Università degli Studi *"Roma Tre"*

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale

X-commerce: Web platform to build single page e-commerce application based on HTML5 Web Components

Laureando

Baljinder Jit

Relatore

Prof. Alberto Paoluzzi

Co-relatore

Dott. Enrico Marino

Anno Accademico 2014/2015

Dedicated to my family

and ...

Contents

List of Figures

Acknowledgements

Thanks to...

Introduction

Today, the Electronic Commerce has changed the way we shop, Internet, in fact, has become a valuable communication tool for the enterprise network. The company, through the website, is able to provide you with a communication and promotion customized, personalized offer, a transaction customized, personalized assistance. Enter the network, it means coming to terms everyday, with the global market, the global consumer, competition global; to remain in the competition and gain greater visibility the company must adopt a new sales channel: e-commerce.

Electronic commerce is one of the main criteria of revolution of Information Technology and communication in the field of economy. Existence of this virtual markets, passages and stores that have not occupy any physical space, allowing access and circulation in these markets for a moment and anywhere in the world without leaving home is possible. Select and order goods that are placed in virtual shop windows at unspecified parts of the world and also are advertising on virtual networks and payment is provided through electronic services, all of these options have been caused that electronic commerce is considered the miracle of our century.

The increased availability of Internet access and the wide spread of mobile devices has allowed us to consolidate the habit of buying online by customers online already active, that have increased the share of online spending on total consumption. But what is the correct definition of e-commerce? "Commerce is the activity of buying and selling of goods and services, especially on

Figure 1: e-commerce wordle

a large scale. The system includes legal, economic, political, social, cultural and technological systems that are in operation in any country or internationally. Thus, commerce is a system or an environment that affects the business prospects of economies. It can also be defined as a component of business which includes all activities, functions and institutions involved in transferring goods from producers to consumers [Wikipedia]."

Other definition: "Interaction between communication systems, data management systems and security, which because of them exchange commercial information in relation to the sale products or services, will be available, so the definition, the main components of electronic commerce are: communication systems, data management systems and security [1]."

In the 1970s, the term electronic commerce, referred to electronic data exchange for sending business documents such as purchase orders and voices electronically. Later, with the development of this industry the term of electronic commerce is used to business of goods and services via the web. When

the first World Wide Web was introduced in 1994 as a comprehensive, many well-known researchers have been predicated this type of business "the web-based business" will became soon an important in the world economy, but it took four years that http based protocols should be widely available to users. The aim of this thesis either to analyze the main systems of e-commerce platforms and either facilitating their realization.

Therefore, this thesis is divided in two parts and organized as follows: Part one consists of five chapters. Chapter One describes the state of the art system of e-commerce and the platforms that their facilitate the realization. Chapter two, analyzes the main companies of e-commerce and the platforms that build systems. Chapter tree analyzes each technology used and describes the methodological approach of each. Fourth chapter analyzes the payment services used. Chapter five provides an overview of Single Page Application development pattern and explains pros and cons and technical functioning. The second part consists of three chapters. Chapter five describes the project as a whole showing the structure and organization pages. Chapters six is focus on specific components that have been developed, Payment Management and on all the theorical and practical concepts that are behind the ideation of the component. Finally, Chapter seven, exposes project conclusions and further implementations of the work.

Part I

Part 1

Chapter 1

E-commerce platforms

This chapter describes the main systems of e-commerce and the most popular platforms that facilitate their realization. The first section is an overview of the systems of e-commerce. The second section describes in a general way the platforms that facilitate the realization of a system of e-commerce. The third section describes what is the difference between a platform to build a system of e-commerce and the e-commerce's system. The fourth and fifth sections will focus on the main systems of e-commerce as Amzon, Ebay. The sixth and seventh sections will focus on platforms that facilitate the creation of an e-commerce as Shopify, BigCommerce.

1.1 E-commerce Overview

As already mentioned, "E-Commerce is the activity of buying and selling of goods and services, especially on a large scale. Today, most of the population, use these virtual stores to shop or simply to inquire. In fact, these systems allow for a timely basis to have information regarding the good we are seeking. In this way these systems help the customer in buying a particular good.[1]"

The e-commerce in Europe has continued to grow even if at different rates

Figure 1.1: e-commerce overview

and in different ways in different countries. Online shopping is a habit well established in Britain, Germany and France, markets that together account for 70-80% of e-commerce Europe, while it is just starting out or is growing in the rest of Europe, including countries such as Italy and Spain. The most rapid growth, however, affect the emerging economies of Eastern Europe, led by Russia, the market for which is expected to grow by up to 200% over the next three years.

In mature markets, growth was driven primarily by an increase in the frequency of purchase by the consumer and by the tendency to spend more through online channels, while in countries where e-commerce is developing growth results especially by the increase in online shoppers[2].

The cabinet is the key factor in the growth of e-commerce. The diffusion of smartphones and tablets has extended much access to the online market, even in Italy, where 29 million end users access the Internet from the mobile. Companies that have not addressed this change have had a decline in the conversion rate on its website, while those who understood the new opportunities brought by the new type of access has been able to develop

the offer of additional products and services dedicated, for example taking advantage of the geolocation of the customer. New entrants are mainly the physical stores, which saw in e-commerce for a way to expand its customer base, and producers of goods and services, they see the distribution companies increasingly as an obstacle to profitability. What are the advantages and disadvantages (risks) of a system of e-commerce?

1.1.1 Advantages and disadvantages of a system of e-commerce

The benefits of electronic commerce are general (system-wide) and specific for the seller or the buyer.

- Benefits for system:
  - It is a global phenomenon and that a potentially global market;
  - the transactions can develop throughout the day without interruption and realtime;
  - the interaction between the parties can be synchronous or asynchronous;
  - there is greater operational flexibility of relations between the parties.
- Benefits for buyer:
  - *amenity*: e-commerce stores are always open every day including holidays: just a few clicks from home or from work to buy what you want. The convenience to receive the goods directly at home is an important added value: we forget the long lines in the parking lot and in front of the chest of the crowded malls and you live a better buying experience;

  - *convenience*: a purchase through the Web is much more convenient. In addition to the discounts and promotions who flock the network there is convenience in movement (no need to move by car or public services) and in the time saved;
  - *information*: buying on the Internet allows you to calmly assess the choice of major purchases due to the large amount of information that you can easily find, to the advice and comments from other consumers, the wide range of products and alternatives;

- Benefits for seller:

  - the *flexibility* according to your needs of its commitments is easy to plan a few hours each day to devote to a new major project sales with the Internet. The mailbox collects communications and orders will be processed as soon as possible;
  - *visibility*: there is no place in the world frequented the Internet, after an initial phase of advertising the same managers are surprised of the amount of visits and contacts received. Those who already have a business and want to open a new sales channel with the Internet, should not overlook the excellent positive return in terms of image that the site produces and that also benefits traditional activity;
  - the *economy*: starting a new project sales with Internet does not require large investments and for the creation of storefront, both for advertising, both for the organization. Furthermore a good design e-commerce based on appropriate contacts with the suppliers allows to reduce to a minimum investment of stock.

Companies that sell products or services on the Internet to be successful must obtain credibility and visibility on the Web and to achieve this it is not enough to have a secure server. To get credibility companies need to

know how to build for the users of their website, a positive experience not only during the purchasing process, but also before and after, so that the user wants to repeat the experience and advice to other users. Creating a positive experience for their customers and by advertising messages and techniques of SEO companies build a reputation or image of successful enterprise. In face-to-face transactions, customers and sellers use a number of physical signals to determine that they are dealing with a trustworthy partner. Retailers can check signatures and identity cards of their clients; buyers can see the badges with the name of employees, try the goods carefully and retain proof of their purchases. On electronic networks, none of these methods is applicable. For this reason, have been developed (and are now fully available and effective) some control systems performing similar functions. The low cost of entry and the ease with which text and graphics can be copied, make it possible for almost anyone to create a website that pairs represent an organization established trading. No new reports of false virtual shops look professional, created to impersonate the Web version of existing activities, in order to illegally obtain credit card numbers. This problem has been resolved. Scammers are able to intercept transmissions. A thief can work hard to get the numbers of credit cards. A competitor or a disgruntled customer can enter an error in the company's website, in order to induce him to refuse service to potential customers or initiate other unauthorized actions. Sources intentional or accidental sometimes cause changes to the content of a communication route. User's name, credit card numbers and total currency are all vulnerable to such alteration. To this we have been developed safety systems to ensure the integrity of all the phases of the transaction. The data coming from America, are explicit: the continued growth of e-commerce is comforting. A concern, however, is the security front. In fact, the problems related to information security are increasing. You can identify two types of Internet attacks to the network:

- *Passive attacks*: Need to get the most information about the network in question, but does not have the purpose of hostile intrusion; (Ex. Eavesdropping: sniffing the information being sent across it in order to acquire the contents of transactions for subsequent analysis on personal data, or on behalf of third parties);

- *Active attacks*: as a result of the information collected by passive attacks, you can attack systems identified to implement changes to the data, lock services, obtain confidential information....

Another worrying phenomenon, in terms of information security, is surely the phishing; It is an illegal system to collect sensitive data such as information about your credit card or access to bank accounts. The vast majority of messages takes place starting from phishing e-mail addresses stolen (ie where the authors have introduced illegally) or forged to perfection (in the eyes of the end user) on the basis of a known syntax or through the use of imitators sites of companies-mirror. These threats can cause a loss of integrity of databases, a loss of profits, increased costs for security systems, a critical data loss, a loss of trade secret information and damage to corporate reputation. Systems e-commerce are the most famous Amazon, Ebay, etc..

1.1.2 Amazon

Amazon.com, Inc., often referred to as simply Amazon, is an American electronic commerce and cloud computing company with headquarters in Seattle,Washington. It is the largest Internet-based retailer in the United States. Amazon.com started as an online bookstore, later diversifying to sell DVDs, Blu-rays, CDs, videodownloads/streaming, MP3 downloads/streaming, audiobook downloads/streaming, software, video games, electronics, apparel, furniture, food, toys and jewelry. The company also produces consumer electronics—notably, Amazon Kindle e-book readers, Fire tablets, Fire

TV and Fire Phone - and is the world's largest provider of cloud infrastructure services (IaaS). Amazon also sells certain low-end products like USB cables under its in-house brand AmazonBasics.

Amazon has separate retail websites for United States, United Kingdom and Ireland, France, Canada, Germany, Italy, Spain, Netherlands, Australia, Brazil, Japan, China, India and Mexico.

Amazon also offers international shipping to certain other countries for some of its products. In 2011, it professed an intention to launch its websites in Poland and Sweden. In 2015, Amazon surpassed Walmart as the most valuable retailer in the United States by market capitalization. The company

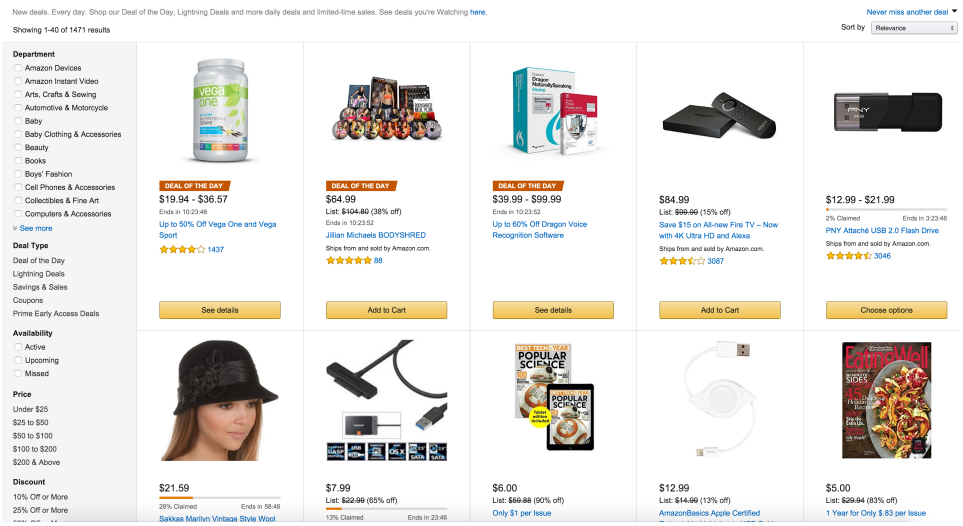

Figure 1.2: Amazon shop

was founded in 1994, spurred by what Bezos called his "regret minimization framework," which described his efforts to fend off any regrets for not participating sooner in the Internet business boom during that time.

1.1.3 Ebay

Ebay Inc. is an American multinational corporation and e-commerce company, providing consumer to consumer & business to consumer sales ser-

vices via Internet. It is headquartered in San Jose, California. eBay was founded by Pierre Omidyar in 1995, and became a notable success story of the dot-com bubble. Today, it is a multibillion-dollar business with operations localized in over 30 countries.

The company manages eBay.com, an online auction and shopping website in which people and businesses buy and sell a broad variety of goods and services worldwide. In addition to its auction-style sales, the website has since expanded to include "Buy It Now" shopping; shopping by UPC, ISBN, or other kind of SKU (via Half.com); online classified advertisements (via Kijijior eBay Classifieds); online event ticket trading (via StubHub); online money transfers (via PayPal) and other services.

The website is free to use for buyers, but sellers are charged fees for listing items and again when those items are sold. The company also makes additional money through its PayPal subsidiary which is used by sellers to collect payment for items sold. AuctionWeb was founded in California, on September 4, 1995, by French-born Iranian-American computer programmer Pierre Omidyar (born June 21, 1967) as part of a larger personal site. One of the first items sold on AuctionWeb was a broken laser pointer for $14.83. Astonished, Omidyar contacted the winning bidder to ask if he understood that the laser pointer was broken. In his responding email, the buyer explained: "I'm a collector of broken laser pointers." The frequently repeated story that eBay was founded to help Omidyar's fiancée trade Pez candy dispensers was fabricated by a public relations manager in 1997 to interest the media, which were not interested in the company's previous explanation about wanting to create a "perfect market". This was revealed in Adam Cohen's book, The Perfect Store (2002), and confirmed by eBay. Reportedly, eBay was simply a side hobby for Omidyar until his Internet service provider informed him he would need to upgrade to a business account due to the high volume of traffic to his website. The resulting price increase (from $30/month to $250)

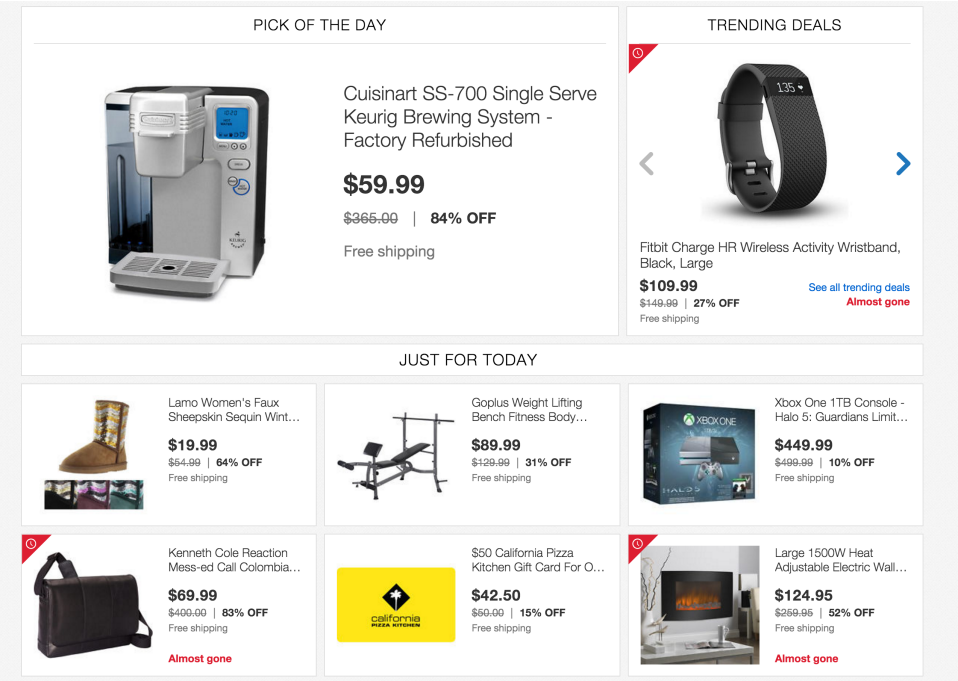

Figure 1.3: Ebay shop

forced him to start charging those who used eBay, and was not met with any animosity. It resulted in the hiring of Chris Agarpao as eBay's first employee to handle the number of checks coming in for fees. Jeffrey Skoll was hired as the first president of the company in early 1996. In November 1996, eBay entered into its first third-party licensing deal, with a company called Electronic Travel Auction to use SmartMarket Technology to sell plane tickets and other travel products. Growth was phenomenal; in January 1997 the site hosted 2,000,000 auctions, compared with 250,000 during the whole of 1996. The company officially changed the name of its service from AuctionWeb to eBay in September 1997. Originally, the site belonged to Echo Bay Technology Group, Omidyar's consulting firm. Omidyar had tried to register the domain name echobay.com, but found it already taken by the Echo Bay Mines, a gold mining company, so he shortened it to his second choice, eBay.com. In 1997, the company received $6.7 million in funding

from the venture capital firm Benchmark Capital. Meg Whitman was hired as eBay President and CEO in March 1998. At the time, the company had 30 employees, half a million users and revenues of $4.7 million in the United States. eBay went public on September 21, 1998, and both Omidyar and Skoll became instant billionaires. eBay's target share price of $18 was all but ignored as the price went to $53.50 on the first day of trading.

1.2 The platforms that build system of e-commerce - Overview

The platforms that help build a system of e-commerce are exactly systems or portals that facilitate the life of a traded that want to start your own online business.

The process of setting up an online store with such systems is very fast and efficient, because essential information is easy to fit. The next moment is the choice of the graphical presentation of the store where the trader can choose from many themes and templates available. These systems handle transparently different services that help create the store such as: the domain, payment management, organizing inventory, shipping and tracking of shipments, invoice management, etc..

The advantages and disadvantages of these platforms are mainly linked to the flexibility of the system itself. In fact, a platform for e-commerce-rich services, has more chance of being used by a growing number of major traded. Obviously, a generic platform so can not meet the needs of every type of merchant because the platform has the purpose of facilitating the realization of a system of e-commerce in a more simple possibilie. Therefore it is difficult to meet the needs of each merchant from any kind of detail. Ease of use is another key point that leads to the platform to be chosen by dealers.

The platform of e-commerce are the most famous Shopify, BigCommerce,

etc..

1.2.1 Shopify

Shopify is a Canadian commerce company headquartered in Ottawa, Ontario that develops computer software for online stores and retail point-of-sale systems.

Shopify was founded in 2004, and was initially based on earlier software written by its founders for their online snowboard store. The company reports that it has 200,000 merchants using its platform, with total gross merchandise volume exceeding $10 billion. Shopify was founded in 2004 by Tobias

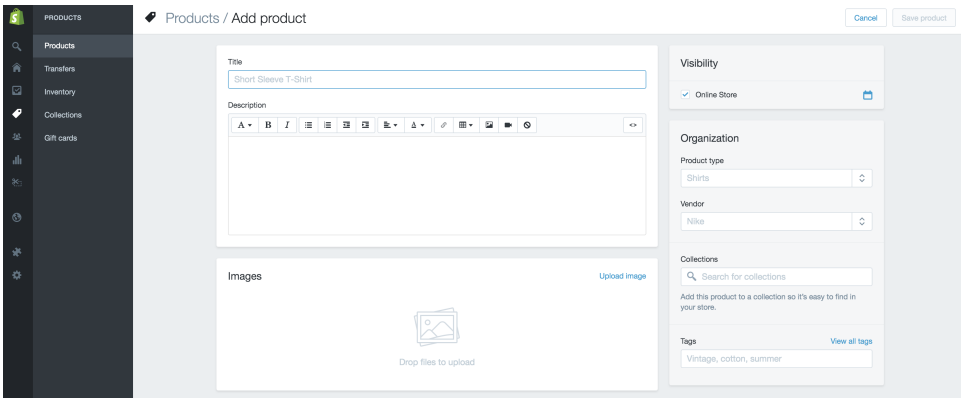

Figure 1.4: Shopify Dashboard

Lütke, Daniel Weinand, and Scott Lake after attempting to open Snowdevil, an online store for snowboarding equipment. Unsatisfied with the existing e-commerce products on the market, Lütke, a programmer by trade, decided to build his own. Lütke used the open source web application framework. Ruby on Rails to build Snowdevil's online store, and launched it after two months of development The Snowdevil founders launched the platform as Shopify in June 2006. In September 2015, Amazon announced it would be closing its Amazon Webstore service for merchants, and had selected Shopify as the preferred migration provider. Shopify's shares jumped more than 20%

upon the news.

1.2.2 Bigcommerce

Bigcommerce is a privately held technology company that develops e-commerce software for businesses. The company was founded in 2009 and has 370 employees with headquarters in Austin, Texas and additional offices in San Francisco, California and Sydney, Australia. The company reports that $5 billion in total sales have been processed by the Bigcommerce platform. Bigcommerce was founded in 2009 by Australians Eddie Machaalani and

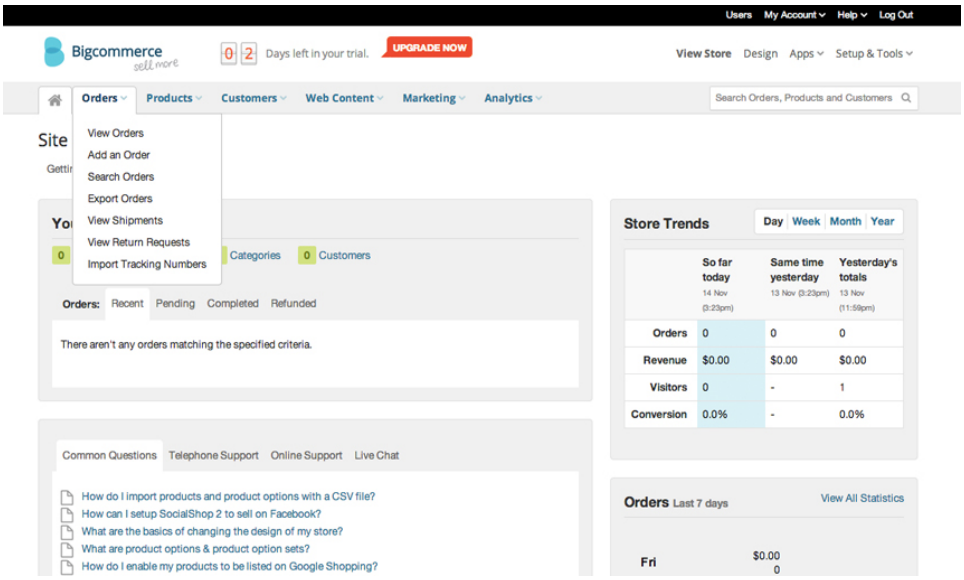

Figure 1.5: Bigcommerce Dashboard

Mitchell Harper following a chance meeting in an online chatroom in 2003. In August 2009, the two relaunched a hosted version of Interspire Shopping Cart called "BigCommerce" and opened its first U.S. office.

Bigcommerce was 100% bootstrapped until July 31, 2011, when it closed $15 million in Series A funding from General Catalyst Partners. At the time, the company announced its client count had grown 680% year over year.

In January 2012, Bigcommerce launched a $2 million integration fund for

developers, which was used to fund 31 applications in the Bigcommerce App Marketplace. The company subsequently received $20 million in Series B financing in September 2012, led by General Catalyst Partners and Floodgate Fund.

Chapter 2

Enabling Technologies

This chapter describes X-Learning enabling technologies. The first three sections concern server-side technologies: MongoDB, NodeJS and Loopback by Strongloop (an IBM company). MongoDB is a NoSQL document-oriented database management system; NodeJS is an event-driven framework to handle Javascript server sides; Loopback is a NodeJS based framework created to use and edit set of APIs. The fourth, fifth and sixth sections are related to client-side technologies: HTML5, Web Components and Polymer-Project by Google. HTML5 is a markup language aimed at web pages structuring; Web Components are a set of standards that allow for the creation of reusable widget and components in web documents; Polymer-Project provides a thin layer of API on top of Web Components and several powerful features, such as custom events, delegation, mixins, accessors and component life-cycle functions, to facilitate the creation of Web Components.

2.1 HTML5

This section provides an overview of HTML5. HTML5 is the latest version of Hypertext Markup Language, the code that describes web pages. There are actually three kinds of code: HTML, which provides the struc-

Figure 2.1: Server and client sides enabling technologies

ture; Cascading Style Sheets (CSS), which take care of presentation; and JavaScript, which makes things happen.

HTML5 has been designed to deliver almost everything it is possible to do online without requiring additional software such as browser plugins. It does everything, from animation to apps, music to movies, and can also be used to build complicated applications that run in browsers.

Moreover, HTML5 isn't proprietary, so it is completely free. It's also a cross-platform standard, which means it doesn't care whether the device is a tablet or a smartphone, a netbook, notebook or ultrabook or a Smart TV: if the browser supports HTML5, it should work flawlessly.

While some features of HTML5 are often compared to Adobe Flash, the two technologies are very different. Both include features for playing audio and video within web pages, and for using Scalable Vector Graphics.

HTML5, on its own, cannot be used for animation or interactivity, it must be supplemented with CSS3 or JavaScript. There are many Flash capabilities that have no direct counterpart in HTML5. See Comparison of HTML5 and Flash. Although HTML5 has been well known among web developers

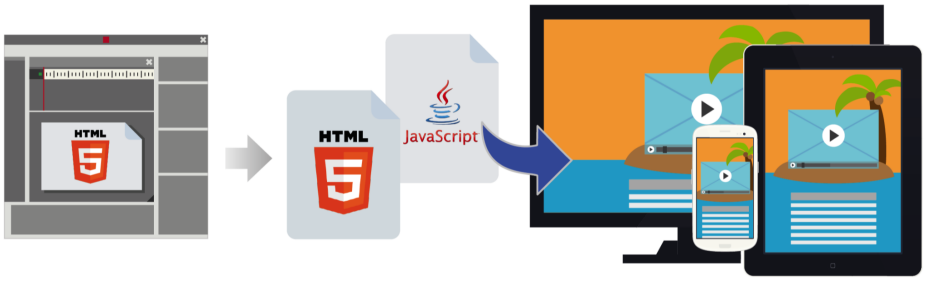

Figure 2.2: Html5 Responsive

for years, its interactive capabilities became a topic of mainstream media around April 2010, after Apple Inc's then-CEO Steve Jobs issued a public letter entitled "Thoughts on Flash" where he concluded that "Flash is no longer necessary to watch video or consume any kind of web content" and that "new open standards created in the mobile era, such as HTML5, will win".This sparked a debate in web development circles where some suggested that while HTML5 provides enhanced functionality, developers must consider the varying browser support of the different parts of the standard as well as other functionality differences between HTML5 and Flash. In early November 2011, Adobe announced that it would discontinue development of Flash for mobile devices and reorient its efforts in developing tools using HTML5.

2.2 Web Components

This section provides an overview of Web Components. Web Components are a set of standards currently being produced by Google engineers

as a W3C specification that allows for the creation of reusable widgets or components in web documents and web applications. The intention behind them is to bring component-based software engineering to the World Wide Web. The components model allows for encapsulation and interoperability of individual HTML elements.

Support for Web Components is present in some WebKit-based browsers like Google Chrome and Opera and is in Mozilla Firefox (requires a manual configuration change). Microsoft's Internet Explorer has not implemented any Web Components specifications yet.[1] Backwards compatibility with older browsers is implemented using JavaScript-based polyfills.[35] Web Compo-

Figure 2.3: Web Components

nents consist of 4 main elements which can be used separately or all together:

- Custom Elements: Custom Elements allow authors to define their own custom HTML elements. Authors associate JavaScript code with custom tag names, and then use those custom tag names as they would any standard tag. Custom elements are still elements. It is possible

to create, use, manipulate, and compose them just as easily as any standard or today.[8]

- Shadow DOM: Shadow DOM addresses the lack of true DOM tree encapsulation when building components. With Shadow DOM, elements can get a new kind of node associated with them. This new kind of node is called a shadow root. An element that has a "shadow root" associated with it is called a "shadow host". The content of a shadow host isn't rendered; the content of the shadow root is rendered instead. Shadow DOM allows a single node to express three subtrees: light DOM, shadow DOM, and composed DOM. Together, the light DOM and shadow DOM are referred to as the logical DOM. This is the DOM that the developer interacts with. The composed DOM is what the browser sees and uses to render the pixels on the screen.[9] Structure of a Shadow DOM An element that has a shadow root associated with it is called shadow host. The shadow root can be treated as an ordinary DOM element, so it is possible to append arbitrary nodes to it. With Shadow DOM, all markup and CSS are scoped to the host element. In other words, CSS styles defined inside a Shadow Root won't affect its parent document, CSS styles defined outside the Shadow Root won't affect the main page.

- HTML Import: This webcomponents.js repository contains a JavaScript polyfill for the HTML Imports specification. HTML Imports are a way to include and reuse HTML documents in other HTML documents. As <script> tags let authors include external JavaScript in their pages, imports let authors load full HTML resources. In particular, imports let authors include Custom Element definitions from external URLs.

- Templates: This specification describes a method for declaring inert DOM subtrees in HTML and manipulating them to instantiate docu-

ment fragments with identical contents.

2.3 Polymer

This section there will be an overview of Polymer. Polymer provides a thin layer of API on top of Web Components and several powerful features, such as custom events and delegation, mixins, accessors and component life-cycle functions, to facilitate the creation of Web Components. Polymer does this by:

- Allowing to create Custom Elements with user-defined naming schemes. These custom elements can then be distributed across the network and used by others with HTML Imports
- Allowing each custom element to have its own template accompanied by styles and behavior required to use that element
- Providing a suite of ready-made UI and non-UI elements to be used and extended in projects
- The elements collection of Polymer is divided into more sections:
- Core Elements — These are a set of visual and non-visual elements designed to work with the layout, user interaction, selection, and scaffolding applications.
- Paper Elements — Implement the material design philosophy launched by Google recently at Google I/O 2014, and these include everything from a simple button to a dialog box with neat visual effects.
- Iron Elements — A set of visual and non-visual utility elements. It includes elements for working with layout, user input, selection, and scaffolding apps.

- Gold Elements — The gold elements are built for e-commerce use-cases like checkout flows.
- Neon Elements — Neon elements implement special effects.
- Platinum Elements — Elements to turn web pages into a true webapp, with push, offline, and more.
- Molecules — Molecules are elements that wrap other javascript libraries.

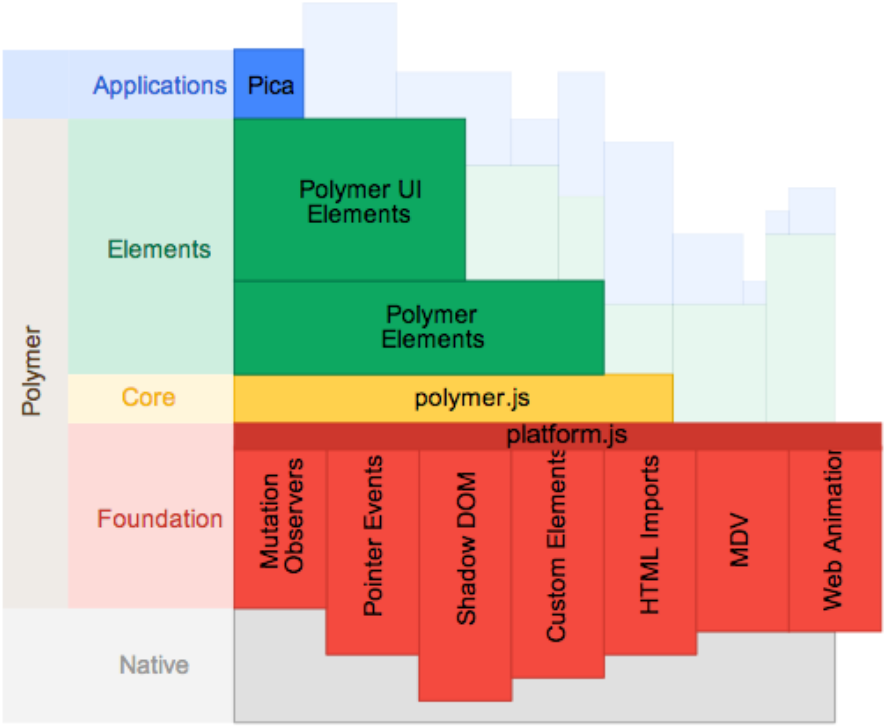

Figure 2.4: Polymer Architecture

Web components standards provide the needed primitives to build new components. It is possible to build custom elements using these primitives, but it can be a lot of work.

The Polymer library provides a declarative syntax that makes it simpler to

define custom elements. Furthermore, it adds features like templating, two-way data binding and property observation to help developers build powerful, reusable elements with less code.
Custom elements. If users don't want to write their own elements, there are a number of elements built with Polymer that it is possible to drop straight into existing pages. These elements depend on the Polymer library, but they can be used without using Polymer directly, as well.[6]
Polymer is one of the first implementations of a user interface library built upon the Web Components standard. Web Components are not fully supported by browsers, but they provide a polyfill library, webcomponents.js, that provides enough functionality to support Web Components and Polymer.
Web Components standard is the result of the evolution of user interface libraries over the past decade, finally reaching the goal of separating HTML, CSS and JavaScript and running HTML through W3C validators. For example, looking at a .css file, it is possbile to easily determine which selectors are actually used in HTML and especially programmatically used in JavaScript. Similarly, it is easy to organize JavaScript code so that everything could be reused efficiently on multiple pages.[23]

2.4 NodeJS

This section provides an overview of NodeJs. Node.js is an open source, cross-platform runtime environment for server- side and networking applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD, NonStop,IBM AIX, IBM System z and IBM i. Its work is hosted and supported by the Node.js Foundation,a Collaborative Project at Linux Foundation.
Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. These tech- nolo-

gies are commonly used for real-time web applications.

Node.js uses the Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a Web server without software such as Apache HTTP Server, Nginx or IIS.

Node.js allows the creation of web servers and networking tools, using JavaScript and a collection of "modules" that handle various core functional- ity. Modules handle file system I/O, networking (HTTP, TCP, UDP, DNS, or TLS/SSL), binary data (buffers), cryptography functions, data streams , and other core functions. Node's modules have a simple and elegant API, reducing the complexity of writing server applications. Frameworks can be used to accelerate the development of applications, and common frameworks are Express.js, Socket.IO and Connect. Node.js applications can run on Microsoft Windows, Unix, NonStop and Mac OS X servers. Node.js applications can alternatively be written with CoffeeScript (an alternative form of JavaScript), Dart or Microsoft TypeScript (strongly typed forms of JavaScript), or any language that can compile to JavaScript. Node.js is primarily used to build network programs such as web servers, making it similar to PHP and Python. The biggest difference between PHP and Node.js is that PHP is a blocking language (commands execute only after the previous command has completed), while Node.js is a non-blocking language (commands execute in parallel, and use callbacks to signal completion).

Node.js brings event-driven programming to web servers, enabling development of fast web servers in JavaScript. Developers can create highly scalable servers without using threading, by using a simplified model of event- driven programming that uses callbacks to signal the completion of a task. Node.js was created because concurrency is difficult in many server-side programming languages, and often leads to poor performance. Node.js connects the ease of a scripting language (JavaScript) with the power of Unix network

programming.

2.5 MongoDB

This section provides an overview of MongoDB.

MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open-source software.

MongoDB was created by Dwight Merriman and Eliot Horowitz, who had encountered development and scalability issues with traditional relational database approaches while building Web applications at DoubleClick, an Internet advertising company that is now owned by Google Inc. According to Merriman, the name of the database was derived from the word humongous to represent the idea of supporting large amounts of data. Merriman and Horowitz helped form 10Gen Inc. in 2007 to commercialize MongoDB and related software. The company was renamed MongoDB Inc. in 2013.

The database was released to open source in 2009 and is available under the terms of the Free Software Foundation's GNU AGPL Version 3.0 commercial license. At the time of this writing, among other users, the insurance company MetLife is using MongoDB for customer service applications, the website Craigslist is using it for archiving data, the CERN physics lab is using it for data aggregation and discovery and the The New York Times newspaper is using MongoDB to support a form-building application for photo submissions.

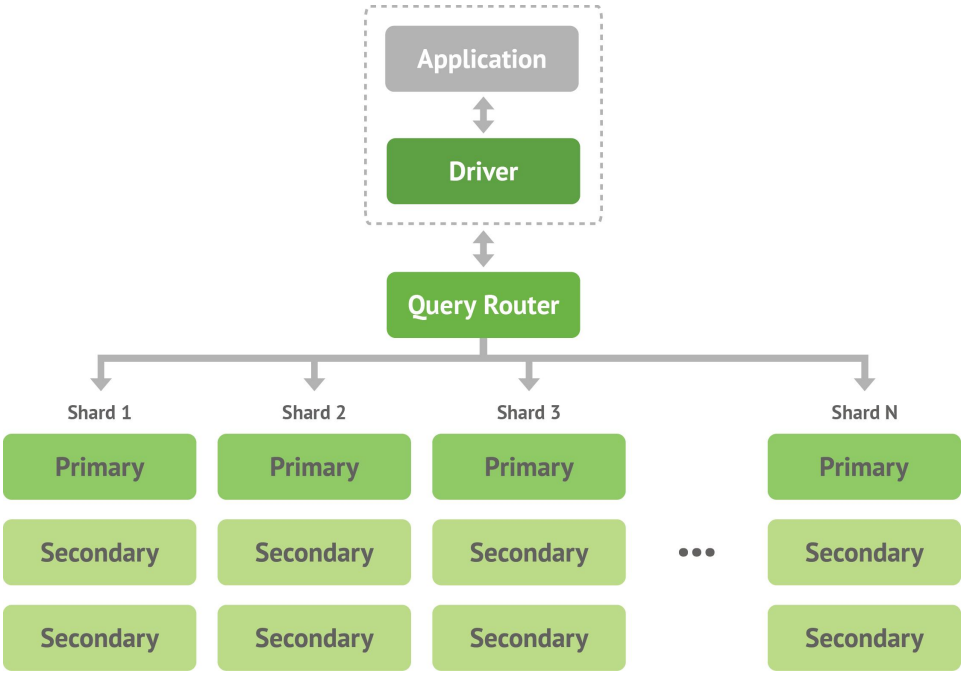

Figure 2.5: MongoDB Architecture

2.6 Strongloop Loopback

This section provides an overview of LoopBack.

Built on top of the open source LoopBack framework, the StrongLoop API Platform is the first end-to-end platform for the full API lifecycle that allows to visually develop REST APIs in Node and get them connected to new and legacy data. In addition, the API Platform features built-in mBaaS features like push and offline sync, plus graphical tools with DevOps features for clustering, profiling and monitoring Node apps.

LoopBack generates model API from the models schemas, to let CRUD operations on models. LoopBack models automatically have a standard set of HTTP endpoints that provide REST APIs for create, read, update, and delete (CRUD) operations on model data:

- **POST /Model** — Create a new instance of the model and persist it into the data source.

- **GET /Model** — Find all instances of the model matched by filter from the data source.
- **PUT /Model** — Update an existing model instance or insert a new one into the data source.
- **PUT /Model/id** — Update attributes for a model instance and persist it into the data source.
- **GET /Model/id** — Find a model instance by id from the data source.
- **DELETE /Model/id** — Delete a model instance by id from the data source.
- **GET /Model/count** — Count instances of the model matched by where from the data source.
- **GET /Model/findOne** — Find first instance of the model matched by filter from the data source.
- **POST /Model/update** — Update instances of the model matched by where from that data source.

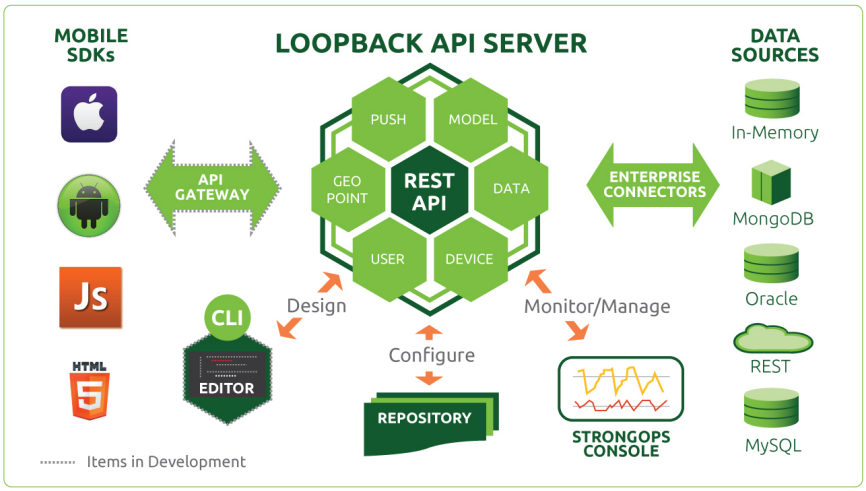

Figure 2.6: Loopback Architecture

A LoopBack model represents data in backend systems such as databases, and by default has both Node and REST APIs. Additionally, developer can add functionality such as validation rules and business logic to models. Every LoopBack application has a set of predefined built-in models such as User, Role, and Application. Developer can extend built-in models to suit application's needs.

The model JSON file defines models, relations between models, and access to models.

```
{
  {
  "name": "ModelName", // See Top-level properties below "description": "A
       Customer model representing our customers.", "base": "User",
  "idInjection": false,
  "strict": true,
  "options": { ... }, // See Options below
  "properties": { ... }, // See Properties below
  "validations": [...], // See Validations below
  "relations": {...}, // See Relations below
  "acls": [...], // See ACLs below
  "scopes": {...}, // See Scopes below
   "http": {"path": "/foo/mypath"}
  }
}
```

Where:

- "name": Name of the model.
- "description": Optional description of the model.
- "base": Name of another model that this model extends. The model will "inherit" properties and methods of the base model.
- "IdInjection": Whether to automatically add an id property to the model:
  - true - id property is added to the model automatically. This is the default.

  - false - id property is not added to the model.

- "strict": Specifies whether the model accepts only predefined properties or not. One of:

  - true - Only properties defined in the model are accepted. Used to ensure that the model accepts only predefined properties.
  - false - The model is an open model and accepts all properties, including ones not predefined in the model. This mode is useful to store free-form JSON data to a schema-less database such as MongoDB.
  - validate - The unknown properties will be reported as validation errors.
  - throw - Throws an exception if properties not defined for the model are used in an operation.
  - undefined - Defaults to false unless the data source is backed by a relational database such as Oracle or MySQL.
  - "options": JSON object that specifies model options.
  - "properties": JSON object that specifies the properties in the model.
  - "relations": Object containing relation names and relation definitions.
  - "acls": Set of ACL specifications that describes access control for the model.

The API can be extended: the developer can add remote functions to models or add hooks to existing API to add custom behavior before and/or after the API handler (to pre-process the re-quest and/or post-process the response). The resulting API is RESTful, cookie free, signed by authentica- tion token. By default, applications have a built-in model that represents a user, with

properties username, email and password and role for authentica- tion and authorization. Loopback also introduces an indirection layer that allows to choose from almost all particular DBMS to be used. In Chapter Two the technologies used for developing this work, have been described. Each technology has been described in realtion to its function and its use in the project.

Chapter 3

Enabling services

This chapter describes enabling services.

The first two sections describes the payment services focus Braintree and Stripe.

A payment service provider (PSP) offers shops online services for accepting electronic payments by a variety of payment methods including credit card, bank-based payments such as direct debit, bank transfer, and real-time bank transfer based on online banking. Typically, they use a software as a service model and form a single payment gateway for their clients (merchants) to multiple payment methods.

These services help the developer to integrate into their application payment systems easily. In particular, these services provide the libraries that are to be imported into your application to help manage the payment form. Each service provides a default form ready to be integrated in the application with a minimal interface(see 3.1 3.2).

The third section will discuss the issue of taxes that is very common problem for systems of e-commerce. And at last section we describes the shipping systems.

3.1 Braintree

Braintree is a full-stack payments platform that makes it easy to accept payments in your app or website. Our service replaces the traditional model of sourcing a payment gateway and merchant account from different providers. From one touch payments to mobile SDKs and foreign currency acceptance, we provide everything you need to start accepting payments today.
All kinds of organizations use Braintree to accept payments in mobile apps and websites. From startups in garages, to not-for-profits, to some of the largest online retailers, we have more experience working with new business models than any other payments provider. However, due to legal and regulatory compliance reasons, Braintree isn't able to work with some business types. How to use Braintree as a payment system? Braintree's consists of complementary client and server SDKs:

- The client SDK enables you to collect payment method (e.g. credit card, PayPal) details;
- The server SDKs manage all requests to the Braintree gateway

Before we get started, there are two key concepts to introduce - the client token and the payment method nonce.
SDK Braintree offer various options to the programmer to integrate the payment service.

3.1.1 Client Token

A client token is a signed data blob that includes configuration and authorization information required by the Braintree Client SDK. These should not be reused; a new client token should be generated for each customer request that's sent to Braintree. For security, Braintree server revoke client tokens if they are reused excessively within a short time period. The server

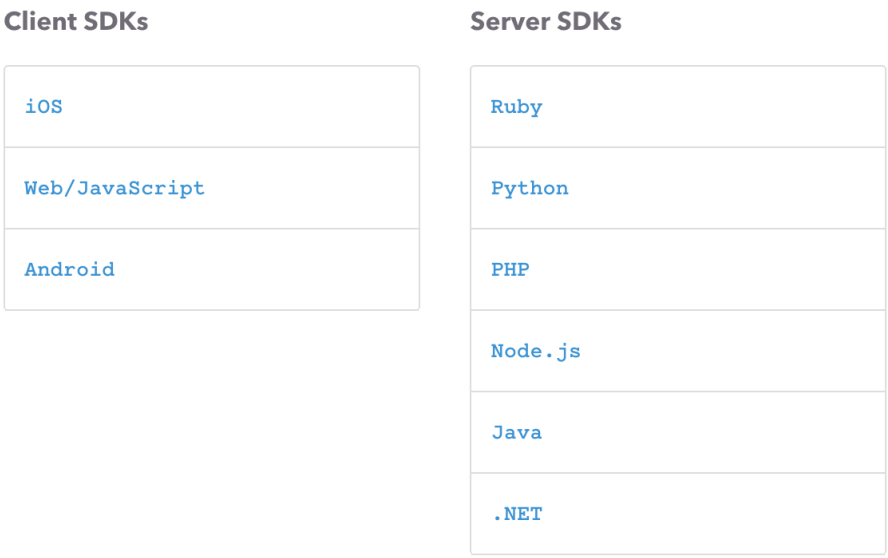

Figure 3.1: Braintree SDK

is responsible for generating the client token, which contains all of the necessary configuration information to set up the client SDKs. When your server provides a client token to your client, it authenticates the application to communicate directly to Braintree. The client is responsible for obtaining the client token and initializing the client SDK. If this succeeds, the client will generate a *payment_ method_ nonce*.

3.1.2 Payment Method Nonce

The payment method nonce is a string returned by the client SDK to represent a payment method. This string is a reference to the customer payment method details that were provided in your payment form and should be sent to your server where it can be used with the server SDKs to create a new transaction request. Payment method nonces expire after 24 hours. The server integration doesn't need to know the payment method type (e.g.

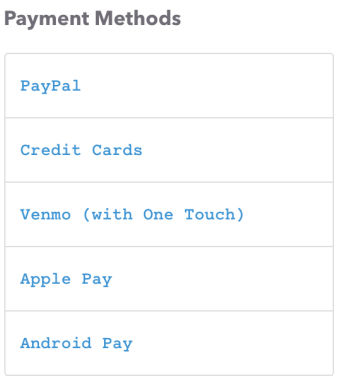

Figure 3.2: Braintree payment method

credit card, PayPal account, Bitcoin) that is represented in the nonce. This means that your first v.zero integration should continue to work with few or no code changes when new payment method types are introduced.

3.1.3 How it work

1. App or web front-end requests a client token from your server in order to initialize the client SDK;

2. Server generates and sends a client token back to your client with the server SDK;

3. Once the client SDK is initialized and the customer has submitted payment information, the SDK communicates that information to Braintree, which returns a payment method nonce;

4. Then send the payment nonce to your server;

5. Server code receives the payment method nonce from your client and then uses the server SDK to create a transaction or perform other Braintree functions.

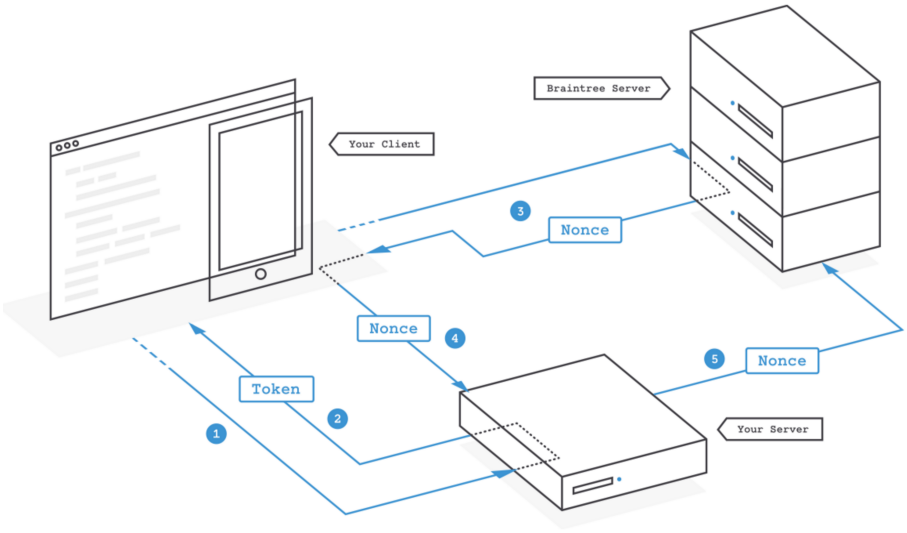

Figure 3.3: Comunication between client and server braintree step by step

The easiest way to use braintree is through Drop-in UI. This type of configuration is certainly the simplest but allows the programmer to customize the input form for entering your payment information. The interface Drop-in looks like this as follows: To create and maintain this form to the client-side

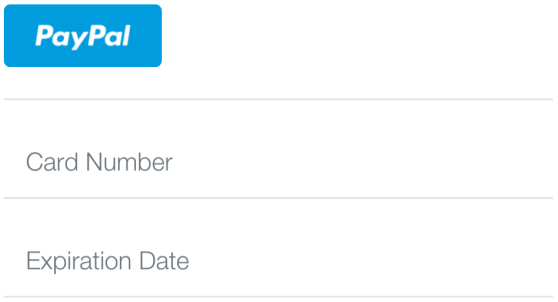

Figure 3.4: Drop-in UI

you must import the scrip *brantree.js* and organize form so as follows:

```
<form id="checkout" method="post" action="/checkout">
  <div id="payment-form"></div>
```

```
  <input type="submit" value="Pay $10">
</form>

<script src="https://js.braintreegateway.com/v2/braintree.js"></script>
<script>
// We generated a client token for you so you can test out this code
// immediately. In a production-ready integration, you will need to
// generate a client token on your server.
var clientToken = client_token;

braintree.setup(clientToken, "dropin", {
  container: "payment-form"
});
</script>
```

To start up, braintree.js needs a client-token generated by your Braintree server SDK. The client-token is unique.

There are a number of ways to get your client token into JavaScript so you can set up Braintree. Many people choose to interpolate the client token into the HTML/JavaScript itself; alternatively, you could load the client token from an AJAX call to your exposed client token URL on your server. Once you've finished this setup. Now we are ready to make payments.

A Braintree client-side integration sends payment information – like a credit card or a PayPal authorization – to Braintree in exchange for a payment method nonce, a one time use value that represents that payment method. On your server, use a payment method nonce with a Braintree server SDK to charge a card or update a customers' payment methods.

By default, *braintree.js* will add a hidden input named *payment_ method_ nonce* to your form. When your user submits the form, if you have not subscribed to the *onPaymentMethodReceived* callback, your form will be submitted with this value.

Braintree provides a sandbox for developers account, credit cards test for testing your application. That said, we would like to use a form that is customizable to the way we would like to stylize the payment form without

using the default. To do this simply specify additional parameters to the client side. How does this see in Chapter 5.

3.2 Stripe

Stripe is the best way to accept payments online. Stripe aims to expand internet commerce by making it easy to process transactions and manage an online business.
Stripe is 365 people and headquartered in an old trunk factory in the Mission district of San Francisco. The company has received around $300 million in funding to date; investors include Sequoia Capital, Visa, American Express, Peter Thiel, and Elon Musk. Stripe enables you to accept payments in minutes. Collect your customers' payment information easily and securely on web or mobile, and create charges server-side. Stripe supports 100+ currencies out of the box. In addition to credit and debit cards, Apple Pay, Android Pay, you can also easily support Bitcoin, Alipay, or Amex Express Checkout.

3.2.1 How it work

Even Stripe, as Braintree, provides a default widget that you can use to integrate payments. To get this widget, just enter the following code in its page:

```
<form action="" method="POST">
  <script
    src="https://checkout.stripe.com/checkout.js" class="stripe-button"
    data-key="pk_test_6pRNASCoBOKtIshFeQd4XMUh"
    data-amount="2000"
    data-name="Demo Site"
    data-description="2 widgets ($20.00)"
    data-image="/128x128.png"
    data-locale="auto">
  </script>
</form>
```

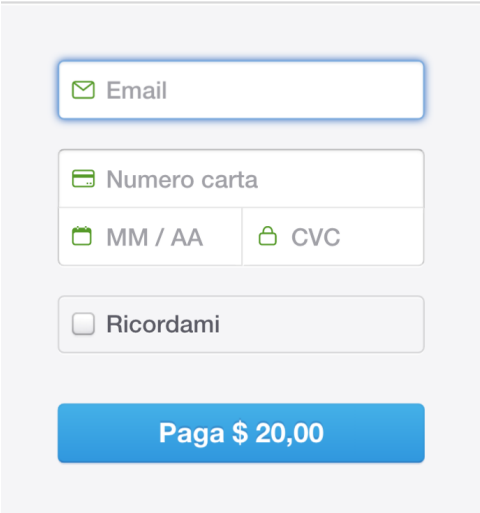

Figure 3.5: Default stripe payment widget

The most important thing to notice is the data-key attribute added to the script tag. This key identifies your account when communicating with Stripe. Stripe also offers the ability to customize the payment form. This and other details can be discussed in Chapter 5.

3.3 Taxation of Electronic Commerce: A Developing Problem

The rapid growth of e-commerce, especially the sale of goods and services over the internet, has fuelled a debate about the taxation regimes to be used. The shift from a physically oriented commercial environment to a knowledge-based electronic environment poses serious and substantial issues in relation to taxation and taxation regimes. Tax administrations throughout the world face the formidable task of protecting their revenue base without hindering either the development of new technologies or the involvement of the business community in the evolving and growing e-market place. These problems will be greater for developing countries.

It 'clear that the solutions to this problem are subject to continuation variations caused by changes of rules in force in each state.
Since it is a very very common problem in systems e-commerce, today, fortunately, there are several services that offer ready-made solutions.
These services (TaxJar, Avalara, CloudTax, etc), give a big hand to the developer to manage the prices of goods or services.

3.3.1 Taxjar

For get precise sales tax rates and calculations at the state, county, city and special taxing district level so you charge the customer the right amount of sales tax, every time, must be updated continuously on the change in the rules in force.
No more building tax tables and dealing with ever-changing sales tax rates in thousands of sales tax districts. So, sales tax is so complicated because every state is different. Some states require that merchants charge sales tax on shipping charges, others don't.
Some states have "origin-based" sales tax sourcing. Others are "destination-based." TaxJar's SmartCalcs API comes handles complicated sales tax sourcing rules so you never have to worry about it. Certain products types, like food or clothing are taxed at a lower rate or even tax exempt in some states. TaxJar SmartCalcs Sales Tax API handles complicated product-level taxability so you never have to worry about customizing sales tax rates.
Taxjar SmartCalcs solves solves all these problems and it provides an easy interface to very complicated problems, RESTful APIs.
Let's see what parameters must be specified for calculating the fees in two locations:

- *from_ country*: String(optional) - ISO two country code of the country where the order shipped from;
- *from_ zip*: String(optional) - Postal code where the order shipped from

(5-Digit ZIP or ZIP+4);

- *from_state*: String(optional) - State where the order shipped from.;
- *from_city*: String(optional) - City where the order shipped from;
- *from_street*: String(optional) - Street address where the order shipped from.;
- *to_country*: String(required) - ISO two country code of the country where the order shipped to;
- *to_zip*: String(conditional) - Postal code where the order shipped to (5-Digit ZIP or ZIP+4);
- *to_state*: String(conditional) - State where the order shipped to;
- *to_city*: String(optional) - City where the order shipped to;
- *to_street*: String(optional) - Street address where the order shipped to;
- *amount*: Long(optional) - Total amount of the order, excluding shipping;
- *shipping*: Long(required) - Total amount of shipping for the order;

Following is an example in code:

```
var taxjar = require("taxjar")(<api-key>);
taxjar.taxForOrder({
  'from_country': 'US',
  'from_zip': '07001',
  'from_state': 'NJ',
  'to_country': 'US',
  'to_zip': '07446',
  'to_state': 'NJ',
  'amount': 16.50,
  'shipping': 1.5
}).then(function(res) {
```

```
  res.tax; // Tax object
  res.tax.amount_to_collect; // Amount to collect
});
</script>
```

Taxjar, also allows you to specify many other parameters such as the order lines, to be precise in the calculation of rates. For more details see documentation.

3.4 Easypost

Another key point in the systems of e-commerce is the delivery and tracking of parcels. From the logistics point of view this is a very complicated problem to which we must pay close attention today because customer satisfaction is strongly dependent on the traceability of its services well. Through this feature the customer is aware of when you receive the order. Easypost

Figure 3.6: shipping and tracking

is a service that solves problems related notes to shipments and tracking of packages offering a solution well established. Following is an example in code to send a parcel:

```
var easypost = require('node-easypost')(apiKey);
```

```
// set addresses
var toAddress = {
    name: "Dr. Steve Brule",
    street1: "179 N Harbor Dr",
    city: "Redondo Beach",
    state: "CA",
    zip: "90277",
    country: "US",
    phone: "310-808-5243"
};
var fromAddress = {
    name: "EasyPost",
    street1: "118 2nd Street",
    street2: "4th Floor",
    city: "San Francisco",
    state: "CA",
    zip: "94105",
    phone: "415-123-4567"
};

// verify address
easypost.Address.create(fromAddress, function(err, fromAddress) {
    fromAddress.verify(function(err, response) {
        if (err) {
            console.log('Address is invalid.');
        } else if (response.message !== undefined && response.message !==
            null) {
            console.log('Address is valid but has an issue: ', response.
                message);
            var verifiedAddress = response.address;
        } else {
            var verifiedAddress = response;
        }
    });
});

// set parcel
easypost.Parcel.create({
    predefined_package: "InvalidPackageName",
    weight: 21.2
}, function(err, response) {
    console.log(err);
```

```
});

var parcel = {
    length: 10.2,
    width: 7.8,
    height: 4.3,
    weight: 21.2
};

// create customs_info form for intl shipping
var customsItem = {
    description: "EasyPost t-shirts",
    hs_tariff_number: 123456,
    origin_country: "US",
    quantity: 2,
    value: 96.27,
    weight: 21.1
};

var customsInfo = {
    customs_certify: 1,
    customs_signer: "Hector Hammerfall",
    contents_type: "gift",
    contents_explanation: "",
    eel_pfc: "NOEEI 30.37(a)",
    non_delivery_option: "return",
    restriction_type: "none",
    restriction_comments: "",
    customs_items: [customsItem]
};
// create shipment
easypost.Shipment.create({
    to_address: toAddress,
    from_address: fromAddress,
    parcel: parcel,
    customs_info: customsInfo
}, function(err, shipment) {
    // buy postage label with one of the rate objects
    shipment.buy({rate: shipment.lowestRate(['USPS', 'ups']), insurance:
        100.00}, function(err, shipment) {
        console.log(shipment.tracking_code);
        console.log(shipment.postage_label.label_url);
```

```
    });
});
```

Part II

Part 2

Chapter 4

x-commerce's design

This chapter presents the core of the thesis project: X-commerce. The first section provides a project's overview, giving reasons of development, listing benefits and functions. The second section describe Single Page Application development pattern. The third section shows the x-commerce architectural stack and the reasons why these technologies have been chosen. The fourth section presents the development methodology that has been thought for the project. In the fifth paragraph shows the models and their APIs basic x-commerce.

4.1 x-commerce overview

X-commerce is a web platform for building e-commerce systems. This platform build full-stack Javascript NodeJS API-centric HTML5 based Single Page Application with Web Components via Polymer-Project. In particular, the core of x-commerce follows the philosophy of Polymer project or all the complex parts of the platform are self-contained and isolated so that responsibilities are well localized. In other words, x-commerce has been developed to facilitate the modifiability of the code, integrity of new services and reusability. This is facilitated thanks to Polymer for which: "Everything

is an element, even a service". So, a Web plathform is essentially built by composing elements together.

For example a page of a product has been divided into various element and each element contributes to realize the same page. In this way any changes related to the code are very well located. In following it is a more abstract level is designed as a page in general. So e-commerce is a platform developed

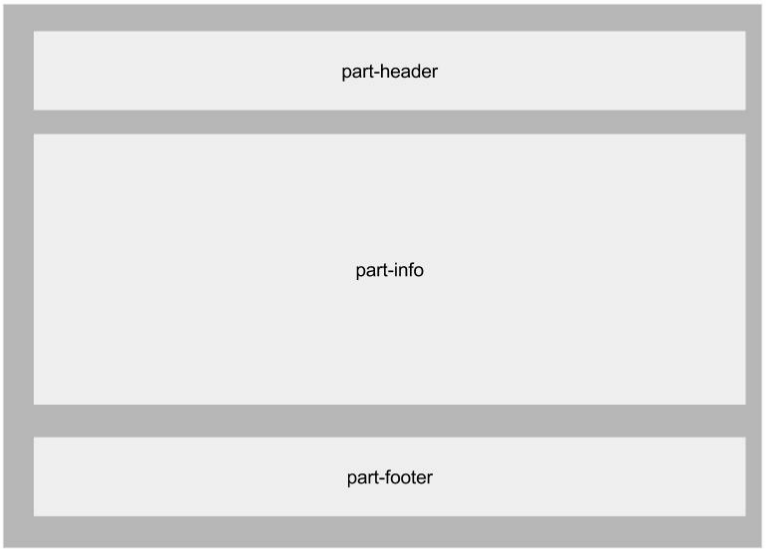

Figure 4.1: Example of how the pages are designed

for compositions of elements. This philosophy also helps code reusability. In fact, in the navigation of web pages, between the previous and the next, there is a lot in common such as header and footer. With web components, so, they also create elements that are often used by most parties. Moreover, thanks to Web Components philosophy, it's easy to gain a clear separation between structure, content, behavior and presentation of elements. It's possible to create components that concern the only presentation part of an

element, such as mixins in which developers can express groups of CSS rules to be applied to different elements.
On a small project, the potential of Polymer may not be very obvious, but when the project is large the philosophy that "everything is an element" helps a lot in design and construction.

4.2 Single Page application

Single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience akin to a desktop application. In a SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[31] or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although modern web technologies (such as those included in the HTML5 pushState() API) can provide the perception and navigability of separate logical pages in the application. Interaction with the single page application often involves dynamic communication with the web server behind the scenes.
SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work hap- pens on the client side, in JavaScript. For the traditional ASP.NET developer, it can be difficult to make the leap. Luckily, there are many open source JavaScript frameworks that make it easier to create SPAs. In a traditional Web app, every time the app calls the server, the server renders a new HTML page. This triggers a page refresh in the browser. In an SPA, after the first page loads, all interaction with the server happens through AJAX calls. These AJAX calls return data - not markup - usually in JSON format. The app uses the JSON data to update the page dynamically, without reloading the page. One benefit of SPAs is obvious: Applications are more fluid and

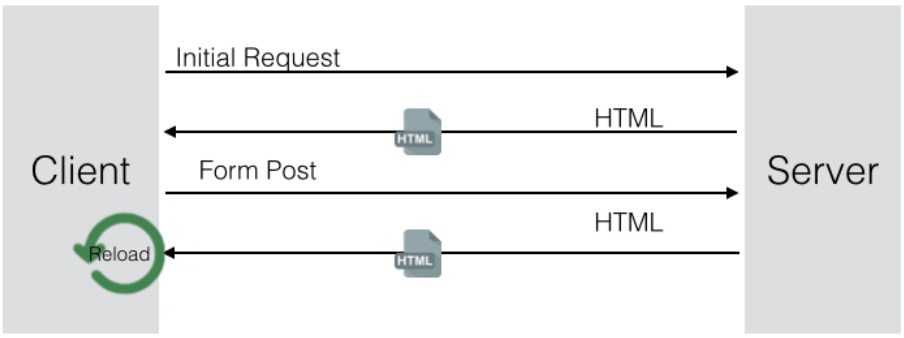

Figure 4.2: The Traditional Page Lifecycle

responsive, without the jarring effect of reloading and re-rendering the page.

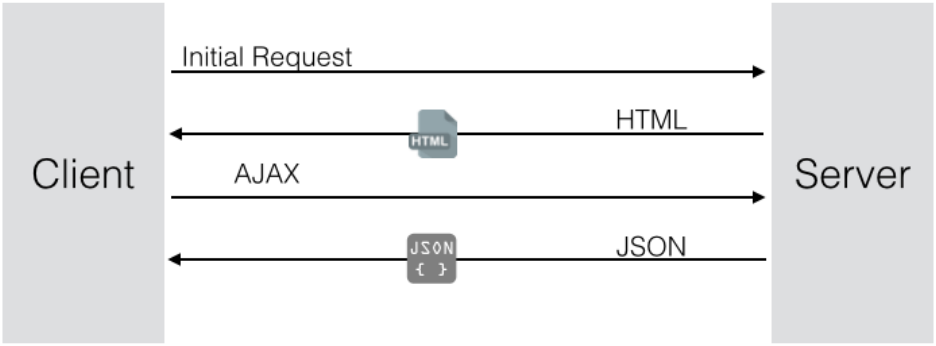

Figure 4.3: The SPA Lifecycle

4.3 x-commerce architecture

A Web application developed by using the x-commerce toolkit, is a full stack JavaScript Single Page Application.

4.3.1 server side

On the server side, an x-commerce app is based on NodeJS (see 2.4) used to create the server environment, MongoDB (see 2.5) used to storage data,

and the Web framework Loopback by Strongloop (see 2.6).
Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world. NodeJS lets to create a ver- tical full-stack application in Javascript. The NodeJS asynchronous development scheme increases performances of web applications, by using downtime caused by HTTP requests.
LoopBack generates model API from the models schemas, to let CRUD operations on models. Loopback is the core of the X-Project server-side. Document oriented API definition guarantees easiness and speed in API creation. Moreover, Loopback, is fully compatible with several DBMS thanks to connectors.
MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas. The document-oriented being of MongoDB allows to horizontal scale in really easy way. Moreover, document models, that replace relational database's rows, are schemaless: this feature gives to MongoDB project high levels of felxibility and manageability.

4.3.2 client side

On the client side, an x-commerce app is based on HTML5 Web Components via Polymer-Project by Google. It is a set of Polymer elements for local routing, API request, forms, lists, style and admin panels.
Polymer-Project is one of the most important emerging realities of the moment. Google team has heavily focused his forces on code reusability and on separation between behavior, presentation and content. Code reusability is a direct effect of Web Components structure: the creation of widget that

can be completely indipendent facilitates code reuse.
Finally, the union between Polymer and Strongloop has been topped by the creation of the document oriented development process described below 4.5.

4.4 x-commerce organization

Nella fase di progettazione, sono state individuate due soluzioni relativi alla gestione delle risorse utilizzate dal client e dall'admin di x-commerce. In particolare i client di x-commerce hanno necessità diverse nell'uso della piattaforma rispetto all'admin. Mentre il primo è più orientato alla fruizione delle informazioni relative ai prodotti ed eventalmente alle ordinazioni, l'admin è più interessato all'inserimetno di nuovi prodotti e altre informazioni al fine di organizzare al meglio le risorse e l'interfaccia offerta al client. Considerando che i due attori principali coinvolti in queste interrazioni richiedono livelli di sicurezza diversi cioè si vuole che soltanto l'admin possa apportare modifiche alle informazioni mentre tutti gli altri possono fruire dei dati.
Per separare questa gestione dei client e admin di x-commerce, si è deciso di separare le pagine di client e dell'admin in due directory diverse. In questo modo si può ospitare/gestire l'admin su un server diverso rispetto al server su cui accedono i client di x-commerce. Cio non esclude la possibilità di offrire i due tipi di servizi da un unico server.
Questa organizzazione delle pagine per il client e l'admin permette infine di assegnare lo stesso nome alle pagine e alle sue parti.

4.5 Document-Driven web Development Process

The process to build a web application based on x-commerce toolkit consists of the following four steps: models schemas definition, HTTP RESTful API definition, UI components definition and UI components assembly.

4.5.1 1st step - Model schemas definition

A description of entities, properties, relations and data access policies are defined as JSON documents.
The models of e-commerce are many and are expected to grow further with the integration and implementation of new services. For the moment there are 30 models. Loopback in the definition of each model is done by filling in a JSON file where declare properties that interest us. In the following are the main models of x-commerce:

- *product*: the model has the properties we are specifying the property of a product which interest, for which:
  - a product has a title string (we are also saying that this field is required);
  - a product has a description string;
  - a product has a price of type Number, etc..

  The model has also produced the "relations". This field identifies relatiosn current model ("product") with other models such as:
  - a product has a relationship with the model "Image" type "many". This report was modeled because the product has a lot of pictures;
  - a product has the relationship with the model "Comments" type "many" because a product has many comments or reviews, etc..
  - it is possible also specify "acls (access control lists)" to control access to the system. Specifying who can do what with a JSON file, you can have in a transparent way all security mechanisms.

```
{
  "name": "Product",
  "properties": {
    "title": {
      "type": "String",
```

```
    "required": true
  },
  "description": {
    "type": "String"
  },
  "price":  {
    "type": "Number"
  },
  ..........
},
"relations": {
  "images": {
        "type": "hasMany",
        "model": "Image"
      },
  "comments": {
    "type": "hasMany",
    "model": "Comment"
  },
  "collections": {
    "type": "hasAndBelongsToMany",
    "model": "Collection",
    "through": "CollectionProduct"
  },
  "options": {
    "type": "hasMany",
    "model": "ProductOption"
  },
  "product_type": {
    "type": "belongsTo",
    "model": "ProductType"
  },
  "variants": {
    "type": "hasMany",
    "model": "ProductVariant"
  },
  "vendor": {
    "type": "belongsTo",
    "model": "Vendor",
  },
  ..........
}
```

```
}
```

- *customers*: this time there are two new things:
  - the presence of *acls*: in this case acls block all calls except "find" and count" that can call all;
  - note the presence of the *"base": "Users"*: "Users" is a model of loopback that is offered for free. This model brings back all funtionality related to login, sign up, forgot etc.. "Customer" model extends the "Users" loopback - and in this way the "Customer" legacy capabilities and therefore all methods APIs and "Users" including methods for login and logout.

```
{
  "name": "Customer",
  "base": "User",
  "properties": {
    "first_name": {
      "type": "String",
      "required": true
    },
    "last_name": {
      "type": "String",
      "required": true
    },
    "date_of_birth" : {
      "type": "Date"
    },
    "gender": {
      "type": "String",
      "enum": ["M", "F"]
    },
    "email": {
      "type": "String",
      "required": true
    },
    ...........
  },
  "acls": [
```

```
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "find"
    },
    {
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "ALLOW",
      "property": "count"
    },
    ...........
  ]
}
```

Each file JSON is also accompanied by a js file. This file is used to define the so-called "hooks" or the methods to define new APIs customized. These new APIs are added to the API generated by the JSON file.

```
module.exports = function (Product) {
};
```

Inside the function offered to extend the API, it is possbile to define new APIs. The name of the API is declared costs as follows:

```
module.exports = function (Product) {
  Product.remoteMethod('generate_variants', {
    accepts: { arg: 'product_id', type: 'string', required: true },
    returns: { arg: 'variants', type: 'Array' },
    http: { verb: 'get', path:'/generate' }
  });
};
```

The remote method remote method takes two parameters:

- name of the method to execute the call of the API in question - In this way, the method name is "generate_variants". This method will be executed when it is called api: "/api/Products/generate";

- as the second parameter, the remote method accepts a JSON object consisting of key-value pairs:
  - "accepts": indicates the input parameters ie parameters selling sent the body of the request. In this case the input parameter is the only one and is of type String and is required.
  - "returns": It indicates what type is the response;
  - "http": This is a parameter that defines url API. Therefore, url for call the generate API is: "/api/Products/generate". When you call this API, it is invoked and executed the remote method declared as the first parameter. The remote method must be defined freely by the programmer.

Therefore, the complete example to define new APIs via remote methods, which are not generated by default from JSON file, is the following:

```
module.exports = function (Product) {
  Product.generate_variants = function (product_id, callback) {
    // to do anythings. Genreate a result
    callback(null, result);
  };

  Product.remoteMethod('generate_variants', {
    accepts: { arg: 'product_id', type: 'string', required: true },
    returns: { arg: 'variants', type: 'Array' },
    http: { verb: 'get', path:'/generate' }
  });
};
```

4.5.2 2nd step - HTTP RESTful API definition

CRUD operations on models are automatically generated by the web framework (on the basis of input JSON documents) and further custom actions can be defined. Following models of e-commerce are:

- *products*: as already mentioned, it is the main model of the project. The properties of this model are easy to imagine because it's property that we all seek when we buy a product or a fine. So a product has:
  - *title*: type string;
  - *description*: type string;
  - *price*: type Number;
  - *compare_at_price*: type Number - This data is related to the "free" product and is used to show a reduced price to the client-side;
  - *is_charge_taxes*: type Boolean - This data is used to load the order of taxes or not. Some customers may be absent from paying taxes;
  - *sku*: type String - stock keeping unit or SKU - is a number or string of alpha and numeric characters that uniquely identify a product;
  - *barcode*: type String - is the small image of lines (bars) and spaces that is affixed to retail store items, identification cards, and postal mail to identify a particular product number, person, or location;
  - *track_quantity*: type Boolean - used to keep track of the amount of products.
  - *quantity*: type Number;
  - *sell_after_purchase*: type Boolean;
  - *unit_measure_weight*: type String;
  - *weight*: type Number;
  - *is_published*: type Boolean;
  - *published_at*: type Date;
  - *tags*: type Array of String

- *article*:

```
{
  "name": "Article",
  "properties": {
    "title": { "type": "string", "required": true },
    "subtitle": { "type": "string" },
    "summary": { "type": "string" },
    "content": { "type": "string" },
    "created_at": { "type": "date" },
    "updated_at": { "type": "date" },
    "published_at": { "type": "date" },
    "tags": { "type": ["string"] }
},
"relations": {
"author": { "type": "belongsTo", "model": "Manager" },
"category": { "type": "belongsTo", "model": "Category" },
    "images": {
      "type": "hasMany",
      "model": "Image",
      "foreignKey": "article_id"

    }
  }
}
```

- *collection_filters*:

```
{
  "name": "CollectionFilter",
  "properties": {
    "name": {
      "type": "String"
    },
    "type": {
      "type": "String"
    },
    "relation": {
      "type": "String"
    },
    "value": {
      "type": "String"
    }
```

```
  }
}
```

- *collections*:

```
{
  "name": "Collection",
  "properties": {
    "title": {
      "type": "String",
      "required": true
    },
    "description": {
      "type": "String"
    },
    "is_published": {
      "type": "Boolean"
    },
    "published_at": {
      "type": "Date"
    }
  },
  "validations": [],
  "relations": {
    "images": {
      "type": "hasMany",
      "model": "Image",
    },
    "products": {
      "type": "hasAndBelongsToMany",
      "model": "Product",
      "through": "CollectionProduct"
    }
  }
}
```

- *comment_ replies*:

```
{
  "name": "CommentReply",
  "properties": {
    "text": {
```

```
      "type": "string"
    },
    "created_at": {
      "type": "date"
    }
  },
  "validations": [],
  "relations": {
    "author": {
      "type": "belongsTo",
      "model": "Customer",
      "foreignKey": "author_id"
    }
  }
}
```

- *comments*:

```
{
  "name": "Comment",
  "properties": {
    "title": {
      "type": "String"
    },
    "text": {
      "type": "String"
    },
    "created_at": {
      "type": "Date"
    }
  },
  "relations": {
    "author": {
      "type": "belongsTo",
      "model": "Customer"
    },
    "replies": {
      "type": "hasMany",
      "model": "CommentReply"
    }
  }
}
```

- *coupons*:

```
{
  "name": "Coupon",
  "properties": {
    "name": {
      "type": "String",
      "required": true
    },
    "description": {
      "type": "String"
    },
    "discount": {
      "type": "Number",
      "required": true
    },
    "code": {
      "type": "String",
      "required": true
    },
    "date_from": {
      "type": "Date"
    },
    "date_to": {
      "type": "Date"
    }
  },
  "relations": {
    "order": {
      "type": "belongsTo",
      "model": "Order",
    }
  }
}
```

- *customers*:

```
{
  "name": "Customer",
  "base": "User",
  "properties": {
    "first_name": {
      "type": "String",
```

```
      "required": true
    },
    "last_name": {
      "type": "String",
      "required": true
    },
    "date_of_birth" : {
      "type": "Date"
    },
    "gender": {
      "type": "String",
      "enum": ["M", "F"]
    },
    "email": {
      "type": "String",
      "required": true
    },
    ........
  },
  "relations": {
    "shipping_addresses": {
      "type": "hasMany",
      "model": "Address"
    },
    "wishlists": {
      "type": "hasMany",
      "model": "Wishlist"
    }
  }
}
```

- *images*:

```
{
  "name": "Image",
  "strict": false,
  "idInjection": false,
  "options": {
    "validateUpsert": true
  },
  "properties": {
    "thumbs": {
```

```
      "type": "array"
    },
    "description": {
      "type": "string"
    },
    "filename": {
      "type": "string"
    },
    ........
  }
  "relations": {}
}
```

- *invoices*:

```
{
  "name": "Invoice",
  "properties": {
    "type": {
      "type": "String",
      "required": true
    },
    "number": {
      "type": "String",
      "required": true
    },
    "date": {
      "type": "Date"
    },
    "taxable": {
      "type": "Number"
    },
    "tot_taxable": {
      "type": "Number"
    },
    "tax_percentual": {
      "type": "Number"
    },
    ........
  },
  "relations": {
    "invoice": {
```

```
      "type": "belongsTo",
      "model": "Order"
    }
  }
}
```

- *managers*:

```
{
  "name": "Manager",
  "base": "InvitableUser",
  "properties": {}
}
```

- *invitable_ users*:

```
{
  "name": "InvitableUser",
  "base": "User",
  "mongodb": {
    "collection": "invitable_users"
  },
  "properties": {
    "role": {
      "type": "string",
      "enum": [
        "admin",
        "editor",
        "author"
      ]
    },
    "fullname": {
      "type": "string"
    },
    "location": {
      "type": "string"
    },
    "is_main_admin": {
      "type": "Boolean",
      "required": false
    }
  },
```

```
  "relations": {
    "articles": {
      "type": "hasMany",
      "model": "Article"
    }
  },
  "mixins": {
    "TimeStamp" : true
  }
}
```

- *orders*:

```
{
  "name": "Order",
  "properties": {
    "status": {
      "type": "String",
      "enum": ["open", "pending", "paid", "closed"],
      "default": "open",
      "required": true
    },
    "discount": {
      "type": "Number"
    },
    "shipping_cost": {
      "type": "Number"
    },
    "taxes": {
      "type": "Number"
    },
    "total": {
      "type": "Number"
    },
    .......
  },
  "validations": [],
  "relations": {
    "customer": {
      "type": "belongsTo",
      "model": "Customer",
      "foreignKey": "customer_id"
```

```
    },
    "order_items": {
      "type": "hasMany",
      "model": "OrderItem",
      "foreignKey": "order_id"
    },
    "payments": {
      "type": "hasMany",
      "model": "Payment",
      "foreignKey": "order_id"
    },
    "taxes": {
      "type": "hasMany",
      "model": "Tax",
    }
  }
}
```

- *order_items*:

```
{
  "name": "OrderItem",
  "properties": {
    "quantity": {
      "type": "Number"
    }
  },
  "relations": {
    "product": {
      "type": "belongsTo",
      "model": "Product"
    },
    "product_variant": {
      "type": "belongsTo",
      "model": "ProductVariant"
    }
  }
}
```

- *payments*:

```
{
```

```
  "name": "Payment",
  "properties": {
    "payment_method": {
      "type": "String"
    },
    "payment": {
      "type": "Object"
    }
  },
  "relations": {}
}
```

- *product_ options*:

```
{
  "name": "ProductOption",
  "properties": {
    "name": {
      "type": "String"
    },
    "values": {
      "type": ["String"]
    },
    "type": {
      "type": "String"
    }
  },
  "relations": {}
}
```

- *product_ types*:

```
{
  "name": "ProductType",
  "properties": {
    "name": {
      "type": "String",
      "required": true
    },
    "description":{
      "type": "String"
    }
```

```
  },
  "relations": {}
}
```

- *product_ variants*:

```
{
  "name": "ProductVariant",
  "properties": {
    "name": {
      "type": "String",
      "required": true
    },
    "combo": {
      "type": ["String"],
      "required": true
    },
    "price": {
      "type": "Number"
    },
    "sku": {
      "type": "String"
    },
    "barcode": {
      "type": "String"
    },
    ......
  },
  "relations": {
    "product": {
      "type": "belongsTo",
      "model": "Product"
    }
  }
}
```

- *reviews*:

```
{
  "name": "Review",
  "properties": {
   "title": {
```

```
      "type": "String"
    },
    "text": {
      "type": "String"
    },
    "rating": {
      "type": "Number"
    },
    "closed": {
      "type": "Boolean",
      "default": false
    }
  },
  "relations": {
    "product": {
      "type": "belongsTo",
      "model": "Product"
    },
    "customer": {
      "type": "belongsTo",
      "model": "Customer"
    },
    "replies": {
      "type": "hasMany",
      "model": "ReviewReply"
    }
  }
}
```

- *reviews_replies*:

```
{
  "name": "ReviewReply",
  "properties": {
    "text": {
      "type": "string"
    },
    "created_at": {
      "type": "date"
    }
  },
  "relations": {
```

```
    "author": {
      "type": "belongsTo",
      "model": "Customer"
    }
  }
}
```

- *services*:

```
{
  "name": "Service",
  "properties": {
    "name": {
      "type": "String"
    },
    "public_key": {
      "type": "String"
    },
    "private_key": {
      "type": "String"
    },
    "params": {
      "type": "Object"
    }
  },
  "relations": {}
}
```

- *stores*:

```
{
  "name": "Store",
  "properties": {
    "name": {
      "type": "String",
      "default": "x-commerce",
      "required": true
    },
    "description": {
      "type": "String",
      "required": true
    },
```

```
    "mobile_phone": {
      "type": "String"
    },
    "office_phone": {
      "type": "String"
    },
    "email": {
      "type": "String",
      "length": 64,
      "required": true
    },
    "policy": {
      "type": "String"
    }
  },
  "relations": {
    "nexus": {
      "type": "belongsTo",
      "model": "Nexus"
    },
    "image": {
      "type": "hasOne",
      "model": "Image"
    }
  }
}
```

- *tasks*:

```
{
  "name": "Task",
  "properties": {
    "data": {
      "type": "Object"
    },
    "handler": {
      "type": "String"
    },
    "created_at": {
      "type": "Date"
    },
    "priority": {
```

```
      "type": "String",
      "default": "low",
      "enum": ["low", "medium", "high"]
    },
    "last_retry_at": {
      "type": "Date"
    },
    "retry_count": {
      "type": "Number"
    },
    "done_at": {
      "type": "Date"
    },
    "done": {
      "type": "Boolean"
    }
  },
  "relations": {}
}
```

- *taxes*:

```
{
  "name": "Tax",
  "properties": {
    "name": {
      "type": "String"
    },
    "description": {
      "type": "String"
    },
    "reason":{
      "type": "String"
    },
    "import": {
      "type": "Number"
    }
  },
  "relations": {}
}
```

- *vendors*:

```
{
  "name": "Vendor",
  "properties": {
    "name": {
      "type": "String",
      "required": true
    },
    "description": {
      "type": "String"
    }
  },
  "relations": {}
}
```

- *wishlists*:

```
{
  "name": "Wishlist",
  "properties": {
    "product_id": {
      "type": "String"
    },
    "product_variant_id": {
      "type": "String"
    },
    "description": {
      "type": "String"
    }
  },
  "relations": {
    "product": {
      "type": "belongsTo",
      "model": "Product"
    },
    "product_variant": {
      "type": "belongsTo",
      "model": "ProductVariant"
    }
  }
}
```

All of them are exposed as HTTP RESTful API. APIs generated for the basic model Product:

- **POST /products** — Create a new instance of the model and persist it into the data source;
- **GET /products** — Find all instances of the model matched by filter from the data source;
- **PUT /products** — Update an existing model instance or insert a new one into the data source;
- **PUT /products/id** — Update attributes for a model instance and persist it into the data source;
- **GET /products/id** — Find a model instance by id from the data source;
- **DELETE /products/id** — Delete a model instance by id from the data source;
- **GET /products/count** — Count instances of the model matched by where from the data source;
- **GET /products/findOne** - Find first instance of the model matched by filter from the data source;
- **POST /products/update** — Update instances of the model matched by where from that data source;
- **GET /products/id/collections** — Queries collections of Product. A product has a relationship with collection type: hasAndBelongsToMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;

- **GET /products/id/comments** — Queries comments of Product. A product has a relationship with Comment type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;
- **POST /products/id/images** — Queries images of Product. A product has a relationship with Image type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;
- **POST /products/id/options** — Queries options of Product. A product has a relationship with ProductOption type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;
- **POST /products/id/product_type** — Fetch belongTo relation product_type. A product has a relationship with ProductType: belongTo.
- **GET /products/id/variants** — Queries variants of Product. A product has a relationship with ProductVariant type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;
- **GET /products/id/vendor** — Fetch belongTo relation vendor. A product has a relationship with Vendor: belongTo.

Finally, APIs generated for the model Customer:

- **POST /customers** — Create a new instance of the model and persist it into the data source;
- **GET /customers** — Find all instances of the model matched by filter from the data source;

- **PUT /customers** — Update an existing model instance or insert a new one into the data source;
- **PUT /customers/id** — Update attributes for a model instance and persist it into the data source;
- **GET /customers/id** — Find a model instance by id from the data source;
- **DELETE /customers/id** — Delete a model instance by id from the data source;
- **GET /customers/count** — Count instances of the model matched by where from the data source;
- **GET /customers/findOne** - Find first instance of the model matched by filter from the data source;
- **POST /customers/update** — Update instances of the model matched by where from that data source;
- **GET /customers/id/shipping_addresses** — Queries shipping_addresses of Customer. A Customer has a relationship with Address type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET;
- **GET /customers/id/wishlists** — Queries wishlists of Customer. A Customer has a relationship with Wishlist type: hasMany. Loopback then it generates all possible API to manage the relations. In this case it is shown only the GET.

4.5.3 3th step - UI components definition

Several components are defined at this stage to compose a page. As already mentioned, these components are often reusable.

This idea of composing page composing elements helps to intelligently manage the complexity of the pages. Following are some elements that make up the page of a product:

- `<part-product-info product="{{product}}" error="{{error}}"> </part-product-info>`:

  This element manages information based on a product such as the name and description. This management and all its complexity is encapsulated in the item itself. in this case all elements are passed also two parameters that help to bind variables. So at the page level, the page goes to each element information related to a product and each element internally manages to manipulate the data that interests. Similarly, each element is also passed an object that represents the error, and each element controls and displays errors related to their corporate data;

- `<part-product-pricing product="{{product}}"></part-product-pricing>`

  this component is in charge of the product price;

- `<part-product-actions></part-product-actions>`

  This element is responsible to represent the buttons on rescue and cancellation of a product.

Decomposing the complexity of the design elements in self-contained, it is easy to create complex parts of individual pages.

4.5.4 4th step - UI components assembly

Distinct UI components are finally mounted to compose the application views. Assembly is kept as simple as possible: it only consists of a composition of HTML5 elements. So, the entire development process results driven by: JSON documents describing entities of the application and HTML

template documents describing the UI components. Following is shown the compositions for page elements of a product:

```
<part-product-header product="{{product}}"></part-product-header>
<part-product-info product="{{product}}" error="{{error}}"></part-product-
    info>
<part-product-visibility product="{{product}}" error="{{error}}" on-change
    -visibility="on_change_visibility"></part-product-visibility>
<part-product-image product="{{product}}" on-uploaded-image="
    on_upload_image"></part-product-image>
<part-product-pricing product="{{product}}"></part-product-pricing>
<part-product-inventory product="{{product}}"></part-product-inventory>
<part-product-shipping product="{{product}}"></part-product-shipping>
<part-product-organization product="{{product}}" on-error="on_error"></
    part-product-organization>
<part-product-variants id="variants" product="{{product}}"></part-product-
    variants>
<part-product-actions></part-product-actions>
```

All these elements are in turn used in the interior of the element <layout-admin> that defines the menu of the page. This design of the product page about the side-admin. The same philosophy to arrange everything in

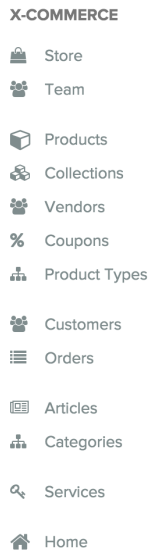

Figure 4.4: Admin navbar

components has also been used for the client-side.

Following shows some screenshots of the pages that make up the admin interface:

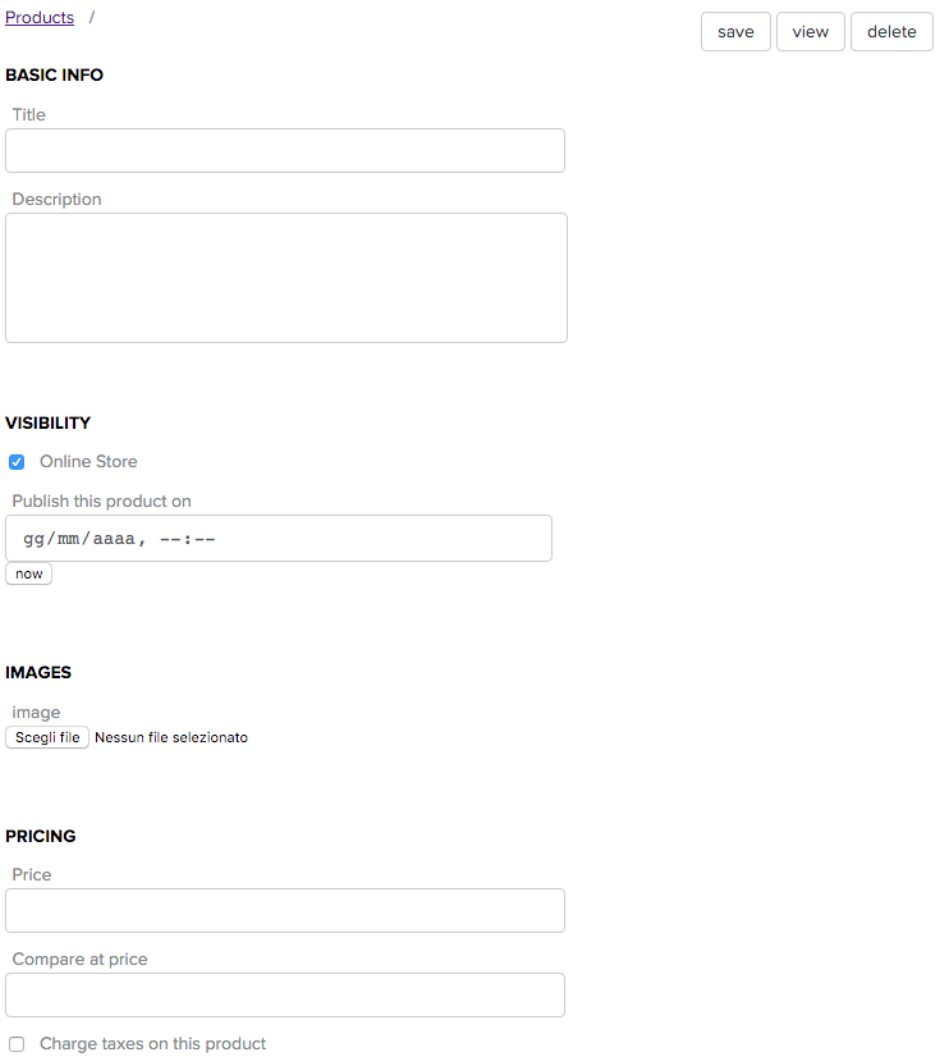

Figure 4.5: Page product example - first part form interface

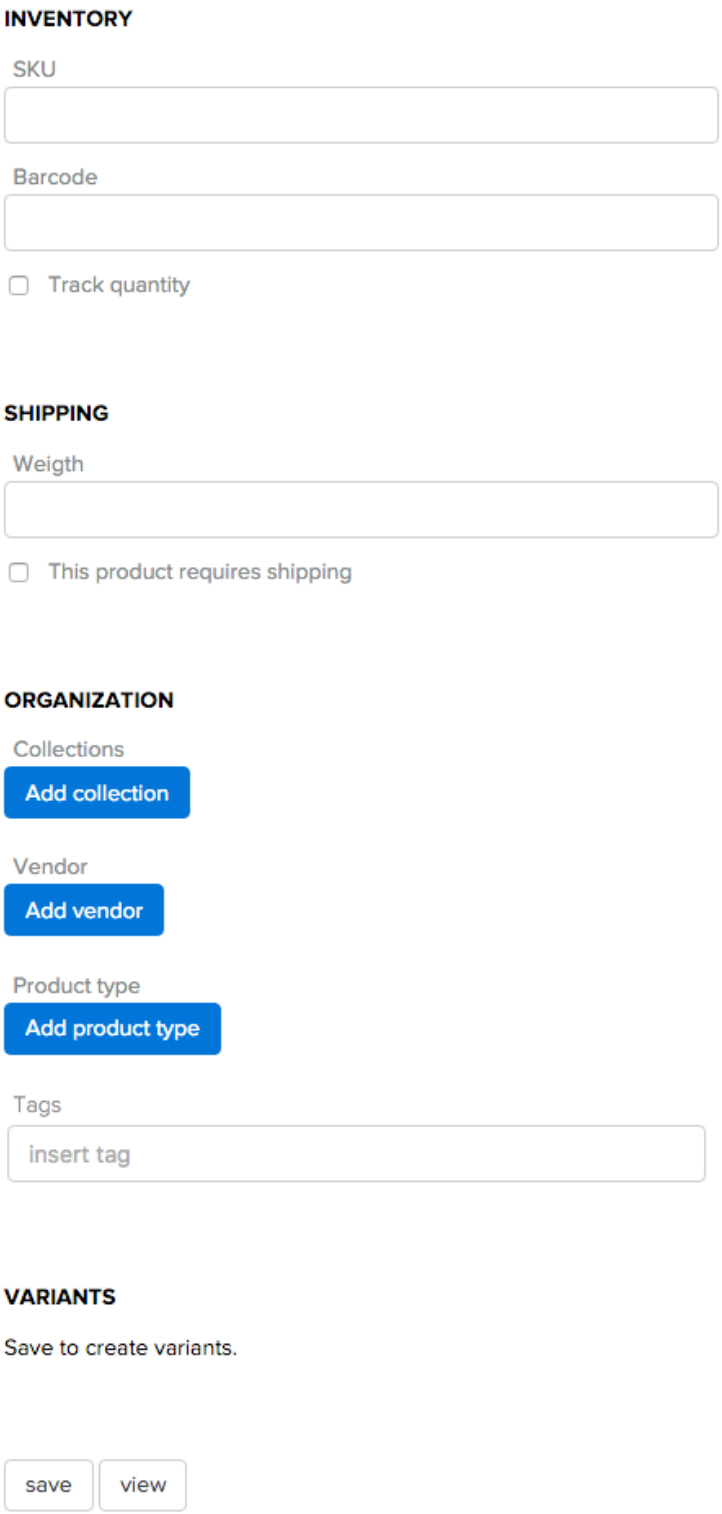

Figure 4.6: Page product example - second part form interface

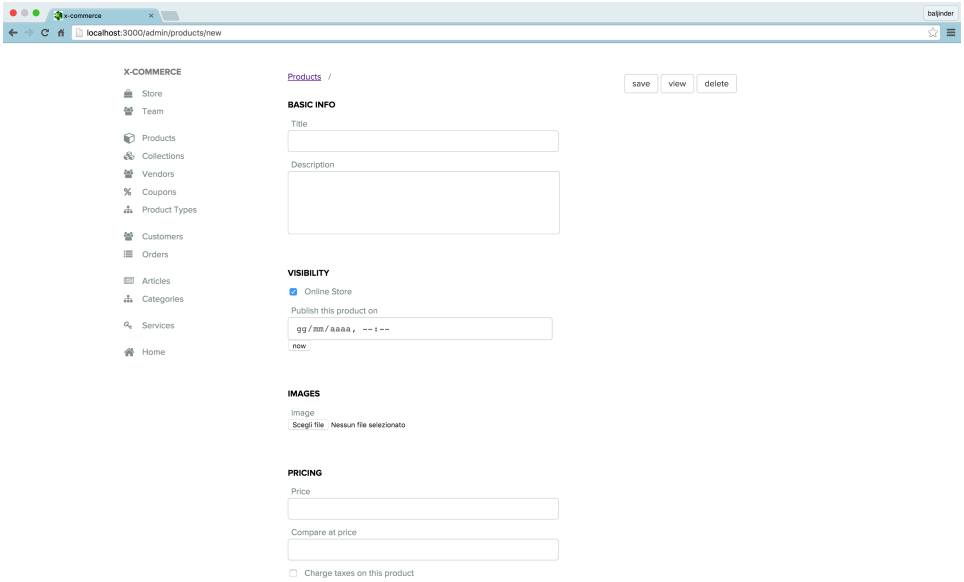

Figure 4.7: Page product example - first part form interface - Browser screenshot

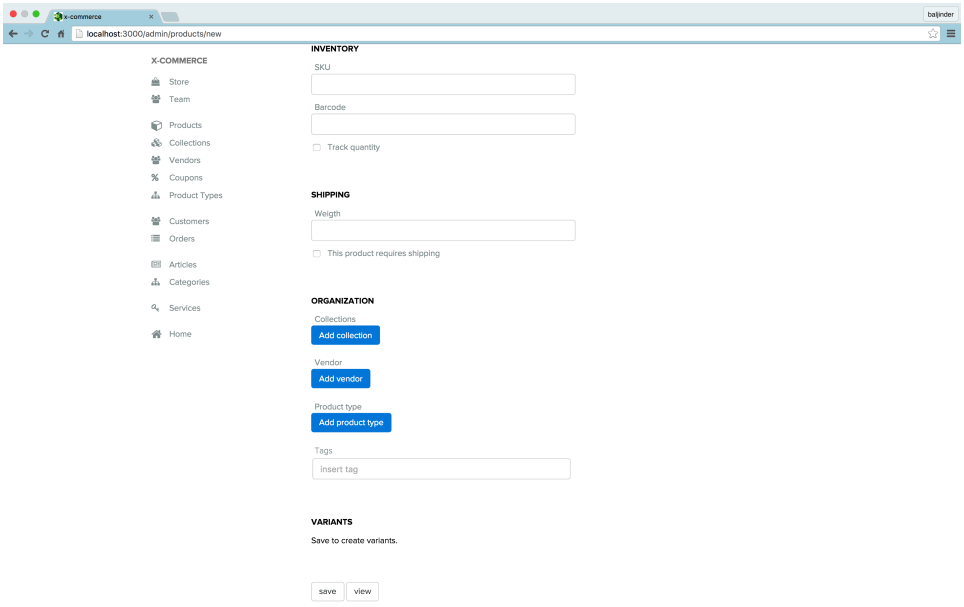

Figure 4.8: Page product example - second part form interface- Browser screenshot

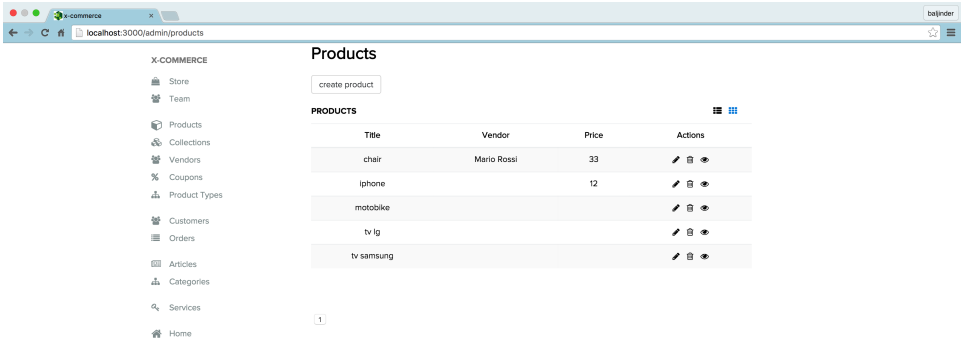

Figure 4.9: Page products example

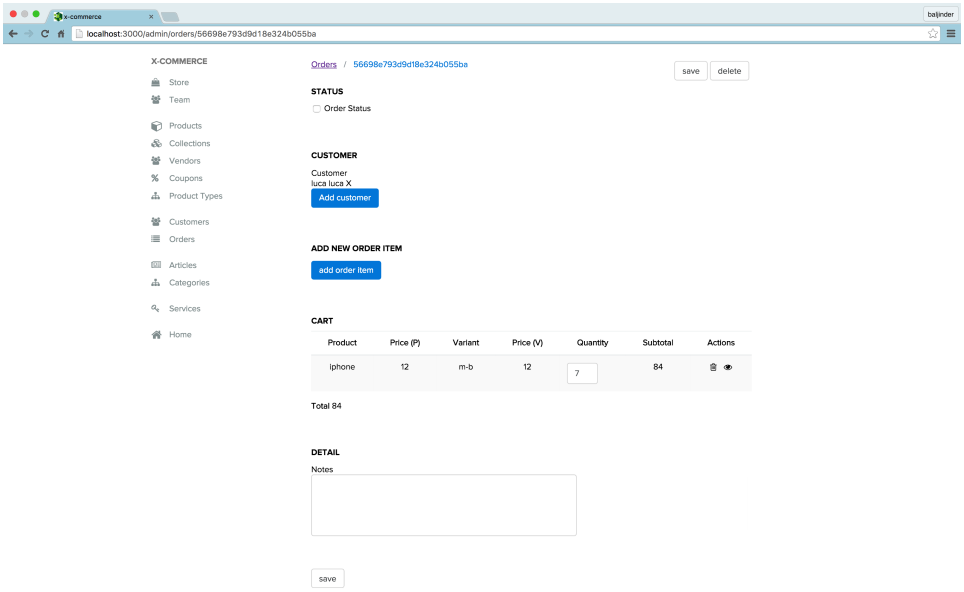

Figure 4.10: Page order example

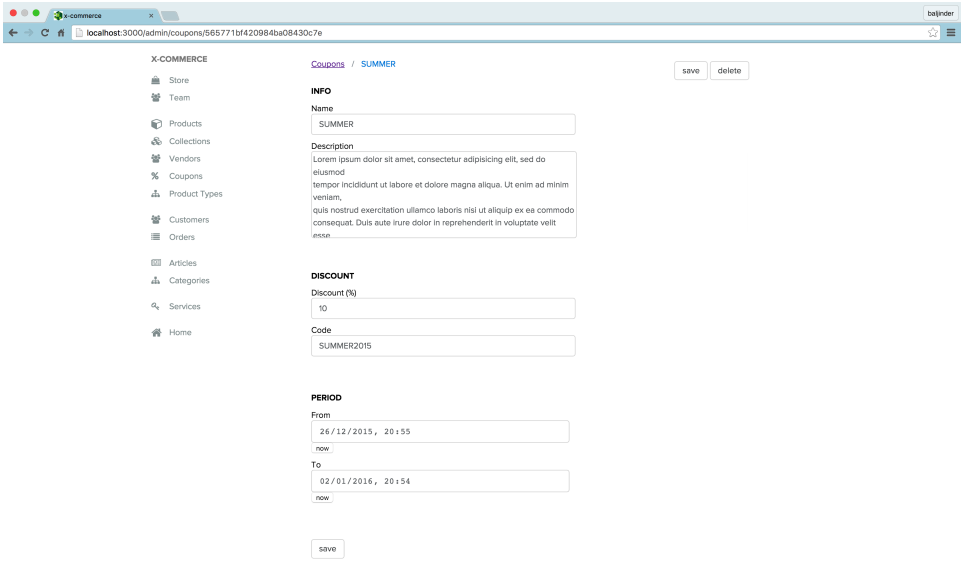

Figure 4.11: Page coupon example

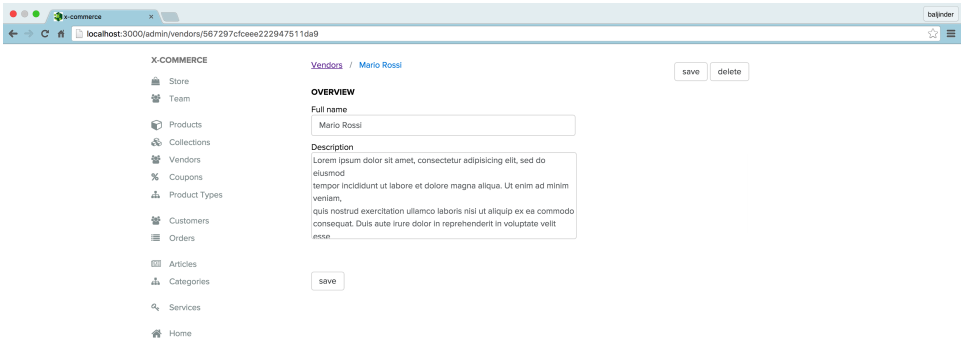

Figure 4.12: Page vendor example

Chapter 5

Payment management component

A payment system is any system used to settle financial transactions through the transfer of monetary value, and includes the institutions, instruments, people, rules, procedures, standards, and technologies that make such an exchange possible.

What makes a payment system a system is the use of cash-substitutes; traditional payment systems are negotiable instruments such as drafts (e.g., checks) and documentary credits such as letters of credit. With the advent of computers and electronic communications a large number of alternative electronic payment systems have emerged. These include debit cards, credit cards, electronic funds transfers, direct credits, direct debits, internet banking and e-commerce payment systems.

Payment systems are used in lieu of tendering cash in domestic and international transactions and consist of a major service provided by banks and other financial institutions.

This chapter describes how they are integrated payment services in x-commerce. In particular, the first paragraph sets out some principals companies that provide services like payment service provider. In the following paragraphs

we will enter more and more in detail in the use of these services.

5.1 x-commerce payment system overview

In the management of payments of e-commerce, there are three major players involved:

- *x-commmerce client*: it is the one who initiates the transaction. The customer enters their credit card details in the specific form and send those to the server of x-commerce;
- *x-commmerce server* the x-commerce server verifies the data received from the client and communicates with the server to start a transaction of Braintree;
- *Braintree server* The server braintree also called a "gateway" initiates the transaction on the basis of the coordinates of the credit card. The needed results of the operation is sent to the server of x-commerce which in turn notifies the customer.

5.1.1 Braintree customized form

In section 3.1 shows how to set the base form provided by Braintree to integrate payments into your application. Following example shows the code to create a custom form:

```
<form action="/api/orders/checkout" id="form_card">
  <div>
    <label for="card-number">Card Number</label>
    <input type="text" data-braintree-name="number">
  </div>
  <div>
    <label for="expiration-date">Expiration Date</label>
    <input type="number" data-braintree-name="expiration_month">
    <input type="number" data-braintree-name="expiration_year">
  </div>
```

```
  <input id="pay" type="submit" value="pay">
</form>
<script src="https://js.braintreegateway.com/v2/braintree.js"></script>
<script>
  braintree.setup("YOUR_CLIENT_TOKEN", "custom", { id: "form_card" });
</script>
```

This way you can customize the form with CSS code. Note that the first thing that comes to imported script *braintree.js*. This library includes all the logic for the management of sending data. In particular, the data on credit cards do not travel on the network as this script first hides such data with a token. Network so traveling a token that summarizes the data of the credit card. A server-side that is sufficient token to decode the token to get clear in the coordinates of the credit card. This token is generated from the keys and other information obtained from Braintree and is unique. In fact if you start two transactions and both have the same token then the second is discarded because it was considered duplicated. Immediately after the values are dictated to the environment variable "braintree".

```
<script>
  braintree.setup("YOUR_CLIENT_TOKEN", "custom", { id: "form_card" });
</script>
```

Where:

- *YOUR_ CLIENT_ TOKEN*: it's the ID of braintree's client that get with an AJAX call. The customer must be registered in Braintree. Braintree provides APIs to register a new account and get ID assigned to it;
- *custom*: it indicates that the form of payment is not the default (*dropin*) but is customized;
- *id*: specific ID of the form that must be processed by the library *braintree.js* to submit the form.

Finally the script *braintree.js*, to process the data of credit cards, requires that each input field has certain characteristics.

```
<div>
<label for="card-number">Card Number</label>
<input type="text" data-braintree-name="number"></div>
<div>
<label for="expiration-date">Expiration Date</label>
<input type="number" data-braintree-name="expiration_month">
<input type="number" data-braintree-name="expiration_year">
</div>
```

Where:

- *data-braintree-name="number"*: it's need to add this property to the input field of the form. This input field contains the number of the credit card that is used by the script *braintree.js* to generate the token. This is necessary because the form is submitted, the script braintree.js parses the data entered in the form in particular select input fields having these specific properties;
- *data-braintree-name="expiration_month"*: it's need to add this property to the input field of the form that will contain the month of expiration of the credit card;
- *data-braintree-name="expiration_year"*: it's need to add this property to the input field of the form that will contain the year of expiration of the credit card.

So, in this way the customer x-commerce is able to send the data of the credit card to the server of x-commerce, then start a payment transaction.

5.1.2 Payment transaction initialization - server side

For initilize, the x-commerce server must import the Braintree di module. This module must be initialized with the keys that uniquely identify the merchant.

These keys are obtained from braintree and are used to make authentication on Braintree and are the following:

- *merchant ID*: identifies the merchant
- *public key*: It is the public key of the asymmetric encryption used in sending data from client to server;
- *private key*: It is the public key of the asymmetric encryption used to decrypt the data;

In the following is a function that, thanks to these keys, it connects to the gateway braintree:

```
var gateway;
function connect_braintree () {
  return new Promise(function (resolve, reject) {
    if (gateway) {
      resolve(gateway);
      return;
    }
    get_service('braintree').then(function (service) {
      gateway = braintree.connect({
        environment: braintree.Environment.Sandbox,
        merchantId: service.params.merchant_id,
        publicKey: service.public_key,
        privateKey: service.private_key
      });
      resolve(gateway);
    }).catch(function (err) {
      reject(err);
    });
  });
}
```

In this way, through the variable *gateway*, it is possible to communicate with braintree.

Once you are done with authentication braintree and received the token (*payment_ method_ none*), which summarizes the data of the credit card, one

can try to make the payment.
When the customer does submit the payment form, it is call API: */api/orders/checkout*. At the call of this API, a server-side function is called *checkout_ braintree*, which performs the following functions:

1. get the customer from his ID. A query is made to the database that returns the customer associated with the ID;
2. creation a new order as required by the customer. To do this, run a series of operations such as block related products due cause, occurs if the customer used a coupon, etc..;
3. communicates with braintree server for initilize a new payment trasction;
4. creating review to allow the customer to leave a review for each product order;
5. mark current order as closed;
6. save the response of Braintree to keep track of the attempted payment;
7. creation of a new bill to be sent to the customer;
8. save the data required to retry the payment in case the first attempt went wrong;
9. prepare response for showing to client;

The point 3, as already said, start a new transaction to try the payment. Following example shows the code for this:

```
var braintree_checkout = function (data) {
  return function (next) {
    connect_braintree().then(function (gateway) { // connects with
        Braintree server
      var sale_data = {
        amount: 1,
```

```
        paymentMethodNonce: data.payment_method_nonce,
        options: {
          submitForSettlement: true
        }
      };
      gateway.transaction.sale(sale_data, function (err, res) {
        next(err, res);
      );
    }).catch(function (err){
        next(err, null);
    });
  };
};
```

Where:

1. connection with the server Braintree;
2. preparing payment data such as the amount, payment_method_nonce encoding information of the credit card use by the customer;
3. finally send payment;

Once you send a payment, the following operations to be performed depend on the outcome of the transaction. In particular, if the transaction is successful then you must carry out steps 4, 5, 7, 8, 9 otherwise runs the operation of point 6 which repeats the whole procedure.
In the next section it describes in detail the operation that is performed in point 6.

5.2 Execute tasks to retry payment

The payment of an order can have different outcomes in particular:

- It can fail for the following reasons:
  - error in data entry of credit card;

  - for network problems;
  - insufficient credit;
  - unknown reasons;

- transaction successfully completed;

In each of these cases the customer e-commerce must be notified of the outcome of the transaction.
In the case in which the customer inserts the data of the credit card incorrect then it is immediately alerted.
If the data on the card are correct but there are other problems for which the first transaction fails to complete for reasons listed above, Braintree then returns a reply containing identifier of the failed transaction.
This transaction ID (and other information) is stored in the DB to retry payment in particular is used to store the template *task* this information. In particular, if a payment transaction fails then the server x-commerce creates a instance of the task model with the following data:

```
var get_task_braintree = function (data) {
  var date_now = moment().format().split('+')[0] + 'Z';
  var task = {
    data: {
      order_id: data.order.id,
      transaction_id: data.payment_status.transaction.id,
      customer_id: data.customer.id,
      payment_system: 'braintree'
    },
    handler: 'retry_payment',
    created_at: date_now,
    priority: 'medium',
    last_retry_at: date_now,
    retry_count: 1,
    done: false
  };
  return task;
};
```

When starts of x-commerce starts a cron that appropriate and regular intervals starts and checks if there are tasks to be performed. A Cron is a time-based job scheduler in Unix-like computer operating systems. The function of this scheduler is to verify the presence of tasks to be performed in particular:

- if there are no tasks to be performed then the crohn falls asleep;
- if there are any cron task then it takes all tasks and we select a task to be carried out with a policy implemented in the following function:

```
var get_next_task = function (tasks) {
  var test = false;
  var task = null;
  for(var i=0; i < tasks.length && !test; i++) {
    var last_retry_at = new Date(tasks[i].last_retry_at)
    var date_now = new Date(moment().format().split('+')[0] + 'Z');
    var minutes_past = (date_now - last_retry_at)/1000/60;
    if (minutes_past > Math.pow(tasks[i].retry_count, 4.09)) {
      task = tasks[i];
      test = true;
    }
  }
  return task;
};
```

  Selecting a task to perform depends on two main factors:

  - the number of attempts to retry the task;
  - the time since the last time the task was executed

  In particular, the probability that a task is selected decreases as the number of attempts made for the task. For example:

  1. if a task with the retry_count $= 0 \Rightarrow 0^{4,09} = 1$. Then this task is selected to run if it is spend at least one minute;
  2. if a task with the retry_count $= 1 \Rightarrow 1^{4,09} = 4,09$. Then this task is selected to run if they spent at least 4,09 minutes;

3. if a task with the retry_count = 2 => $2^{4,09} = 17,02$. Then this task is selected to run if they spent at least 17.02 minutes;
4. if a task with the retry_count = 3 => $3^{4,09} = 89,41$. Then this task is selected to run if they spent at least 89,41 minutes;
5. if a task with the retry_count = 4 => $4^{4,09} = 290,01$. Then this task is selected to run if they spent at least 290,01 minutes;

As you can see, a new task is now selected to be tried again. Instead other tasks that continue to fail will gradually discarded.
This idea to try to make payment by executing the task is very important namely when the customer has entered the data of the credit card and did checkout then it is the responsibility of the administrator of the platform, the transaction was completed successfully without requiring the customer to try again.
This is appropriate because the customer could change his mind if you continue to ask him to refuse the payment. So it is advisable that the customer confirms the payment once only and all that needs to be managed at the server side.

5.3 Payment component

Until now it has been shown how it works internally handling payments. Let's see how to mask this complexity in integrating the payment system in x-commerce.
All logic of the payment is hidden in the component *payment-braintree*.
Following is illustrated in detail the use of the component *payment-braintree*:

```
<payment-braintree id="braintree"></payment-braintree>
```

So just import this tag to integrate into your system the payments system. All this thanks to the web components.
The result of having imported this component is the following: The man-

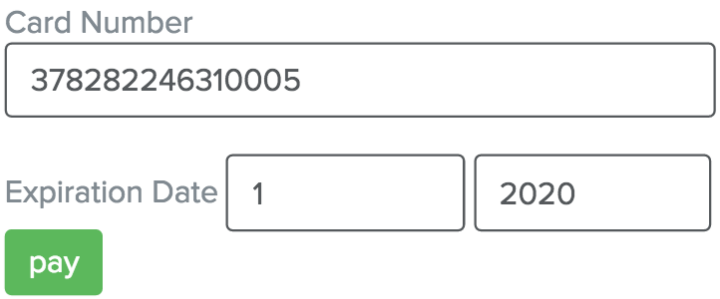

Figure 5.1: Braintree form for payment

agement of payments with Stripe everything works the same way but does not implement the logic of gestone task when paying anadato bad. In fact, to import the payment form stripe just include the following tag:

```
<payment-stripe id="stripe"></payment-stripe>
```

The result of having imported this component is the following:

Figure 5.2: Stripe form for payment

Chapter 6

Conclusions and future work

In this chapter...

6.1 Section 1

In this section...

6.2 Section 2

In this section...

In conclusions...

Part III

Appendix

Appendix A

Appendix 1

In this appendix...

This is the appendix.

Appendix B

Appendix 2

In this appendix...

This is the appendix.