Analysis, design and development of a web-shop template using SPHERE.IO e-commerce platform

Laura Luiz Escoriza

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director:
Hajo Eichler

Company:
commercetools GmbH

Advisor:
Carles Farré Tost

Department:
Enginyeria de Serveis i Sistemes d'Informació (ESSI)

Master thesis

Degree in Informatics Engineering (2003)

January 2014

DADES DEL PROJECTE

Títol del projecte:	Analysis, design and development of a web-shop template using SPHERE.IO e-commerce platform.
Nom de l'estudiant:	Laura Luiz Escoriza
Titulació:	Enginyeria en Informàtica (2003)
Crèdits:	37,5
Director:	Hajo Eichler
Empresa del director:	commercetools GmbH
Ponent:	Carles Farré Tost
Departament del ponent:	ESSI

MEMBRES DEL TRIBUNAL *(nom i signatura)*

1. President:	Antoni Urpí Tubella
2. Vocal:	Klaus Gerhard Langohr
3. Secretari:	Carles Farré Tost

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

ABSTRACT

In the present thesis a possible next generation of e-commerce solutions with a platform-as-a-service model is presented and analyzed. This generation tries to fill the gap of missing developer-friendly alternatives to build systems with e-commerce components. Current offered solutions are mostly aimed for the comfortable use of designers and other non-technical roles, usually in the shape of out-of-the-box products. These solutions are usually limiting the ability of developers to integrate technologies or build innovative business models, thus sometimes forcing companies to invest in projects that have to be built practically from the start.

This document describes the development of the first web-shop built with one of these solutions, SPHERE.IO, an e-commerce platform-as-a-service developed in Berlin by commercetools GmbH. During this process, questions are being answered about the suitability of e-commerce platforms to develop web-shops, a product most developers have to face when providing e-commerce solutions to companies. The web-shop has a dual purpose, as it will also serve as the first open-source template provided by the platform to help other developers build their own solutions.

ACKNOWLEDGMENTS

I would especially like to thank Hajo for accepting being the director of my thesis and reading all these endless pages despite of having so much to do (really, how can you find the time!). Thanks for your support and always good advice to improve this project.

I would also like to thank all the rest of the SPHERE.IO team for creating and raising such a great product, giving me the opportunity to work with them.

From the ones that were:
Aurélie, Jens, Roman, Ian, Lenni, Christian and Martin.
(I loved the time we spent together, guys)

To the ones that are:
Nicole, Peter, Gregor, Dirk, Oleg, Nicola, Martin, Michael and Sven.

I would like to thank my teacher Carles, who guided me through this project from the distance. Thank you especially for DSBW, which documentation I check constantly, not only for this project.

Thanks to my roommate Sebastian, the most experienced online shopper I have ever met, whose advice was very convenient I must say. Thank you for helping me and have such loving cats that kept me well entertained during the long days locked in my room.

Many thanks as well to those friends that helped me somehow to be where I am now.
Particularly Hèctor and Pau, who help me the most. Thank you.

And of course thousand thanks to the most important person in my life, my husband Héctor, whose constant care, support and help was so necessary during all my career life.
I would not be here without you, you know that.

Finally, I thank my parents, from whom I inherited the love for arts and business.
Because they raised me to be a person capable of anything.
And I chose to be a computer engineer.

INDEX

GLOSSARY

AJAX	Stands for Asynchronous JavaScript and XML, and is a technique used on the client side of a system to perform asynchronous operations (although synchronous are also allowed), such as communicating with a server, without keeping the client idle. The use of XML is not required despite the name.
API	Stands for Application Programming Interface, and in general means a specification that defines how to interact with another software component.
CDN	Stands for Content Management Network, and it is a large distributed system of servers deployed in multiple data centers across the Internet, allowing to server content with high availability and performance.
CSS	Stands for Cascading Style Sheets, and is a style sheet language used for describing the look and formatting of a document written in a markup language.
DOM	Stands for Document Object Model, and is an object model and programming interface for documents, such as HTML, in which case defines all elements as objects with associated properties, methods and events.
HTML	Stands for HyperText Markup Language, and is a markup language designed for creating web pages.
HTTP	Stands for Hypertext Transfer Protocol, and is an application protocol for distributed, collaborative, hypermedia information systems.
HTTPS	Stands for Hypertext Transfer Protocol Secure, and is the result of layering the HTTP protocol on top of the SSL/TLS protocol, in order to add security capabilities of SSL/TLS to standard HTTP communications.
IDE	Stands for Integrated Development Environment, and is a software application that provides support for software development; usually in the areas of source code edition, build automation and debugging.
Internet	Global system of interconnected computer networks that use the standard TCP/IP to communicate.

JSON	Stands for JavaScript Object Notation, and is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs.
JVM	Stands for Java Virtual Machine, and it a virtual machine that can execute Java bytecode.
OAuth	A protocol for authentication and authorization.
PaaS	Stands for Platform-as-a-Service, and is a category of cloud computing services that provides a computing platform as a service, so that the consumer creates software using tools and/or libraries from the provider.
PCI	*See PCI DSS.*
PCI DSS	Stands for Payment Card Industry Data Security Standard, and includes a set of twelve specific requirements to cover six different goals, in order to create a secure environment for payment data.
PHP	Stands for PHP: Hypertext Preprocessor, and is a server-side scripting language originally designed for web development.
REST	Stands for Representational State Transfer, and is an architectural style for designing networked systems, based on a set of principles, such as using stateless requests or a uniform interface for resources.
RESTful	A RESTful system follows REST architectural principles and uses HTTP as a transmission protocol.
TCP/IP	Stands for Transmission Control Protocol/Internet Protocol, and is considered the Internet protocol suite, as it is the combined use of the TCP transport protocol and the IP communication protocol.
SaaS	Stands for Software-as-a-Service, and is a category of cloud computing services that provides a complete software, along with its data, as a service.
Scala	A object-functional programming language that runs on JVM and is compatible with Java scripts.
SDK	Stands for Software Developer Kit, and is a set of software development tools to create applications for a certain software package.

SEO	Stands for Search Engine Optimization, and means to improve the ranking position of a website in Internet search engines.
SMTP	Stands for Simple Mail Transfer Protocol, and is a standard for email transmission through the Internet network.
SSL/TLS	Stands for Transport Layer Security/Secure Sockets Layer, and are two related cryptographic protocols designed to provide communication security over Internet.
URL	Stands for Uniform Resource Locator, and is a specific character string that constitutes a reference to a resource, also known as web address when used with HTTP.
WWW	Stands for World Wide Web, also known as W3 or simply the Web, and is a system of interlinked hypertext documents accessed via the Internet with a web browser.
XML	Stands for Extensible Markup Language, and is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

1 INTRODUCTION

Traditional companies and institutions are making use of e-commerce to overcome the boundaries of space and time: it allows them to globalize their operations and offer a more personalized service to the customer. Moreover, many entrepreneurs took advantage of the benefits of e-commerce and created new business models[1].

E-commerce has greatly evolved for forty years of existence and is still evolving continuously, as well as the software offered to support it. When searching for e-commerce solutions, almost all offered systems are focused on building web-shops, despite of the fact that electronic commerce is not just about web shopping any longer.

1.1 MOTIVATION

As it will be demonstrated in the section Current Alternatives 1.3.4, it is safe to say that customizable software solutions are too rigid to give an efficient and profitable answer to a future e-commerce context. So instead of offering a final product pre-built on top of an e-commerce platform and customize it to fit the merchant's needs, why not provide a robust and flexible e-commerce platform to let merchants build what they need from the beginning?

It was that particular question that led the team of SPHERE.IO to design and develop this new commerce platform-as-a-service. Their main goal was creating an e-commerce platform with a set of tools to allow developers to easily access and process all commerce data from within any kind of application.

The SPHERE.IO backend system, where all commerce data is stored, is directly accessible via a RESTful web API, allowing any external system to use the data, regardless of the programming language or chosen framework. In order to provide a more comfortable development experience, they built the first of many open-source client libraries to come: a Java client library. On top of it they created the SPHERE.IO Play SDK, a toolkit especially aimed for building web applications with ease using the Play web framework (Figure 1.1).

[1] Deal-of-the-day (short time limited offers) or Discovery Commerce (periodical delivery of selected goods) are only some of the new business models that became popular thanks to e-commerce [Pie12].

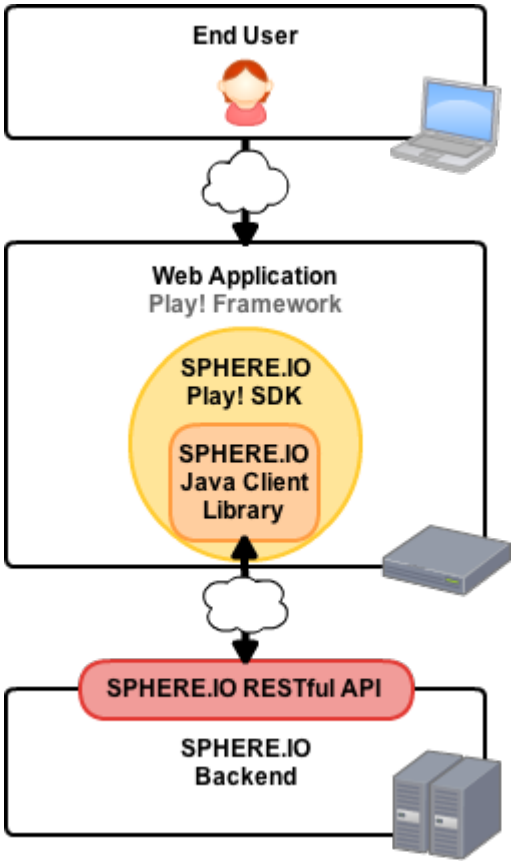


Figure 1.1: Diagram of a typical Java web application using SPHERE.IO.

Since no system has ever been developed with this platform before, except for some small testing sites, it is required to implement a realistic e-commerce application on top of it in order to test its benefits and correct its flaws. It is also necessary to have a bootstrap project from which other future internal or external projects may reuse its code.

1.2 Objectives

The project has two main objectives that are strongly connected. One objective is to analyze, design and develop the first web-shop using the SPHERE.IO platform. This implementation will provide a first open source template, which then can be reused by developers to build their own web-shops. Additionally the source code will serve as a live documentation about how to use the platform.

The second objective is to evaluate the capability of SPHERE.IO to compete with current e-commerce solutions. Better alternatives will be proposed to the platform development team when needed, aiming to improve its initial design. This evaluation can be achieved by the development of the previously mentioned web-shop, which will assist as a practical example to test the suitability of the platform to build these kind of popular applications.

1.3 Background

Before attempting to evaluate any e-commerce solution it is necessary to exactly define what is understood as electronic commerce (section What is e-commerce 1.3.1). It is also important to take a look at history in order to comprehend what elements made e-commerce become what it is today (section History of e-commerce 1.3.2). With that knowledge in hand, we will get a glimpse of the future and decide whether the current e-commerce solutions have a place in it (section Future of e-commerce 1.3.3).

1.3.1 What is e-commerce

As a general definition, commerce is the branch of business that includes all activities, which directly or indirectly are involved in exchanging goods or services. The trade can be held between businesses or individuals, eventually achieving the goal of transferring goods from producers to consumers. When information and communication technologies are applied to support these activities, we are referring to electronic commerce, also commonly known as e-commerce [Akr11].

Currently there are four major types of e-commerce, classified based on the roles involved in the trade: business-to-business (B2B), business-to-consumer (B2C), consumer-to-business (C2B) and consumer-to-consumer (C2C). Other lesser types may involve roles such as government, employee or manager in order to define more specialized e-commerce business models. Though any of those types can be considered to be subtypes of the four major models [Nem11].

Business-to-business comprehends those commerce transactions held between businesses. In e-commerce the volume of B2B sales is around twice the size of B2C [Hoa12], mainly because a typical supply chain has multiple B2B transactions involving suppliers, manufacturers and

distributors; while there is only one B2C transaction with the end customer at the very end of the chain. Communication and collaboration interactions between businesses and within companies are also included as part of B2B, in the form of email or a more specialized system to exchange business data, like Electronic Data Interchange (EDI).

Business-to-consumer describes all activities involving businesses providing products or services to end consumers. B2C is the type of e-commerce with higher number of sales behind B2B, but it is by far the most familiar amongst the population. Companies invest a lot of resources in improving the customer experience when interacting with their e-commerce interfaces. These can be in the form of electronic retailing or media to communicate with customers such as email. Electronic retailing, also known as e-tailing, is the largest part of B2C e-commerce, consisting of an online catalog of a retail store or shopping mall, usually called web-shop when it is accessed from the web.

In consumer-to-business the B2C process is reversed, thus the end customer is the one that offers goods or services to the company to complete a business process or gain competitive advantage. It is common in specialized blogs and forums, where companies offer money to their owners in exchange of advertisement or review of their products.

Consumer-to-consumer is a way of e-commerce where a third party facilitates transactions between consumers electronically. The most popular example of C2C e-commerce is the online auction, in which customers bid to purchase an item that another consumer has posted for sale, while the third party charges a commission for the exchange.

Besides these four major forms of e-commerce, there are other interesting concepts that have gained popularity these last years: social and mobile commerce. Social commerce is the adoption of social networking features in the context of e-commerce. When it comes to offline shopping, it is a natural practice to gather information from oneself's personal social networks before purchasing a product. People usually consult their family and friends for advice, and they often speak to the shopkeeper about suitable products. Joining social networks with online stores allows customers to have the same experience, with the advantage of reaching a largest range of people in a shorter time [GWL11].

On the other hand m-commerce, an abbreviation for mobile commerce, is any kind of e-commerce activity that relies on the usage of wireless devices, such as cell phones, personal digital assistants (PDA) and smartphones. The range of devices enabled for m-commerce also includes general purpose wireless computers, like tablet and laptop computers [TNW01], but

are not usually part of research studies. The reason behind that is the existence of hybrid devices between mobile phones and computers, such as smartphones, that are more specifically designed for m-commerce.

1.3.2 HISTORY OF E-COMMERCE

E-commerce has been gaining more and more relevance in the business context since the moment it was introduced back in the mid-sixties, when the standard that became known as EDI was developed and started replacing traditional mailing and faxing documents. Later in 1979 the British inventor Michael Aldrich invented what he called teleshopping, an early version of online shopping [Ald10].

The system consisted of a domestic television connected via telephone line to a real-time transaction processing system, with a shopping transaction program installed. It used a slightly modified television with capabilities to communicate via domestic telephone line thanks to a modem chip originally used in the Prestel system[2]. Aldrich's system was not properly using the Prestel system, but the Prestel data transmission protocol to communicate with computers via telephone line, and therefore to convert televisions into real-time terminals [Ald11].

[2] System developed by the British public telephone system, whereby news and any text information were received by a television through a telephone line.

Figure 1.2: 1979 pre-production online shopping system by Redifon.

Source: Michael Aldrich Archive

Redifon Computers[3], the company for which Michael Aldrich was working at that time, started selling this online shopping system (Figure 1.2) and installed the first operational application for Thomson Travel Group[4] in 1981. Aldrich initially designed his system for B2C online shopping: it worked from an inexpensive domestic television, with simple human interface and using domestic telephone line; despite of that, initial demand was B2B online shopping for holiday travel, vehicle and spare parts, sales, loan finance and credit ratings.

It was in 1984 when Aldrich's teleshopping system finally reached Jane Snowball's home, a seventy-two-year-old woman who became the first ever online home shopper when she ordered some groceries from the supermarket chain Tesco (Figure 1.3). The system she used was called GSS (Gateshead Shopping Service), and was part of a social service experiment project in the English city of Gateshead, aimed at elderly people who were not able to go shopping. Another larger project appeared two years later in another city of England, Bradford, for disadvantaged citizens. In both projects it was necessary to develop an early version of what we know today as a cart shopping system [Ald09].

---

[3] Company belonging to Rediffusion Group of Companies.

[4] Currently known as Thomson Holidays.

Figure 1.3: Mrs. Snowball ordering groceries from her home in 1984.

Source: Michael Aldrich Archive

Elsewhere in Europe similar systems appeared which involved an interactive television using telephone line. Especially important was Minitel system invented in 1982 by France Télécom, that can be considered the most successful of all these early online services. But even in this case teleshopping was only successful in some B2B activities. B2C was not commercially viable due to the difficulty of common people to access the necessary technology. The only working systems were social experiments run by local governments in partnership with supermarkets to deliver groceries to senior and disabled citizens [BB05].

E-commerce needed a way to reach a wider variety of people to work, especially outside business-to-business context. Tim Berners-Lee offered that possibility in 1990 when he joined hypertext technology with the Internet creating the World Wide Web [Ber00]. Despite of having the technology, commercial use of Internet was not allowed[5] when the web appeared. In 1991 this restriction was lifted, but only under the condition of paying a fee according to the usage of the network, which was destined to fund the networking infrastructure. These limitations were also resolved in 1995 when commercial use of the Internet became completely free [Off93].

Before that, in 1994 Netscape launched the first commercial browser, with the cryptographic protocol SSL along with it. With the web being accessed by an increasingly amount of people

[5] In 1990 most of Internet backbone networks belonged to the National Science Foundation Network. This network was destined to research and educational purposes and had an *Acceptable Use Policy* that prohibited purely commercial traffic from using it.

and with a protocol to ensure secure online sales, companies finally had the chance to build a profitable business for B2C in an expanding environment. The first web-shops started to appear, as well as e-commerce solutions built to allow merchants sell online. Only one year later Amazon.com and eBay were born, both considered to be amongst the largest online store corporations nowadays.

Of course all these changes were accompanied by a revolution in payment systems. A series of innovations have been introduced to our daily life during the last thirty years, being the most significant to e-commerce: debit and prepaid cards, online banking and mobile payments via cell phone. All of them contributed to increase the number of payment service providers offered, thus facilitating online payments. Nowadays one of the most important e-commerce payment systems is PayPal, in charge of processing payments between the merchant and the acquirer bank, therefore allowing to send and receive payments securely over the Internet [KRSS12].

1.3.3 FUTURE OF E-COMMERCE

Despite of being only forty years old, e-commerce has become a very important area in the business environment. Looking back we see the way every technology has changed the e-commerce scenario and given more importance to it. From new protocols to innovative devices, including payment systems and social trends; all has been quickly adopted by companies in order to gain competitive advantage. The introduction rate of new elements in the e-commerce context seems to have grown exponentially over the last few years, as well as the worldwide population involved.

In North America the percentage of digital buyers is currently of 72% of each Internet user and is expected to grow up to 77.7% by 2017. A similar growth is expected in Western Europe, with a great difference between northern and southern countries: U.K. and Germany account 87% and 80% of e-commerce customers in 2013, respectively, and is expected to grow around 3%. On the other hand, in Spain the percentage is 54% and in Italy 44%, with a predicted growth of 10%. The highest penetration rate of Internet users in e-commerce will happen in China, where the amount of digital buyers is going to increase 20% [eMa613].

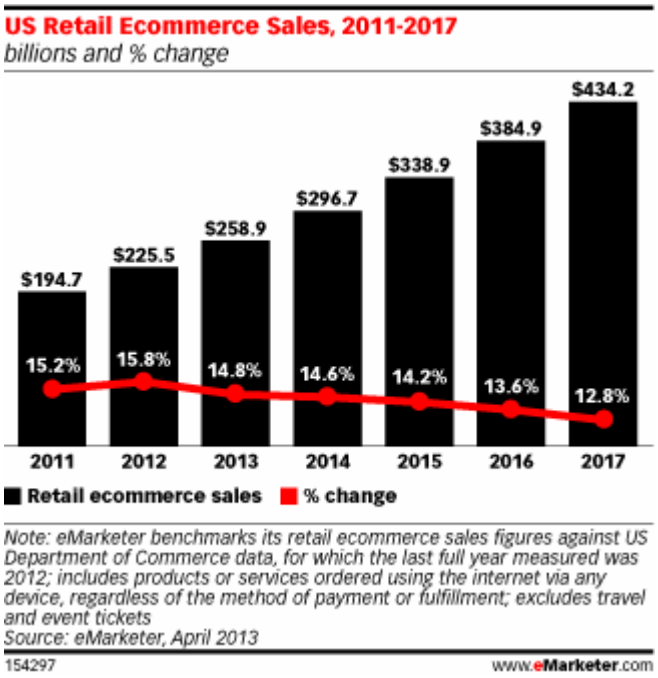


Figure 1.4: U.S. retail e-commerce sales from 2011 to 2017.

Source: eMarketer, April 2013

In number of sales, all major studies foresee a continuous growth in U.S. e-tailing income in the next few years at a compound annual rate that goes from 10% to 14%, reaching between $370 and $434.2 billion from e-tailing sales in 2017 (Figure 1.4) [eMa413]. When it comes to Western Europe obtained results are similar with a growth rate of 11%, reaching even 18% in southern European countries where the e-commerce market is not yet as mature as in North America [OG13]. Same reason applies to Asia and Latin America with the highest growth rates in the world, around 25% growth per year. Particularly high are the growthing rates in China and Indonesia, where a 65% and 71% is expected in 2013, respectively [eMa613].

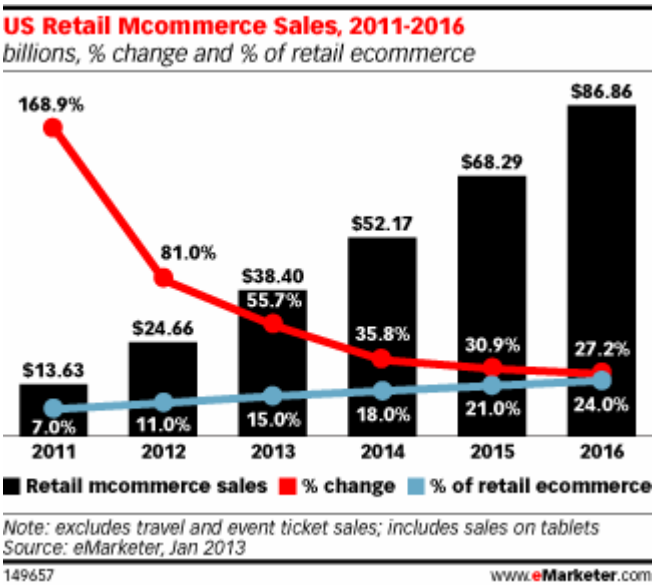


Figure 1.5: U.S. retail m-commerce sales from 2011 to 2016.

Source: eMarketer, January 2013

It is also a fact that mobile devices are being quickly adopted by both merchants and customers, due to all possibilities that they offer in the expanding e-commerce scenario. Some studies show how retail sales made on smartphones will grow from $8 billion in 2012 to $31 billion by 2017, becoming a 9% of e-commerce total sales [MERJ13]. When tablet computers are also included in the research, m-commerce sales grow from $24.66 to $86.86 billions in 2016, having 24% of retail e-commerce (see Figure 1.5) [eMa113].

A look at the future of e-commerce reveals a continuous growth in sales and customers, as well as the fast adoption of new technologies such as mobile devices. Therefore it can be expected that new devices and different ways of commerce activities will appear in the near future. It will be therefore necessary for merchants to integrate all their existing e-commerce infrastructure in every context in order to gain advantage from the expected growth.

1.3.4 CURRENT ALTERNATIVES

Are the current e-commerce solutions ready for bringing quick and affordable integration to future scenarios? Almost all shopping cart solutions offered for online shopping are designer-oriented built web-shops, with multiple plug-in components, customizable options and

exchangeable templates. The merchant requires the e-commerce solution to offer a certain feature set in order to use it, otherwise the cost of implementing it may not be worthwhile or simply impossible.

With the raising of cloud computing, many licensed products have move to a more flexible software-as-a-service (SaaS) model, allowing merchants to easily scale their web-shops as their businesses grow. Despite of that, merchants are still very limited to what the software is offering, and depend on the product to evolve in order to expand their e-commerce infrastructure to different environments. Of course, they can also use an independent product to support the missing scenario, but with the high cost of having to maintain two or more different backend data, having to connect them all together.

A very interesting example of these models is Magento, a PHP open-source project that was initially launched in 2008 and nowadays enjoys great popularity. The first version offers a typical out-of-the-box web-shop, highly customizable and with a wide variety of plug-in and templates. The experience of building a web-shop with it was very comfortable, since all the changes were done directly with an administration interface, not requiring any technical knowledge.

Everything was comfortable until the moment the template was not offering a functionality or simply not the way it was needed (e.g. displaying the breadcrumb in a different way or place). In that moment finding the files where the logic that needed to be changed was located became a very hard task and changes were not easy to make either. This experience reflects exactly how developer-unfriendly these kind of models usually are.

Magento also offers a SaaS version of this product, called Magento Go. The experience is even worse, since any kind of customization is limited to what they offer in the administration page, impossibiliting any modification on the code. In 2010 Magento announced the release of the second version, Magento 2.0. This version promised to be more developer friendly, but so far the product has not been released.

1.4 PLANNING

The planning of the project includes the choice of the methodology used and a description of the development process. According to the chosen methodology and the analysis of the major risks of the project, an initial timeline is proposed. It is important to understand that the

current project is part of a bigger project that involves the development of a template that integrates all the functionalities offered in the platform. That means that both projects are connected, hence the teams have to work closely.

1.4.1 METHODOLOGY SELECTION

Some considerations must taken into account to decide the methodology that is going to be used in this project. Scope, timing, risks and objectives, as well as the methodology already used by the SPHERE.IO team; turned out to be the most decisive criteria to determine what methodology is more appropriate in this particular case. An explanation of each criterion is presented below.

On the one hand, the scope of the project is just limited in the present thesis, but the scope of the whole project is attached to the scope of the platform project, which is undefined due to the nature of it. The few milestones defined by the company are meant to be more a guidance and a time deviation is expected. The same applies to the deadline of this thesis, which is not yet fixed and can be moved forward if necessary.

When it comes to objectives it is required to generate rapid and continuous value, because the template is going to be used in company presentations and different events before the official release. Besides any developer interested in the platform should be able to use it as an example of the current capabilities of the system. Consequently regular small releases of the template with new added functionalities are highly desirable.

According to these arguments, an agile method is preferred over a heavyweight methodology in this situation, because of the undefined scope, the flexible deadlines, the major use of new technologies and the need of rapid value [Kha04]. Given that the methodology used by the SPHERE.IO team is based on Scrum, an iterative and incremental agile software development framework, it is only natural to follow the same methodology as they do. Therefore the design and implementation of the template is going to be integrated inside the SPHERE.IO development process.

Although all the documentation necessary for the software development is considered part of the development process and thus it is included in the SPHERE.IO workflow, it has been decided to exclude the proper wording of the present documentation from this workflow. The reason is that Scrum does not explicitly cover the drafting of such a large documentation with

no direct influence on the software implementation. In this particular case a simple sequential development process allows a better planning for this set of tasks.

1.4.2 DESCRIPTION OF THE METHODOLOGY

Scrum follows the principles of the Agile Manifesto[6], characteristic of all agile methodologies. In order to adapt these principles, Scrum defines a number of flexible practices that can be followed in software development projects to improve some processes [SS13]. Many tools can be used to support these practices, in this particular case a simple project management software called AgileZen is used to manage and assign tasks. Next it is described the methodology as it is used in the current project.

In Scrum, the development process is divided into small cycles called sprints. Every two weeks a "Sprint Review Meeting" and a "Sprint Retrospective" is held to conclude a sprint cycle. In the Sprint Review Meeting each member of the team shows a short demonstration of all the tasks he was able to complete during the sprint. On the other hand, in the Retrospective Meeting all elements that went well and wrong in the past sprint are written down in order to improve them during the following sprints.

The same day after an hour break, a "Sprint Planning Meeting" follows in order to prepare the next sprint. In this meeting all the team decides the next sprint's tasks, which are then assigned to the corresponding members of the development team. In Scrum, the group of tasks of a sprint is called the "Sprint Backlog" and is selected from the project's requirement list, also known as "Product Backlog". Both lists are managed collaboratively with AgileZen, the project management tool referred above.

Every day of the sprint starts with the so-called "Daily Stand Up Meeting" at 10:15, when each member briefly explains the tasks he did the day before and the ones he plans for today, as well as mentioning any blocking issue found. This is done via updating the task board, a physical whiteboard where each task is represented with a sticky note. Those notes are being moved through all the different status (i.e. "to do", "in progress", "done") until completion.

---

[6] See more information: http://agilemanifesto.org

1.4.3 RISK MANAGEMENT

All projects have risks threatening their smooth development. Agile methodologies are already reducing negative effects of unexpected outcome thanks to the fast delivery of working software, that allows to quickly detect and fix any problem without major issues. Despite of that, risk management is advisable in order monitorize and evaluate all major risks every start of the sprint. For that reason risks should be identified along with a strategy to manage each risk beforehand.

The current project is particularly risky due to the extensive use of a technology under development. The SPHERE.IO platform may not yet allow all functionalities planned for the template or it may be delayed some weeks or even months. That would directly affect the current project's timeline. This is a very likely risk to happen and with a high impact on the project, but the project itself would be meaningless without the use of this particular platform. Therefore the only way to deal with this risk is tolerate a large deviation in the project for high priority functionalities and dismiss those functionalities with lower priority when a long delay takes place.

Some other technologies in development or with no previous experience working with them will be used in this project. In each case, some research should be done before in order to guarantee that it fits the project. In spite of this, if the technology appears to be unsuitable for the project during the implementation process, it should be replaced with an alternative technology or the functionality it provides should be discarded.

Many other risks are related to the fact of developing the project in a business company, especially in a startup. These kind of companies are particularly prone to change, whether be it a change in the development team, in the company's activities or simply the company ceases to exist. In any case, these risks cannot be avoided. If the project is on a very early stage the best decision would probably be to start a new project, otherwise the topic can always be adapted to the new situation.

1.4.4 INITIAL TIMELINE

As described before, in Scrum a new planning is elaborated from scratch every start of the sprint. Additionally, requirements are likely to change over time. These characteristics make an initial timeline not entirely befitting for a project following an agile methodology. Despite of

that it has been considered appropriate to prepare an initial planning of the whole project (see Figure 1.6) in order to make a rough estimation of the total amount of work. This initial planning will be updated every end of the sprint.

The project officially starts February 1st 2013, when the task of developing the first public template of the platform was assigned as the main topic of this project. Before that, a testing period of SPHERE.IO Play SDK took place with the implementation of a basic sample web-shop. This period had the objective of adapting the SDK to Java developers, at the same time that the toolkit was being built. Although the tasks during that stage were no planned parts of this project, it is fair to include it as a training period due to the level of importance of that timeframe in order to become familiar with the platform.

On April 2nd 2013 the SPHERE.IO platform was going to be publicly announced as a beta release, along with the first template providing some basic functionalities (i.e. browsing and purchasing products). The implementation of the template was expected to finish on April 10th 2013, before the platform's final release on July 1st 2013.

After the development cycles are finalized all remaining time is being used on writing the documentation. Some usability and performance tests will be run after the platform is released and stable, planned on July 16th 2013. When the documentation stage finishes on August 18th 2013, and after a week of revision of the whole project, the presentation is being prepared over five days. Therefore on August 26th 2013 the project was expected to finish. Given that the deadline of the project's presentation is on January 24th 2014, it allowed a time deviation of up to five months.

In order to count the total amount of hours of the timeline, it must be taken into account that every stage might have a different dedication time. It is considered in this planning that the training period takes 1 hour per day, because as explained before the tasks in that time were not focused in this current project. The documentation stage takes around 4 hours per day in order to combine it with other projects' tasks, while the remaining stages have 6 hours per day on average. With that, this planning results in a total of 725 hours of work, divided in 73 hours for training, 60 hours for specification and design, 240 hours of implementation and 352 hours in documenting the project.

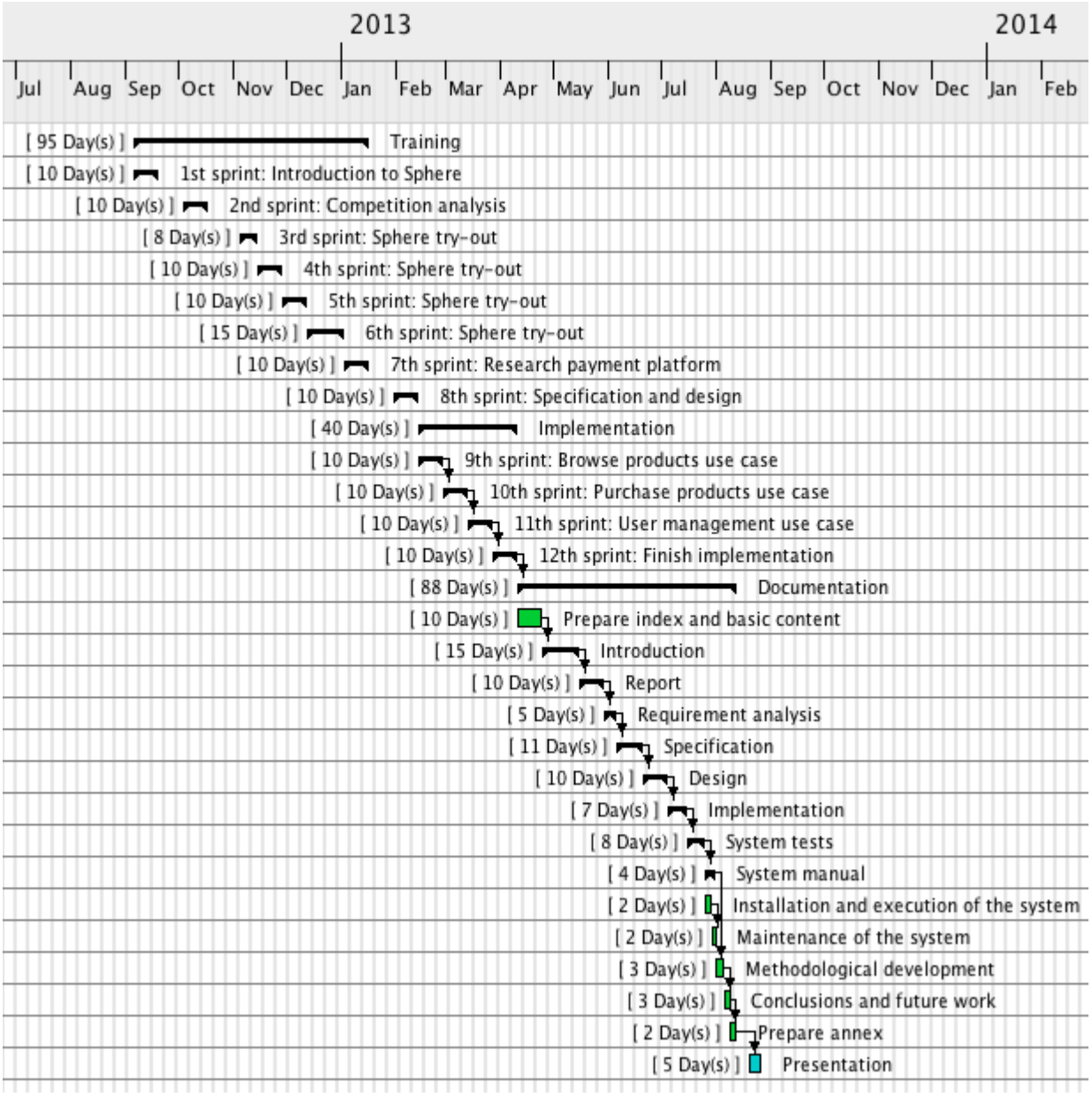


Figure 1.6: Gantt diagram of the initial timeline of the project.

2 REQUIREMENT ANALYSIS

In order to start gathering requirements, first it is necessary to identify each group affected by this project and understand everyone's needs (section Stakeholders Analysis 2.1). With that information in hand, an initial list of the desired functional and non-functional requirements (see sections Functional Requirements 2.2 and Non-Functional Requirements 2.3) can be put into the Product Backlog in the form of user stories. Every sprint these requirements may change, reason why in this section are described only the final requirements that are part of the current Product Backlog of the project (see Appendix B Product Backlog).

2.1 STAKEHOLDERS ANALYSIS

Potential clients of the project, simplified as "clients" from now on, are companies in need of an e-commerce solution, especially those looking for an interface from where to sell goods or services to individuals or other companies. They need a system that satisfies their current requirements and allows them to easily implement future required functionalities. They hire developers, usually working in agencies, to build these tailored applications.

Potential users of the template are precisely developers, who need to implement a web-shop or a similar application. They might have to decide whether to develop a web-shop based on SPHERE.IO Play SDK. They may want to use the template as a live documentation of how to use the system or directly use it as a bootstrap project on which to build their own web-shop. They need to easily understand the sample code and quickly identify those lines of code with shop logic coming from SPHERE.IO. Code quality and correct use of technologies may be important for them.

End users or customers[7], equally mentioned as "users" and "customers" in this document, are those actors of the system who would buy in the web-shop in case it went live. Even being an hypothetical situation, this template will be used as a bootstrap project so eventually it will become a real web-shop with end users. With no other specific information about these users, it can only be assumed that they need an intuitive layout that allows them to shop with ease in a few clicks.

[7] For more information about the differences between the term "customer" and "client", see http://www.dailywritingtips.com/customer-vs-client/

Inside the company there are two major stakeholders: the SPHERE.IO product owners and the developer team. The product owners need a final product where all the platform features are implemented in order to measure the actual progress of the project. This product also allows them to have a sample web-shop to show to potential clients in meetings and conferences even before the platform is released.

On the other hand, the development team is in charge of designing and implementing the platform that the template is using. Their primary need towards the template is to verify that their implementation is adjusting to both developers and clients needs. They might create temporal limitations on the template design, but at the same time any suggestion may be quickly adopted with no need to change the web-shop requirements.

2.2 FUNCTIONAL REQUIREMENTS

In order to prove the value of the platform and identify any possible lack of functionality, the application should have all the common features of a regular web-shop. Accordingly, it has been considered that the initial appropriate set of functionalities for this project include those related to browsing and purchasing products, as well as management of a customer account. The detailed behavior expected for the web-shop is described below.

In the home page all products are displayed sorted by popularity. From here the user can select a category; then all products belonging to that category or any descendent will be displayed. Whenever a set of products is listed, those products on sale will be highlighted and the user will be given the option to sort and filter amongst all products. The sorting can be performed by name or price, and the filtering by price and color. Each product thumbnail consists of a picture of the product, its name and price, as well as a list of all color variants available.

Clicking on a product thumbnail redirects the user to the product detail page, where besides name and price also a description is shown. Here the user is able to select any color and size to visualize the corresponding picture. In any moment the user can add the selected product to the shopping cart, afterwards the updated cart contents and total price will be displayed. Accessing the cart details also grants the user the possibility to change the number of units of an item or remove any particular item from the shopping cart.

From here the user can choose to start the checkout process, where he is asked to fill a form with shipping information (i.e. shipping address and method) and billing information (i.e. billing address and payment data). During the checkout process the order summary (i.e. the list of purchased items and pricing details) is displayed all along and kept up to date. Right before finishing the checkout process, the user is informed of all introduced shipping and billing information as well as the order summary. Once the checkout process is finished, another summary is shown along with a successful purchase message.

The user can decide to sign up in our systems, in which case he must provide his full name, email address and a password. After signing up he is redirected to his user profile, where he can update his personal data, change his password, manage his address book or review his previous orders in detail. The address book allows the user to store a set of postal addresses that can later be selected as shipping or billing address in the checkout process. The user is allowed to add new addresses to the address book, as well as update or remove any stored address.

While logged in, the user can choose to log out in order to become an anonymous customer. In any moment, he can log in again providing his email address and password. In case the user forgot his password, he can request to recover it by entering his email address, in which case an email is then sent to the address provided containing a web link can then be accessed within the next hour, where the user can provide a new password.

2.3 NON-FUNCTIONAL REQUIREMENTS

In its first stage, the web-shop template is required mainly to analyze the platform capabilities, show code examples to developers and attract potential customers. For this reason all non-functional requirements are highly focused on those areas. Other areas of great importance as well, such as compatibility and performance, are left aside from the current project because of the excessive workload that it means.

From a developer point of view the quality of the code takes a very important role, so it should be well organized, easy to understand and reusable. Therefore it would be considered a good practice to use variables and functions with self-explanatory names and keep a well commented code. To the extent possible, the generic shop logic should be separated from the most specific code in order to facilitate the use of it as a live documentation of the platform.

The platform should allow to test any web application built on top of it. In order to prove it is allowed, the template should be completed with automated functional tests, being careful of keeping these tests independent from the backend data in use. That way a change in the data, very likely to happen in a template web-shop, will not affect the results. The same principle should be applied to the code in general, to keep the template from being non-functional when the data used is different.

Although major part of the required security is located on the e-commerce and payment platforms, there are some risks server side that must be top priority when it comes to online shopping. For example some data needs a careful treatment, like user related data such as addresses, passwords and payment information. Particular attention must be paid with the checkout process in order to avoid fraud.

When online payment is involved in an application, payment data needs to be processed and stored somewhere. The system to process and store this data needs to be PCI DSS[8] compliant. Being a sample web-shop it is most appropriate in this case to leave this role to the payment platform, thus sending any payment data to the template's web server must be strictly avoided.

The template should be intuitive and use latest design tendencies, especially those allowing a faster navigation experience. The user should be able to use all functionalities of the web-shop in a smooth way, trying to minimize the number of times the page is fully reloaded. This will also speed up the communication with the web server, thereby favoring a more efficient interaction with the web-shop.

The colour scheme should be neutral but pleasant in order to match any web-shop topic, with a winter sports related theme. The URL structure of each page needs to be user-friendly, meaning it has to be easily identifiable with the product or category linked when reading it. At the same time it has to follow some basic SEO rules in order to promote any website based on this template.

[8] PCI DSS stands for Payment Card Industry Data Security Standard and includes a set of twelve specific requirements to cover six different goals, in order to create a secure environment for payment data.

3 SPECIFICATION

Agile methodologies suggest to elaborate documentation only as needed, without having any required artifacts for each stage as traditional methodologies usually do. The reason why heavy documentation is not recommended is because requirements are expected to change constantly during the development process, forcing to update every diagram and text each time a change is applied, with the consequent loss of time that could have been otherwise used to develop the product.

For this reason, only some simplified diagrams were drawn during the specification and design stages, the necessary to understand the system and share ideas with the SPHERE.IO team. Therefore most of the artifacts presented in both this section and section Design 4, were made after the product was already built, intended to assist the reader in understanding better the system.

The specification section here presented describes the necessary system to fulfill the functional requirements previously gathered. Here are first described the set of use cases that are initially planned for the project, which corresponds to the final Product Backlog (see list in Appendix B Product Backlog). The conceptual model of this system is then presented and, for each use case, is explained the expected behavior of the system with the user.

3.1 USE CASE MODEL

There are three actors that interact with the system: the customer, the payment platform and the SPHERE.IO e-commerce platform (Figure 3.1). The customer can either be an anonymous customer or an identified customer previously existing in the SPHERE.IO platform. Since the required functionalities of the present project were mainly designed to test the SPHERE.IO platform, it is no surprise that the platform is present in every single use case of the system whatsoever, so for the sake of readability it will be omitted from the use case diagrams henceforth.

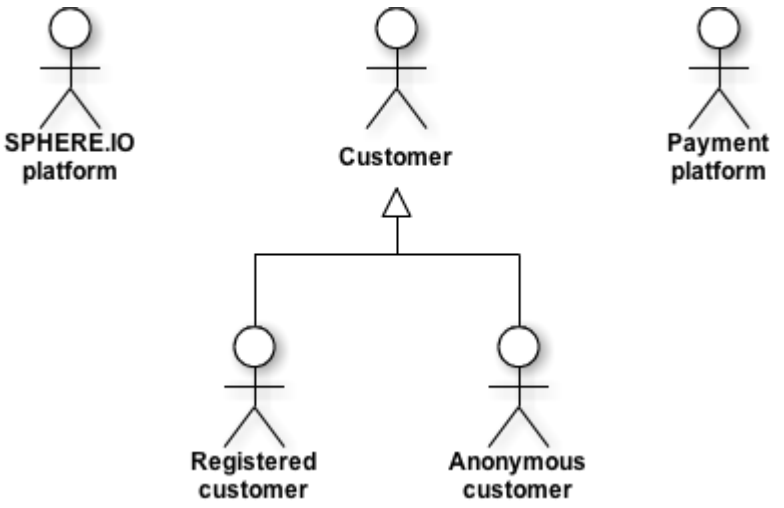


Figure 3.1: Diagram of the actors involved in the system.

As mentioned earlier, the system has three functionalities where all use cases fall into: display products, purchase products and manage customer account (Figure 3.2). The customer is present in all use cases of the system, while the payment platform is only involved in the functionality for purchasing products.

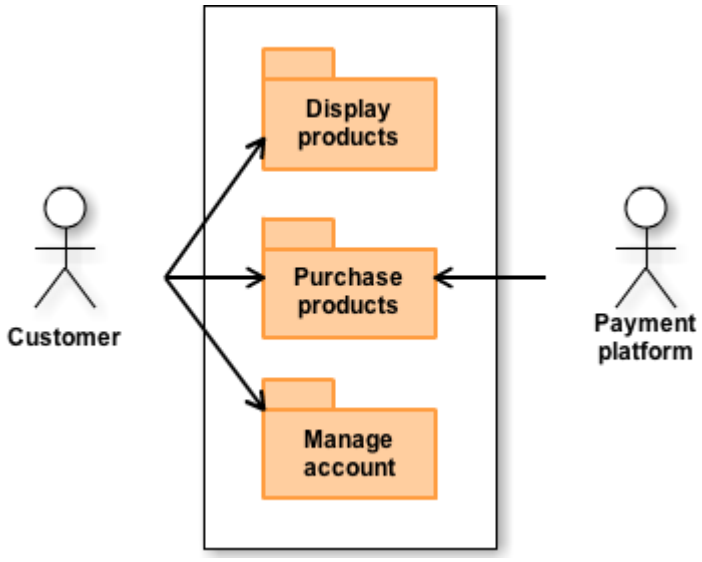


Figure 3.2: Diagram of the use case packages of the system.

The use cases for displaying products are shown below in Figure 3.3. The customer can either list a set of products or display a particular product. Further additional functionalities can be applied to the product listing, individually or combined together, in order to alter the list itself (i.e. filtering) or the way the products are listed (i.e. sorting and pagination).

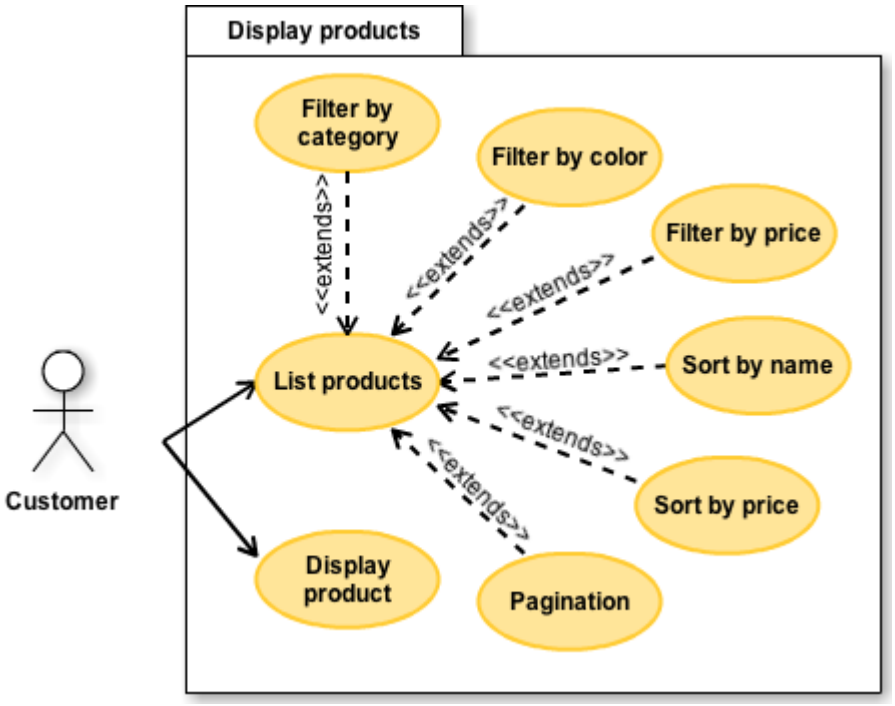


Figure 3.3: Diagram showing the use cases of the *display products* package.

Figure 3.4 shows the use cases related to purchasing products. They can be clearly divided into two different topics: on the one hand all those use cases for managing the shopping cart (i.e. adding, updating and removing items), on the other hand those related to placing and listing orders. When placing an order the customer may be requested to pay online, in which case the payment platform will provide the necessary means. Anonymous as much as registered customers can place orders, but only customers that have been identified are able to list their own orders, otherwise they are requested to identify themselves.

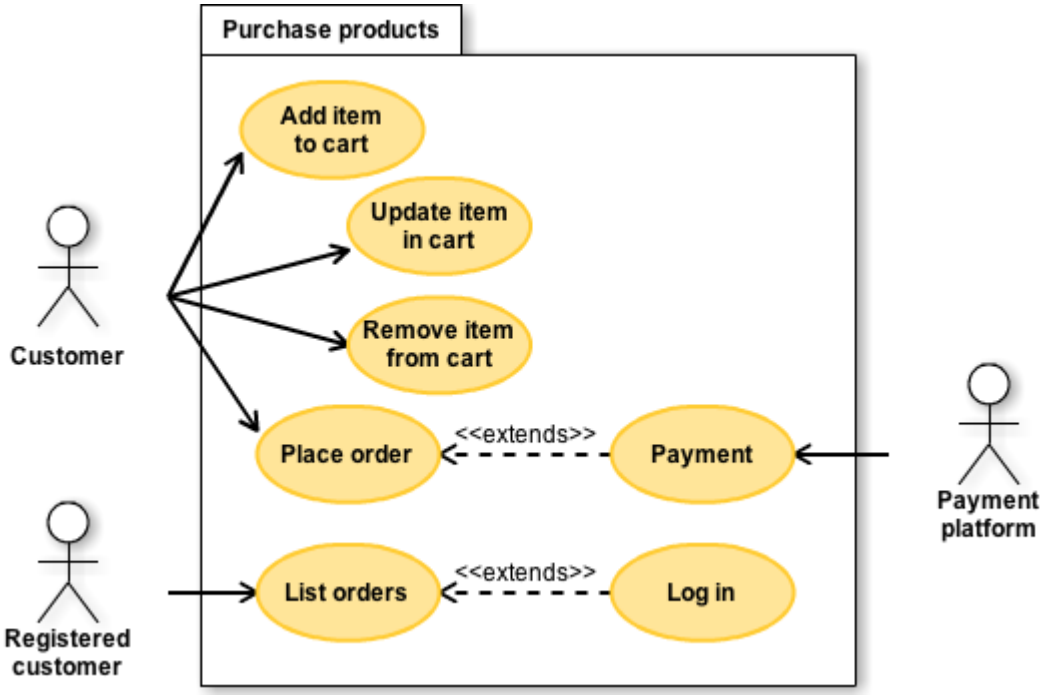


Figure 3.4: Use case diagram showing the use cases of the *purchase products* package.

Finally, for the use cases related to account management (Figure 3.5), a registered customer can manage his address book (i.e. add, update or remove postal addresses) or update his account (i.e. change his personal data or password). He can as well decide to log out from the system and become an anonymous customer. As an anonymous customer, he can sign up a new account or log in with an existing one. In case he cannot remember his credentials, he will be given the option to recover his password.

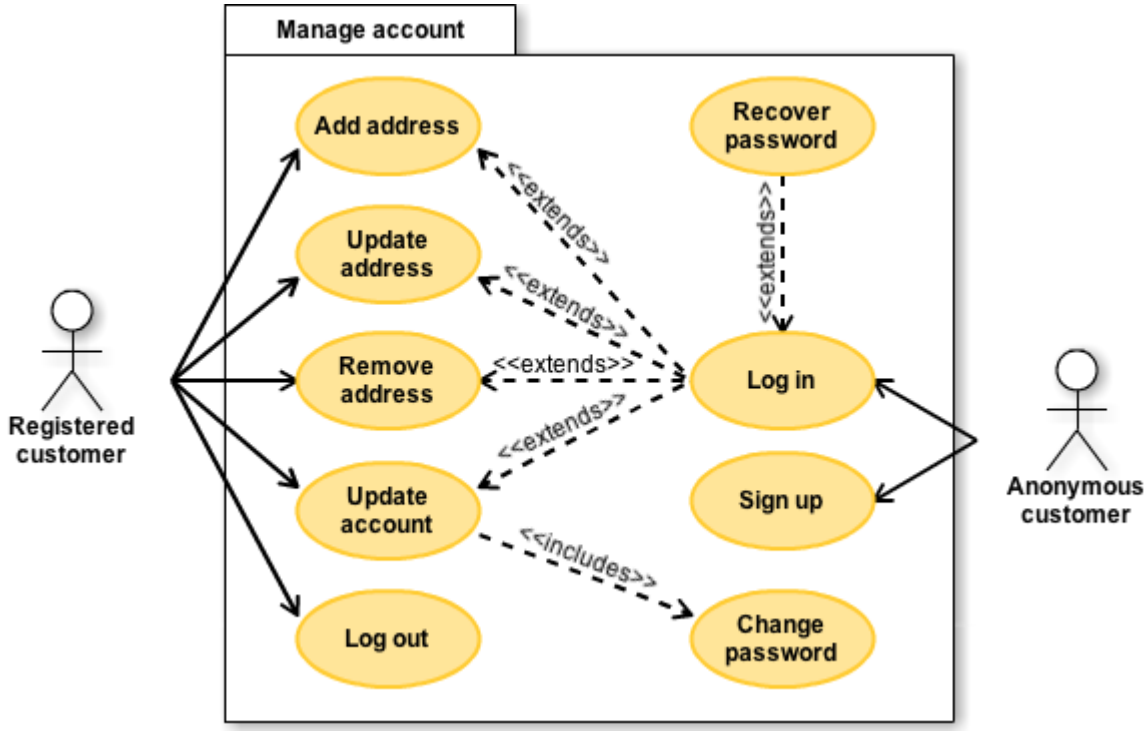


Figure 3.5: Use case diagram showing the use cases of the *manage account* package.

The previously explained use cases are mostly useful to define the scope of the project and understand its functionalities. For example, these use cases can be helpful to estimate tasks and elaborate the development plan, as well as a guide to determine the necessary functional tests for the system. But these use cases are too granular for other purposes, such as defining acceptance tests or describing the sequence of user interactions with the system. These tasks require a more abstract level of use cases, focused on user goals instead of functionalities, sometimes called top-level use cases.

A top-level use case describes a single elementary business process that allows a particular user to fulfill a goal. In this system there are mainly three goals that a customer may want to achieve when he uses the web-shop, as shown in Figure 3.6. The first one consist of browsing the catalog and selecting those products of interest. At some moment, the user can decide to review the selected items and eventually buy them, which is the second goal. Finally, the third goal involves checking the payment or shipping status of the order, or any additional related information.

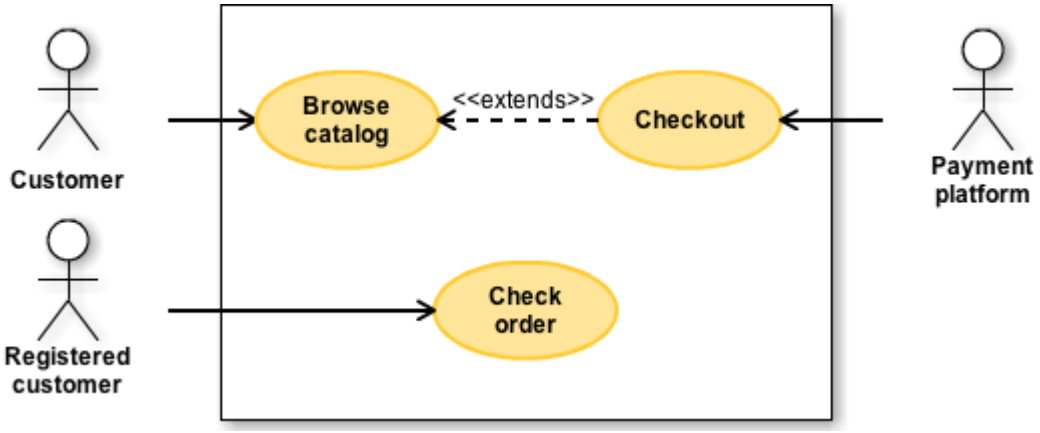


Figure 3.6: Diagram showing the main top-level use cases of the system.

All low-level use cases defined earlier are actually providing the functionalities to fulfill these three goals. Both low-level and top-level use cases are being used indistinctly throughout this document to elaborate other diagrams and descriptions, its use responding mostly to the level of abstraction that fits best the explanation. In any case, the term "top-level" is expressly used when referring to this type of use case.

3.2 System behavior model

Almost all the low-level use cases of this project consist of only one interaction between the user and the system. This may be useful for projects that require very detailed information about the system to be developed, possibly because its behavior is very specific and unique. But this is not the case of this project whatsoever, the use cases defined here are precisely very common amongst web-shops, so any operation offered by this system is considered to be self-explanatory.

As mentioned before, the top-level use cases are here more appropriate to describe the user communication with the system. This is because they provide information not only about the system behavior, but also about the sequence of interactions that the customer usually performs in order to achieve a goal.

Figure 3.7 displays the sequence diagram for the *browse catalog* top-level use case, one of the many possible success scenarios. In this case the user will usually go to the home page, select a

category and then filter or sort the products until he eventually finds one of interest. Then he will probably ask for the details of the product and next he will add it to the shopping cart.

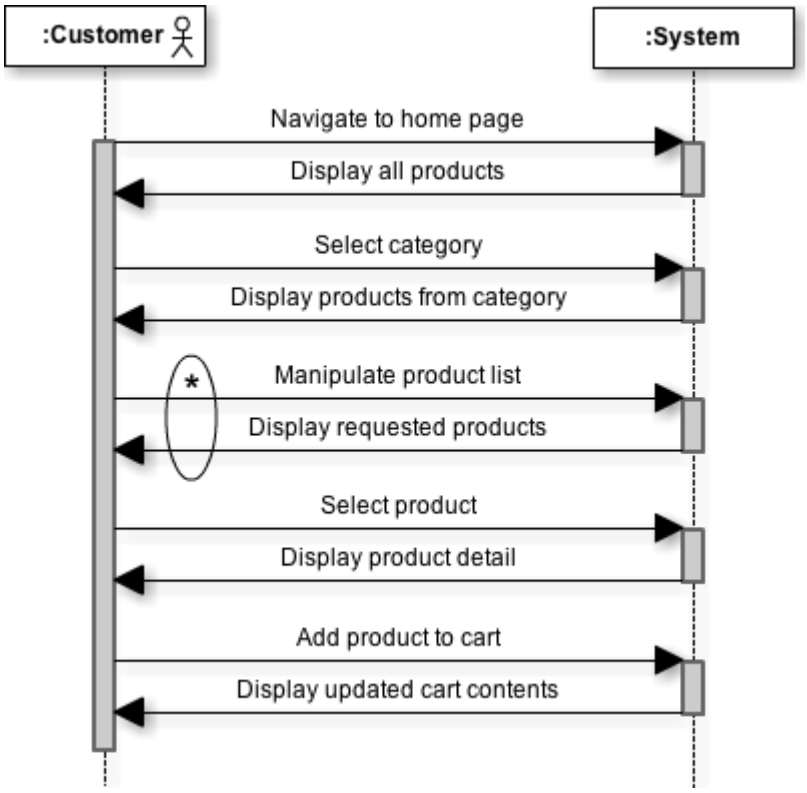


Figure 3.7: Sequence diagram of the *browse catalog* top-level use case, success scenario.

The *checkout* top-level use case is shown in Figure 3.8. Once the customer has some line items in his shopping cart, the next step is to navigate to the cart page. Here the user can remove or modify his line items until he is ready to start the checkout process. There, after entering all shipping and billing information, the customer will confirm the purchase and the system will request the payment platform to process the payment, displaying the order details in response to the customer.

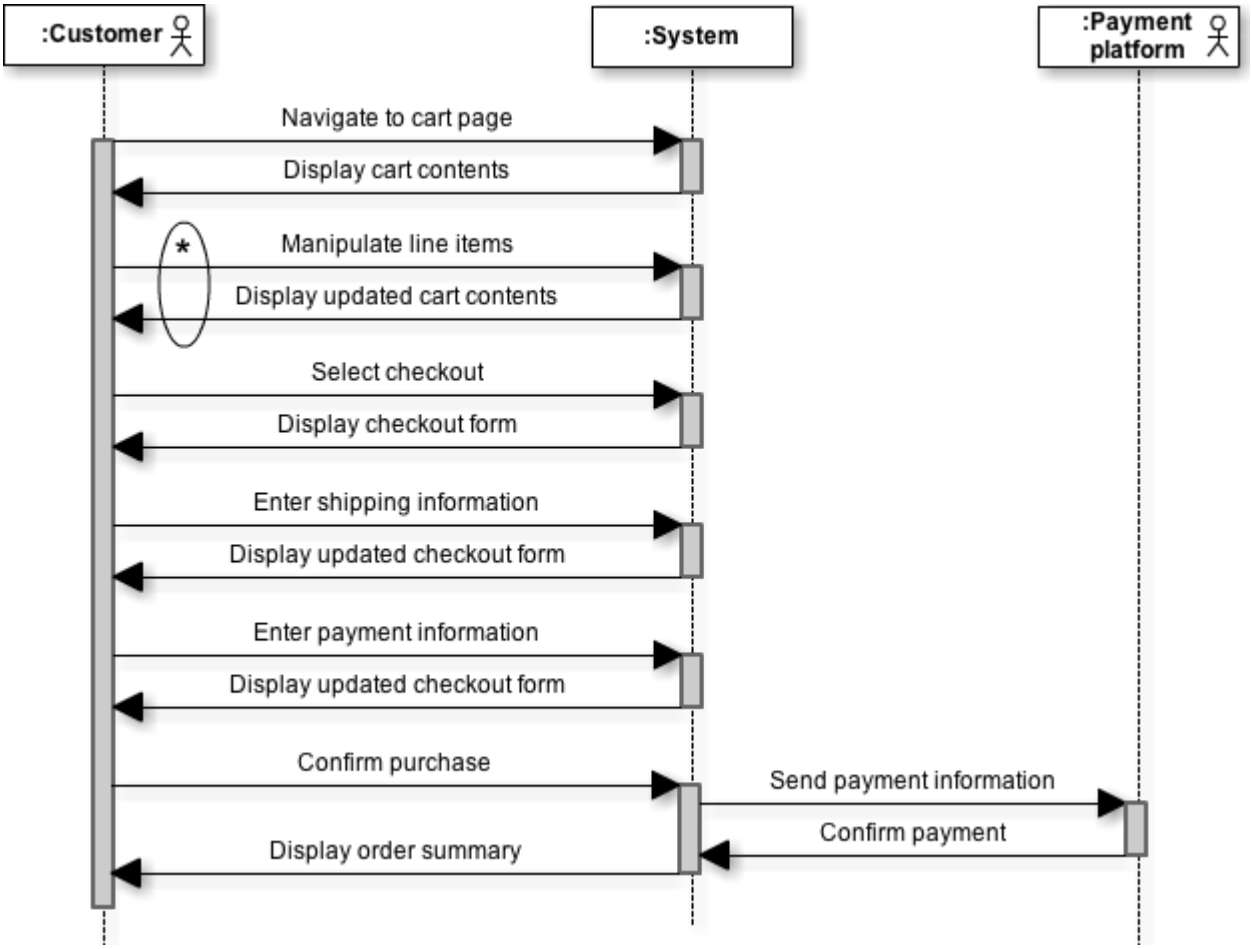


Figure 3.8: Sequence diagram of the *checkout* top-level use case, success scenario.

The last sequence diagram displays the interactions that the customer has to perform in order to check the state of an order (Figure 3.9). This scenario requires the customer to previously sign up to the system and purchase some items as a registered customer. Then at any moment the user can go to the login page and enter the login information to access his customer profile. There he can select to list all his orders and select the one he wants to view in detail.

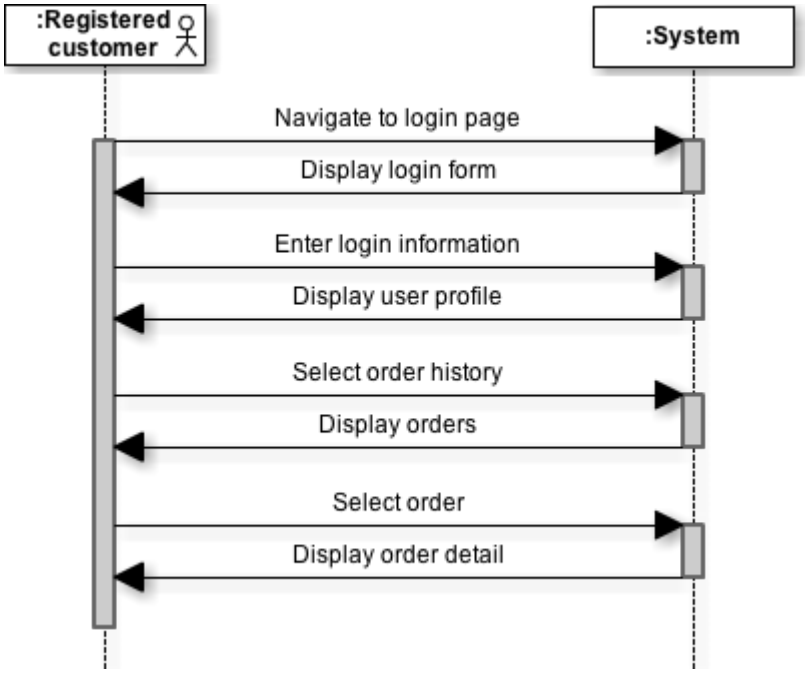


Figure 3.9: Sequence diagram of the *check order* top-level use case, success scenario.

3.3 CONCEPTUAL MODEL

The conceptual model of this project revolves around the cart concept, while all other system elements are there to provide the required information to the cart, as seen in the class diagram below (Figure 3.10). Products are related to carts as a list of product variants, forming line items. Variant is a concept to define the part of the product that contains the particular characteristics of it, such as color or size, even having sometimes a different price. Therefore every product has at least one variant, each one with different price or attributes.

Similarly, a cart can be associated with one of the shipping methods available in the system, resulting in a shipping item, necessary to manage taxes. Both products and shipping methods have a particular tax category, that can be variable for products and fixed in the case of shipping. When one of these elements are added to the cart, a tax rate is assigned to the item according to this tax category and the shipping address of the cart.

As mentioned above carts can have a shipping address, but can have as well a billing address. A cart can belong to a registered customer, otherwise it is considered to have an anonymous customer. Once the checkout is finished a cart becomes an order, with information about the current payment, shipping and order status. If the customer was not anonymous, this order

will be associated with that customer, along with any of his previous orders. Every customer can also have a list of addresses comprising the address book.

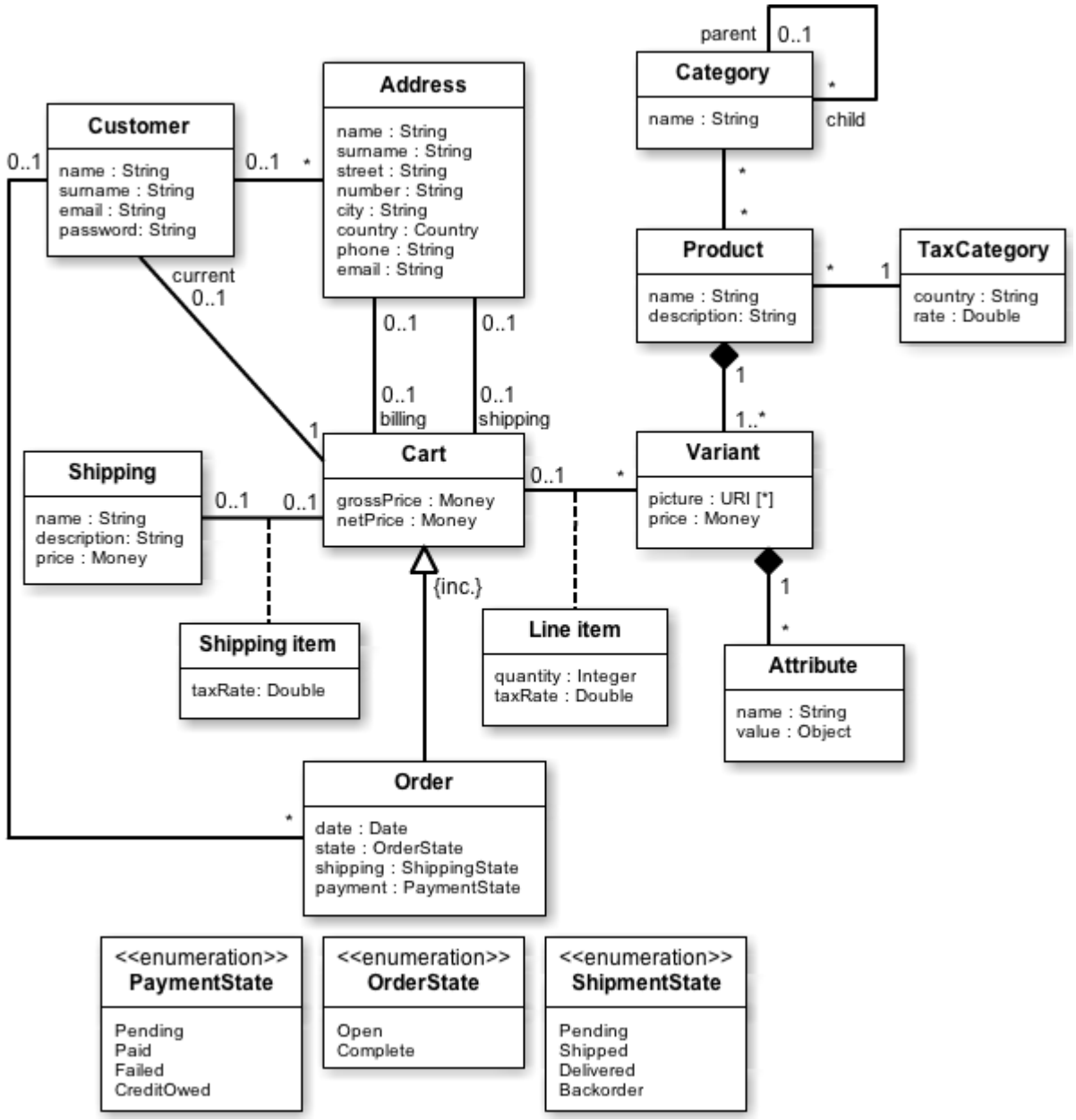


Figure 3.10: Class diagram of the system.

Products, addresses and shipping methods can change or disappear over time, but the orders associated with them must stay in the system for an indefinite period of time, having exactly

the original information. To solve this issue, cart is not related to the original instances, but to instances that were created exclusively for this particular cart as a snapshot of those original instances. While the current cart may optionally have associated information, this information is mandatory in an order instance.

For simplicity, the conceptual model only accepts product and shipping prices that do not include taxes. Allowing taxes in prices can be achieved by simply adding a boolean attribute indicating whether the price in question has taxes included or not. So assuming that taxes are not included, the net total price in the cart must be the sum of all the line item prices (i.e. the quantity in each line item multiplied by the corresponding variant price) associated with it, plus the price of the shipping method selected. In order to calculate the gross total price, taxes must be added up to this resulting net price. Taxes are calculated multiplying the price of each shipping or line item by its corresponding tax rate.

Lastly when the shipping address is set in the cart, all tax rates from shipping and line items are calculated. Only those products that include a tax category corresponding to the zone (e.g. state, country) of the shipping address can be part of the cart. Missing the tax category means that the price cannot be calculated, thus the product is not available in that zone.

3.4 STATE DIAGRAMS

There are two interesting state diagrams of this system, both related to the cart element. The first diagram (Figure 3.11) describes how a cart instance changes until it becomes a complete order. As the diagram below shows, the current cart is the initial state, which allows to change its contents in multiple ways, such as adding or removing line items or selecting a shipping address. Once the checkout is finished the cart becomes an order, being this an irreversible change. From now on the order can only change from an open to a complete state, and vice versa.

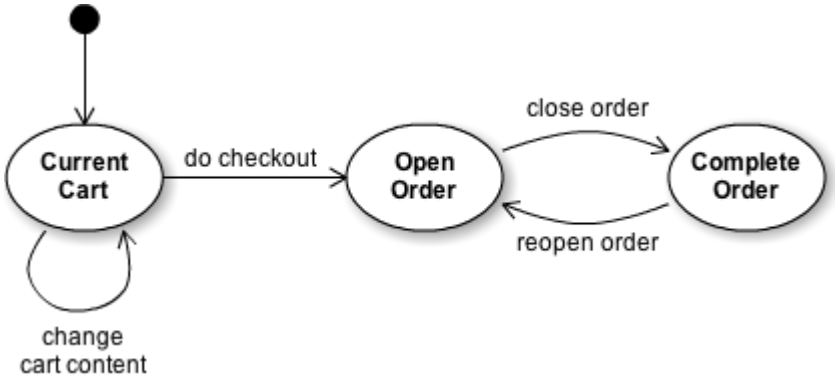


Figure 3.11: State diagram of the cart class.

The second diagram (Figure 3.12) describes the whole process of managing the shopping cart and eventually purchasing these products in the checkout process. This diagram will become especially useful when designing the checkout interface, as it clearly displays the requirements of each step of the checkout process.

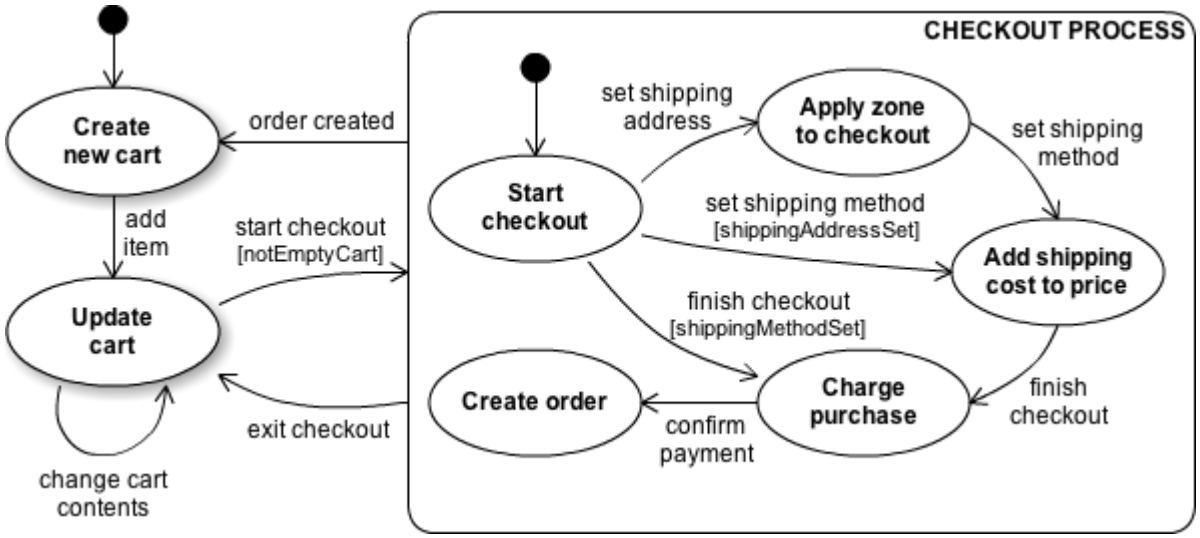


Figure 3.12: State diagram of the use case for purchasing products.

At the beginning of the process a new cart is created. Once the cart contains an item it can be further updated, then at any moment the user can start or exit the checkout process. Initially the checkout process requires a shipping address to display the shipping methods, then it

requires a shipping method to display billing options. Of course this sequence can be skipped if the cart has already these requirements.

When the user provides the billing information and finalizes the checkout, the system charges the customer. The order is then created after the payment platform confirms that the payment was successful. The moment the previous cart becomes an order, a new cart is created for the customer in order to start the process once again.

4 DESIGN

The software design describes the final details of a system before it is implemented. During the design process decisions are taken in order to meet the gathered requirements, decisions that are then applied to the system defined in the section Specification 3. Both physical and logical designs of the system are described in detail in the current chapter (sections System Physical Architecture 4.1 and System Logical Architecture 4.2), with an overview of how the resulting product needs to be implemented. Every technology used is carefully justified and the major characteristics are explained (section Description of Used Technologies 4.2.1).

The selection of a technology is a decisive process aimed to obtain the optimal results of a project. An unwise decision can sometimes seriously affect the total resources needed or the successful fulfillment of the proposed objectives. It is also important to design correctly the structure of the system, for example identifying and applying the software patterns that can solve existing problems in this particular project.

4.1 SYSTEM PHYSICAL ARCHITECTURE

The designed system follows a client-server architecture with three tiers: the client, the web application server and the data server tier. The data tier corresponds to the SPHERE.IO backend, which offers a scalable cloud-based platform for the e-commerce data, having the capability of scaling up as the demand increases. The application tier needs an enterprise hosting solution, suitable for a company web-shop. In order to take advantage of the scalability of the data tier, a good matching web hosting solution would be a cloud service with easy and fast scalability, letting the shop grow as the number of customers grow, without any bottleneck.

At the time the system was designed there were only two cloud platforms with built in support for deploying Play applications: Heroku and Cloudbees; although at the end of 2013 the number of services has been doubled and the offer will probably continue to increase in the future. Both services enable a simple automated deployment of the web application to the platform, which will allow developers to have a working hosted application within minutes.

The specific hosting solution used for this project is irrelevant in terms of requirements, given that it is only intended to host the test web-shop for SPHERE.IO, and both platforms promise the same level of quality. In spite of that, it is wise to choose the most likely option the future

developers will use, so that it is tested beforehand. While Cloudbees also offers integrated tools to support development of Java projects, Heroku is a much popular alternative with support for several programming languages and a wide range of plugins, thus becoming a preferable option for the project. Unlike SPHERE.IO, Heroku is scalable only under demand.

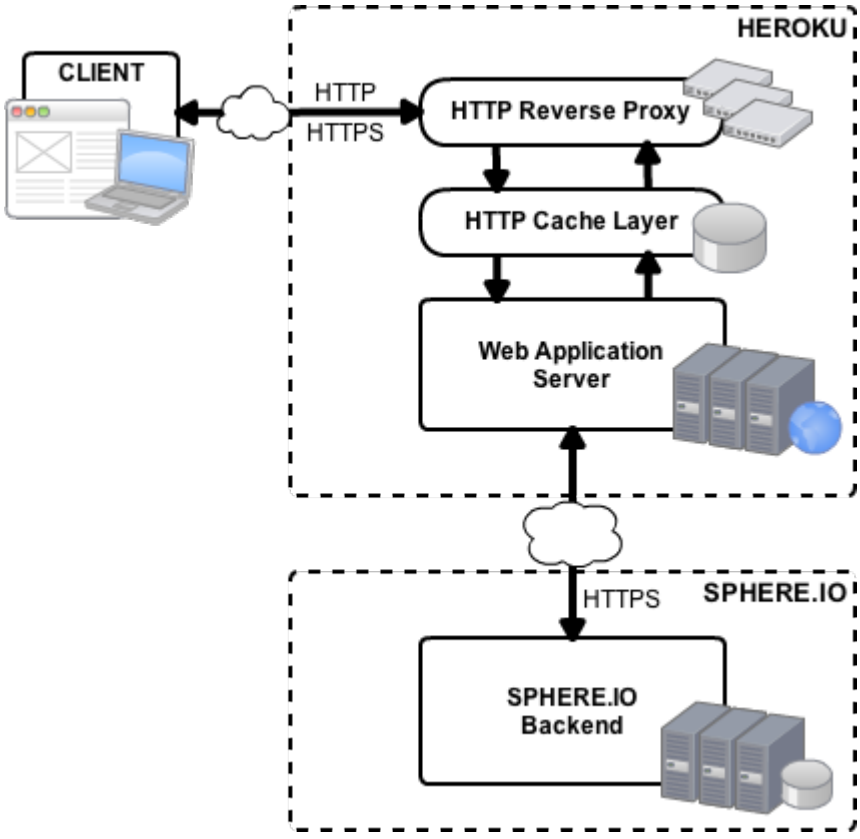


Figure 4.1: Diagram of the physical architecture of the system.

Figure 4.1 illustrates the physical architecture of the system. As appears in the diagram, any request to a Heroku deployed web application is first processed by one of the many platform's reverse proxies. The reverse proxy forwards the request to a HTTP cache layer, which returns the page if it is cached, otherwise forwards the request to the corresponding web application [Rob11].

The communication between the web application and the SPHERE.IO backend is always held with HTTPS as a requirement of the e-commerce platform. Instead, the protocol of the requests between the client and the web server are decision of the developer. For this project

the most reasonable option would be to use HTTPS whenever customer data is being transferred. This is typically the case of the checkout process, as well as any time the customer is logged in.

4.2 SYSTEM LOGICAL ARCHITECTURE

The logical architecture of the system is designed after the MVC (Model-View-Controller) architectural pattern, which is widely used in web applications design. Its use in this project is required, since MVC is the architecture pattern followed by Play Framework, the web framework on which SPHERE.IO Play SDK has been developed.

As the name suggests, the system logic is divided into three components: Model, View and Controller. As a rough definition, the Model manages business logic and domain data, the View is responsible of displaying the information, and the Controller is in charge of changing Model and View accordingly to the user input.

The specific MVC design of the current system is shown in Figure 4.2 below. One of the particularities of this design is that SPHERE.IO Play SDK is the main component of the Model, since it controls all the domain data of the application, as well as most of the business logic. Only some business rules are added to the Model in order to validate form input coming from the user, before sending this data to SPHERE.IO Play SDK, as well as some external functionalities such as email sending and online payment.

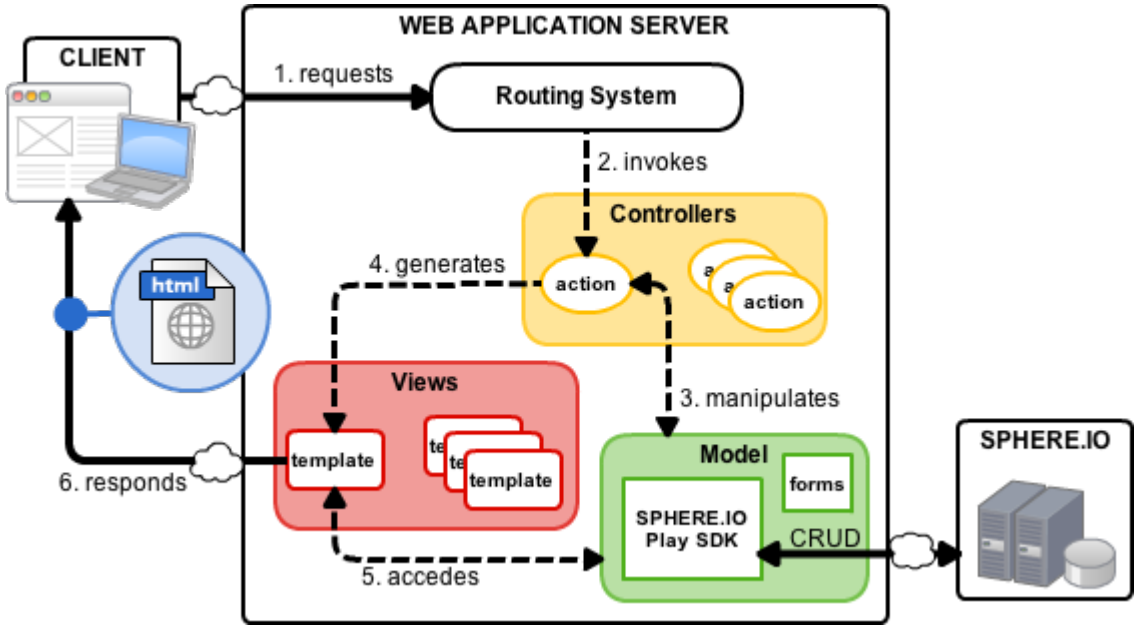


Figure 4.2: Diagram of the logical architecture of the system between contexts.

When the request reaches the web application server, a routing system analyzes the HTTP request and invokes a particular action of the corresponding controller. Then the controller interprets all required input parameters coming from the user and requests the appropriate changes to the model. In the model, SPHERE.IO Play SDK executes the request, which usually involves communication with the SPHERE.IO backend in order to create, read, update or delete (CRUD) some of the stored data.

Once the model finishes processing the request, the controller selects the appropriate template and sends all information related to the current request to the view. With this information and some other obtained directly from the model, the view generates a HTML document that is sent back to the client via a HTTP response.

With this design, a new whole web page must be loaded from the server every time the user wants to interact with the system. This is known as a “thin client” design, because all the logic is located in the server, leaving the client with the only task of rendering the web page. In comparison with that, a “fat client” hosts all the logic of the system; hence Controller, View and Model are located on the client side, leaving in the server just those parts of the Model responsible for the security and management of persistence.

A fat client allows the user to interact with the system while never reloading the web page, only updating those specific components of the page that changed during the interaction. This behavior enhances the user experience, because the user can continue interacting with the system while operations are taking place. Information can also be presented in an incremental way, so that the user can start interacting with some elements of the page while further information is being retrieved. Another important fact is that traffic between the client and the system is reduced to simple data with no presentation information, which speeds up the communication with the system and decreases network use.

While a fat client solves some external design issues, it also creates several technical problems. Since the web page is never reloading, the browser can no longer control the routing, caching or history management of it. Therefore it is the responsibility of the system to replace those functionalities that the browser is unable to perform.

These technical problems can be considered a too expensive price to pay in order to improve the user experience. The amount of resources needed to implement a reliable system with a pure fat client is several times higher than the equivalent with a thin client. Moreover the complexity of the code is also very significant, which makes this design not suitable for a template that must be understandable and easy to learn.

A mixed approach between a fat and a thin client can be the solution to improve the user experience without giving up on the browser logic. The website can be divided into different contexts that offer the user some common functionalities. Between contexts the web page is fully reloaded, while operations within the contexts only update some parts of the page [Con13]. By way of example, each product detail page is a different context, but adding a product to the shopping cart only updates the mini-cart displayed, while the user never leaves the page.

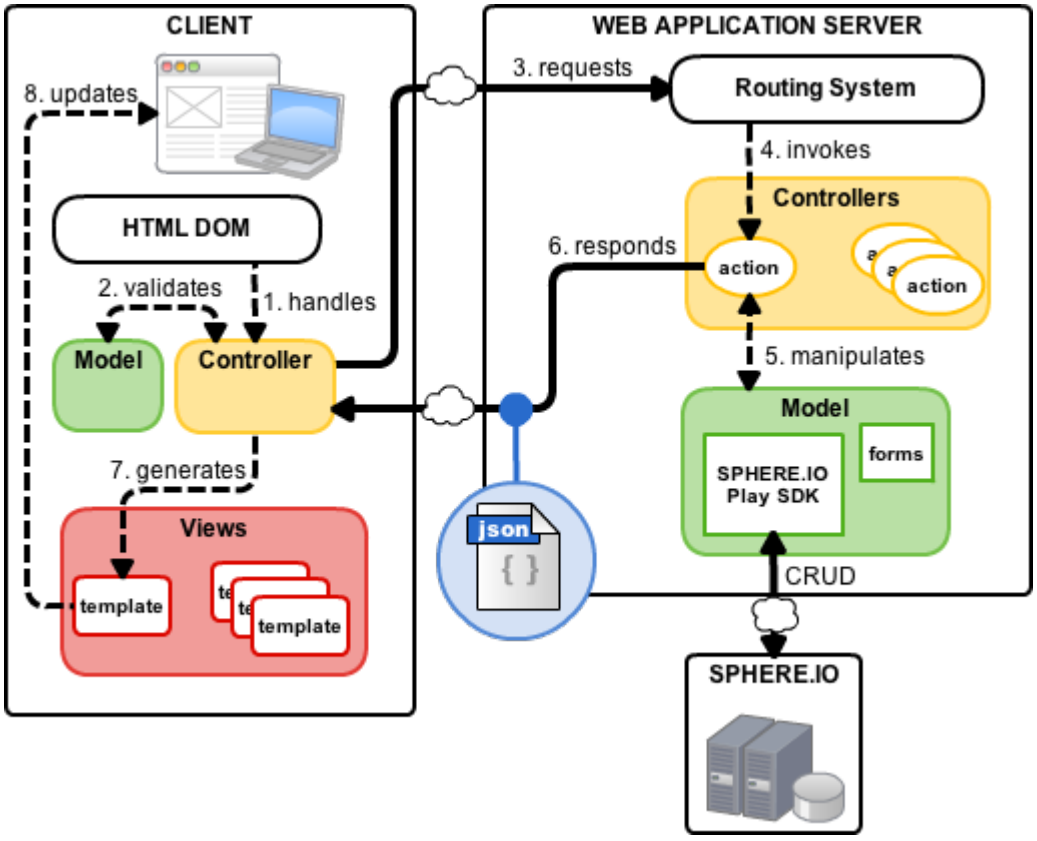


Figure 4.3: Diagram of the logical architecture of the system within a context.

In order to facilitate understanding of the logical architecture of the system, its design has been divided into two different diagrams: the one corresponding to the scenario between contexts and the one displaying the scenario within a context. The former has already been explained before, so the following explanation will focus on the differences and characteristics between both scenarios.

Every time a new context is loaded or the user interacts with the web page in some way, an event is fired by HTML DOM[9]. The controller on the client side can handle these events, in which case it gathers the required information and requests the client-side model to validate this information in order to avoid unnecessary calls to the server. If the validation was

[9] Stands for HTML Document Object Model and is an object model and programming interface for HTML, that defines all HTML elements as objects with associated properties, methods and events.

successful, the controller sends the corresponding HTTP request to the server, which is analyzed by the routing system and handed over to a controller action the same way as before.

As well as before the controller requests the appropriate changes to the model, but this time when the model finishes, the controller generates JSON data using the information related to the current request coming from the model. This JSON data is sent back to the controller located on the client, which in turn selects a template and sends this data to the view. With that, the view generates a HTML fragment that uses to replace the corresponding component on the web page.

4.2.1 DESCRIPTION OF USED TECHNOLOGIES

The current project has several technologies that are fixed by the requirements, starting with SPHERE.IO Play SDK. This SDK is designed to be used with Play Framework, and specifically with the Java language version. Besides the framework has a significant influence on several other server-side technologies as well, depending on the support it provides. On the other hand, the payment platform needs to be carefully chosen, because it has inevitably a great impact on the template reusability and the analysis of the platform.

All client-side technologies need to be selected, specially the templating solution in the view component. Furthermore, given that maximizing developer experience (i.e. user experience applied to developers) is one of the main requirements of the project, this system needs technologies to help organizing and simplifying the code, particularly complex because of the logical architecture design.

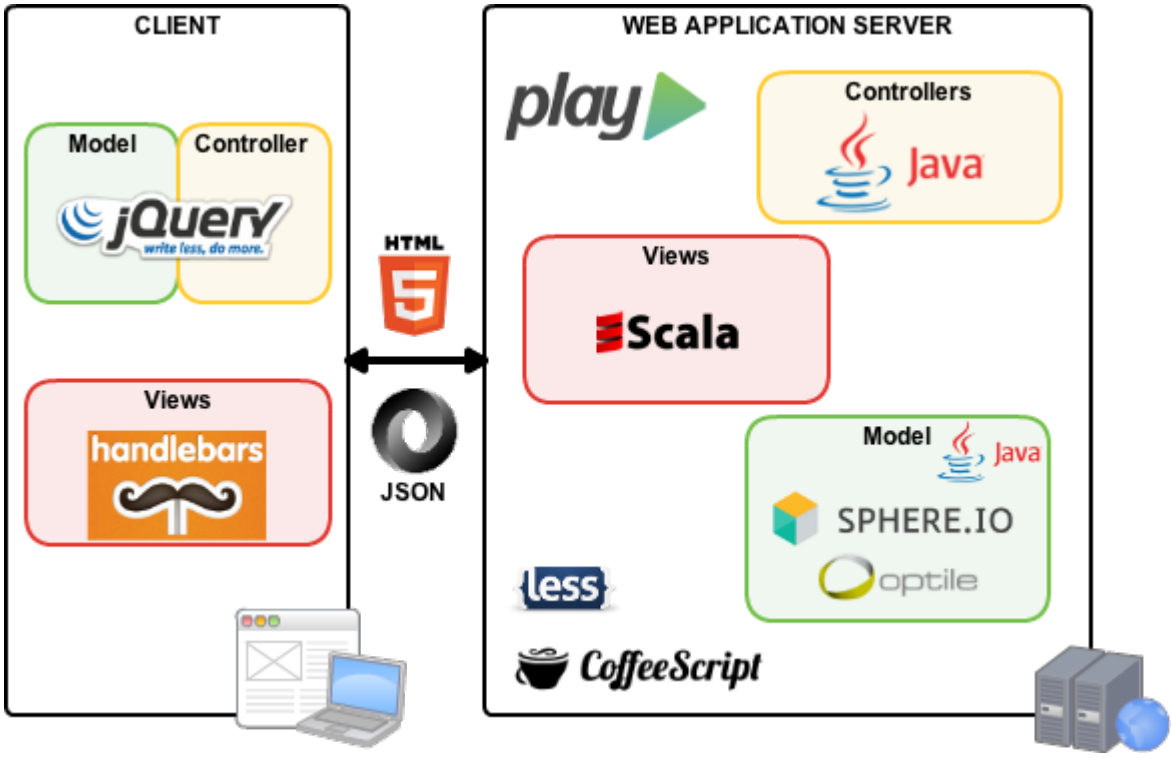


Figure 4.4: Diagram of the technologies used in each logical component.

Figure 4.4 above illustrates the use of technologies in each component. As it shows, Play is the web application framework, that uses the programming language Scala in the templates, and Java in both model and controllers. In the model SPHERE.IO provides the main commerce business logic of the system, while Optile supports the payment functionality. Additionally, LESS and CoffeeScript are used server-side to generate CSS and JavaScript files, respectively.

The server is using HTML5 and JSON files to send information to the client. The logic of the client side is supported by jQuery and the templating system is implemented with Handlebars. A description of each chosen technology and the characteristics that influenced the decision-making are detailed below.

4.2.1.1 SPHERE.IO

Figure 4.5: SPHERE.IO official logo.

SPHERE.IO is a cloud-based commerce platform, aimed to unify e-commerce data in a single place where any kind of external system can access this information. These external systems are typically web-shops but can actually be any type of application, even those not related to e-commerce. SPHERE.IO provides a platform to store and process all this data according to commerce business rules, while at the same time offers several ways to access it.

The primary entry point to the backend is provided by a RESTful API, that offers an interface for programmatic access to the data and associated functionality. The API services are using JSON to communicate, always over HTTPS, with previous authorization using OAuth2[10]. Although it is actually the core of the platform, its direct use might be tedious for slightly complex applications. That is the reason why it is recommended to use client libraries and SDKs to communicate with the API, and so improving the development experience.

The SPHERE.IO team chose Java as the first programming language to have a client library due to its versatility. This library is open source, as it is intended to be improved or used as a reference to build other libraries by the developer community. In order to provide a better environment to build websites, a SDK was built on top of the client library: the SPHERE.IO Play SDK. It allowed to adapt the Java client library to the processes and structure of the Play Framework.

A command-line interface (CLI) is also available, especially aimed for managing SPHERE.IO user accounts and projects from a command-line shell. It is also necessary to use the CLI in order to manipulate and query data in batches or for automated jobs, such as importing

[10] OAuth 2.0 is a protocol for authentication and authorization.

products into SPHERE.IO. As opposed to the API, the CLI is not using OAuth2 since all operations are done under a user account.

So far all the tools for accessing and managing the backend data were focused on developers, but merchants have also the possibility to view and update the data using a web application called Merchant Center. Besides that, merchants can also export and import data between SPHERE.IO and other external systems using elastic.io[11] as an integration platform.

4.2.1.2 Optile

Figure 4.6: Optile official logo.

Optile is a payment platform that allows to access a set of heterogeneous payment methods and providers (e.g. credit cards, direct debit, PayPal) under a common interface. Once the web-shop has the platform integrated, the set of payment options can be extended or reduced without any extra implementation effort. Optile has five different main levels of integration: redirected, hosted, half-native and native with and without PCI. These are implemented one on top of the other without losing the previous implementation, that way one can go back to lower levels very easily.

The first level is the universal redirect, where the customer is completely redirected to the payment platform and there he enters his payment data. In the second level that form is hosted in the system via a HTML frame or JavaScript. With the third level the system is in charge of querying the platform about the payment options and display them, but once the user selects one he is redirected to the platform to provide the payment data. The fourth level displays both payment options and forms, taking care of submitting the forms to the platform.

[11] A cloud-to-cloud integration platform that allows to connect data between popular services and tools.

The last level requires the system to be PCI compliant, because it gathers the payment data and queries the platform to charge the customer with the provided data.

In opposite to traditional payment platforms, the successful integration of Optile will attest that this web-shop supports a wide range of payment methods and providers, as well as multiple different ways of integration. It is also a good choice for developers, who will have a system with several online payment methods already implemented. Optile first integration can become a little bit tedious, but its flexibility will be profitable for this project.

A valid alternative is Paymill, a popular payment solution which characteristics are completely opposed to Optile: the integration is very fast and easy, but the payment providers offered are limited to credit card and direct debit. Also the customer is never redirected to the payment server, yet there is no need to be PCI compliant. The reason is that the payment form is never submitted, but its data is sent to the payment server via a JavaScript library, returning a token in exchange that the system can use to charge the customer from the server side. Nevertheless the selected solution is Optile, because its implementation will benefit more the project than Paymill.

4.2.1.3 Play Framework

Figure 4.7: Play Framework official logo.

The use of Play Framework comes as a requirement to test the suitability of SPHERE.IO Play SDK, which was build to create web-shops using this specific framework. Play is an open source web application framework that was first released in 2007 and written in Java. In 2012

a second release was announced, with a core completely rewritten in Scala[12]. This is precisely the version that SPHERE.IO Play SDK works with.

This second version of Play uses Scala in its web template system. Projects in Play are built and deployed with SBT, a build tool for Scala and Java projects, allowing developers to choose between these two programming languages in order to implement the logic of their web applications. Despite this, currently SPHERE.IO Play SDK is supported only in Java projects.

Play follows the MVC logical architectural pattern and is completely RESTful, which means amongst other things that is stateless, unlike other Java frameworks. It was also designed to support full asynchronous HTTP programming, to serve long-lived requests without tying up other threads. Play also includes the Jackson library to manipulate JSON data and native support for the software testing frameworks JUnit and Selenium. Moreover it also has a compiler for CoffeeScript and LESS, two programming languages that compile into JavaScript and CSS respectively.

4.2.1.4 CoffeeScript

Figure 4.8: CoffeeScript official logo.

CoffeeScript is a programming language that compiles into JavaScript, adding syntactic sugar[13] to greatly improve the developer experience. The new syntax provides a better readability of the code and helps developers to write complex scripts much more easily. The increased readability of the code goes along with a decreased number of lines compared to the same code in JavaScript, around one third fewer lines. Another interesting feature is an additional syntax to use JavaScript's native prototyping as a class-based system, making

[12] Scala is a object-functional programming language that runs on JVM and is compatible with Java scripts.

[13] Syntactic sugar refers to those syntactic elements introduced to make things easier to read or express.

object-oriented programming with JavaScript less complex, particularly when it comes to inheritance.

Improving developer experience is a priority in this project, so CoffeeScript will contribute to make client-side code easier to understand and modify. It will also be considerably helpful with the development of the JavaScript code, which is pretty complex due to the logical design of the system. Therefore its use is very appropriate, especially since a CoffeeScript compiler comes included in Play Framework.

4.2.1.5 LESS CSS

Figure 4.9: LESS CSS official logo.

Similarly to CoffeeScript, LESS is a language that compiles into CSS. But unlike CoffeeScript, LESS does not modify the syntax of CSS, but only extends it with dynamic behavior, such as variables, operations and functions. This makes LESS very easy to learn and converts a simple CSS-based file into a powerful dynamic stylesheet.

LESS will allow to better organize the stylesheet of the web-shop, thus facilitating a swift development, fast edition and easy understanding of its code. Although there are other CSS preprocessors like the popular Sass, the provided functionalities are quite similar and Play Framework already comes with a native support of the LESS compiler.

4.2.1.6 jQuery

Figure 4.10: jQuery official logo.

jQuery is a very powerful and fast JavaScript library that allows to easily do DOM scripting (i.e. HTML elements manipulation and event handling), perform animations and simplify the use of AJAX[14] programming; altogether very necessary in this project. The main alternatives, such as MooTools or YUI Library, are also very satisfactory solutions in the mentioned areas, with no significant differences. The final choice of jQuery has been mainly determined by the fact that it has the largest community amongst the options.

4.2.1.7 Handlebars.js

Figure 4.11: handlebars.js logo.

Handlebars is the web template system used in this project to manage client-side templates. It is a JavaScript implementation of the platform-independent Mustache project, that allows to render input data in a template using a very clean syntax. Mustache has a so-called logic-less

[14] AJAX is a technique used on the client side of a system to perform asynchronous operations, such as communicating with a server, without keeping the client idle.

template syntax because there are no explicit control flow statements, all needed logic comes exclusively from the data in the form of booleans, arrays or lambdas[15].

On the contrary Handlebars templates are compiled, allowing to define helpers to reuse code for presentation. It also comes with built-in helpers to control the default flow of the template, such as loops or conditional statements. Handlebars comes also with better support for paths to access the data. In short, this solution makes easier to implement templates than Mustache while still keeping logic separated from presentation.

There is another project, Dust.js, with the same strong points as Handlebars and with useful additional features like template composition. Apparently is a better choice but the project has been abandoned for two years, maybe the reason why Handlebars has the largest community. During the last year LinkedIn has been contributing actively to a separated Dust.js project that the company is using for its website [Bas12]. Regardless it has been considered that Handlebars is a safer option, since the additional features are not indispensable for this project.

4.2.2 EXTERNAL DESIGN

The external design of a system requires the developer team to work closely with a designer team. While the designers are in charge of creating the interface and aesthetic design, the other external design areas of the system need a more technical approach. These areas consist of the navigation and architecture design, as well as some parts of the content design. All these aspects are covered by the User Experience (UX) Model explained here. It describes, in particular, how the dynamic content will be structured and organized in different screens, and how the user will navigate amongst those screens to reach a particular goal.

For this project, the designer team together with the developer team decided to implement a layout structure and behavior inspired on different web-shops with innovative design, always focusing on offering the user a smooth interaction with the system. Each screen is going to be presented and described with wireframe prototypes, and then the storyboard sequence and navigational paths will illustrate the connections between those screens.

---

[15] Lambda is an anonymous function, meaning that it is not bound to any kind of identifier.

4.2.2.1 Display products

Product listing is the first functionality that a customer uses when arriving at the web-shop and the one he will be using for longer periods of time, so it needs to have a comfortable way to display and paginate the products. At best, traditional web-shops usually have very rigid ways of listing products: pagination consists of an interface that allows to select the page and the amount of products per page, while display options let the user select between a list or a grid type of view.

So instead of showing a traditional shop catalog, it was considered a better option to let the products flow freely through the web page, using all the width and height possible to show at once the maximum amount of products to the user (Figure 4.12). On the other hand, the pagination needs to be natural without losing already viewed products, so when the user reaches the bottom of the page new products should appear automatically under the previous ones.

The product thumbnails, besides price and name, will be showing a picture of the product and the different color variants. The selected variant will be highlighted, and when hovering a different color the thumbnail will be updated with that variant information, such as picture and price, if different. The thumbnail will also include a button to add the selected product variant to the shopping cart. In case the product has different sizes available, when hovering the button a list of the different sizes will be shown, so that the user can select the desired size he wants to add to the cart.

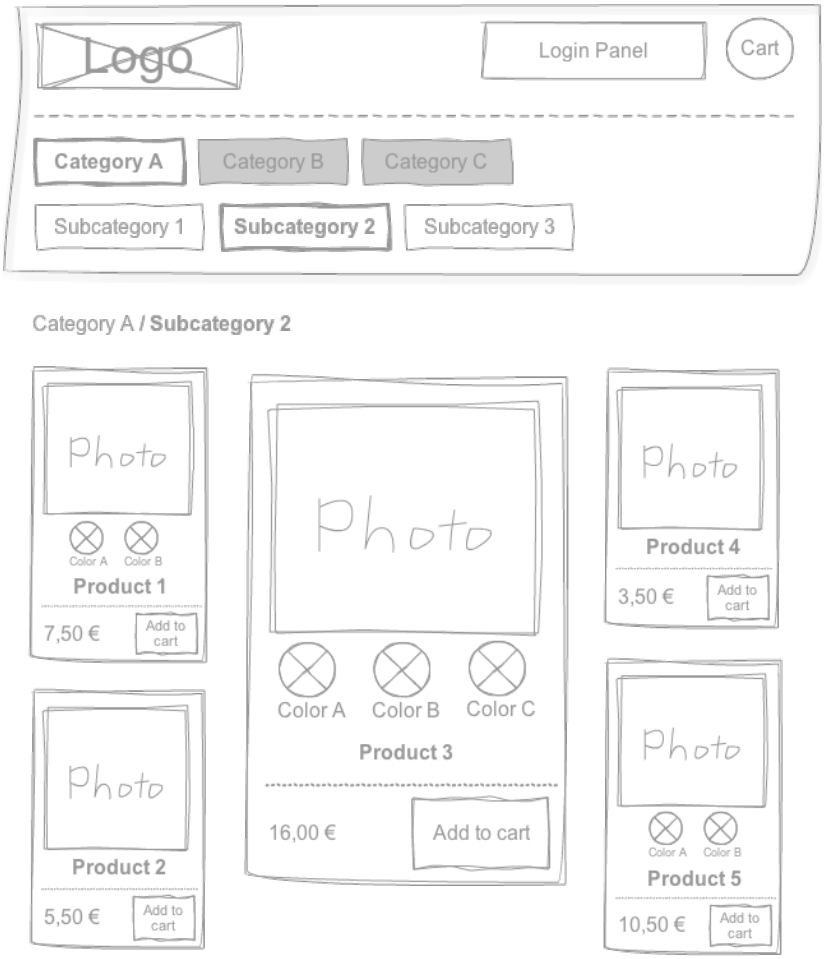


Figure 4.12: Wireframe prototype of the product listing screen, filtering by category.

When clicking on a product thumbnail the user will be redirected to the product detail of the variant he had selected (Figure 4.13), if any. There he can select any other color variant, in which case a new page will be loaded in order to update the URL, to let the user share the product URL that points to this particular color. He can also select a different size, but in this case the page is not reloading, as it was considered that the user does not have a need to share the exact size. Below one can add the selected product variant to the cart, optionally indicating the exact quantity.

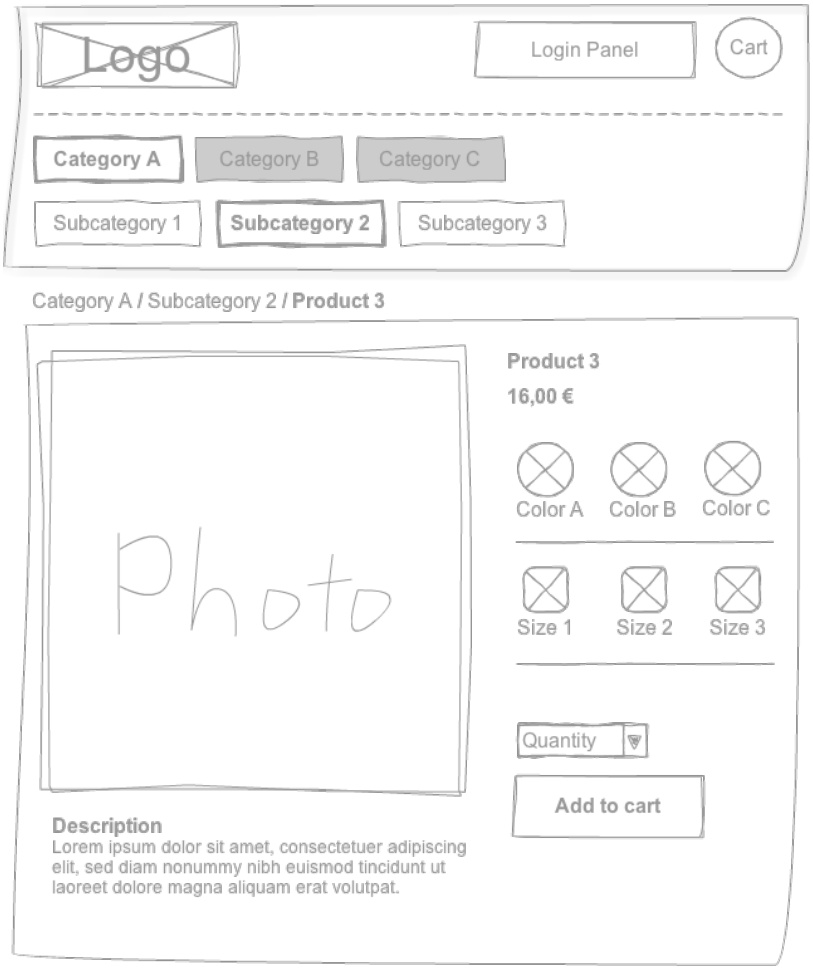


Figure 4.13: Wireframe prototype of the product detail screen.

The header contains a mini-cart and the login panel throughout the website. In any product list or product detail page, the header also contains the categories and subcategories of the shop to let the user filter products by category. The rest of the pages should contain a button to allow the user go back to the last category or product he visited. When scrolling, the header is always kept at the top of the page. Below the header, a breadcrumb is showing the current category path.

Whenever a product is added to the cart, the mini-cart located on the header appears for a few seconds, to let the customer know that the product was added successfully. At any time the user can see again the contents of his shopping cart when hovering the cart button on the header, that will be closed automatically when moving the cursor away from the mini-cart.

Below, Figure 4.14 and Figure 4.15 presents the storyboard sequence and the navigational paths, respectively, of the screens just described. In the storyboard sequence it is shown how the same interactions defined in the sequence diagrams of the specification are performed through the different screens. This connections are also captured in the following navigational paths diagram.

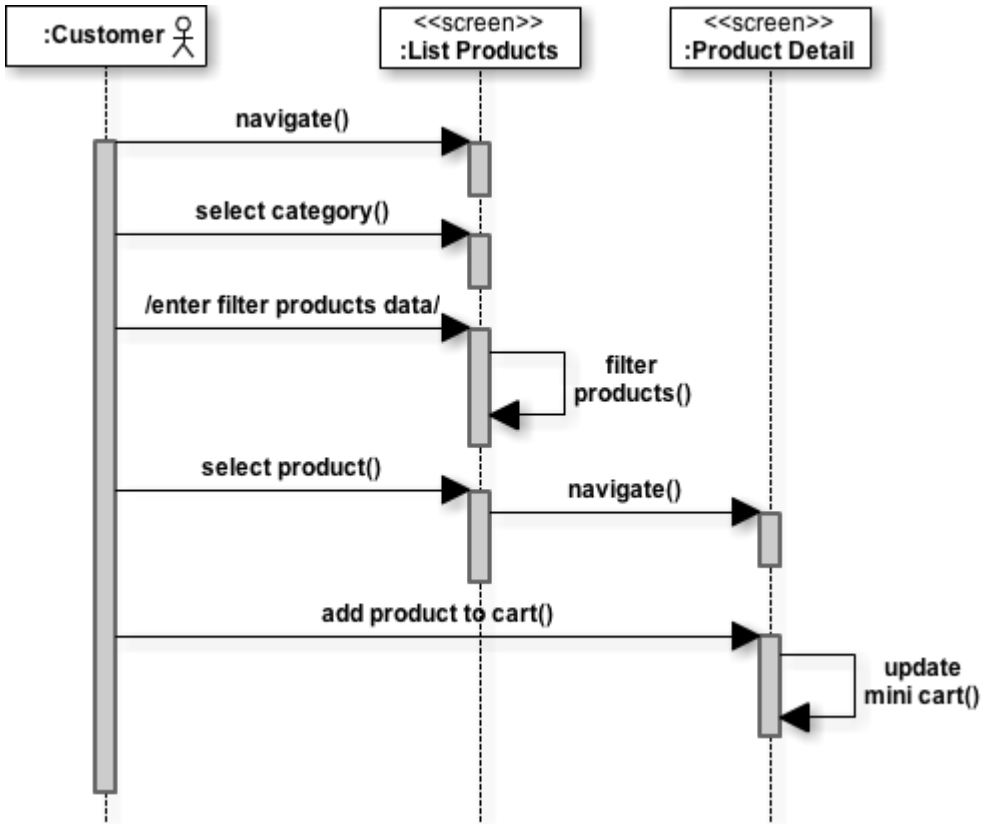


Figure 4.14: Storyboard sequence of the *browse catalog* top-level use case.

The resulting screens and navigational path diagram, besides the connections between screens, shows the detailed content of each screen. The screen for listing products displays the selected filtering options and contains a single form to change them. It also has the name of the current category and the list of matching products, with their different variants for color and size. This screen has an option to add a product to the cart, while the product detail page has a form to let the user specify the quantity of items he wants to add. Both screens renders the category tree so that the customer can list all products from a particular category at any moment.

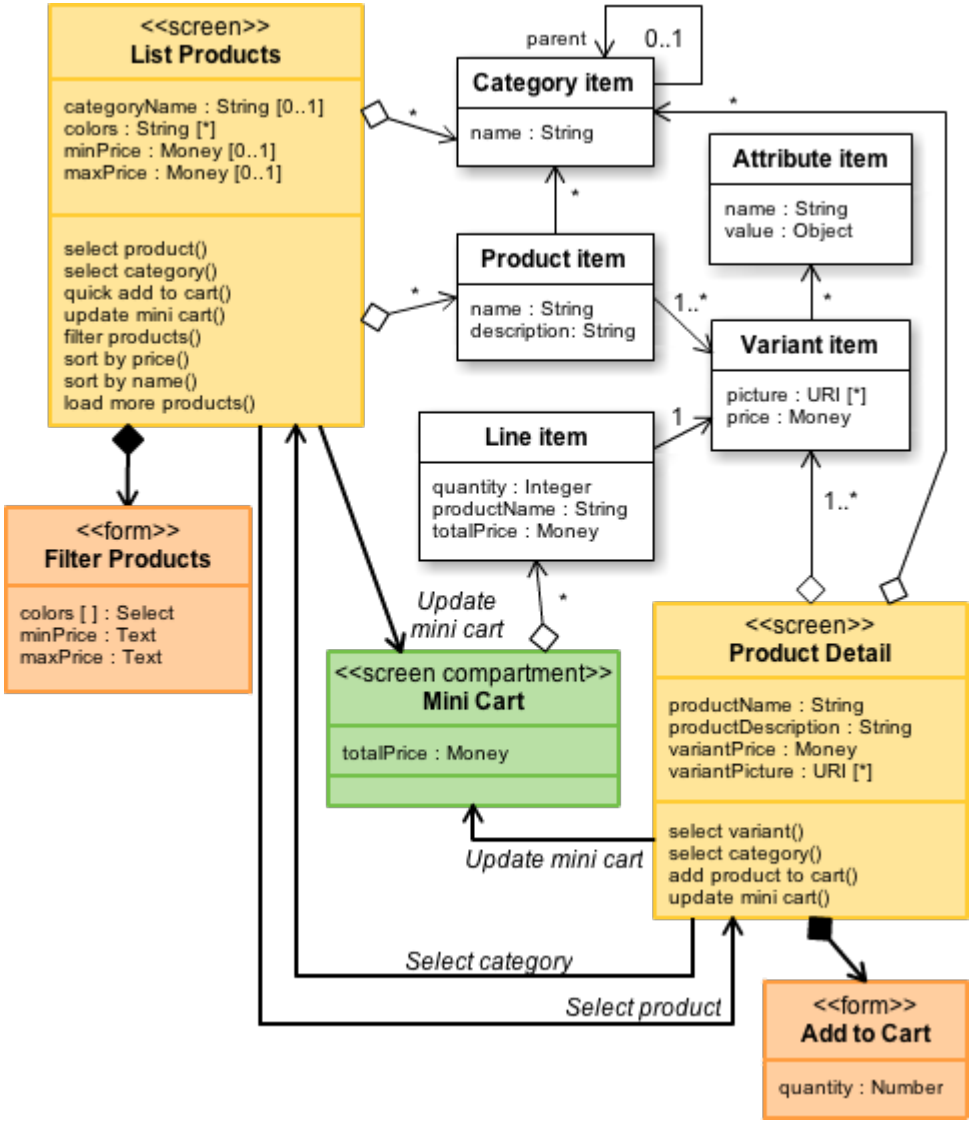


Figure 4.15: Screens and navigational paths of the *browse catalog* top-level use case.

4.2.2.2 Purchase products

In order to start the checkout process, the user will first access the cart detail page by clicking on the cart button. This page shows the items and their details, along with the possibility to remove them or change the number of units of each item (Figure 4.16). Both actions are performed without reloading the page, just updating the contents of the shopping cart and the pricing details accordingly.

Figure 4.16: Wireframe prototype of the shopping cart detail screen.

The checkout page can be accessed from both mini-cart and cart detail page. The checkout page is probably one of the least frequented pages of a web-shop, but it is for sure the most important when it comes to user experience. The customer needs to feel he has control of the flow and that he is able to quit at any time. The checkout needs to be a secure and robust environment to the user.

Traditional web-shops usually reload when moving from one checkout step to the other, and it can be sometimes difficult to change the data of a step that is not immediately before the active one. In some cases it is also hard to know what changes are modifying the price or to review what was entered on previous steps. All these issues are affecting negatively the feeling of control the user has.

Figure 4.17: Wireframe prototype of the checkout screen, shipping section.

For this design it was considered a good idea to display all the steps throughout the page as sections that can be expanded, so that the user modifies them (see Figure 4.17). Once edited, the section closes again and displays a summary with the selected options. Every change automatically updates the pricing details that are always shown at the bottom of the page. As a way of guiding the customer through the checkout process, the user can only open new sections sequentially. Also when a form is still not available due to missing requirements (e.g. shipping method cannot be displayed until shipping address is set) a message will be shown instead until the requirements are met.

The checkout is divided into three steps: first a cart summary, to verify the items are correct; second the shipping information, to determine where and how the goods are being delivered; and third the billing information, to select the way the products are being paid. Both shipping

and billing sections have on the left side a form to set the postal address and on the right side the shipping and payment options, respectively. When the customer is logged in, his address book will appear on the right side, allowing him to select one of his addresses, which data will then be copied to the corresponding address form.

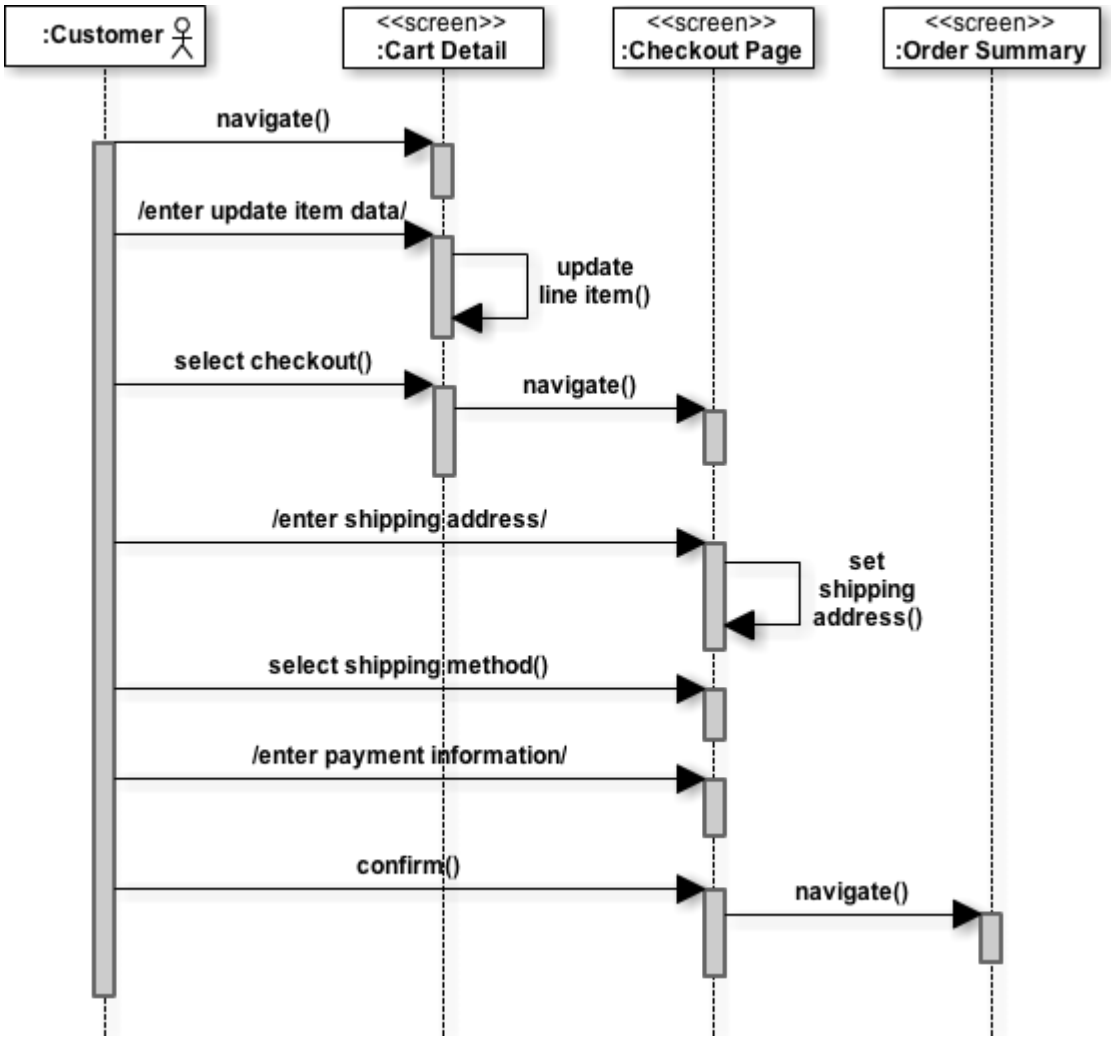


Figure 4.18: Storyboard sequence of the *checkout* top-level use case.

The storyboard sequence from Figure 4.18 is showing how this checkout process is distributed with the presented screens. Below in Figure 4.19, the screens and navigational paths diagram expose that all screens have almost the same data, such as a set of line items and some pricing details, while the checkout page and the order summary have also a billing and shipping

address, all them with different multiplicities. The cart has forms to update line items, as many as line items are in the cart. The checkout is also composed of different forms, one for each element that the customer must fill, to allow saving each element individually.

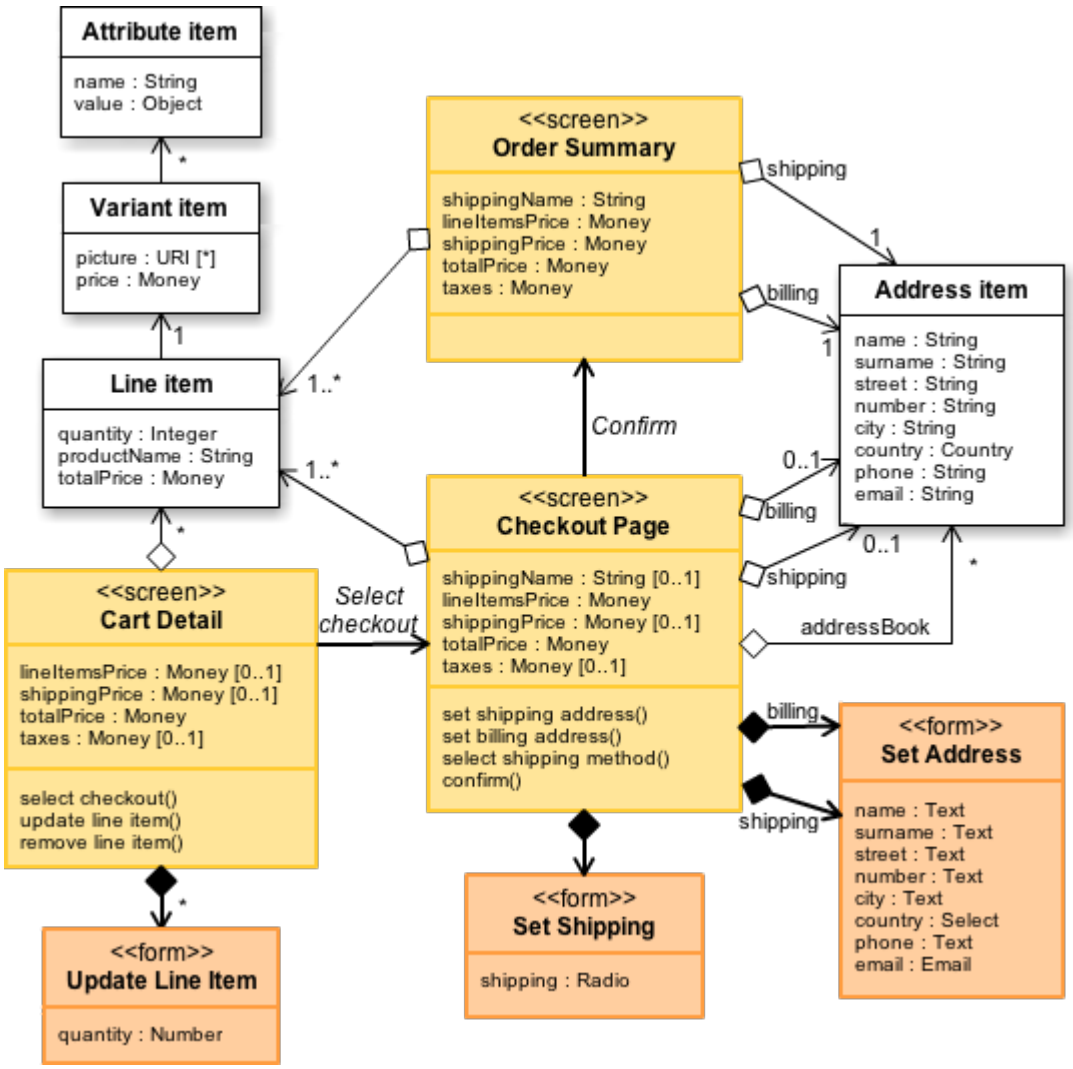


Figure 4.19: Screens and navigational paths of the *checkout* top-level use case.

4.2.2.3 User management

Before attempting to access his profile page, the user needs to identify himself to the system. This is done in the login screen, a page that also contains a form to register into the system (see Figure 4.20). In case the user forgot his password, the login form contains an option to

recover it, which renders a modal window where an email address is requested when the option is clicked.

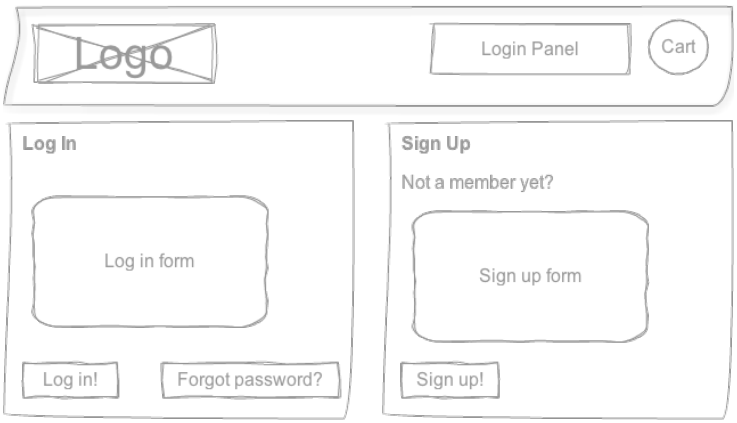


Figure 4.20: Wireframe prototype of the login screen.

Submitting this form will send an email to the user with a new URL, that redirects to the same login page but with a different modal window to enter a new password. Once the password is submitted the modal window closes, thus showing the login form again to allow the user enter his new credentials.

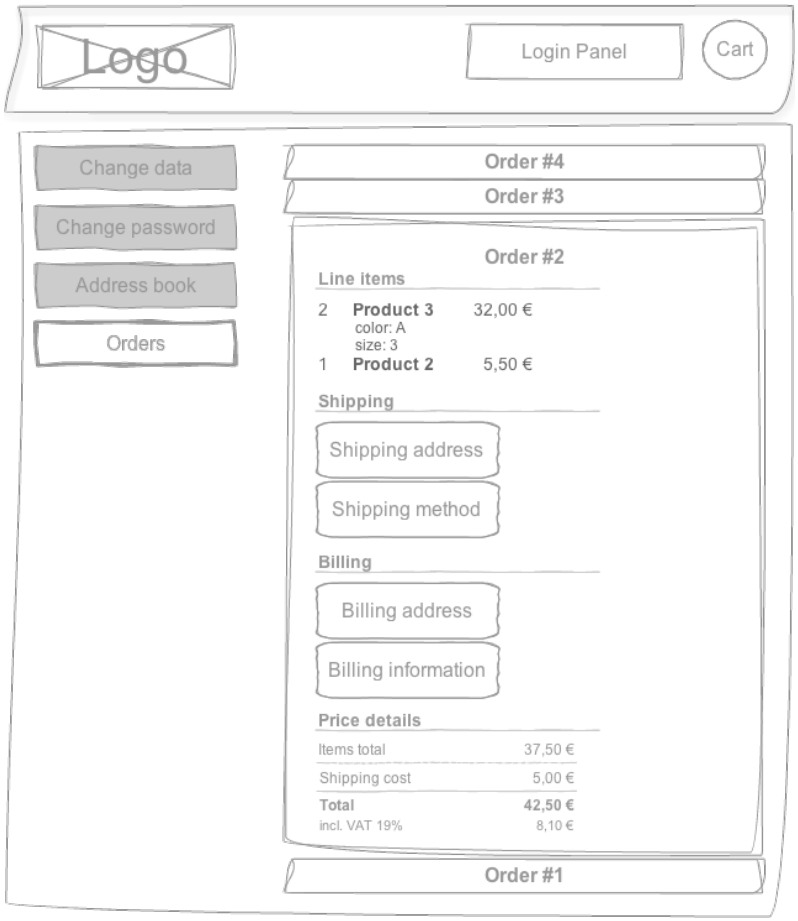


Figure 4.21: Wireframe prototype of the user profile screen, order list section.

The user profile is a single page with sections to change user data, password, manage the address book and view the list of orders (see Figure 4.21). The latter consists of some stockable sections, each one containing all information about a particular order, such as the products purchased, the price details and all shipping and billing related information. When clicking on a section, this one expands showing its contents, while all other sections remain closed.

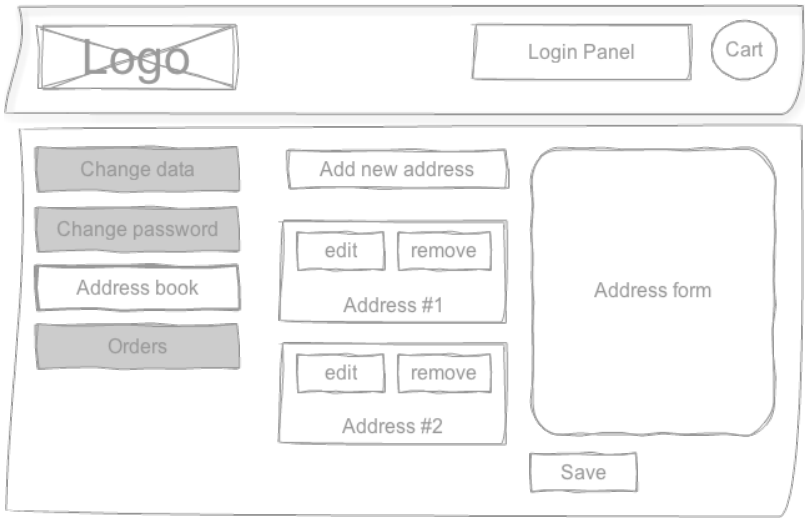


Figure 4.22: Wireframe prototype of the user profile screen, address book section.

The address book is the only section with a slightly complex design. This component has a list of existing addresses on the left and an empty form on the right to add a new address (Figure 4.22). When the user selects an address the form changes into edition mode, highlighting the address and copying its data to the empty form. A button at the top allows the user to return the form to its initial mode. Whenever the user adds, updates or removes an address, the list of addresses is updated accordingly.

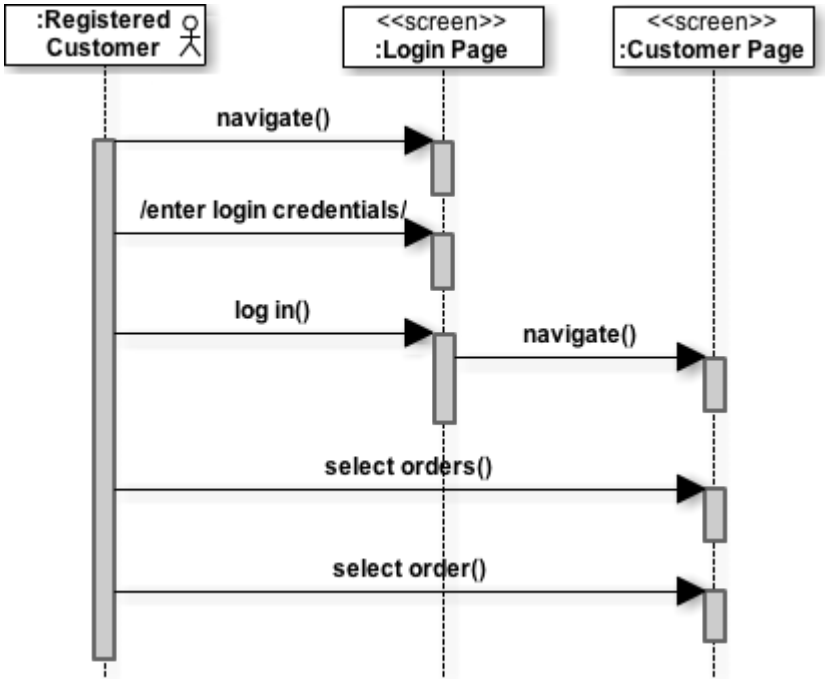


Figure 4.23: Storyboard sequence of the *check order* top-level use case.

Again, the specification sequence diagram is adapted to the given design, illustrating the different screens that participate in every interaction in the *check order* top-level use case (see Figure 4.23). Figure 4.24 below shows the screens and navigational paths diagram, displaying every screen component that belong to the customer profile screen, each with its own data and forms.

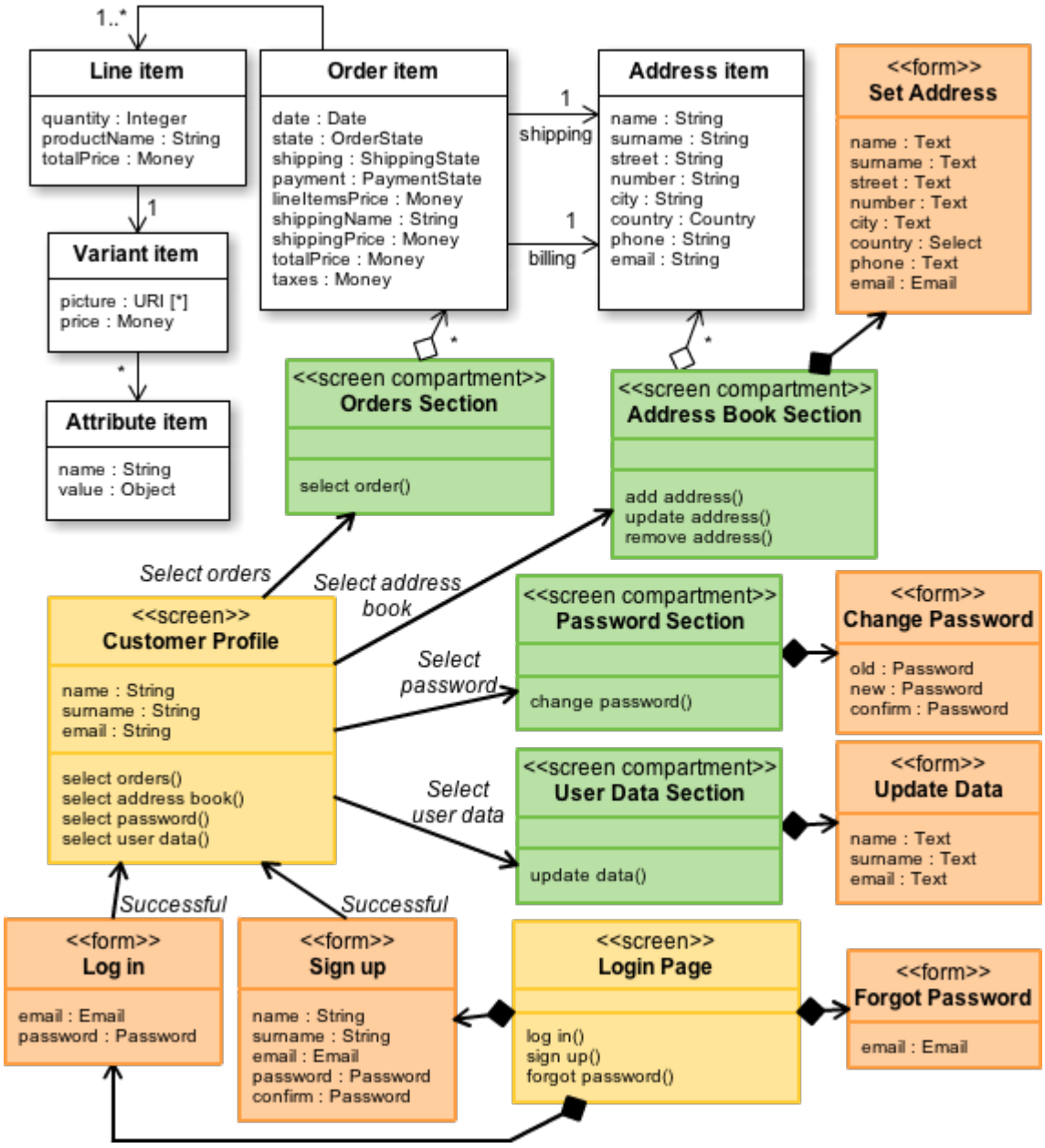


Figure 4.24: Screens and navigational paths of the *check order* top-level use case.

4.2.3 *Internal design*

After representing the external design of the system, every diagram of the UX Model needs to be adapted to the chosen technologies. An internal class and sequence diagram are resulting from this process, showing the communication between each component. These diagrams are

done with the Web Application Extension (WAE[16]) to UML, which allows to represent every significant element of the web presentation layer with the UML modeling language.

In this project, the new class stereotypes of WAE enabled to represent server-side pages, client-side pages, forms and script libraries; every class with its own attributes and methods. There are also new association stereotypes to represent the different ways of communication between elements. These can be, amongst others, a common link with parameters (link), a form data submission (submit), a forwarding of the request to another element (forward), a generation of HTML output (build) or an import of a JavaScript file (script).

Internal design diagrams can become quite complex when trying to represent all files that participate in a use case, particularly when applying some design patterns. For this reason, in this section only those diagrams that illustrate some special behavior or structure are displayed, simplifying any characteristic that is later described in the following sections, such as design patterns that apply to all use cases.

The internal class diagram in Figure 4.25 represents the whole *browse catalog* top-level use case. It is worth noticing that usually a client page links first to a server page, which then forwards to a server page with a "scala" extension. The first server page symbolizes a file in the Controller component, while the second corresponds to a Scala template from the View, which builds the client page.

An exception to this rule appears within contexts, when the client page asynchronously makes a request to the server. The response is simple data, so no HTML output is built, and it is the client page itself which updates its content with the information sent via forward parameters. The methods that allow to update the information from the client page are always coming from the JavaScript files, but generally are represented as methods from the client page. Only exception appears when the same method is shared between two pages, in which case it is left in the script for simplicity [Oli10].

[16] See more information in [Con03].

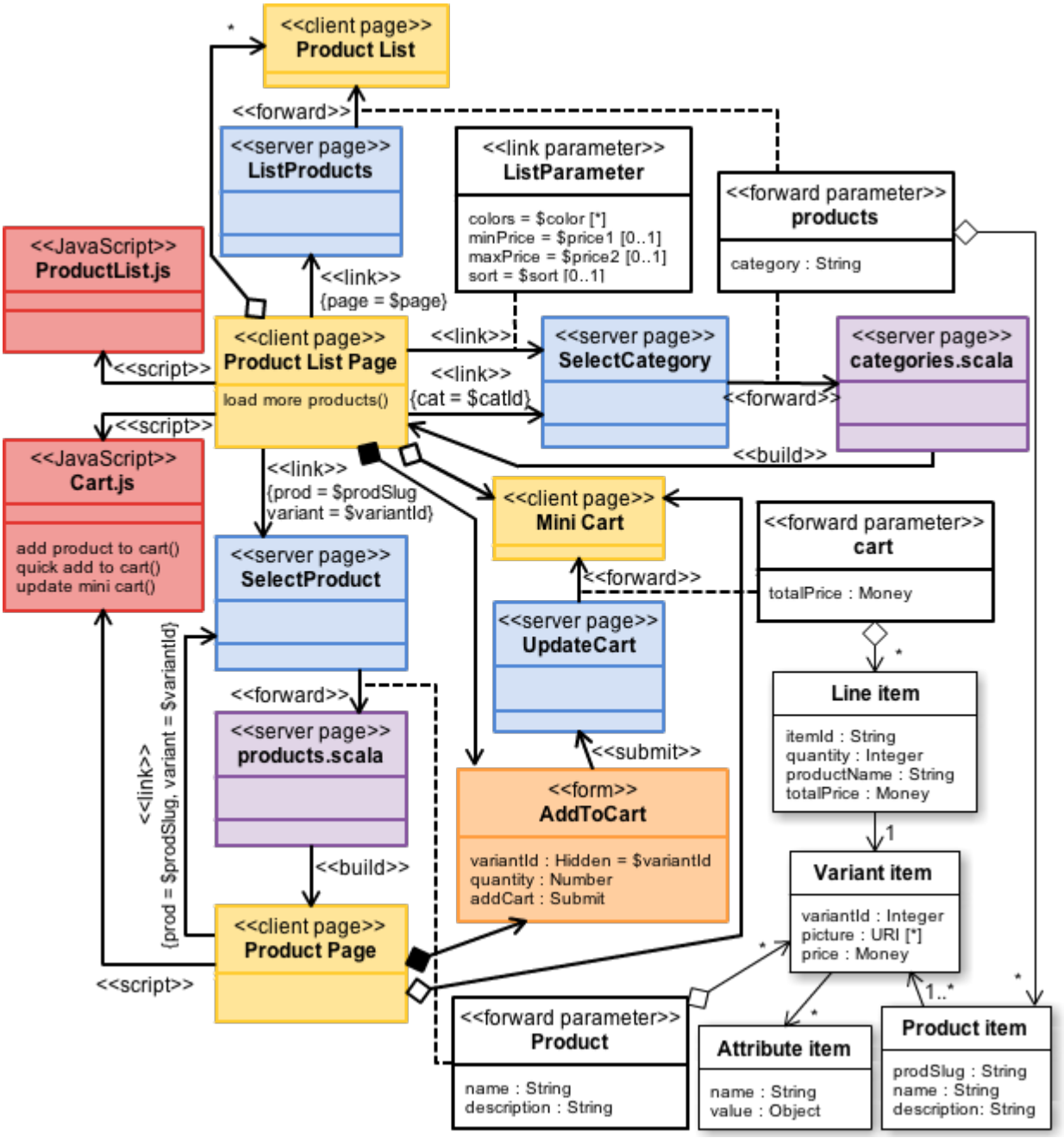


Figure 4.25: Internal design class diagram of the *browse catalog* top-level use case.

So an example of this particular behavior can be observed when adding a product to the cart from a product page or a product list page. The server page that updates the cart sends to the mini cart component all the information related to the shopping cart contents. With this data,

the JavaScript in charge of the cart updates only the mini cart component with the new information.

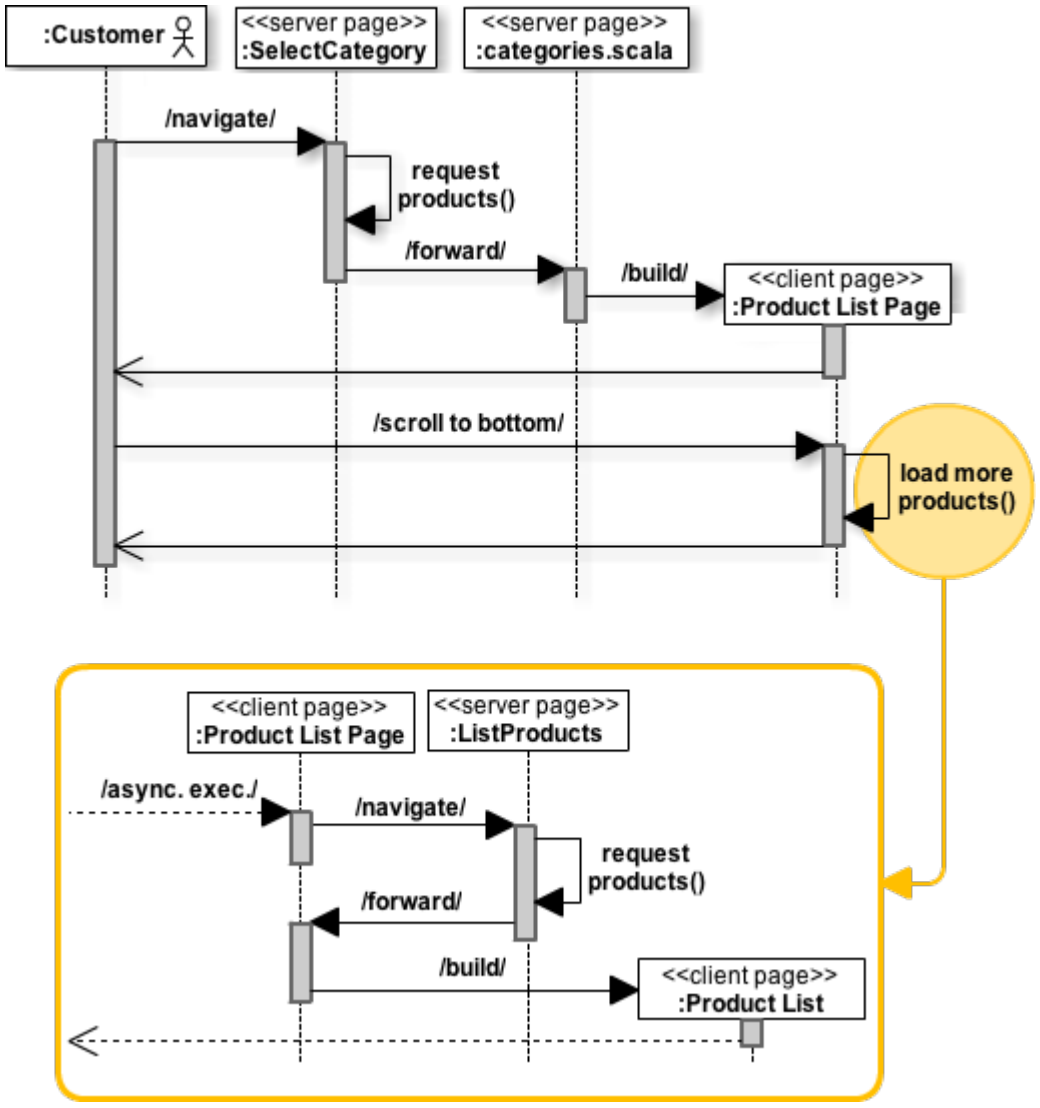


Figure 4.26: Internal design sequence diagram of the *pagination* use case.

Another example is the functionality to load more products when scrolling to the bottom of the page. In this case also the corresponding sequence diagram is presented (Figure 4.26) to observe in detail how the components are behaving. First, the customer navigates to the *SelectCategory* server page, which requests to SPHERE.IO the list of products for a particular

category. This server page forwards all necessary information to the corresponding view, which creates the entire client page and sends it back to the customer.

Then when the customer scrolls to the bottom of the page, the *load more products* method is called from the client page, which starts an asynchronous request to the server while the control is given back to the customer. In this request the script itself communicates with the server page *ListProducts*, which again makes a request to the model, this time asking for the next page. The resulting data is sent to the client page, that renders and appends a new *Product List* component with the received products.

Figure 4.27 below is the internal design class diagram for updating and removing line items from the cart. The update functionality requires a form where the customer specifies the new quantity, while removing is just a direct link. Both actions end up in the server page for updating the cart, which forwards the new cart information to the cart page.

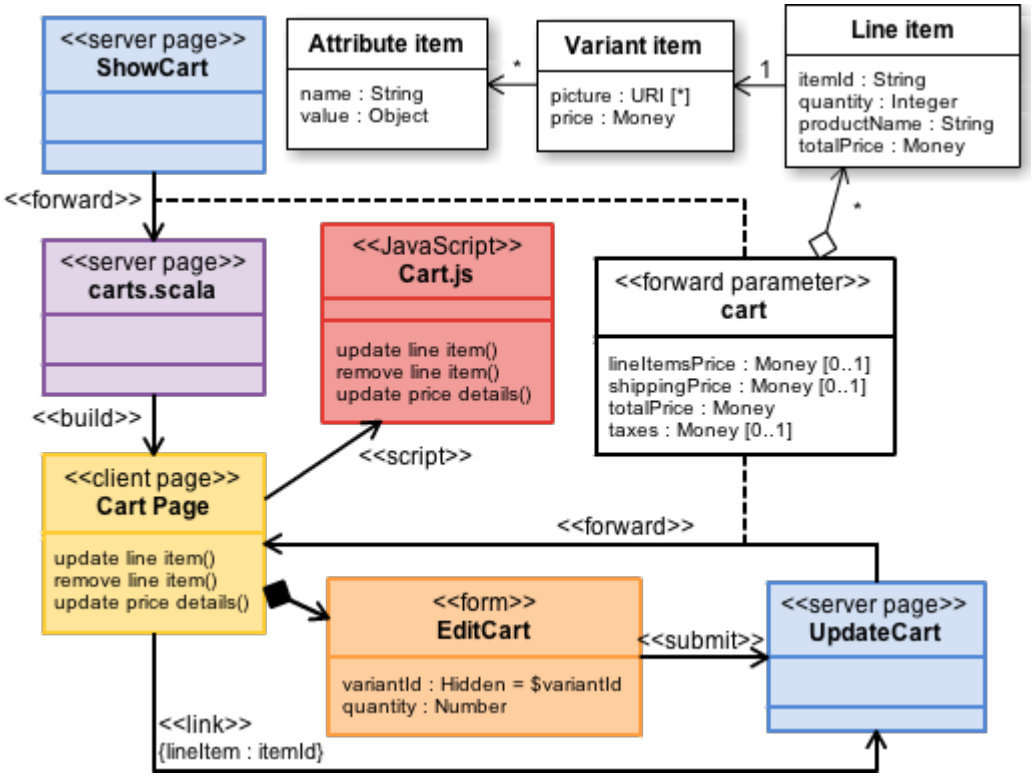


Figure 4.27: Internal design class diagram of the *checkout* top-level use case, detail of the cart update.

Figure 4.28 below presents the internal sequence diagram corresponding to the *update item in cart* use case, although the diagram corresponding to the deletion is very similar. The initial process to access the client page is the same as any other use case: the customer navigates to the server page, which forwards the information to the template, which builds the client page.

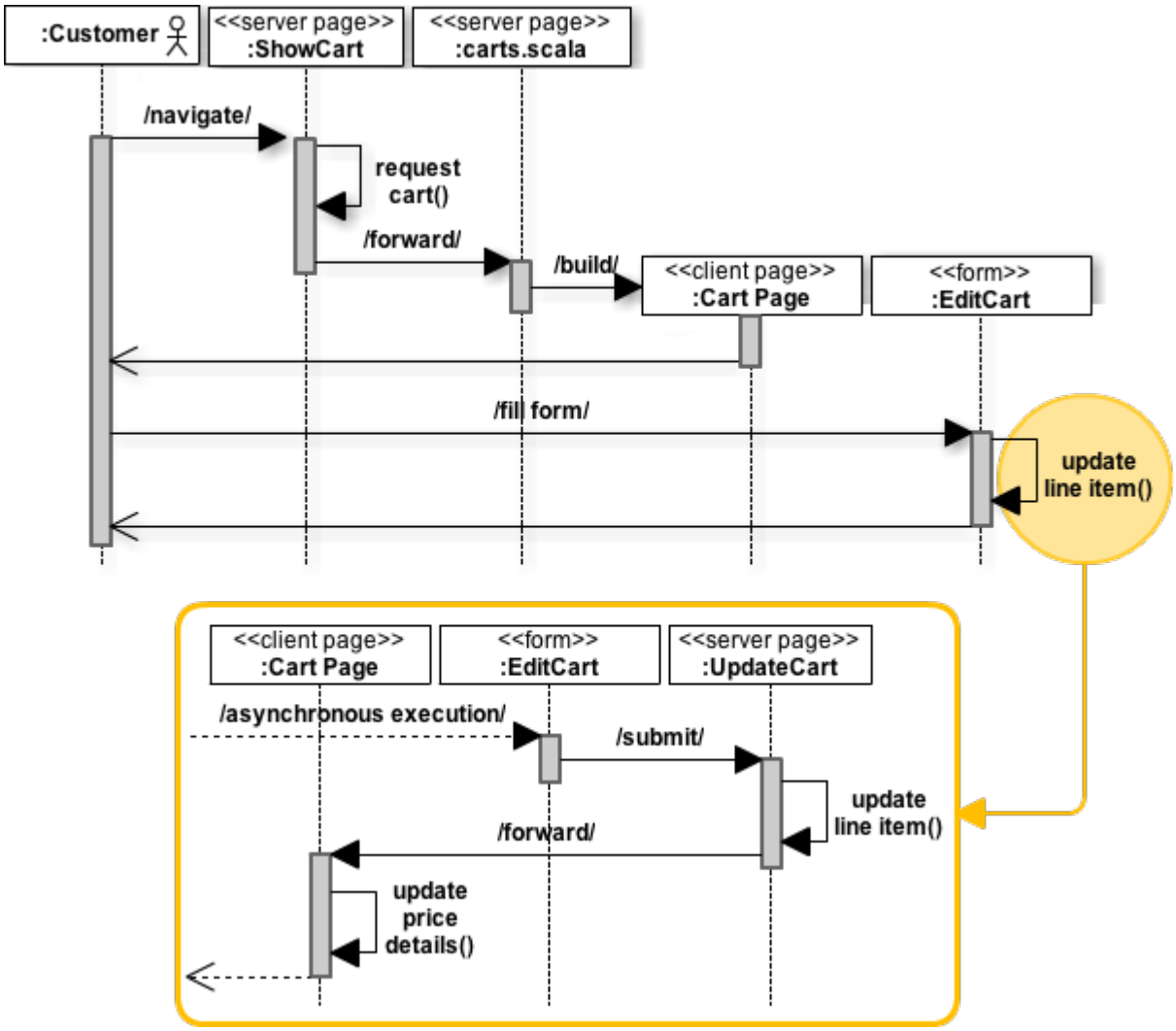


Figure 4.28: Internal design sequence diagram of the *update item in cart* use case.

Unlike the use case for removing line items, when updating them the customer is filling a form. But the form does not have a submit button, instead it is automatically submitted when the user changes the quantity value. That is the reason why after filling the form the page is calling

the *update line item* method, which asynchronously submits the form to the server page *UpdateCart*.

As in any other background form submission in this project, the server page requests to the model the corresponding change and the related information is forwarded to the initial client page. From this page is called the *update price details* method, which in fact builds several page components and replaces with them the corresponding elements displayed in the page.

The next two diagrams illustrate together the *place order* use case, being Figure 4.29 focused on the checkout page, and Figure 4.30 on the order creation and display of a summary. The first diagram has several forms, one for each element to be modified individually: the shipping address, the shipping method, the billing address and finally the payment form.

The payment form does not appear in the diagram because its submission is not directed to this system, but to the payment platform, as it will be described in the corresponding sequence diagram. In any case, it would only consist of a submit button and a dynamic payment form loaded with the data coming from Optile. As the actual content is unknown, it was considered best to leave it out from the diagram.

All other forms from the diagram are submitting in the background as the previous example. In every asynchronous call, the updated information regarding the current checkout state is forwarded to the checkout page. On the other hand, some of the data needs to be requested explicitly to the server, such as the list of shipping methods or the address book of a registered customer, to avoid performing repeated unnecessary calls to the SPHERE.IO backend.

The second diagram follows a simple design: the moment the customer is redirected from the payment platform back to the system with a successful payment, the server page creates the order and forwards all its information to the template, which in turn creates a page with a summary of the order to be displayed to the customer.

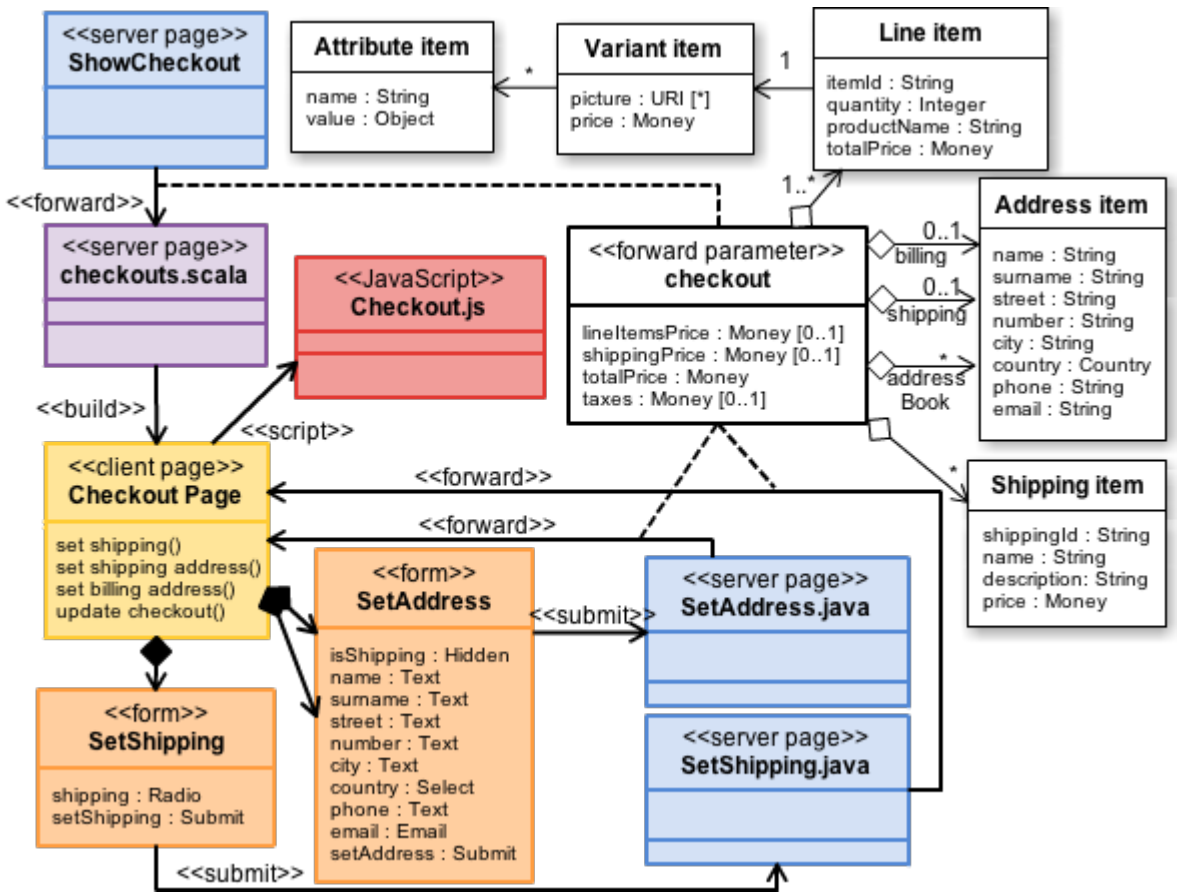


Figure 4.29: Internal design class diagram of the *checkout* top-level use case, detail of the checkout process.

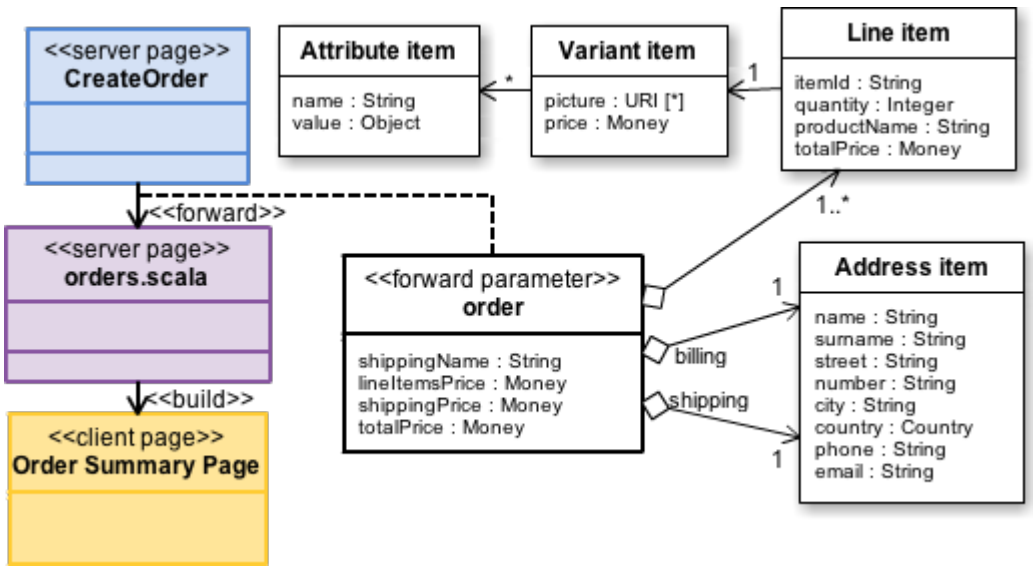


Figure 4.30: Internal design class diagram of the *checkout* top-level use case, detail of the order creation.

In contrast, the internal sequence diagram of the order creation is much more complex than the class diagram (see Figure 4.31). Once the checkout form is filled and the customer decides to submit it, the data is sent directly to the payment platform in order to avoid being PCI compliant. The platform proceeds charging the customer and then redirects him to the system. There the order is created and the order summary is displayed.

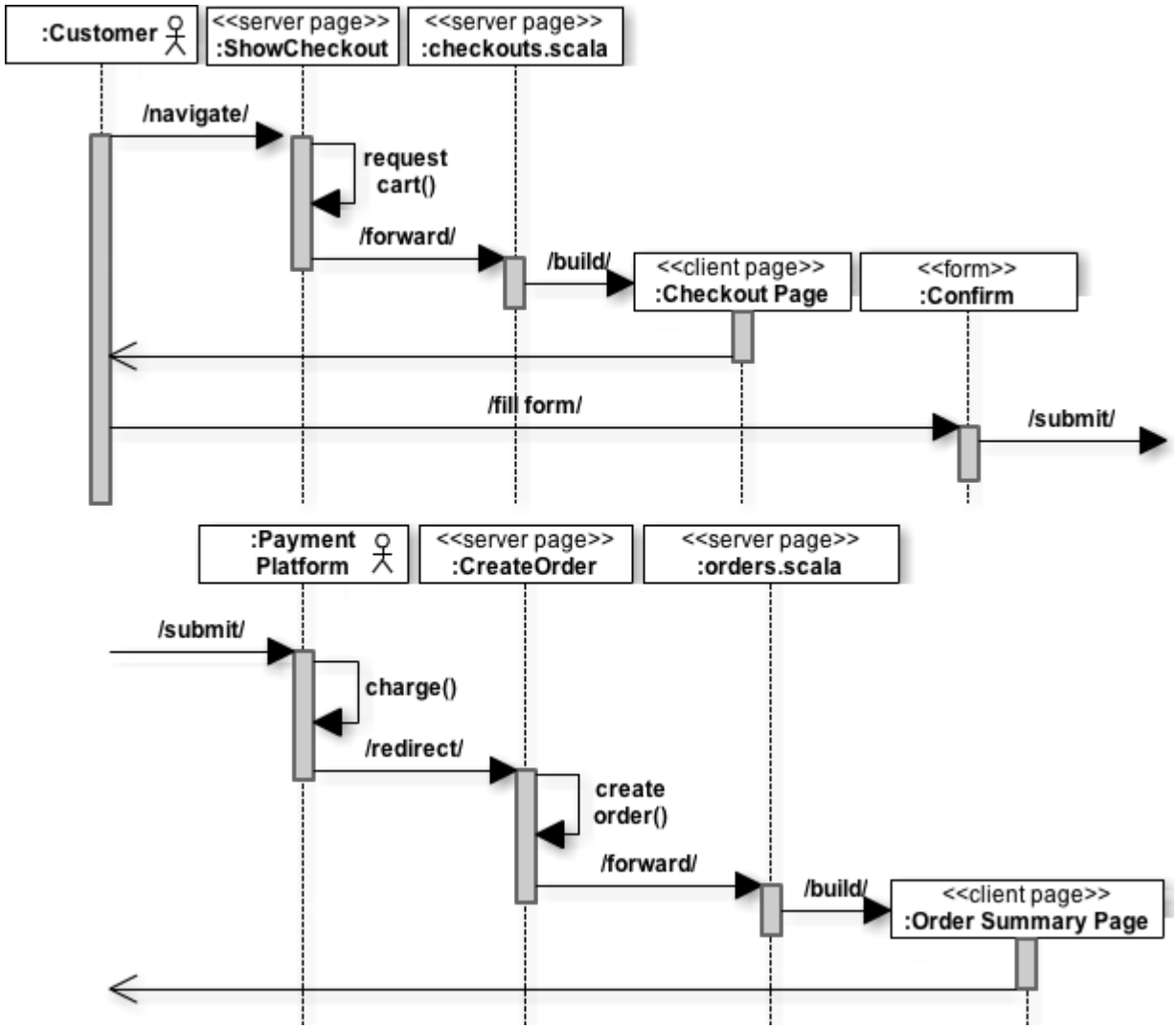


Figure 4.31: Internal design sequence diagram of the *place order* and *payment* use cases.

The payment platform redirects the customer to a different server page based on whether the charging operation was successful. The order is therefore only created when the customer is

redirected to the successful server page. But this system would easily lead to fraud, because the user could try to access the page directly without being charged. To solve this issue, the order is created but always with a pending payment status.

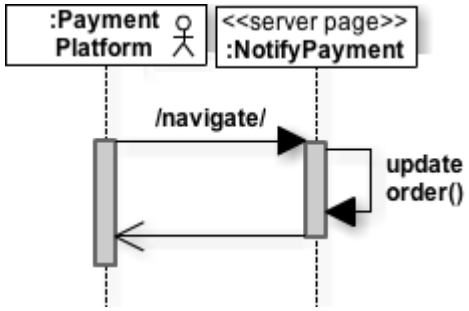


Figure 4.32: Internal design sequence diagram of the notification event in the *payment* use case.

The only possible way to update the payment status from the web-shop is to access the *NotifyPayment* server page, which access will be limited to the payment platform (see Figure 4.32). Optile requires this notification request to update the payment status of a certain order, something necessary when there is a change in the payment. This can easily happen when using PayPal, because the payment needs to be approved first, therefore changing multiple times of status during an undefined period of time.

4.2.4 Design of the Model component

As described at the beginning of the section System Logical Architectural 4.2, the logic of the model component is largely located in the SPHERE.IO Play SDK, which contains all the commerce logic and allows to access all data stored in the e-commerce backend. The *Sphere* class shown in the diagram below (Figure 4.33) is precisely the entry point for SPHERE.IO.

There is also a *Payment* class, a small library that will help to communicate with the Optile API, that requires the messages to be sent using XML. As explained before (see section 4.2.1.2), Optile needs to be implemented in an incremental way, reason why the library can effortless cover all five levels of integration, thus allowing developers to easily switch to the level it fits best for them.

The system also requires a class to send emails through any email system of preference. The *Mail* class will cover this functionality, as long as the SMTP[17] details of the email system are provided. Given that Heroku does not provide an internal SMTP server, the deployed version of this project will need to use an external server like Mailjet, a cloud emailing platform that offers several features that may be of interest for potential clients.

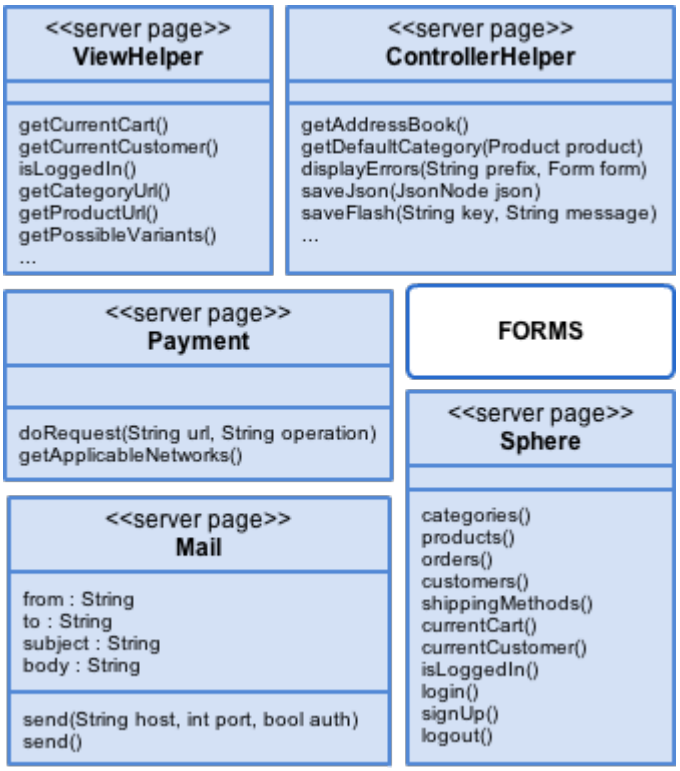


Figure 4.33: Overview of the server-side Model component.

The Model component is also containing different helpers, where some particular logic of this web-shop is located. The *ControllerHelper* is composed of methods that allows to abstract some common logic that is used in the Controller component (i.e. logic to handle and display messages and errors) or data coming from SPHERE.IO requiring some previous manipulation

[17] SMTP stands for Simple Mail Transfer Protocol and is a standard for email transmission through the Internet network.

before it is used (i.e. get default category of a product or get address book of the current customer).

On the other hand, the View Helper is a common design pattern that allows to separate logic that otherwise needs to be integrated in the template, in this project applied with the *ViewHelper* class. Although templates in Play Framework enables to use all the potential of the programming language Scala, it is a good practice to keep complex logic out of the templates. All this logic is then placed in these helper classes and called from the views as necessary.

Lastly, there are a group of classes related to the web forms and the payment information received by the system (see Figure 4.34). They handle all the server-side validation for every parameter and may also provide helpful getters and setters to easily convert model data into form data, and vice versa (e.g. an Address class instance would be converted into the appropriate form fields street, city, country, etc.)

These form classes also host the methods generating the different content that must be sent back to the client in relation to the result of the form submission. For example, when updating a line item from the cart, a success response contains a message for the user and all the shopping cart related information. This related information is generated with some other methods located in the forms as well, that convert a model class instance into JSON data.

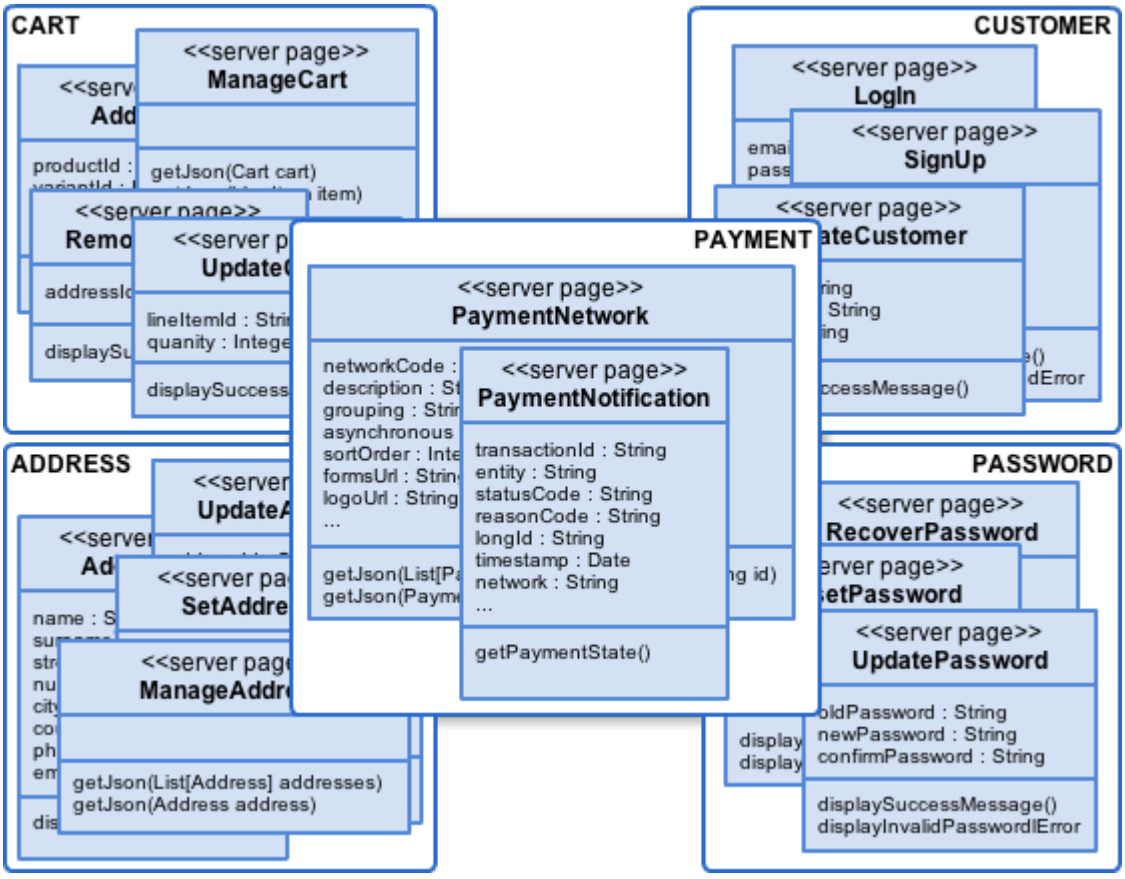


Figure 4.34: Overview of the Form package of the server-side Model component.

Due to the design of the system, the client side has also some important elements from the Model component. Although the separation between Model and Controller is not properly defined in the client side, mainly because of the simplicity of the business logic located there, there are also some client-side classes that are dedicated mainly to processing and validation purposes (see Figure 4.35).

Figure 4.35: Overview of the client-side Model component.

On the one hand there are some classes in charge of different elements that are affected by multiple operations throughout the web-shop, such as the mini cart and the pricing detail of a cart. Aside from the methods to control the behavior of the elements, there are methods that allow to replace the page component data with some JSON data fetched from the server. On the other hand there is a *Form* class that gives support to all forms of the system, by validating, displaying messages, marking invalid fields and managing AJAX calls when submitting.

4.2.5 Design of the View component

The View component is formed of several templates that are directly called by the Controller component, as seen in the internal design diagrams of the system. All these templates are making use of a main template file that provides a common HTML structure to all the pages, such as the basic contents and imports of scripts and stylesheets. They achieve this by importing the main template, following the design pattern called View Composition (Figure 4.36).

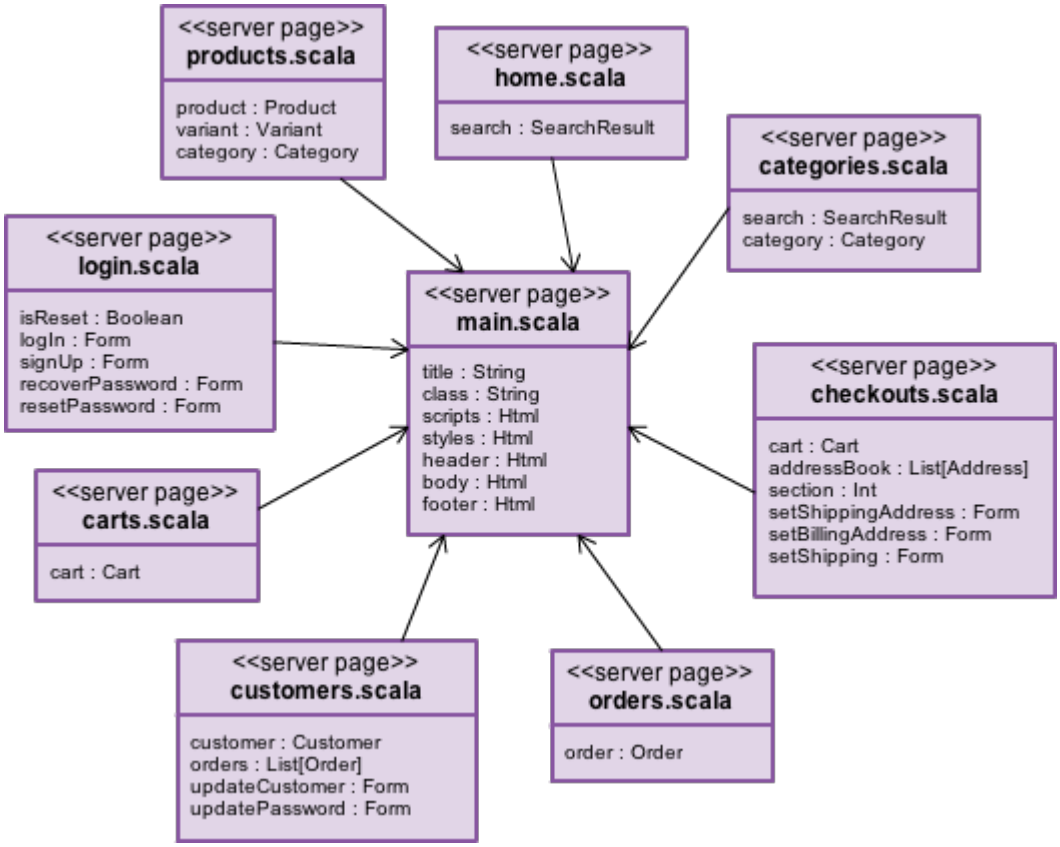


Figure 4.36: Overview of the composite views and main template of the server-side View component.

Just as in the previous example, the composite views are importing the contents of many other simple views, like the helpers and forms displayed in Figure 4.37. Both types of templates are usually offering components that are repeated throughout all client pages, although sometimes the separation is only intended to extract large pieces of meaningful HTML content from the composite views, as it is usually the case of the form templates. Unlike forms and helpers, mail templates are not meant for view composition, but are directly used to generate the HTML body of the emails sent by the system.

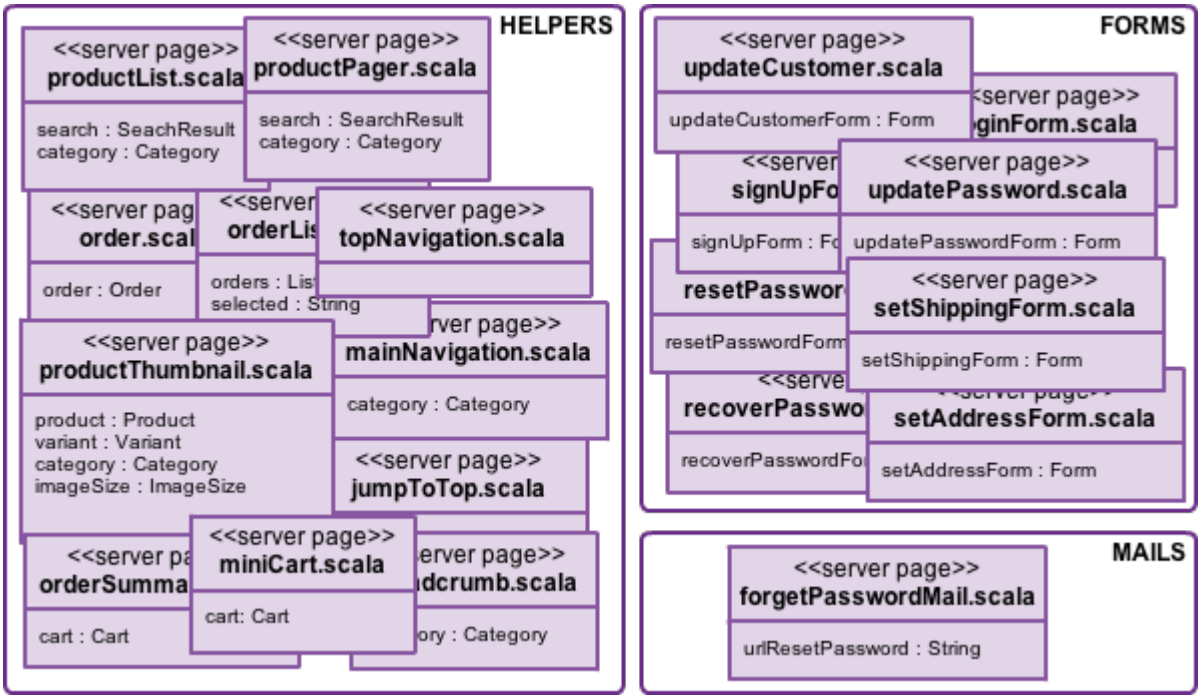


Figure 4.37: Overview of the simple views of the server-side View component.

As explained earlier (see section 4.2.1.7), the system is using Handlebars templates on the client side in order to generate updated content. Below all necessary templates for the system are displayed (see Figure 4.38), which belong on the one hand to the mini cart and pricing details components, and on the other hand to the use cases for product pagination, checkout and address management.

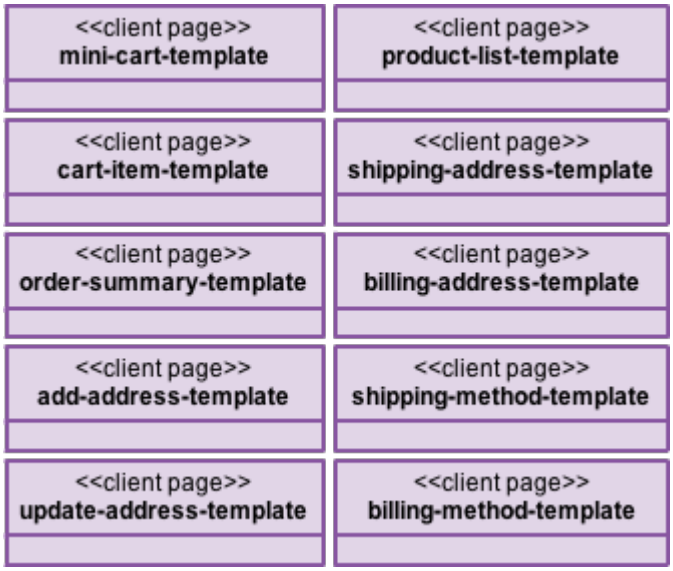


Figure 4.38: Overview of the templates of the client-side View component.

4.2.6 *Design of the Controller component*

Play Framework is applying the Front Controller and Application Controller design patterns to split the Controller logic between the common logic, located in the Front Controller, and the one specific for each request, called Application Controller. The Front Controller is mainly formed by the *RoutingSystem*, which is in charge of receiving, analyzing and dispatching every request to the appropriate application controller.

All application controllers of the system are shown in Figure 4.39, along with some classes that are used as filters. These filters intercept the application controller invocation and allows to execute code before and after the action is invoked. Filters are applied as a chain of filters to any desired controller action, although for simplicity the diagram is not specifying which particular actions are using the filter. A typical filter example is the *Authorization* class, that verifies the customer is correctly identified to the system before accessing a restricted functionality, otherwise it redirects the user to the login screen.

The *CartNotEmpty* filter checks the shopping cart has at least one line item, a requirement to access certain areas such as the cart detail page or the checkout, otherwise it redirects to the last visited browsing page. This last visited page is actually saved using the *SaveContext* filter, so that only actions intercepted by this filter are saved as a return page. It was considered a

good navigation behavior to save only pages related to products (i.e. product list and detail screens), leaving out checkout and customer management pages.

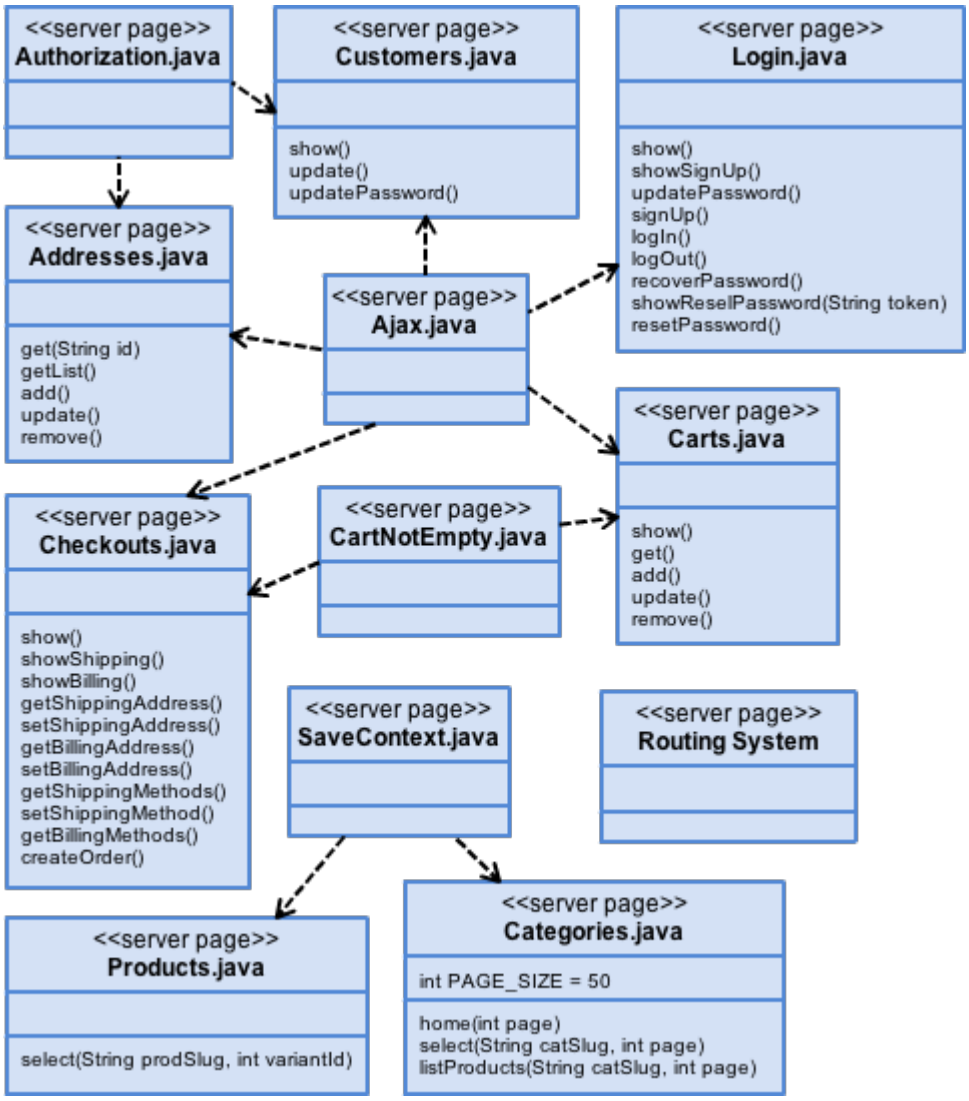


Figure 4.39: Overview of the controllers and filters of the server-side Controller component.

Finally, the *Form* filter class enables the transparent handling of a form submission response, independently of the technology used. This allows to switch from AJAX to a regular HTML form without any changes, only disabling the form submission handling in JavaScript. The filter helps developers to build both thin- and fat-client systems on top of the same Controller and View components.

Figure 4.40 shows the files of the Controller component on the client side. These files are dedicated to handle some of the events triggered by the user (e.g. form submission or page scrolling) in a specific page or component. The same files host other presentation logic used by the handlers, as well as methods to load new components from JSON data fetched from the server.

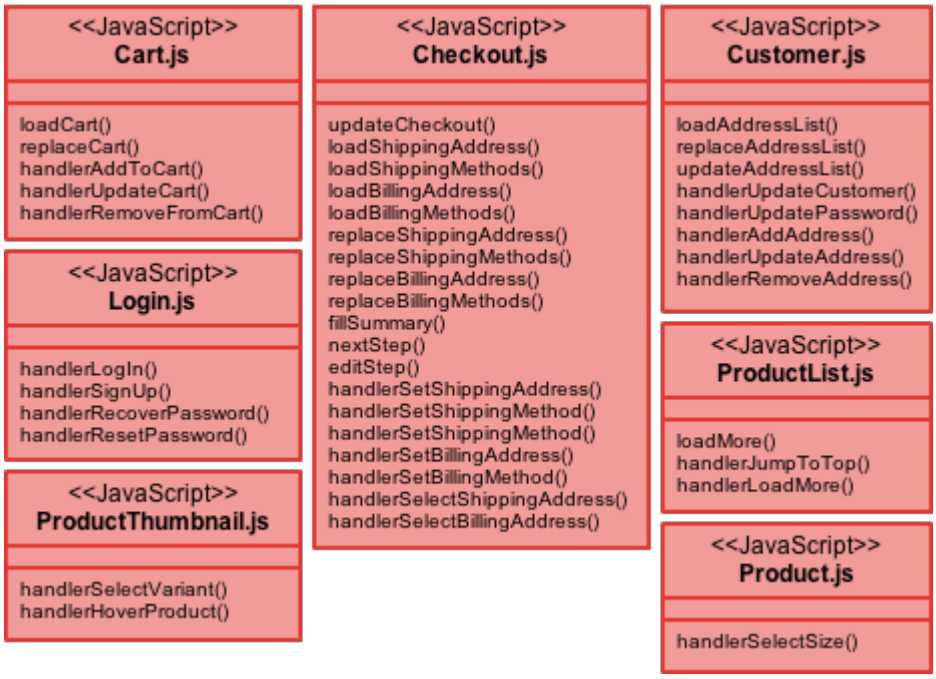


Figure 4.40: Overview of the controllers of the client-side Controller component.

5 IMPLEMENTATION

The implementation section is describing the actual process followed to develop the web-shop presented in the previous chapters. On the one hand is explained the set environment (section Development Environment 5.1), such as the software to assist the implementation or the stages of the development pipeline. On the other hand, some examples are detailed, along with the corresponding pieces of code, where it is shown how every technology is applied in order to solve some particular requirement (section Examples of Used Technologies 5.2).

5.1 DEVELOPMENT ENVIRONMENT

The development pipeline consists of three stages: development, staging and production. The development environment is a portable machine with OS X as operative system. The source code is developed with the support of the Java IDE Intellij IDEA Community Edition, mainly used for its debugging and code edition features. Google Chrome is the preferred web browser, which has a built in developer tool, Chrome DevTools, highly useful to inspect HTML DOM and CSS, as well as debugging JavaScript code.

Git is used as a revision control system. The most notable characteristic of Git is its distributed system, in which each user has his own local repository where changes are committed. Only when the developer deems it convenient, the local changes are then synchronized with the remote repository, thus making them accessible to the whole team. The remote repository is hosted by GitHub, with a very interesting social networking functionality useful for future collaboration with the developer community.

In everyday's development, Continuous Deployment technique is followed (see Figure 5.1). Jenkins is used in the staging environment as a continuous deployment tool, triggering a process to deploy the system every time changes are pushed to the remote repository. This process consists of building and testing the system, running automated acceptance test and deploying the project to production once staging is stable and ready. Whenever these steps fail at some point, the process is stopped and feedback is registered in order to solve the problem.

For this to work, every new feature developed for the system should always go along with tests validating that feature. Automating these tests on a staging system allows to flawlessly merge small pieces of code with the mainline of the project at a rapid pace. The code merging also

triggers a review process with all developers involved, which results in higher code quality. Besides, acceptance tests that verify the business logic can be run each time to ensure that the project requirements are met.

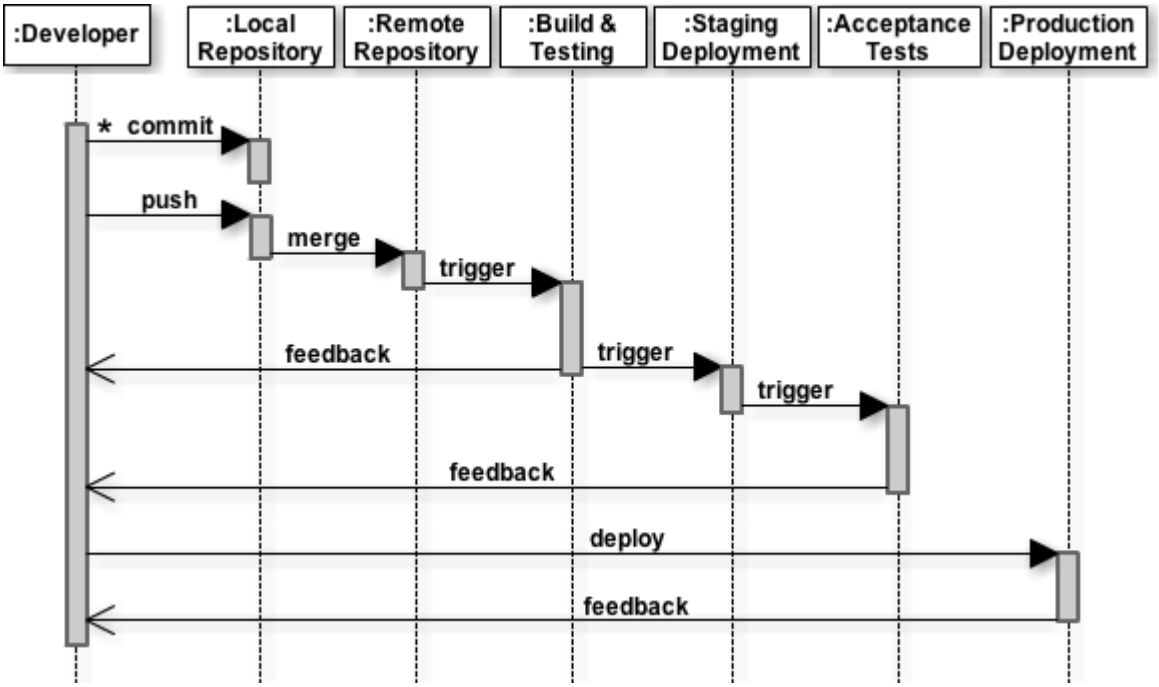


Figure 5.1: Sequence diagram of the continuous delivery process.

5.2 EXAMPLES OF USED TECHNOLOGIES

Next are presented three different examples of functionalities that allow to demonstrate how technologies are used in the project. These examples have been selected according to the importance of the process within the system or because it presented some challenges that are worth being mentioned.

Code generation required some extra effort to guarantee the stability of the system when data from the e-commerce backend is changed. Although the set of data used in the template has some fixed characteristics, the system has been implemented to support any kind of data that may be variable, such as product attributes, categories or currencies. The primary goal is to avoid crashing the system or break some elemental functionality when data is changed.

5.2.1 FORMS

Any form submission requires the use of most of the technologies presented in this project, making this functionality very appropriate to show examples of how the technologies are used. In particular, the *update item in cart* use case is going to be explained, which design was already detailed in the section Internal Design 4.2.3.

As mentioned before, form data can be either sent as a regular HTML form submission or via an AJAX call, indistinctly. The server logic processes both requests the same way thanks to the intercepting filter Form.java, which is in charge of returning the most convenient response to the client. This system improves understandability and changeability of the code, as it will be explained along this section.

The flowchart below (Figure 5.2) illustrates the process that takes place from the moment the user submits the form until the server returns a response. As a guide, on the left side is indicated the element or file where each action is executed in this particular use case, although the process itself is the same for every use case with form submission. The source code corresponding to some of these actions is presented in this section, along with some detailed explanation.

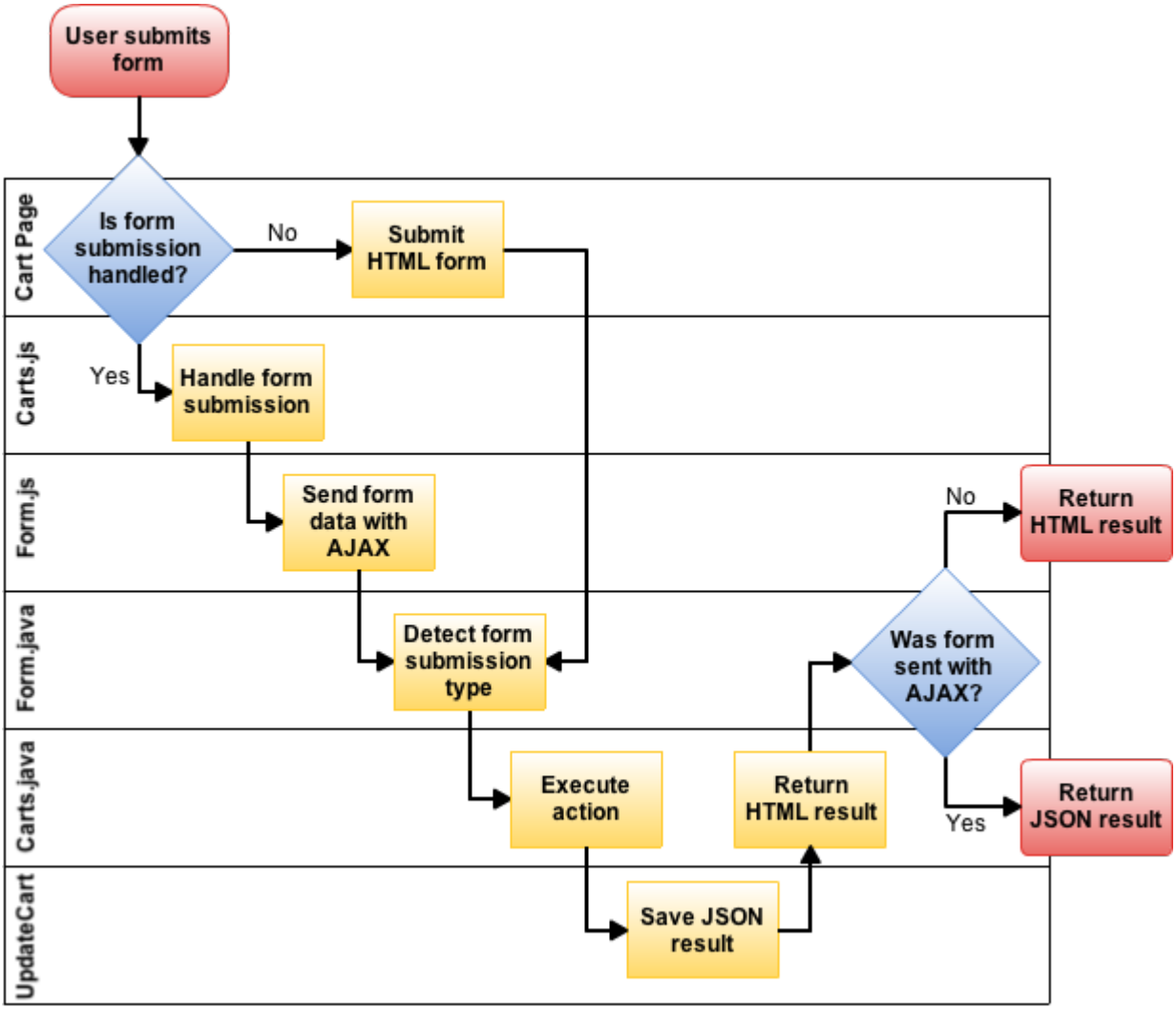


Figure 5.2: Flowchart of the form submission process.

When the user is submitting a form, the HTML DOM triggers an event that is then captured by a handler. The handler stops the regular form submission and sends the request via AJAX. If the handler is not found, the form is submitted in a traditional way to the server via HTML forms. Below is displayed the specific Coffeescript code that implements the handler for the *update item in cart* use case (see Figure 5.3).

```
$ ->
    # When changing a line item quantity, submit new quantity to server
    delay = {}
    $("#cart").on "change", "form.form-update-cart input[name=quantity]", ->
        # Create a new form instance with line item HTML form
        updateCart = new Form $(this).closest('form')
        # Execute common logic for starting form submission
        updateCart.startSubmit()
        # Extract form data from HTML form
        url = updateCart.form.attr("action")
        method = updateCart.form.attr("method")
        data = updateCart.form.serialize()
        # Reset delayed submission for this line item
        item = updateCart.inputs.filter('[name=lineItemId]').val()
        clearTimeout delay.timeout if item is delay.item and delay.timeout?
        # Create a delayed submission of 800ms for this line item
        delay.item = item
        delay.timeout = setTimeout ( ->
            # Submit form to server
            xhr = updateCart.submit(url, method, data)
            # When submission was successful...
            xhr.done (res) ->
                # Execute common logic for successful submission
                updateCart.doneSubmit res
                # Replace cart price details component with new data
                orderSummary.replace res.data
                # Replace mini cart component with new data
                miniCart.replace res.data
                # Replace cart detail component with new data
                replaceCart res.data
            # When submission failed, execute common logic for failed submission
            xhr.fail (res) -> updateCart.failSubmit res
            # In any case execute common logic for finishing form submission
            xhr.always -> updateCart.stopSubmit()
        ), 800
```

Figure 5.3: Source code for handling form submission, in cart.coffee.

In this particular case the handler is not called after submission, but after changing the value of the input field for the quantity. It is interesting to notice the use of the “on” jQuery method to bind the event handler. As shown below there are three different ways to bind an event: the first method is preferred and is actually a shortcut of the second one (see Figure 5.4). The difference between these two and the last method lies in the object to which the event handler is attached when the script is loaded.

```
$("#cart form.form-update-cart input[name=quantity]").change ->

$("#cart form.form-update-cart input[name=quantity]").on "change", ->

$("#cart").on "change", "form.form-update-cart input[name=quantity]", ->
```

Figure 5.4: Comparison of the different jQuery methods to bind an event handler to the "change" JavaScript event.

While the handler is directly attached to the quantity input in the first two options, in the latter is attached to the HTML object identified by "cart", specifying as a parameter the path of the quantity input. This difference is important when a particular object that can trigger the event may not exist at load time, which is the case of this form, since all the line items are being loaded via AJAX after the page is loaded. In this case the event is attached to a stable parent object, so that jQuery executes the handler for every descendant object that matches the path.

Submitting the form each time the input value is changed can lead to an excess of calls to the server if the user changes the quantity value too often. This has been solved with a delayed submission, implemented with the JavaScript event *setTimeout*, which is restarted every time the user changes the input field (see Figure 5.5 for an example of CoffeeScript code based on this functionality). This JavaScript timing event executes a function that is called after the specified period of time, in this case sending the form data via AJAX and processing the response accordingly.

```
clearTimeout delay.timeout if item is delay.item and delay.timeout?

if (item === delay.item && (delay.timeout != null)) {
    clearTimeout(delay.timeout);
}
```

Figure 5.5: Example of CoffeeScript code and the corresponding generated JavaScript after compilation.

In order to handle forms easily a new instance of the Form class is created, which processes all data related to the form to allow quick access to the information (see Figure 5.6). The method for sending the form data is simply calling the jQuery "ajax" function, additionally specifying in the URL that the request is performed via AJAX. The method returns jqXHR, a

XMLHttpRequest[18] object with jQuery added information. A series of methods can be chained to the jqXHR object, and executed once the server response has been received under certain circumstances. It is the case of “done” for successful requests, “fail” for failures and “always” for any condition; all of them used in this use case.

```
class window.Form
    # Form class constructor with attributes
    constructor: (@form, @dataType = 'json', @async = true) ->
        @saved = true
        @allowSubmit = false
        # Set common form elements for quick access
        @labels = @form.find('label, fieldset')
        @inputs = @form.find(':input')
        @buttons = @inputs.filter('[type=submit]')
        @messages = @form.find('.messages')

    # Submit form data
    submit: (url, method, data) ->
        url += (if url.split('?')[1] then '&' else '?') + "ajax=true"
        xhr = $.ajax url,
            type: method
            async: @async
            data: data
            dataType: @dataType
```

Figure 5.6: Source code for sending form data with AJAX, in lib-form.coffee.

Once the request reaches the server, the routing system built in Play Framework analyses the incoming HTTP method and the request path to decide the action controller to be invoked. For that matter, the system reads a file where every request pattern is linked with an action controller (see Figure 5.7), and selects the first pattern that matches the request. In addition, routing patterns can also have dynamic parts using the power of regular expressions.

[18] API to send HTTP or HTTPS requests to a web server and receive information in response, from a web browser scripting language such as JavaScript.

```
POST    /cart/add               controllers.Carts.add()
POST    /cart/update            controllers.Carts.update()
POST    /cart/remove            controllers.Carts.remove()
```

Figure 5.7: Extract of the Route file for cart management.

Figure 5.8 below shows the invoked controller action for this particular use case. Right before the method definition one can notice the "With" annotation, which Play Framework provides in order to attach a set of intercepting filters to an action. In this case the filter that intercepts the call is the *Form* class, in charge of detecting, before the action in called, whether the request from the client was made via AJAX.

The controller has a static attribute of an instance of Play's native *Form*, wrapping the class *UpdateCart* that handles incoming parameters to execute this action. When starting the call, the request is bound to the form instance and is checked for missing or invalid data, in which case errors are prepared for being displayed and the cart page is returned as a "bad request" (i.e. with a 400 HTTP status code).

```
final static Form<UpdateCart> updateCartForm = form(UpdateCart.class);

@With(FormHandler.class)
public static Result update() {
    Form<UpdateCart> form = updateCartForm.bindFromRequest();
    Cart cart;
    // Case missing or invalid form data, display errors
    if (form.hasErrors()) {
        displayErrors("update-cart", form);
        cart = sphere().currentCart().fetch();
        return badRequest(carts.render(cart));
    }
    // Case valid cart update, update quantity and display success message
    UpdateCart updateCart = form.get();
    CartUpdate cartUpdate = new CartUpdate()
            .setLineItemQuantity(updateCart.lineItemId, updateCart.quantity);
    cart = sphere().currentCart().update(cartUpdate);
    updateCart.displaySuccessMessage(cart);
    return ok(carts.render(cart));
}
```

Figure 5.8: Source code for executing the action for updating the cart, in Carts.java

When the request contains no errors, the SPHERE.IO Play SDK is called to update the cart, setting the new quantity requested for a particular line item. Then the success messages are prepared and the cart page is returned, this time with a 200 HTTP status code. The controller code appears therefore simple and clean of all related presentation logic. This is particularly important for this project, as controllers are the main place where SPHERE.IO Play SDK is used to execute e-commerce logic. Here is where code is most required to be understandable.

As seen in Figure 5.9 below, the message method consists of saving on the one hand a success message for the HTML result and, on the other hand, the JSON data with the same success message and the necessary cart information. The source code is also showing how request parameters are defined in a form, so that Play can bind and validate them. Some annotations are provided in order to specify constraints to incoming parameters, such as declaring a value required or accepting only a certain range. Any other type of pattern-based limitation can be applied as well with regular expressions.

```
@Constraints.Required(message = "Line item required")
public String lineItemId;

@Constraints.Required(message = "Quantity required")
@Constraints.Min(1)
public int quantity;

public void displaySuccessMessage(Cart cart) {
    String message = "Item updated!";
    // Save for HTML result
    saveFlash("success", message);
    // Save for JSON result
    ObjectNode json = Json.newObject();
    json.put("success", message);
    json.putAll(getJson(cart));
    saveJson(json);
}
```

Figure 5.9: Source code for handling form submission and saving JSON result, in UpdateCart.java.

Once the action call finishes, the intercepting filter again takes control, receiving as a result of the call a HTML file with a HTTP status code (see Figure 5.10). In some cases the result can also be a URL redirection instead of a HTML file, like is the case of the login use case. On the other hand, the filter has access as well to all JSON data that was stored earlier. So according to the type of incoming request, the filter returns either the result coming from the action call

or a JSON response with the same HTTP status code. With JSON, if the result was a redirection, the URL is extracted and sent to the client as part of the JSON data.

This system allows future developers to add new ways of handling form submissions without affecting the core of the system. Disabling a method, such as AJAX submission, is as simple as removing the event handler bound to the submit. Therefore the changeability of the code is not affected by the complexity of the presentation logic meant to improve user experience.

```
public Result call(Http.Context ctx) throws Throwable {
    // Before, detect submission type
    boolean isAjax = Boolean.valueOf(ctx.request().getQueryString("ajax"));
    // Call action, always returns HTML file or URL redirection
    Result result = delegate.call(ctx);
    // After, return:
    // - JSON data when AJAX and no redirection
    // - JSON with URL when AJAX and redirection
    // - Result from action otherwise
    if (isAjax) {
        // Extract saved JSON data
        ObjectNode json = Json.newObject();
        json.put("data", Json.toJson(ctx.args.get("json")));
        // Extract URL location header
        String url = JavaResultExtractor.getHeaders(result).get("Location");
        if (url != null) {
            // AJAX and redirection
            json.put("redirect", url);
            result = ok(json);
        } else {
            // AJAX and no redirection
            result = status(JavaResultExtractor.getStatus(result), json);
            ctx.flash().clear();
        }
    }
    ctx.args.remove("json");
    return result;
}
```

Figure 5.10: Source code for returning corresponding result to the client, in FormHandler.java.

When the AJAX call started on the client receives the response from the server, the methods chained to the jqXHR are then executed, depending on the HTTP status code obtained. In the current use case, if the response was successful, the pricing details and the mini cart will be replaced with new generated content, and each line item will be updated.

Moreover, some common logic for successful form submissions is executed, which source code is shown in detail below (see Figure 5.11). This common logic allows to handle properly the response sent by the Form intercepting filter: when required, it forces to handle URL redirections by executing a redirection with JavaScript, otherwise it updates automatically some tagged data from the page and displays the success message.

```
# Method to execute common logic for successful submission
doneSubmit: (res) ->
    # If expected data type is JSON...
    if @dataType is 'json'
        # Redirect page when URL received
        return window.location.href = res.redirect if res.redirect?
        # Otherwise update page with received data
        $.each res.data, (key, value) ->
            elem = $("span[data-form-update-#{key}]")
            elem.text(value)
    # Display success message
    @saved = true
    @displaySuccessMessage(res.data['success'])
```

Figure 5.11: Source code for handling successful form submissions, in lib-form.coffee.

As mentioned before, each line item on the page is updated with every successful submission. Actually the only element from the line items that needs to be updated is the total line item price (i.e. the product price multiplied by the new quantity), and this calculation could even be performed on the client side. But all the required data is already included in the JSON data nonetheless, needed to update the mini cart. Besides it is safer to always force a complete update of the cart contents, in case the user changes the cart from another browser window.

```coffee
$ ->
    # Compile template on page load
    html = $("#cart-item-template").html()
    template = Handlebars.compile $.trim(html) if html?

    # Method to replace the whole cart detail
    cartContent = $('#cart-content')
    replaceCart = (cart) ->
        # Do nothing when missing template or data
        return unless template? and cart?
        # Remove current cart content
        cartContent.empty()
        # Append each item rendering the template
        cartContent.append(template item) for item in cart.item
```

Figure 5.12: Source code for replacing the cart contents, in cart.coffee.

The method displayed in Figure 5.12 is the one in charge of replacing the cart line items. To do so, the Handlebars template that renders a single line item is obtained and compiled once when the page is loaded. When the method is called, the current contents of the cart are removed from the cart container and each line item is rendered with the template and then appended to it.

The following two figures helps one understand how these templates are being rendered. The first one corresponds to the template itself (Figure 5.13), with the Handlebars expressions surrounded by curly brackets. When the expression starts with a hash symbol, the expression corresponds to a helper to control the flow, like the looping statement “each” used here. Otherwise it refers to a value extracted from the JSON data sent as parameter to the template.

```
<script id="cart-item-template" type="text/x-handlebars-template">
  <li id="item-line-{{itemId}}" class="item-line">
    ...
    <div class="item-picture">
      <img src="{{image.small}}" alt="{{productName}}"/>
    </div>
    ...
    <form class="form-update-cart" action="@routes.Carts.update" method="post">
      <input type="hidden" name="lineItemId" value="{{itemId}}"/>
      <input type="number" name="quantity" value="{{quantity}}" min="1"/>
    </form>
    ...
    <form class="form-remove-from-cart" action="@routes.Carts.remove" method="post">
      <input type="hidden" name="lineItemId" value="{{itemId}}"/>
      <button type="submit" name="remove" data-loading-text="Removing...">
        Remove
      </button>
    </form>
    ...
    <ul class="unstyled">
    {{#each attribute}}
      <li><strong>{{name}}:</strong> {{value}}</li>
    {{/each}}
    </ul>
    ...
  </li>
</script>
```

Figure 5.13: Simplified source code of the Handlebars template to render a cart line item, in carts.scala.html.

An extract of the JSON data is presented next (Figure 5.14), as it is received by the client after the request. It should be noted that the *replaceCart* method is only sending to the template one "item" object each time. It is easy then to understand the correspondence between the JSON data and the expressions appearing in the template.

```
{
   "data":{
      "success":"Item updated!",
      "currency":"EUR",
      "totalPrice":"666.00",
      "item":[
         {
            "itemId":"f5c2db6b-b489-41f5-9368-2b61054dc9eb",
            "productId":"fcb47fc1-ac26-453a-be13-678c7848c485",
            "productName":"MISSION EST",
            "quantity":3,
            "currency":"EUR",
            "price":"118.00",
            "totalPrice":"354.00",
            "attribute":[
               {
                  "name":"size",
                  "value":"XL"
               }
            ],
            "image":{
               "thumbnail":"https://.../252169001_1-bn2zqPOG-thumb.jpg",
               "small":"https://.../252169001_1-bn2zqPOG-small.jpg",
               "medium":"https://.../252169001_1-bn2zqPOG-medium.jpg",
               "large":"https://.../252169001_1-bn2zqPOG-large.jpg",
               "original":"https://.../252169001_1-bn2zqPOG.jpg"
            }
         },
         {
            "itemId":"18e5b820-ee7e-4757-8ded-d727a3fcac96",
            "productId":"..."
         }
      ]
   }
}
```


Figure 5.14: Example of JSON data received from the server, in *update item in cart* use case.

5.2.2 LIST PRODUCTS

Product listing is another interesting feature in the implementation process. As explained in the section Internal Design 4.2.3, pages of products are being loaded automatically when the user is near the bottom of the page. This way customers are not fully aware of browsing

through pages, from their point of view new products are constantly presented as they scroll down the window of the browser, in what appears to be a large single page.

Figure 5.15 below illustrates the code of a function that decides when the page needs to load more products and when a button to allow a quick return to the top of the page is shown. The criteria to execute both actions depend on the vertical scrolling position of the display, so this position needs to be checked regularly for changes. One option was to trigger the function whenever the user scrolls, but this event is fired too often when scrolling and would affect negatively the performance of the page. An alternative is to execute the function in a fixed interval of time, that balances a smooth navigation with a fast loading before the user reaches the bottom.

```
$ ->
  # Check scroll position to fire 'jump to top' and 'load products' actions
  checkScrollingPosition = ->
      # Show the jump to top tag if it is not in the page top
      isNearTop = $(window).scrollTop() < 100
      jumpToTop.fadeOut() if isNearTop and jumpToTop.is(":visible")
      jumpToTop.fadeIn() if not isNearTop and jumpToTop.is(":hidden")
      # Load more products if it is near the page bottom
      limit = $(document).height() - $(window).height() - distance
      isNearBottom = $(window).scrollTop() >= limit
      loadMore() if isNearBottom

  setInterval checkScrollingPosition, 400
```

Figure 5.15: Source code to load more products and other actions related to page scrolling, in product-list.coffee.

The method to load more products calls the server via AJAX, requesting to the controller action below a particular page of a list of products. The action starts building the product search request that is sent to SPHERE.IO backend in order to get the desired list of products, first requesting all products available. If a category slug[19] was provided, the category object is fetched from SPHERE.IO and is used to filter the initial list in the *filterBy* method, which source code is presented next (Figure 5.16).

[19] Slug is understood in this context as the human-readable keywords identifying a resource, as part of a URL.

```
public static Result listProducts(String categorySlug, String sort, int page) {
    Category category = null;
    List<Category> categories = new ArrayList<Category>();
    SearchRequest<Product> searchRequest = sphere().products().all();
    if (!categorySlug.isEmpty()) {
        category = sphere().categories().getBySlug(categorySlug);
        categories.add(category);
    }
    searchRequest = filterBy(searchRequest, categories);
    searchRequest = sortBy(searchRequest, sort);
    searchRequest = paging(searchRequest, page);
    SearchResult<Product> searchResult = searchRequest.fetch();
    return ok(ListProducts.getJson(searchResult, category, sort));
}
```

Figure 5.16: Source code to list products for a particular page and matching criteria, in Categories.java.

The search of products can be performed with either regular filters (e.g. all products within a price range) or faceted searches, in which is returned, besides the list of filtered products, all possible options for that attribute and the number of products that belong to each options (e.g. a faceted search for the color attribute returns all possible colors for that search and, for each color, the number of products with that color).

```
// Filter by category
searchRequest = searchRequest.filter(
        new FilterExpressions.CategoriesOrSubcategories(categories));
// Filter by request parameters
searchRequest = searchRequest.filter(bindFiltersFromRequest(ProductFilters.filters));
// Facet by request parameters
searchRequest = searchRequest.facet(bindFacetsFromRequest(ProductFilters.facets));
```

Figure 5.17: Source code to filter the list of products, in Categories.java.

SPHERE.IO Play SDK provides a set of filter and faceted search expressions to flexibly build requests for products matching a certain criteria for a certain attribute. For example, above is used a category filter expression to get all the products that are present in the set of categories provided, or any of the descendent categories (see Figure 5.17). The SDK also offers some classes to help implementing filters for the user interface, such as search boxes to find

products. These classes have methods to parse the filter values from the URL query string and build the product search request from those values easily. They can also provide some useful data for specific filters, like the maximum and minimum prices of a product list for the price filter.

```
// Fulltext search
public static final Filters.Fulltext fulltextSearch =
        new Filters.Fulltext().setQueryParam("q");

// Filter price
public static final Filters.Price.DynamicRange filterPrice =
        new Filters.Price.DynamicRange().setQueryParam("price");

// A special collection for filters that checks parameter name conflict
public static final List<Filter> filters = Arrays.<Filter>asList(
        filterPrice,
        fulltextSearch
);
```

Figure 5.18: Source code with the declaration of filters, in ProductFilters.java.

Figure 5.18 shows the code that declares two of the filters used in this project with the corresponding keyword used in the URL query string. They are both included in the filter list that is directly bound to the request in the Figure 5.17. The only piece missing to have a functional product search is the template rendering the search form, shown below (Figure 5.19). The HTML input name and value are obtained from the *fullTextSearch* parameter defined before, hence connecting the user filter interface with the SPHERE.IO filter request.

```
@import utils.ProductFilters
<div class="searchBox">
    <form id="searchForm" method="GET" action="@routes.Categories.search">
        <input name="@ProductFilters.fulltextSearch.getQueryParamName"
               type="text" placeholder="Product name, attributes..."
               value="@ProductFilters.fulltextSearch.parseValue(request.queryString)"/>
        <button type="submit">Search</button>
    </form>
</div>
```

Figure 5.19: Template of the product search box, in searchBox.scala.html.

As can be observed, in a Scala template expressions to be evaluated with Scala are preceded by a "@" sign. This way classes can be imported and used directly in the template like is the case with the filter *fullTextSearch*. It is also interesting to notice how URLs are generated, using the corresponding "reverse routing" method for the controller action that handles searches.

```
if (sort.equals("price_asc")) {
    searchRequest = searchRequest.sort(ProductSort.price.asc);
} else if (sort.equals("price_desc")) {
    searchRequest = searchRequest.sort(ProductSort.price.desc);
} else if (sort.equals("name_asc")) {
    searchRequest = searchRequest.sort(ProductSort.name.asc);
} else if (sort.equals("name_desc")) {
    searchRequest = searchRequest.sort(ProductSort.name.desc);
}
```

Figure 5.20: Source code to sort the list of products, in Categories.java.

```
if (currentPage < 1) currentPage = 1;
// Convert page from 1..N to 0..N-1
currentPage--;
return searchRequest.page(currentPage).pageSize(PAGE_SIZE);
```

Figure 5.21: Source code to paginate the list of products, in Categories.java.

Using the methods for sorting (Figure 5.20) and paging (Figure 5.21) is very simple, as the two pieces of code above demonstrates. Sorting is achieved by specifying the correct sorting criterion, provided via a *ProductSort* enumeration that contains all the possibilities currently offered by the SDK. Pagination requires the desired page and the amount of products per page. Once the request is completely built, the list of products is fetched from the SPHERE.IO backend and sent to the client as JSON data.

```coffee
$ ->
    # Get DOM elements
    productList = $("#product-list")
    productPager = $("#product-pager")
    # Compile templates
    template = {
        list: Handlebars.compile $.trim($("#product-item-template").html())
        pager: Handlebars.compile $.trim($("#product-pager-template").html())
    }
    # Initialize Masonry
    masonry = new Masonry(productList.get(0), {itemSelector: ".product-item"})

    # Append products to the product list
    appendProducts = (data) ->
        return unless template.list? and template.pager? and data?
        # Replace previous pager
        pagerHtml = template.pager data
        productPager.empty().append(pagerHtml).fadeTo(0, 0)
        # Append products
        page = $(template.list data).fadeTo(0, 0)
        empty = !$.trim(productList.html())
        productList.append page
        imagesLoaded productList, ->
            page.fadeTo(0, 1)
            if empty
                masonry.reloadItems()
                masonry.layout()
            else
                masonry.appended(page.get())
            productPager.fadeTo("slow", 1)
```

Figure 5.22: Source code to append a page of products to the list, in product-list.coffee.

When the response is received on the client side, the method *appendProducts* shown above (Figure 5.22) is called to attach the new data to the previous list and update the information about the next page. In order to achieve the product grid layout described in the section External Design 4.2.2, the JavaScript library Masonry has been used. This library takes all available space of the parent container and places each product thumbnail in optimal position, forming a condensed grid where elements perfectly fit together. A small library called imagesLoaded is also used to fire an event when all the images of the products have been fully loaded, so that Masonry can use the correct final size of each thumbnail.

Thumbnails are showing at first a certain product variant, but they should also allow to see and buy a different variant directly from the thumbnail. This functionality is only enabled for

color and size attributes variations, as more attributes would saturate the design. A list of variants is received from the server, with matching attributes but different color. The different variants are displayed as small pictures, and when the user hovers the cursor on these pictures the thumbnail variant changes to the hovered one, replacing all variant related information.

```
{{#each matchVariant}}
<li class="{{#if isActive}} active {{/if}}">
    <a href="{{url}}"
       data-image="{{#image this ../isFeatured}}{{this.url}}{{/image}}"
       data-price="{{price}}"
       data-sizes="{{#sizes this}}{{/sizes}}"
       data-variant="{{id}}">
        <img src="{{image.thumbnail.url}}"
             class="img-circle" title="{{attribute.color}}"
             width="{{image.thumbnail.width}}" height="{{image.thumbnail.height}}"/>
    </a>
</li>
{{/each}}
```

Figure 5.23: Source code of the Handlebars template to display matching color variants, in productList.scala.html.

As shown in the Handlebars template presented in Figure 5.23, there are several data that can change amongst variants: image, price, available sizes and, of course, the variant identifier. In order to determine the image and sizes for each variant, Handlebars helpers have been used. The helper to generate the image is displayed in Figure 5.24, which sets as the new context a specific size of the picture, depending on whether the product is featured or not.

```
Handlebars.registerHelper('image', (variant, isFeatured, options) ->
  image = if isFeatured then variant.image.medium else variant.image.small
  options.fn image
)
```

Figure 5.24: Handlebars helper to get correct variant image according to product characteristics.

When a product variant has different sizes, a list of sizes is shown to the customer when he hovers on the “quick buy” button, to allow him select the correct size. Next is shown the style

applied to the list of sizes, as an example of the use of LESS to generate CSS stylesheets (see Figure 5.25). The example uses the most important features of LESS: variables, here used for colors; functions, used to apply the CSS *border-radius* property cross-browser[20]; mixins, to remove the default style of the list; and nested rules, to better specify inheritance.

```less
@background: #F1F1F1;
@foreground: #FFFFFF;
@dark-gray:  #696969;
@light-gray: #CBCBCB;

.unstyled {
  margin-left: 0;
  list-style: none;
}

.border-radius(@radius) {
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
  border-radius: @radius;
}

#product-list {
  /* Rules of product-list here */
  .product-item {
    /* Rules of product-item here */
    ul.product-info-variants-size {
      display: none;
      position: absolute; bottom: 0;
      margin-bottom: -5px;
      background-color: @foreground;
      border: 1px solid @light-gray;
      .unstyled;
      .border-radius(3px);
    }
  }
}
```

```css
#product-list .product-item
ul.product-info-variants-size {
  display: none;
  position: absolute; bottom: 0;
  margin-bottom: -5px;
  background-color: #ffffff;
  border: 1px solid #cbcbcb;
  margin-left: 0;
  list-style: none;
  -webkit-border-radius: 3px;
  -moz-border-radius: 3px;
  border-radius: 3px;
}
```

Figure 5.25: Example of LESS stylesheet on the left and the corresponding generated CSS stylesheet on the right.

[20] Cross-browser refers to the ability of a web application feature to work correctly in all browsers, regardless of whether they provide the required functionality.

5.2.3 PAYMENT

A notable initial effort was required in order to integrate Optile in a flexible way. But despite of the fact that the payment platform was in the end correctly integrated in the system during the implementation process, the final result was not entirely satisfactory and was replaced for the second option: Paymill. The problem resided on a certain incompatibility in the checkout process designed for the SPHERE.IO Play SDK and the workflow expected by Optile.

The Optile workflow was already explained (see section 4.2.1.2), but the most characteristic behavior to remember is that it uses notification requests in order to keep the payment state of an order updated. Each notification needs to be handled, otherwise the SPHERE.IO backend may have wrong payment information.

On the other hand, the SPHERE.IO Play SDK was initially designed so that the creation of an order cannot be reversed. This means that once the order is created the customer's cart does not longer exist, so the only possible moment to create the order is right after the payment has succeed. But then the order creation process can still fail under some expected escenarios[21] and charge the customer without actually buying the items. The source code below shows the required workflow by SPHERE.IO Play SDK in order to create an order, and how the creation needs to be stopped if the customer changes anything from the cart (Figure 5.26).

```
// Case cart changed, show checkout with warning
if (!sphere().currentCart().isSafeToCreateOrder(cartSnapshot)) {
    flash("info", "Your cart has changed, please review everything is correct");
    return redirect(routes.Checkouts.show());
}
// Case order can be created, create it pending
Order order = sphere().currentCart().createOrder(cartSnapshot, PaymentState.Pending);
return ok(orders.render(order));
```

Figure 5.26: Source code example to create an order with SPHERE.IO Play SDK.

[21] When the customer is filling his billing information and opens another browser window to modify the cart, if he did not reload the checkout page before submitting the checkout form, the order will fail because the cart submitted does not match the current cart.

Of course a payment cancellation can always be requested in failure cases, the customer is usually already charged but then he is refunded, depending on the payment provider. But this behavior was considered to be too uncomfortable for the customer, especially because of the uncertainty of a scenario that should be perfectly controlled. Besides, there were other minor issues adding more instability to the process, so it was not considered a good solution for the implementation of the template until the issues were solved.

Paymill, despite of accepting only direct debit and credit cards as payment methods, allows on the other hand to keep control of the workflow along the process, thus allowing to charge the customer only when the order can be created. This is achieved by separating the submission of the payment data from the proper charging to the customer.

```
class window.Paymill
    ...
    # Method to handle form submission
    submit: (responseHandler) ->
        return false if @error
        # Get values according to payment type
        switch @paymentType
            when "ELV" then params = {
                    number: @elvNumber.val()
                    bank: @elvBank.val()
                    accountholder: @elvHolder.val()
                }
            else params = {
                    number: @cardNumber.val()
                    exp_month: @cardMonth.val()
                    exp_year: @cardYear.val()
                    cvc: @cardCvc.val()
                    cardholder: @cardHolder.val()
                    amount: @cardAmount.val() * 100
                    currency: @cardCurrency.val()
                }
        # Send data to Paymill and receive token
        paymill.createToken(params, responseHandler)
```

Figure 5.27: Source code to send payment data to Paymill and return a token, in Paymill.js.

The payment data is sent via JavaScript to the platform, using a library provided by Paymill. This request is executed the moment the customer submits the payment form, and returns a token in response. The token is attached to the checkout form, so when the customer submits the final form this token is sent to the server. Figure 5.27 shows the code snippet that is part of the class that was implemented to help validating the payment form and sending the data.

Once the checkout form submission reaches the server, the system has complete control over the order creation and payment execution, as the next code shows (Figure 5.28). First the cart is checked for any changes, then the customer is charged and, only when it is successful, the order is created. This process is robust and ensures that no customer is charged without the order being created.

```
// Case cart changed
if (!sphere().currentCart().isSafeToCreateOrder(cartSnapshot)) {
    doCheckout.displayCartChangedError();
    redirect(routes.Checkouts.show());
}
// Case payment failure
try {
    // Get payment object from token
    Paymill.setApiKey(paymillKey);
    PaymentService paymentSrv = Paymill.getService(PaymentService.class);
    Payment payment = paymentSrv.create(doCheckout.paymillToken);
    // Set transaction details
    TransactionService transactionSrv = Paymill.getService(TransactionService.class);
    Transaction transaction = new Transaction();
    Money money = getPrice(getCurrentCart());
    transaction.setPayment(payment);
    transaction.setAmount(Ints.checkedCast(money.getCentAmount()));
    transaction.setCurrency(money.getCurrencyCode());
    // Execute charge transaction
    transactionSrv.create(transaction);
} catch (PaymillException pe) {
    play.Logger.error(pe.getMessage());
    flash("error", "Payment failed unexpectedly, please try again");
    return internalServerError(showPage(4));
}
// Case success purchase
Order order = sphere().currentCart().createOrder(cartSnapshot, PaymentState.Paid);
flash("success", "Congratulations, you finished your order!");
return ok(orders.render(order));
```

Figure 5.28: Source code to execute payment and create order, in Checkouts.java.

But this system lacks flexibility, because it expects that the payment is executed immediately in a single step, which result cannot be changed. Unfortunately some payment providers, such as PayPal, are first requested and then the payment is executed at some moment in the future. Therefore Paymill is not a solution strong enough for this template either, but at least it offers a robust solution until SPHERE.IO and Optile evolve to become fully compatible.

6 SYSTEM TESTS

This section describes all kind of tests that have been run against the system, both periodical and one-time tests. As explained earlier (see section 5.1), every time a change is merged with the remote repository, a series of tests are executed to verify those changes did not introduce any error that may prevent the system from meeting the agreed requirements. This system triggers only those tests that allow to check the functional requirements of the system, called functional tests, but any other type of automated test can be executed as well to check, for example, that the response time of the system is within the agreed limit.

After functional testing is explained (section Functional Tests 6.1), the next section covers the usability tests used to check those non-functional requirements related to user and developer experience (section Usability Tests 6.2). And although it was not explicitly a requirement, it has been considered important to run some performance tests in order to detect and possibly fix those issues that may slow down the application (section Performance Tests 6.3).

The rest of requirements do not need any special setup to be checked. For example, in order to check the stability of the system, it is enough to change the SPHERE.IO project to another project with different structure of data and see if it meets the specified requirement (see list in Appendix B.2).

6.1 FUNCTIONAL TESTS

As explained in the section Development Environment 5.1, this project follows a continuous deployment process which requires functional tests to be implemented along with the feature, to ensure the functionality is working correctly every time a change is merged with the project. A major difficulty of developing functional tests in this project is that they should work with any set of commerce data, so that when developers switch from the test web-shop data to their own, the tests are still functional.

Tests must therefore be independent from the backend data used. The best way to achieve this is mocking the *Sphere* class that handles the requests and responses from the SPHERE.IO backend, to simulate the call to the backend getting only controlled information as response. This means that during the tests execution, all objects returned by the class whenever a call to the backend is theoretically executed are actually objects constructed by the testing code.

Not only this solution allows to execute the tests independently of the system's environment, but moreover performance is improved significantly since no remote resources are used, which hastens the feedback provided by the continuous development process. Also, mocking allows to easily test any situation that may prove complicated otherwise, like provoking certain errors from the backend (e.g. a "500 Internal Server Error" HTTP status code).

At the moment this project was developed there was no support for mocking coming from the SPHERE.IO Play SDK to ease this process. Besides the initial design was not contemplating the possible need to manipulate the response, so mostly all classes had private constructors, thus forcing the necessity to mock practically all SDK classes involved in the system and stub every method before any test could actually be executed. In most cases, SDK logic that was intended to work directly had to be simulated first in order to mock other classes, sometimes changing excessively the original behavior of the SDK.

Not only this process proved to be notably complex, but also testing code became messy and at some point it was even difficult to be sure that the tests were really meaningful. So in the end only the classes related to browsing products were completely mocked, ensuring at least some example tests to include with the template until a better approach is discussed and provided with the SPHERE.IO Play SDK. One of the best solutions for the developers would be to offer some methods to directly mock the response from the server, without requiring the developer to understand the internal structure of the SDK.

The selected technology to mock Java code was Mockito, a very popular testing framework due to the simplicity of the resulting code, yet a complete solution like any of the alternatives. Figure 6.1 presents an extract of the source code in charge of mocking the search request for products. In this example, the only element that is actually interesting to modify amongst tests is the list of products returned, all other stubbed methods and mocked objects are just adding unnecessary complexity to the code.

```
SearchRequest<Product> request = mock(SearchRequest.class);
when(request.filter(any(FilterExpression.class))).thenReturn(request);
when(request.filter(any(Iterable.class))).thenReturn(request);
when(request.facet(any(FacetExpression.class))).thenReturn(request);
when(request.facet(any(Collection.class))).thenReturn(request);
when(request.sort(any(ProductSort.class))).thenReturn(request);
when(request.page(anyInt())).thenReturn(request);
when(request.pageSize(anyInt())).thenReturn(request);
int total = products.size();
int offset = page * pageSize;
int count = Math.min(total - offset, pageSize);
List<Product> result = new ArrayList<Product>();
if (count > 0) result = products.subList(0, count);
SearchResult<Product> searchResult =
        new SearchResult<Product>(offset, count, total, result, mockFacets(), pageSize);
when(request.fetch()).thenReturn(searchResult);
```

Figure 6.1: Source code to mock the product search request, in SphereTestable.java.

The functional tests are designed after the descriptions given for the functional requirements of the system, which are detailed in the Appendix B.1. These aspects should be eventually covered by the set of unit, integration and acceptance tests as a whole; always taking into consideration that unit tests are using fewer resources than the others, in opposition to acceptance tests, which are the most resource-intensive of all. This means that unit testing will be preferably used to test everything that can be possibly covered by it, while acceptance tests will be left for giving feedback about the proper functioning of the system to the future clients.

6.1.1 UNIT TESTS

Unit tests are focused on checking the correct behavior of individual components when they are isolated from the rest of the system. JUnit is the testing framework provided by Play to implement unit tests for the web application, allowing testing each route in the routing system, each controller action and each template individually.

Template testing is useful to check the correct behavior of some common elements, such as the mini-cart or the breadcrumb generation shown in Figure 6.2. To assert on HTML content returned by the implementation, the library Jsoup is used to find and extract the data in a simple and clean way thanks to DOM traversal and CSS selectors.

```
@Test
public void checkBreadcrumbProductPage() {
    Category cat = mockCategory("cat", 2).get(1);
    Category par = cat.getParent();
    Product prod = mockProduct("prod", 1, 0, 0);
    Content html = views.html.helper.breadcrumb.render(cat, prod);
    assertThat(contentType(html)).isEqualTo("text/html");
    Document d = contentAsDocument(html);
    assertThat(d.select(".breadcrumb > *").size()).isEqualTo(4);
    assertThat(d.select(".breadcrumb >.step").size()).isEqualTo(3);
    assertThat(d.select(".breadcrumb >.step:eq(0) a").text()).isEqualTo("Home");
    assertThat(d.select(".breadcrumb >.step:eq(1) a").text()).isEqualTo(par.getName());
    assertThat(d.select(".breadcrumb >.step:eq(2) a").text()).isEqualTo(cat.getName());
    assertThat(d.select(".breadcrumb >.active").size()).isEqualTo(1);
    assertThat(d.select(".breadcrumb >.active").text()).isEqualTo(prod.getName());
}
```

Figure 6.2: Source code to test the breadcrumb view, in ViewsTest.java.

The routing system is tested by verifying that all the required routes are found by the system, while the controller actions are tested setting up different requests and checking the response is as expected. The test should analyze the HTTP status code, content type and charset of the response are correct, as well as to verify that the content is well formed and displays correct information.

In Figure 6.3 below it is shown how the Sphere class is mocked with certain products to be returned by the product request. Then the action is called with a category slug and requesting the second page. The type of result is verified and its content, in this case JSON data, is then analyzed to check the required products are sent. After that, the search request prepared for the backend is examined with Mockito, verifying that the correct filters were applied and the second page was actually requested before mocking the response. Unfortunately the former case is still not possible because the *FilterExpression* is lacking an overridden *equals* method to compare them, so currently the test would always fail.

```
@Before
public void mockSphere() {
    running(fakeApplication(), () -> { sphereTestable = new SphereTestable(); });
}

@Test
public void pagingProducts() {
    running(fakeApplication(), () -> {
            List<Category> categories = mockCategory("cat", 1);
            Category cat = categories.get(0);
            mockProductRequest(15, 1, 10);
            Result result = callAction(
                    routes.ref.Categories.listProducts(cat.getSlug(), "", 2));
            assertOK(result, JSON_CONTENT);
            JsonNode data = contentAsJson(result);
            assertThat(data.get("product").size()).isEqualTo(5);
            FilterExpression expression = new FilterExpressions
                    .CategoriesOrSubcategories(categories);
            verify(sphereTestable.searchRequest).filter(expression);
            verify(sphereTestable.searchRequest).page(1);
    });
}
```

Figure 6.3: Source code to test the product list returned by the action controller, in CategoriesTest.java.

6.1.2 *Integration tests*

Integration tests are in charge of checking that the previously tested components are correctly working together. To prove that, Play Framework provides Selenium WebDriver to start a test browser and FluentLenium to easily write tests with it, wrapping the Selenium WebDriver in a convenient API with a fluent interface. These tests will validate the proper functioning of the components when they are integrated in the web application.

Unlike unit testing, here the tests only need to verify that the response is the expected when an action is requested. The content is not checked, as this should be already verified by the unit tests. Figure 6.4 below shows an example of how the test browser navigates to the home page and there clicks on a category, which displays the category page. In this case the class used in the body element, as well as the title, are used to check whether the response is correct, but other ways are valid as well, such as checking the URL. The navigation related logic have been separated from the testing logic, so that tests are better isolated and the navigation code is reused.

```java
@Test
public void showHome() {
    running(testServer(3333, fakeApplication()), HTMLUNIT, (browser) -> {
            goToHome(browser);
            assertThat(browser.$("body.home").isEmpty()).isFalse();
    });
}

@Test
public void showCategory() {
    running(testServer(3333, fakeApplication()), HTMLUNIT, (browser) -> {
            goToCategory(browser);
            assertThat(browser.$("body.category").isEmpty()).isFalse();
            assertThat(browser.title()).isEqualTo("cat1Name");
    });
}


public void goToHome(TestBrowser browser) {
    browser.goTo("http://localhost:3333");
}

public void goToCategory(TestBrowser browser) {
    goToHome(browser);
    browser.$("#link-category-cat1Slug").click();
}
```

Figure 6.4: Source code of some integration tests, in IntegrationTest.java.

6.1.3 ACCEPTANCE TESTS

Acceptance tests are ensuring that the main requirements agreed for the project are met in the current version. So they need to prove that it is possible for a user to achieve at least the main goals for which he is using the web application, reason why they will be covering the top-level use cases described in the early section Use Case Model 3.1. Given that acceptance tests are guaranteeing the correctness of the current version, it is imperative to use real data instead of mocking it.

These type of tests need to be supervised by the client of the product, usually non-technical people. This makes it a requirement to be easy to understand them by using a plain language to define the rules. Cucumber is used in this project as a tool to write these acceptance tests, allowing to write the rules in plain text (see Figure 6.5), while describing the technical details of each rule in Ruby in a separated file.

```
Feature: Browse catalog
  In order to buy products
  As a customer
  I want to browse the catalog and save those products of my interest

  Scenario: Add a product to the cart successfully
    Given I visit the web shop
    And I select a product
    When I add the product to the cart
    And I go to the cart
    Then I have only the chosen item
    And The total price is correctly calculated
```

Figure 6.5: Example of rules to verify that browsing catalog can be achieved, in browseCatalog.feature.

Figure 6.6 shows how the rules are implemented for registering a new customer account. The query language XPath is used to select the different HTML elements in order to click on them, change their value or verify their existence. Error messages are always specified along with every action, so it becomes easy for non-technical people to identify any issue.

```
When /^I try to sign up with valid data$/ do
    click("//a[@id='link-sign-up']", "Can't find sign up page link.")
    fill_signup("Jane", "Doe", "webtests+#{@identifier}@commercetools.de", "Test123!")
    click("//form[@id='form-sign-up']//button[@type='submit']", "Can't find sign up.")
end

Then /^I am logged in$/ do
    find("//a[@id='link-log-out']", "Can't find log out.")
end
```

Figure 6.6: Example of the rules implementation for registering a new customer, in steps.rb.

6.2 USABILITY TESTS

With iterative methodologies, usability tests would be ideally performed by the end of the sprint, so that tasks for fixing any issue resulting from the test output can be assigned to the next sprint. This iteration is repeated until the reports from the tests are satisfactory, meaning that the user story is concluded.

But this system may not be convenient for most of the projects if the tests are meant to be too strict. Usually during the Sprint Review Meeting, when a member of the team is showing the tasks he managed to accomplish, the other members are participating in the demonstration by

evaluating and giving feedback, so that the feature can be improved further. This process already acts like an informal usability test every sprint.

Formal usability tests are still necessary in order to receive feedback from some controlled and prepared environments, giving a more reliable source of information. Given that this project needed to reach some usability requirements focused both on developers and end users, it was necessary to set up two different scenarios separately.

On the one hand, an API hackathon about e-commerce was organized in April 2013, called Berlin ECOMMHACK I/O. This hackathon consisted of developing an application connecting two or more of the participant platforms, amongst which was SPHERE.IO. Contestants could use the web-shop of the current project as a template to implement their applications or just as documentation of the SPHERE.IO Play SDK. As a result, two of the winning projects were based on the web-shop, developed by teams of four members average in within twelve hours.

Around September, a smaller hackathon of six hours was held internally, only for the staff of commercetools GmbH. About five teams used the web-shop as a template, achieving notable results as well. Both events and the feedback received allowed to improve the template, but also to confirm that the understability, learnability and changeability requirements expected for the current project were safely met.

On the other hand, a test was designed in order to evaluate the operability and likeability of the template by the users. Five people were selected for this test [Nie12], all with different profiles of online shopping experience and computer expertise. Their task consisted first of registering a new customer account and adding a postal address to their address book. After that, they were told to add two specific products to the cart, which later they were asked to remove or modify in the cart. Next, they had to purchase the products, using the address they previously entered.

Besides measuring the total time it took them to actually buy the items, it is most important to observe their reactions and ask them about their first thoughts regarding what they expect when they perform an operation. This will give an insight of the most common usability issues and the best way to solve them.

The results were very satisfactory and provided some issues with clear solutions. For example, most of them had problems with the lack of loading animations when AJAX is performing a request in the background, which is something that has repeatedly been postponed, although

it is practically implemented since the beginning. Some evident examples they gave were the poor feedback the system shows when adding an address or selecting one during the checkout process.

In contrast, they apparently did not have any trouble with browsing products and navigating throughout the web-shop. Especially giving very positive feedback about pagination and the checkout process, which were the main concerns of the design process. Also their average time required for purchasing an item for the first time was considerably low, around 1:50 minutes. In addition, they liked the whole look and feel of the web-shop, although some reported missing some visual aid to understand the behavior of certain elements, such as the category navigation.

In short, participants considered it very easy to operate, which can be translated as a success of the logical design of the system. The negative evaluation responds to missing features rather than bad design or implementation. The animation on AJAX operations was already planned and the implementation was always taking into account this issue, although it had never enough priority to be implemented. Also the category navigation menu is missing feedback from a designer, so it still has a very basic design, enough to be operative.

6.3 Performance tests

Constantly during the project development, the tool Chrome DevTools was used to check the performance of the web-shop, paying special attention to repeated calls or some unexpected behavior resulting from a flaw in the software. This way it was possible to detect a bug in the endless scroll for the product list, which executed repeated calls to the web application server even when there were no more product available. Also some methods with high response times could be fine-tuned with this tool.

Another tool from Google, PageSpeed Insights, is an online performance test that analyzes the web page looking for elements that may affect its fast execution, such as resources that may unnecessary block the page. This test suggested to use minified versions of the JavaScript and CSS files, which was an easy task thanks to Play Framework built-in support.

It also reported that there were too many JavaScript and CSS files being fetched before the page could even be loaded, which meant that the browser had to wait until the last file was fetched in order to allow the user take control of it. It was necessary to find a solution for this,

because in order to make the system's CoffeeScript more understandable for the developer, the code was split into several files and classes, which in some cases raised the amount of files fetched to more than ten.

The best way to face this issue was to use RequireJS, an asynchronously module and file loader for JavaScript files, that allowed to fetch a JavaScript file in the background first, and then load all its dependencies in parallel. Although it required a bit of effort to integrate with the current code, the results were very satisfactory. It is also worth mentioning that almost all third-party client-side libraries are being fetched from CDNJS, a community-driven CDN[22] for web libraries that allows to decrease the loading time considerably.

A load test was also performed against the system to check the maximum operating capacity and detect bottlenecks. As explained in the Physical Design section, both the web application and the data tier are scalable systems, although Heroku needs to be scaled up under demand. Therefore the bottleneck should theoretically be located in the web application tier. The test was executed with Blitz, a very interesting tool that allows to easily integrate load testing in the Continuous Integration process, although in this case only the online tool provided in their website is going to be used for these one-time tests.

A first test with a duration of one minute, going from 1 to 250 concurrent users, will serve to test the regular configuration of the system, which consists of a single processing unit with 512MB of memory RAM in Heroku. After repeating the same test severals time to check for anomalies, the average results show how the web application has no problems with light pages (e.g. a product detail page), although the response time is not stable, which suggests that might have problems with a bigger amount of users (see Figure 6.7).

On the other hand, heavier pages like any product list, have grave problems with memory, which starts failing with around 170 concurrent users, and then Heroku keeps serving a "503 Service Unavailable" error. This is more prominent as the number of displayed products increases, being the best example the home page where all products are listed. In this case, the memory fails before reaching 100 users.

[22] CDN stands for Content Management Network and it is a large distributed system of servers deployed in multiple data centers across the Internet, allowing to server content with high availability and performance.

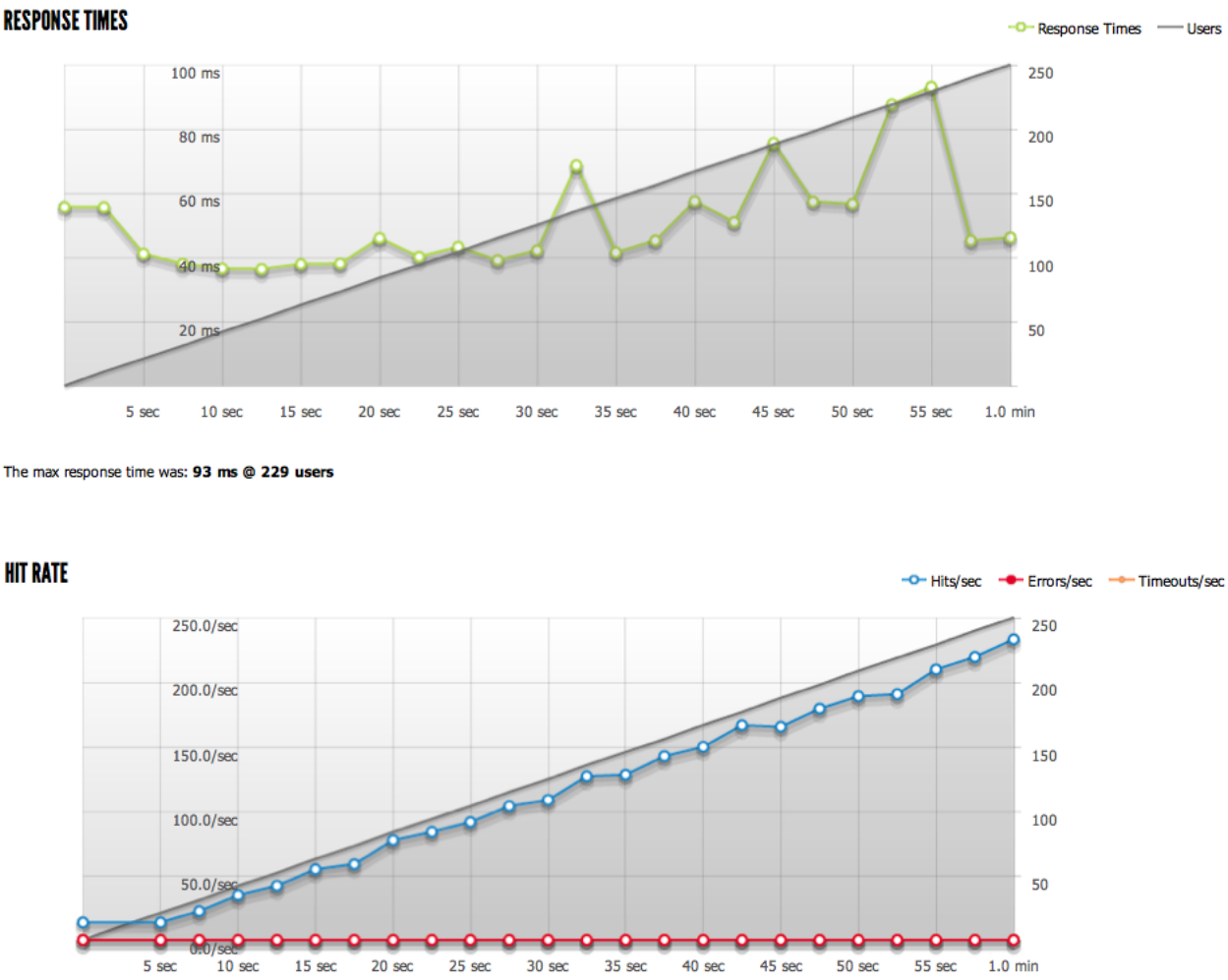


Figure 6.7: Load test results of 1 minute with 250 users against product detail page with 1 processing unit.

This problem has two solutions: either the number of processing units is increased, so the load can be distributed and units can be replaced when they fail, or the unit can be upgraded to have double memory. If the number of processing units is increased by one, having then two units, then the product detail page shows stable responses and the product list page with few products has no more memory problems, but the home page is still having trouble, this time at 170 concurrent users (see Figure 6.8).

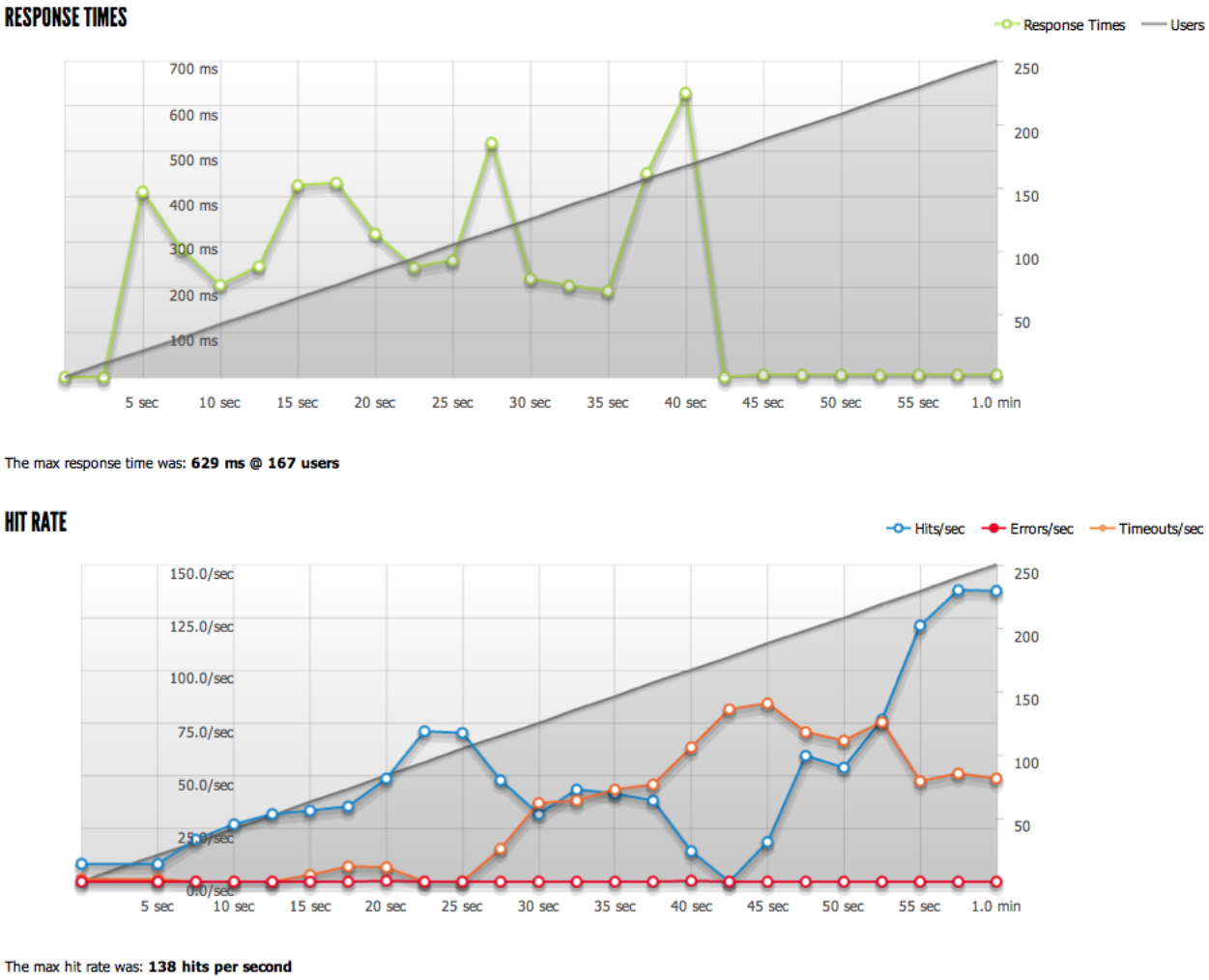


Figure 6.8: Load test results of 1 minute with 250 users against home page with 2 processing units.

Increasing up to four processing units solves the problem, but still 10% of the responses end up exceeding 1.000 milliseconds. This memory issue may be related to a memory leak from Play Framework[23] or from the same application, but in any case probably has a solution that should be found in order to increase the performance of the system.

[23] See unresolved issue for more information: http://stackoverflow.com/questions/19400493/after-updating-to-play-2-2-i-get-memory-quota-exceeded-errors-on-heroku

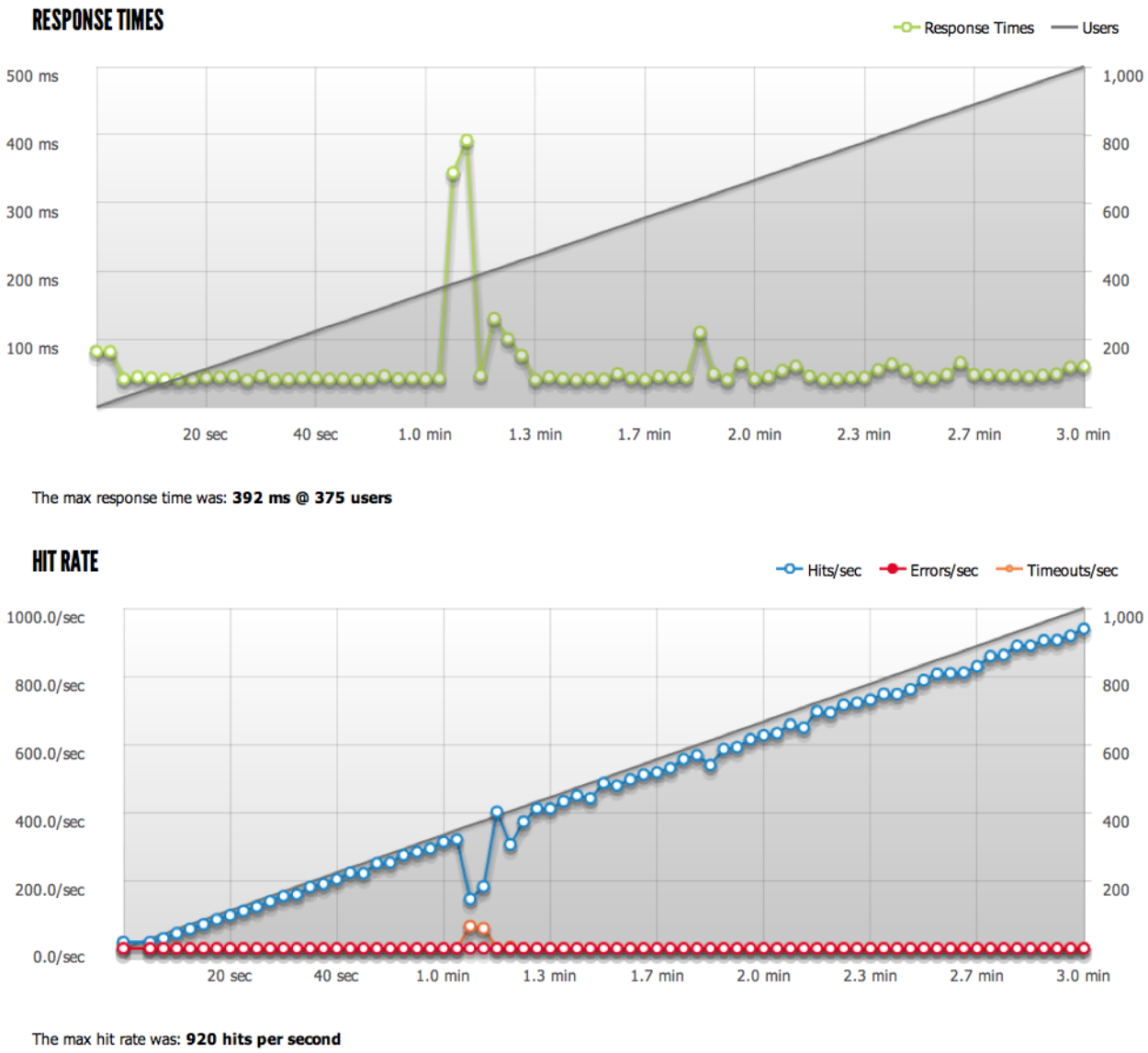


Figure 6.9: Load test results of 3 minutes with 1.000 users against category page with 4 processing units.

Lastly, a load of 1.000 concurrent users during 3 minutes was tested against the light product list page with these four processing units (see Figure 6.9). The results are completely satisfactory, with very stable responses except for one timeout period after the first minute, which could be related to Heroku reacting to the load or simply isolated network problems. Here the average response time was of 58 milliseconds, which is more or less the average of the whole web-shop when no bigger issues are affecting the performance.

7 ACTUAL PLANNING AND ECONOMICAL ANALYSIS

Now that every step of the project has been described, it is time to review the actual planning resulting from the final developing process. Analysing the deviation and the reasons behind it will help to better plan similar projects in the future (section Deviation from Initial Timeline 7.1). At the same time, knowing the total amount of hours invested allows to estimate the total cost of the project (section Economical Valuation 7.2), which helps as well for others projects, especially the development of a SPHERE.IO based web-shop that needs to be budgeted to be presented to the client.

7.1 DEVIATION FROM INITIAL TIMELINE

As it was already contemplated as a possibility during the risk analysis, it was necessary to sacrifice the constraint of time[24] of the project. The 725 total hours of work initially planned became 1.361 hours; resulting from an increase of 60 hours in implementation and 576 hours in documentation. Also, the project was postponed some months in order to finish some other projects that required a higher priority in the moment.

Moreover, several times during implementation, some malfunction or missing functionality of SPHERE.IO Play SDK had to be reported and in some cases a workaround had to be applied until a fix was provided, therefore delaying the current development. But it was necessary just one extra sprint of two weeks to deliver an acceptable product that allowed to fulfill the objectives of the project. In fact, the main responsible of the large deviation between estimated and actual planning was the drawing up of the documentation, which was expected to be two months shorter. Taking a closer look at the planning (see Figure 6.10), there are three sections that were considerably underestimated: Introduction, Design and Implementation.

[24] A project is considered to have three constraints (scope, time and cost) that cannot be optimized at the same time, at least one constraint will inevitably suffer.

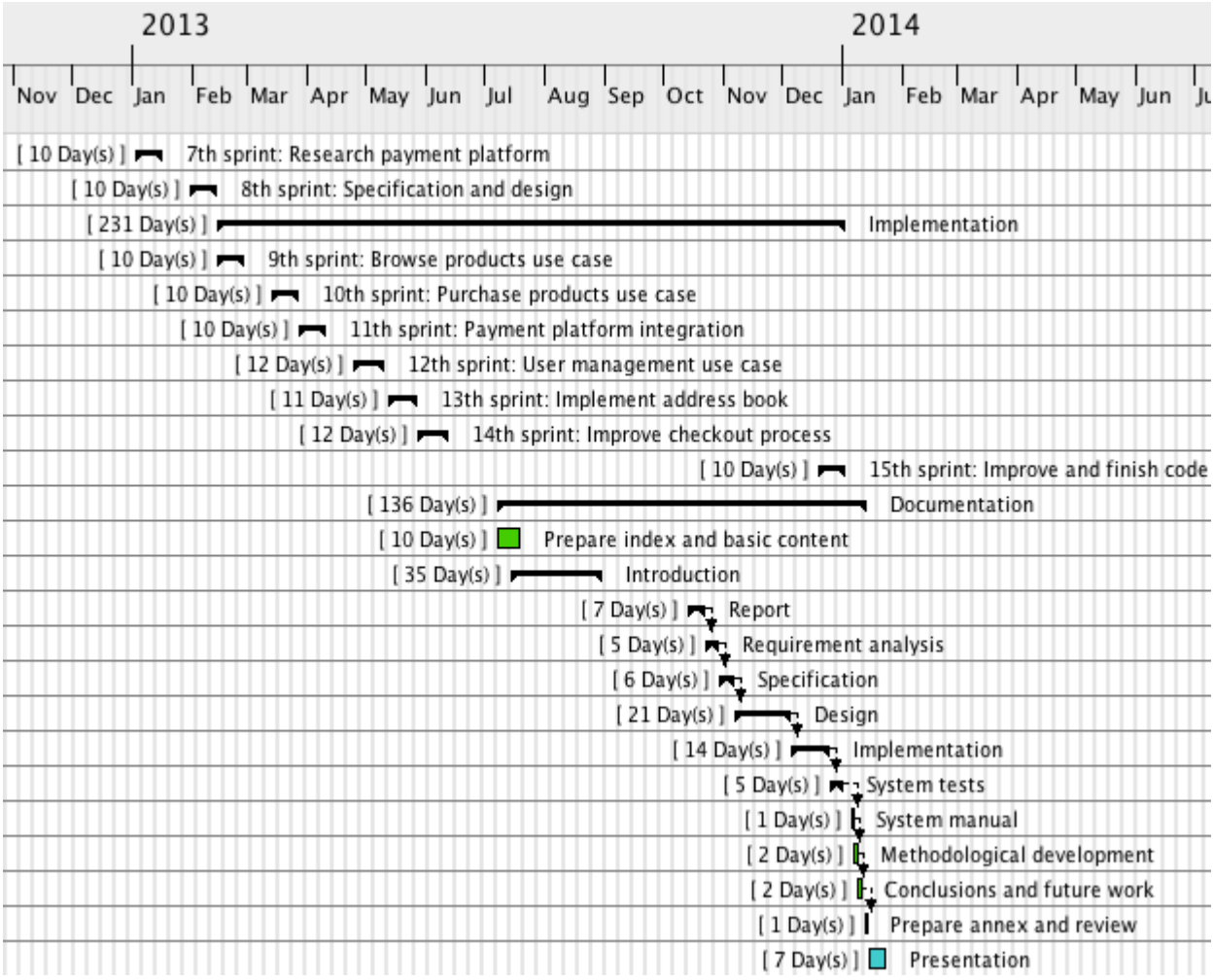


Figure 6.10: Gantt diagram of the actual timeline of the project.

The main reason is that the amount of valuable information of the project was considered to be less extensive than it actually was, especially related to technical aspects of the system. The same can be applied to the Introduction, because initially it was not contemplated the need of explaining the definition and the history of e-commerce, but eventually it was decided best to give some solid background before SPHERE.IO could be evaluated.

The scope was mostly kept the same, but many small user stories were left out of the current product due to its low priority, while others had to be released as a slimmed down version of the original feature, mainly because of SPHERE.IO Play SDK deficiencies that could not be fixed in time. Most of these cases were already mentioned, like the issues with the payment platform or the functional tests.

Product filtering and sorting was also affected, but in this case due to missing external design. While the necessary logic was fully implemented, the user interface component was never included in the template. Despite of that, the objectives are still covered since the developer can use the code to learn how to filter products or build his own product sorting reusing the code.

7.2 ECONOMIC VALUATION

Once the total amount of working hours is known, it can be estimated the total cost of the project. This calculation consists of adding up the costs of human resources, equipment and facilities used during the development process. Below are presented two tables with detailed information about the human resources used and the total cost of the project.

All the tasks described in this document and the document itself were developed by the same person during an internship at the company commercetools GmbH. The contract established a gross monthly salary of 1.200 €, for forty hours a week and twenty days of leave. Adding up 20%[25] of the employee's gross salary as social security contributions, results in the 17.280 € yearly cost for the company. Considering 232 working days during 2013[26], this resource costed around 9,31 €/hour, that applied to 1.361 hours of work ends up in 12.671 € cost for the whole project (see Table 1).

Although other IT professionals collaborated with some minor changes for the template, none of their tasks are covered by this particular project, reason why they are not considered in the economic valuation. On the other hand, the designer team was composed by a single freelance worker, but the collaboration was terminated before the project was finished, thus lasting less than one week. Also usability tests were not contemplated in the company's budget, therefore the testers were picked from the social environment and received no monetary compensation.

[25] The employee worked at the office located in Berlin, Germany, where in 2013 social security contributions of companies were around 20%.

[26] Resulting from 252 working days in Berlin for 2013 minus 20 days of leave.

Resource	Yearly cost	Hourly cost	Hours	Total cost
Intern	17.280 €/year	9,31 €/hour	1.361 hours	12.671 €
Designer		45 €/hour	24 hours	1.080 €
Total				**13.751 €**

Table 1: Detail of the human resources costs of the project.

Facilities costs, such as supplies and office rental, are shared with multiple teams and projects in the company, so their cost is not relevant for the project. As for the equipment, a 15-inch MacBook Pro laptop was used, with a processor Intel Core i7 of 2.2 GHz and 8GB of DDR3 memory, from the late 2011. The price is not known but it is guessed to have costed around 2.000 €. It was used for this project during a period of 16 months, although it was shared with other projects and therefore only 75% of the time is counted, making a total of one year. Considering that laptops are paid along three years[27] and therefore this laptop is still being paid, its cost for this project is around 667 €.

All platforms used in the project (i.e. Heroku, Paymill, Mailjet, SPHERE.IO) are permanently in trial or free version, with no associated costs. AgileZen, used for project management, was a free beta version during the development of the project. GitHub, the project hosting service, is free of charge for projects in public repositories, which is the case of this particular project. All other technologies and software were used under free licenses, most of them belonging to open source projects.

The total cost is then calculated only with the human resources and the computer cost, which results in 14.418 € (see Table 2). This number is of course just an estimation, but it is enough to give an idea of how much this project costed, especially interesting since it is not a project for a customer and thus it is not supposed to produce any direct revenue.

[27] In Germany, annual depreciation of computers is of 33,3% in 2013.

Resource	Total Cost
Human	13.751 €
Equipment	667 €
Total	**14.418 €**

Table 2: Detail of the total cost of the project.

8 CONCLUSION

E-commerce needed the web to be born in order to start being a profitable business. Nobody knows for sure what other technologies may appear to change the e-commerce world we know today, but running businesses should get ready for them beforehand. SaaS product model is an easy-to-use but powerful e-commerce solution, although it is really slow reacting to changes and it needs to be designed for a particular business model and technologies.

The new e-commerce PaaS model not only allows to give a fast answer to new e-commerce scenarios, but also gives flexibility to developers to choose their programming language and framework of preference. Furthermore, it can still emulate mostly all advantages of software solutions by distributing open source templates, like the one developed in the current project, that can be easily deployed to a cloud platform.

The future of e-commerce relies on PaaS solutions, like SPHERE.IO, to have a specialized e-commerce data backend without technology and business boundaries. A product ready to be used from any system, independently of the uncommon business process that the company follows or the experimental technology that needs to be used.

Although SPHERE.IO is still in development and improvement process, it is already a very competitive solution with many built scenarios that attest this capability of working in any environment. Starting with the fancy web-shop developed in this project, many other followed pursuing to check different shop concepts. As a direct opposite example, a traditional shop with regular pagination, filters, sorting and displaying options was developed, with additional multi language support in Italian, Spanish, Chinese and Russian, that demonstrated no issues with character encoding (see “Fedora store” in Appendix A.6 for more details).

Another example is a subscription model with detailed billing information, which was possible thanks to the easy integration with other platforms, Pactas and Paymill in this case. At the same time this shop proved how comfortable was to develop something apparently so trivial but actually so complicated with SaaS solutions as a single-product web-shop (see “I Want Donuts shop” in Appendix A.6 for more details).

Other interesting business models were implemented with SPHERE.IO, like a web-shop where customers could purchase a product and ask somebody else to pay it for them, in which case this person receives an email with a link to a webpage where to fill his billing information; a model especially thought for kids (see “PayforMe” in Appendix A.6 for more details).

All these web-shop models may seem simple to implement because the required changes do not look difficult whatsoever. But as in the case of the single-product web-shop, the reality is quite different. As mentioned before, SaaS solutions are modelled for a specified system, and anything outside this system can fail. Instead, PaaS simply supports commerce data and logic, while the system is built on top of it as the client requires. So as long as the platform supports all possible data and logic, or it can be connected to another platform that supports the missing functionality, anything can be built.

In this case, supporting a functionality does not mean literally to provide a full support, but rather to allow integrating any custom feature with the e-commerce platform. As an example, currently SPHERE.IO does not fully support discounts, but instead provides custom line items that flexibly allows the developer to add or subtract a specific amount of money from the cart. The amount has to be calculated by the developer according to the business rule applied, but the logic to integrate any discount or fee to the cart is supported by SPHERE.IO.

To put it briefly, well-built e-commerce SaaS solutions might provide full support to all the most demanded business models, but they shall unavoidably create limits on what they really support. They simply cannot cover all possible cases, especially when it comes to innovative technologies or new business models. Well-built e-commerce PaaS solutions do not require to provide any fully supported model, they just have to make sure they are not limiting what they really support. Developers should then use templates in order to reuse code or find inspiration to implement their own web-shop requirements.

During the development of this web-shop, many flaws were found in SPHERE.IO Play SDK, most of them already solved, while the biggest problems are currently in progress. These major issues have been already explained in detail in this project: the resulting system is not entirely testable and the checkout is incompatible with some payment platforms; and it can be added a certain limitation on filtering and sorting products with complex rules.

But none of these problems is structural, because all come from the SPHERE.IO Play SDK, not the platform itself. Also they respond to missing functionalities rather than design problems. This not only means that the solution is easier to apply, but also that the core of the system is well built. Moreover, given that the SDK is an open-source project, any developer can fix any bug or collaborate with missing functionalities, or simply extend it to support his needs as a parallel project.

The template developed in this project, as the first template offered by the platform, ended up being a versatile web-shop with a flexible code and design, as the usability tests results describes (see Usability Tests 6.2). It is difficult to count the number of success cases of this template, but in any case its code has been reused to build more than five web-shop templates in less than a year and even real business cases.

Some objectives were changed during the development process and others took longer than expected to finish, but the result is that the template is widely used, which makes the project a success. Even at an early stage, the template was already public and it was used to develop other systems, like it was the case in the e-commerce hackathon ECOMMHACK I/O (see section 6.2 for more details).

But this is normal, since the template will be constantly evolving at the same time it is used, including new features as the platform grows and also improving the structure as these new features are introduced. In the end, the product described in this document is no more than a snapshot of the template as it was. It is also possible that this template has already finished its lifecycle and is not being further developed, but then certainly one of the templates based in this one has taken over the leading role.

8.1 Future work

Independently of the main template being developed for the platform, there are many open issues that need to be fixed or improved, as well as new functionalities to be implemented. As a priority issue, once SPHERE.IO Play SDK provides support for functional testing, all unit and integration tests need to be completed. Missing acceptance tests must be implemented as well, and some JavaScript testing needs to be provided, due to the extensive use of it.

Another important feature missing from the initial planning is the Optile integration. At the moment of implementing the payment process, both SPHERE.IO and Optile were products that had released their first version not so long ago. They were therefore young products giving full support to only certain selected workflows, reason why they did not offer much flexibility to work together. One of the solutions that solves this incompatibility is, on the one hand, to allow an early creation of the order before the payment is executed and reverse it when it fails and, on the other hand, to improve communication between the web server and the payment platform.

Currently Optile is already offering repeated notification calls when the web server has failed to return a successful response, ensuring that the e-commerce backend has the latest payment state. This modification has already allowed to use Optile in some of the newest SPHERE.IO based web-shops successfully, but still some work must be done in order to offer it as a simple template solution.

The rest of pending features are closely related to the missing design, being the lack of filters and sorting interface components the most noticeable. Although possibly the most important is to ensure that the template is responsive to different devices and has cross-browser support. Currently it has some minimal support to small devices and it is rendered correctly in most of the newest browsers available (i.e. the latest versions of Chrome, Firefox, Opera, Safari and Internet Explorer), but the support is not guaranteed and the design still has to be applied.

Other smaller improvements for the template include elements to improve the user experience, such as the loading animation for all those components that are loaded in the background. As well as saving in the browser's history the last page of the product list that was loaded, so that the user returns to the same product list area when he uses the browser's back functionality. It may also be useful to modify the browser's URL path when the customer selects a different color in the product detail page, to make sure the user can still share the URL of the specific product variant with no need to reload the page.

Additionally, it would be also interesting to further improve performance, specially analyzing the reason why the web application is running out of memory when listing products. And of course, the template should continue to integrate features as SPHERE.IO grows, like different catalogs, internationalization, wish list and any other new functionality that can be interesting for future developers.

8.2 Personal conclusions

After explaining how I have contributed to make this project a reality, I did not want to finish this document without expressing before how this project has contributed in my academic and professional learning. This has been an exceptional year in which I had the opportunity to work in a very dynamic and innovative environment, surrounded by extremely talented and splendid people.

In this atmosphere I was encouraged to try new approaches and technologies to solve design and implementation problems. In fact, it is precisely the acquired technological knowledge I value the most of my learning, especially when compared with my previous experience, which basically consisted of traditional and outdated technologies. This obtained knowledge largely includes the complete functional testing of a web system, a pending subject in my professional and academic life.

Working with a proper development environment was also a gratifying experience that I never had the chance to put in practice before. I enjoyed as well getting to know agile methodologies from inside instead of learning the theory from a book. And not only the methodology, but also the work philosophy of the company, which can be simply summarized with the feeling of joining a professional team that cares about doing things right from the start, valuing quality over quantity.

And of course, the development of this template became a very challenging and motivational project to me, with a concept completely different from those I have developed before. It was particularly a pleasure to work with requirements that involved a more humanistic approach of the solution, such as user and developer experience. And yet it was quite surprising how technological these requirements can be.

BIBLIOGRAPHY

[Ald09] M. Aldrich. Finding Mrs. Snowball. *Michael Aldrich Archive*, 2009.

[Ald10] M. Aldrich. E-commerce, e-business and online shopping. *Sussex Enterprise Magazine*, 2010.

[Ald11] M. Aldrich. Online shopping in the 1980s. *Annals of the History of Computing, October-December 2011 edition, volume 33, number 4*, pages 57-61, 2011.

[BB05] J. Barlow and M. Breeze. Teleshopping for older and disabled people: an evaluation of two pilot trials. *Joseph Rowntree Foundation*, 2005.

[Ber00] T. Berners-Lee. Weaving the Web: the original design and ultimate destiny of the World Wide Web by its inventor. *HarperBusiness, 1st edition*, 2000.

[Con03] J. Conallen. Building Web applications with UML. *Addison-Wesley, 2nd edition*, 2003.

[GWL11] S. Guo, M. Wang and J. Leskovec. The role of social networks in online shopping: information passing, price of trust, and customer choice. *Stanford University*, 2011.

[Kha04] A. Khan. A tale of two methodologies: heavyweight versus agile. *University of Melbourne*, 2004.

[KRSS12] S. Koulayev, M. Rysman, S. Schuh and J. Stavins. Explaining adoption and use of payment instruments by U.S. consumers. *Harvard University*, 2007.

[MERJ13] S. Mulpuru, P. F. Evans, D. Roberge and M. Johnson. US mobile retail forecast, 2012 to 2017. *Forrester*, 2013.

[Nem11] R. Nemat. Taking a look at different types of e-commerce. *Department of IT, Al-Azhar University. World Applied Programming*, 2011.

[OG13] M. O'Grady. Forrester research online retail forecast, 2012 to 2017 (Western Europe). *Forrester*, 2013.

[Oli10] E. de Oliveira. Um estudo de caso de utilização da WAE para UML em aplicações GWT. *Adilson Vahldick*, 2010.

[SS13] K. Schwaber and J. Sutherland. The Scrum Guide, the definitive guide to Scrum: the rules of the game. *Scrum.org*, 2013.

[TNW01] P. Tarasewich, R. C. Nickerson and M. Warkentin. Wireless/mobile e-commerce technologies, applications, and issues. *Seventh Americas Conference on Information Systems*, 2001.

[Akr11] G. Akrani. What is commerce? Meaning and importance of commerce. *Source: http://kalyan-city.blogspot.com/2011/03/what-is-commerce-meaning-and-importance.html [Visited on August 16th 2013]*

[Bas12] V. Basavaraj. The client-side templating throwdown: mustache, handlebars, dust.js and more. *Source: http://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more [Visited on November 20th 2013]*

[Con13] T. Conroy. Cutting the fat: when to use Ajax and when to reload. *Source: http://www.slideshare.net/Codemotion/cutting-the-fat-by-tiffany-conroy-21208849 [Visited on December 17th 2013]*

[eMa113] eMarketer. Record retail sales on smartphones, tablets take greater ecommerce share. So*urce: http://www.emarketer.com/Article/Record-Retail-Sales-on-Smartphones-Tablets-Take-Greater-Ecommerce-Share/1009595 [Visited on September 1st 2013]*

[eMa313] eMarketer. Ecommerce sales topped $1 trillion for first time in 2012. *Source: http://www.emarketer.com/Article/Ecommerce-Sales-Topped-1-Trillion-First-Time-2012/1009649 [Visited on August 29th 2013]*

[eMa413] eMarketer. Retail ecommerce set to keep a strong pace through 2017. *Source: http://www.emarketer.com/Article/Retail-Ecommerce-Set-Keep-Strong-Pace-Through-2017/1009836 [Visited on August 29th 2013]*

[eMa613] eMarketer. B2C ecommerce climbs worldwide, as emerging markets drives sales higher. *Source: http://www.emarketer.com/Article/B2C-Ecommerce-Climbs-Worldwide-Emerging-Markets-Drive-Sales-Higher/1010004 [Visited on August 28th 2013]*

[Hoa12] A. Hoar. US B2B e-commerce sales to reach $559 billion by the end of 2013. *Source: http://blogs.forrester.com/andy_hoar/12-10-18-us_b2b_ecommerce_sales_to_reach_559_billion_by_the_end_of_2013 [Visited on October 5th 2013]*

[Nie12] J. Nielsen. How many test users in usability study? *Source: http://www.nngroup.com/articles/how-many-test-users/ [Visited on December 26th 2013]*

[Off93] Office of Inspector of National Science Foundation. Review of NSFNET. *Source: http://www.nsf.gov/pubs/stis1993/oig9301/oig9301.txt [Visited on August 12th 2013]*

[Pie12] A. Pietka. E-commerce new business models. *Source: http://www.slideshare.net/AnnaPietka/e-commerce-new-business-models-14291829 [Visited on September 23th 2013]*

[Rob11] T. Robinson. How does Heroku work? *Source: http://www.quora.com/Scalability/How-does-Heroku-work [Visited on December 4th 2013]*

Appendix A DEVELOPER MANUAL

A.1 LIVE DEMO

Visit a live demo of this template at http://snowflake.sphere.io.

You can also consult the source code of this template at https://github.com/commercetools/sphere-snowflake.

A.2 SET IT UP

- Install Play 2.1.5 (http://www.playframework.com/documentation/2.1.x/Installing).
- Clone[28] sphere-snowflake project from GitHub, or download it as a zip file (https://github.com/commercetools/sphere-snowflake/archive/master.zip).
- Run "play run" command in root project directory.
- Open your browser and point it to http://localhost:9000.

A.3 CONFIGURE IT

To connect your web-shop with SPHERE.IO[29]:

- Go to SPHERE.IO Administration page (https://admin.sphere.io) and log in with an existing account or register a new account.
- Go to "Developers > API Clients" to retrieve your project data.
- To connect your web-shop with your SPHERE.IO project, modify "sphere.project", "sphere.clientId" and "sphere.clientSecret" from "conf/application.conf".

[28] For more information see: https://help.github.com/articles/github-glossary#clone

[29] For more information about SPHERE.IO see: http://www.sphere.io

To connect your web-shop with Paymill[30]:

- Go to Paymill page (https://app.paymill.com/en-gb/auth/) and login with an existing account or register a new account.
- Go to "Paymill Cockpit > My account > Settings > API Keys" to retrieve your keys.
- To connect your web-shop with Paymill, modify "paymill.apiKey" from "conf/application.conf".

A.4 Deploy it

- Install Heroku Toolbelt (https://toolbelt.heroku.com) and create an account if needed.
- Run "heroku create" command in root project directory[31], it will create a new remote for Git.
- Run "git push heroku master" to push the project to Heroku and deploy it.
- Run "heroku open" to open your deployed website in a browser.

A.5 Develop it

- Install your favourite IDE (preferably IntelliJ, Eclipse or Netbeans).
- Generate configuration files for your chosen IDE, following these instructions: http://www.playframework.com/documentation/2.1.x/IDE
- Run "play" command in root project directory.
- Inside Play Shell[32], type "clean test" for compiling and testing it.
- Check SPHERE.IO Play SDK documentation (http://www.sphere.io/dev/play-sdk.html) to further develop your application.

Have fun!

---

[30] For more information about Paymill see: http://www.paymill.com

[31] For more information see: http://www.playframework.com/documentation/2.1.x/ProductionHeroku

[32] For more information about Play see: http://www.playframework.com/documentation/2.1.x/Home

A.6 OTHER SPHERE.IO WEB-SHOPS

- **Fedora shop**, a traditional webshop with multi-language.
  Running on: http://fedora.sphere.io
  Source code: https://github.com/commercetools/sphere-fedora
  Main developers: Laura Luiz

- **I Want Donuts store**, a single-product webshop with subscription payment system.
  Running on: http://iwantdonuts.com
  Source code: https://github.com/commercetools/sphere-donut
  Main developers: Martin Koníček, Laura Luiz, Christoph Menge and Nicola Molinari.

- **PayforMe**, a web-shop with request of payment to a third person.
  Source code: https://github.com/payforme/payforme
  Main developers: Hajo Eichler, Mathias Fiedler, Gregor Goldmann, Jonas Knipper, Nicola Molinari and Christian Zacharias.

- **Kokon**, a production web-shop with BC2 and B2B support.
  Main developers: Hajo Eichler, Laura Luiz and Sven Straubinger.

Appendix B PRODUCT BACKLOG

General structure of a user story described in this document:

{User story name}: As a {role}, I want {goal}, so that {benefit} ({priority}).

B.1 FUNCTIONAL REQUIREMENTS

B.1.1 BROWSE PRODUCTS

- **List products**: As a customer, I want to list all products of the shop (1).
- **Filter by category**: As a customer, I want to see only those products that belong to a particular category and any category descendant, so that I can narrow down the list to what fits best my needs (2).
- **Filter by price**: As a customer, I want to see only those products from the product list which prices fall within a specific price range, so that I can narrow down the list to best fit my economic requirements (3).
- **Filter by color**: As a customer, I want to see only those products from the product list which main color matches any of the colors I selected, so that I can narrow down the list to best fit my liking (4).
- **Sort by name**: As a customer, I want to sort the products from the product list by their name in an ascendant or descendant order (9).
- **Sort by price**: As a customer, I want to sort the products from the product list by their price in an ascendant or descendant order (8).
- **Pagination**: The product list needs to be displayed divided into pages and the customer should be given the ability to browse through them (3).
- **Product detail**: As a customer, I want to see all information regarding a particular product and its variants, so that I can make a better decision about buying it (1).

- **Breadcrumb**: As a customer, I want to be informed of my location inside the category tree via a breadcrumb, so that it can help me to navigate and have a better understanding of the web-shop structure (6).
- **Empty list message**: As a customer, I want to be informed with an informative message when a product list request has no results (5).
- **Not found message**: As a customer, I want to be informed with an informative message when a category or product I requested cannot be found (10).

B.1.2 PURCHASE PRODUCTS

- **Add item to cart**: As a customer, I want to add a particular product to the shopping cart, so that I can buy it with the next order (1).
- **Update item in cart**: As a customer, I want to change the number of units of a particular item in the shopping cart, so that I can buy a different quantity of the product with the next order (6).
- **Remove item from cart**: As a customer, I want to remove a particular item from the shopping cart, so that I do not buy it with the next order (3).
- **Place order**: As a customer, I want to place an order, so that I can actually buy the items in my shopping cart (2).
- **Payment**: As a customer, I want to be able to pay online my orders, so that I can pay immediately the moment I buy them instead of using other possibly unpleasant billing options (4).
- **List orders**: As a registered customer, I want to see a list of my orders, so that I can see all the purchases I did in the past (5).
- **Mini cart**: As a customer, I want to be able to see my current shopping cart from any page via a so-called mini-cart, so that I can always be aware of its contents and pricing details (5).

B.1.3 User management

- **Sign up**: As an anonymous customer, I want to sign up a new customer account, so that I can place orders more easily and take advantage of many other benefits (4).
- **Log in**: As an anonymous customer, I want to log in with an existing customer account, so that I take advantage of the benefits of being a registered customer (4).
- **Log out**: As a registered customer, I want to logout from my customer account, so that nobody else can use it from the same machine (5).
- **Recover password**: As an anonymous customer, I want to be able to recover my password, so that I can log in with my account when I forget my current password (7).
- **Update account**: As a registered customer, I want to update my personal data such as the email address used (6).
- **Change password**: As a registered customer, I want to change my current password to another one of my choice (5).
- **Add address**: As a registered customer, I want to add a postal address to my address book, so that I can select it as shipping or billing address when placing an order (5).
- **Update address**: As a registered customer, I want to update the data of a particular postal address from my address book, so that it corresponds to my current situation (6).
- **Remove address**: As a registered customer, I want to remove a particular postal address from my address book, so that I cannot longer select it when placing an order (5).

B.2 Non-Functional Requirements

B.2.1 Usability

- **Understandability**: As a developer, I want to identify and understand a particular business logic code of the template in less than 1 person-hour (2).
- **Learnability**: As a developer, I want to learn how the application is structured and how I can modify and extend it to build my own web-shop in less than 8 person-hour (3).

- **Operability**: As a customer, I want to learn how to purchase an item for the first time in less than 5 minutes (7).
- **Likeability**: The template should achieve an average of 7 out of 10 points from users when asked of how much do they like it (10).

B.2.2 MAINTAINABILITY

- **Testability**: As a developer, I want to be able to test any new feature of my web application with unit, integration and acceptance tests, in less than 2 person-hour (4).
- **Stability**: As a developer, I want to be able to safely change the set of data fetched from the platform without affecting the proper functioning of the template, at least in 95% of the cases (6).
- **Changeability**: As a developer, I want to be able to perform simple changes on user interface elements of the template in less than 10 minutes (4).

B.2.3 FUNCTIONABILITY

- **Security**: The system must block any external attacker from reading or modifying sensitive information (6).
- **Compliance**: The system must avoid to store or process any payment data. (2)
- **SEO-Friendly**: As a merchant, I want my web-shop with a SEO-friendly URL structure, so that my web-shop can improve its ranking position in Internet search engines (8).
- **User-Friendly**: As a customer, I want the web-shop with a human-readable URL structure, so that I can identify the type of content before visiting the web page.

Appendix C TESTING INFORMATION

C.1 UNIT TESTING DESIGN

List of scenarios to be tested for each user story.

C.1.1 BROWSE PRODUCTS

List home products

- When the home page is requested, display all products of the shop.
- When no products found, show an informative message.

List products from a category

- When a category is requested, the chosen category will be indicated and its immediate children categories will be displayed, as well as all products belonging to that category or any descendant category.
- When the selected category does not exist, show a not found error message.
- When no products found, show an informative message.

Filter products by price

- When a price range filtering is requested in any product list page, the chosen price range will be displayed, as well as all products from the previous list (discarding any previous price filtering) whose price falls within the range.
- When minimum and maximum price are swapped, recover exchanging them.
- When invalid price provided, dismiss the price filter request.
- When no products found, show an informative message.

Filter products by color

- When a color filtering is requested in any product list page, the chosen color will be displayed, as well as all products from the previous list whose main color matches the selected color.

- When more than one color is selected at once, products whose main color matches any of the selected colors will be displayed.
- When no products found, show an informative message.

Show product detail

- When a product is requested, the chosen product will be displayed with all its information and pictures, as well as all the possible variants of that product.
- When the selected product does not exist, show a not found error message.

Show product variant detail

- When a product variant is requested, the chosen product variant will be displayed with all its information and pictures, as well as all other possible variants of that product.
- When the selected product variant does not exist, display the default variant instead.
- When the selected product does not exist, show a not found error message.

C.1.2 PURCHASE PRODUCTS

Show cart detail

- When the shopping cart is requested, display the cart contents and the price details.
- When the shopping cart is empty, show an informative message.

Add item to cart

- When a product is requested to be added to the shopping cart, add the selected variant in the cart and display the updated cart content.
- When the product is already in the cart, the quantity will be updated with the addition.
- When the selected product does not exist, show a bad request error message.

Update item in cart

- When the quantity of an item in the shopping cart is requested to be updated in the cart detail page, replace the previous quantity with the new one provided and display the updated cart content.
- When the item does not exist in the cart, show a bad request error message.
- When the new quantity is invalid, show a bad request error message.

Remove item from cart

- When a product from the shopping cart is requested to be removed in the cart detail page, remove the item from the cart and display the updated cart content.
- When the item does not exist in the cart, show a bad request error message.

Start checkout

- When the checkout process is requested to start, display an order summary (i.e. cart content and price details) and the corresponding shipping and billing forms.
- When the shopping cart is empty, display the last visited page.

Finish checkout

- When the checkout process is requested to finish, display success message and all order details (i.e. cart content, price details, shipping and billing options).
- When invalid data provided, show a bad request error message and pre-fill the forms.
- When the shopping cart is empty, display the last visited page.

C.1.3 User management

Show user profile

- When the user profile is requested, display the user data, change password and address book forms, as well as the list of orders from the user.
- When the user is not logged in, show an unauthorized error message and display login.

Do sign up

- When signing up a new user is requested, register the user and display the user profile.
- When user already registered, show a bad request error message and pre-fill form.
- When invalid data provided, show a bad request error message and pre-fill the form.

Do log in

- When logging in a user is requested, sign in with the user and display the user profile.
- When invalid credentials provided, show an unauthorized error message and pre-fill form.
- When user already logged in, display the user profile.

Do log out

- When logging out a user is requested, sign out the user and display the last visited page.
- When user already logged out, display the last visited page.

Edit user data

- When the user data is requested to be updated, edit data and display updated user profile.
- When invalid data provided, show a bad request error message and pre-fill the form.
- When the user is not logged in, show an unauthorized error message and display login.

Edit user password

- When the user password is requested to be updated, change password and display user profile.
- When invalid current password provided, show a bad request error message.
- When the user is not logged in, show an unauthorized error message and display login.

Recover password

- When the user password is requested to be recovered, send email to the address provided with a temporary link to the recovery page, where the user can insert a new password.
- When the email provided does not exist, show a bad request error message and pre-fill form.

Add address to address book

- When an address is requested to be added to the address book in the user profile page, add the selected address in the address book and display the updated user profile.
- When the address is invalid, show a bad request error message and pre-fill the form.

Update address in address book

- When an address is requested to be updated in the user profile page, replace the previous address with the new provided and display the updated user profile.
- When the address does not exist in the address book, add it to the address book.
- When the address is invalid, show a bad request error message and pre-fill the form.

Remove address from address book

- When an address is requested to be removed in the user profile page, remove the address from the address book and display the updated user profile.
- When the address does not exist in the address book, show a bad request error message.

C.2 Acceptance Tests Design

Cucumber based list of rules.

Browse catalog

- Given I visit the web shop
  And I select a product
  When I add the product to the cart

And I go to the cart
Then I have only the chosen item
And the total price is correctly calculated
When I add one more item
Then the total price is correctly updated

Checkout

- Given I visit the web shop
  And I select a product
  When I add the product to the cart
  And I go to the checkout
  And I enter a valid address
  Then taxes are correctly calculated
  And the shipping methods are listed
  When I select a shipping method
  Then shipping cost is added to the total price
  And the payment form is displayed
  When I enter valid payment data
  And I finish the checkout
  Then I have purchased the product

Check order

- Given I visit the web shop
  And I go to the signup page
  When I enter valid personal information
  Then I am successfully registered
  When I select a product
  And I add the product to the cart
  When I go to the checkout
  And I enter a valid address
  And I select a shipping method
  And I enter a valid payment data

When I finish the checkout
And I go to the user profile page
And I go to the order history
Then I have only one order
When I select the order
Then the total price is correct
And the address is correct
And the shipping method is correct
And the payment is paid

C.3 Usability Tests Results

- **Subject #1 (GK)**

  Age: 64
  Online shopping experience: Medium
  Computer expertise: Low
  Necessary time to purchase: 01:56 minutes
  Liking rate: 8/10
  Problems with:
  - Missing feedback on adding an address.
  - Optional fields in form.

- **Subject #2 (HB)**

  Age: 29
  Online shopping experience: Medium
  Computer expertise: Medium
  Necessary time to purchase: 00:49 minutes
  Liking rate: 8/10
  Problems with:
  - Missing feedback on adding address.
  - Selecting address in checkout.
  - Understanding categories listing.

- **Subject #3 (RE)**

  Age: 53

  Online shopping experience: Low

  Computer expertise: Medium

  Necessary time to purchase: 02:18 minutes

  Liking rate: 9/10

  Problems with:

  - Missing feedback on adding address.
  - Noticying address book listed in checkout.
  - Optional fields in form.
  - Noticying quantity field in cart.

- **Subject #4 (AL)**

  Age: 16

  Online shopping experience: Low

  Computer expertise: Medium

  Necessary time to purchase: 01:18 minutes

  Liking rate: 9/10

  Problems with:

  - Noticying address book.
  - Optional fields in form.
  - Noticying quantity field in cart.

- **Subject #5 (SK)**

  Age: 37

  Online shopping experience: High

  Computer expertise: High

  Necessary time to purchase: 01:12 minutes

  Liking rate: 7/10

  Problems with:

  - Missing feedback on adding address.
  - Noticying address book.

C.4 Performance Test Results

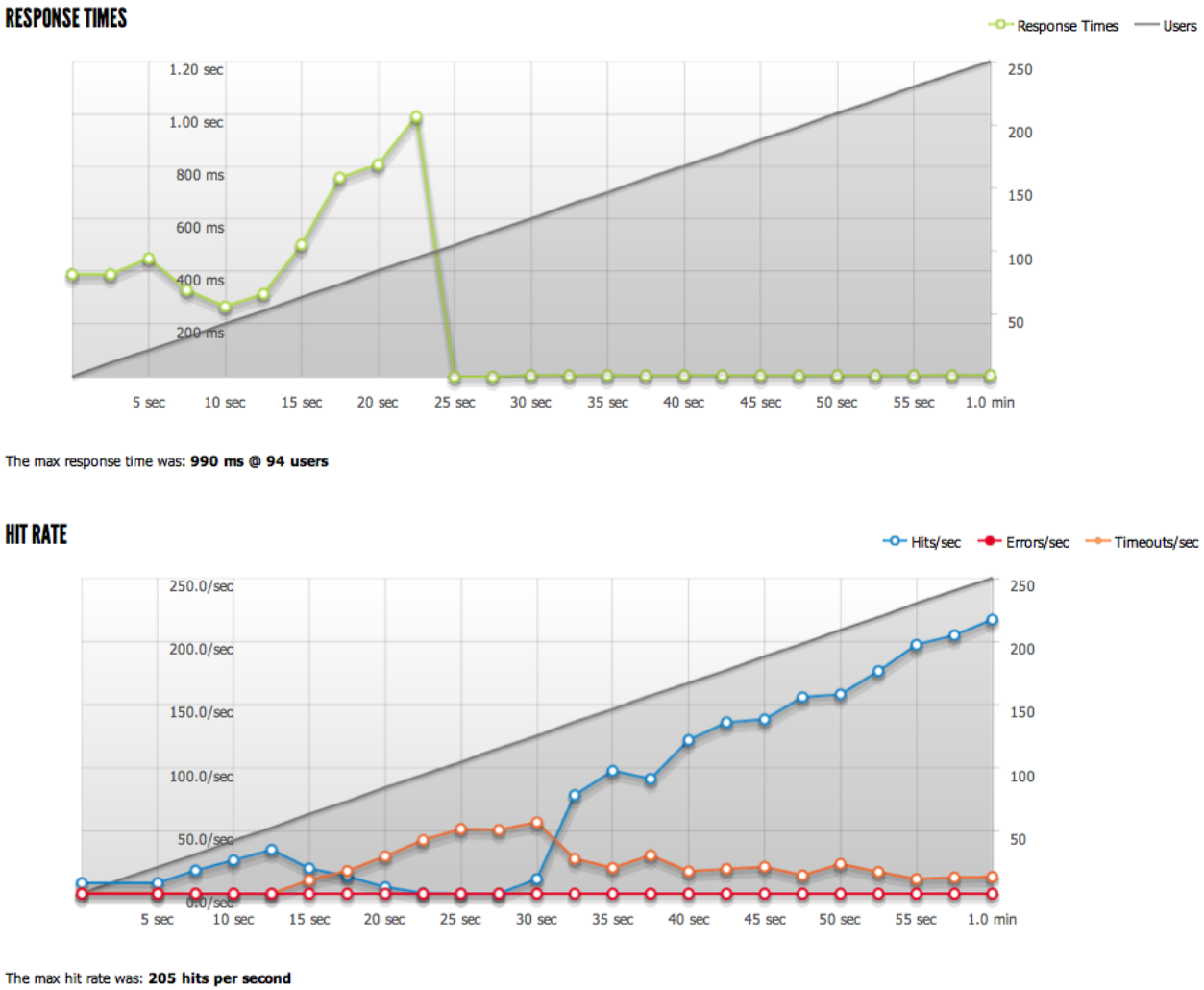


Figure C1: Load test results of 1 minute with 250 users against home page with 1 processing unit.

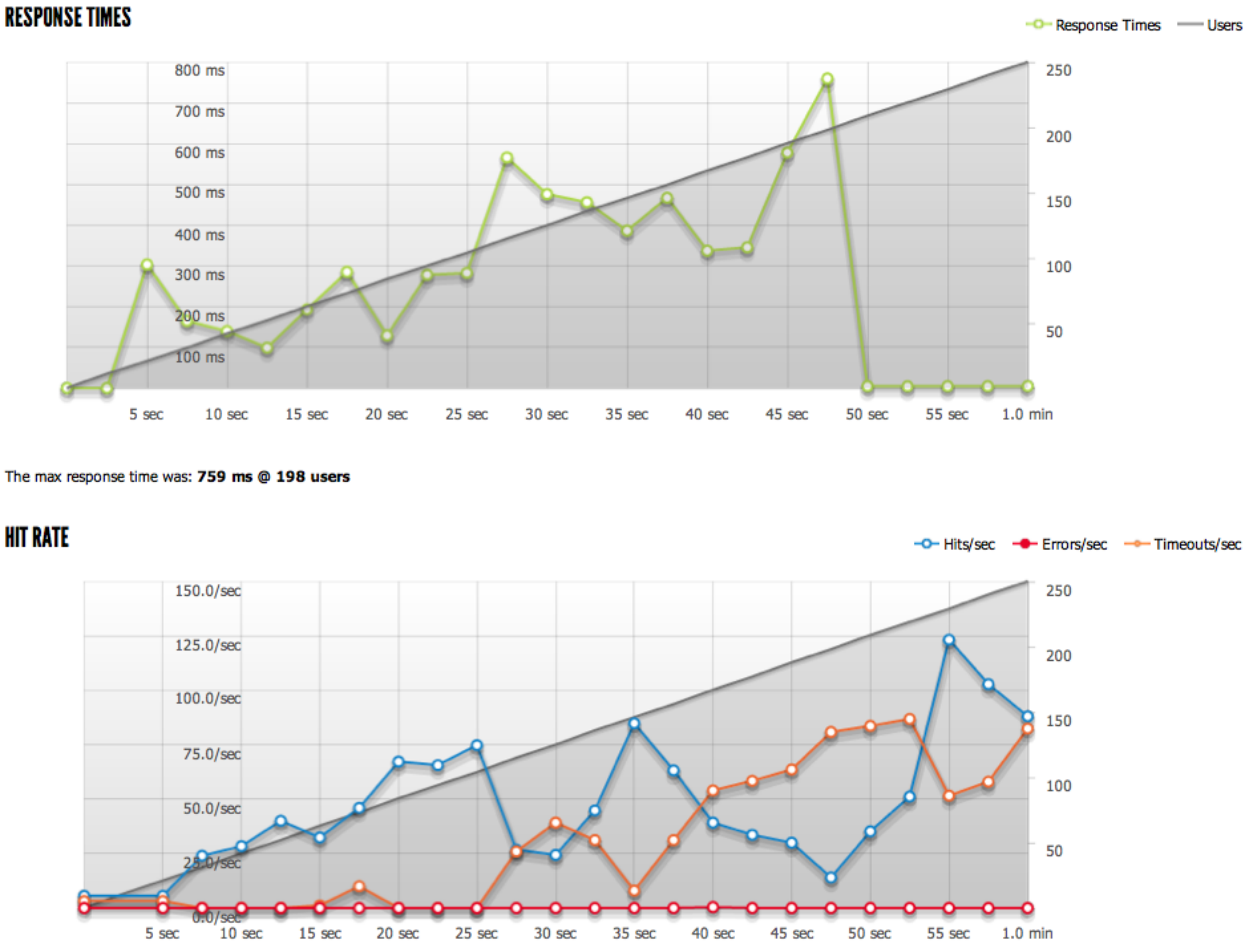


Figure C2: Load test results of 1 minute with 250 users against category page with 1 processing unit.

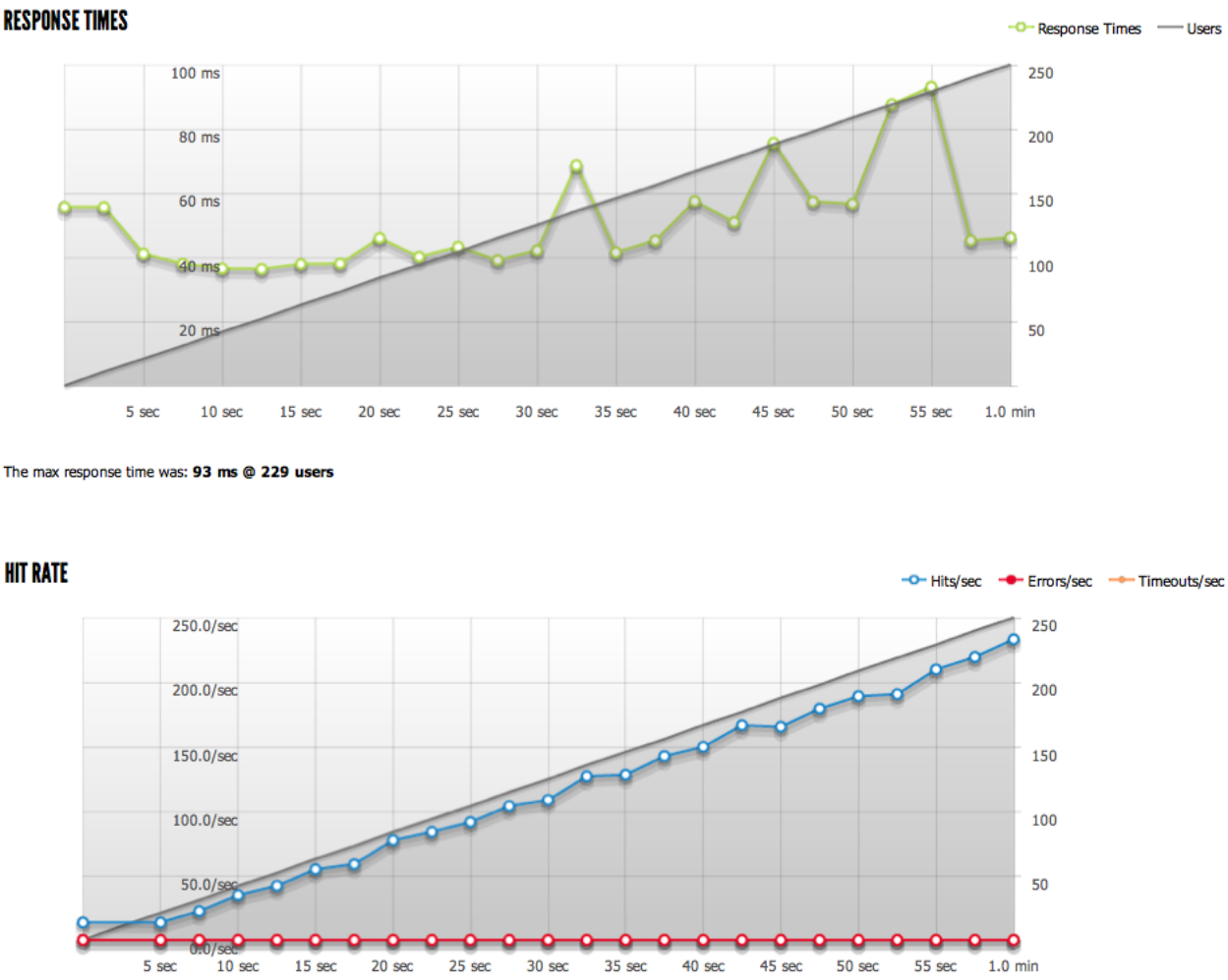


Figure C3: Load test results of 1 minute with 250 users against product detail page with 1 processing unit.

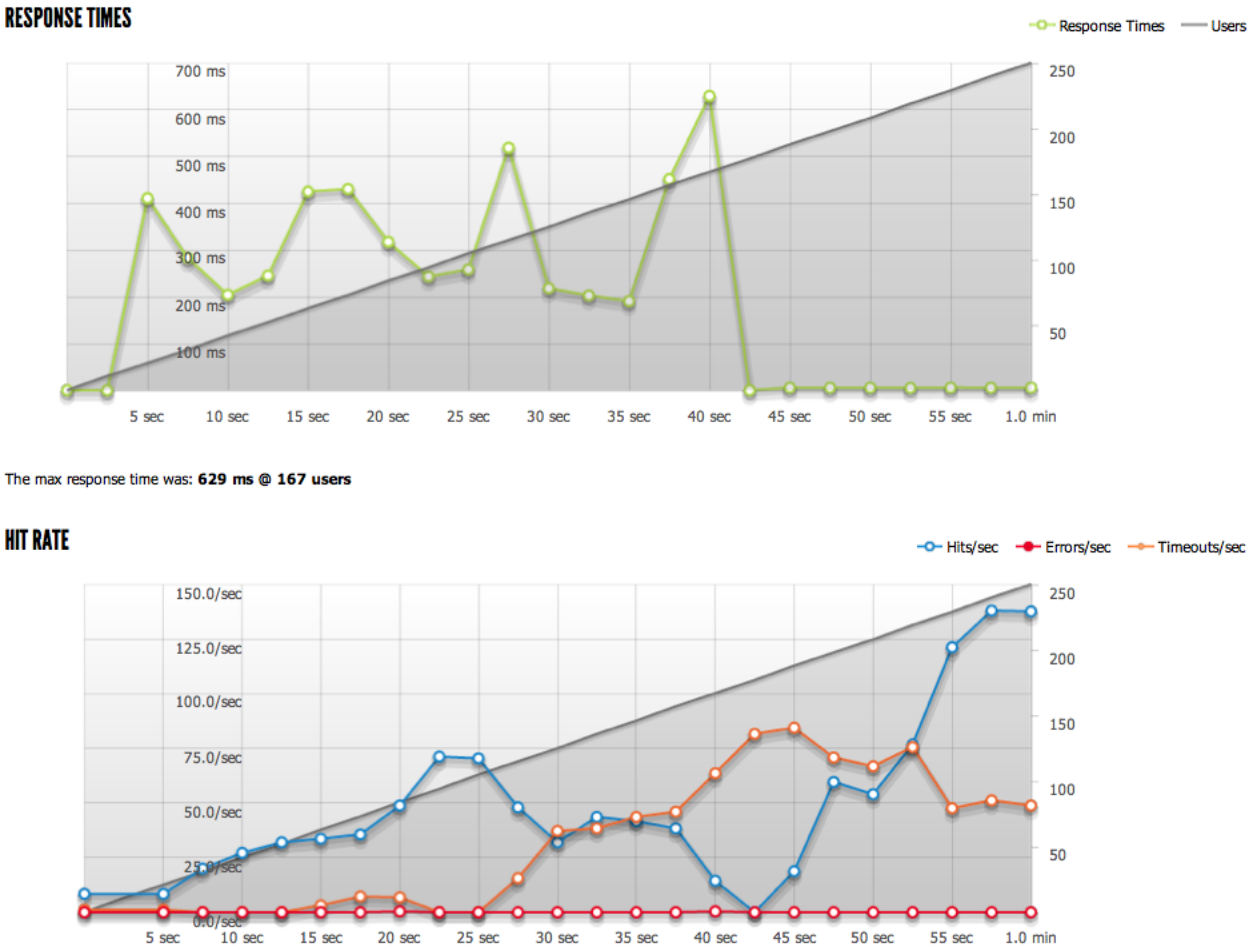


Figure C4: Load test results of 1 minute with 250 users against home page with 2 processing units.

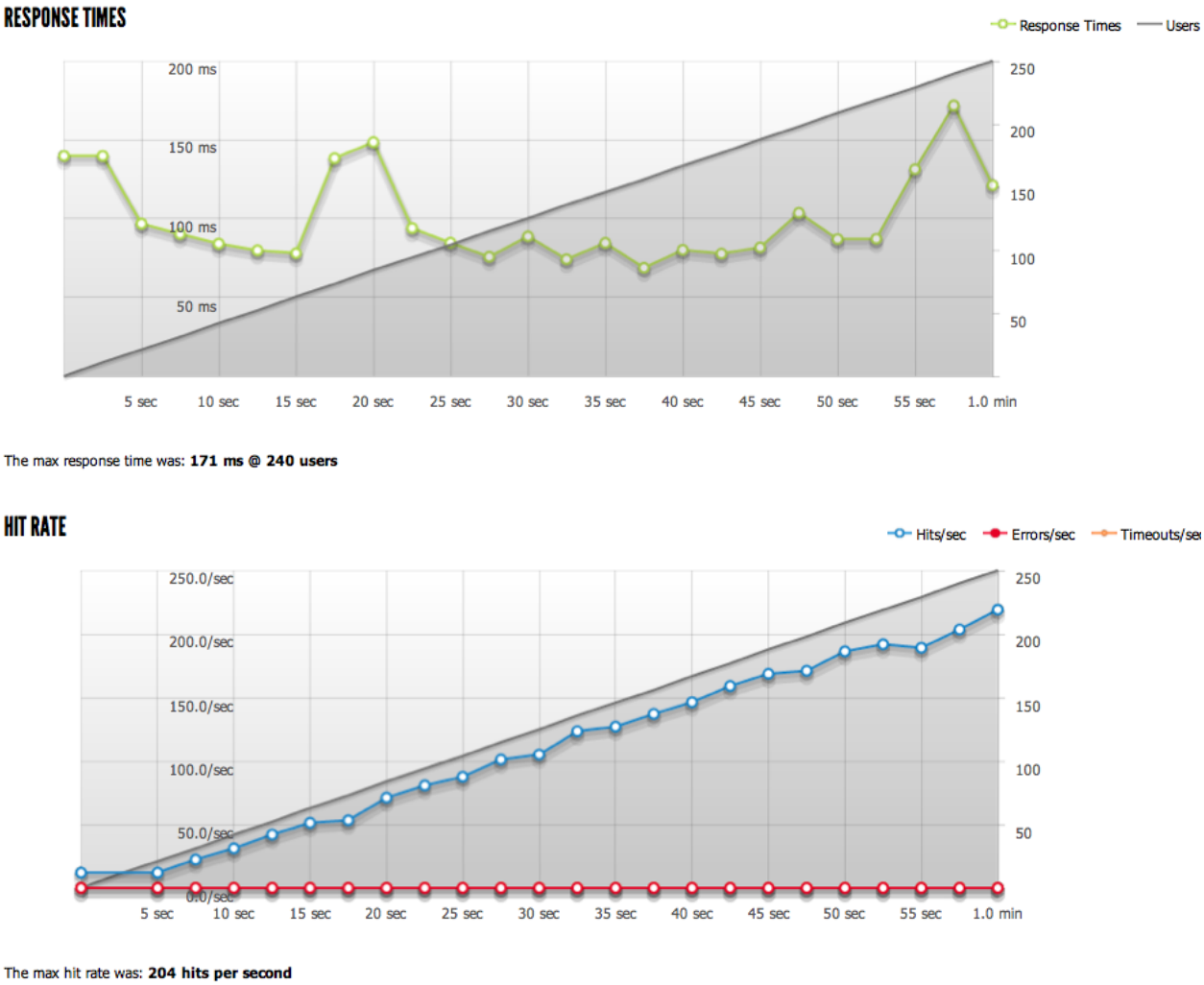
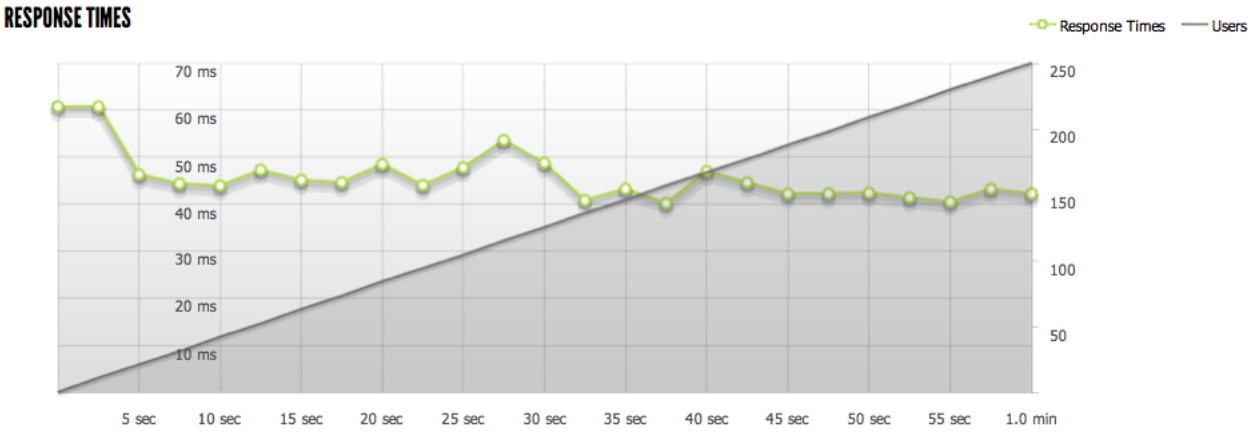
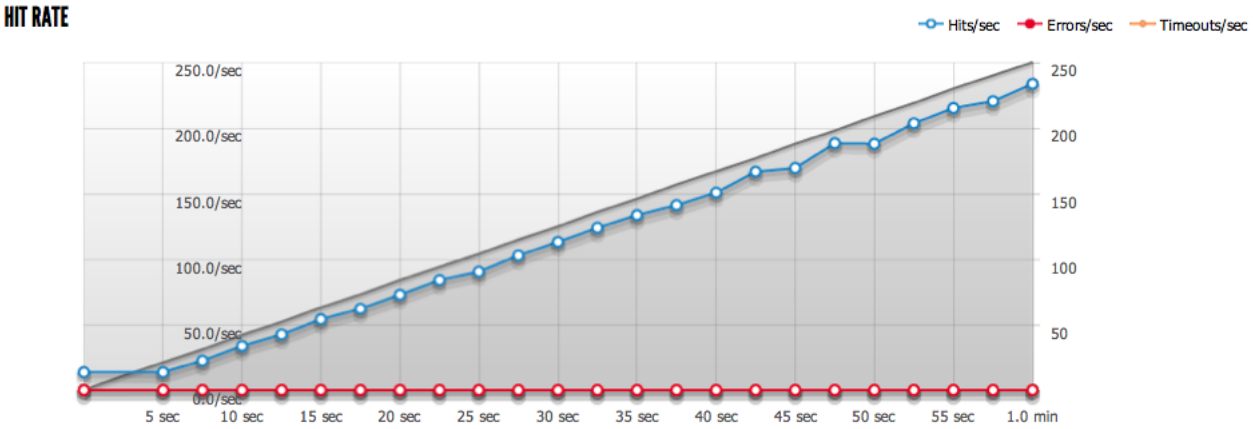


Figure C5: Load test results of 1 minute with 250 users against category page with 2 processing units.



The max response time was: **60 ms @ 11 users**



The max hit rate was: **220 hits per second**

Figure C6: Load test results of 1 minute with 250 users against product detail page with 2 processing units.

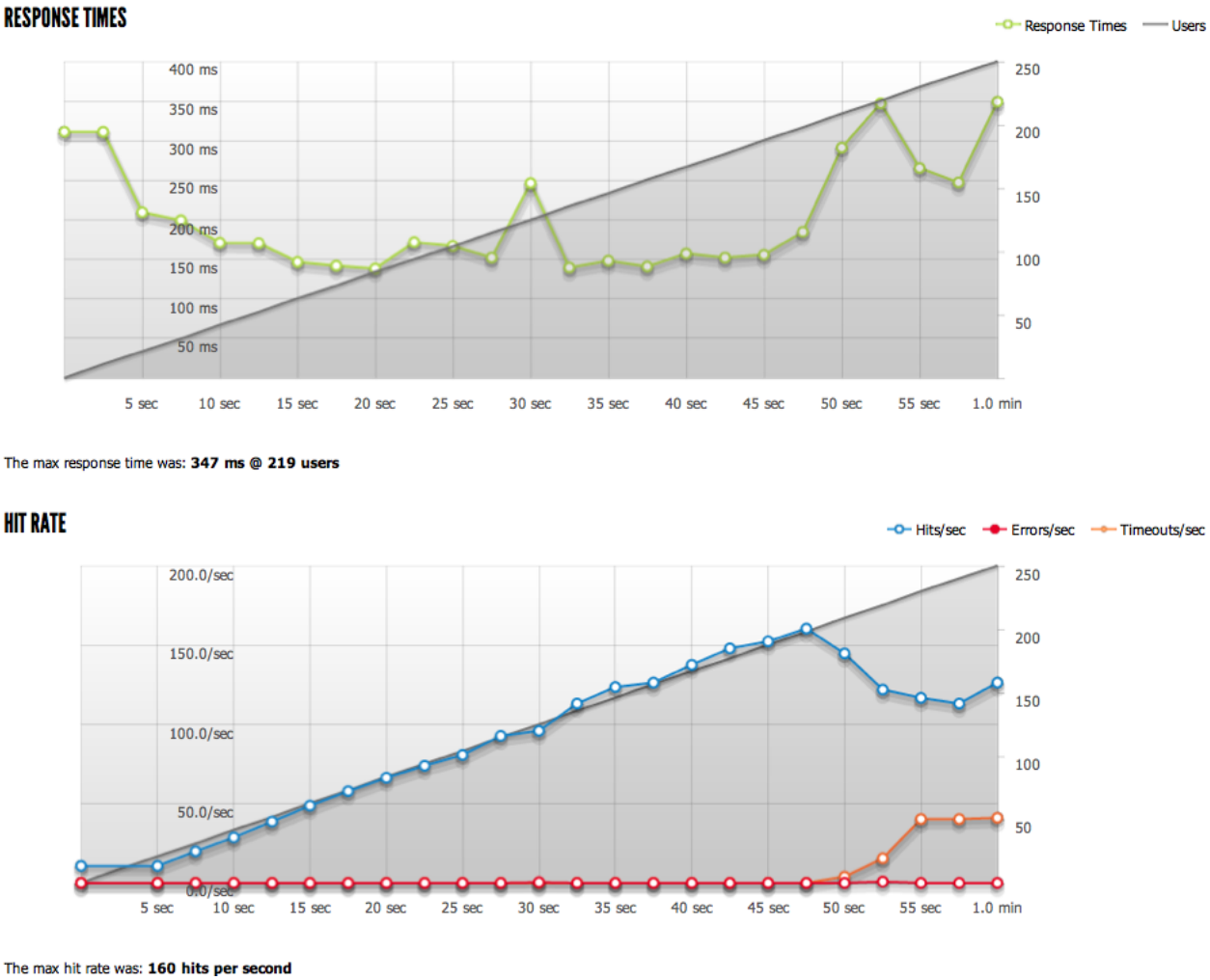


Figure C7: Load test results of 1 minute with 250 users against home page with 4 processing units.

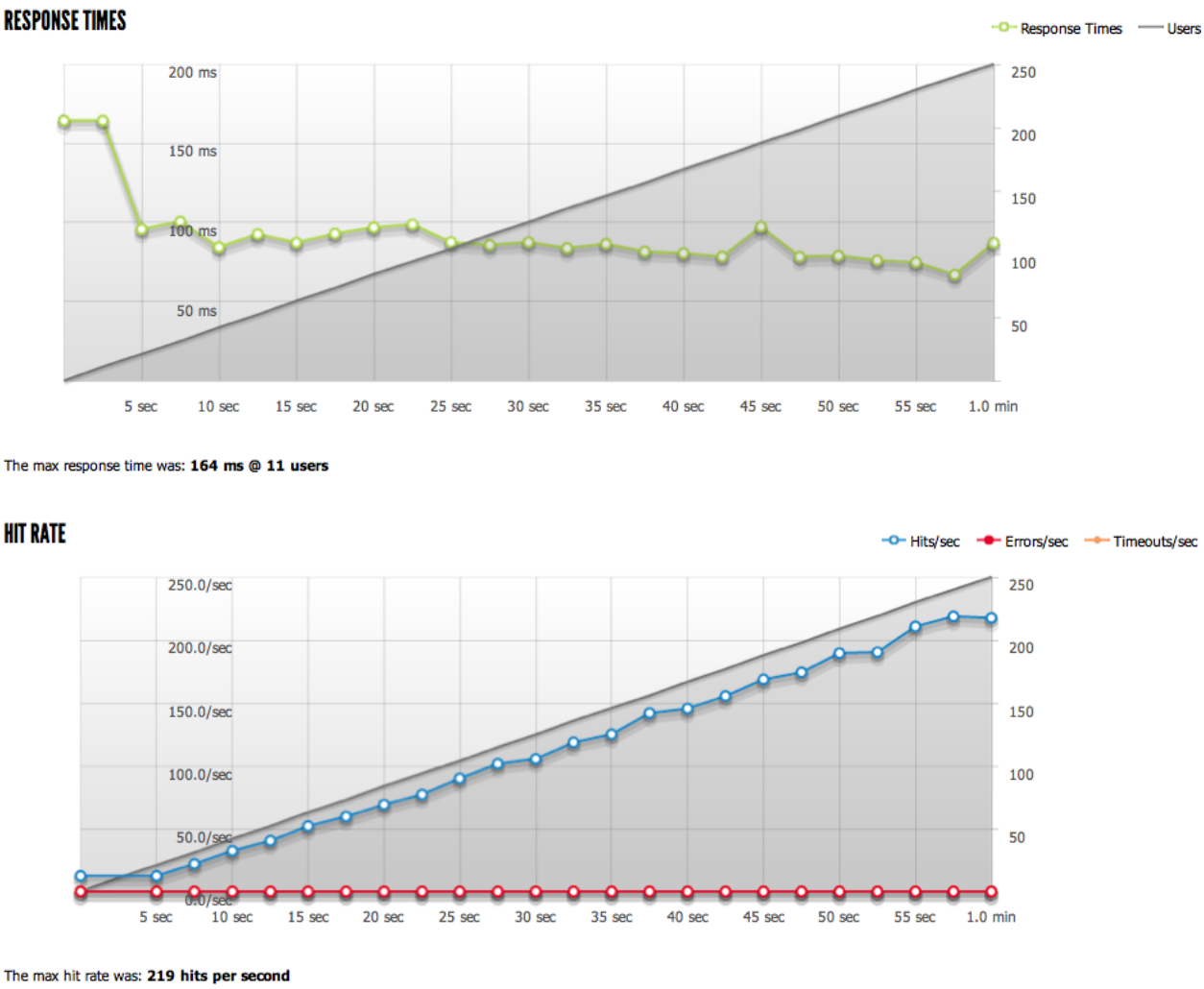


Figure C8: Load test results of 1 minute with 250 users against category page with 4 processing units.

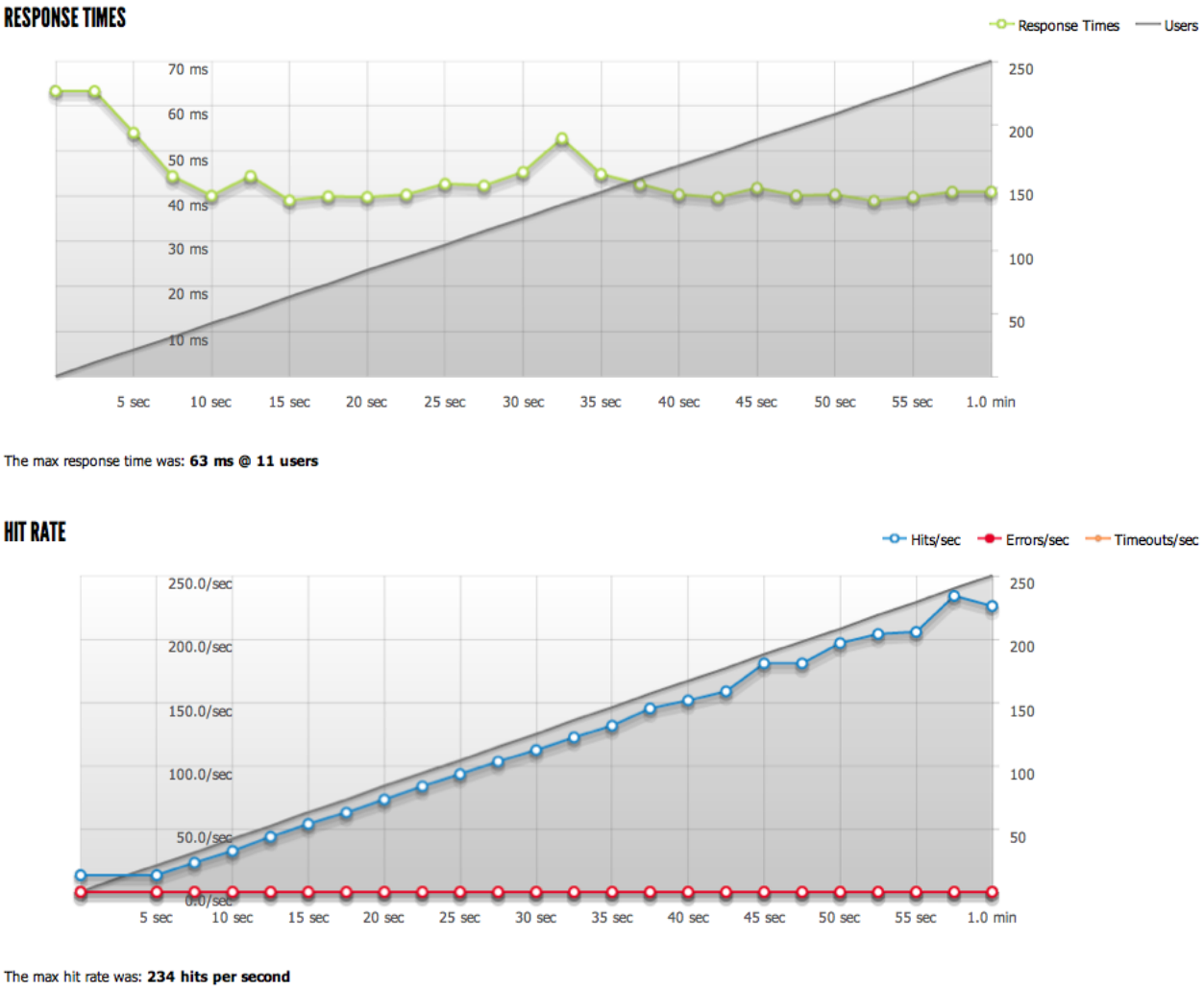


Figure C9: Load test results of 1 minute with 250 users against product detail page with 4 processing units.

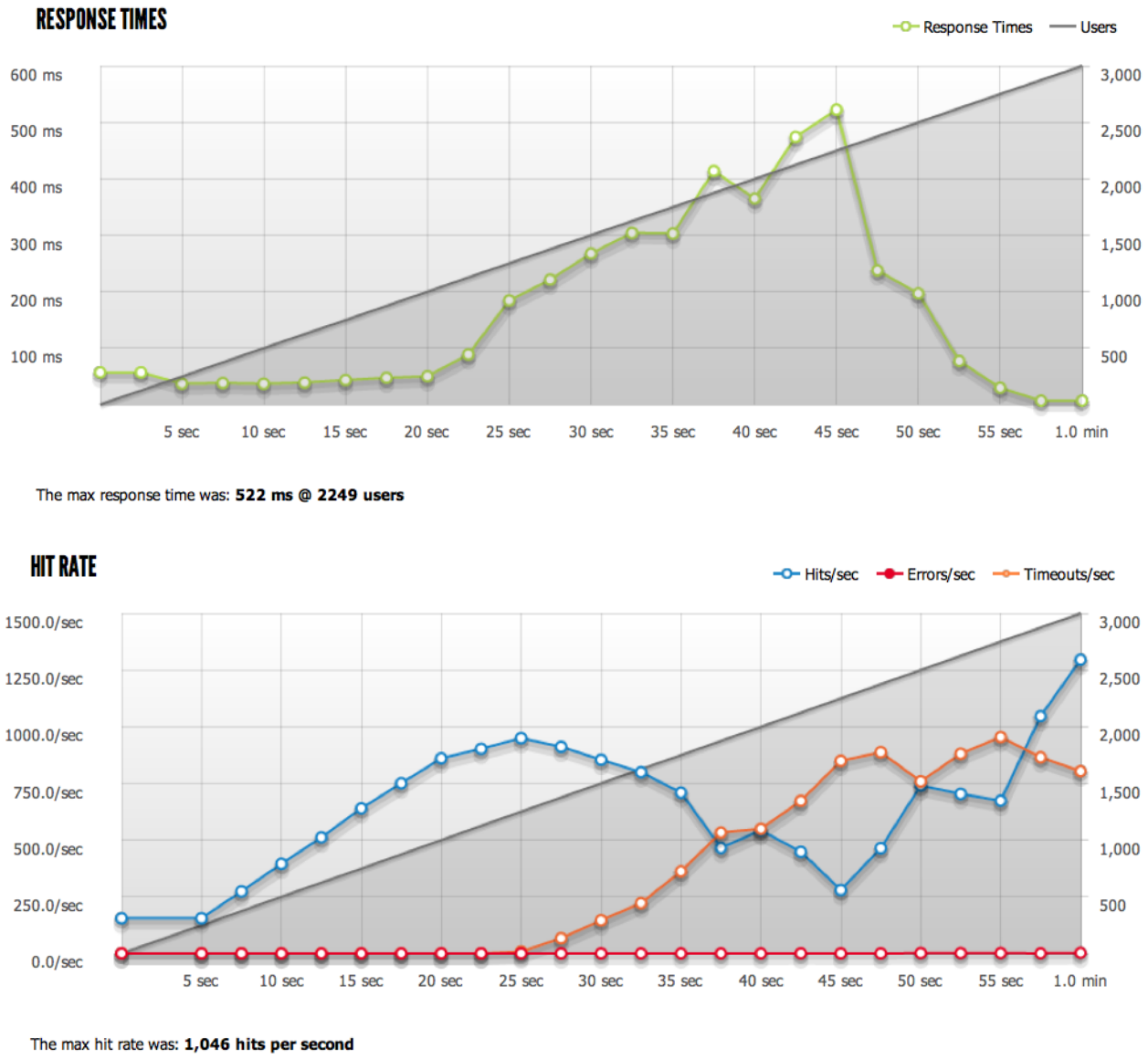


Figure C10: Load test results of 1 minute with 3000 users against category page with 4 processing units.

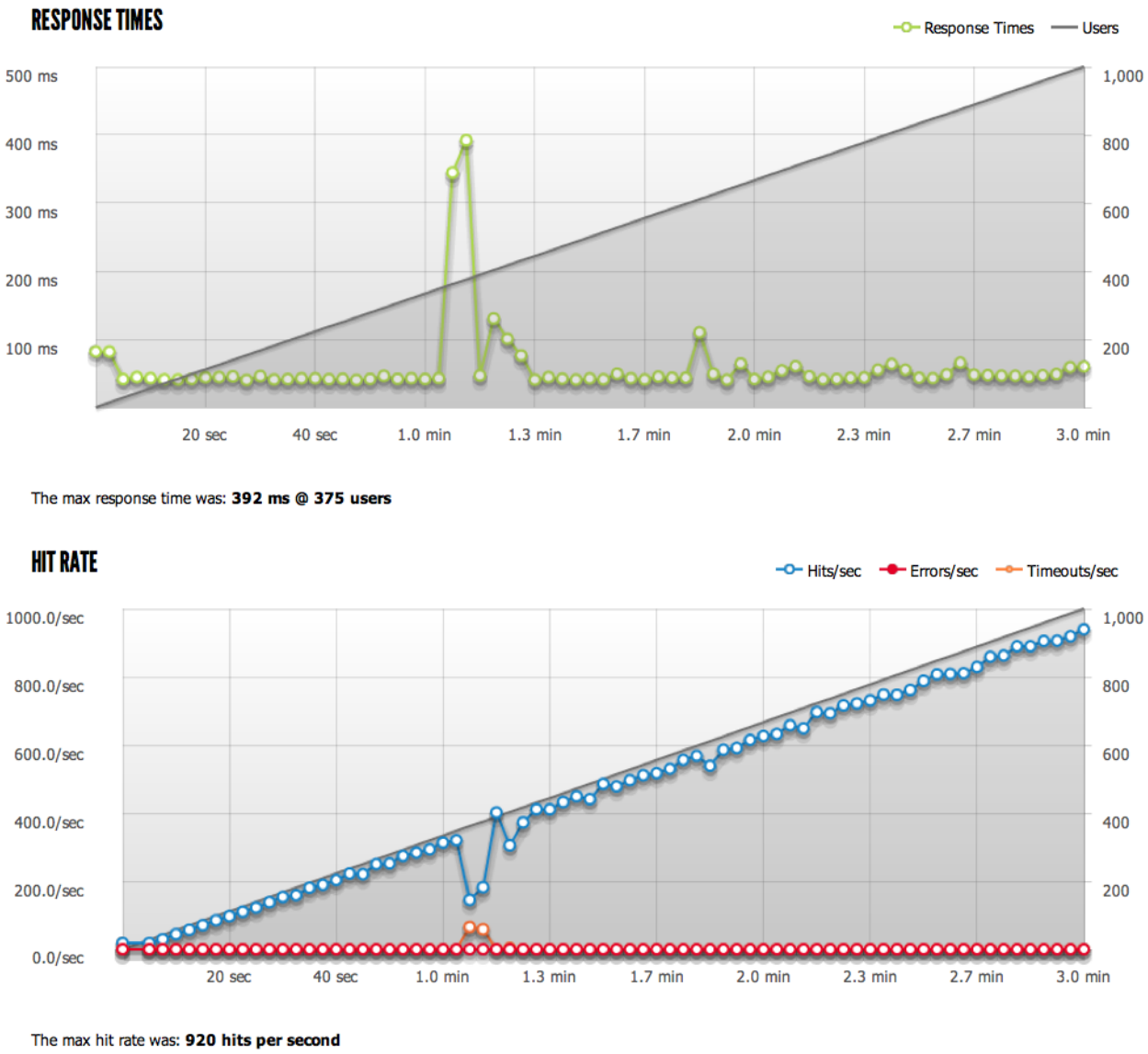


Figure C11: Load test results of 3 minutes with 1.000 users against category page with 4 processing units.