



Università degli Studi “*Roma Tre*”

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di laurea magistrale

***X-Project - Document-Oriented HTML5  
Application Toolkit Based on Web Components***

Laureando

Andrea D’Amelio, Tiziano Sperati

Relatore

Prof. Alberto Paoluzzi

Correlatori

Dott. Enrico Marino,

Dott. Federico Spini

Anno Accademico 2014/2015



*Dedicated to our families*

# Contents

<b>Acknowledgements</b>	<b>vii</b>
<b>Introduction</b>	<b>viii</b>
<b>Introduzione</b>	<b>xi</b>
<b>I Part 1</b>	<b>1</b>
<b>1 Content Management System</b>	<b>2</b>
1.1 CMS overview . . . . .	2
1.2 CMS classification . . . . .	4
1.3 Joomla! . . . . .	5
1.3.1 History . . . . .	5
1.3.2 Extensions . . . . .	7
1.4 Wordpress . . . . .	8
1.4.1 History . . . . .	8
1.4.2 Themes . . . . .	9
1.4.3 Plugins . . . . .	10
1.5 KeystoneJS . . . . .	10
1.6 Ghost . . . . .	12
1.6.1 The Marketplace . . . . .	13

<b>2</b>	<b>Enabling Technologies</b>	<b>14</b>
2.1	HTML 5 . . . . .	14
2.2	Web Components . . . . .	16
2.3	Polymer . . . . .	19
2.4	NodeJS . . . . .	21
2.5	MongoDB . . . . .	23
2.6	StrongLoop LoopBack . . . . .	24
<b>3</b>	<b>Single Page Application</b>	<b>29</b>
3.1	Single Page Application . . . . .	29
3.2	Single Page Application: pros and cons . . . . .	30
3.2.1	Search engine optimization (SEO) . . . . .	31
3.2.2	Client/Server code partitioning . . . . .	33
3.2.3	Analytics . . . . .	34
3.2.4	Speed of initial load . . . . .	35
3.3	Single Page Application: how it works . . . . .	35
3.3.1	Local Routing . . . . .	36
3.3.2	Server architecture . . . . .	36
<b>II</b>	<b>Part 2</b>	<b>39</b>
<b>4</b>	<b>X-Project</b>	<b>40</b>
4.1	X-Project overview . . . . .	40
4.2	Why the name? . . . . .	42
4.3	User Management example: login . . . . .	42
4.4	X-Project Architecture . . . . .	44
4.4.1	Server side . . . . .	44
4.4.2	Client side . . . . .	45
4.5	Document-Driven Web Development Process . . . . .	45
4.5.1	1st step - Models schemas definition . . . . .	46

4.5.2	2nd step - HTTP RESTful API definition . . . . .	46
4.5.3	3rd step - UI components definition . . . . .	47
4.5.4	4th step - UI components assembly . . . . .	47
4.6	X-Elements . . . . .	47
4.6.1	Elements for local routing . . . . .	48
4.6.2	Elements for API . . . . .	48
4.6.3	Elements for forms . . . . .	50
4.6.4	Elements for lists . . . . .	51
4.6.5	Elements for style . . . . .	51
4.6.6	Elements for admin panel . . . . .	52
4.6.7	Elements for pages . . . . .	52
<b>5</b>	<b>Media Management: S3 Component</b>	<b>54</b>
5.1	Amazon AWS . . . . .	54
5.2	Amazon S3 . . . . .	55
5.3	CORS . . . . .	56
5.4	Digital Signature . . . . .	58
5.4.1	How digital signatures work . . . . .	58
5.5	Media Management S3 Component . . . . .	59
5.5.1	S3 Component elements . . . . .	63
5.6	S3 Component - Example . . . . .	66
5.6.1	Example 1 - part-s3-upload . . . . .	66
5.6.2	Example 2 - part-list . . . . .	67
5.6.3	Example 3 - part-list-item . . . . .	68
<b>6</b>	<b>User Management: Login toolkit</b>	<b>71</b>
6.1	User management . . . . .	71
6.1.1	User Management services . . . . .	72
6.1.2	User Management standard API in LoopBack . . . . .	75
6.1.3	User Management remote methods . . . . .	75

<i>CONTENTS</i>	iv
6.2 User Managament elements . . . . .	77
6.3 Mandrill . . . . .	82
<b>7 Use Case: X-Blog</b>	<b>85</b>
7.1 Case study . . . . .	85
7.1.1 1st step - Models schemas definition . . . . .	85
7.1.2 2nd step - HTTP RESTful API definition . . . . .	87
7.1.3 3rd step - UI components definition . . . . .	94
7.1.4 4th step - UI components assembly . . . . .	95
<b>8 Conclusions</b>	<b>100</b>
8.1 Work performed . . . . .	100
8.2 Future developments . . . . .	101
<b>Bibliography</b>	<b>103</b>

# List of Figures

1.1	CMS's actors schema . . . . .	3
1.2	Joomla! Dashboard . . . . .	8
1.3	WordPress Dashboard . . . . .	11
1.4	KeystoneJS Dashboard . . . . .	12
1.5	Ghost Dashboard . . . . .	13
2.1	Server and client sides enabling technologies . . . . .	15
2.2	Html5 Responsive . . . . .	16
2.3	Polymer Architecture . . . . .	20
2.4	MongoDB Architecture . . . . .	24
2.5	LoopBack Architecture . . . . .	26
3.1	The Traditional Page Lifecycle . . . . .	31
3.2	The SPA Lifecycle . . . . .	31
4.1	Login Element Example . . . . .	44
4.2	X-Project Architectural Stack . . . . .	46
5.1	Flowchart showing Simple and Preflight XHR by Bluesmoon .	57
5.2	Digital Signature Process . . . . .	59
5.3	S3 direct upload process . . . . .	62
5.4	S3 CORS Configuration Editor Panel . . . . .	63
5.5	S3 Component example - Upload and preview functions . . .	70



6.1	Stormpath User Management API . . . . .	73
6.2	Auth0 Overview . . . . .	74
6.3	Login Element . . . . .	78
6.4	Signup Element . . . . .	79
6.5	Reset Element . . . . .	80
6.6	Login Element . . . . .	80
6.7	Settings Element . . . . .	83
7.1	Screenshot: X-Blog - Edit an article . . . . .	97
7.2	Screenshot: X-Blog - Articles page . . . . .	98
7.3	Screenshot: X-Blog - Article page . . . . .	99

# Acknowledgements

Firstly, we would like to express our sincere gratitude to our advisors, Enrico Marino and Federico Spini, for the continuous support of our project, for their patience, motivation, and immense knowledge. Besides our advisors, we would like to thank Prof. Alberto Paoluzzi for his supervision and the opportunity that he offered to us. Moreover we would like to thank all CVDLAB labmates, especially the Marcos.

# Introduction

Since the beginning of Internet, the ability to create and publish content on the web has made the success of Content Management Systems. Products like Joomla! or WordPress, born to handle simple websites or blogs, have evolved to support web applications of any sort (from personal portfolio to online shopping). running as of January 2015 more than 25% of the top ten million websites [30] has been managed from the major CMSs. This evolution has been allowed by a plug-in based architecture, where each plug-in is responsible for handling a functionality subset of the whole application, presenting the user through a simple accessible configuration and management interface. The large number of available plug-ins covers most of the common and frequently required customizations, thus avoiding writing ad-hoc code. Nevertheless, the implementation of specific functional characteristics inevitably requires to intervene at code level.

When the effort required to add custom features to a CMS results too expensive, a web framework can be adopted instead. A web framework consists of a set of software facilities aimed to alleviating the overhead associated with common development activities. Web application coding effort, while eased by the web framework, is anyway rewarded with an increased level of extensibility and customizability of the resulting application. The most desirable features for a web framework are user management, session management, database access via HTTP RESTful API. In order to effectively speed up web applications development, these facilities should be provided

relying mostly on external configuration files and less on procedural code [21].

In this thesis a software platform named X-Project is introduced. It consists of a Web Component library, a development methodology and a set of guidelines.

The Web Components library is applied over a powerful web framework, Loopback by Strongloop, and creates an hybrid prototypal tool which brings together the customizability of a modern web framework with the ease of use of traditional CMSs.

A document-driven development process has been naturally prompt by the use of the toolkit. This methodology polarizes the concept of reusing the code whose overall readability, maintainability and extendibility result dramatically increased.

Guidelines have been drafted in order to provide users with the aid to get straight to the entire structure.

First goal of this project is the creation of a platform that can wrap the features of CMSs and web application frameworks. X-Project aims to simplify developers' life in creating web applications and to give them the chance to easily compose, via web components libraries, pages made of reused components. Reusability and access to reusable components are important features that X-Project borrows from Web Components standard. As said further in 4.1, "Everything is an element, even a service" is the X-Project's philosophy: every kind of function or structure is encapsulated in Polymer elements, and brings the Web Components concept to extreme levels. Moreover, by using Single Page Application pattern, X-Project tries to streamline loading times of web applications.

The choice of cutting edge technologies, such as Polymer-Project by Google, has been made both to meet the challenge of new and unknown development pattern, and to test the newest technologies.

X-Project is located at the end of a natural path composed of user oriented CMSs and developer oriented web application framework (see 1.2), trying to fuse CMSs ease of use with web application frameworks customization level.

The rest of this document is divided in two parts and organized as follows. Part One consists of three chapters. Chapter One describes the state of the art of CMS's world, giving a classification and focuses on four of the major CMSs on the market. Chapter Two analyzes every used technology and describes the methodological approach of each one. Chapter Three provides an overview of Single Page Application development pattern and explains pros and cons and technical functioning.

Part two is composed of four chapters. Chapter Four describes the project in its entirety, by exposing the core of X-Project, its functionalities and its features. Chapters Five and Six focus on specific components that have been developed, Media Management S3 Component and User Management Component, and on all the theoretical and practical concepts that are behind the ideation of the component. Chapter Seven presents a real use case and highlights, by means of configuration steps and code snippets the major aspects of a web application implementation with X-Project. Finally, Chapter Eight, exposes project conclusions and further implementations of the work.

# Introduzione

Sin da le origini di Internet, la possibilità di creare e pubblicare contenuti sul web ha fatto sì che i Content Management Systems prendessero piede e avessero successo. Prodotti come Joomla! o WordPress, nati per gestire siti web o semplici blog, si sono evoluti per supportare applicazioni web di qualsiasi tipo (dal portfolio personale allo shopping on-line). Da statistiche di gennaio 2015 risulta che oltre il 25 % dei primi dieci milioni di siti web è gestito con i maggiori CMS[30]. Questa evoluzione è stata permessa da un architettura basata su plug-in, dove ogni plug-in è responsabile di gestire un sottoinsieme di funzionalità di tutta l'applicazione, presentandosi all'utente attraverso una semplice interfaccia di configurazione e una gestione estremamente accessibile. Il gran numero di plug-in disponibile copre la maggior parte delle personalizzazioni più comunemente richieste, evitando così la scrittura di codice ad-hoc. Tuttavia, l'attuazione di specifiche caratteristiche funzionali richiede inevitabilmente l'intervento dello sviluppatore a livello di codice.

Quando lo sforzo richiesto per aggiungere funzioni personalizzate a un CMS risulta troppo costoso, è utile adottare un web framework. Un web framework è costituito da una serie di servizi software che mira ad alleviare il lavoro con comuni attività di sviluppo. Lo sforzo di codifica delle Applicazioni Web, sebbene venga facilitato dal web framework, è comunque premiato con una maggiore possibilità di estensione e personalizzazione dell'applicazione risultante. Le caratteristiche più desiderabili per un web framework sono:

la gestione degli utenti, gestione delle sessioni, l'accesso al database tramite HTTP API RESTful. Al fine di velocizzare in modo efficace lo sviluppo di applicazioni web, queste strutture dovrebbero essere fornite basandosi principalmente sul file di configurazione esterni e meno sul codice di procedura [21].

In questa tesi viene introdotta una piattaforma software chiamata X-Project. Essa consiste in una metodologia di sviluppo contornata da una libreria di componenti Web e una serie di linee guida.

La libreria dei componenti Web viene applicata su un web framework potente, Loopback di Strongloop, realizzando uno strumento prototipale ibrido che riunisce la personalizzazione di un framework web moderno con la facilità d'uso dei tradizionali CMS.

Tale libreria è nata in maniera naturale durante l'utilizzo del toolkit e, successivamente, è stato standardizzato un processo di sviluppo documento driven che polarizza il concetto di riuso del codice la cui leggibilità, manutenibilità e estendibilità risulta estremamente aumentata.

Le linee guida sono state redatte in modo da fornire un aiuto agli utenti per avere diritti su tutta la struttura

Primo obiettivo di questo progetto è la creazione di una piattaforma in grado di avvolgere le caratteristiche del CMS e di un web framework per applicazioni. X-Project mira a semplificare la vita agli sviluppatori di applicazioni web nella creazione e dare loro la possibilità di comporre facilmente, tramite le librerie di componenti web, pagine fatte di componenti riutilizzati. Riusabilità e l'accesso ai componenti riutilizzabili è una caratteristica importante che X-Project prende in prestito dallo standard dei Web Components. Come detto in ??, "Tutto è un elemento, anche un servizio" questa è la filosofia di X-Project: ogni tipo di funzione o di struttura è incapsulato in elementi Polymer, portando a livelli estremi il concetto dei componenti web. Inoltre, utilizzando il modello Single Page Application, X-Project cerca di

ottimizzare i tempi di caricamento delle web application.

La scelta di tecnologie all'avanguardia, come Polymer-Project di Google, è stata dettata sia dalla sfida insita nell'uso di pattern di sviluppo sconosciuti, sia dal fascino rappresentato dal "nuovo".

X-Project si colloca alla fine del percorso naturale composto dai CMS orientati all'utente e dai web application framework orientati allo sviluppatore (vedi 1.2), provando a fondere la semplicità d'utilizzo dei CMS con il livello di possibilità di personalizzazione dei web application frameworks.

Questa tesi è suddivisa in due parti ed è organizzata come segue. La Parte Uno è composta da tre capitoli. Il Capitolo Uno descrive lo stato dell'arte del mondo dei CMS, fornendo anche una classificazione per tipo e soffermandosi ad analizzare quattro dei maggiori prodotti presenti sul mercato. Il secondo capitolo analizza le tecnologie utilizzate, descrivendo l'approccio tecnico e metodologico di ognuna di esse. Il terzo capitolo fornisce una panoramica sul pattern di sviluppo detto Single Page Application, esponendone vantaggi e svantaggi ed il funzionamento tecnico.

La seconda Parte è composta da quattro capitoli. Il capitolo quattro descrive il progetto nella sua totalità, esponendo l'anima di X-Project, le funzionalità e le caratteristiche. Il quinto e il sesto capitolo si concentrano su componenti specifici, rispettivamente, il Media Management S3 Component e lo User Management Component, e su tutti i concetti teorici e pratici che si celano dietro all'elemento finale. Il settimo capitolo mostra un caso d'uso evidenziando, con snippet e screenshot, i punti salienti dell'implementazione di una web application tramite X-Project. Infine, l'ottavo ed ultimo capitolo, espone le conclusioni tratte e fornisce dei possibili sviluppi futuri per il progetto.



Part I

Part 1

# Chapter 1

## Content Management System

This chapter describes CMS's world.

In the first section a CMSs overview is provided: what CMSs are, what they are for and how do they work. In the second section it's reported a classification of currently available CMSs. The third and fourth sections focus on user oriented CMSs: respectively Joomla! and Wordpress. Fifth and sixth sections focus on developer oriented CMSs: respectively KeystoneJS and Ghost.org.

### 1.1 CMS overview

This section consists of an overview about Content Management Systems.

A Content Management System is an application, that provides capabilities to multiple users with different levels of permission to manage content, data or information of a website project, or internet application [1].

Managing content refers to creating, editing, archiving, publishing, collaborating on, reporting, distributing website content, data and information. So, a CMS is an application that allows creating and publishing content from a central interface. CMSs are often used to run websites containing blogs, news, and shopping. Many corporate and marketing websites use CMSs.

CMSs typically aim to avoid the need for hand coding, but may help it for specific elements or entire pages [32].

A Web CMS may catalog and index content, select or assemble content at runtime, or deliver content to specific visitors in a requested way, such as other languages. Web Content Management Systems usually allow client control over HyperText Markup Language - based content, files, documents, and Web hosting plans based on the system depth and the niche it serves.

CMSs world involves three main actors: the developer, the admin and the user. Each actor plays a fundamental role in the life of a CMS managed website. The developer is the one who physically creates the website, manages the functionality, designs the graphics and makes provide the Admin with set-up. The administrator is not a technician (not a programmer) who manages the website content: he can create, edit, publish and administer the material to be shown. He manages the contents of the site through an administration panel specifically created by the developer. The administrator works in a reserved area, only accessible via a login panel. Finally, the user is the one who benefits of the contents made available by the admin. User cannot access the administration panel and, therefore, can not create, edit or delete content (sometimes users can comment on existing content).

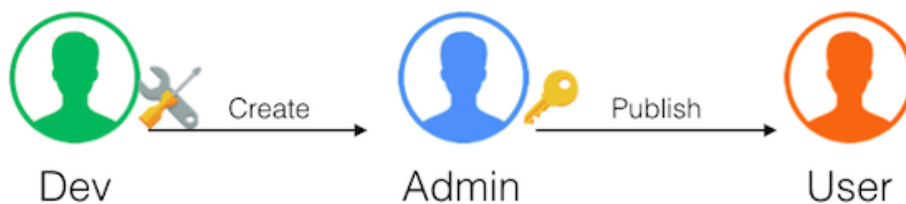


Figure 1.1: CMS's actors schema

## 1.2 CMS classification

The classic and most popular CMS (e.g. Joomla! and Wordpress) have nearly eliminated the role of the developer, by inserting graphics creation tools and providing presets, covering a large part of the most common requests. Other CMSs, that are emerging at this time, have made the developer even stronger, by heavily focusing on the possibility of personalization and customization of the product.

Since the beginning of the Internet, the ability to create and publish content on the web has marked the success of Content Management Systems. Products like Joomla! or WordPress, born to handle simple websites or blogs, have evolved to support web applications of any sort (from personal portfolio to on-line shopping), running as of January 2015 more than 25% of the top ten million websites [30]. This evolution has been allowed by a plug-in based architecture, where each plug-in is responsible to handle a functionality subset of the whole application, by presenting the user through a simple accessible configuration and management interface. The large number of available plug-ins covers most of the common and frequently required customizations, thus avoiding to write ad-hoc code. Nevertheless, the implementation of specific functional characteristics inevitably requires a code-level intervention.

In recent years, however a return to basics has occurred. New CMSs are giving importance to the developer's role. CMSs as KeystoneJS and Ghost.org make the customization a strength. Therefore, they sacrifice the ease of use and the possibility of utilization even for non-experts, in favour of product customization. Nowadays, two categories of CMSs can be distinguished:

- User oriented CMSs: mostly visual and presetted;
- Developer oriented CMSs: customizable frameworks that need written

code to be used by the admin.

## 1.3 Joomla!

Joomla is a free and open-source content management system (CMS) for publishing web content. It is built on a model–view–controller web application framework that can be used independently of the CMS. [33]

Joomla is written in PHP, uses object-oriented programming (OOP) techniques (since version 1.5 [14]) and software design patterns, stores data in a MySQL, MS SQL (since version 2.5), or PostgreSQL (since version 3.0) database [7], and includes features such as page caching, RSS feeds, printable versions of pages, news flashes, blogs, search, and support for language internationalization. As of February 2014, Joomla has been downloaded over 50 million times [17]. Over 7,700 free and commercial extensions are available from the official Joomla! Extension Directory, and more are available from other sources. It is estimated to be the second most used content management system on the Internet, after WordPress [30]. Like many other web applications, Joomla may be run on a LAMP stack. Many web hosts have control panels for automatic installation of Joomla. On Windows, Joomla can be installed using the Microsoft Web Platform Installer, which automatically detects and installs dependencies, such as PHP or MySQL. Many web sites provide information on installing and maintaining Joomla sites.

### 1.3.1 History

Joomla was the result of a fork of Mambo on August 17, 2005. At that time, the Mambo name was a trademark of Miro International Pty Ltd, who formed a non-profit foundation with the stated purpose of funding the project and protecting it from lawsuits. The Joomla development team claimed that many of the provisions of the foundation structure violated previous agree-

ments made by the elected Mambo Steering Committee, lacked the necessary consultation with key stakeholders and included provisions that violated core open source values. Joomla developers created a website called OpenSourceMatters.org (OSM) to distribute information to the software community. Project leader Andrew Eddie wrote a letter that appeared on the announcements section of the public forum at mamboSERVER.com. Over one thousand people joined OpenSourceMatters.org within a day, most posting words of encouragement and support. The website received the Slashdot effect as a result. Miro CEO Peter Lamont responded publicly to the development team in an article titled The Mambo Open Source Controversy, 20 Questions With Miro. This event created controversy within the free software community about the definition of open source. Forums of other open-source projects were active with postings about the actions of both sides. In the two weeks following Eddie's announcement, teams were re-organized and the community continued to grow. Eben Moglen and the Software Freedom Law Center (SFLC) assisted the Joomla core team beginning in August 2005, as indicated by Moglen's blog entry from that date and a related OSM announcement. The SFLC continue to provide legal guidance to the Joomla project. On August 18, Andrew Eddie called for community input to suggest a name for the project. The core team reserved the right for the final naming decision, and chose a name not suggested by the community. On September 22, the new name, Joomla!, was announced. It is the anglicised spelling of the Swahili word jumla meaning all together or as a whole which also has a similar meaning in at least Amharic, Arabic and Urdu. On September 26, the development team called for logo submissions from the community and invited the community to vote on the logo; the team announced the community's decision on September 29. On October 2, brand guidelines, a brand manual, and a set of logo resources were published. Joomla won the Packt Publishing Open Source Content Management System Award in 2006, 2007,

and 2011 [33].

### 1.3.2 Extensions

Joomla extensions extend the functionality of Joomla websites. Five types of extensions may be distinguished: components, modules, plugins, templates, and languages. Each of these extensions handles a specific function.

- Components are the largest and most complex extensions. Most components have two parts: a site part and an administrator part. Every time a Joomla page loads, one component is called to render the main page body. Components produce the major portion of a page because a component is driven by a menu item.
- Plugins are advanced extensions and are, in essence, event handlers. In the execution of any part of Joomla, a module or a component, an event may be triggered. When an event is triggered, plugins that are registered to handle that event execute. For example, a plugin could be used to block user-submitted articles and filter text. The line between plugins and components can sometimes be a little fuzzy. Sometimes large or advanced plugins are called components even though they don't actually render large portions of a page. An SEF URL extension might be created as a component, even though its functionality could be accomplished with just a plugin.
- Templates describe the main design of a Joomla website. While the CMS manages the website content, templates determine the style or look and feel and layout of a site.
- Modules render pages in Joomla. They are linked to Joomla components to display new content or images. Joomla modules look like

boxes, such as the search or login module. However, they don't require html in Joomla to work.

- Languages are very simple extensions that can either be used as a core part or as an extension. Language and font information can also be used for PDF or PSD to Joomla conversions [33].

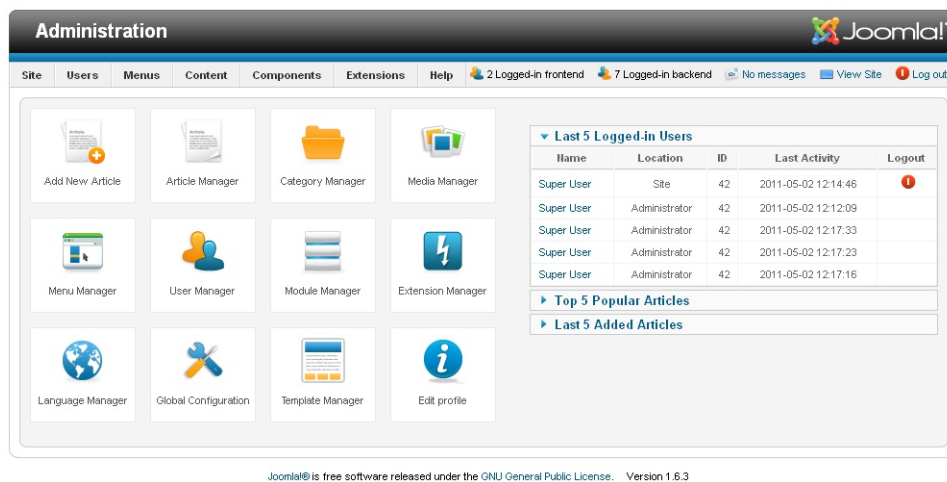


Figure 1.2: Joomla! Dashboard

## 1.4 Wordpress

WordPress is a free and open-source content management system (CMS) based on PHP and MySQL. [36] Features include a plugin architecture and a template system. WordPress was used by more than 23.3% of the top 10 million websites as of January 2015. WordPress is the most popular blogging system in use on the Web, at more than 60 million websites [28].

### 1.4.1 History

It was first released on May 27, 2003, by its founders, Matt Mullenweg and Mike Little, as a fork of b2/cafeblog. The license under which WordPress



software is released is the GPLv2 (or later) from the Free Software Foundation. b2/cafeblog, more commonly known as simply b2 or cafeblog, was the precursor to WordPress. b2/cafeblog was estimated to have been installed on approximately 2,000 blogs as of May 2003. It was written in PHP for use with MySQL by Michel Valdrighi, who is now a contributing developer to WordPress. Although WordPress is the official successor, another project, b2evolution, is also in active development. WordPress first appeared in 2003 as a joint effort between Matt Mullenweg and Mike Little to create a fork of b2. Christine Selleck Tremoulet, a friend of Mullenweg, suggested the name WordPress. In 2004 the licensing terms for the competing Movable Type package were changed by Six Apart, resulting in many of its most influential users migrating to WordPress. By October 2009 the Open Source CMS MarketShare Report concluded that WordPress enjoyed the greatest brand strength of any open-source content-management system [12].

### 1.4.2 Themes

WordPress has a web template system using a template processor. WordPress users may install and switch between themes. Themes allow users to change the look and functionality of a WordPress website and they can be installed without altering the content or health of the site. Every WordPress website requires at least one theme to be present and every theme should be designed using WordPress standards with structured PHP, valid HTML and CSS. Themes may be directly installed using the WordPress Appearance administration tool in the dashboard or theme folders may be uploaded via FTP. The PHP, HTML (HyperText Markup Language) and CSS (Cascading Style Sheets) code found in themes can be added to or edited for providing advanced features. WordPress themes are in general classified into two categories, free themes and premium themes. All the free themes are listed in the WordPress theme directory and premium themes should be purchased

from marketplaces and individual WordPress developers. WordPress users may also create and develop their own custom themes if they have the knowledge and skill to do so. If WordPress users do not have themes development knowledge then they may download and use free WordPress themes from [wordpress.org](http://wordpress.org) [12].

### 1.4.3 Plugins

WordPress's plugin architecture allows users to extend the features and functionality of a website or blog. WordPress has over 39,078 plugins available, each of which offers custom functions and features enabling users to tailor their sites to their specific needs. These customizations range from search engine optimization, to client portals used to display private information to logged in users, to content displaying features, such as the addition of widgets and navigation bars. But not all available plugins are always abreast with the upgrades and, as a result, they may not function properly or may not function at all [12].

## 1.5 KeystoneJS

KeystoneJS is an open source framework for developing database-driven websites, applications and APIs in Node.js, built on Express and MongoDB. KeystoneJS provides a strong backbone for a simple content management system. Keystone.js is available for free and sits under the MIT license. In addition, there are sixty four contributors to the open source repository and daily commits. KeystoneJS runs with two of the most powerful elements of Node.js: Express and MongoDB. Express.js is the most popular web application framework for Node.js and MongoDB is a document based database whose queries are written in JavaScript notation. Learning to use both Express and MongoDB alongside KeystoneJS will allow a user to progress and

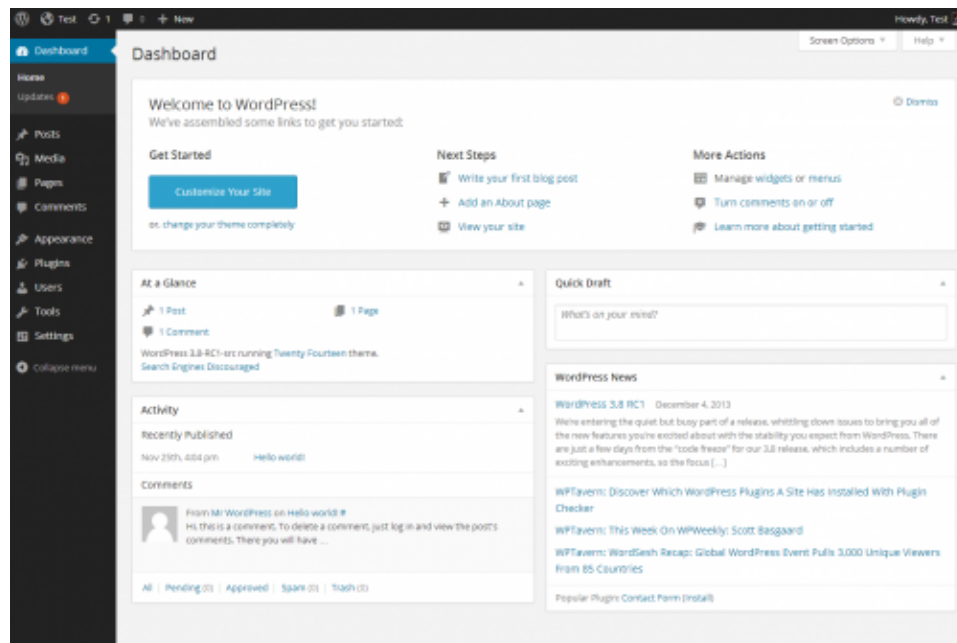


Figure 1.3: WordPress Dashboard

create web applications in Node.js without KeystoneJS. In addition to using popular frameworks with Node.js, the content management system is also built upon an MVC architecture. Learning this Model-View-Controller Architecture will allow a user to take principles away from KeystoneJS and apply them elsewhere in future applications. [27]

There are some key features of KeystoneJS that stand out as needed features of a CMS. The CMS comes prepared with standard data types such as name, email and password which can be easily incorporated into online forms to speed up validation and database storage. KeystoneJS provides an out-of-the-box Admin User Interface which represents the custom fields and content-types created by the user. A user can easily use this to build a website from the ground up, or use this as a CMS during production. Lastly, KeystoneJS provides natural integration with email. It provides a template email system and incorporates integration with MailChimp, a leading mail service for web applications. [15]

The screenshot displays the KeystoneJS Dashboard interface. At the top, a dark blue navigation bar contains links for 'Keystone Demo', 'Posts' (which is highlighted), 'Galleries', 'Enquiries', 'Users', and 'Field Tests'. A 'Sign Out' link is located on the right side of this bar. Below the navigation bar, a light gray header area shows 'Posts' and 'Post Categories'. The main content area has a blue 'Posts' link on the left and a slug field on the right with the value 'cms-without-db-running-keystonejs-without-mongodb' and a 'new post' button. A green success message states 'Your changes have been saved.' Below this, the title of the post is 'CMS without DB – Running KeystoneJS without MongoDB'. The form fields include: 'Slug' (pre-filled with the same slug), 'State' (set to 'Draft'), 'Author' (with a search dropdown showing 'Fred Flintstone' as a suggestion), 'Published Date', 'Image' (with an 'Upload Image' button), and 'Content Brief' (with a rich text editor toolbar containing bold, italic, bulleted list, numbered list, link, unlink, and code icons).

Figure 1.4: KeystoneJS Dashboard

## 1.6 Ghost

Ghost is a fully featured blogging platform built entirely with JavaScript. Ghost is a Node.js application powered by the Express framework. Ghost ships with SQLite, and everything is connected through JugglingDB ORM.

Ghost is available via NPM, so it's really easy to install on major environments. Ghost theming is done with Handlebars, which keeps business and view logic separated. This platform allows to customizing and extending blog with additional functionality via plugins with helpers and data filters. Ghost supports international translations using Node Polyglot.js. Finally, Ghost is released under the MIT License. [13]

The project is organised and run by a small, Non Profit Organisation called the Ghost Foundation, but it is developed in public, by a large group of contributors all over the world.

### 1.6.1 The Marketplace

The Ghost Open Marketplace is a long term project which aims to create a large directory of themes, apps, and resources which have been implemented for Ghost. It's a sort of Apple App store or Google Play store, but filled with both free and premium products which are all for Ghost.

The current iteration of the Marketplace is a placeholder to showcase all of the themes which have already been created for Ghost in this early stage of its life.[22]

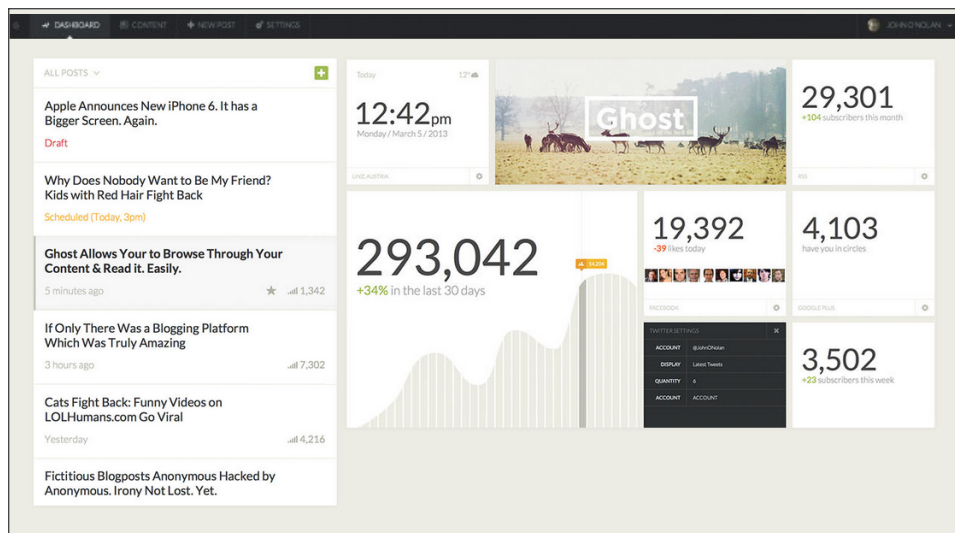


Figure 1.5: Ghost Dashboard

This chapter describes the world of CMSs and provides the analysis and a classification of this environment. Finally, it describes the four main CMSs currently on the market that fit two categories previously identified.

## Chapter 2

# Enabling Technologies

This chapter describes X-Project 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 HTML 5

This section provides an overview of HTML5.

HTML5 is the latest version of Hypertext Markup Language, the code



Figure 2.1: Server and client sides enabling technologies

that describes web pages. There are actually three kinds of code: HTML, which provides the structure; 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.

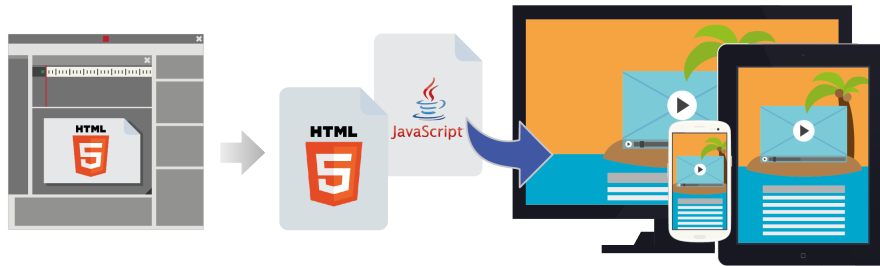


Figure 2.2: Html5 Responsive

Although HTML5 has been well known among web developers 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 COMPONENTS

### TEMPLATES

```
<template id="">
</template>
```

### SHADOW DOM

```
div
  #document-fragment
  span
```

### HTML IMPORTS

```
<link rel="import"
href="part.html">
```

### CUSTOM ELEMENTS

```
<my-elem>
</my-elem>
```

Web Components 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 `<div>` or `<span>` 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 document 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.

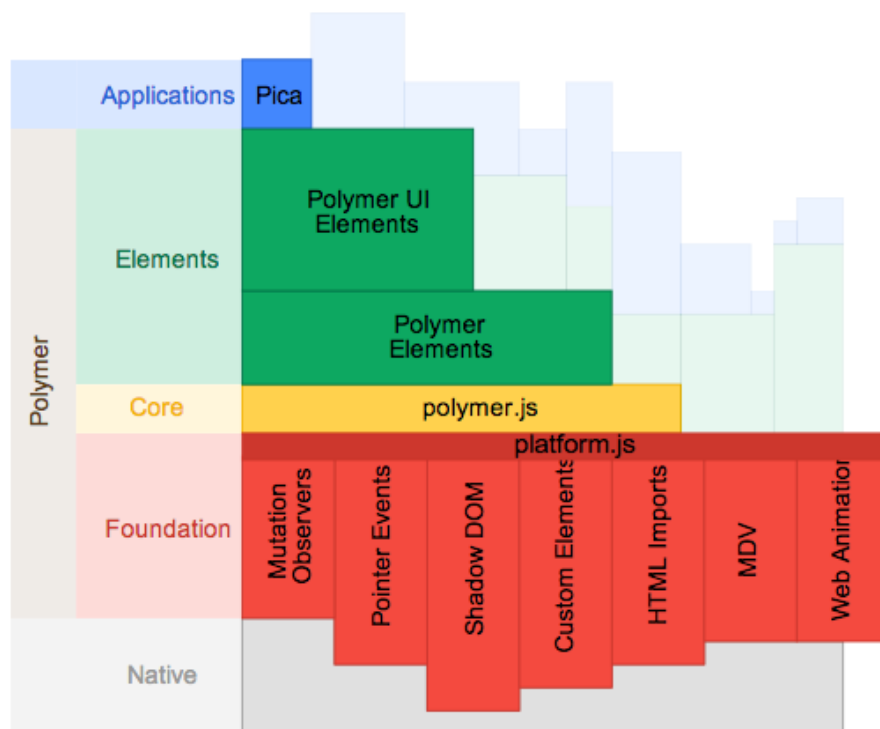


Figure 2.3: Polymer Architecture

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 possible 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 technologies 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 functionality. 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 de-

velopment 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.

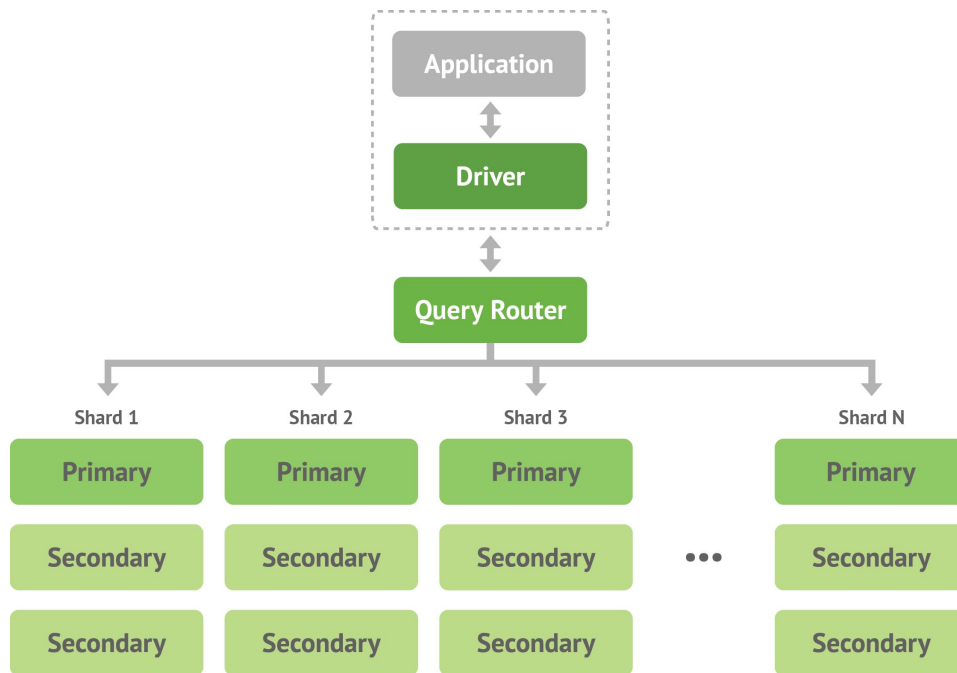


Figure 2.4: MongoDB Architecture

## 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<sup>1</sup>

<sup>1</sup>Mobile Backend as a service (MBaaS), also known as “backend as a service” (BaaS), is a model for providing web and mobile app developers with a way to link their applications to backend cloud storage and APIs exposed by back end applications while



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.

---

also providing features such as user management, push notifications, and integration with social networking services.

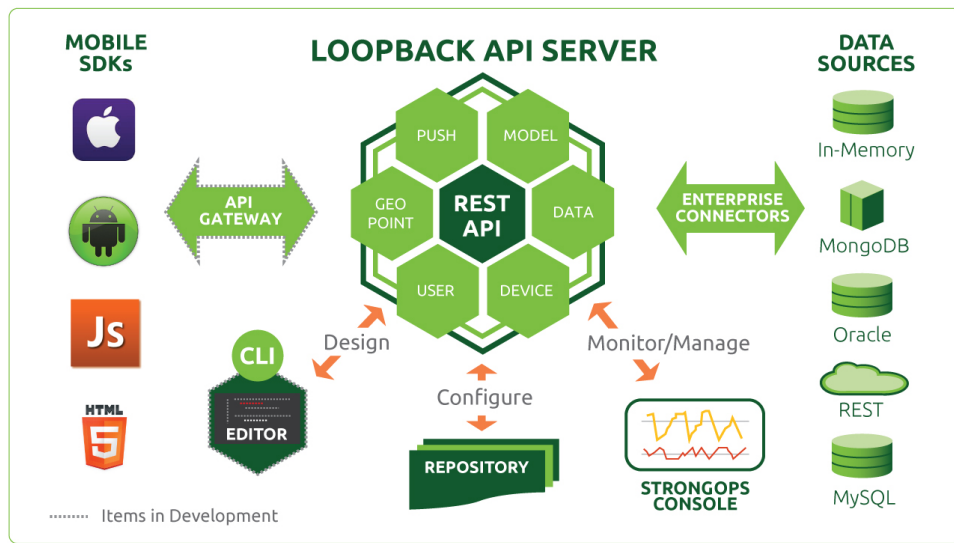


Figure 2.5: LoopBack Architecture

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 request and/or post-process the response). The resulting API is RESTful, cookie free, signed by authentication token. By default, applications have a built-in model that represents a user, with properties username, email and password and role for authentication 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 relation to its function and its use in the project.

## Chapter 3

# Single Page Application

This chapter provides an overview of Single Page Application development pattern. The first section analyzes the pattern in its totality and provides general and specific definitions. The second section presents pros and cons of the use of Single Page Application, such as SEO problems and speed of load. The third section explains technical functioning of the pattern.

### 3.1 Single Page Application

A 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 happens 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.

### 3.2 Single Page Application: pros and cons

One benefit of SPAs is obvious: Applications are more fluid and responsive, without the jarring effect of reloading and re-rendering the page. Another benefit is provided by ending the app data as JSON creates a separation between the presentation (HTML markup) and application logic (AJAX requests plus JSON responses). This separation makes it easier to design and evolve each layer. In a well-architected SPA, it is possible to change the HTML markup without touching the code that implements the application logic. In a pure SPA, all UI interaction occurs on the client side, through JavaScript and CSS. After the initial page load, the server acts purely as a service layer. The client just needs to know what HTTP requests to send. It doesn't care how the server implements things on the back end. With this architecture, the client and the service are independent. It is possible to replace the entire back end that runs the service, and as long as the API doesn't change, the client won't break. The reverse is also true.

Because the SPA is an evolution away from the stateless page-redraw model that browsers were originally designed for, some new challenges have

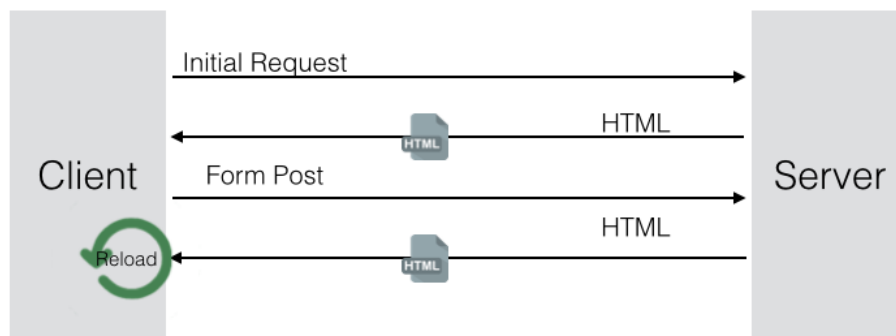


Figure 3.1: The Traditional Page Lifecycle

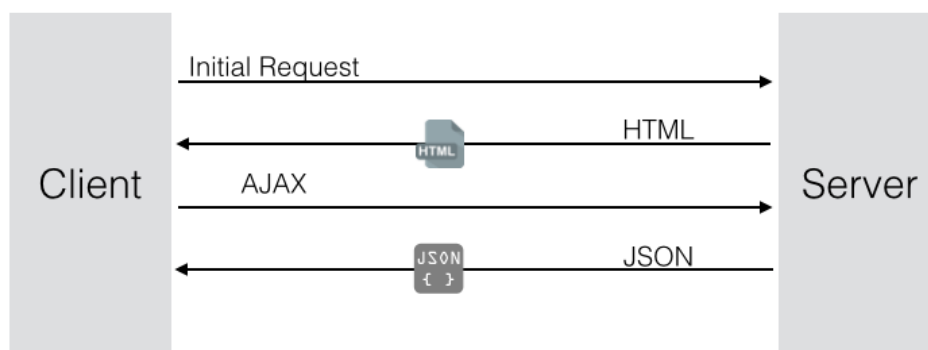


Figure 3.2: The SPA Lifecycle

emerged. Each of these problems has an effective solution with:

- Client-side JavaScript libraries addressing various issues.
- Server-side web frameworks that specialize in the SPA model.
- The evolution of browsers and the HTML5 specification aimed at the SPA model.

### 3.2.1 Search engine optimization (SEO)

Because of the lack of JavaScript execution on crawlers of some popular Web search engines, SEO (Search engine optimization) has historically presented a problem for public facing websites wishing to adopt the SPA

model. Google currently crawls URLs containing hash fragments starting with `#!`. This allows the use of hash fragments within the single URL of an SPA. Special behavior must be implemented by the SPA site to allow extraction of relevant metadata by the search engine's crawler. For search engines that do not support this URL hash scheme, the hashed URLs of the SPA remain invisible. Alternatively, applications may render the first page load on the server and subsequent page updates on the client. This is traditionally difficult, because the rendering code might need to be written in a different language or framework on the server and in the client. Using logic-less templates, cross-compiling from one language to another, or using the same language on the server and the client may help to increase the amount of code that can be shared. Because SEO compatibility is not trivial in SPAs, it's worth noting that SPAs are commonly not used in a context where search engine indexing is either a requirement, or desirable. Use cases include applications that surface private data hidden behind an authentication system. In the cases where these applications are consumer products, often a classic page redraw model is used for the applications landing page and marketing site, which provides enough meta data for the application to appear as a hit in a search engine query. Blogs, support forums, and other traditional page redraw artifacts often sit around the SPA that can seed search engines with relevant terms. Another approach used by server-centric web frameworks like the Java-based ItsNat is to render any hypertext in the server using the same language and templating technology. In this approach, the server knows with precision the DOM state in the client, any big or small page update required is generated in the server, and transported by AJAX, the exact JavaScript code to bring the client page to the new state executing DOM methods. Developers can decide which page states must be crawlable by web spiders for SEO and be able to generate the required state in load time generating plain HTML instead of JavaScript. In case of the ItsNat



framework, this is automatic because ItsNat keeps the client DOM tree in the server as a Java W3C DOM tree; rendering of this DOM tree in the server generates plain HTML in load time and JavaScript DOM actions for AJAX requests. This duality is very important for SEO because developers can build with the same Java code and pure HTML-based templating the desired DOM state in server; on page load time, conventional HTML is generated by ItsNat making this DOM state SEO-compatible. As of version 1.3, ItsNat provides a new stateless mode, client DOM is not kept in the server because, in stateless mode client, DOM state is partially or fully reconstructed in the server when processing any AJAX request based on required data sent by client informing of the current DOM state; the stateless mode may be also SEO-compatible because SEO compatibility happens in load time of the initial page not affected by stateful or stateless modes. There are a couple of workarounds to make it look as though the web site is crawlable. Both involve creating separate HTML pages that mirror the content of the SPA. Server could create a HTML-based version of the site and deliver that to crawlers, or it's possible to use a headless browser such as PhantomJS to run JavaScript application and output the resulting HTML. Both of these do require quite a bit of effort, and can end up giving a maintenance headache for the large complex sites. There are also potential SEO pitfalls. If server-generated HTML is deemed to be too different from the SPA content, then the site will be penalized. Running PhantomJS to output the HTML can slow down the response speed of the pages, which is something for which search engines – Google in particular – downgrades the rankings.

### 3.2.2 Client/Server code partitioning

One way to increase the amount of code that can be shared between servers and clients is to use a logic-less template language like Mustache or Handlebars. Such templates can be rendered from different host languages,

such as Ruby on the server and JavaScript in the client. However, merely sharing templates typically requires duplication of business logic used to choose the correct templates and populate them with data. Rendering from templates may have negative performance effects when only updating a small portion of the page—such as the value of a text input within a large template. Replacing an entire template might also disturb a user’s selection or cursor position, where updating only the changed value might not. To avoid these problems, applications can use UI data bindings or granular DOM manipulation to only update the appropriate parts of the page instead of re-rendering entire templates [34].

### 3.2.3 Analytics

Analytics tools such as Google Analytics rely heavily upon entire new pages loading in the browser, initiated by a URL change. SPAs don’t work this way. After the first page load, all subsequent page and content changes are handled internally by the application. So the browser never triggers a new page load, nothing gets added to the browser history, and the analytics package has no idea who’s doing what on the site.

It’s possible to add page load events to an SPA using the HTML5 history API; this will help integrate analytics. The difficulty comes in managing this and ensuring that everything is being tracked accurately – this involves checking for missing reports and double entries. Some frameworks provide open source analytics integrations addressing most of the major analytics providers. Developer should integrate them into the application and make sure that everything is working correctly, but there’s no need to do everything from scratch.

### 3.2.4 Speed of initial load

Single Page Applications have a slower first page load than server-based applications. This is because the first load has to bring down the framework and the application code before rendering the required view as HTML in the browser. A server-based application just has to push out the required HTML to the browser, reducing the latency and download time.

There are some ways of speeding up the initial load of an SPA, such as a heavy approach to caching and lazy-loading modules when needed. But it's not possible to get away from the fact that it needs to download the framework, at least some of the application code, and will most likely hit an API for data before displaying something in the browser. This is a “pay me now, or pay me later” trade-off scenario. The question of performance and wait-times remains a decision that the developer must make [34].

## 3.3 Single Page Application: how it works

An approach to implementing the single page pattern on today's web that makes it easy to get it right. It's based on three principles.

- Every view must have a real URL. On a normal web page, at any time I can grab what's in my browser's URL bar and either share it or bookmark it.
- Every link must be a real link. On a normal web page, I can view a link's destination by hovering it with my mouse; right-clicking produces a menu of link-specific options. So many otherwise-good pages try to implement their own link-like behavior by catching clicks on specific DOM nodes but then screw up corner case behaviors like middle-click.
- It's ok to be less awesome on old browsers. (Of course, whether this is actually true depends on site's goals.) One tactic for old browsers

is just to fail gracefully: if the site still works but is just slower, that's ok. For example, Gmail used to implement each of its buttons with a soup of DOM nodes to get the gradients and rounded corners to show on IE6; on today's web, maybe it's ok to just use some newer CSS for those effects and allow the buttons to be square and flat on IE6.

Because of principle 1, the app must be capable of rendering any given view from scratch, because the URL load might be a freshly started browser that just loaded a bookmark. Because of principle 2, links must be implemented as plain old `<a>` tags with an href that points at the URL of the resulting view. [24]

### 3.3.1 Local Routing

This section presents the operation and the importance of the local router to manage page requests made by the client. When the standard pattern is used HTTP GET request is triggered to require a new page: the browser itself sends the page request to the server, which responds by sending the page or performing the desired action. With the pattern SPA, the workflow is different: the use of a client side router prevents the browser's default behavior via the `preventDefault` method. The implementation of this pattern involves the following steps: listening to the path change via `onChange` function; for each event of this type, the request is handled by a mapping path or a handler, in which each path is associated with a function. In general, these functions have the task of retrieve the template and the content needed to fill it.

### 3.3.2 Server architecture

#### Thin server architecture

A SPA moves logic from the server to the client. This results in the role of the web server evolving into a pure data API or web service. This archi-

tectural shift has, in some circles, been coined “Thin Server Architecture” to highlight that complexity has been moved from the server to the client, with the argument that this ultimately reduces overall complexity of the system.

#### **Thick stateful server architecture**

The server keeps the necessary state in memory of the client state of the page. In this way, when any request hits the server (usually user actions), the server sends the appropriate HTML and/or JavaScript with the concrete changes to bring the client to the new desired state (usually adding/deleting/updating a part of the client DOM). At the same time, the state in server is updated. Most of the logic is executed on the server, and HTML is usually also rendered on the server. In some ways, the server simulates a web browser, receiving events and performing delta changes in server state which are automatically propagated to client.

This approach needs more server memory and server processing, but the advantage is a simplified development model because a) the application is usually fully coded in the server, and b) data and UI state in the server are shared in the same memory space with no need for custom client/server communication bridges.

#### **Thick stateless server architecture**

This is a variant of the stateful server approach. The client page sends data representing its current state to the server, usually through AJAX requests. Using this data, the server is able to reconstruct the client state of the part of the page which needs to be modified and can generate the necessary data or code (for instance, as JSON or JavaScript), which is returned to the client to bring it to a new state, usually modifying the page DOM tree according to the client action which motivated the request.

This approach requires that more data be sent to the server and may require more computational resources per request to partially or fully reconstruct the client page state in the server. At the same time, this approach

is more easily scalable because there is no per-client page data kept in the server and, therefore, AJAX requests can be dispatched to different server nodes with no need for session data sharing or server affinity. [34]

This chapter has provided an overview of Single Page Application development pattern. The first section has analyzed the pattern in its totality and has provided general and specific definitions. The second section has been presented pros and cons of the use of Single Page Application, such as SEO problems and speed of load. The third section has explained the technical functioning of the pattern.

Part II

Part 2

## Chapter 4

# X-Project

This chapter presents the core of the thesis project: X-Project.

The first section provides a project's overview, giving reasons of development, listing benefits and functions. The second section briefly explains of the reasons behind the name. The third section presents a practical example of some of X-Project functions. The fourth section shows the X-Project architectural stack and the reasons why these technologies have been chosen. The fifth section presents the development methodology that has been thought for the project. The sixth section presents a set of elements: most important practical part of the project.

### 4.1 X-Project overview

X-Project is a platform to build full-stack Javascript NodeJS API-centric HTML5 based Single Page Application with Web Components via Polymer-Project. X-Project is composed of guidelines, methodology and a library of elements. The document driven web development methodology and guidelines allow to build a very structured and usable Single Page Application.

“Everything is an element, even a service” is the philosophy of the project.

The joint use of Web Components and Strongloop Loopback framework



following the document driven web development methodology allowed to create vertical widget that influence every level of the stack. With a descriptive implementation it is possible to give life to API, on the server side, and visual and functional widget on the client side.

A Web Application is essentially built by composing elements together.

The goal of X-Project is to allow to build a Web Application by composing existing elements. This assumption comes from web application's sharing of essential non-specific components. Moreover, X-Project's goal is to achieve the same easiness of adding an html element, adding logic and function elements. In fact, in X-Project, an element is a part of the application including comprises both the client side and the server side.

X-Project was born to exploit Web Components benefits and extend them to extreme levels. X-Project, borrows all Web Components and Polymer-Project benefits and adds to them the ones that come from document driven development methodology. The Huge benefits coming from web components environment are reusability and access to reusable code. In fact, by creating stand alone vertical widgets, it's possible to use components in various forms. The encapsulation, typical of web components, allows to reuse elements with no concerns of dependencies and specific behaviors. Benefits of reuse are enhanced by the creation of a set of elements (see 4.6) that provide a wide range of chances of composition. This library is composed by a large number of elements of various type: local routing, API, forms, lists, style and page elements. X-Project, according to its philosophy, "elementize" everything: local routing elements handle the SPA client-side router (See 3.2); API elements encapsulate AJAX requests and handle server and database operations; forms, lists and page elements have been created to compose parts and functions of pages; style elements encapsulate a group of CSS rules that is applicable to any type of element.

Moreover, thanks to Web Components philosophy and X-Project guide-

lines, 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. This aspect also makes style extremely reusable. In X-Project guidelines this pattern is applied to every kind of element.

Moreover, in the world of web application building platform, there are limits and gaps in terms of ease of use and orientation. As said in 1.2, it is possible to make a classification of these platforms based on orientation patterns. Nowadays, user oriented platforms have lacks when projects assume big dimensions mainly because of their lack of structure. Developer oriented platforms don't have the same limits, but are defective in ease of use.

## 4.2 Why the name?

The name X-Project has been chosen for three different reasons. First of all, X character, in maths, can represent any value. Just as this project's soul, the X is variable. X-Project can be used to make any kind of web application, from blogs to e-commerce websites. So, this versatility of mathematical X, fits this project soul. Secondly, all experimental elements' names created by Google during the dawn of Polymer-Project were preceded by a "X-" prefix. Finally, the X-Project is a sort of a tribute to X-Tag: the Mozilla's web components library [20].

## 4.3 User Management example: login

The example is based on the user login service. For this function both the client side and the backend has been implemented. So, for the client side the element `<form-login>` has been developed , for the server side the element `<api-user-login>` has been developed.

The elements that have been developed are:

### **<api-user-login>**

Login a user with the given credentials.

```
<api-user-login credentials="{{credentials}}"  
collection="{{collection}}"  
response="{{response}}" error="{{error}}"/>
```

Where:

- credentials email and password of user.
- collection name of collection(Object).
- response HTTP response message(String).
- error object of the error response(Object).

### **<form-login>**

Create a login form.

```
<form id="form" on-submit="on_submit">  
  <div class="field">  
    <label class="label">email</label>  
    <input class="input" is="iron-input" type="text"  
      placeholder="email"  
      bind-value="{{credentials.email}}">  
  </div>  
  <div class="field">  
    <label class="label">password</label>  
    <input class="input" is="iron-input" type="password"  
      placeholder="password"  
      bind-value="{{credentials.password}}">  
  </div>  
  <input type="submit" value="login"/>  
</form>
```

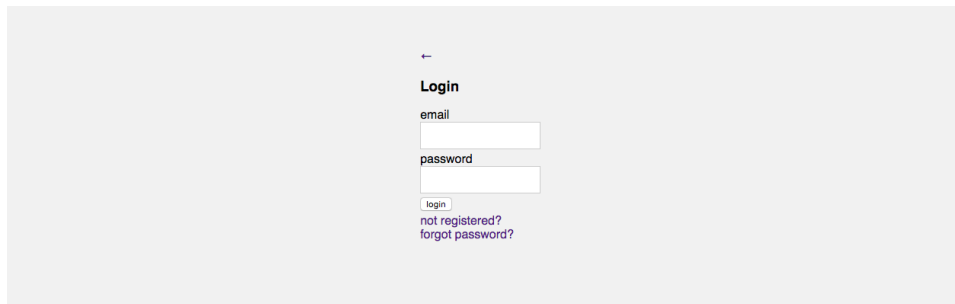


Figure 4.1: Login Element Example

## 4.4 X-Project Architecture

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

### 4.4.1 Server side

On the server side, an X-Project 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 vertical 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.

#### 4.4.2 Client side

On the client side, an X-Project app is based on HTML5 Web Components via Polymer-Project by Google. X-Elements lies on the top of this stack. It is a set of Polymer elements for local routing, API request, forms, lists, style and admin panels (as listed further 4.6).

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 independent 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.

### 4.5 Document-Driven Web Development Process

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

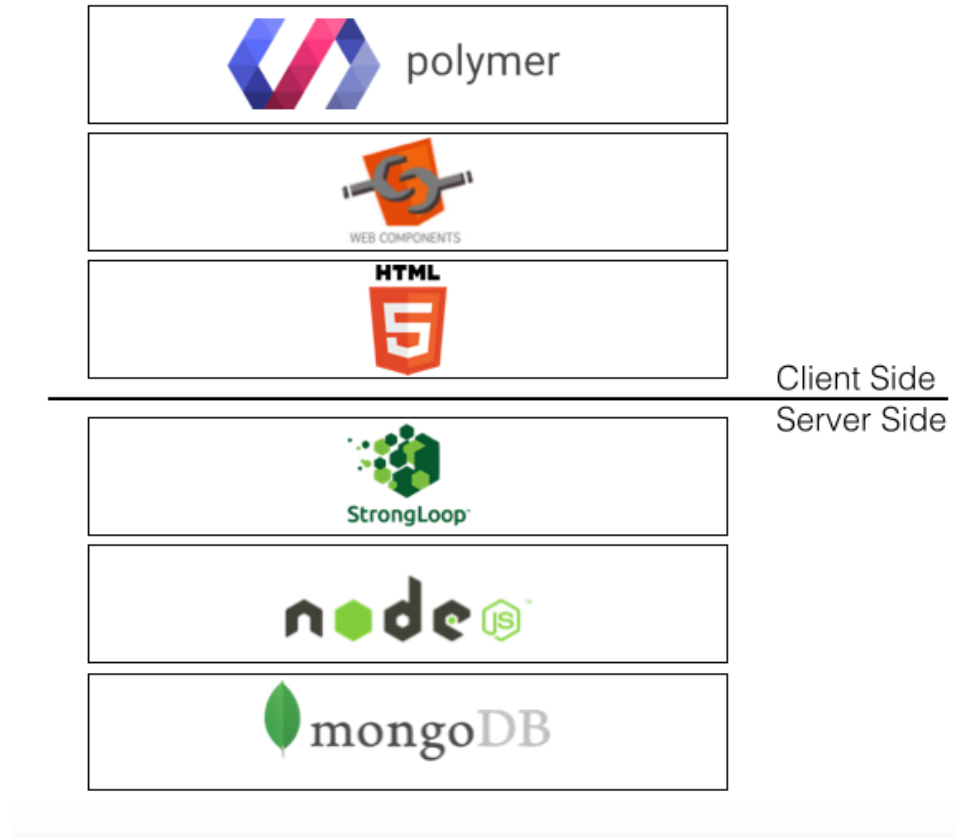


Figure 4.2: X-Project Architectural Stack

#### 4.5.1 1st step - Models schemas definition

A description of entities, properties, relations and data access policies are defined as JSON documents.

#### 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. All of them are exposed as HTTP RESTful API.

### 4.5.3 3rd step - UI components definition

Distinct UI components can be defined, or retrieved from a collection of predefined components, configured and adapted. They represent the building blocks of the whole UI.

### 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.

## 4.6 X-Elements

All the following elements are retrievable on a Github repository [37].

“Everything is an element”, from an AJAX request to an entire web page. Every part of the website is encapsulated inside an element.

X-project provides a set of Polymer elements for local routing, API requests, forms, lists, style and admin panel, as listed below.

Elements can be customized through their attributes. Attributes can act as inputs parameters (values having effects on the element) or output parameters (values that are returned by the element). Values in parameters could be hard-coded (if they never change) or stored in variables.

Different parameters in different elements could use the same variable, so, the value of an output parameter of an element could be used as input in an input parameter of another element.

### 4.6.1 Elements for local routing

The following elements perform local routing (for Single Page Application).

**<x-router>** Implements local routing using HTML5 Push State API. It represents the core element of the app. It intercepts routes, creates pages, and passes parameters to the page.

**<x-route>** Represents a route-to-page mapping. Parameters presented in an URL are sent as attributes to the corresponding page.

```
<x-route route="{{route}}" page="{{page}}"/>
```

**<x-link>** Is an extension of the anchor element `<a>` that prevents the default behavior when a click event occurs, blocking page request to the server and redirecting the request to the local router.

```
<a is="x-link" href="{{href}}">{{link}}/>
```

### 4.6.2 Elements for API

The following elements are used to create API.

**<api-model-create>** Create new instance of Model, and save to database.

```
<api-model-create model="{{model}}" data="{{data}}"
collection="{{collection}}" response="{{response}}"
error="{{error}}"/>
```

Where:

- model name of model (String)
- data user data (Object)
- collection name of collection (Object)



- response HTTP response message (String)
- error object of the error response (Object)

**<api-model-get>** Get the id property name of the constructor.

```
<api-model-get model_id="{{model_id}}"
collection="{{collection}}"
response="{{response}}" error="{{error}}"/>
```

Where:

- model-id id of model (String)
- collection name of collection (Object)
- response HTTP response message (String)
- error object of the error response (Object)

**<api-model-update>** Update multiple instances that match the where clause.

```
<api-model-update model_id="{{model_id}}"
"collection="{{collection}}"
response="{{response}}" error="{{error}}"/>
```

Where:

- model-id id of model(String).

**<api-model-find>** Find all model instances that match filter specification.

```
<api-model-find where="{{where}}" "collection="{{collection}}"
response="{{response}}" error="{{error}}"/>
```

Where:

- where where clause (Object)

**<api-model-delete>** Deletes the model from persistence.

```
<api-model-delete model_id="{{model_id}}"
collection="{{collection}}"
response="{{response}}" error="{{error}}"/>
```

Where:

- model-id id of model (String)

**<api-model-exists>** Check whether a model instance exists in database.

```
<api-model-exists model_id="{{model_id}}" exists="{{boolean}}"
collection="{{collection}}" response="{{response}}"
error="{{error}}"/>
```

Where:

- model-id id of model (String)
- exists True if the instance with the specified ID exists; false otherwise (Output)

**<api-model-count>** Check whether a model instance exists in database.

```
<api-model-count count="{{count}}" collection="{{collection}}"
response="{{response}}" error="{{error}}"/>
```

Where:

- count number of instances updated (Output)

### 4.6.3 Elements for forms

The following elements are used to create forms.

**<x-input>** Is an extension of the input element.

```
<x-input type="{{type}}" label="{{label}}" value="{{value}}"/>
```

Where:

- type can be string, number, date, email, url, location (with auto-completion based on Google Place API) and file.

**<x-form>** Dynamically generates a form from a model schema, to create/update a model.

```
<x-form schema="{schema}" model="{model}"/>
```

#### 4.6.4 Elements for lists

The following elements are used to manage lists.

**<x-table>** Dynamically generates a table of models from a model schema.

```
<x-table schema="{{schema}}" items="{{items}}"/>
```

Where:

- `schema` is used to generate the columns of the table.
- `items` is used to generate the rows (the values) of the table.

**<x-pager>** Generates the list of links to handle pagination.

```
<x-pager perpage="{{perpage}}" count="{{count}}"
current="{{page}}"/>
```

Where:

- `count` is the total number of items to paginate.
- `perpage` is the number of items per page.
- `current` is the current page selected by the user.

By itself, pagination doesn't paginate any list, but it can be used in conjunction with `<api-collection-get>` (as shown in the case study), where the current output parameter of `<x-pager>` is the input page parameter of `<api-collection-get>`.

#### 4.6.5 Elements for style

The style is based on `iron-flex-layout`, a CSS library of style mixins for cross-platform Flexible Box layouts.

### 4.6.6 Elements for admin panel

Even a page can be encapsulated in an element. x-project provides a set of pages for the admin part of the app, `<page-collection>` and `<page-model-edit>`, presented below.

### 4.6.7 Elements for pages

**<x-header>** This element is used to insert an header at the top of the page.

```
<x-header links="{{links}}" brand="{{brand}}"/>
```

Where:

- `links` is the total number of items to paginate.
- `brand` is the number of items per page.

**<x-footer>** This element is used to insert footer at the bottom of the page.

```
<x-footer links="{{links}}" notes="{{notes}}"/>
```

Where:

- `links` is the total number of items to paginate.
- `notes` is the number of items per page.

**<x-crew>** This element is used to insert the section for the presentation of the team of a project.

```
<x-crew team="{{team}}"/>
```

Where:

- `team` is the total number of items to paginate.

**<x-contact>** This element is used to insert the section for the presentation of the contact.

```
<x-contact contact="{{contact}}" />
```

Where:

- `links` is the total number of items to paginate.
- `brand` is the number of items per page.

In this chapter the main part of thesis project has been described: X-Project. First of all, it an overview of the project has been provided, later, the name explanation it has been introduced. In the last sections, X-Project architecture functionality have been introduced.

## Chapter 5

# Media Management: S3 Component

This chapter analyzes the component that has been implemented to manage media in X-Project via Amazon AWS storage service named Amazon S3.

The first two sections present informations about Amazon AWS and Amazon S3 storage service. The third section introduces the Cross-Origin Resource Sharing (CORS) development pattern. The fourth section presents the security aspect of the component introducing Digital Signature (or Asynchronous Encryption) algorithm. In the fifth section the component is technically analyzed and, in section six, an example is provided.

### 5.1 Amazon AWS

Amazon Web Services (AWS), a collection of remote computing services, also called web services, make up a cloud-computing platform offered by Amazon.com.[2] These services operate from 11 geographical regions across the world. The most central and well-known of these services arguably include Amazon Elastic Compute Cloud and Amazon S3. Amazon markets these products as a service to provide large computing-capacity more quickly

and more cheaply than a client company building an actual physical server farm [4].

AWS is located in 11 geographical “regions”: US East (Northern Virginia), where the majority of AWS servers are based,[19] US West (northern California), US West (Oregon), Brazil (São Paulo), Europe (Ireland and Germany), Southeast Asia (Singapore), East Asia (Tokyo and Beijing) and Australia (Sydney). There is also a “GovCloud”, based in the Northwestern United States, provided for U.S. government customers, complementing existing government agencies already using the US East Region.[4] Each Region is wholly contained within a single country and all of its data and services stay within the designated Region.[citation needed]

Each Region has multiple “Availability Zones”, which are distinct data centers providing AWS services. Availability Zones are isolated from each other to prevent outages from spreading between Zones. Several services operate across Availability Zones (e.g., S3, DynamoDB) while others can be configured to replicate across Zones to spread demand and avoid downtime from failures. Amazon web services hold 1.79% market share. As of December 2014, Amazon Web Services operated an estimated 1.4 Million servers across 28 availability zones [18].

## 5.2 Amazon S3

Amazon S3 (Simple Storage Service) is an online file storage web service offered by Amazon Web Services. Amazon S3 provides storage through web services interfaces (REST, SOAP, and BitTorrent).Amazon launched S3, its first publicly available web service, in the United States in March 2006 and in Europe in November 2007 [25].

Amazon S3 is reported to store more than 2 trillion objects as of April 2013.[10] S3 uses include web hosting, image hosting, and storage for backup systems. Amazon S3 provides an API (Application programming interface)

for third-party developers. It describes various API operations, related request and response structures, and error codes.[38] Web services interface can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers. Today, there are different kinds of file managers for Amazon S3. An effective solution for Amazon provides a user interface to Amazon S3 accounts, files and buckets, allowing to browse, create and delete files and buckets.

### 5.3 CORS

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated.

A web page may freely embed images, stylesheets, scripts, iframes, videos and some plugin content (such as Adobe Flash) from any other domain. However embedded web fonts and AJAX (XMLHttpRequest) requests have traditionally been limited to accessing the same domain as the parent web page (as per the same-origin security policy).“Cross-domain” AJAX requests are forbidden by default because of their ability to perform advanced requests (POST, PUT, DELETE and other types of HTTP requests, along with specifying custom HTTP headers) that introduce many cross-site scripting security issues.

CORS defines a way in which a browser and server can interact to safely determine whether or not to allow the cross-origin request. It allows for more freedom and functionality than purely same-origin requests, but is more secure than simply allowing all cross-origin requests. It is a recommended standard of the W3C.



The CORS standard describes new HTTP headers which provide browsers and servers with a way to request remote URLs only when they have permission. Although some validation and authorization can be performed by the server, it is generally the browser’s responsibility to support these headers and respect the restrictions they impose.

For AJAX and HTTP request methods that can modify data (usually HTTP methods other than GET, or for POST usage with certain MIME types), the specification mandates that browsers “preflight” the request, soliciting supported methods from the server with an HTTP OPTIONS request header, and then, upon “approval” from the server, sending the actual request with the actual HTTP request method. Servers can also notify clients whether “credentials” (including Cookies and HTTP Authentication data) should be sent with requests [11].

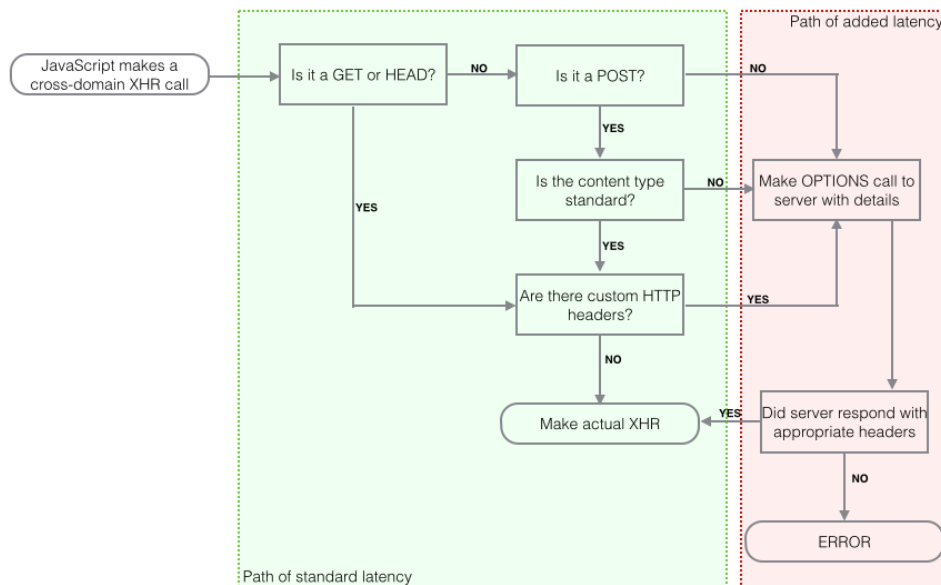


Figure 5.1: Flowchart showing Simple and Preflight XHR by Bluesmoon

## 5.4 Digital Signature

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document. The digital equivalent of a handwritten signature or stamped seal, but offering far more inherent security, a digital signature is intended to solve the problem of tampering and impersonation in digital communications. Digital signatures can provide the added assurances of evidence to origin, identity and status of an electronic document, transaction or message, as well as acknowledging informed consent by the signer.

### 5.4.1 How digital signatures work

Digital signatures are based on public key cryptography, also known as asymmetric cryptography. Using a public key algorithm such as RSA, one can generate two keys that are mathematically linked: one private and one public. To create a digital signature, signing software (such as an email program) creates a one-way hash of the electronic data to be signed. The private key is then used to encrypt the hash. The encrypted hash, along with other information, such as the hashing algorithm, is the digital signature. The reason for encrypting the hash instead of the entire message or document is that a hash function can convert an arbitrary input into a fixed length value, which is usually much shorter. This saves time since hashing is much faster than signing.

The value of the hash is unique to the hashed data. Any change in the data, even changing or deleting a single character, results in a different value. This attribute enables others to validate the integrity of the data by using the signer's public key to decrypt the hash. If the decrypted hash matches a second computed hash of the same data, it proves that the data hasn't changed since it was signed. If the two hashes don't match, the data has either been tampered with in some way (integrity) or the signature

was created with a private key that doesn't correspond to the public key presented by the signer (authentication). [16]

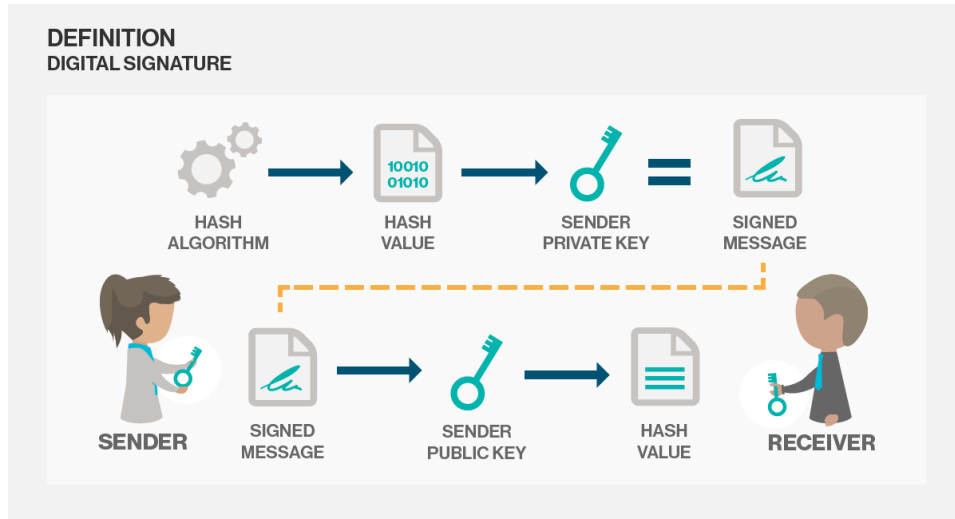


Figure 5.2: Digital Signature Process

## 5.5 Media Management S3 Component

This section analyzes the Media Management S3 Component, developed to store images and documents on cloud through Amazon S3 Service. This specific component has been developed in order to store and manage media files in Amazon S3 service. Operations that can be done, like upload, delete or get media, are ruled by CORS (5.3) pattern. These operations are direct: none of them is done via server. Server is only used to get the authorization to communicate with the S3 cloud storage service. In fact, client makes an AJAX request to the server in order to get the Signed URL, that is the request URL combined with the name of S3 Bucket and the dedicated Region. The Signed URL pattern represents the Digital Signature security mechanism, as described in 5.4.

The Signed URL is requested via an ad hoc API, that is exposed to client side through a remote method. Get, Upload and Delete functions have been

implemented on the server side by using an additional library named AWS SDK for Javascript NodeJS <sup>1</sup> [3].

Each function creates a request for a Signed URL to its server. Using file's name, bucket's name, bucket's region and user key, the server gives back an URL to which the client can address his real HTTP request.

Once the client receives the Signed URL it can pass the operation to S3 Service and carry it out. Bucket and Region names are stored in `.env` file and must be set up before on S3 website. Moreover, users must set Bucket's CORS Configuration on S3 Admin Panel: CRUD operations must be allowed by the user in the CORS Configuration Editor panel.

### Upload Signed Url Request

```
Image.signed_put = function(file_name, file_type, callback) {  
    var s3 = new aws.S3();  
    var s3_params = {  
        Bucket: S3_BUCKET,  
        Key: file_name,  
        Expires: 60,  
        ContentType: file_type,  
        ACL: 'public-read'  
    };  
    s3.getSignedUrl('putObject', s3_params,  
        function (err, signed_url) {  
            if (err) {  
                callback(err);  
                return;  
            }  
            callback(null, signed_url);  
        });  
};  
  
Image.remoteMethod('signed_put', {  
    http: { verb: 'get' },
```

---

<sup>1</sup> The AWS SDK helps take the complexity out of coding by providing JavaScript objects for AWS services including Amazon S3, Amazon EC2, DynamoDB, and Amazon SWF.

```
accepts: [
  {arg: 'file_name', type: 'string'},
  {arg: 'file_type', type: 'string'}
],
returns: {arg: 'signed_url', type: 'string'}
});
```

## Get Signed Url Request

```
Image.signed_list = function (folder, callback) {
  var s3 = new aws.S3();
  var s3_params = {
    Bucket: S3_BUCKET,
    EncodingType: 'url',
    Prefix: folder,
    MaxKeys: 1000
  };
  s3.getSignedUrl('listObjects', s3_params,
    function (err, signed_url) {
      if (err) {
        callback(err);
        return;
      }
      callback(null, signed_url);
    });
};

Image.remoteMethod('signed_list', {
  http: { verb: 'get' },
  accepts: { arg: 'folder', type: 'string' },
  returns: { arg: 'signed_url', type: 'string' }
});
```

## Delete Signed Url Request

```
Image.signed_delete = function(file_name, callback) {
  var s3 = new aws.S3();
  var s3_params = {
    Bucket: S3_BUCKET,
    Key: file_name
  };
  s3.getSignedUrl('deleteObject', s3_params,
```

```
function (err, signed_url) {  
  if (err) {  
    callback(err);  
    return;  
  }  
  callback(null, signed_url);  
});  
};  
  
Image.remoteMethod('signed_delete', {  
  http: { verb: 'get' },  
  accepts: {arg: 'file_name', type: 'string'},  
  returns: {arg: 'signed_url', type: 'string'}  
});
```

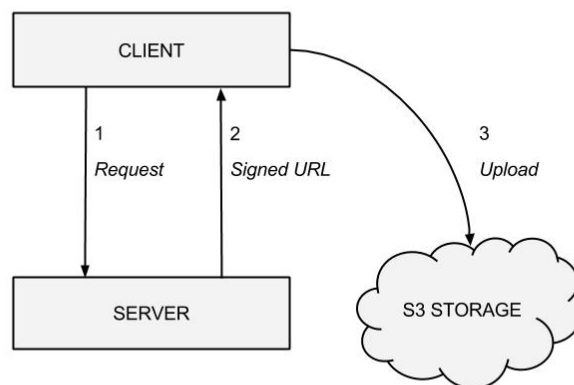


Figure 5.3: S3 direct upload process

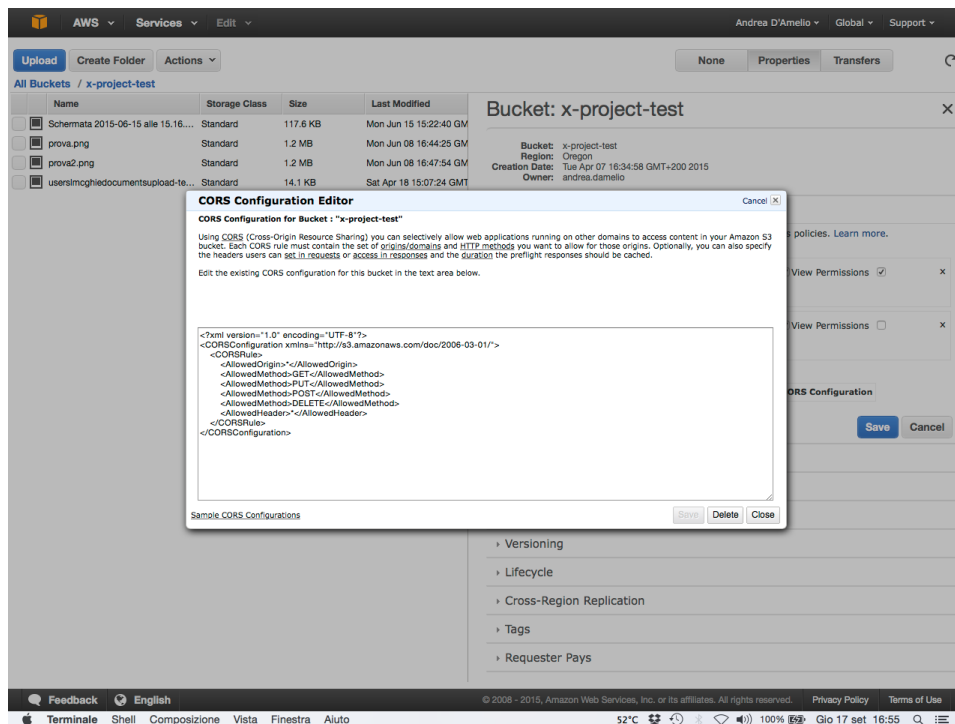


Figure 5.4: S3 CORS Configuration Editor Panel

### 5.5.1 S3 Component elements

In order to use Media Management S3 Component in an X-Project App there's need to combine six different X-Elements:

- api-s3-upload
- api-s3-list
- api-s3-delete
- part-s3-list
- part-s3-list-item
- part-s3-upload

Each element provides the opportunity to use one of the basic function of Amazon S3 service.

**api-s3-upload**

The element named `api-s3-upload` has been designed to upload new files to the S3 Bucket.

```
<api-s3-upload id="upload" folder="{{folder}}"  
  file="{{file}}" file-name="{{fileName}}">  
</api-s3-upload>
```

**api-s3-list**

The element named `api-s3-list` has been designed to get the list of elements currently in the S3 Bucket.

```
<api-s3-list id="list" list="{{list}}">  
</api-s3-list>
```

**api-s3-delete**

The element named `api-s3-delete` has been designed to delete an element currently in the S3 Bucket.

```
<api-s3-delete id="request" file-name="{{item.key}}">  
</api-s3-delete>
```

**part-s3-list**

The element named `part-s3-list` has been developed to show the list of elements currently in the S3 Bucket

```
<template>  
  <api-s3-list id="list" list="{{list}}">  
  </api-s3-list>  
  
  <template is="dom-repeat" items="{{list}}" filter="filter">  
    <part-s3-list-item item="{{item}}" on-delete="update">  
    </part-s3-list-item>  
  </template>  
</template>
```



The first tag, `<api-s3-list>`, calls the GET API that handles the request to S3, and returns the list of elements currently in S3 Bucket. The second tag, `<part-s3-list-item>`, creates a visual element for each object currently in the bucket. This cycle is expressed via the attribute `"dom-repeat"`.

### **part-s3-list-item**

The element named `part-s3-list-item` has been developed to show a preview of a selected files and to let the user delete it.

```
<api-s3-delete id="request" file-name="{{item.key}}">
</api-s3-delete>
...
<button id="delete" on-click="delete_image">delete
</button>
...
delete_image: function () {
    this.$.request.send();
}
...
```

The first tag, `<api-s3-delete>`, set the DELETE API that handles the request to S3. The button tag creates a button that, when clicked, trigger the `delete image` function, that makes the call via `<api-s3-delete>` element.

### **part-s3-upload**

The element named `part-s3-upload` has been developed to allow to users to upload new files to the S3 Bucket.

```
<api-s3-upload id="upload"
    file="{{file}}" file-name="{{fileName}}">
</api-s3-upload>

<input id="input" type="file" on-change="on_change">
on_change: function () {
    var file = this.$.input.files[0];
```

```

    if (!file) {
        return;
    }

    this.fileName = this.fileName || file.name;
    this.file = file;
}

```

The first tag, `<api-s3-upload>`, sets the upload API that handles the request to S3. The input tag creates an input form that lets the user choose the file to upload and trigger the `on_change` function, that sends the file via `<api-s3-upload>` element.

## 5.6 S3 Component - Example

The first example concerns the upload actions of the component. As shown before, `part-s3-upload` element calls an `api-s3-upload` element to handle the transfer from a local computer to S3 cloud service. In the HTML code of `part-s3-upload` element both the informations and input tags and the binding with the API are implemented.

### 5.6.1 Example 1 - `part-s3-upload`

```

<link rel="import"
href="/components/api-s3-upload/api-s3-upload.html">

<dom-module id="part-s3-upload">
  <template>
    <api-s3-upload id="upload"
      file="{{file}}" file-name="{{fileName}}">
    </api-s3-upload>
    <input id="input" type="file" on-change="on_change">
  </template>
</dom-module>

<script>
  Polymer({
    on_change: function () {

```

```

    var file = this.$.input.files[0];
    if (!file) {
        return;
    }
    this.fileName = this.fileName || file.name;
    this.file = file;
    this.$.upload.send();
}
});
</script>

```

In this case, the `api-s3-upload` is triggered by the `on-change` function applied on the input box. As shown above, `on-change` function checks the integrity of the file and saves its name, then, using `$` selector that puts off to the tag with `id="upload"`, triggers the API.

### 5.6.2 Example 2 - part-list

Second example concerns the retrieval of data to be shown. As shown before, `part-s3-list` element calls an `api-s3-list` element to handle the GET request and the retrieval of data. In the HTML code of `part-s3-list` element there is a binding to `part-s3-list-item` element that manages the visualization of the retrieved informations.

```

<link rel="import"
    href="/components/api-s3-list/api-s3-list.html">
<link rel="import"
    href="part-s3-list-item.html">

<dom-module id="part-s3-list">
    <template>
        <api-s3-list id="list" list="{{list}}">
        </api-s3-list>
        <template is="dom-repeat" items="{{list}}"
            filter="filter">
            <part-s3-list-item item="{{item}}" on-delete="update">
            </part-s3-list-item>
        </template>
    </template>
</dom-module>

```

```

</dom-module>

<script>
  Polymer({
    update: function () {
      this.$.list.send();
    },
    filter: function (item) {
      var url =
        'https://' + this.bucket + '.s3.amazonaws.com/' + item.key;
      item.url = url;
      return item;
    }
  });
</script>

```

In this case, the `api-s3-list` is triggered at page loading time. Moreover, in `filter` function, that is called for each object, the url is saved for further communications.

### 5.6.3 Example 3 - `part-list-item`

The third example concerns the retrieval and visualization actions of the component. As shown before, `part-list-item` is triggered by `part-list` element. `part-list-item`, in turn, triggers `api-s3-delete` API. This element shows the image saved in S3 Bucket and its additional informations. Moreover `part-list-item` provides a button to delete the image from the bucket.

```

<link rel="import"
href="/components/api-s3-delete/api-s3-delete.html">
<dom-module id="part-s3-list-item">
  <template>

    <api-s3-delete id="request" file_name="{{item.key}}">
    </api-s3-delete>

```

```

<div id="image">
  
  <template is="dom-if" if="{{preview}}">
    <ul id="image_labels">
      <li>Name:
        <span id="image_details">{{item.key}}</span>
      </li>
      <li>Modified:
        <span id="image_details">{{item.lastMod}}</span>
      </li>
      <li>Size:
        <span id="image_details">{{item.size}}</span>
      </li>
    </ul>
    <button id="delete" on-click="on_click">
      Delete
    </button>
  </template>
</div>

</template>
</dom-module>
<script>
  Polymer({
    delete_image: function () {
      this.$.request.send();
    }
  });
</script>

```

In this case, the `api-s3-delete` is triggered by the `on-click` function applied on the delete button. As shown above, `on-click` function, using `$` selector that puts off to the tag with `id="request"`, triggers the API.

This chapter has analyzed the component that has been implemented to manage media in X-Project via Amazon AWS storage service named Amazon S3.

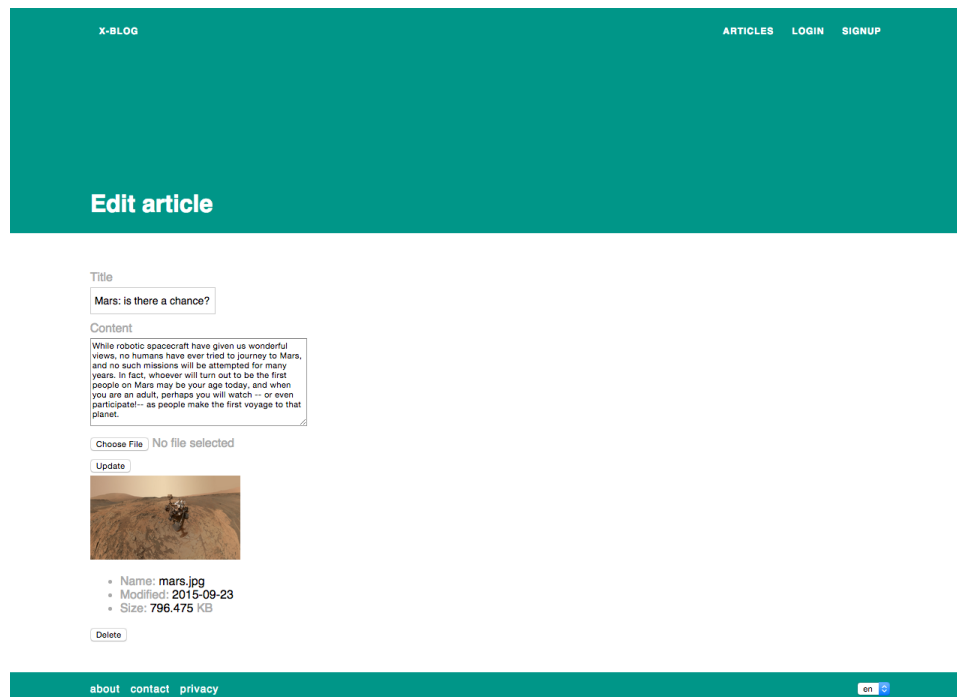


Figure 5.5: S3 Component example - Upload and preview functions

In Chapter Five overviews of Amazon AWS and Amazon S3 storage service have been provided. The third section has introduced the CORS development pattern. The fourth section has presented the security aspect of the component: Digital Signature (or Asynchronous Encryption) algorithm. In the fifth section the component has been technically analyzed and, in section six, an example has been provided.

## Chapter 6

# User Management: Login toolkit

This chapter analyzes the component that has been implemented to manage users with the support of Mandrill service. In the first section some services for user management such as Stormpath, Auth0 and UserApp have been introduced. In the second section the user management elements developed within X-Project have been analyzed. Each element has been presented with a description and a snippet such as `<api-user-login>` and `<api-user-signup>`; the first is a login element and the second is a signup element. In the last section the Mandrill service used to implement the user elements has been presented.

### 6.1 User management

The application list addressed to a community of users / customers have to implement the management of user.

The mechanisms for base user management are:

- user login

- user signup
- user logout
- user management

Further advanced management mechanisms may be associated, such as:

- email verification
- email changing
- password recovery/reset
- password changing

The mechanisms of advanced management, involving the use of email, must necessarily be based on an email management service. Challenge of this project is to be able to encapsulate these mechanisms / behaviors in their elements, to allow it to integrate user management as easily as it is possible to insert HTML elements on a page.

### 6.1.1 User Management services

This section talks about the various existing services for user management as: Stormpath, Userapp and Auth0.

#### Stormpath

Stormpath is a User Management API that reduces development time with instant-on, scalable user infrastructure. Stormpath's intuitive API and expert support make it easy for developers to authenticate, manage and secure users and roles in any application.

Stormpath, has a simple goal: to provide developers with a complete user management system, so that they can focus on building great applications.[26]



- Pre-built authentication & authorization.
- Schemaless, secure user data & profiles.
- Code-free Active Directory, Facebook & Google login.
- Open Source SDKs & complete sample apps.

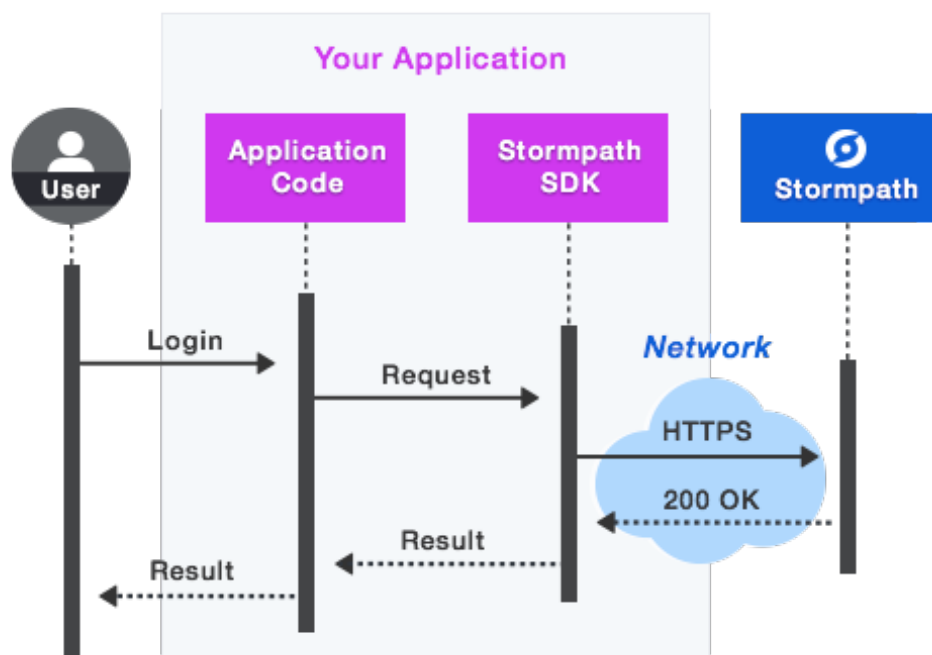


Figure 6.1: Stormpath User Management API

## Userapp

UserApp is a cloud-based user management API for web apps. The purpose is to relieve developers from having to program logic for login, sign up, calculate payments, turn on or off features, etc. And instead focus on their core product.

UserApp provides user management functionality that results in faster development, faster revenue, more users, and the ability to serve users better

by engaging with them more efficiently.[29]

- User authentication - Have the user authentication ready today. It is possible just a few lines of code away with one of SDKs.
- One-click integrations - Integrate the users with third-party services with just one click.
- Mobile, web, and server - No matter where is need to integrate the user authentication, it got covered.

## Auth0

Auth0 is an enterprise-grade platform for modern identity. Auth0 gives the tools that eliminate the friction of authentication for the applications and APIs - all accessible through the account dashboard.[5]



Figure 6.2: Auth0 Overview

It is possible to connect any application, written on any language or stack to Auth0, and separately to define how users of that application authenticate:

- Custom credentials: username/passwords.
- Social network logins: Google, Facebook, Twitter and any OAuth2 or OAuth1 provider.
- Enterprise directories: LDAP, Google Apps, Office 365, ADFS, AD, SAML-P, WS-Federation, etc.,

- Password-less systems: TouchID, one time codes on SMS.

### 6.1.2 User Management standard API in LoopBack

X-project is based on StrongLoop LoopBack on the server side. LoopBack's built-in User model provides essential user management features such as:

- Registration and confirmation via email.
- Login and logout.
- Creating an access token.
- Password reset.

In general it is possible extend the User model to suit specific needs, so in most cases, it is don't need to create the own User model from scratch. By default, a LoopBack application has a built-in User model defined by `user.json` In particular, it was implemented the connection with MailChimp service, named Mandrill.

### 6.1.3 User Management remote methods

In addition to the standard APIs, remote methods for email and password changing have been implemented.

#### Change Email

```
Author.change_email = function (new_email, confirm_email, password, cb) {  
  if (new_email !== confirm_email) {  
    cb({ error: 'email not confirmed' }, null);  
    return;  
  }  
  
  var userId = getCurrentUserId();
```

```
Author.findById(userId, function (err, user) {
  if (err) {
    cb(err, null);
    return;
  }

  user.hasPassword(password, function (err, match) {
    if (!match) {
      cb({ error: 'invalid password' }, null);
      return;
    }

    user.updateAttribute('email', new_email, function (err, user) {
      if (err) {
        cb(err, null);
        return;
      }

      cb(null, true);
    });
  });
});

Author.remoteMethod('change_email', {
  http: { path: '/change_email', verb: 'post' },
  accepts: [
    { arg: 'new_email', type: 'string' },
    { arg: 'confirm_email', type: 'string' },
    { arg: 'password', type: 'string' }
  ],
  returns: { arg: 'changed', type: 'boolean' }
});
```

## Change Password

```
Author.change_password = function (new_password, confirm_password,
  password, cb) {
  if (new_password !== confirm_password) {
    cb({ error: 'password not confirmed' }, null);
    return;
  }
}
```

```

var userId = getCurrentUserId();

Author.findById(userId, function (err, user) {
  if (err) {
    cb(err, null);
    return;
  }

  user.hasPassword(password, function (err, match) {
    if (!match) {
      cb({ error: 'invalid password' }, null);
      return;
    }

    user.updateAttribute('password', new_password, function (err, user
      ) {
      if (err) {
        cb(err, null);
        return;
      }

      cb(null, true);
    });
  });
});

Author.remoteMethod('change_password', {
  http: { path: '/change_password', verb: 'post' },
  accepts: [
    { arg: 'new_password', type: 'string' },
    { arg: 'confirm_password', type: 'string' },
    { arg: 'password', type: 'string' }
  ],
  returns: { arg: 'changed', type: 'boolean' }
});

```

## 6.2 User Managment elements

Some elements have been created for user managment. Both style and behavior of these elements have been developed , so that every user can

easily customize it at will and choose to use either one side or both sides of the element. The main elements are the ones developed for the management as: login, logout, signup and reset. The specifications for each element are indicated below.

### **<api-user-login>**

Login a user with the given credentials.

```
<api-user-login credentials="{{credentials}}"  
collection="{{collection}}"  
response="{{response}}" error="{{error}}"/>
```

Where:

- `credentials` email and password of user.
- `collection` name of collection(Object).
- `response` HTTP response message(String).
- `error` object of the error response(Object).

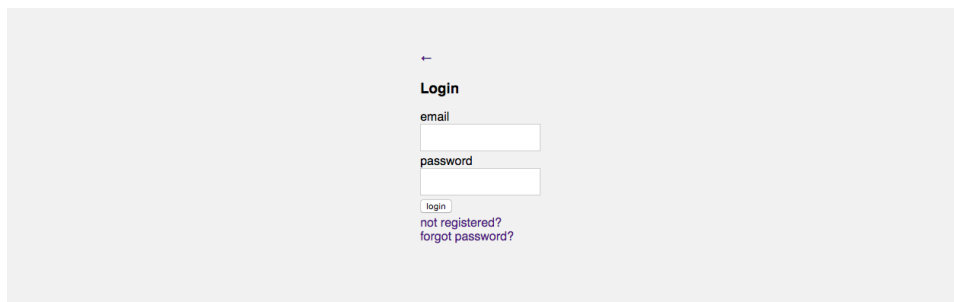


Figure 6.3: Login Element

### **<api-user-logout>**

Logout a user with the given accessToken id.

```
<api-user-logout collection="{{collection}}"  
response="{{response}}" error="{{error}}"/>
```

Where:

- `credentials` email and password of user.
- `collection` name of collection (Object)
- `response` HTTP response message (String)
- `error` object of the error response (Object)

### **<api-user-signup>**

Signup a user by with the given general information.

```
<api-user-signup credentials="{{credentials}}"  
collection="{{collection}}"  
response="{{response}}" error="{{error}}"/>
```

Where:

- `credentials` email, password, name and phone-number of user.
- `collection` name of collection (Object)
- `response` HTTP response message (String)
- `error` object of the error response (Object)

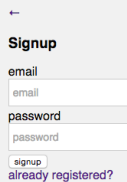


Figure 6.4: Signup Element

**<api-user-reset>**

Create a short lived access token for temporary login. Allows users to change passwords if forgotten.

```
<api-user-reset email="{{email}}"  
collection="{{collection}}"  
response="{{response}}" error="{{error}}" />
```

Where:

- email email of user (String)
- collection name of collection (Object)
- response HTTP response message (String)
- error object of the error response (Object)

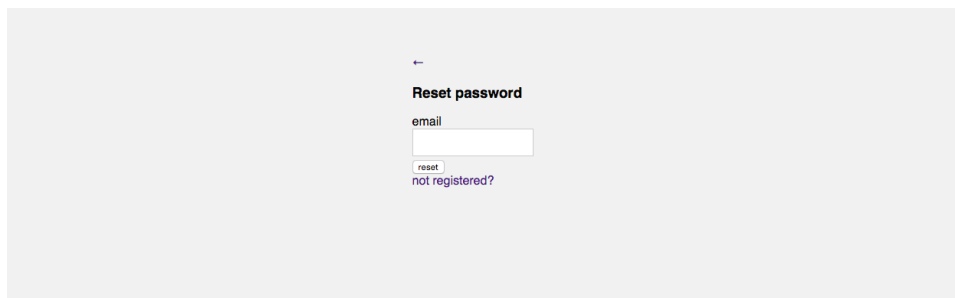


Figure 6.5: Reset Element

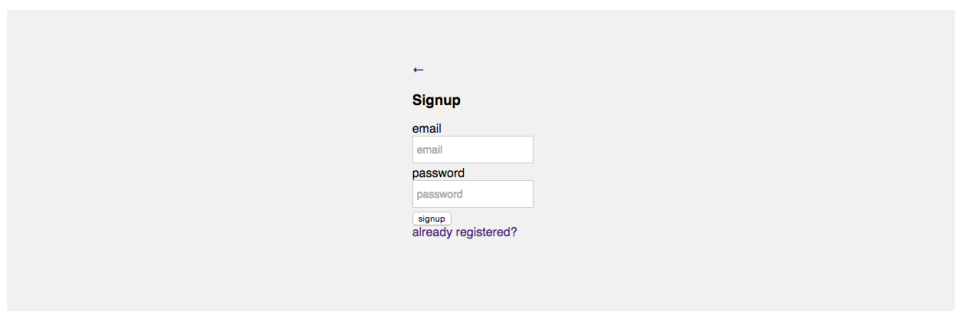


Figure 6.6: Login Element



**<form-login>**

Create a login form.

```
<form id="form" on-submit="on_submit">
  <div class="field">
    <label class="label">email</label>
    <input class="input" is="iron-input" type="text"
      placeholder="email"
      bind-value="{{credentials.email}}">
  </div>
  <div class="field">
    <label class="label">password</label>
    <input class="input" is="iron-input" type="password"
      placeholder="password"
      bind-value="{{credentials.password}}">
  </div>
  <input type="submit" value="login"/>
</form>
```

**<form-signup>**

Create a signup form.

```
<form id="form" on-submit="on_submit">
  <div class="field">
    <label class="label">email</label>
    <input class="input" is="iron-input" type="text"
      placeholder="email"
      bind-value="{{credentials.email}}">
  </div>
  <div class="field">
    <label class="label">password</label>
    <input class="input" is="iron-input" type="password"
      placeholder="password"
      bind-value="{{credentials.password}}">
  </div>
  <div class="field">
    <label class="label">firstname</label>
    <input class="input" is="iron-input" type="text"
      placeholder="firstname"
      bind-value="{{credentials.firstname}}">
  </div>
```

```

</div>
<div class="field">
  <label class="label">lastname</label>
  <input class="input" is="iron-input" type="text"
    placeholder="lastname"
    bind-value="{{credentials.lastname}}">
</div>
<div class="field">
  <label class="label">phone-number</label>
  <input class="input" is="iron-input" type="text"
    placeholder="phone-number"
    bind-value="{{credentials.phone-number}}">
</div>
  <input type="submit" value="confirm"/>
</form>

```

### <form-reset>

Create a reset form.

```

<form id="form" on-submit="on_submit">
  <div class="field">
    <label class="label">email</label>
    <input class="input" is="iron-input" type="text"
      placeholder="email"
      bind-value="{{credentials.email}}">
  </div>
  <input type="submit" value="reset"/>
</form>

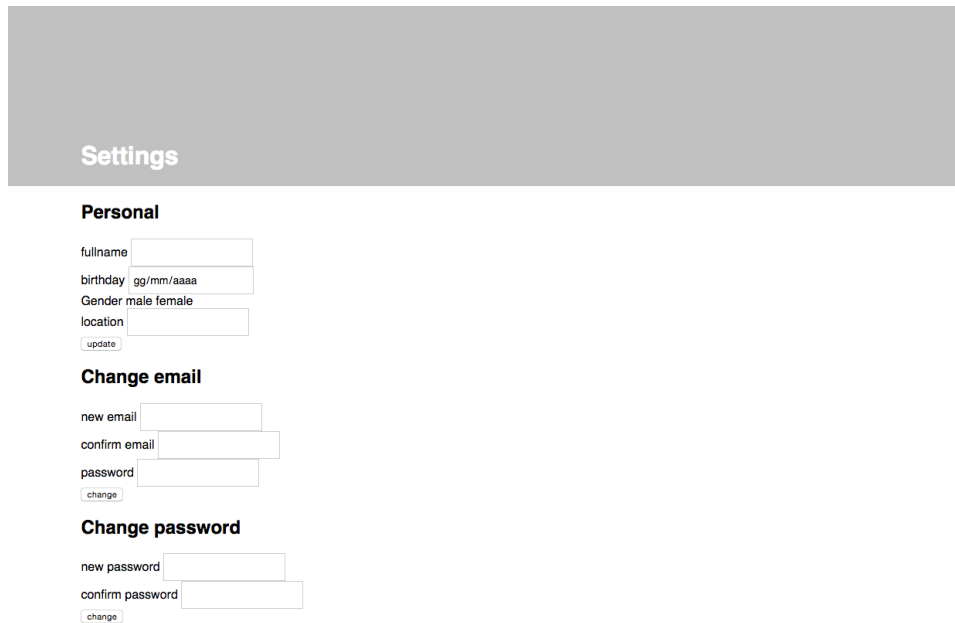
```

## User Settings

The User Settings is the page where it is possible to manage the personal data such as e-mail and password.

## 6.3 Mandrill

Mandrill is a reliable, scalable, and secure delivery API for transactional emails from websites and applications. It's ideal for sending data-driven



**Settings**

**Personal**

fullname

birthday

Gender ☐ male ☐ female

location

**Change email**

new email

confirm email

password

**Change password**

new password

confirm password

Figure 6.7: Settings Element

transactional emails, including targeted e-commerce and personalized one-to-one messages. Trusted by more than 500,000 customers, Mandrill runs on a globally distributed infrastructure that can deliver emails in milliseconds. Mandrill was developed by MailChimp, a company with more than 10 years of experience building a robust email marketing platform. MailChimp sends 15 billion emails every month for more than 8 million customers.

## Mailchimp

MailChimp is an email marketing service and trading name of its operator, a United States company, founded in 2001. The MailChimp service is accessed through a web- or mobile-based application; for some features there is an offline application. MailChimp began as a paid service and added a freemium option eight years later.[5] It was originally going to be called ChimpMail, but the name was changed after the company discovered that they could not get that domain name. The company's logo is a chimpanzee,

and the site includes numerous chimp-related graphics and humor on its website and in its communications.

In this chapter, the general characteristics of the component have been presented for user management. This component allows developers to integrate into their application functions required for managing user as login or signup.

This element lets the developers to integrate in applications important user management functions, such as login and signup. In particular Mailchimp service has been used for development and management of signup and reset password.

## Chapter 7

# Use Case: X-Blog

This chapter presents a use case of the project, the implementation of a blog via X-Project toolkit. Sections provide the representation of elements equipped with several code snippets and examples.

### 7.1 Case study

In this section the design and the implementation of a blog platform are presented.

#### 7.1.1 1st step - Models schemas definition

As to a blog platform, the essential entities to be modelled are the following: Article, Category and Author.

##### Model - Article

The model `article` defines the main post structure that must be published. Main features are: title, content and creation, update and publication date. This model has relations with `author`, `category`, `tag` and `image` models.

```
{
```

```
"name": "Article",
"properties": {
  "title": {
    "type": "string"
  },
  "subtitle": {
    "type": "string"
  },
  "summary": {
    "type": "string"
  },
  "content": {
    "type": "string"
  },
  "created_at": {
    "type": "date"
  },
  "updated_at": {
    "type": "date"
  },
  "published_at": {
    "type": "date"
  }
},
"relations": {
  "author": {
    "type": "belongsTo",
    "model": "Author"
  },
  "category": {
    "type": "belongsTo",
    "model": "Category"
  },
  "image": {
    "type": "hasOne",
    "model": "Image"
  }
}
}
```

### Model - Category

The `model` tag represents the keyword or an associate word associated to a particular article, that makes it classifiable and indexed.

```
{
  "name": "Category",
  "properties": {
    "name": {
      "type": "string"
    }
  }
}
```

### Model - Author

The `model` `author` represents the article creator. Its features are: nickname, role and full name. The author has relations with other authors, categories, tags and images.

```
{
  "name": "Author",
  "properties": {
    "name": {"type": "string"},
    "role": {"type": "string"},
    "fullname": {"type": "string"},
    "location": {"type": "string"}
  }
  "relations": {
    "articles": {
      "type": "hasMany",
      "model": "Article"
    }
  }
}
```

#### 7.1.2 2nd step - HTTP RESTful API definition

These models result in the following HTTP RESTful API, automatically generated by Loopback server.

## Authors API

**GET /api/Authors** Find all instances of the model matched by filter from the data source.

**POST /api/Authors** Update an existing model instance or insert a new one into the data source.

**PUT /api/Authors** Create a new instance of the model and persist it into the data source.

**DELETE /api/Authors/id** Delete a model instance by id from the data source.

**POST /api/changeEmail** Change email of a model instance.

**POST /api/changePassword** Change password of a model instance.

**GET /api/count** Count instances of the model matched by where from the data source.

**POST /api/login** Login a user with username/email and password.

**POST /api/logout** Logout a user with access token.

**POST /api/reset** Reset password for a user with email.

**POST /api/update** Update instances of the model matched by where from the data source.

## Articles API

**GET /api/Articles** Find all instances of the model matched by filter from the data source.

**POST /api/Articles** Update an existing model instance or insert a new one into the data source.

**PUT /api/Articles** Create a new instance of the model and persist it into the data source.

**DELETE /api/Articles/id** Delete a model instance by id from the data source.

**GET /api/Articles/id/author** Fetches belongsTO relation author.



**GET /api/Articles/id/category** Fetches belongsTO relation category.

**GET /api/Articles/id/image** Fetches belongsTO relation image.

**GET /api/Articles/count** Count instances of the model matched by where from the data source.

### Categories API

**PUT /Categories** Update an existing model instance or insert a new one into the data source.

**GET /Categories** Find all instances of the model matched by filter from the data source.

**POST / Categories** create a new instance and persist it into the data source.

**GET / Categories/id/articles** Queries articles of category.

**POST / Categories/id/articles** Creates a new instance in articles of this model.

**DELETE / Categories/id/articles** Deletes all articles of this model.

### Images API

**POST /Images** create a new instance and persist it into the data source.

**PUT /Images** Update an existing model instance or insert a new one into the data source.

**GET /Images** Find all instances of the model matched by filter from the data source.

**POST /Images/upload** Upload a new instance into data source.

**GET /Images/id** Find a model instance by id from the data source

**PUT /Images/id** Update attributes of a model instance and persist it into the data source.

**DELETE /Images/id** Deletes a model instance by id from the data source.

## Remote Methods

### Change Email

```
Author.change_email = function (new_email,
  confirm_email, password, cb) {
  if (new_email !== confirm_email) {
    cb({ error: 'email not confirmed' }, null);
    return;
  }

  var userId = getCurrentUserId();

  Author.findById(userId, function (err, user) {
    if (err) {
      cb(err, null);
      return;
    }

    user.hasPassword(password, function (err, match) {
      if (!match) {
        cb({ error: 'invalid password' }, null);
        return;
      }

      user.updateAttribute('email',
        new_email, function (err, user) {
          if (err) {
            cb(err, null);
            return;
          }

          cb(null, true);
        });
    });
  });
};

Author.remoteMethod('change_email', {
```

```
http: { path: '/change_email', verb: 'post' },
accepts: [
  { arg: 'new_email', type: 'string' },
  { arg: 'confirm_email', type: 'string' },
  { arg: 'password', type: 'string' }
],
returns: { arg: 'changed', type: 'boolean' }
});
```

## Change Password

```
Author.change_password = function (new_password,
confirm_password, password, cb) {
  if (new_password !== confirm_password) {
    cb({ error: 'password not confirmed' }, null);
    return;
  }

  var userId = getCurrentUserId();

  Author.findById(userId, function (err, user) {
    if (err) {
      cb(err, null);
      return;
    }

    user.hasPassword(password, function (err, match) {
      if (!match) {
        cb({ error: 'invalid password' }, null);
        return;
      }

      user.updateAttribute('password', new_password,
        function (err, user) {
          if (err) {
            cb(err, null);
            return;
          }

          cb(null, true);
        });
    });
  });
});
```

```

    });
};

Author.remoteMethod('change_password', {
  http: { path: '/change_password', verb: 'post' },
  accepts: [
    { arg: 'new_password', type: 'string' },
    { arg: 'confirm_password', type: 'string' },
    { arg: 'password', type: 'string' }
  ],
  returns: { arg: 'changed', type: 'boolean' }
});

```

### Upload signed url request

```

Image.signed_put = function(file_name,
  file_type, callback) {
  var s3 = new aws.S3();
  var s3_params = {
    Bucket: S3_BUCKET,
    Key: file_name,
    Expires: 60,
    ContentType: file_type,
    ACL: 'public-read'
  };
  s3.getSignedUrl('putObject', s3_params,
    function (err, signed_url) {
      if (err) {
        callback(err);
        return;
      }
      callback(null, signed_url);
    });
};

Image.remoteMethod('signed_put', {
  http: { verb: 'get' },
  accepts: [
    {arg: 'file_name', type: 'string'},
    {arg: 'file_type', type: 'string'}
  ],
  returns: {arg: 'signed_url', type: 'string'}
});

```

```
});
```

### Delete signed url request

```
Image.signed_delete = function(file_name, callback) {  
    var s3 = new aws.S3();  
    var s3_params = {  
        Bucket: S3_BUCKET,  
        Key: file_name  
    };  
    s3.getSignedUrl('deleteObject', s3_params,  
        function (err, signed_url) {  
            if (err) {  
                callback(err);  
                return;  
            }  
            callback(null, signed_url);  
        });  
};  
  
Image.remoteMethod('signed_delete', {  
    http: { verb: 'get' },  
    accepts: {arg: 'file_name', type: 'string'},  
    returns: {arg: 'signed_url', type: 'string'}  
});
```

### Get signed url request

```
Image.signed_list = function (folder, callback) {  
    var s3 = new aws.S3();  
    var s3_params = {  
        Bucket: S3_BUCKET,  
        EncodingType: 'url',  
        Prefix: folder,  
        MaxKeys: 1000  
    };  
    s3.getSignedUrl('listObjects', s3_params,  
        function (err, signed_url) {  
            if (err) {  
                callback(err);  
                return;  
            }  
        }  
    );  
};
```

```

        callback(null, signed_url);
    });
};

Image.remoteMethod('signed_list', {
  http: { verb: 'get' },
  accepts: { arg: 'folder', type: 'string' },
  returns: { arg: 'signed_url', type: 'string' }
});

```

### 7.1.3 3rd step - UI components definition

UI components definition consist of the juxtaposition of previously created HTML tags. This short example shows the creation of an article page. To compose the article page it is necessary to get article informations: title, content, author and images. As shown below, HTML code is extremely simple: considering that even the page is a Polymer element, it's enough to insert other tags in desired order.

```

<link rel="import" href="../../app-page/app-page.html">
<link rel="import" href="../../part-jumbotron/part-jumbotron.html">
<link rel="import" href="../../part-image/part-image.html">
<dom-module id="page-article">
  <template>
    <part-jumbotron id="jumbotron">
      <header id="header">
        <h1 id="title">
          <span>{{article.title}}</span>
          <span>{{article.author}}</span>
        </h1>
      </header>
    </part-jumbotron>

    <div class="container">
      <div>
        <p id="content"> {{article.content}}</p>
      </div>
      <div class="container">
        <part-image id="image"> </part-image>
      </div>
    </div>
  </template>
</dom-module>

```

```
    </div>
  </template>
</dom-module>

<script>
  Polymer({
    attached: function() {
      this.$.image.update();
    }
  });
</script>
```

The article page component has two main HTML tags: `part-jumbotron` and `part-image`.

In `part-jumbotron` article informations are showed: `article.title` and `article.author`. These informations are extrapolated from `article` property that is injected in the session by the router.

The other tag, `part-image`, loads and shows the article's image, by triggering other APIs: the activation of these APIs is made by the `attached` function that is triggered at page load time.

#### 7.1.4 4th step - UI components assembly

Since a snippet is worth a thousand words, in the following pages, the code of the pages of the app is shown. It is important to remark how easily a page can be built without writing code, but by assembling elements.

The admin part is composed of `page-collection` and `page-model-edit`. These pages are accessible via the following routes.

```
<x-router>
  <x-route route="/admin/:collection"page="page-collection"/>
  <x-route route="/admin/:collection/:id"page="page-model-edit"/>
</x-router>
```

Where: the parameter `:collection` is the name of the collection to inspect; the parameter `:id` is the id of the model to edit. These parameters are set as attributes of the page element.

- `<page-collection>` Shows the models of a collection.

```
<template name= "page-collection">
  <api-collection-schema name="{{collection}}"
    schema="{{schema}}" />
  <api-collection-get name="{{collection}}"
    where="{{where}}" page="{{page}}" perpage="{{perpage}}"
    items="{{items}}" count="{{count}}"/>
  <api-collection-where schema="{{schema}}"
    where="{{where}}"/>
  <x-table schema="{{schema}}" items="{{items}}"/>
  <x-pager count="{{count}}" perpage="{{perpage}}"
    current="{{page}}"/>
</template>
```

Where: the value `collection` is picked from the url, via the parameter `:collection`; the value `schema` is the output of `<api-collection-schema>` and the input of `<api-collection-get>` and `<x-table>`; the value `items` is the output of `<api-collection-get>` and the input of `<x-table>`; the value `where` is the output of `<api-collection-where>` and the input of `<api-collection-get>`; the value `count` is the output of `<api-collection-get>` and the input of `<x-pager>`; the values `perpage` and `page` are the outputs of `<x-pager>` and the inputs of `<api-collection-get>`. Every time the user (the admin) interacts with the pagination (`<x-pager>`) or the advanced search options (`<api-collection-where>`), `<api-collection-get>`, he regenerates the request to get the list of models using pagination and query parameters.

- `<page-model-edit>` Shows the forms to update a model.

```
<template name="page-model-edit">
  <api-collection-schema name="{{collection}}"
    schema="{{schema}}"/>
  <api-model-get name="{{collection}}"
    model-id="{{id}}" model="{{model}}"/>
  <x-form schema="{{schema}}" model="{{model}}"/>
  <api-model-put name="{{collection}}"
    model-id="{{id}}" model="{{model}}"/>
```



</template>

**x-blog**  
 articles  
 categories  
 profile

### Edit article

**title**  
 Unde veniam perferendis quæ perspicatis.

**summary**  
 Incidunt accusantium aut.

**content**  
 Vero ipsum animi sed.  
 Minima enim at quam.  
 Nemo deleniti quia eligendi.  
 Eum eos aut.  
 Ea qui facere vitae ut possimus qui.  
 Quis ad exercitationem iste aut.

**image**

**author**  
☐

**category**

**tags** ☐

**published at**

Figure 7.1: Screenshot: X-Blog - Edit an article

Where: the value schema is the output of `<api-collection-schema>` and the input of `<x-form>`; the value model is the output of `<api-model-get>` and `<x-form>` and the input of `<api-model-put>`. Once the page is ready (initialized and served by the local router): `<api-collection-schema>` fetch the schema; `<api-model-get>` fetch the model (a article or a tag) identified by id; `<x-form>` shows the form to edit the model. When the model changes (is updated via the form) `<api-model-put>` sends a request to the server to update the database (via the corresponding HTTP RESTful API).

The user part is essentially composed by two pages: `page-articles` and `page-article`.

```
<x-router>
  <x-route route="/"page=" page-articles"/>
  <x-route route="articles/:id" page=" page-article"/>
</x-router>
```

- `<page-articles>` Shows the list of articles.

```

<template name="page-articles">
  <api-collection-get name="Article"
    perpage="10" page="{{page}}"
    items="{{articles}}" count="{{count}}"/>
  <template is="dom-repeat" items="{{articles}}">
    <li>{{item.title}} {{item.date}}</li>
  </template>
  <x-pager perpage="10" total="{{count}}"
    current="{{page}}"/>
</template>

```

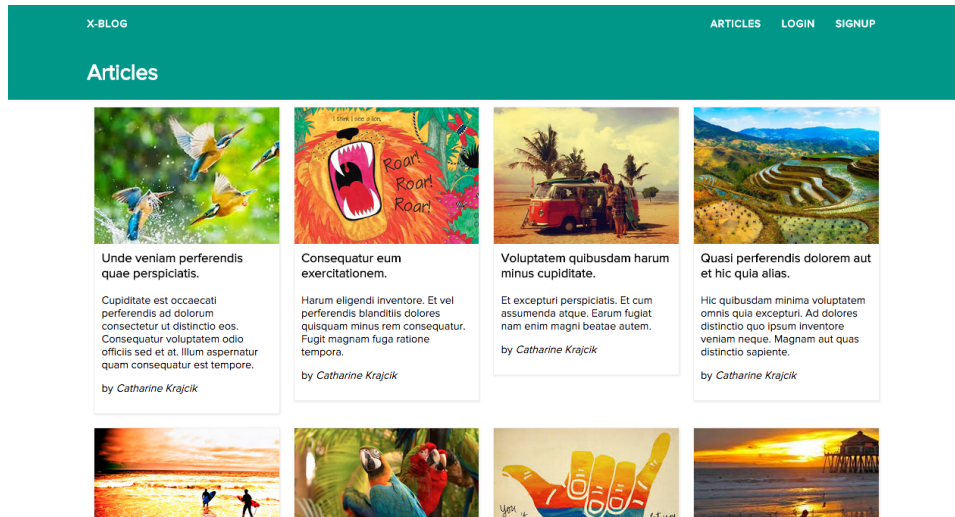


Figure 7.2: Screenshot: X-Blog - Articles page

Where: the value `articles` is the output of `<api-collection-get>` and the input of the `<template>` iterator; for each item in `articles`, a list item with the article info (title and publishing date) is printed; `<x-pager>` component is the same used in the page `<page-collection>`.

- `<page-article>` Shows a article.

```

<template name="page-article">
  <api-model-get name="Articles" model-id="{{id}}"
    model="{{article}}"/>
  <h1>{{article.title}}</h1>
  <h2>by{{article.author}}</h2>

```

```
<h3>on{{article.date}}</h3>
<div>{{article.content}}</div>
</template>
```

Once `<api-model-get>` has fetched the article (identified by id), title, author, date and content of the article will be shown. It has been shown how to use `x-project` toolkit to build a simple blog application. It can be extended, by adding new features, by following the 4-steps document-driven process: defining new models, generating corresponding API, defining new UI components, assembling the UI. The growth of the complexity of the application does not affect the complexity of the development. `x-project` source code is published on GitHub at <http://github.com/x-project>.

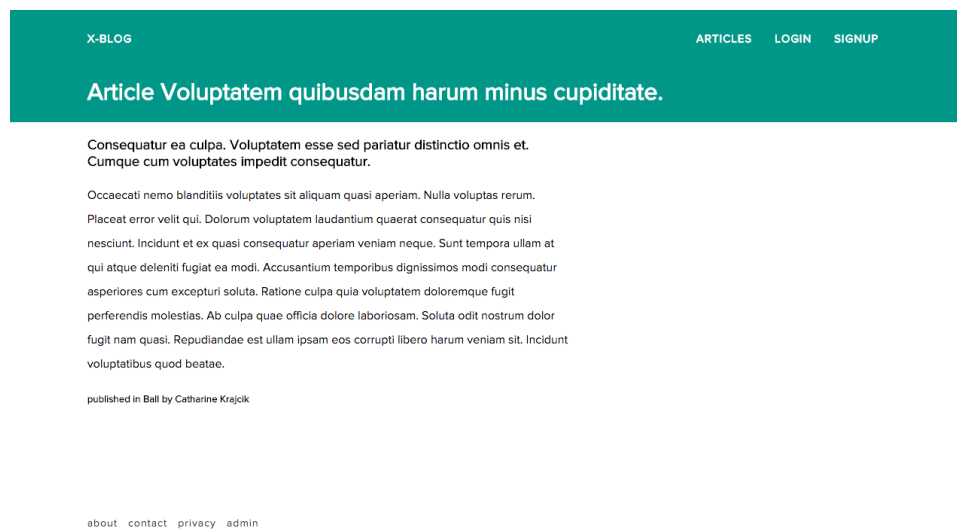


Figure 7.3: Screenshot: X-Blog - Article page

This chapter the case of use has described, its implementation and the representation of elements equipped with code snippets and examples have been presented.

## Chapter 8

# Conclusions

### 8.1 Work performed

Most of the work has been performed to lay X-Project groundwork. As seen in previous chapters, the work is based on the union of several technologies and services. This mix gives birth to an hybrid platform that lies between content management systems and web application frameworks. The output is a functional and cutting edge product.

X-Blog, the use case analyzed, has been real useful to understand benefits and limits of this platform, and to realize where, this project, needs improvements. As said before, client-side benefits are borrowed from Polymer and Web Components environment. Code reusability and access to reusable code are, clearly, one of the most important aspects of the project that lead to the implementation of a set of elements (X-Elements, see 4.6). Moreover, the encapsulation and the isolation of various parts of a component (presentation, behaviour and content) lead to the creation of stand alone service elements (S3 component), behavioural elements(API elements) and mixins(style elements). Server-side benefits are represented by the easy way of API creation: describing procedures and model definitions are direct and can generate API and behavioural elements as well.

Finally, the union between server-side and client-side technologies allowed to create vertical elements that cross all the architectural stack.

## 8.2 Future developments

At the moment X-Project shows up as basic platform on top of which many applications can be implemented. Future developments are already the present. Nowadays, X-Blog (see chapter 7), X-Journal and X-Commerce are in different development stages. The goal of these project is to make products to categorize and fill specific needs. For example, X-Commerce is an application dedicated to transactions for the sale of goods and services. In parallel with the creation of new platforms, X-Elements (see 4.6) is constantly replenished with new elements. In X-Commerce case these elements are implemented to manage credit cards, handle transactions and integrate payment gateways.

Moreover, X-Project needs to be more functional. The development of a service that allows to install the whole platform with a command can surely raise the functionality level: this feature is being implemented via Yeoman Generator<sup>1</sup>.

Media Management Component (see chapter 5) needs a thumbnail generator to show different dimensions images according to the page in use (preview or extended visualization). Moreover, other services, in addition to Amazon S3, must be integrated to let the user to choose.

User Management Component (see chapter 6) needs the implementation of elements that unite the various services offered by many social networks. The advantage of this service is to simplify any interaction between user and services, such as logging in and signing up via Facebook or Google account.

Finally, X-Project is a work in progress that will evolve in the future, in

---

<sup>1</sup>Yeoman is an open source client side development stack, consisting of tools and frameworks intended to help developers quickly build high quality web applications.

order to reach a customization and ease of use high level.

# Bibliography

- [1] AIIM. *What is Web CMS*. URL: <http://www.aiim.org/What-is-Web-CMS-WCM-System-Content-Management>.
- [2] AWS Amazon. *About AWS*. URL: <http://aws.amazon.com/about-aws/>.
- [3] AWS Amazon. *AWS SDK for Javascript NodeJS*. URL: <https://aws.amazon.com/sdk-for-node-js/>.
- [4] AWS Amazon. *What is Cloud Computing?* URL: <http://aws.amazon.com/what-is-cloud-computing/>.
- [5] Auth0. *Auth0 Single Sign On and Token Based Authentication*. URL: <https://auth0.com>.
- [6] Polymer Authors. *Custom elements extend the web*. URL: <https://www.polymer-project.org/1.0/docs/start/what-is-polymer.html>.
- [7] Joomla! Community. *New Features in Joomla! 2.5*. URL: [community.joomla.org/blogs/community/1533-new-features-in-joomla-25.html](http://community.joomla.org/blogs/community/1533-new-features-in-joomla-25.html).
- [8] WebComponents.org contributors. *CustomElements*. URL: <http://webcomponents.org/polyfills/custom-elements/>.
- [9] WebComponents.org contributors. *Shadow-DOM*. URL: <http://webcomponents.org/polyfills/shadow-dom/>.

- [10] Jeff Barr for Aws Official Blog. *Amazon S3 – Two Trillion Objects, 1.1 Million Requests / Second*. URL: <https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/>.
- [11] Arun Ranganathan for Mozilla. *Cross-site xmlhttprequest with CORS*. URL: <https://hacks.mozilla.org/2009/07/cross-site-xmlhttprequest-with-cors/>.
- [12] WordPress Foundation. *WordPress.org*. URL: <http://wordpress.org>.
- [13] ghost.org. *Just a blogging platform*. URL: <http://ghost.org>.
- [14] Joomla! *Introduction for developing a Model-View-Controller Joomla 1.5 Component*. URL: [docs.joomla.org/Developing\\_a\\_Model-View-Controller\\_Component/1.5/Introduction](https://docs.joomla.org/Developing_a_Model-View-Controller_Component/1.5/Introduction).
- [15] KeystoneJS. *KeystoneJS Overview*. URL: <http://keystonejs.com>.
- [16] Michael Cobb Margaret Rouse. *Digital Signature*. URL: <http://searchsecurity.techtarget.com/definition/digital-signature>.
- [17] marketwired.com. *Joomla! CMS Passes 50 Million Downloads*. URL: [www.marketwired.com/press-release/joomla-cms-passes-50-million-downloads-1882565.html](http://www.marketwired.com/press-release/joomla-cms-passes-50-million-downloads-1882565.html).
- [18] Lee Matthews. *Just how big is Amazon's AWS business?* URL: <http://www.geek.com/chips/just-how-big-is-amazons-aws-business-hint-its-absolutely-massive-1610221/>.
- [19] Rich Miller. *Estimate: Amazon Cloud Backed by 450,000 Servers*. URL: <http://www.datacenterknowledge.com/archives/2012/03/14/estimate-amazon-cloud-backed-by-450000-servers/>.



- [20] Xtag Mozilla. *XTag The custom elements polylib*. URL: <http://www.x-tags.org>.
- [21] V. Okanovic. “Web application development with component frameworks”. In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*. 2014, pp. 889–892. DOI: 10.1109/MIPRO.2014.6859693.
- [22] John O’Nolan. *ghost.org Kickstarter Campaign*. URL: <https://www.kickstarter.com/projects/johnonolan/ghost-just-a-blogging-platform/description>.
- [23] Pankaj Parashar. *CAn Introduction to Web Components and Polymer*. URL: <http://www.sitepoint.com/introduction-to-web-components-and-polymer-tutorial/>.
- [24] Dan Pupius. *Links in a single page app*. URL: <http://neugierig.org/software/blog/2014/02/single-page-app-links.html>.
- [25] Amazon S3. *About S3*. URL: <http://aws.amazon.com/s3/>.
- [26] Stormpath. *Stormpath User Management*. URL: <https://stormpath.com>.
- [27] Paul Suda. *A review of Node JS content management system*. URL: <http://waitingfortheelevator.com/a-review-of-node-js-content-management-systems/>.
- [28] trends.com. *cms trends*. URL: <http://trends.builtwith.com/cms>.
- [29] Userapp. *User Management for Web and Mobile Apps with UserApp*. URL: <https://userapp.io>.

- [30] W3Techs. *Usage Statistics and Market Share of Content Management System for Websites*. URL: [http://w3techs.com/technologies/overview/content\\_management/all](http://w3techs.com/technologies/overview/content_management/all).
- [31] Mike Wasson. *Single Page Application: Build Modern Responsive Webapps with ASP.NET*. URL: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>.
- [32] Wikipedia. *Content Management System*. URL: [https://en.wikipedia.org/wiki/Content\\_management\\_system](https://en.wikipedia.org/wiki/Content_management_system).
- [33] Wikipedia. *Joomla!* URL: <https://en.wikipedia.org/wiki/Joomla>.
- [34] Wikipedia. *Single Page Application*. URL: [https://en.wikipedia.org/wiki/Single-page\\_application](https://en.wikipedia.org/wiki/Single-page_application).
- [35] Wikipedia. *WebComponents*. URL: [https://en.wikipedia.org/wiki/Web\\_Components/](https://en.wikipedia.org/wiki/Web_Components/).
- [36] Wikipedia. *WordPress*. URL: <https://en.wikipedia.org/wiki/WordPress>.
- [37] X-project. *API Element*. URL: <https://github.com/x-element/>.