following *UberRule* class:

```
export default class UberRule extends Rule {
    currentWeather= Weather.None;
    currentDriverSupply = 1000000;
    currentTime:number = 1000000;

    constructor(private triggerWeather:Weather = Weather.Sunny, private timeFrom:
    number, private timeTo:number, private driverSupply:number) {
        super();
    }


    validate(page: string, type: string, clientId: string, product?:
    ProductInterface): boolean {
        return this.currentWeather == this.triggerWeather && this.currentTime > this
    .timeFrom && this.currentTime < this.timeTo && this.currentDriverSupply < this.
    driverSupply;
    }
}
```

It shows the flexibility of the rules. The class has different properties which represent the environment variables such as weather information, driver supply and the current time. It also implements certain trigger properties which have to be in a specific state for the rule to trigger.

If for example the rule is set to trigger when the weather is clear, the current time is between eight PM and three AM and the driver supply is under ten, the rule will be applied when these specific conditions are fulfilled. Then the prices can be updated.

These facts prove the point that the framework and its rules are flexible and can be used in multiple ways.

6.2 Performance Analysis

Excellent performance is an essential aspect when it comes to dynamic pricing. As in [17] or [6], dynamic pricing is used in areas where the demand or user behavior can rapidly change. Therefore, it is inevitable to offer a performant, fast and reliable solution for dynamic pricing.

It is hard to determine what the word "fast" means. People with different backgrounds or companies with diverse application areas have different understandings of how quickly a system should react. So, the hypothesis is that the system should react faster than an average human can respond, which is approximately 250 milliseconds. If the system beats that time, it can be considered as fast.

The performance test has been done on a desktop computer with Windows 10[1] and Chrome[2] as a browser. The PC hardware can be considered as average. The whole test is running locally and thus requires Cross-Origin Resource Sharing (CORS) to be allowed.

Table 6.1 shows the performance of the system. The performance has been measured by acquiring the timestamps at certain stages of a use case, beginning by sending an analytics request to the server, looking if any rule applies and ending by updating the product information on the client. For these measurements, the demand rule has been used. The demand rule updates a specific product price whenever a customer visits its detail page. To measure the actual time whenever an analytics request has been sent, the evaluation project has been used, because time tracking of analytics requests in the analytics library of AMPs is not supported and the date in the header of a data package is too inaccurate.

A row in the table defines a specific measurement, and the columns declare a timestamp. The first column shows at which time the analytics request has been sent from the client. The second column shows when the server has received the request. The next column indicates the time when the demand rule has been applied, and the product information has been sent to the client. Last but not least is the time when the product has been updated to its dedicated price on the browser.

The measurements show that the time required to receive the analytics request and finally updating the product information is way below the reaction time of a human being. But the time needed can vary based on the client's device, internet connection and the software architecture used on the client- and the server-side. With all that said, the system can be still considered as fast.

[1] https://www.microsoft.com/de-de/itpro/windows-10

[2] https://www.google.com/chrome

Table 6.1: Performance table of the system.

	Analytics request sent	Analytics request received	Rule applied & product sent	Product price updated on client
M#1	3:0:24.570 (t)	t+2ms	t+3ms	t+5ms
M#2	3:1:20.919 (t)	t+2ms	t+3ms	t+6ms
M#3	3:1:27.807 (t)	t+3ms	t+4ms	t+5ms
M#4	3:2:13.628 (t)	t+2ms	t+4ms	t+5ms
M#5	3:2:44.715 (t)	t+1ms	t+3ms	t+5ms
M#6	3:3:12.249 (t)	t+2ms	t+4ms	t+5ms
M#7	3:3:44.386 (t)	t+2ms	t+4ms	t+5ms
M#8	3:4:15.562 (t)	t+3ms	t+4ms	t+5ms

6.3 Possible Extensions

Based on the requirements, the framework is completed, but it is not yet production ready. However, the framework shows the potential it has for e-commerce businesses, and that other approaches to this type of problem are possible. The framework could be made even more generic in a way that further functionality can be offered to the companies, for example requesting certain products based on type or relevance, in order to be more useful for real-life scenarios. Also, the decision on how the data is sent or processed could be given to the developer that uses the framework to comply with company-specific rules. That also includes certificates for the WebSocket and analytics server to enable secure transmissions. These are currently not supported because the framework runs in a controlled environment. Apart from this, more rules can be introduced to support more use cases and lower the time businesses need to understand the system and write their own custom rules. Also, the system currently supports e-commerce businesses, but there are far more application areas for dynamic pricing. For this kind of task, the system has to be restructured and extended to support more than one use case.

Chapter 7

Conclusion

Dynamic pricing is considered as a vast increasing topic for online retailers. But it is not only used in e-commerce markets. Long before the internet, companies have been changing prices based on demand, quality and other factors. These techniques haven't been altered a lot, but with today's technology, it is quite more comfortable and more frequently possible to change prices. There are a lot of application areas for dynamic pricing, which results in many different use cases. Dynamic pricing software has to be generalized or individually tailored to the application area to cover these sort of use cases. That yields a big challenge which many software developers try to solve by creating in-house solutions like Amazon, that are fit to specific use cases. But there are other generalized solutions out there which mostly offer an API for receiving product prices based on competitor pricing or other data. Commonly, these are expensive and don't provide a great deal of functionality and customizability because these are separated from the central system.

Dynamic pricing often relies on user analytics data to precisely adjust the prices to user expectations. With no input from the actual web application, the third-party software can't make customer-based prices. These software solutions heavily rely on environment data which the software can gather independently of the web application. In many cases, this solution fits the goal, but in application areas like ferries, user-dependent data is required, for example, the size of a car.

This thesis tries to take a different approach by offering a generalized solution with high customizability. It can be integrated into an existing system, which makes it fast, reliable and adjustable. The server push capability yields an excellent opportunity for either changing products very fast or catching user behavior. As section 6.2 shows, the framework reacts nearly instantly upon changed environment variables or user behavior patterns. With the right usage and customized dynamic pricing strategies, the framework can increase the profit of an e-commerce application.

The framework also fulfills the requirements, which have been defined at the planning stage of the system, as the section 6.1 shows. Therefore, the framework can be used in most of the backend and frontend solutions and can provide customization for specific application areas and use cases. These allegations are also supported by the use of various technologies like Data Access Objects for database connections, a configuration file, the Builder pattern for combining different rules, extendable rules for offering the possibility to define company dependent dynamic pricing strategies and standardized

protocols for receiving and transmitting data from and to the client. These ultimately result in a highly customizable and generalized framework which can be embedded in any number of existing systems.

Although new technologies and tools for dynamic pricing and analytics are implemented continuously and made available, the approach to these areas practically has not been changed for many years. Pricing based on competitor prices and demand has been in use for as long as trading exists. With these many choices of software and little diversity between them, companies face the problem of picking the right one suitable for their use case. Dynamic pricing is expanding into more and more use cases, but the software is not. Therefore, companies usually stick to in-house solutions.

Altogether dynamic pricing and analytics is a very active field with new inventions, especially in the machine learning area. Dynamic pricing is introduced to many new use cases every day, which indicates the importance of the topic for retailers and e-commerce businesses. But only the future will tell how vital user analytics combined with dynamic pricing will become.

Appendix A

Content of the CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Files

Path: /

thesis.pdf	Master thesis document

A.2 Project Data

Path: /implementation/project

src/amp-realtime/ . . .	AMP project
src/amp-realtime/extensions/amp-realtime/	Real-Time extension for the AMP
src/amp-realtime/examples/amp-realtime*.html	HTML files of the AMP
src/app/	Folder of the dynamic pricing framework and the server
src/app/*.json	Config files for npm, Node.js and the server
src/app/gulpfile.js . . .	Build script for gulp
src/app/**.ts	TypeScript source files
mongodb/**.bson . . .	mongoDB dump JSON documents in binary-encoded format
mongodb/**.json . . .	mongoDB dump JSON documents

A.3 Evaluation-Project Data

Path: /implementation/evaluation_project

*.json	Config files for npm, Node.js and the server
master.db	SQL database file
gulpfile.js	Build script for gulp
src/app/	Client folder
src/app/*.ts	TypeScript source files

src/app/*.scss	Sass files
src/app/*.html	HTML files
src/app/lib/*.js	JavaScript libraries
src/server/	Server folder
src/server/**.ts	TypeScript source files

A.4 Online Literature

These files are copies of the webpages and youtube videos used as references. The file names are numbers which correspond to the reference number in the literature. For example [34] corresponds to 34.pdf in the folder.

Path: /online_literature

[reference_number].mp4	Youtube videos
[reference_number].pdf	Webpages

A.5 Miscellaneous

Path: /images

*.jpg, *.png	Raster-Graphics
*.pdf	Vectorized images

References

Literature

[1] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies, Second Edition*. Prentice Hall PTR, May 2003 (cit. on p. 24).

[2] Christopher Bayliss et al. "Dynamic pricing for vehicle ferries: Using packing and simulation to optimize revenues". *European Journal of Operational Research* 273.1 (Feb. 2019), pp. 288–304 (cit. on p. 19).

[3] C.J. Carmona et al. "Web usage mining to improve the design of an e-commerce website: OrOliveSur.com". *Expert Systems with Applications* 39.12 (Sept. 2012), pp. 11243–11249 (cit. on p. 18).

[4] Andre Charland and Brian Leroux. "Mobile application development: web vs. native". *Communications of the ACM* 54.5 (May 2011), pp. 49–53 (cit. on p. 4).

[5] Brian Clifton. *Advanced web metrics with Google Analytics, 3rd Edition*. John Wiley & Sons, Mar. 2012 (cit. on p. 16).

[6] Sriram Dasu and Chunyang Tong. "Dynamic pricing when consumers are strategic: Analysis of posted and contingent pricing schemes". *European Journal of Operational Research* 204.3 (Aug. 2010), pp. 662–671 (cit. on pp. 15, 18, 50).

[7] Evanthia Fasoula and Karsten Schweikert. *Price regulations and price adjustment dynamics: Evidence from the Austrian retail fuel market*. Hohenheim Discussion Papers in Business, Economics and Social Sciences 08-2018. 2018 (cit. on p. 18).

[8] Brian Fling. *Mobile design and development: Practical concepts and techniques for creating mobile sites and Web apps*. O'Reilly Media, Inc., Aug. 2009 (cit. on pp. 6, 7).

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Oct. 1994 (cit. on pp. 26, 39).

[10] Jochen Gönsch, Robert Klein, and Claudius Steinhardt. "Dynamic Pricing – State-of-The-Art". *Journal of Business Economics* 3 (Feb. 2009), pp. 1–40 (cit. on pp. 1, 14, 15, 18, 19).

[11] Wesley Hales. *HTML5 and JavaScript Web Apps: Bridging the Gap Between the Web and the Mobile Web*. O'Reilly Media, Inc., Nov. 2012 (cit. on p. 4).

[12] S. Hernández et al. "Analysis of Users' Behavior in Structured e-Commerce Websites". *IEEE Access* 5 (May 2017), pp. 11941–11958 (cit. on pp. 12, 14, 16, 18).

[13] Kim Hoyoung et al. “An Empirical Study of the Use Contexts and Usability Problems in Mobile Internet”. In: *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. Jan. 2002, pp. 1767–1776 (cit. on p. 7).

[14] Yu-Shiang Hung et al. “Web usage mining for analysing elder self-care behavior patterns”. *Expert Systems with Applications* 40.2 (Feb. 2013), pp. 775–783 (cit. on p. 18).

[15] Bernard J. Jansen. *Understanding User-Web Interactions via Web Analytics*. Morgan & Claypool, Aug. 2009 (cit. on p. 12).

[16] P. K. Kannan and Praveen K. Kopalle. “Dynamic Pricing on the Internet: Importance and Implications for Consumer Behavior”. *International Journal of Electronic Commerce* 5.3 (Mar. 2001), pp. 63–83 (cit. on pp. 1, 15).

[17] R. Preston McAfee and Vera L. te Velde. “Dynamic Pricing in the Airline Industry”. *Economics and Information Systems* (Jan. 2006), pp. 527–569 (cit. on p. 50).

[18] L M Minga, Yu-Qiang Feng, and Yi-Jun Li. “Dynamic pricing: ecommerce - oriented price setting algorithm”. In: *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*. Vol. 2. Nov. 2003, pp. 893–898 (cit. on pp. 14, 15).

[19] Wendy W. Moe. “Buying, Searching, or Browsing: Differentiating Between Online Shoppers Using In-Store Navigational Clickstream”. *Journal of Consumer Psychology* 13.1-2 (Jan. 2003), pp. 29–39 (cit. on pp. 12, 13).

[20] Wendy W. Moe and Peter S. Fader. “Dynamic Conversion Behavior at E-Commerce Sites”. *Management Science* 50.3 (Mar. 2004), pp. 326–335 (cit. on p. 12).

[21] Rainer Olbrich and Christian Holsing. “Modeling Consumer Purchasing Behavior in Social Shopping Communities with Clickstream Data”. *International Journal of Electronic Commerce* 16.2 (Dec. 2011), pp. 15–40 (cit. on p. 12).

[22] Paul A. Pavlou and Mendel Fygenson. “Understanding and Predicting Electronic Commerce Adoption: An Extension of the Theory of Planned Behavior”. *MIS Quarterly* 30.1 (Mar. 2006), pp. 115–143 (cit. on p. 12).

[23] Orit Raphaeli, Anat Goldstein, and Lior Fink. “Analyzing online consumer behavior in mobile and PC devices: A novel web usage mining approach”. *Electronic Commerce Research and Applications* 26 (Nov. 2017), pp. 1–12 (cit. on pp. 12, 14).

[24] J. Ben Schafer, Joseph A. Konstan, and John Riedl. “E-Commerce Recommendation Applications”. *Data Mining and Knowledge Discovery* 5.1-2 (Jan. 2001), pp. 115–153 (cit. on pp. 12, 18).

[25] N. Serrano, J. Hernantes, and G. Gallardo. “Mobile Web Apps”. *IEEE Software* 30.5 (Sept. 2013), pp. 22–27 (cit. on p. 4).

[26] Qiang Su and Lu Chen. “A method for discovering clusters of e-commerce interest patterns using click-stream data”. *Electronic Commerce Research and Applications* 14.1 (Jan. 2015), pp. 1–13 (cit. on pp. 12, 18).

[27] Qiong Tian et al. “Dynamic pricing for reservation-based parking system: A revenue management method”. *Transport Policy* 71 (Nov. 2018), pp. 36–44 (cit. on p. 19).

[28] Katerina Tzafilkou and Nicolaos Protogeros. “Diagnosing user perception and acceptance using eye tracking in web-based end-user development”. *Computers in Human Behavior* 72 (July 2017), pp. 23–37 (cit. on p. 18).

[29] Y. Wang. “Dynamic pricing considering strategic customers”. In: *2016 International Conference on Logistics, Informatics and Service Sciences (LISS)*. July 2016, pp. 1–5 (cit. on p. 15).

[30] J. Yang et al. “Characterizing User Behavior in Mobile Internet”. *IEEE Transactions on Emerging Topics in Computing* 3.1 (Mar. 2015), pp. 95–106 (cit. on p. 10).

Audio-visual media

[31] *Building Beautiful, Interactive AMP Pages for E-Commerce & Beyond (Google I/O '17)*. May 2017. URL: https://www.youtube.com/watch?v=bGry_GtKtQw (visited on Mar. 5, 2019) (cit. on p. 8).

[32] *Intro to AMP (Accelerated Mobile Pages)*. Feb. 2016. URL: https://www.youtube.com/watch?v=lBTCB7yLs8Y (visited on Mar. 5, 2019) (cit. on pp. 7, 8).

Software

[33] *AMPHTML base-element.js*. 2018. URL: https://raw.githubusercontent.com/ampproject/amphtml/master/src/base-element.js (visited on May 11, 2019) (cit. on p. 9).

Online sources

[34] *AMP helps The Washington Post increase returning users from mobile search by 23 %*. 2015. URL: https://developers.google.com/web/showcase/2016/pdfs/wapo.pdf (visited on Nov. 10, 2018) (cit. on pp. 11, 55).

[35] *AMP HTML Specification*. URL: https://amp.dev/documentation/guides-and-tutorials/learn/spec/amphtml?referrer=ampproject.org (visited on May 11, 2019) (cit. on p. 8).

[36] *AMP - Supported Browser*. URL: https://amp.dev/support/faq/supported-browsers (visited on Mar. 6, 2019) (cit. on p. 11).

[37] *amp-mustache*. URL: https://amp.dev/documentation/components/amp-mustache (visited on May 11, 2019) (cit. on p. 30).

[38] *Analytics Usage Distribution in the Top 1 Million Sites*. URL: https://trends.builtwith.com/analytics (visited on Feb. 17, 2019) (cit. on p. 17).

[39] Digital Analytics Association. *Web Analytics Definitions*. 2008. URL: https://www.digitalanalyticsassociation.org/Files/PDF_standards/WebAnalyticsDefinitions.pdf (visited on Mar. 6, 2019) (cit. on p. 11).

[40] Eric Enge. *Mobile vs Desktop Usage in 2018: Mobile takes the lead*. 2018. URL: https://www.stonetemple.com/see-our-2018-study-of-mobile-vs-desktop-usage/ (visited on Feb. 19, 2019) (cit. on pp. 5, 6).

[41] *Facebook Analytics - Features*. URL: https://analytics.facebook.com/features (visited on Jan. 5, 2019) (cit. on p. 16).

[42] *IBM Dynamic Pricing*. URL: https://www.ibm.com/us-en/marketplace/dynamic-pricing/details (visited on Jan. 11, 2019) (cit. on p. 19).

[43] *JavaScript and Gulpfiles*. URL: https://gulpjs.com/docs/en/getting-started/javascript-and-gulpfiles (visited on May 16, 2019) (cit. on p. 44).

[44] *Learn about Google Analytics*. URL: https://developers.google.com/analytics/devguides/platform/ (visited on Mar. 14, 2019) (cit. on pp. 16, 17).

[45] *Omnia - Dynamic Pricing*. URL: https://www.omniaretail.com/dynamic-pricing (visited on Mar. 18, 2019) (cit. on p. 20).

[46] *priceedge - E-Commerce Pricing Made Easy*. URL: https://priceedge.eu/ecommerce (visited on Mar. 18, 2019) (cit. on p. 20).

[47] *Spree Commerce - Features*. URL: https://spreecommerce.org/features/ (visited on Mar. 18, 2018) (cit. on p. 20).

[48] Statista. *Year-on-year growth in time spent per mobile app category in 2017*. 2018. URL: https://www.statista.com/statistics/251096/fastest-growing-shopping-app-categories/ (visited on Mar. 7, 2019) (cit. on p. 13).

[49] Synced. *AI-Powered Dynamic Pricing Is Everywhere*. Nov. 2018. URL: https://medium.com/syncedreview/ai-powered-dynamic-pricing-is-everywhere-4271a9939d11 (visited on Jan. 8, 2019) (cit. on p. 19).

[50] *U.S. mobile retail commerce sales as percentage of retail e-commerce sales from 2017 to 2021*. 2017. URL: https://www.statista.com/statistics/249863/us-mobile-retail-commerce-sales-as-percentage-of-e-commerce-sales/ (visited on Mar. 7, 2019) (cit. on p. 14).

[51] *Who is AMP for?* URL: https://www.ampproject.org/learn/who-uses-amp/ (visited on Nov. 10, 2018) (cit. on pp. 10, 11).