CROWD FUNDING OF CIVIC PROJECTS

by Andrei Hodorog

Student number: c1332008

CM3203 – One Semester Individual Project

Module credits: 40

Supervisor: Prof. Dr. Omer Rana

Moderator: Prof. Dr. Jing Wu

Cardiff School of Computer Science and Informatics

Cardiff University

May 2017

Table of contents

Abstract

The aim of the project is to develop a working prototype for a crowdfunding platform for Civic Projects and individuals with great business ideas that promotes micro-financing, peer reviews of the ideas and gathers statistics in the background.

The key outcome of the project consists of a working prototype that meets the mandatory requirements specifications mentioned in Section 3.1.1 and a part of the optional functional requirements criteria mentioned in Section 3.1.2, leaving a more extended set of functionalities for later stages, mentioned in Section 6. The end prototype should also meet the non-functional requirement of being a horizontally scalable platform that supports growth over time and stores that of users in a secure manner. The developed application should overcome some of the key limitations of the existing similar crowdfunding platforms.

The approach used to achieve this was to design and implement front end with a graphical user interface for the users and admins where the projects can be posted, viewed and moderated and a back end in which all the data related to users and campaigns is stored and processed. A dashboard back office has also been developed that contains all the statistics and is only visible to the members of the admin committee. The back end is linked to a non-relational database designed to be scalable and efficient for the storage, retrieval and persistence of new data. The two concerns were separated and the communication between them has been done using a REST API acting as the connective interface, through which the data circulates.

The key outcomes of the end prototype have been achieved and tests for the evaluation of both the front end and the back end have been conducted. The front end attractiveness and usability was verified through a survey targeting a sample of millennial potential respondents, the back end performance and scalability was tested using a configuration of multiple application instances. The results of these tests supported the proposed functions and the non-functional requirement for horizontal scalability for the prototype of XpressStarter, a crowdfunding platform for civic projects.

Acknowledgements

I would like to express my sincere thanks to my project supervisor, Prof. Dr. Omer Rana for his continuous help, support and encouragement which he has provided me throughout the course of this project and his constant trust in my abilities to complete this project successfully. He has always replied promptly to my queries when I needed advice. In addition to academic support, the meetings with him were always entertaining and enjoyable, enhancing the overall project experience.

I would also like to express my thanks to Mr Akmal Hanuk, chief executive of the Cardiff-based Islamic Banking and Finance Centre for his insight into the project objectives and advice from a commercial perspective.

I place on record, my sincere thank you to Professor Stuart Allen, the head of school of Computer Science, for his continuous encouragement, patience and promptitude of addressing my queries regarding marks, rules, regulations and other academic matters.

I would also like to express my thanks to Prof. Dr Walter Colombo, Prof. Dr. Pete Burnap, Prof. Dr Ioan Petri and Prof. Dr. Thomas Beach for giving me the opportunity to get involved in research projects which have provided me with a lot of valuable knowledge that has also been used as a complementary aid to this project.

I am also very grateful to my girlfriend, miss Elena-Andreea Mut, and my mother, miss Diana Hodorog for their continuous encouragement and moral support offered throughout the development of this project.

I would also like to take this opportunity to express my gratitude to all the members of the Computer Science department for their help and support.

1. Introduction

The world of entrepreneurship is constantly generating business ideas, from initiatives that aim to generate profit to the ones that support civic causes. The online environment has made it easier for the promoters of such ideas to connect with potential benefactors via dedicated websites or platforms that enable the functioning of what is known as “crowd funding” - the process where funding for a business idea is not gathered via the traditional routes (banks) but through a community of people that offer financial support in exchange for a benefit offered at a later stage. A number of such sites have emerged over recent years but they fail to differentiate between potentially successful and unsuitable initiatives, allowing potential investors to invest blindly. Once funded, many ideas often fail or are unsuitably put into practice, in a way that is completely different to the one initially advertised to the benefactors.

This report will outline the design decisions made during the system design and implementation process of the XpressStarter, a web platform that matches founders of civic business initiatives with potential benefactors and also presents an extra element of transparency and quality assurance, that aims to address the problems outlined above. This is achieved by a team of administrators that ensure that only the submitted ideas that satisfy certain standards are published and become active. The report will also discuss the challenges faced during the development process of XpressStarter.

1.1. Personal Involvement

Having myself been involved in the world of entrepreneurship (running a limited company in the UK that provides web design, web development and online marketing services to other entrepreneurs), I have come to experience first hand the difficulties of securing funding, as well as the gaps in the process of successfully identifying worthwhile business ideas, from an investor’s perspective. In the case of social causes in particular, having a platform that promotes ideas in a transparent way and funding represents both a social and technical issue of great importance for me.

1.2. Aims and Objectives

The aims of the projects are to:

- identify key features and limitations of existing crowdfunding platforms (credit unions, crowd funding, peer to peer lending, micro-finance based sites) and embed these findings into the system design and development of XpressStarter;
- highlight key areas of their Customer Relationship Management (CRM) and Risk Management tools;
- develop a front end and design a visual user interface that ensures enough information is presented in an appropriate style to attract donors and maintain their “loyalty” (e.g. recurring visits and funding);
- develop a front end that ensures all the HCI and UX / UI principles are met; an appropriate combination of colours, animations and visual effects are used; the website is displayed appropriately across all screen sizes / resolutions;
- implement a back end system (e.g. a database, an API) that enables donor and campaign information to be kept in a secure manner and ensure scalability and growth over time.

The end outcome was tested using the following methods:

- A questionnaire based on the SUS (Simple Usability Scale) method: the usability and attractiveness of the designed Graphical User Interface and front end implementation was tested through distributing the Questionnaire in Appendix 2 to potential users. This had a strong focus on the perception of the user of the colour schemes and types of layout chosen, as well as exploring the idea of user trust in crowdfunding platforms, and whether the design correlates to their perception. The questionnaire was distributed via social media platforms and was aimed at socially active millennials with a potential interest in startups and business ideas. The front end evaluation and its results are described in detail in Section 5.1 . A description of the evaluation of the demographics of the questionnaire respondents can be found in Section 5.2 .
- Simultaneous requests emulation to a load balancer using JMeter: the back end performance and scalability has been tested through the emulation of the activity of multiple users (groups of 10, 100 and 1000 users) using JMeter. This test also aimed to verify whether the platform is horizontally scalable. Using the OpenStack account provided by Cardiff University, a basic environment of 2 application servers and a load balancer was set up. The back end performance and scalability testing and its results are outlined in detail in Section 5.3 .
- jUnit tests: these tests that are programmatically set up to check the output of various portions of code in order to make sure that they do not affect the output of other independent features. A complete description of these tests and a complete outline of the output can be found in Section 5.4 .

1.3. Beneficiaries

The platform creates a peer-review-type system of identifying viable business ideas through a community rating system and a network of intermediaries whose job it is to keep a high standard of the business ideas submitted and to ensure goal completion after funding. Correspondingly, the audience for this project is split into three main groups:

- “**Funding beneficiaries**”: social entrepreneurs searching for funding that are unwilling to use banks;
- “**Benefactors**”: individuals who wish to donate in social business ideas in exchange of either a pre-ordered product or some other kind of benefit. Equity is usually not part of the benefit package;
- “**Administrators**”: admin committees will have the responsibility to identify the unviable business ideas hosted on the website in order to ensure a high standard of idea submission. They will also be in charge with ensuring the commitments of the beneficiaries to the benefactors are being met after their idea is funded. This is seen as the unique selling point of the platform, allowing an added level of quality control which is not offered in similar platforms.

1.4. Scope and Constraints

The broad scope of this project is comprised of creating a crowdfunding platform that is compatible across different devices and platforms and is focused on civic projects and trust. The main technical constraint is represented by the fact that this project is web-based and must be accessible from a web browser. However, despite this, the

foundations for developing a mobile app in the future have been set during the development of the platform. Through the use of a REST API, integrating the back end in third party apps has been made simple, as they this would only rewire the front end code implementing the API specifications. If the wish to develop a mobile application (Android / iOS / Windows Mobile) would arise in the future, the back end would not need any modifications. Based on the final outcome, an app can easily be developed should this be required. A full description of the concept related to the fact that a future-proof back end has been developed through the design choice of splitting the application in 2 separate micro services and the use of a REST API is outlined in full in Section 3.3.2 .

A significant constraint was the limitation of time, which has made me prioritise the set of functionalities to implement at the prototype stage. The functionalities selected to be implemented are:

1. The system must be able to handle 3 types of users: benefactors, beneficiaries and admin committees;
2. The benefactors must be able to view campaigns proposed by different beneficiaries, split into different categories;
3. The campaigns proposed by the beneficiaries must clearly show their description, the category they belong to, the amount pledged by the benefactors, the target donation amount and the progress percentage towards the donation goal;
4. The benefactors must be able to express their interest in campaigns proposed by potential beneficiaries that have not yet been approved by a member of the admin committee;
5. The beneficiaries must be able propose new campaigns;
6. The benefactors must be able to pledge amounts towards the goal of a campaign;
7. The admin committee members must be able to approve or reject the campaigns proposed by the potential beneficiaries.

All the functionalities listed above are part of the mandatory requirements outlined in Section 3.1.1. The plan to implement the functionalities related to the optional requirements in Section 3.1.2 is described in detail in Section 6. Future work .

Another constraint of this project is that the User Interface needs to be kept similar to other similar existing solutions, in order to encourage users to transition from other crowdfunding platforms to ours, that promotes XpressStarter, where civic causes are promoted apart from profitable business ideas. Nevertheless, this constraint should not impede improvements to the user interface as compared to other platforms or impede the integration of original ideas that attract and encourage both beneficiaries to post their business ideas and benefactors to invest.

If XpressStarter were to be deployed in a real world scenario and available to the general public, several regulatory constraints will need to be taken into account. Some of them are closely related to the implementation, such as the payments processing. A full outline of the regulatory constraints under which XpressStarter might fall can be found in Appendix 3: Regulatory compliance of crowdfunding platforms in the UK.

1.5. Approach

The XpressStarter platform will be comprised of a front end (GUI) and a back end in which all the data related to users and campaigns will be stored and processed. The

system is therefore split between a user interface, a back end database and the connective interface between the two. The front end for the users and admins is where the projects can be posted, viewed and moderated. A dashboard back office will contain all the statistics and will only be visible to the members of the admin committee. The back end will contain a non-relational database designed to be scalable and efficient for the storage, retrieval and persistentance of new data. The data will be retrieved by the front end in the JSON format through a REST API.

My initial research into the crowdfunding environment revealed that trust is an important requirement for both potential benefactors and idea submitters. I have taken this knowledge into the development of the platform, by introducing a system of administrators whose main task is the quality assurance of business ideas. The platform will therefore have three types of users: benefactors, beneficiaries and administrators. The administrators will have the right to review and disapprove (if necessary) the proposals from the beneficiaries if they do not demonstrate sufficient experience and expertise in the field of the industry their proposal is related to. The beneficiaries will have the opportunity to demonstrate this on their personal profile, which will have a structure similar to a CV. The beneficiaries will also have the opportunity to show interest in multiple projects without donating, through liking the projects and stating the reasons for interest in the dedicated comments area of each campaign. This extra function of community approval is linked to present trends of involving users into the review of products and services and ultimately leading to customer empowerment in the digital age ([1] Hunter and Garnefeld, 2008).

1.6. Assumptions

In order to construct the system, a number of assumptions were made in order to aid the design of the platform.

Community approval as quality assurance - the modern product is no longer sold through intense “spotless” ads, but rather through community acceptance and authentic user content. The economy is now embedding authenticity and user reviews in every part of commercial life. A similar principle was applied for this social funding platform. Instead of simply advertising a business idea and waiting for potential benefactors, the content is letting admin assessments speak for the product.

Community roles - the platform assumes the users are split into three main categories: beneficiaries or founders, benefactor or donors and administrators. The level of access will differ according to these groups, with admin members able to see a front-end dashboard with relevant stats. The community roles levels of access are crucial to the functioning of the platform.

Trust as a channel for risk reduction - community trust acts as the main guarantee for users within the platform. A delegated team of admins is using a data-driven approach to identify the most likely and the least likely projects to succeed, therefore giving a more quantifiable view to the act of funding a business idea and so reducing the inherent risk in doing so.

1.7. Summary of methodology

The concept behind XpressStarter has been informed by research into the latest design trends, as well as the key positive traits of the platforms presented in section 2.

Background. An initial questionnaire (fully outlined in Appendix 1: Questionnaire for the evaluation of front end design, attractiveness demographics) was launched in order to inform the design of the platform and gain insight into any relevant user attitudes. The questionnaire was split in 3 main sections: Design, Trust and Demographics, each collecting quantifiable data on areas crucial for the development of the platform.

The *Design* section contained a selection of screenshots from the proof of concept and enquired user perceptions on the layout and structure presented. This informed changes and supported the design choices made later on in the project.

The *Trust* section looked at user perceptions of current platforms in contrast with the offering due to be launched by XpressStarter, where admin staff will be dedicated to filtering down to the most interesting of business ideas and ensuring quality along and beyond the funding process.

Finally, the *Demographics* section collected data on questionnaire respondents that will inform later stages of the website development and a more in-depth user segmentation.

The results of the *Design* and *Trust* sections of the questionnaire are outlined in Section 5.1 and the results of the *Demographics* section are outlined Section 5.2 .

1.8. Key Challenges

The main challenges faced throughout the project are mainly related to prioritising of work and learning new technologies in order to deliver a robust prototype.

The key challenges faced are outlined below:

Requirements prioritisation
Time constraints and the need to manage sponsor expectations meant I needed to implement a series of requirements at prototype stage while developing the platform in a way that allows for future additions and increased functionality. Having used open-source technologies, this proved to be not always easy to do. I opted to implement the minimum requirements as well as a part of the optional requirements outlined in Section 3.1 following that a more extended set of requirements will be implemented later on.

Choice of technologies
With the abundance of front end and back end technologies I needed to select the suitable set of programmes that match the scope of XpressStarter. For the front end, I opted for open-source frameworks, which offered me a wide enough spectrum of functionalities while also being a growing segment (Developer.telerik.com, 2017).

Balancing waterfall and agile ways of working
The nature of the work on this project involved responding dynamically to challenges both concerning the development process and the wider overall project scope. This, combined with the time limitations around submitting a working prototype, meant I ultimately needed to operate in an agile way, despite only having a limited number of iterations, due to supervisor availability and the regulations of project work. This

meant I needed to adapt to both methodologies and take elements from each in a way that guaranteed the completion of the project.

Front end and back end integration
A similar point impacted by the time limitation was integrating the front and back end, having in mind the tight deadline for creating a mock design after having presented the requirements. This was challenging at times, as any change needed to be reflected in both front end and back end.

New front end technology
Having had experience with certain front end technologies during my placement year, I have initially planned to use those I was familiar with. However, after researching the benefits of a number of other channels, I came to the conclusion that Express JS offered some useful functionality that could further build on those offered by the technologies I was already familiar with. This meant that while developing the working prototype I also needed to learn this new technology. Ultimately, this contributed to my development, as outlined in Section 8. Reflection on learning

Achieving scalability and performance
Since the design and specification phase of this project, scalability and performance had to be taken into account. The project needs to be robust and scalable enough to meet loads ranging from tens of users to thousands of users, while providing a responsive enough service to ensure high client satisfaction;

Making the implementation future-proof
The technologies used in the project needed to be modern but mature at the same time, ensuring that the tools and libraries that are in use are maintained by the open-source community throughout the lifecycle of the project.

Standardisation of data flows and interface contracts
A standardised API (that follows certain design patterns widely used in the industry) was needed in order to be able to design a robust front end and easy integration with third party systems;

Picking the right frameworks, tools and libraries
Frameworks allows writing high level code, helping to have a short path from concept (design) to implementation, whilst also retaining a choice of vendors at any point. Using the right tools allows the platform to be agile in the event of architecture change.

1.9. Summary of main outcomes

- A data-driven system of identifying the business ideas that have a high potential for success;
- A system of different permission levels for the platform users; allowing admin members to see a dashboard with insight into the ideas with the greatest potential to succeed;
- Allowing for business ideas that have not been backed by the members of the admin committee to still be featured on the platform if supported by general platform users;
- Ensuring an effective process for verifying the successful implementation of the business goals, post-funding.

2. Background

A business idea will inevitably need a financial basis in order to go to market. Traditionally, this has been achieved either via personal savings, help from family or friends or, more typically, through an investment from a bank. This has long been the channel of choice for entrepreneurs at the beginning of their journey. However, the funding landscape has seen a shift from banks and other financial institutions being viewed as main funding sources, to including a more open ecosystem of potential benefactors ([3] Aldrich, 2014). Instead of the regulated environment offered by the bank, an open community is seen as a better, more flexible source of financial support. While this unmediated approach of crown funding channels offers more flexibility, it often lacks the necessary expertise to differentiate between projects likely to succeed and projects unlikely to do so. Additionally, the platforms are often not involved in the steps post full funding, where the benefactors are offered the benefit packages they acquired through their donation. The XpressStarter platform aims to solve these issues by using a community of administrators to maintain a high standard of idea submissions. This level of transparency will support good business ideas and help to isolate and identify the ones less likely to succeed.

The second problem that this project aims to solve is the lack of structured, data-based channels of civic projects. Traditionally, civic projects are less likely to receive the same amounts of funding as for-profit business ideas, mainly because of the view that there is no real later gratification for the benefactors (Hollow, 2013). However, the overall benefits of funding such ideas is undisputable when it comes to the impact on society and a correlation is starting to form between these civic initiatives and increasing funding amounts ([4] Harding, 2004). On Kickstarter, a popular crowdfunding platform, projects labeled “civic” were fully funded 81% of the time ([5] FastCompany, 2014). Having identified this gap, XpressStarter aims to be a platform that identifies and promotes the best of civic business initiatives, in a transparent and risk-reduced environment.

2.1. Theory associated with the problem area

In a broader commercial setting, consumers are becoming less and less sensitive to generic advertisements and sales pitches. Instead, they are seeking product advice and proof of use from a wider community, calling not only friends but a wider network of users to help them ([6] Matsuo and Yamamoto, 2009). This is an indication of a shift in the mentality of the consumer from a general trust towards the brand to an inclination to seek out authentic content produced by real buyers or service users, as well as an overall tendency towards customer empowerment in the ‘digital age’.

This idea of the community acceptance economy is applicable not only in a commercial setting. In areas other than the strictly commercial one, the user is increasingly having a consumer mentality. By creating a platform where good business ideas are filtered from within, the success rate of business funding can increase, ultimately creating a more streamlined experience for both founders and benefactors. The overall expertise of the community will come together to ensure transparency and a better understanding for the person wishing to donate money.

2.2. Solutions relevant to the problem area

There are a number of crowdfunding platforms that deal exclusively with civic projects, or have civic projects as a category within a broader offering. This section gives a brief overview of these, while identifying key strengths and weaknesses that were later used in developing the concept behind XpressStarter.

Kickstarters offers an attractive user interface and design, and a view targeted at each type of user. It is split into 3 options 'Discover, Start a project, About us' - aimed at donors, creators and people seeking to find out more about the platform. Projects can get over-funded (past target), which is not necessarily a positive aspect from the perspective of those seeking the fair distribution of donations, and the website operates on a basis of deeming an idea unsuccessful and returning the money if the full amount is not reached in due time.

Bright colours are used extensively throughout the site, which seems to be an integral part of the site's branding. Creators and their projects are displayed front and centre. They are the first thing a user sees when they land on the page. Kickstarter is effective at placing Call to Action links directly into the header of the site, so encouraging users to sign up or start a project themselves. While this approach is suitable for Kickstarters, XpressStarter will aim to focus on emphasizing the available business ideas on the website through the interface and not use that space as a way to attract users.

The significant problem that civic projects would likely face when using Kickstarter is that it is a rewards based platform. This means that for every pledge that is made there usually tends to be an associated reward (e.g. an early version of a product, a thank you t-shirt, etc.); essentially meaning the benefactor is an early customer of the company.

As civic projects will not offer any direct rewards such as the ones described above, they are not well suited to this specific model of crowdfunding. However, that is not to say that Kickstarter is of no use, as there are certain aspects that would lend themselves very well to such an application. With that in mind, an alternative method that lists the final outcome as the effective reward would be well suited.

Spacehive has a similar model to Kickstarter. However, Spacehive is tailored to cater for civic projects it and it does so by having a vetting system in place. Projects listed on Spacehive must show the impact they would have for the benefit of the community. This idea will be used in the development of XpressStarter, but the assessment will not only be made subjectively by the proposer of the business idea but also through community approval and admin assessment.

Spacehive places a large focus on community. They highlight a partner account on Spacehive who are behind a number of projects. By aggregating the projects in a community page potential backers can follow the community and receive targeted updates. Community pages are intentionally made to feel like those of a social media platform, something XpressStarter will also aim to integrate. Potential backers on Spacehive are encouraged to follow community pages, in a similar fashion to Twitter, to receive updates.

Project pages are clear and well structured. Each project shows the title towards the top of the page, where the creator, location and development stage of the project are also highlighted. This is useful as it gives an overview of the project in around 10 words.

Akhuwat is another popular choice for civic projects funding with an interesting investment model. In order to fund projects, Akhuwat accepts relatively large donations from backers and then provides these as part of an interest free loan. The key difference between Akhuwat and the two micro-financing solutions mentioned above is that the investment is returned in this case, allowing it to be reinvested within the community and more importantly, amongst its people. However, such a solution may not be viable for larger civic projects.

Whereas modern micro-financing solutions focus heavily on online, on-demand project financing, Akhuwat caters to an audience who may not have an internet connected device readily available to them; thus its approach is more traditional in the sense that they use spiritual locations and volunteers to spread their message. A key objective proposed by the team is to convert borrowers into donors, which is perhaps the opposite of what Kickstarter hopes to achieve - turning backers into creators.

Akhuwat provides a simple and clear overview of the currently active projects. Each listing features the project title, its location, a short description and its funding goals.

In terms of platform design, the page is however not to a particularly high standard. There are instances of overlapping texts and unaligned elements throughout the page.

JustGiving

JustGiving is the final example of a crowdfunding platform whose functionality was used in the development of XpressStarter. The platform offers an option specifically targeted at undecided users, whereby money can be donated not directly to a specific project but rather to a “pot” dedicated to the overall platform. This is a unique concept in the online crowdfunding landscape and something aimed to be developed in later stages for XpressStarter.

In addition to the platforms above, a number of other similar websites were identified. There however were targeted at individual, mostly ad-hoc civic ideas that generally do not aim to become an actual business. Examples for these websites include GoFundMe.com and FundingCircle.com.

3. Specification and Design

This section is going to present the requirements of the project agreed with the project supervisor and the client. This is followed by an overview of the business logic of the application and the database design that correlates with that business logic. An overview of the technical application architecture together with a brief description of the data flow is also presented. The final part of the section outlines the Graphical and User Interface Design specifications. In this section, the design choices made for this project are presented at a high level. The lower level details are presented in the next chapter, Chapter 4: Implementation.

3.1. Requirements specification

The requirements specification was agreed with the project supervisor, Prof. Omer Rana and briefly outlined in the initial plan. These requirements were also reviewed from a commercial perspective by Mr Akmal Hanuk, the chief executive of the Cardiff-based Islamic Banking and Finance Centre. He is considered to be the client at the heart of the system and the primary beneficiary.

The first step of the design process was to arrange a meeting with the parties mentioned above and to have an open ended discussion about what the system I was going to develop was expected to achieve. This was a great opportunity for me to ask questions and gain a deeper insight into their wishes and vision of the problem. It was agreed that I would take the outcome from this discussion and formulate it into a formal system specification, which would be reviewed at another meeting by both parties, and changes could be made to these before they were carried forward into the final version of the system specification. It was also agreed that these requirements were subject to change in the future, which could be accommodated since I was going to implement the system by following a mix of the Waterfall and Agile methodologies.

3.1.1. Mandatory functional requirements specification

These requirements are considered to be a core part of the system and should be given priority in implementation.

1. The system must be able to handle 3 types of users: benefactors, beneficiaries and admin committees;
2. The benefactors must be able to view campaigns proposed by different beneficiaries, split into different categories;
3. The campaigns proposed by the beneficiaries must clearly show their description, the category they belong to, the amount pledged by the benefactors, the target donation amount and the progress percentage towards the donation goal;
4. The benefactors must be able to express their interest in campaigns proposed by potential beneficiaries that have not yet been approved by a member of the admin committee;
5. The beneficiaries must be able propose new campaigns;
6. The benefactors must be able to pledge amounts towards the goal of a campaign;
7. The admin committee members must be able to approve or reject the campaigns proposed by the potential beneficiaries.

3.1.2. Optional functional requirements specification

The following optional specifications were decided as being classed as features which would be make a useful addition to the project, but were not a core part of the system and could be added at a later stage if desired (after the mandatory requirements have been fully implemented and tested).

1. The users should be able to register and authenticate using third-party social services, such as Facebook and Google through the oAuth protocol;
2. The users should be able to provide feedback and comment on the existing campaigns using a dedicated comments system, either natively implemented or through an instance of the Disqus API;
3. The users should be able to pledge amounts towards the goal of a campaign using an integrated payments system, such as PayPal;
4. The system should be able to determine similar campaigns to the selected campaign using an algorithm based on keywords / tags;
5. The users should be able to perform advanced searches of campaigns, based on criteria associated with any field of a campaign, such as the target amount, the date created and / or filter them based on certain tags.

3.2. Business logic

3.2.1. The Users entity

The users of the application were split into *beneficiaries* and *benefactors*. This enables splitting project organisers from people who are willing to help. A beneficiary has the ability to add campaigns and extend them. A committee of admin users decides if a campaign should be allowed to gather funds. Campaigns are split by categories such as *Food*, *Arts*, *Infrastructure*, etc. A system for 'likes' has been implemented to allow us to track the interest in different campaigns and different types of campaigns. This is an element of utmost importance in our system, since it allows the administrators to see what are the campaigns in which the users expressed interest but has not yet been approved.

A user may register as a beneficiary and launch a campaign with an initial duration (e.g. 30 days). Once it is approved by a member of an admin committee, users (*benefactors*) are allowed to donate (*pledge*). A pending campaign can receive *likes*. The *likes* system was implemented in order to enable users (potential benefactors) to express interest towards a campaign and the admin committees to identify the campaigns that receive a certain amount of interest, but which have not yet been approved.

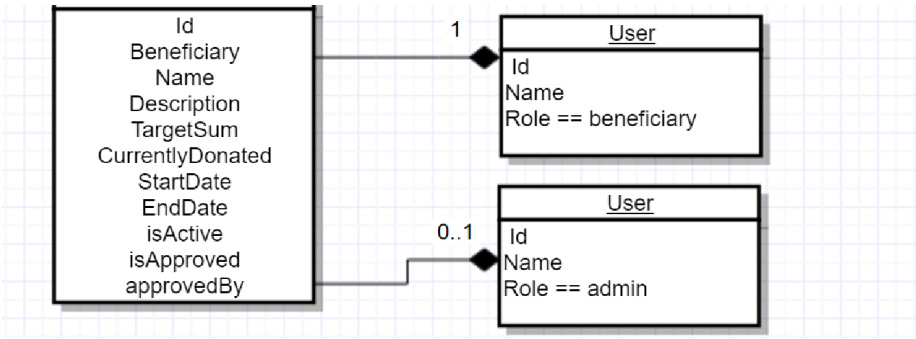


Fig. 3.2.1.1: Entity-relationship diagram between entities: Campaigns, User (benefactor or beneficiary), Admin

3.2.2. The Campaigns, Donations and Likes entities

Each *donation* and each *like* is treated as a transaction. Therefore, an object called *Like* was created, that stores the user that gave the *like*, the *campaign* that was *liked* and also the time when the *like* was given.

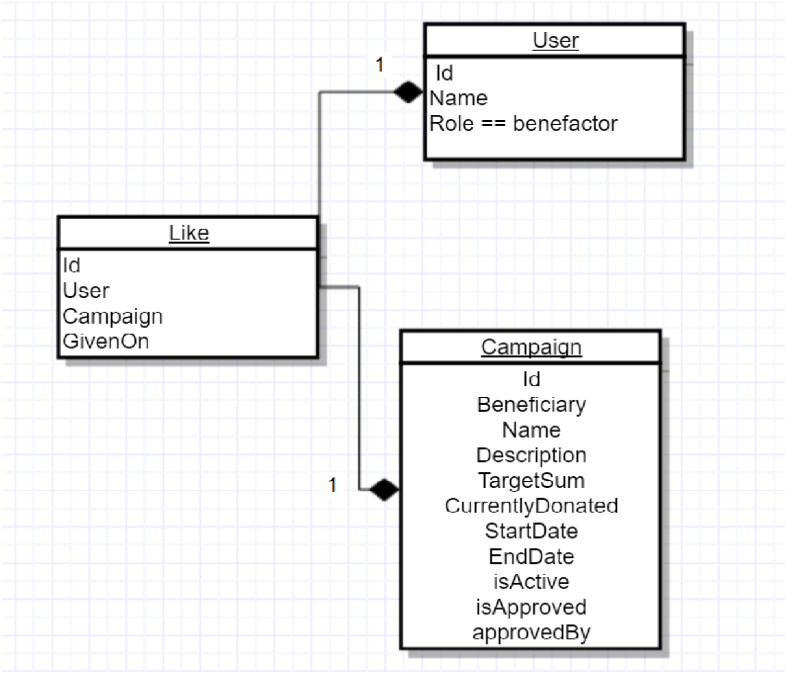


Fig. 3.2.2.1: Entity-relationship diagram between entities: Campaigns, Users and Likes

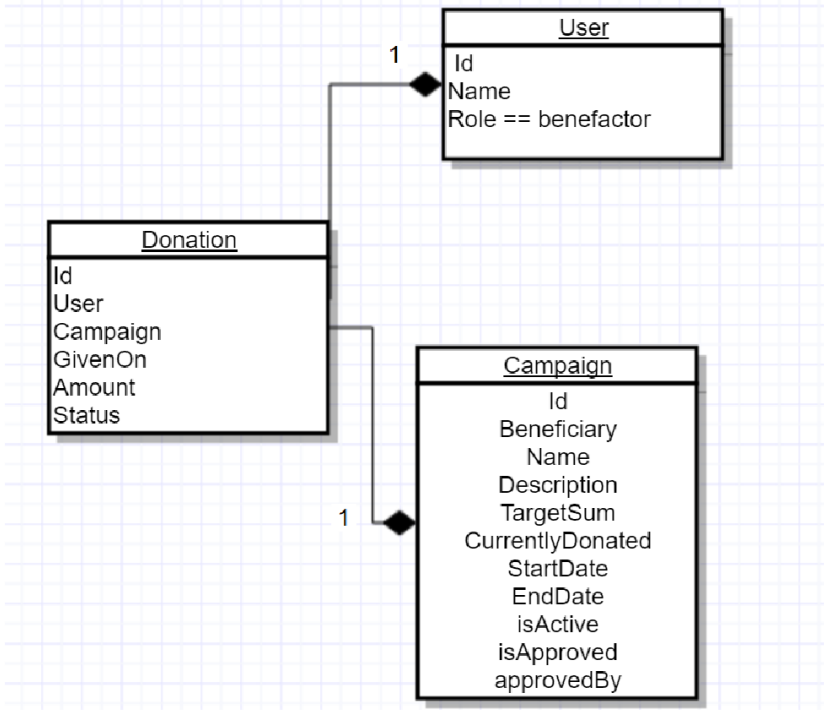


Fig. 3.2.2.2: Entity-relationship diagram between entities: Users, Donations, Campaigns

As it can be observed from the Entity-relationship diagrams above, the relation between *Donation*, *Users* and *Campaigns* is a one-to-one relationship. Therefore, through using these the following 4 classes, a *CRUD (Create-Read-Update-Delete)* model was achieved, which supports our business model. All these four operations can be performed with all the objects in our business model: *Users*, *Campaigns*,

Donations and *Likes*. All the campaigns can also be sorted (both in ascending and descending order) based on any of the fields illustrated in the diagram above (most popularly, by `targetSum`, the pledge target, `currentlyDonated`, the currently pledged amount, `startDate`, `endDate` and `likeCount`).

The UML diagram corresponding to the entities described above and that also corresponds to the back end implementation is presented in Appendix 2.

3.2.3. The process of adding a campaign

An abstraction of the process of adding a campaign is illustrated below:

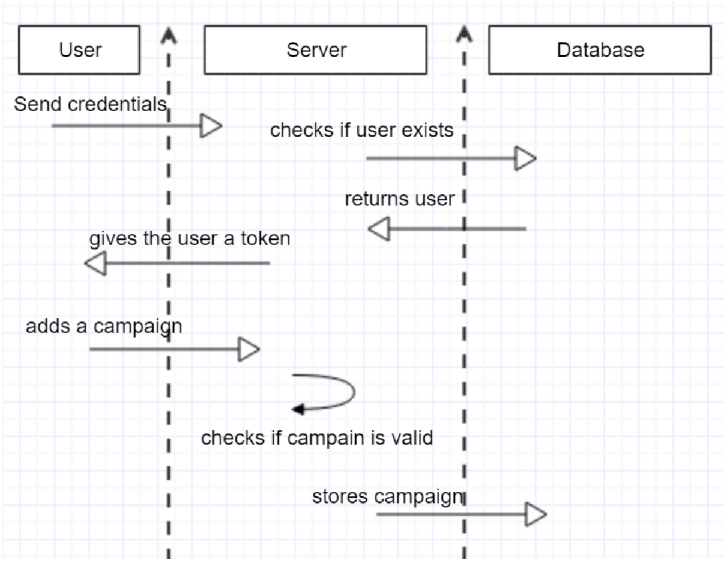


Fig. 3.2.3.1: Abstraction of the process of adding a campaign

This was the initial design. The need for a gallery of images for campaigns and a profile picture for the users was noticed at a later stage and the necessary corresponding fields were subsequently added to the objects.

Another feature that was planned to be implemented was the ability of a user to login with his / her Facebook or Google account (through the OAuth protocol). The enabling mechanism of this feature is illustrated below:

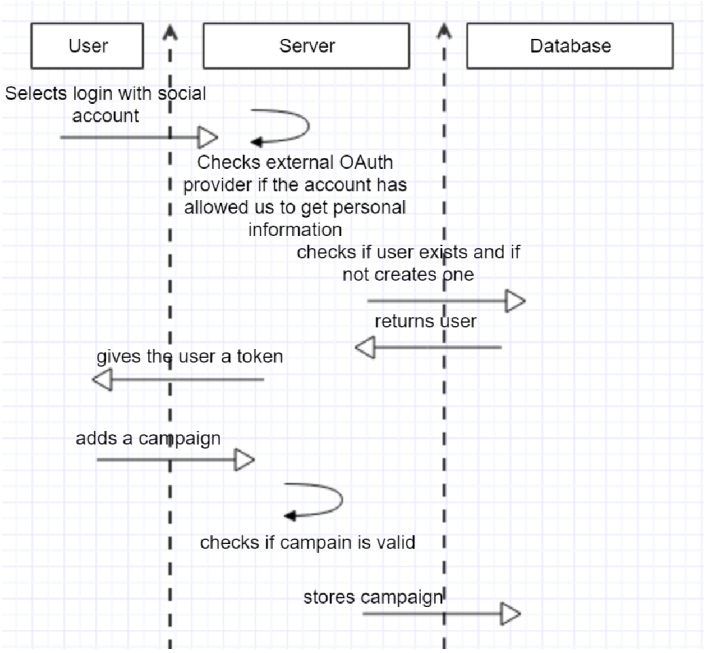


Fig. 3.2.3.2: Abstraction of the OAuth protocol with a Facebook or Google account

3.2.4. Compliance with 3NF

In order to ensure that the data is stored in the most efficient manner possible, both for retrieval and for modification, the database design needed to be compliant with 3NF (the third normal form), matching the business logic at the same time. This way, the system is better understandable, more scalable and more consistent.

MongoDB allows storing an object in the database while keeping its structure close to the structure of the same object described (defined) in the application back end source code. As this is a NoSQL database, it does not have the constraint of needing a table with columns, allowing the storage of the object in its original form, without splitting it into multiple tables.

MongoDB provides the *DBRef* annotation, which allows establishing a relationship between objects. For each relationship that is established, the related collection and ID of the related object are specified. Below there is a snippet from the MongoDB database that describes how *DBRef* is used:

```
{
    "_id" : ObjectId("58dfbf820a96a5493d4a2195"),
    "_class" : "com.xpressstarter.entity.Donation",
    "amount" : 100.0,
    "donatedOn" : ISODate("2017-04-01T14:56:02.966Z"),
    "status" : "OK",
    "user" : {
        "$ref" : "user",
        "$id" : ObjectId("58dfbf820a96a5493d4a2032") // THE REFERENCED USER OBJECT
    },
    "campaign" : {
        "$ref" : "campaign",
        "$id" : ObjectId("58dfbf820a96a5493d4a2194") // THE REFERENCED CAMPAIGN OBJECT
    }
}
```

Although MongoDB is not a type of relational database, the structure of our designed schema complies with the 3NF (third normal form) requirements ([7] E. Codd, Wikipedia, 2017):

- All of the non-prime attributes (*User*, *Campaign*) are handled via the fields of `_id` / references. As a *Campaign* can have multiple *Likes* or *Donations* and so can a *User*, the mapping in the Like and Donation objects is done via the `_id` field. Therefore, our design complies with 1NF (first normal form) as there are no repeating elements or group of elements;
- *Likes* and *Donations* are stored separately and use IDs for *Users* and *Campaigns* that are unique and generated by MongoDB following the UUID specification. Moreover, *Users* and *Campaigns* do not depend on the *Donations* / *Like* objects. Therefore, our design complies with 2NF (second normal form) as it complies with 1NF and there are no partial dependencies on a concatenated key (formed by the `_id` of a *User* and the `_id` of a *Campaign*);
- All the non-key attributes (all the attributes of the objects that are not `_id` - that are not references to other objects or to itself) have no meaning (cannot exist) outside of their object. Therefore, our design complies with the 3NF, as it complies with both 1NF and 2NF and there are no dependencies on the non-key attributes. Below there is a snippet of the *User* object that demonstrates this statement:

```
{
    "_id" : ObjectId("58dfbf800a96a5493d4a1daa"),
    "_class" : "com.xpressstarter.entity.User",
    "firstname" : "Andrei",
    "lastname" : "Hodorog",
    "email" : "andrei@test.com",
    "passwordHash" : "ewjiqoe1oji12310931je10jewqle",
    "wantsToReceiveEmail" : false,
    "memberSince" : ISODate("2017-04-01T14:56:00.725Z"),
    "role" : "ADMIN",
    "totalDonated" : 0.0
}
```

The 3NF allows the database schema to be efficient by not storing duplicate data and ensuring that the integrity is maintained when updating objects. Since `_id` references are used to link the objects, updating an object triggers the update of all the objects that use that reference.

Further insight into the database design can be seen in the entity-relationship diagrams provided in this chapter, in Section 3.2.

3.3. Technical application architecture

This section presents an overview of the technical application architecture, the flow of data in the system and an outlined of the presented components.

3.3.1. General overview

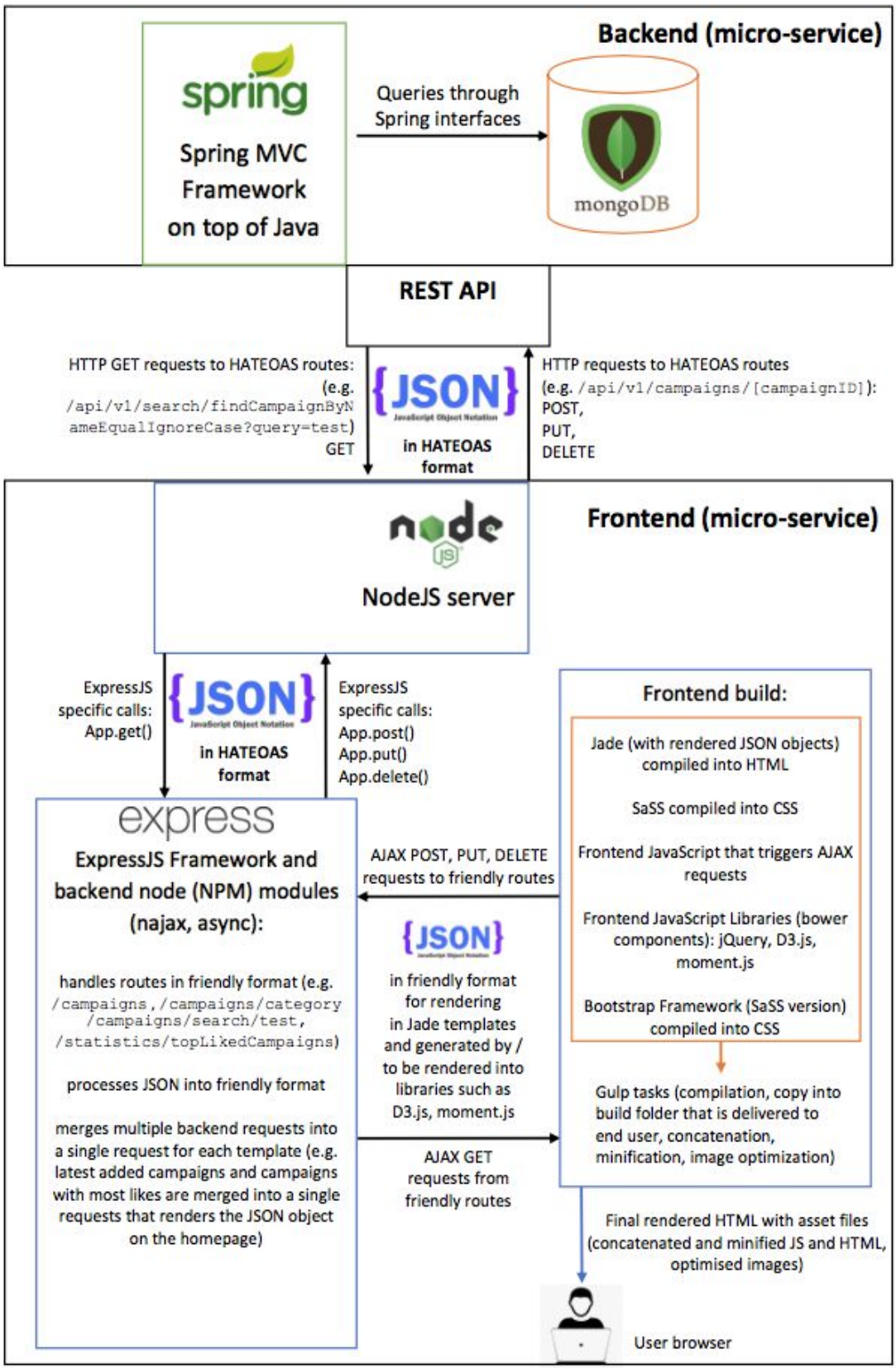


Fig. 3.3.3.1: General application architecture

XpressStarter, like many other modern web applications, has the concerns generically separated into two main components: the back end and the front end. This technique is called Separation of Concerns and involves the split of the application into different layers that have different purposes for the functionality. The back end is providing a way to handle the logic in our platform with background operations that include authentication, fetching the data for the campaign feed, fetching the highest rated campaigns, persisting a new campaign in the database and all the CRUD (Create, Rename, Update, Delete) operations with our entities. On the front end, the main focus is on the user interaction with all these operations: how they are triggered, how the data is rendered and displayed, how the switch from one context to another is made. The diagram above clearly explains the architectural plan for our crowdfunding platform, as well as the processes of communication between each module.

The data flow starts and ends with our back end microservice. Everything is built around the back end. This is where the state of the application is handled, the state of the current campaigns is processed and subsequently persisted to the database. The backend is mainly composed by the Spring Framework that runs on top of Java, querying a the MongoDB database through the Spring Interfaces. The back end implementation and the flow of the data through its components in it is described in detail in Section 4.2 and briefly in Section 3.4.

The data is being forwarded to the front end on top of the REST (Representational State Transfer) Architecture. For security and scalability reasons, the REST architecture is currently considered as the primary option for developing web services, being used by 69% of the web applications ([8] RestCase, 2015). Allowing a client to query a database directly is not only a technological drawback, but it could also lead to potentially destructive security breaches. Through using REST architecture, the control of our application is maintained by defining the application logic. Our clients must follow the imposed predefined rules that are designed when the API is implemented. A full justification for the use of the REST API can be found in Section 3.3.2.2.

The data flow continues on the front end on another web application framework, Express JS, which creates an abstraction layer that intermediates the communication with the back end microservices individually. This abstraction layer has two primary roles:

1) It translates the URLs from the format handled by HATEOAS into a friendly format for the user. For example, a request initiated from the front end to `/campaigns/search/test-campaign` is translated into the HATEOAS specific format to

`/api/v1/search/findCampaignByNameEqualIgnoreCase?query=test-campaign`. This technique helps to encapsulate the logic of the back end and to not expose the HATEOAS URIs to the user. Moreover, this helps the application to expose SEO (Search Engine Optimisation) friendly URLs to the public and facilitates the easy identification of resources. For example, a user who bookmarks an URL ending in `/campaigns/arts` is more likely to remember what page it is associated with rather than a URI ending in `/api/v1/search/findCampaignByCategory?query=arts`

2) It merges multiple requests to the back end microservice into a single request that needs to be executed from the front end. For example, on the home page, there are two requests that need to be made to the back end, one to retrieve the most recently added campaigns and one to retrieve the top campaigns sorted by the number of

likes. When a request is triggered by the client to fetch the content of the home page, the two associated requests needed to fetch the information are executed automatically by the Express JS abstraction layer in the background. The abstraction layer also merges the two JSON objects that are returned by the requests into a single JSON object that is rendered in the template.

The body of the request associated with any CRUD (GET, POST, UPDATE, DELETE) operation that is triggered on the front end is being forwarded to the corresponding microservice. The Express JS uses namespaces and routes that make implementing meaningful, intuitive, and ensures that the API routes are consistent regardless of the back end microservice that is used.

With Express JS, the HTTP responses from our microservices are synchronised and the data that they return is unified and used in the data flow. The data is bound with Jade, the templating engine integrated with Express. The data is presented to the end users as Jade templates which will are instantly converted into markup, alongside Javascript libraries such as D3.JS and Moment.JS when a page is requested.

On the front end, the dynamically generated components are styled with CSS, which is generated by our Sass source files. End users are able to continue the data flow by triggering new actions through AJAX calls that are forwarded to the Express JS instance. Finally, the actions triggered (requests) arrive back to the back end microservices, right where the data flow had initially started.

From our Express Server to the front end, there are a number of steps that are executed. In development mode, there are a number of Node modules added on top of Node for better productivity (nodemon, browsersync, sourcemaps, jshint). All these modules are described in greater detail in Section 4.1.4. In production mode, no debugging overlay is added and all code code (HTML, CSS and Javascript), images, fonts and icons are minified in order to minimise the load time of the page. All the optimisation techniques implemented on the front end are outlined and explained in Section 4.1.5.

3.3.2. Justification for the use of two separate microservices

3.3.2.1. General reasoning

Separation of concerns

This approach allows delegating separate responsibilities between the two microservices, simplifying the code and making the replacement of individual modules easier. If a new technology emerged for the front end and it would be convenient to migrate to it and the back end would not need to be recompiled or refactored.

Code maintainability

This approach improves maintainability as a developer working on one piece of the application does not need to be aware of the functionality of the other piece. This leads to loosely-coupled code. Two separate teams could work on this project (one on the back end and one on the front end) and only communicate via release notes and feature requests as different iterations of the API would come out. The URI for the API is versioned (e.g. `/api/v1`), in order to ensure backwards compatibility.

Scalability

When the platform reaches a significantly large audience, scalability would represent the primary concern. Using microservices allows the scaling of each microservice independently. MongoDB has been selected for this project due to its ability to run in a

sharded cluster configuration, allowing the database to be scalable. Since the REST API uses HTTP, it can be scaled over multiple machines distributing requests between them.

Fault tolerance

Even though there is a high demand for Infrastructure as a Service (IAAS) , no company / provider can guarantee 100% uptime. Even major providers can experience unexpected downtimes, as it occurred last year with Amazon ([9] Time Inc, 2017). If a specific service becomes unavailable, or a specific data center from a region is affected, the uptime of the whole platform will be maintained. As the application is scalable, the downtime of one of the services can be compensated by running multiple instances of each microservice.

Programming environment independence

Even though the back end microservice is written in Java and the front end is mainly written in Javascript, the only part that needs to maintain its consistency for the system to function properly is the format of the REST API. The back end or the front end could be replaced by a different microservice written in a different language without affecting functionality, therefore making our application language agnostic.

3.3.2.2. The use of a REST API

Using an API means that a contract is established between front end and back end, a protocol for communicating between the two. This means that integrating in third party apps could also be made simple, as they only need to implement the API specifications. If the need of a mobile application would arise (Android / iOS / Windows), the back end would not need any modifications.

REST stands for Representational State Transfer and allows web resources manipulation through an uniform and predefined set of stateless operations in a textual representation. It is used in conjunction with the HTTP protocol, using HTTP verbs (GET, POST, PUT, DELETE) to interact with resources.

The popularity of REST APIs has increased steadily within the last 10 years ([8] RestCase, 2015). This is the reason why the majority of the libraries on the market are designed for seamless integration with the REST API.

The most popular format for the data used with a REST API is JSON, which stands for JavaScript Object Notation. This is easily readable for humans and easily parseable for systems, machines and services. Moreover, it is a widely accepted standard by the majority of services in the industry ([10] Webber R., 2013), due to its easy consumption / integration by JavaScript.

Through the use of a REST API, the scalability of the back end in ensured, since it demands the use of the HTTP protocol. A proxy configured on a load balancer could be used at a later stage to split the requests to multiple back end instances. A proof of concept of this technique is outlined in Section 5.3: back end performance and scalability evaluation.

3.4. An overview of the system workflow depending on user type

The diagram below illustrates an overview of the steps that a user of a particular type (benefactor, beneficiary or admin committee) would take while using the systems. The directional arrows indicates the next possible step, as well as the possibility to return to a previous step.

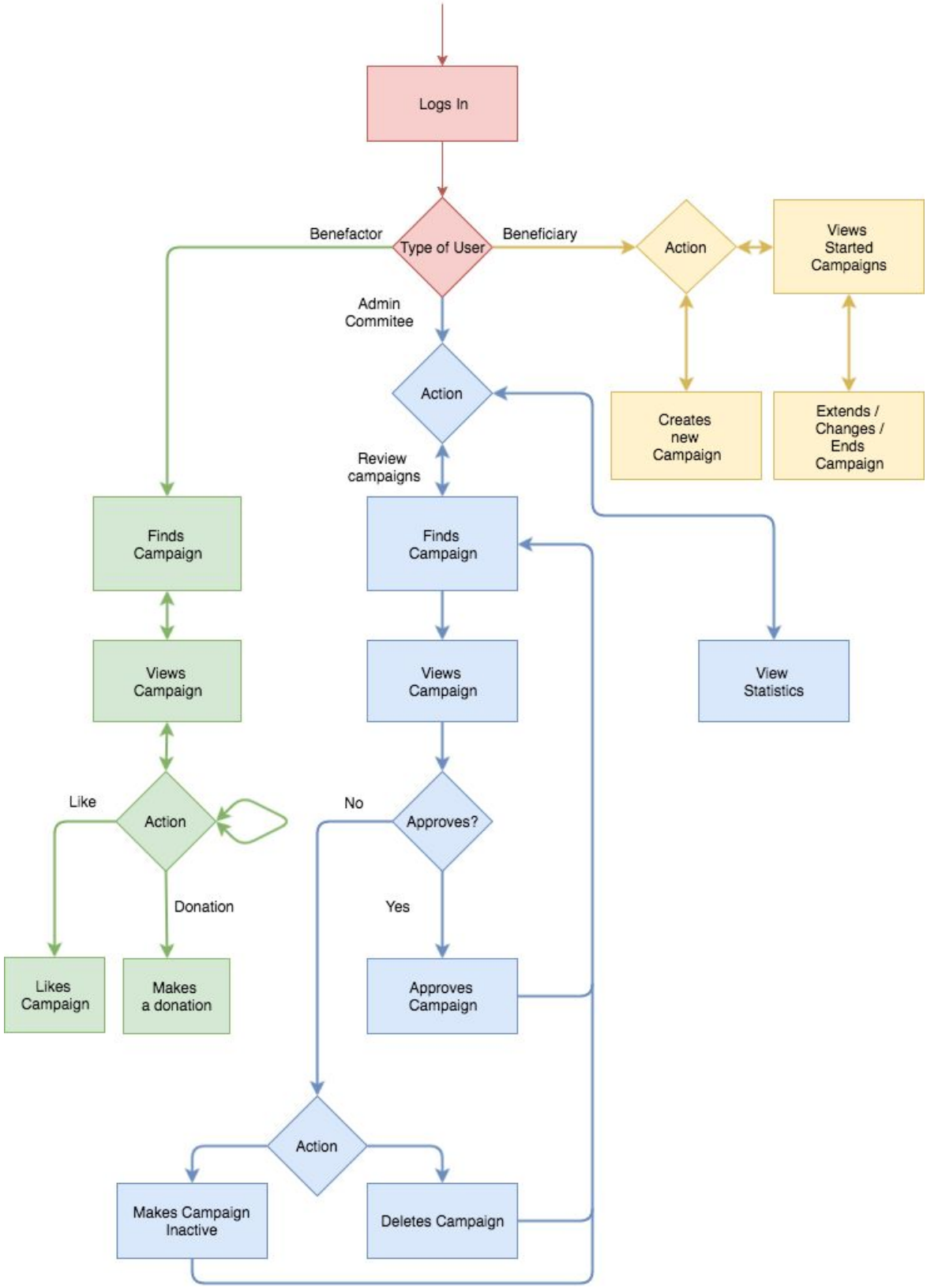


Fig. 3.4.1: An overview of the system workflow

The system contains four main component types: components that fetch data (GET requests), components that add data (POST requests), components that modify data (PUT requests) and components that delete data (DELETE requests).

Fetching data from the system is performed by the following actions outlined in *Fig 3.4.1* above: *Finds campaigns*, *Views campaign*, *Views started campaigns* and *Views statistics*. For these actions, the front end initiates a GET request to the back end to retrieve data. The following actions are performed when a GET request is executed:

1. The front end executes a request that is translated to the following URI in HATEOAS format: `/api/v1/<<entity>>/[id]`, where `<<entity>>` takes the value of the entity name request that can be *Campaigns*, *Users*, *Likes*, *Donations*, *Statistics*;
2. The Controller requests from the Hibernate instance the object of entity type requested (*Campaign*, *Donation*, *User*, *Like*, *Statistics*) with the ID from the URI;
3. The Hibernate instance executes a query that retrieves the data from the database and then deserializes the results into memory objects and then returns them to the Controller;
4. The Controller uses the `ObjectMapper` instance and serializes them into JSON format;
5. The Controller returns the objects back to the client in JSON format.

Adding data to the system is performed by the following actions outlined in *Fig 3.4.1* above: *Likes campaign*, *Makes a donation*, *Creates new campaign*. Modifying data is performed by the following actions outlined in *Fig 3.4.1* above: *Approves a campaign*, *Makes Campaign inactive*, *Extends / Changes / Ends campaign*.

For all the actions that add data, a POST request is sent to the back end and for all the actions that modify data a PUT request is sent to the back end.

The POST requests are executed to URIs taking the format of
`/api/v1/<<entity>>`
The PUT request is executed to URIs taking the format of
`/api/v1/<<entity>>/<<id>>`, because a PUT request can only be executed on a web resource that is identified by the `<<id>>` field and is of `<<entity>>` type.

The following actions are performed when a POST or PUT request is executed:

1. A POST or PUT request is made to the server that encapsulates the entity object in the body of the request;
2. If a PUT request is made to a non-existent URI and returns a response code of `404 (Not found)`;
3. The `ObjectMapper` deserializes the JSON object into a memory object and checks if the entity is valid according to the constraints defined (the size of a string, the donation amount needs to be positive). If this is valid, the data is passed to the controller. Otherwise, a `400 (Bad request)` response code is returned, with the erroneous fields mentioned in the response body.
4. The request is received in the Controller. If a handler exists for the entity, the object is passed to that handler;
5. In the handler, the object is passed to one of the sets of methods with the following annotations that are executed before and after the persistence in the database (described in detail in Section 4.2.3. Critical sections):

    a. for POST: `@beforeCreate` and `@afterCreated`;
    b. for PUT: `@beforeSave` and `@afterSave`.
6. If none of the above methods throw any error (exception), the object is passed to the Hibernate instance that persist the data in the database;
7. The newly created (in the case of a POST request) / modified (in the case of a PUT request) object is returned to the controller;
8. The Controller uses the ObjectMapper instance and serializes it into JSON;
9. The Controller returns the object back to the client in JSON format and returns one of the following response code:
    a. `201 (Created)` code for POST;
    b. `200 (OK)` code for PUT.

Removing data from the system is performed by the following actions outlined in *Fig 3.4.1* above: *Unlikes campaigns* (the reverse of the Likes campaign operation) and *Deletes campaign*.

The following actions are performed when a POST or PUT request is executed:

1. The client executes a DELETE request to an URI taking the format `/api/v1/<<entity>>/<<id>>`
2. If the ID is invalid, `a 404 (Not found)` code is returned;
3. The Controller uses the `ObjectMapper` to deserialize the object into memory;
4. The Controller requests the Hibernate instance to delete the entity with the ID provided;
5. If Hibernate is successful, the Controller returns a `204 (No content)` response.

The operation of retrieving statistics (*View statistics*) runs a query on the database that retrieves and aggregates data based on different criteria:

1. In order to retrieve top X campaigns, a GET request is executed to an URI taking the format of `/api/v1/statistics/getTopCampaigns?type=x&number=y`, where X can take one of the following values:
    a. 1 for top Y campaigns based on activity (number of likes and donations);
    b. 2 for top Y campaigns based on donation count;
    c. 3 for top Y campaigns based on the total pledged amount.
2. In order to retrieve the top X nearly funded campaigns (campaigns that have almost reached their goal), a GET request is executed to an URI taking the format of `/api/v1/statistics/getNearlyFunded?number=X` (where X can take any integer value between 1 and the total number of campaigns present in the system);
3. In order to retrieve the average donation amount per campaign category, a GET request is executed to an URI having the suffix of `/api/v1/statistics/avgDonation`;
4. In order to retrieve the top X donating users (the top of users that have pledged the highest amounts), a GET request is executed to an URI having the suffix of `/api/v1/statistics/topdonatingusers?number=X` (where X can take any integer value between 1 and the total number of users present in the system).

When adding and modifying data, some intermediary actions need to take place before and after persisting the data to the system. These actions are defined in Handlers for each entity and specify which methods should be called before the persisting happens and which methods should be called after persisting takes place.

This section outlines a brief description of these handlers, that are described in more detail in Section 4.2. Backend implementation technical specification.

The following diagram presents a general overview of the handlers that are included in the system:

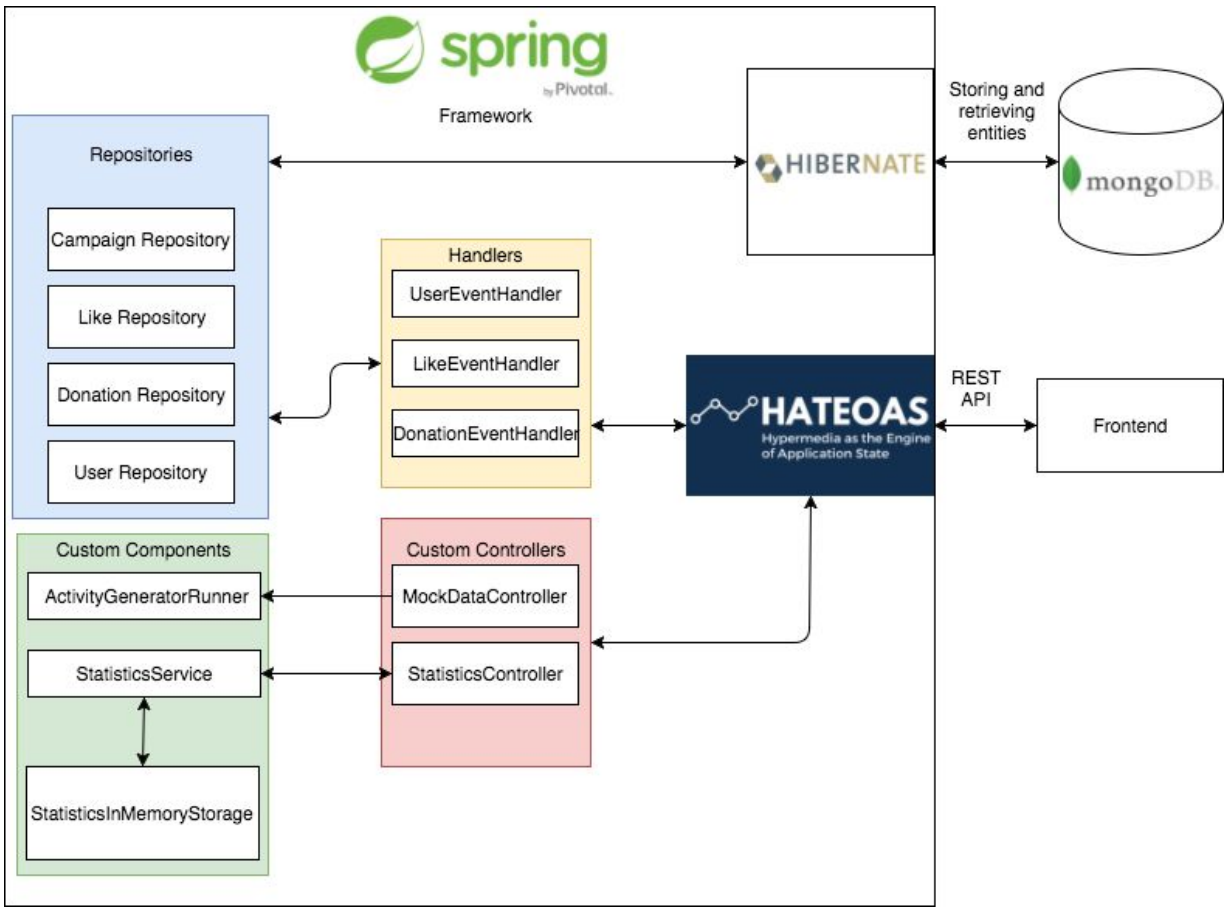


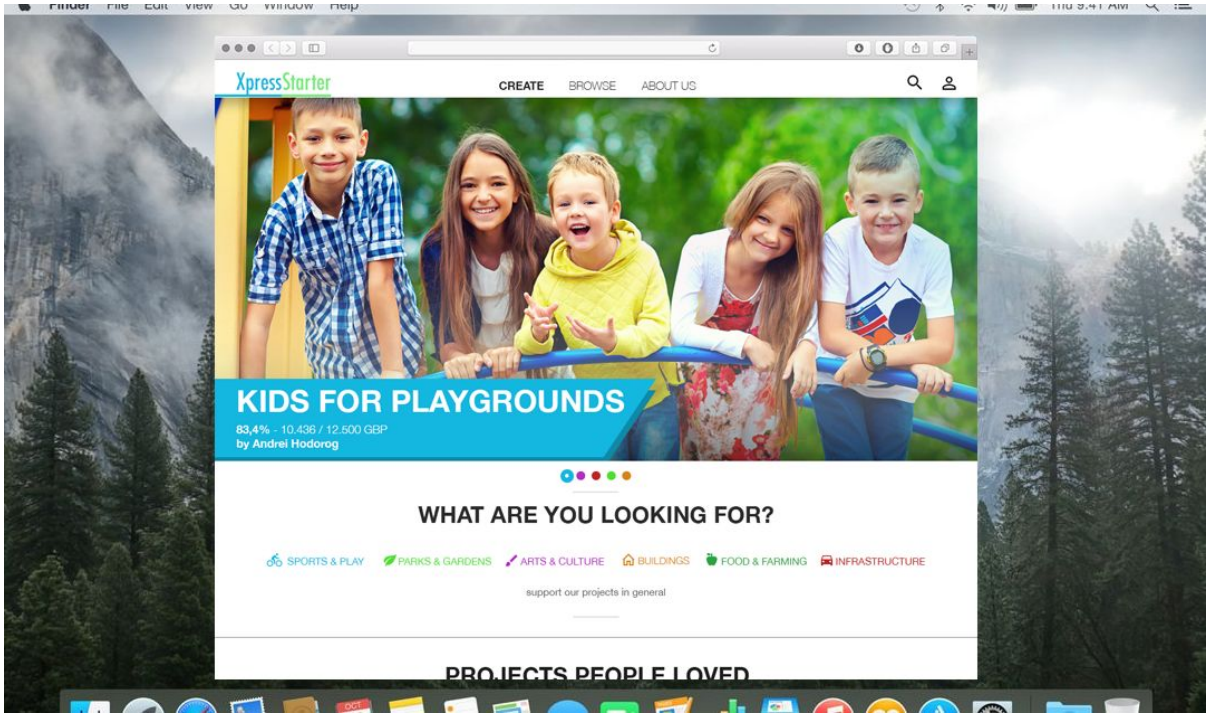
Fig. 3.4.2: An overview of the back end system architecture

Donation handler:
When a donation is triggered (a POST request is created), the time of donation is initialised (`donatedOn`) that synchronises with the time of the server. If a request is forged (the time from the body of the POST request is changed to the past or the future), the time overwrites with the current date of the server. This happens before the entity is persisted to the database. After the *Donation* entity is persisted, the total amount donated for campaign and user are recomputed.

Like handler:
When a Like is posted, before being persisted to the database, the value of the field `givenOn` is set with the server time and then the system checks if the like have not been already given (posted). If the request proves to be correct, it is persisted to the database. If not, an exception is thrown. After the *Like* entity is persisted, `likeCount` is recomputed for the respective campaign.

User handler:
Before being persisted to the database, the `memberSince` field is overwritten with the server time and then the email address is checked if it already exists in the system. If there is, an exception is thrown. Otherwise, the entity is persisted.

3.5. Graphical and User Interface Design



3.5.1. Design introduction

The foundation for the design process was set by deciding on the key quality criteria for a high quality user experience. The first step was researching a range of fundraising websites (most notably: JustGiving, KickStarter, GoFundMe) and analysing the strengths and weaknesses of their design choices. The focus of interaction on such a crowdfunding website was deemed to be the main page, the search and category selection tools and the project presentation pages. Therefore, my analysis focused on these three key destinations in the user's online interaction.

As a first step, the main page of the leading fundraising websites was analysed. A primary discovery was that most platforms use their header space to promote their business itself, as well as to encourage visitors to promote their own project. The main body usually contains the projects overview as well as an overview of the overall project categories. Normally campaigns would be presented in different categories on the page, according to popularity or other filtering criteria. Other areas of the page would traditionally be used as advertising space for uploading a new project.

In terms of individual project overview, selected projects are presented by an image and title, typically next to an overview of the fundraiser's status (financial goal, supporters as well as remaining time). Underneath the title, the potential benefactor can read a description, created by the project creator, as well as a table of rewards for supporting said project. Most fundraising platforms also make donation- and sharing-buttons permanently visible while scrolling, so as to encourage visitors to

engage with the projects.

Project can usually be accessed either via the search function or by exploring the different project categories. As a norm, filtering options are expected on these platforms. Some websites (such as kickstarter.com) would still separate the projects by popularity, recommendations and such.

The next step was researching common design patterns in the styles used. This has proven to be more difficult than expected, as most websites retain a large part of their appeal purely based on their appearance. Kickstarter for example uses a modern style across their website, underlined by sharp corners, narrow lines, strong colours and a modern font style. By contrast, GoFundMe uses a lot of rounded elements and gradients less pronounced colours, reminiscent of the skeumorphic design era ([11] Tony Thomas, 2012) The main common design element found in most fundraising websites was the use of “card”-styled boxes for the projects, rather than a vertical list or other display method.

Lastly, the implemented tools were examined. Most websites have proven to use the same functionality: user created content, separating interesting content from other (popular and featured projects), enabling social promoting (sharing on Facebook and Twitter), a rewards-system and community features (such as comments and likes).

3.5.2. Overall structure

The next step was deciding which ideas to be integrated into my own project, as well as deciding which elements would be structured differently, all while fulfilling the technical requirements for the project. Following is a description focused on those main parts of the platform.

3.5.2.1. First page

Fig. 3.5.2.1.1: Menu and featured projects

The final decision was to use the header space for promoting recommended projects rather than promoting the idea of inserting a new project. Most users of fundraising

platforms are aware of the possibility of promoting an own project (the button on the menu “Create” would still highlight this option), but rather choose to support the projects. The reasoning behind this thought is that in order to have a functioning platform there have to be more benefactors than creators. By placing the most interesting projects right at the top of the page potential supporters would directly have access to the most interesting campaigns, therefore streamlining the fundraising process.
Above the featured projects the user finds the most relevant navigation tools: The logo acting as a direct link to the first page, the creation link, browse option, an about us section as well as search and login/register-tools. The menu keeps in line with the rest of the page by using a white body which floats above the featured content (implementation of a drop shadow).

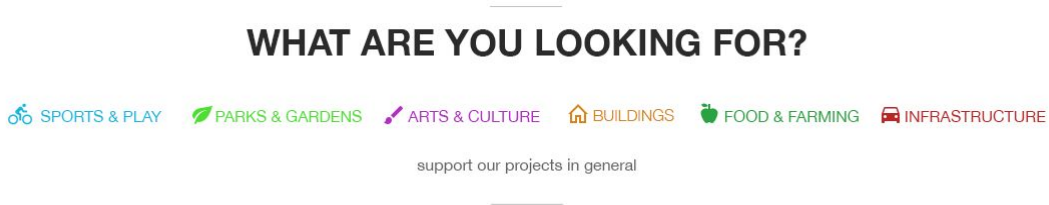


Fig. 3.5.2.1.2: Category selection

Streamlining was the main focus of placing the category selection underneath the highlighted projects. A user that simply browses the website is more likely to be scrolling down than a user who already knows what he wants to finance. This is also the reason for not hiding the category selection behind a menu button or on the side.

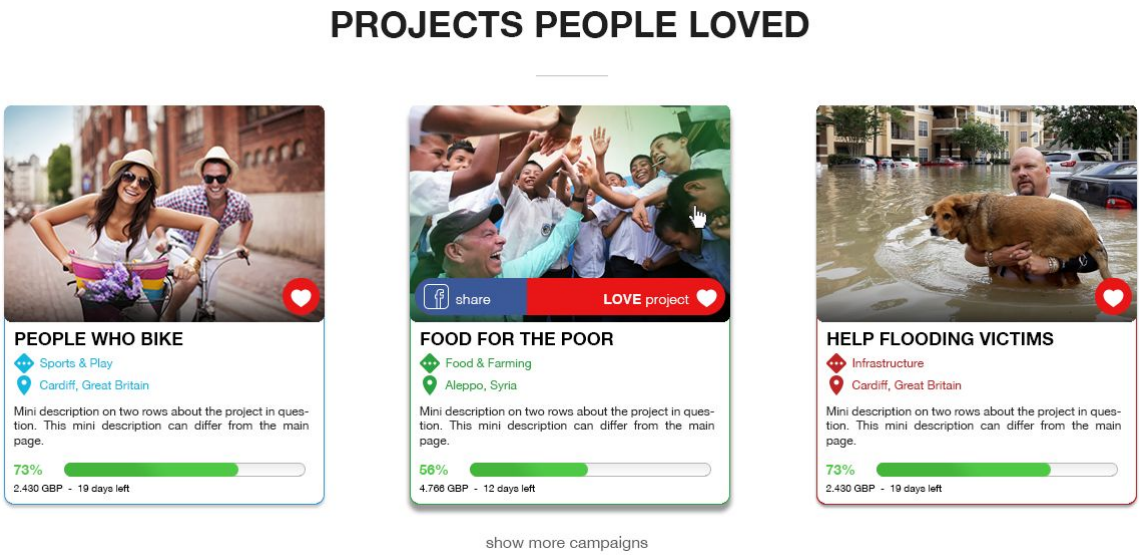


Fig. 3.5.2.1.3: Main space of the first page with campaign-cards

The main space of the first page is inspired by other platforms, having a separated popular projects and new projects section. The platform offers therefore three different types of projects on the first page (incl. the recommended-section in the header). Campaigns are displayed by using playing-cards looking boxes, giving an added element of gamification that is likely to encourage user engagement. Depending on the funding stage of the campaign card displayed, successful projects will incorporate a "Success" indicator and failed ones will have greyed out images. This style is a clear indication for a potential benefactor regarding which projects can still be supported and which do not, while still retaining a clean aesthetic.

Fig. 3.5.2.1.4: Section "top contributors"

An aspect usually overlooked by other fundraising platforms is the focus on the project creators. Given that the project focuses on civic campaigns, the people that stand behind those ideas were highlighted, as they are likely to have an impact on lives and communities across the world. Underneath the main body, a new section dedicated to those people was created. The persons displayed are selected by their overall contribution to the community (campaigns created and money donated). This is likely to inspire trust to platform users, by offering a more personal touch to the crowdfunding community.

3.5.2.2. Selected project

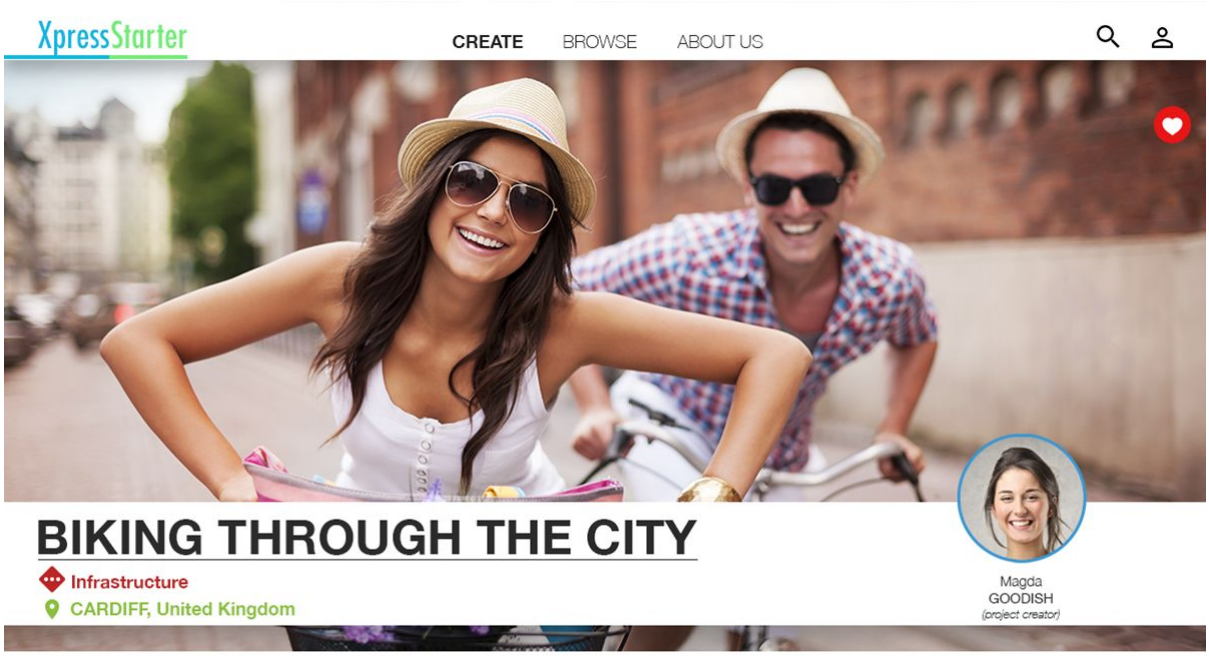


Fig. 3.5.2.2.1: Menu and header for a selected project

The page for a selected project is, in terms of body structure, similar to other platforms for crowdfunding. The header part is showing a representative image of the campaign, partly covered by a content box (title, category, location and creator). Any project selected is thereby instantly unique and recognisable.

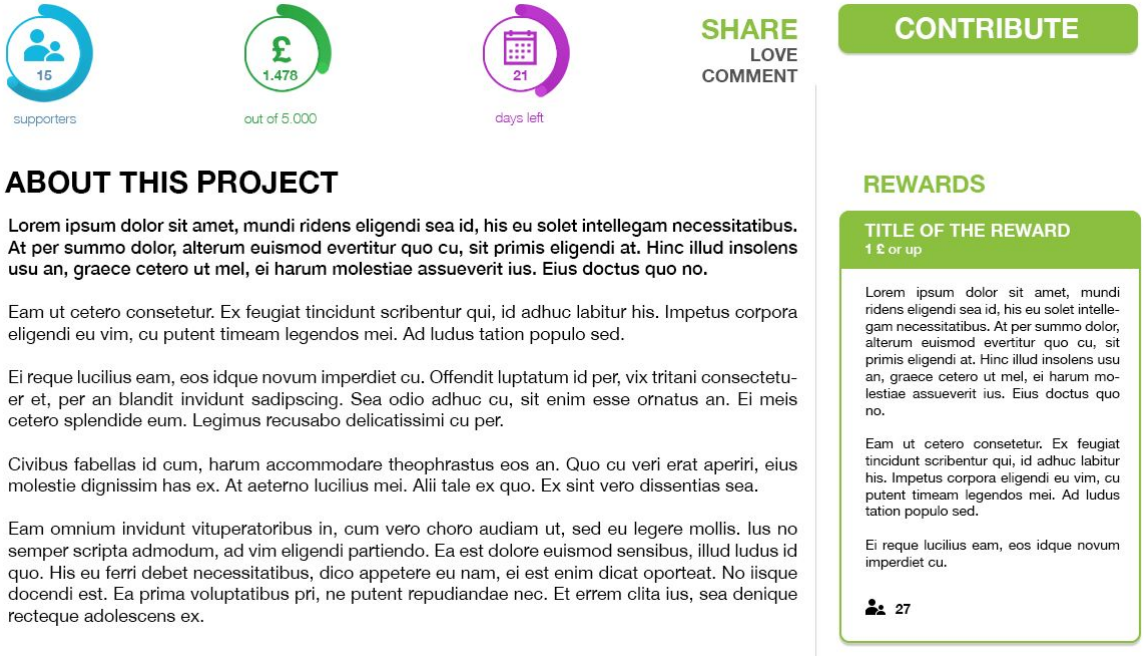


Fig. 3.5.2.2.2: Content in a selected project

The main body of the project page is split in three sections: main data, description with rewards and recommended projects. The main data displays the supporters, the funding status, time left as well as interaction options (share, love and comment) next to a highlighted contribution-button. Placing these elements directly under the image aids the idea of streamlining the experience. The description of the project contains three font types: title, regular as well as a stylized rendering of the creators' names (highlighting, again, the importance of the people behind the campaigns). The rewards section contains boxes in similar style to the campaign boxes. Although room for individuality was attempted to be found, this structure, inspired by other popular fundraising platforms was found to fulfill the requirements set by modern web design.

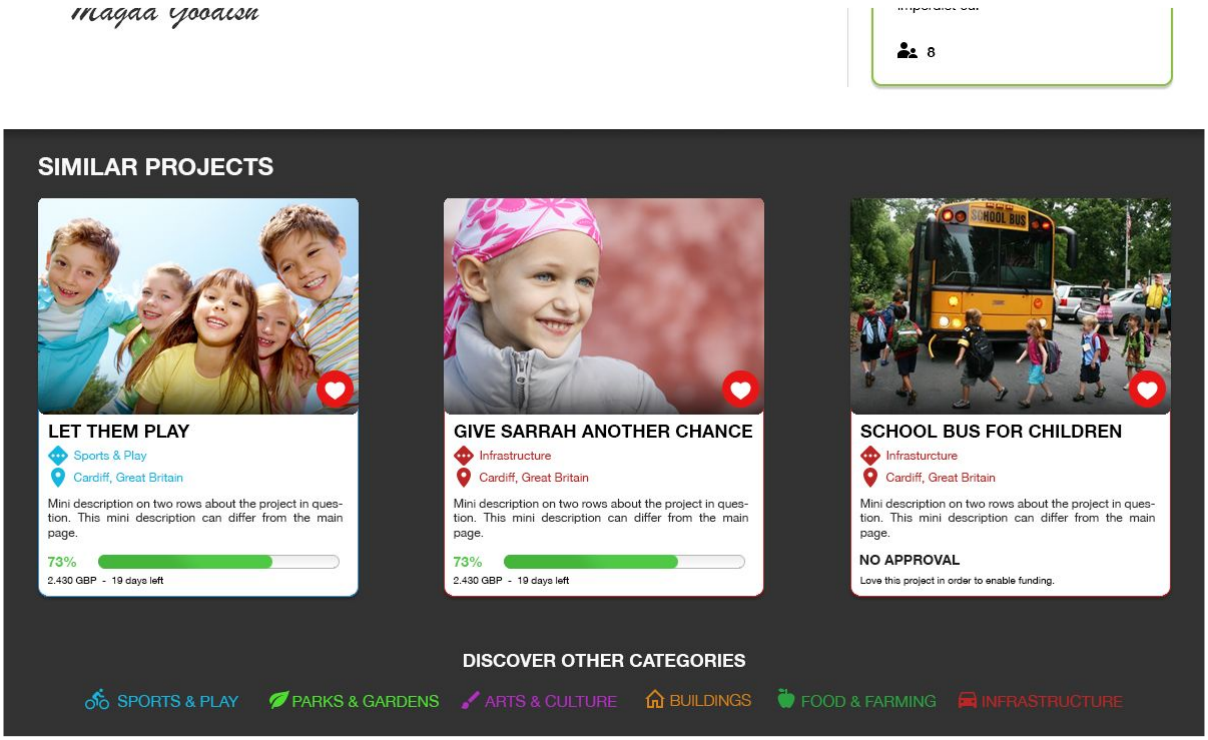


Fig. 3.5.2.2.3: Similar projects being displayed after a project presentation

Similar projects are being displayed on a dark background by using playing-card-boxes identical to the ones on the first page. Underneath, the user has direct access to the categories section, thereby aiding website retention times.

3.5.2.3. Category selection and search

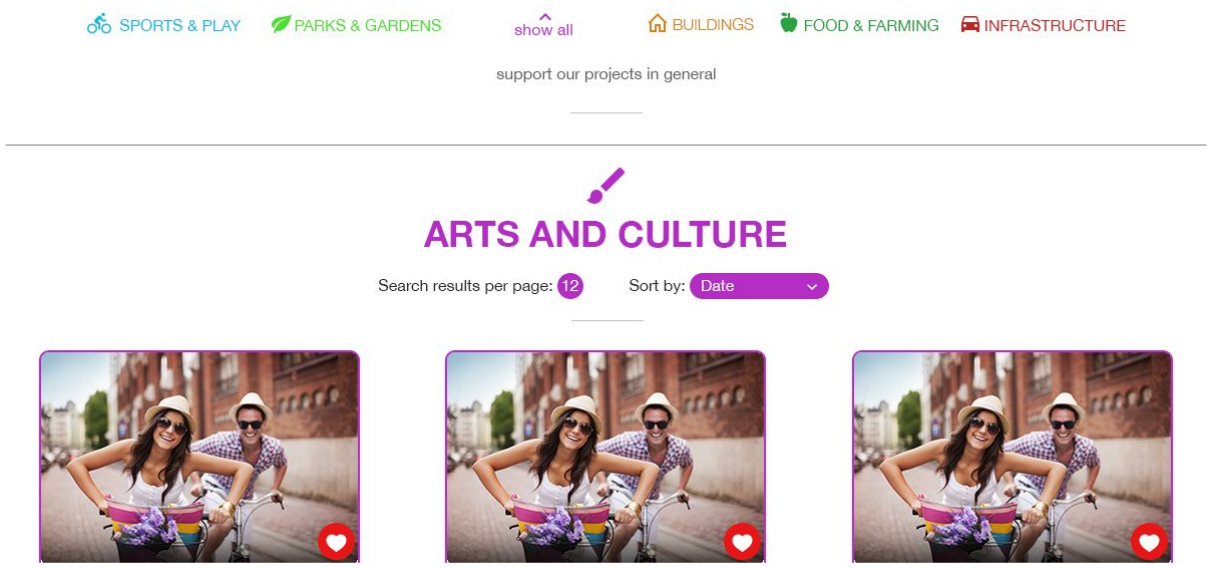


Fig. 3.5.2.3.8: A selected category being displayed

By selecting a category or using the search tool the website displays relevant results. The category selection and search results still highlight featured projects that met the selected criteria. Underneath, the category selector is still displayed when selecting a category, through changing the selected category into a “show all” button. When using a search tool the category selector is hidden as it would visually disrupt the search action and the results. Following the category selector (or, in case of the search page the search term indicator) the user can select the number of campaigns being displayed per page as well as sort them by different criteria (such as popularity, age or relevance when using the search tool). Underneath those tools results are being displayed in play-card-boxes, introduced by the title of the selection in the corresponding colour. The page ends by displaying a page navigation tool.
This design is still reminiscent of other websites as there is no room for individuality in displaying search results. The platform, however, still keeps its individuality by displaying featured projects and categories.

3.5.3. Design patterns, colour, typography and animations

After preparing the basic structure of the website the design patterns were developed and adapted the typography to the functionality of the website. Research also went into choosing the use of colours and integrating modern animations.

3.5.3.1. Design patterns

The development of the design followed multiple steps.

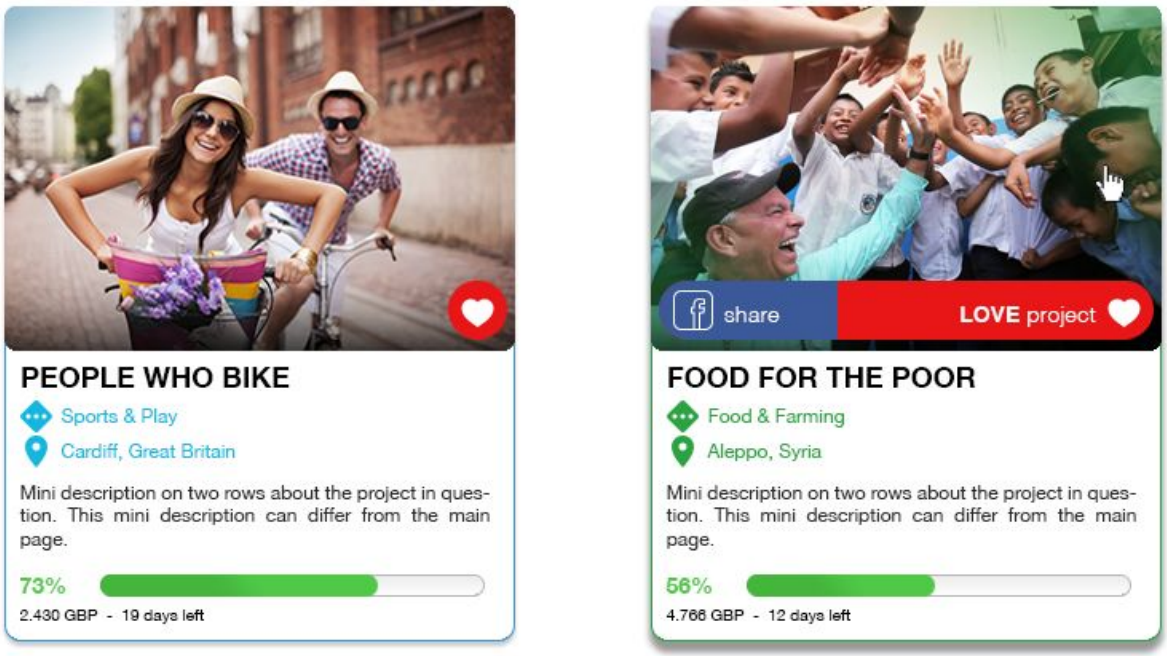


Fig. 3.5.3.1.1: Campaign-boxes (without and with mouseover effect)

Firstly, a decision was made on the basic shapes of the elements. The first decision taken was to use rounded elements rather than sharp ones for a more approachable and vibrant feel ([12] Keith Bryant, 2012). Some of the main examples for this style are the campaign-boxes. Choosing a rounded look, those playing-card inspired boxes are much more pronounced. This is further aided using a soft drop shadow, which gives the campaigns the impression of “floating” above the main body. This effect not only aids the user in recognising the intractability of the elements (clicking on them gets you to the project), but much more separates the campaigns clearly from the rest of the page, underlining their essential status as the base of the fundraising platform.

Fig. 3.5.3.1.2: Data for a selected project

Using rounded elements also creates the impression of “play” and “fun”, which can be also exemplified in the page for a selected project. Underneath the presentation image there are three main status indicators: supporters, financial and time left. Those perfectly round elements are being surrounded by initially incomplete rings which, depending on the evolution of the project, fill up. This element, inspired by the success of the fitness-rings used in the UI of the Apple Watch, not only give a clear and fast indication of the project (it is faster to see the half ring filled rather than that a £15.000 project needs a further £7.500 to be fully funded) but gives the website an own and, in the fundraising-market, unique approach. Other examples of rounded objects are: the love-icon in the campaign-card, the images for the users, buttons (contribute and mouse over).

Fig. 3.5.3.1.3: Description box for a featured project.

The decision was made, however, to also use some strategically placed sharp objects. On the main page, the description for the recommended projects are placed in a razor cut box in order to attract more attention by being a clear separation from the background image. The main body elements are also clearly separated: from the featured image to the category selector down to the different types of projects there is always a visual “cut” (no shadow underneath the featured image, a full width line between other elements). This style is also used when the user accesses a campaign, as underneath the description and rewards section the box for other similar projects is being highlighted with a darkened background, a visual “break” from the usual white body style.

Another aspect of the design pattern is the use of full width elements (e.g. featured projects, category selection). By doing so, it enables the website to develop over time following a modular architecture, as new ideas or functionality can be placed between other already existing elements, eliminating the need for resourceful redesigns.

3.5.3.2. Colours

Colours have also been a key subject in the development of this platform. This included researching the colour trends for this year as well as deciding on the use of mixed colours (such as in gradients). 2015 and 2016 were dominated by the use of vibrant, almost neon looking, colours, most notably the major redesign of TheVerge ([13] Carrie Cousins, 2016). This trend, aided by Google’s Material Design release, could however end in 2017, replaced by more natural or neutral tones ([14] Carrie Cousins, 2017). As this would signify a great departure from the bold colour evolution of the last two years, this new evolution is doubtful, especially considering that laptop and mobile phone screens are capable of displaying an ever increasing amount of colours (e.g. the use of P3-capable screens on the newest Apple products) that encourages the use of vibrant colours. In the context of the fundraising world, especially in the environment, a need for bold and bright colour is considered necessary, as those not only sustain the feeling of “wellbeing”, but can also be used for greatly separating different aspects of the website. Following this train of thought, a mostly vibrant colour pallet for the different categories of uploaded campaigns (e.g. Sports & Play is baby blue, Parks & Gardens neon green, etc.). This colour identity is transferred to the titles of the featured projects, the category selector, the border of the campaigns and in the category selection page. To support the use of vibrant colours,

a decision was made to retain the background in plain white. This decision also enables the use of other soft elements, such as drop shadows. All in all, by choosing such a colour style, the platform has a clean, rounded and friendly look, an image is strongly believed to aid the caritative aspect of the website.

3.5.3.3. Typography

Fig. 3.5.3.3.1: Assortment of type-elements used on the platform

Typography usually revolves around two question: the right font type (e.g. Serif or sans serif) and the right colour. Choosing the font type to Helvetica Neue was a comparatively easy decision: Sans Serif fonts can be read much easier on low dpi – displays ([15] Stacey Kole, 2013) and Helvetica Neue is a generally well regarded modern font that is recognisable and therefore familiar to most users.
The general colour of the font is dark grey (*#1f1f1f*), as it still contrasts well on the perfect white body while still being easy on the eye while reading long texts, such as the descriptions for the campaigns. Some text is however in colour (e.g. The category selection for maintaining the identity-structure of the campaigns) or in light grey (as those options are not generally as important).
Another aspect is the use of bold and all capitals in a number of elements (such as the campaign titles or the different content types (“What are you looking for?”; “Projects people loves”). This approach was followed as bold text immediately highlights important information to the user and therefore makes the navigation process more streamlined and easier, all while giving the website a modern look. Furthermore, some elements do not contain bold but retain all capitals. Those elements (such as the category titles and the name of contributors) are generally more important than plain text, but not as important as the bold elements, which is why the medium-highlighted style was used.
The project also extensively uses different font sizes to highlight or hide certain text, depending on their importance for the user experience.

3.5.3.4. Animation

Animations were also an important part of the development process. I chose to generally enable smaller and less intrusive animations in line with modern web design developments ([16] Karol K., 2015). By doing so, the user isn’t distracted by the animations, still being able to focus on the important content. Meanwhile the experience is enhanced by giving the website the impression of a “living and breathing” entity. Examples for using experience enhancing effects are:

- featured campaigns are displayed using a parallax effect while scrolling;
- the drop shadow of selected campaign cards increases on mouse over to sustain the impression of “floating”;
- when opening a project the information rings fill up to their current status;
- mouse over on the heart-icon on a campaign card reveals sharing options.

3.6. Project management, versioning and issue tracking

In order to adjust the code more efficiently, a versioning system was used (Git). This allowed for the tracking of all the changes made to the codebase during the development of the project and for reverting to previous versions of the code in the event of failures hindering the functionality of the application or the development process. All the important changes to the codebase, such as those marking the achievement of a milestone or the finishing of an important feature being implemented or an important issue being fixed were marked by code commits.

Furthermore, separate branches could be used for the development of certain features, so that they can be developed independently from the codebase of other features and integrated into the main codebase (the `master` branch) at a later stage. When the application will expand, this process could also allow other developers to review the code pushed before merging, through the use of *pull requests*.

Git was used in conjunction with Github (external cloud service providing the hosting of Git repositories), which allowed issue tracking and prioritisation through the use of coloured labels. Github could also facilitate easy collaboration with other developers in the future.

3.7. Specification and Design conclusions

XpressStarter was designed to be a crowdfunding web platform where users (benefactors and beneficiaries) are able to raise money for various civic projects. The technologies used in creating the platform needed to enable it to be scalable and future-proof. Since the projected user base would be high, the data flow, the application architecture and the database design need to be efficient in processing and storing the data.

Users will be split between benefactors, beneficiaries and admin committees. Beneficiaries will have the ability to add (propose) campaigns and manage them once they are approved. Benefactors will have the ability to show interest through the system of *Likes* in the campaigns and pledge amounts towards the goal of that campaign through *Donations*. Admin committees will have the responsibility to verify a campaign and approve it once it meets the eligibility criteria.

Users (*benefactors*) will have the ability to express interest both in the campaigns that have been approved and the campaigns that have been proposed but are not active. If a campaign that is not considered relevant enough by a member of the admin committee to approve it, the admin committee might decide to approve it based on the number of *Likes*.

A design choice was made to split the back end and the front end in order to allow scalability, fault tolerance and programming environment independence. The HTTP was selected as the protocol for communication between these two microsystems. The REST architecture enables the integration with third party systems. The JSON format for the exchanged data allows easy consumption and integration with JavaScript. Using the REST API in conjunction with the HTTP protocol enables the web platform to be scalable. Specifically, the back end could be scaled to meet future

needs of third party systems (such as crowd funding review websites that aggregate data) and other activity / request generating systems (another version of the front end architecture using different technologies or a mobile phone application).

MongoDB was selected to be used as the database system, since it allows the storage of the object using a structure of the object close to the one described in the code base. It can also be used in the clustering use case, meeting the requirements for scalability and future proofing.

The database system was structured to be compliant with 3NF (the third normal form), in order to make the retrieval and storage of data as efficient as possible and to make sure that data integrity is preserved.

The front end was designed in order to meet the most modern design patterns, be adaptable to mobile devices and attract potential investors to donate in the project showcased on the web platform.

Versioning and issue tracking was used in order to handle the complexity of the project given by the use of so many technologies that had to be integrated together. Given the design choice of using 2 separate, independent microservices, separate repositories were created for the front end and the back end respectively.

4. Implementation

The implementation chapter below provides a finer level of details regarding the system architecture and design, down to the code level. Since the decision was made to split the system into two separate microservices (decision previously justified in Section 3.3.2), the implementation chapter was split into two main subchapters: 4.1 for the technical specifications of the front end and 4.2 for the technical specifications of the back end. Each of these subchapters is further split into a justification of the technologies used, issues and difficulties encountered during the implementation process and a description of the critical sections of each microservice.

4.1. Front end implementation technical specifications

This section starts with a general overview of the client side application architecture and the data flow in the front end. A detailed description of the components follow: the core technologies used, the client side dependencies and the server side dependencies. The section ends with the presentation of the optimisation techniques applied and the issues encountered during the process of front end development and how they were overcome.

4.1.1. A general description of the technologies used and structure of the front end

A general overview of the Frontend architecture and the technologies used is outlined in the figure below:

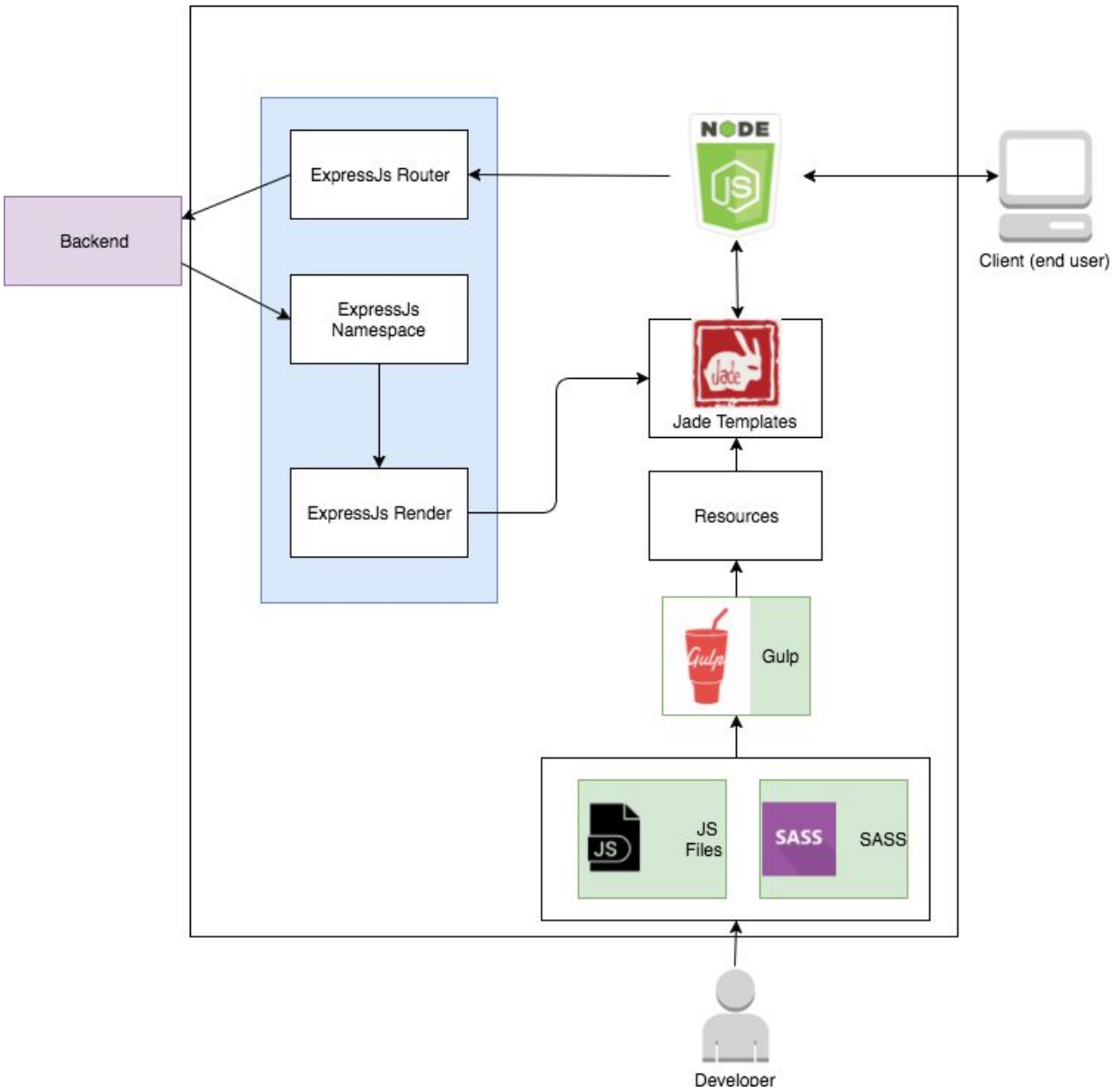


Fig. 4.1.1: A general overview of the front end architecture

The flow of data in the front end is started by the client when a request is executed to render a specific page (e.g. the Arts category page, that is associated with the URI in

user-friendly format ending in `/campaigns/arts`). The request is intercepted by the NodeJS server, that forwards it to the Express Router (which is part of the Express JS web application framework that runs on top of NodeJS).

The Express Router translates the API route into a format that can be handled by HATEOAS (in our example, `/campaigns/search/findCampaignByCategory?query=arts`) then forwards the request to the translated route in the back end.

When the back end replies with a response in JSON format, it is intercepted by the Express JS namespace associated with the set of views from which the request was initiated (in our example, the *Campaigns* namespace). This processes the JSON response in a format that is suitable for rendering on the front end, through stripping the unnecessary overhead caused by the HATEOAS format and matching it to the custom object structure defined in the template. The processed JSON object is then passed to the renderer that renders it into the appropriate Jade template.

The Jade template is instantly rendered into HTML on the server side (by NodeJS) and delivered back to the client in a format readable in the browser used by the end user. The rendered template is linked to the static resources files (static images, CSS stylesheets, custom Javascript files, external Javascript libraries) that are bundled together from the source (`src`) folder into the build folder (`build`) through Gulp. This has custom tasks defined that concatenate and minify all the static resources files, compiling the Sass code into CSS at the same time. In the production environment, this process is done only once when the application is deployed. In the development environment, this process is done every time one of the source files changes.

4.1.1.1. NodeJS and Javascript

NodeJS and Javascript are currently two of the most popular runtimes that can be used to create dynamic and robust web applications in conjunction with frameworks such as HAPI or Express JS ([17] NewStack, 2016). The interesting aspect is related to the fact that it is a relatively new concept, compared with classic frameworks written in C#, Java, Ruby or Python. Javascript, from ECMA family of scripting languages, was intended initially to run inside browsers (on the client side).

In the early days of web programming, running server-side Javascript was considered a naive approach. However, it succeeded very quickly to be used in large technology companies in production environments, producing impressive results ([18] RisingStack, 2016).

NodeJS is not only extremely popular, but it is also supported by the largest open source community. At this moment, there are at least 450,000 packages (libraries) available for Node on The Node Package Manager ([19] NPM Inc., 2017).

The primarily goal of NodeJS is to empower developers to write highly scalable web applications in a very simple way. In order to achieve that, Node is distinguished from traditional servers (like Apache) by achieving concurrency in a reactive manner. By that means, everything should be non blocking.

The original NodeJS author, explains the difference that NodeJS brings from the traditional threading model ([20] Ryan Dahl, 2012): the traditional web server model involves creating new threads for each incoming request. The downside of this approach is that after a considerably large number of concurrent connections, the response time drops significantly. Context switching between threads are very expensive operations, there is no surprise that tasks will be significantly slower to completed as the number of concurrent connections increases.

Another main problem is thread blocking ([21] NodeJS Foundation, 2017). Database queries are widely used in developing any kind of web application. When a database query is run, in the traditional model, the thread blocks until a response is given. This is a huge waste of resources. There are a number of multithreading paradigms in which performance can be increased, but they are significantly difficult to work with, and productivity drops drastically.

In a reactive approach, not only the query to the database is passed, but also a callback function, a function that will be called automatically when the blocking operation is completed. The internal event loop in NodeJS addresses this aspect.

The principle is simple: there is a main thread where all the operations are executed in a non-blocking manner and a thread pool which executes the asynchronous callbacks in an efficient manner. Falling under this category, tasks like file I/O and querying an external service are done in a reactive manner.

Below there are two code snippets that illustrate the difference between the blocking and non-blocking thread execution.

Traditional approach (thread blocking execution):

```
var result = connection.query('SELECT * FROM USERS');

console.log(query); // THREAD BLOCKS
```

Optimised approach (non-blocking):

```
var query = connection.query('SELECT * FROM USERS', function (result) {

       console.log(result);

}); // EXECUTION FLOW CONTINUES
```

Despite the fact that this is not a new paradigm, the Javascript community is familiar with this model, and the same model is used in most of the modern browsers.

In the current ecosystem, Javascript is the only language that has a large array of asynchronous libraries that support operations like database queries, file I/O, event handling ([22] Mozilla Developer Network, 2017).

In summary, the main reasons for using Javascript on top of NodeJS as the primary development technologies for the front end are:

- Javascript has no threads, the task parallelization is done automatically by an internal thread pool;
- It is highly scalable, supports a large number of concurrent connections;
- It is easy to get started, as making an HTTP Server requires just a few lines of code;
- V8, the Javascript engine developed by Google which runs in Chrome has improved drastically.

4.1.1.2. Express JS

Express JS appears to be the most popular web application framework based on NodeJS. It is very well documented, tested, and it has a large community behind it. It is now at the 4th major iteration. The API provided by Express is very well appreciated by the open source community. Even frameworks from other languages and environments are using Express conventions to make their API intuitive ([23] StrongLoop, IBM, 2017).

With Express, significantly increased productivity and code maintainability is achieved by putting together the application microservices into a common model, followed by passing the data in a structured way to Jade, the templating engine. Following this approach, Express creates an abstraction layer which aggregates all the data received from the back end microservice, processes it, and then forwards the data to the Jade templates in a format that is easily rendered through custom defined objects in the template.

Express handles routing in an intuitive way. The logic is separated in different files, depending of the context needed, called namespaces. Each set of views has its own namespaces allocated (for example, all the pages related to Campaigns have the Campaigns namespace allocated) In the `app.js` file the controllers are registered and each controller is further described in its own separate file.

In this way, a true MVC application is achieved, by managing routes with Express, unifying the model from the micro services, and rendering the data with Jade. Express is considered to be appropriate for the use in this application, since it brings clear separation between Model, View and Controller, which makes the project modular and maintainable. The developer has a radically friendlier experience, as the process of passing data between Model-View-Controller is handled by Express.

It is worth mentioning that the same logic separation is achieved through using Jade views. Express integrates a list of view engines by default, Jade being the most popular one.

4.1.1.3. Transpilers

Another key aspect in improving the productivity of the front end application development workflow, is extending the core functionality of HTML and CSS with transpilers: Jade for HTML, and SASS for CSS.

4.1.1.3. The Jade (Pug) templating engine

Jade engine brings us the ability to extend the HTML markup with some key features ([24] *The official Jade Documentation*):

- variables, conditional statements, loops: In Jade, the data in JSON format sent by the server (in our case, the Express JS server) is rendered. Jades provides the ability to implement any procedural logic for the generated markup, just like any other regular programming language, by using statements including: `if`, `else`, `for` / `while` loops;
- automatic markup generation of duplicate components: in basic HTML, the components need to be manually re-written every time they are used. With Jade, repetitive work can be avoided through the use of mixins that can also receive parameters. In the current application, 5 important mixins can be distinguished: `campaign`, `categories`, `pagination`, `section-title`, and `site-navigation`;

- partial views, shared layouts: Code maintainability of our views is assured by the use of the Jade Shared Layout. New views can be created by importing the existing shared layout, and then specific content can be added for that view;
- clean Syntax based on whitespaces: The source code is significantly reduced and the syntax is equivalent with standard indented syntax from languages like Python;
- Integration with Express: All the variables are passed as Models to the View from the corresponding controller. This pipeline is done automatically in Express, following the configuration in the `app.js` file.

With all these features, the integration of Jade and Express opens new ways of extending the application by reusing the existing code, and adding new content with minimal intervention. Another key aspect is performance. All the previously mentioned features do not cause a decrease in performance, as the Jade files are translated to standard HTML code very efficiently, through instant server-side rendering, instantly when the page is loaded.

4.1.1.4. The Sass transpiler

CSS3 is a powerful tool in styling web application of all categories. In the last decade, Web Applications quickly became extremely complex. The initial design of CSS is now considered a major drawback in developing complex web applications. "*When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't.*" ([25] Hampton C., Natalie W., Chris E., 2017)

Nesting:

"*Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice.*" ([25] Hampton C., Natalie W., Chris E., 2017)

Variables, conditional statements and loops:

Data can be stored and replaced all over the codebase through the use of variables.

An example of using the map object for defining the colour of the different campaign categories in XpressStarter is illustrated below:

```
$color-sports: #3b97d3

$color-parks: #4fdf33

$color-arts: #b42dc4

$color-buildings: #dd9334

$color-food: #2ba342

$color-infrastructure: #cd5e5e

$categoryMap: ( SPORTS: $color-sports, PARKS: $color-parks, ARTS: $color-arts, BUILDINGS:
$color-buildings, FOOD: $color-food, INFRASTRUCTURE: $color-infrastructure )

.site-section--title
        text-align: center
        text-transform: uppercase
        @each $category in $categories
        &.#{$category}
                color: map-get($categoryMap, $category)
        .mdi
        @each $category in $categories
                &.#{$category}
```

```
    color: map-get($categoryMap, $category)
```

Partials and Imports

In Sass, the style sheets can be separated in smaller chunks, and then imported into the application wherever they are needed. The drawback of `@import` in CSS is that it creates an additional HTTP request. Sass on the other hand, collects and merges the existing code and the result is delivered through a single request.

As an example, the main page is intuitively styled by separate independent components and the code is very self descriptive.

In summary, the reasons for Sass being a suitable option for the use in XpressStarter are:

- SASS converts into pure CSS;
- SASS Cache keeps track of compilation status at every step, so there is no need to recompile the whole project every single time, it only compiles the modified section;
- SASS allows bundling all the separate files into one single file that is loaded with a single HTTP Request.

4.1.1.5. Gulp as a build tool

As mentioned above, repetition in programming is considered a bad practice. Jade, Sass are used in order to avoid repetitive tasks and automate unnecessary work. The same problem applies to build tools. In order to develop and run the project of our modern web application, a number of tasks need to be performed:

- Transpile Sass into CSS;
- Transpile Jade Views into plan HTML;
- Bundle Javascript, CSS, Images and fonts;
- Minify Javascript, and CSS files;
- Watch for code changes;
- Reload the NodeJS server when code changes are detected;
- Refresh the webpage when code changes are detected.

If a single line of code is changed, all these tasks are required in order to apply the desired effect. This is not only time consuming in terms of resources, but it produces difficulties for the developers. Luckily, the workflow can be optimised. The problem of task repetition and waiting for completion can be solved through using a task runner. For the XpressStarter platform, Gulp seems to be the best choice considering its features and principles.

The workflow can be automated through Gulp in the following ways

- tasks are defined to run in a particular order;
- for each task, the specific files are loaded into Gulp stream in order to be processed;
- at the end of a task the output to is sent to a destination (which could be another task).

Before diving into specific modules used for workflow of our web application, there are a couple of reasons that should encourage the use of Gulp as the primary option for a contender build tool and task runner. Below there is a comparison between Gulp and another popular build tool, Grunt:

- Gulp is the most popular build tool among Javascript developers, counting so far at least 40% of open source Javascript projects ([26] Ashley Nolan, 2015);
- Not only that is extremely popular and has a large community, but it is very well documented, on the official website, and on a large variety of technical blogs ([27] Github, 2017. *The official Gulp Documentation*);
- Gulp is focusing on code over configuration, whilst Grunt is mainly focusing on configuration over code. Instead of writing large configuration files, with Gulp is based on Javascript code;
- Gulp has more built-in modules;
- Better logic separation than Grunt: each module is designed to do only one task;
- Gulp is loading all the files into memory and pipes them from a task to another. In Gulp there is no need for additional writing in the intermediate steps. So, there is no need to create temporary files either. For this reason, Gulp is significantly faster than Grunt, especially as the number of pipes is larger.

On top of the built-in functionalities, the following list of modules is used with their Gulp integration in order to achieve better productivity:

- SourceMaps (described in Section 4.1.4.1);
- Nodemon (described in Section 4.1.4.2);
- BrowserSync (described in Section 4.1.4.3);
- JsHint (described in Section 4.1.4.4);
- Bower (described in Section 4.1.4.5).

Our Gulp workflow consists of two parallel branches: *development* and *production*.

For both branches there is a list of common processes that are executed from the very beginning:

- Jade is transpiled into HTML;
- Sass is transpiled into CSS and bundled into a single file;
- all Javascript source files are concatenated into a bundle file with Gulp-Concat.

For development workflow, the following additional tasks are executed on the development branch:

- The SourceMaps module is added in order to be able to debug easier in browser;
- Jshint is run in order to check common mistakes (e.g. syntax errors) in the file changed before effectively running (deploying) the entire application;
- Nodemon is launched, that watches the files and reruns the application whenever code changes are detected;
- Browsersync to refreshes all the browser connected to the browsersync proxy address.

On the production branch only the following tasks are performed:

- The JavaScript and CSS files are bundled together and minified;
- The images are compressed in a lossless manner in order to achieve a smaller image size without decreasing the quality.

No debugging overlay is added, since modules like Nodemon, Jshint, Browsersync are not relevant for the end user. In production, code changes monitoring is not needed. Therefore, the revision on the production branch is updated with a specific Gulp command that does not consume additional CPU and disk resources. As one of the objectives of our platform is to provide the end user with the best possible experience, all the Gulp tasks try to minimise the response time for our server as

much as possible. Therefore, minification of source files and images is crucial to decrease the loading time for dependencies and deliver the same outcome in terms of functionality.

4.1.1.6. The Bootstrap framework

Bootstrap is the most popular mobile-first, responsive front end development framework that was created by two developers at Twitter. At the current time is counting over 850 contributors and is one of the highest rated open source project on Github ([28] Github, 2017. *The official Bootstrap repository*).

Bootstrap contains a number of features that speed up the development for a responsive website. The design principle of developing responsive websites with Bootstrap involves splitting the webpage into a grid made up by 12 columns. ([29] Mark Otto, 2017. *The official Bootstrap Documentation*)

Each component in our HTML will inherit a Bootstrap class, in order to take a specific width in the 12 column grid. Bootstrap adjusts the sizes automatically depending of the class that is used on different screen sizes.

Following the mobile first approach is essential for the front end design of a modern web application. Mobile devices use was predicted to account for 75% of the Internet usage in 2017 ([30] Search Engine Journal, 2016). The most problematic part is that mobile phones and tablets can have very different screen ratios and screen sizes. On desktop, the problem is easier to solve. Screen monitors ratios are standardised, however their actual screen sizes can vary. This can be particularly problematic when encapsulating fonts and images into the design of a website. Bootstrap is addressing these issues through providing standard breakpoints that match most of the screen sizes.

In order to make the markup more meaningful, the Sass format for Bootstrap can be imported. This can facilitate the extension of the CSS rules applied on the application components using Sass specific inheritance. This way, the long sequences of Bootstrap CSS class names are avoided and encapsulation is achieved avoiding code repetition and naturalising the process of styling of the application.

4.1.2. The Javascript modules pattern

The primary goal of Javascript is to run as a lightweight programming environment inside the browsers. Code separation in Javascript was traditionally done by having a global namespace and separating our source files. In the source files every variable and function is being added to the global namespace. As the web application become more complex, and server side Javascript became popular, the pollution of the global namespace was becoming a serious problem in the Javascript ecosystem. This has led to an increased level of difficulty in applying the traditional code reusability techniques. Mozilla was the first organization to publicly make an initiative to solve this issue with Common JS. Later on, NodeJS was released and it contained a module system based on Common JS principles. Nowadays, NodeJS modules are being used in the whole industry and therefore, Common JS became almost obsolete, as the creator of NPM states. ([31] Schlueter, Isaac Z., 2013)

For the front end of XpressStarter, the logic was separated, keeping our namespace clean by using the same principles that Node modules are using underneath the hood. The front end Javascript code for our platform is divided into five main modules:

- formatters: a module for date formatting and time calculations;

- helpers: Module for working with regular expressions which are used to apply transformation in our URL for each page;
- init: entry point module for charts;
- request handlers: module for binding events for our search functionality;
- statistics: module for defining our rules for the objects rendered into D3 charts.

Below there is the code snippet from the formatters module that converts the dates retrieved by the backend in ISO format into a friendly format:

```
var formatters = (function() {
  var module = {};
  module.formatDates = function() {
        $('[data-type="iso-date"]').each(function() {
               if($(this).data('item') === 'campaign-date') {
                      var isoDate = $(this).data('original-date');
                      $(this).text(moment(isoDate).format('ddd MMM YY hh:mm'));
               }
        });
  };
  return module;
})();
```

This module can be called from any point in the system using the following pattern:

```
formatters.formatDates();
```

The code of the module patterns is approximately isomorphic, rendering the code more maintainable, reusable and preponderantly following the same object oriented structure both on the front end and on the back end. The only notable difference is embedding our browser code in an anonymous closure, process that is also performed internally by Node modules.

4.1.3. Critical external Javascript libraries used and their role in project development

In order to enhance the functionality of the front end of our system in a standardised way without ‘reinventing the wheel’, two major external libraries were used, that are also two very well maintained npm packages by the open source community.

4.1.3.1. Moment JS

Moment JS is a Javascript library for parsing, validating, manipulating dates and times ([32] Gregor Martynus, 2017). In the context of the web application, Moment JS providing reliable functionality needed for a crowdfunding platform. Moment JS powers the core calculation for the time that . Secondly, Moment Js is especially useful when internationalization becomes an issue. Moment JS has built in date formats and conventions available all over the globe.

4.1.3.2. D3 JS

D3 JS is the most influential open source project over the last year in addressing the problem of interactive data visualisation ([33] Github, 2017. *The official D3 Documentation*). D3 Js is written mostly in Javascript, but it also uses browser’s features like Canvas component and SVG icons to increase the performance and diminuate the resource consumption. Being extremely popular, is assures at least two important strengths: the large documentation and the open source community. The D3

API is perhaps one of the top reasons that leads D3 to its popularity. D3 API is simple, intuitive and yet extensible. In our example, on the statistics module, our columns are defined together with their “position values”. By choosing configuration over code approach, a perfectly functional chart can be generated only by following a comprehensive definition of the example JSON configurations. However, the charts can become interactive, it’s API is open for any major code addition.

4.1.4. Node modules used and their role in the project development and management

All the node modules described in this section: Bower, Nodemon, Browsersync, Sourcemaps are managed by NPM and integrated together with Gulp. All their usage is automated by Gulp and their interoperability and logic is maintained by through Gulp Pipes.

4.1.4.1. SourceMaps

When debugging the application, the browser's built in DOM inspector to used to check analyse the DOM hierarchy and the properties of the elements updated at runtime. When the computed stylesheets are analysed, the most common problem that is encountered is related to the fact that the compiled (“transpiled”) code in CSS does not necessarily match the original Sass source files. This renders the debugging of computed stylesheets particularly difficult. In order to prevent that, a module called SourceMaps is used. With Sourcemaps, the computed style is checked and mapped to the original lines of code written in Sass ([34] HTML5Rocks, 2012).

To optimise the development process, the SourceMap tool is run automatically with Gulp every time a code change occurs. Since this process is adding a significant processing overhead, it is only executed in the development branch (as explained in Section 4.1.1.5), being absent in the production environment.

4.1.4.2. Nodemon

Another important module for increasing the speed of the development workflow is Nodemon ([35] Official Nodemon website). It stands for Node Monitor and has the goal to rerun the node server of the front end microservice every time a code change to the source files on the server side is detected. Nodemon continuously watches for changes in the Express application.

4.1.4.3. Browsersync

Every time a CSS change is made, the browser windows need to be refreshed in order to preview the changes. Fortunately, there is a tool that automates this process called BrowserSync ([36] The official Browsersync Documentation). This module creates a proxy server on the same host where the node application runs, but on a different port and automatically forwards the new version of the web application to all the open browser windows when a code change is detected. It does not only allow for a single computer, but is useful for testing the application on multiple screens or multiple computers. It is especially useful during the development of functionality implies checking cross-browser compatibility (e.g. sensitive CSS code). This feature is not only useful when it comes to standalone testing, but it also facilitates the cooperation in teams, being an important piece that insures the scalability of the project for future developments.

4.1.4.4. JsHint

Making a change in the source code requires a series of many processes that are executed when Javascript is interpreted. Being an interpreted language, any error of syntax will be triggered at runtime (when usually the impact of errors on functionality can already be experienced by the end users). In order to prevent common syntax errors, the code is firstly checked with the JsHint module.

JsHint will firstly check for syntax errors, and then run multiple checks to ensure that the front end of the application follow a set of standard coding conventions ([37] Anton Kovalyov, 2017). The first feature helps the developer to save a significant amount of time through checking for possible errors before relaunching the Node application. The second feature, coding standard checking, is particularly useful in keeping a clean and consistent code on the entire application. This is especially useful in the Javascript ecosystem, where there are multiple paradigms used: *functional*, *procedural* and *object-oriented*. In large teams, with developers come from different backgrounds, following a code standard is a key element for productivity and code maintainability.

4.1.4.5. Bower

The Node Package Manager (NPM) is used in the project to manage the server side dependencies of the front end microservice (the node modules that aid in the development and manage the communication with the backend, described in Section 4.1.4). However, to manage the client side dependencies (described in Section 4.1.3), another dependencies manager is used, Bower, a tool developed by Twitter ([38] Twitter, 2017). There are various reasons that make Bower a better option for the client side libraries that only have the role to manipulate the DOM objects rendered in the browser to achieve certain functional requirements:

- there is a clear separation between the node modules and the bower modules;
- NPM choses stability over performance. NPM has a dependency tree, each module having its own dependencies. If two or more modules require the same dependence but with a different version, NPM installs both versions of that particular dependency. Bower uses a flat dependency tree, assuring that browsers will use the same dependency twice and avoiding the unnecessary overload of the application;
- Bower delivers all the modules in their minified version without the need to use another minification tool for them.

4.1.5. Optimisation techniques used in the front end

4.1.5.1. Automated minification of CSS and Javascript files at the deployment stage

Minification of Javascript and CSS is an optimisation technique that became a must in the recent years ([39] Microsoft, 2012). Converting source files into their smaller size equivalent (that generally accounts to 40% of the original size) brings at least five major advantages to the web application:

- The page loads faster for the end user since less data needs to be downloaded;
- The bandwidth cost for the infrastructure is lower;
- There are fewer resources required for the server in order to deliver the content. Therefore, fewer HTTP requests are needed to be initiated before the page can be rendered;
- Mobile users are consuming less bandwidth, preserving more of their mobile data allowance;

- A major factor that Google Takes into account when ranking the web pages is speed. Therefore, page loading time optimisation would bring benefits for the SEO.

Source files minification is achieved through running the following processes in sequential order:

- Elimination of the blank characters, blank lines, tabs, and newline characters;
- Elimination unnecessary curly brackets for statements wherever is possible;
- Renaming of the the variables with letters or short combination of letters.

4.1.5.2. Automated concatenation of CSS and Javascript files

In order to minimise the response time for the pages of our web application, every sources is bundled into four big files (). It is favorable for the browser to load all the scripts and styles at once, since minify and compress better our sources. Also, the fact that the resources are loaded asynchronously, the script and styles can't apply directly until every download is completed.

In order to achieve the smallest file sizes for the application in the production environment, the following steps are taken:

- all the external Javascript libraries are bundled into the *libraries.js* file;
- the custom Javascript source files are bundled into a file called *app-scripts.js*;
- the CSS of the external libraries is bundled into the *libraries.css* file;
- the Sass source files are compiled into CSS and bundled into *app-styles.css*;
- all the files mentioned above are minified.

It is worth mentioning that on top of the minification task run by Gulp, lossless compression is used with GZIP over HTTP.

4.1.5.3. Automated optimisation of images and fonts

After minifying and concatenating the Javascript and CSS sources, it remains to implement optimisation for images and fonts. For the images, a task Gulp called *ImageMin* is run, which creates new equivalent images with smaller sizes and no quality loss. When working with fonts and icons, they are stored in vectorial format (SVG). The vectorial format implies describing the shapes with mathematical functions and then scaling them at the desired size. By using this approach, both the responsiveness and the loading time issues are addressed. Icons and fonts appropriated for the end user screen size are generated at load time and. The size of the data transferred is also reduced compared to the traditional approach, where the icons are sent as images.

From the programming perspective, there is the advantage of having access to the icons and fonts in Jade and Sass. The icons in font format allows managing the icons in the same way as the CSS properties. Each icon has a CSS class associated with it and with Jade integration code duplication can be avoided and the process of attaching icons to the web components is naturalised.

A meaningful example of using font icons can be observed in the process of generating of the categories of the menu dynamically. The class name associated with each icon is declared as a value of the 'icon' property of the object associated with each category:

```
    - categories['sports-and-play'] = { 'cssClass': 'SPORTS', 'title' : 'Sports &amp;amp; Play',
'icon' : 'mdi-bike' }
```

```
    - categories['parks-and-gardens'] = { 'cssClass': 'PARKS', 'title' : 'Parks &amp;
Gardens', 'icon' : 'mdi-leaf' }
    - categories['arts-and-culture'] = { 'cssClass': 'ARTS', 'title' : 'Arts &amp; Culture',
'icon' : 'mdi-brush' }
    - categories['buildings'] = { 'cssClass': 'BUILDINGS', 'title' : 'Buildings', 'icon' :
'mdi-home' }
    - categories['food'] = { 'cssClass': 'FOOD', 'title' : 'Food &amp; Farming', 'icon' :
'mdi-food-apple' }
    - categories['infrastructure'] = { 'cssClass': 'INFRASTRUCTURE', 'title' :
'Insfrastructure', 'icon' : 'mdi-car' }
```

The classes defined in the object are subsequently used to generate the menu that includes the icons:

```
for item, path in categories
       a.item(class=item.cssClass, href="/campaigns/" + path)
              span.mdi(class=item.icon)  
              | !{item.title}
```

To summarise, using class icons presents two major advantages. Firstly, the generation for the dynamic menu is requiring only 4 lines of code. Secondly, any changes in the structure of the `categories` object will require a minimal amount of changes. There is only one line of code required to add a new category that will be automatically rendered.

Another important optimisation technique that has been implemented is the avoidance of render blocking. In order to provide the best possible user experience, the actual content of the pages need to be loaded first, while the Javascript files used for interactive elements are loaded in the background. As the browsers parse the rendered HTML documents from top to bottom when attempting to load the external resources, the references to the Javascript files are declared at the bottom, just before the end of the `<body>` tag. This way, the browser follows the HTML hierarchy, loading the static markup and the stylesheets first and processes the bundled Javascript files only after the rendering of the page is completed, preventing the user from unnecessarily waiting for the Javascript to load and process before they begin to see the actual content.

4.1.6. Issues and difficulties encountered during front end development

XpressStarter is an ambitious project, consisting of a platform that is expected to handle a heavy load of concurrent users. The architectural complexity increases at the same time with the technical complexity. There are two principal categories of difficulties that were encountered during front end development: code difficulties and configuration difficulties.

Problems related to code implementation occurred in the following scenarios:

Syncing the two async microservices in Express JS

Node does not encourage performing simultaneous requests that are normally synchronous. This is the reason why it was imperative to use an external module (Async) in order to be able to synchronise two asynchronous requests and return the merged results. This was applied on the homepage, where both the latest added campaigns and the campaigns with the greatest number of likes needed to be displayed. Fetching these two sets of data involved initiating two simultaneous synchronous requests simultaneously (in an asynchronous way).

Writing Gulp tasks

Writing Gulp tasks is a tedious process, as the order of execution for each task needs to be established before proceeding to the actual code. Redirecting the stream of tasks with pipes (similarly to the UNIX environment) can also become problematic at the beginning, as this is not a very intuitive process.

Making the URLs user friendly and processing the JSON response

Making URLs intuitive for the user is particularly problematic and unexpectedly complex for a relatively simple task. The solution was to use the router provided by Express and allocate a namespace for each view. The router translates the requests initiated to user friendly routes into the HATEOAS specific format needed by the backend. After the request is forwarded to the backend, the backend replies with the associated response in JSON format. This is later processed by the Express namespaces and rendered into the Jade templates.

Problems related to configuration occurred multiples time in separating the workflow of the development and production instances. The design choice of using transpilers and interpreted environments (Node and Javascript) required the configuration of additional development tools:

Nodemon:

Since Node applications do not apply code changes automatically, an external module was needed to watch on the source code and relaunch the application automatically.

Browsersync:

After the server applies the new changes, the browser is still unaware of them. In order to avoid the need of refreshing the browser windows manually, browsersync was needed to be added.

SourceMaps:

Since Sass is compiled (“transpiled”) into CSS, the original source code behind the CSS rendered by the browser could not be easily debugged initially. The developer is not aware of which line in the original Sass source code produced a particular erroneous rule of CSS. To address this issue, Sourcemaps was added, a module that allows the developer to see the equivalent line of Sass when debugging CSS in the browser.

4.2. Back end implementation technical specification

This section provides an overview of the implementation process for the back end of the application. A brief justification of the core technologies that have been used is presented, followed by an overview of the main issues and difficulties encountered during the design and development process of the back end and how they have been overcome. This section finalises with a description of the critical sections of the back end system.

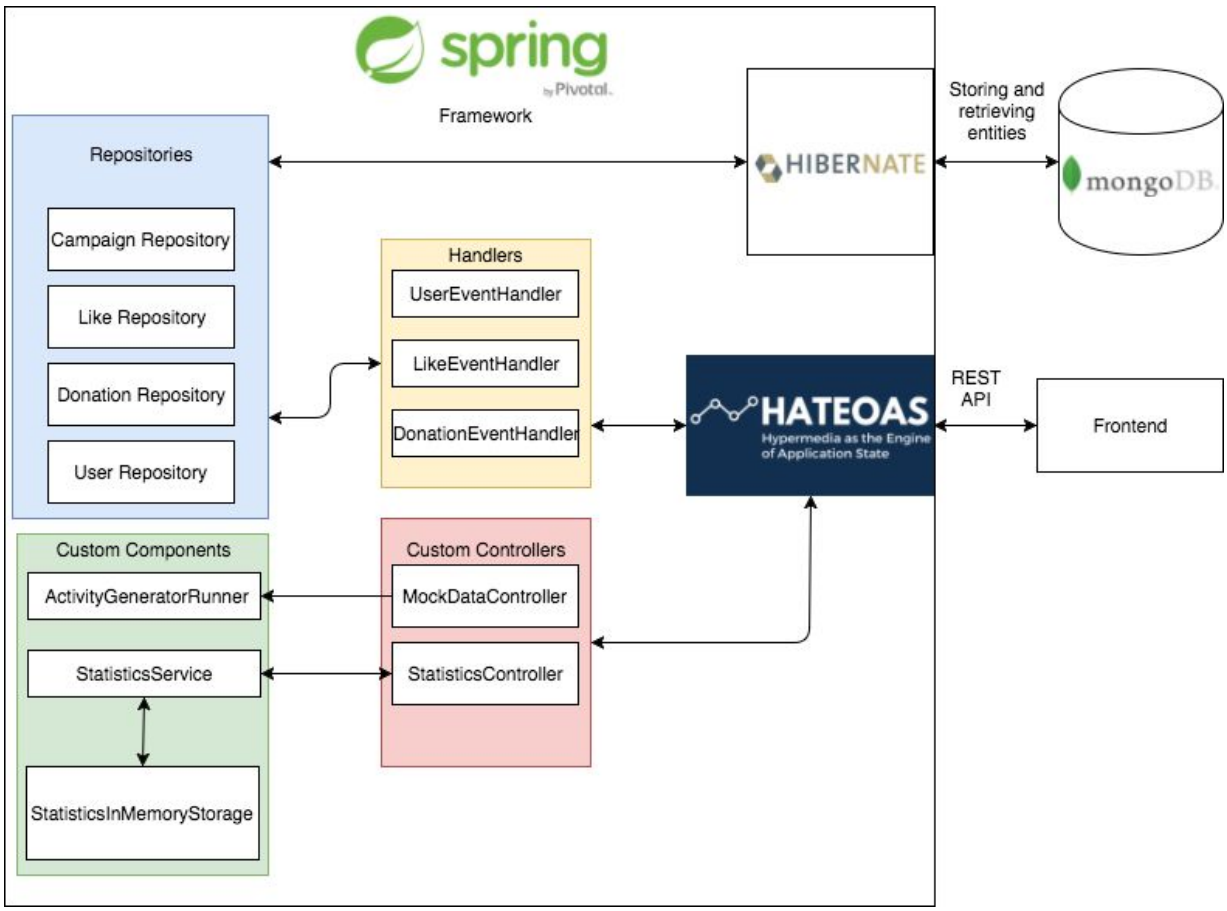


Fig. 4.2.1: An overview of the back end system architecture

The diagram above exposes the structure and the main components involved in the back end implementation. Hibernate and HATEOAS are two of the main modules used by the Spring framework.

The Hibernate ORM (Object-Relational Mapping) framework (described in detail in Section 4.2.1.5) communicates with MongoDB, storing and retrieving data. It also encapsulates a high-level API that translates all the requests into a SQL dialect set in the `application.preferences` file (in our case, the MongoDB dialect).

HATEOAS (described in detail in Section 4.2.1.4) communicates with the clients through a REST API. Since the HATEOAS module generates the controllers based on the repositories, handlers must be implemented to apply the business logic. These process the data, before or after persisting it (e.g.: when storing a like, a check is made before persisting it to make sure the like does not already exist).

When retrieving data, HATEOAS goes directly to the repositories. Entities (described in detail in Section 4.2.1.3) have annotations that describe what the constraints associated with them are in order for data to be valid. If the data submitted is not valid, an error message is sent back through HATEOAS to the client.

For loading of mock data and generating statistics, custom controllers are used. Firstly, `MockDataController` reads the two JSON files provided (`mock_campaigns.json` and `mock_users.json`). Secondly, the *Users* and *Campaigns* entities generated from the json files are persisted in the database. Thirdly, an `ActivityGeneratorRunner` is created for each of the users to generate

activity consisting of mock donations and likes. Multiple threads are used to speed up the process.

The `StatisticsController` relies on the statistics service to process data and retrieve statistics. To improve the speed of processing, the statistics service uses `StatisticsInMemoryStorage`, eliminating the need to deserialize the objects into memory. Once the requested data is retrieved, it is firstly aggregated and sorted based on the criteria set by the client, and secondly based on the number of objects the client requested. Afterwards, the sorted and aggregated data limited to the number of objects requested is sent to the client.

4.2.1. Brief justification of technologies used

For this project, the decision was made to use Java as the programming language, together with Spring Framework. In order to persist the data in the system, MongoDB was selected as the database system. The back end offers a REST API that supports a CRUD system for entities. For the API , HATEOAS was used for the format of the requests and responses. Spring framework provides a lot of boilerplate code, reducing the time needed from the design phase to the prototype stage.

4.2.1.1. Java

Java is a programming language that has quite a heritage for enterprise web applications ([40] TheServerSide, Kurt Marko, 2017). The language is mature, being constantly developed since it was released in 1995 by Sun Microsystems. All the updates of the language have been focused on retaining backwards compatibility. The ability to upgrade the Java package on the server without having to refactor code might not seem a major advantage, but in a real world scenario, the lack of time and resources allocated on refactoring code to work with the latest versions is important. A popular choice for web applications, the PHP language has many known issues when upgrading between major versions ([41] The PHP Group, 2017).

Another reason why Java is a good choice for the application is multithreading support. Built-in collections, classes and helper classes that aid in concurrent programming make it a viable choice in comparison with languages such as Python or PHP. PHP has no native support for threads, while Python has a constraint called GIL (Global Interpreter Lock) that only allows one thread to run at a given time.

Another advantage of Java is that is cross-platform. The slogan of Sun Microsystems is *'Write once, run everywhere'* ([43] Boyarsky J, Selikoff S.). This is due to the fact that Java is an interpreted language. When compiling the code, it is converted into bytecode that is executed inside the JVM (Java Virtual Machine), allowing standardisation between platforms. This means that the back end of the platform could be hosted on a server that runs on any platform, since it only needs to have the JRE (Java Runtime Environment) installed.

To summarise, Java has good concurrency support, is cross platform compatible and a mature language with heritage in the development of enterprise web applications. All these arguments make Java a truly viable option for the use in this project.

4.2.1.2. MongoDB

MongoDB is a NoSQL database program. It has a document-oriented model that instead of tables and rows, stores data as documents. This allows the stored data to

resemble the object and it allows handling the constraints in the code, offering more flexibility. It is also cross-platform compatible and uses JSON both as the response format and as a format to store the objects.

Another option for the persistence layer could have been MySQL that stores data in tables in rows and objects usually spread across multiple tables. A query to return a full object would have been much more complex, involving multiple joins or it would have required joined tables that increase the number of queries needed. This is an area in which the document-oriented model of MongoDB excels, by allowing the retrieval of the object in a single query.

Another reason why MongoDB was used is that it is lightweight and fast compared to MySQL. Predicting a user base of potentially hundreds of thousands of users for the web application, a scalable setup is needed. MongoDB supports sharding natively, meaning that it can spread data over multiple servers to increase performance. This is called horizontal scaling. When a boost of performance is needed, more machines could be added easily. MySQL does not handle sharding natively, making a distributed database setup hard to achieve. The only way to boost the performance of a MySQL server is by running it on a more performant server. This is called vertical scaling.

Sometimes, it is necessary to store files inside a database. For example, if the need arises where multiple back end servers are load-balanced, getting a resource like a profile picture or a binary file can be quite expensive, needing a shared storage between servers. While MySQL handles BLOBs, the way in which is stores them is inefficient. MongoDB handles big chunks of data through a system called *Grid File System* or *GridFS* for short. *GridFS* splits large files in chunks of 256KB and allows storing metadata together with the file and running queries on the metadata. The reverse transformation, from chunks into a single file is done client side.

Therefore, with MongoDB allows using both a distributed database and a distributed file system at the same time. Different replication settings can be configured to ensure the cluster is resilient against failures.

Distributed systems are subject to Brewer's CAP theorem ([42] Brewer E., 2000), which states that distributed systems have a maximum two out of the three properties, Consistency, Availability and Partition-Tolerance. Consistency means that each read is guaranteed to receive the latest write or an error, Availability means that a read receives a non-error response that may or may not be the latest write. Partition Tolerance means that the system continues to function even if a number of packets are dropped or delayed by the network. MongoDB is a CP distributed system. This means that a non-error answer may not always be received to a request if a component is unavailable (does not provide permanent availability). However, the advantages of the other two components (Consistency and Partition Tolerance) are given by MongoDB. The drawback generated by the lack of availability can be compensated through the use of a sharded configuration comprised of multiple replicas, query resolvers and configuration servers. This approach ensures that if a component fails, the service is not affected. A proof of concept of this approach deployed at a small scale is outlined in Section 5.3.

Fig . 4.2.1.2.1: The CAP properties of different distributed systems (taken from http://wbzyl.inf.ug.edu.pl/nosql/images/cap.png)

To summarise, MongoDB was a design choice due to being a fast and lightweight database program, with an integrated distributed file system, that stores data in a document oriented way, allowing deserialization of data to be achieved much easier than building objects from result sets generated by MySQL.

4.2.1.3. Spring Framework

Spring Framework is the most popular Java web application framework as of March 2017 ([44] Simon Maple, 2017).

While Java has a heritage of web applications, it is split between Java SE (Standard Edition) that usually does not handle servlets and web services, and Java EE (Enterprise Edition), that offer a lot of functionality like JMS (Java Messaging Service) and JPA (Java Persistence API). Java EE would bring a lot of unnecessary complexity to this application. Therefore, something lightweight was needed. This is where Spring can help. It is built on top of Java SE, but provides the same functionality, while being modular. This means that the required modules can be used without needing to import extra functionality.

Spring Framework has a flavour called Spring Boot. It reduces the boilerplate code needed even further and allows a very short time between design and implementation. It also includes an embedded HTTP server, so the application can be launched by running the jar file rather than uploading the code to an application server like Tomcat or Glassfish. For this project, Maven is used as a dependency manager that allows building the project by downloading dependencies from a repository.

A notable feature shared between Java EE and Spring is dependency injection. This allows keeping some objects in a bean registry and inject them in different parts of the code. This decouples code and accomplishes a IoC (Inversion of Control) design that decouples the classes and generally renders the code more maintainable and understandable.

To enable the persistence layer with MongoDB, the `spring-boot-starter-data-mongodb` package was imported, that was configured with the default settings for connecting to the database server, assuming it runs on the local machine with the default port and security settings. These can be configured in the `application.preferences` file. The repositories are defined by extending an interface, `CRUDRepository` for basic CRUD, `MongoRepository` for MongoDB extra features and `PagingAndSortingRepository` for pagination and sorting of results. To add different queries, a named query can be defined or a special syntax can be used that allows Spring to build it. For example, in the event of needing to fetch the users filtered by their email address, a method called `User findByEmailAddress (String email)` could be implemented, and as long as the entity *User* has a field called `emailAddress`, a custom query will be created for it.

After defining entities and a way to persist them, a logic was needed for the front end to interact with the back end. To setup a REST API with the HATEOAS format, two modules were needed to be added in the Maven manifest. Through Java reflection, Spring Framework creates an endpoint for each repository handling CRUD operations over HTTP (POST, GET, PUT, DELETE). It also allows the client to use the special queries defined in the repository through the API. For example the method called `findByEmailAddress` is exposed to the endpoint associated with the *User* entity.

Another important aspect to mention is that the format of the requests and responses are serialized based on the format negociation with the client. The default format used by Spring is JSON, but if the client agrees to use XML as the prefered format for communication, this can be done automatically by Spring.

Once the back end was able to communicate with the front end, validating the input from it was the next step. Spring offers standard validators associated with annotations such as `@NotNull`, `@Size(min=,max=)`, `@Email`, etc. The validation system offered also throws back an error through the REST API in a consistent manner, where it could be easily interpreted by the front end.

The lack of controllers and services for these entities introduced a new problem related to the implementation of custom logic for adding / deleting / modifying entries. Spring offers addresses this issue elegantly by allowing the use of handlers where methods can be called before or after an event occurs (the `@beforeCreate` and `@afterCreate` annotations). Spring Framework uses AOP (Aspect Oriented Programming) to allow for more loosely-coupled code. The approach uses handlers that intercept operations, catching events and allowing to implement generic code that is injected in the class and runs before, during or after a method is called. The use of event handlers in our application is sumary described in Section 3.4 and described in detail in Sections 4.2.2.2 and 4.2.3.

Another concern is security, that Spring also addresses. By including the security module, the REST API can be protected through different techniques such as using the `BasicAuth` protocol, form logins, etc. Being supported by a thriving community Spring offers a module called Spring Social which allows users to register and log in using social media accounts such as Facebook, Google, Twitter, etc. By setting a property to `true` in the `application.preferences` file, as well as API credentials for the social service used, a special controller called `Connect` is enabled and logging in with Facebook becomes as simple as redirecting the user to `/connect/facebook`.

4.2.1.4. HATEOAS

HATEOAS is a format for REST APIs that stands for *Hypermedia As The Engine Of Application State*. It allows the client to discover possible actions through hyperlinks and requires no prior knowledge of the structure of the API. This means that each entity has a dynamically generated unique link that points to it. The entity response also has all the links to the other entities that are part of it embedded in. For example, a campaign object retrieved from the API contains a link to itself, a link to the user who is the beneficiary and a link to the admin user that approved it (if the campaign has been approved).

HATEOAS is rated at the maximum level of maturity for a REST API as defined by Leonard Richardson, which is level 3 ([45] Martin Fowler, 2010). The levels start from level 0, which is basic remote procedure invocation through HTTP where a single endpoint and exchange messages that trigger remote procedures are present. Level 1 APIs use multiple resources such as an endpoint for getting some data that is required, and another endpoint to interact with the desired resource. Level 2 assures that HTTP verbs such as GET, POST and PUT are implemented as closely as possible to their original intent (e.g. GET is a safe operation, so it must not change any data). Level 2 also makes use of the HTTP error codes to signal different situations. Level 3 requires no prior knowledge of the API, guiding the client to the desired operation through hypermedia controls.

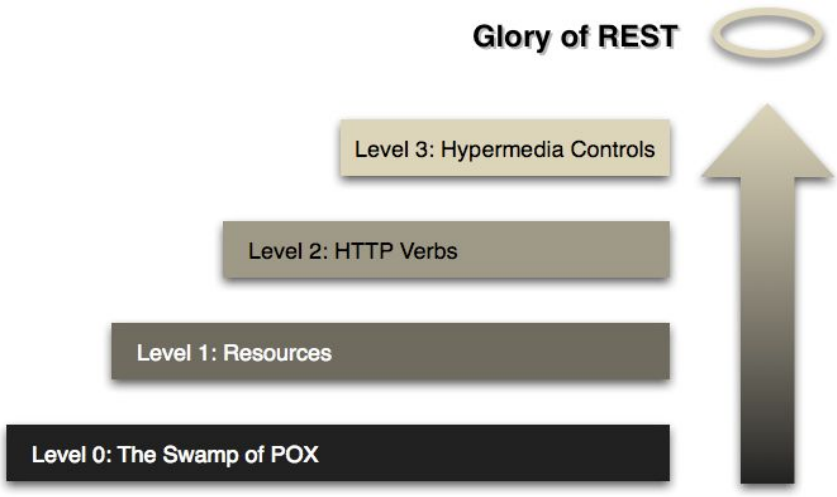


Fig. 4.2.1.4.1: The Richardson Maturity Model ([45] Martin Fowler, 2010)

Using HATEOAS, by accessing the root path (`/api/v1`), a list with all the repositories can be observed, as well as an URI to `/api/v1/profile`.

Below there is an extract of the response that can be observed by accessing the URI at `/api/v1/`. All the entities present in the system (`users`, `likes`, `donations`, `campaigns`) are present in the extract below:


```
{
  "_links": {
    "users": {
      "href": "http://www.xpressweb.site:8080/api/v1/users{?page,size,sort}",
      "templated": true
    },
    "likes": {
      "href": "http://www.xpressweb.site:8080/api/v1/likes{?page,size,sort,projection}",
      "templated": true
    },
    "donations": {
   "href": "http://www.xpressweb.site:8080/api/v1/donations{?page,size,sort,projection}",
      "templated": true
    },
    "campaigns": {
      "href": "http://www.xpressweb.site:8080/api/v1/campaigns{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile"
    }
  }
}
```


By accessing the URI at `/api/v1/profile`, the links describing the structure of each entity can be seen, as well as the associated constraints.


```
{
  "_links": {
    "self": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile"
    },
    "users": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile/users"
    },
    "likes": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile/likes"
    },
    "donations": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile/donations"
    },
    "campaigns": {
      "href": "http://www.xpressweb.site:8080/api/v1/profile/campaigns"
    }
  }
}
```


By accessing the URI for any of the repositories with the `/api/v1/profile/` prefix, the structure of an entity of that repository can be explored. Below there is a snippet describing the structure of a *Donation* entity:


```
{
  "alps": {
    "version": "1.0",
    "descriptors": [
      {
        "id": "donation-representation",
        "href": "http://www.xpressweb.site:8080/api/v1/profile/donations",
        "descriptors": [
          {
            "name": "amount",
            "type": "SEMANTIC"
          },
          {
            "name": "donatedOn",
            "type": "SEMANTIC"
```

```
            },
            {
              "name": "status",
              "doc": {
                "value": "PENDING, OK, REVOKED",
                "format": "TEXT"
              },
              "type": "SEMANTIC"
            },
            {
              "name": "user",
              "type": "SAFE",
                                                                                "rt":
 "http://www.xpressweb.site:8080/api/v1/profile/users#user-representation"
            },
            {
              "name": "campaign",
              "type": "SAFE",
                                                                                "rt":
 "http://www.xpressweb.site:8080/api/v1/profile/campaigns#campaign-representation"
            }
          ]
        }
```

To summarise, HATEOAS allowed the back end REST API to be more discoverable and self documenting.

4.2.1.5. Hibernate

Hibernate is a popular ORM (Object-Relational Mapping) framework that allows the quick persistence and retrieval of objects from a database. In this project, it is used by Spring Framework as the default ORM system to translate high level operations into low level commands for the database.

Hibernate allows an abstraction layer between the high level logic and the low level implementation by interacting with the database with methods that take and return objects. It uses different dialects to persist objects in different database languages. This facilitates the change of storage systems, as the code only interacts with abstract methods. The task of converting objects to queries and back is offloaded to this module instead of being handled by the developer. Another feature of Hibernate is that it can create schemas based on the entities. If it is ever required for the system to switch to MySQL, the only requirement for the developer would be to replace the module in the Maven manifest, and it would automatically create a database and tables. This provides a lot of mobility since the requirements might change over time.

Once the repositories are setup, the controllers can simply be injected in them and used together as if they were concrete classes. This injection is done via the `@Autowired` annotation. To store entities in the MongoDB database, the repository offers a simple method called `save(Object object)`. To retrieve an object by its `id` a method called `find(String id)` is used. Deleting an object can also be done easily through a method called `delete(String id)`.

The triviality of retrieving an object can be observed from the code snippet pasted below, that uses only one custom defined method:

```
…
@AutoWired
LikeRepository lRep;
Like check =
lRep.findByUserIdAndCampaignId(like.getUser().getId(),like.getCampaign().getId());
```

The same is the case for the operation of persisting an object, that can be observed in the snippet pasted below:

```
cRep.save(campaign);
```

Hibernate uses Java reflection in order to perform operations (analyse an entity and create tables, add data to a table, etc.). Therefore, the action of persisting data is much more abstract and leads to a more decoupled architecture. In the event of the need to switch the database servers that back the persisting system, the process is easy, as Hibernate supports different dialects for different programs (MySQL, PostgreSQL, MongoDB, etc.).

The way Hibernate works is by setting annotations on the fields of an entity to signal the type of data they store and the method by which it is stored in the database. The annotations also mark what fields are to be indexed and which of these will be used as an equivalent of a primary key. Joins between entities can be specified using annotations, ensuring the database representation is accurate.

To optimise this process, it uses proxy objects to represent the data retrieved, so only when a method on an entity is called, the actual entity is retrieved from the database. This method is called lazy loading.

The main advantage brought to the project is that it allows to define the way in which the entities in the application interact with the other entities without the need to define or design the schema first. Once the entities are set up, this module can create schemas based on the definitions and relations between them.

4.2.2. Issues and difficulties encountered during back end development

During the back end development of this application a few challenges were encountered, including refactoring code to simplify it, using different modules more efficiently, and addressing design flaws that were not apparent during the design phase.

4.2.2.1. Moving from user defined controllers to HATEOAS generated controllers

Firstly, the services and controllers were written and set out for each entity, ensuring that the API for each one was the same. At least 4 methods were needed to be defined for the HTTP verbs to accept GET, POST, PUT, DELETE requests, as well as special requests for retrieving paginated results. This concluded to the creation of a CRUD interface for the clients. Since controllers should not contain any business logic, these methods were delegated to services.

The project was getting quite complex and started to feel unmaintainable. Since Spring offered so much from the modules that were already implemented, a promising research into a better way to manage the data flow was conducted. The HATEOAS module was discovered, which uses Java reflection to get all the repositories and creates endpoints for each one. Another great feature that HATEOAS offers is that for all the custom queries defined, it created links that enables to search through the repository without the need to write the code for them. This unified and simplified the design and reduced the effort needed to add new entities as the endpoints were generated dynamically.

4.2.2.2. Event handlers

After migrating to dynamically generated controllers, another issue became apparent: the lack of a suitable place to implement the custom business logic when executing operations such as additions and deletions. After reading the documentation, a decision was made to implement the custom handler to intercept the requests and execute the business logic. This has been done using Spring's AOP (Aspect Oriented Programming) principles.

A handler had to be defined for each repository in order to intercept certain events before or after an operation takes place (`beforeSave`, `beforeCreate`, `afterSave`, `afterCreate`). These allowed setting dates on entities in order for the backend to rely on client side validation to make sure the date created was correct.

This method also allowed for the code to be packaged in a structured, easy to interpret way in order to facilitate potential future changes to the code.

4.2.2.3. Loading mock data (the description of the crawler / loadMockData module / activityGenerator module)

Having developed a fully functional CRUD API, a way was needed to test it with the front end and make sure it was performing as expected. Since manually triggered inserts and changes would not be enough, some mock data to test was needed. A list of generic template names was created (e.g.: "Save the <<name>>","<<name>> needs our help", etc.). This way, a list of 100 campaigns with generic names was created. Having such few campaigns, it was still not enough test data. As seen in the initial investigation, SpaceHive was a similar application with real life data. A good way to test the system was to fetch their data and simulate a real workload. A crawler was written that parsed all the campaign names and descriptions from SpaceHive. The source code of this crawler will be attached alongside with the application code.

Having real campaigns, a way to generate activity to test the statistics system was still needed. A website called *Mockaroo* provided a list of 1000 people from which users were created. Campaigns were created with the start date associated with a random number of days ranging from 0 to 100 before the current date. The `activityGenerator` class is then used. It takes each user and for each day, it looks up what campaigns were active on that day, and generates a random number of likes and a random number of donations with random pledged amount.

Having to repeat this operation for 100 users, for 50 days on average, for over 200 campaigns was a time consuming process. A better way of achieving this was to initiate concurrent threads. For each user, an *ActivityGenerator* was created and enqueued in an *ExecutorService* that ran as many threads in parallel as the processor would allow. This reduced the execution time from approximately 30 minutes to under 10 minutes.

After a close inspection of the code it was noticed that for each thread, all the campaigns had to be deserialized from the database into objects in memory. This operation involved increasing the workload of the database up to the point where the application is slowed down considerably. Since the list was only needed to be retrieved once (at startup), the reference to that list of campaigns was passed to all the threads and they were let to iterate and find which campaigns were active during a

certain day, reducing the execution time to 1 minute. For testing purposes, this duration was considered as being acceptable.

Since the mock data loader was not part of the API, a different path was needed, `/loadMockData` being chosen. Calling this loads the data from two JSON files (`mock_campaigns.json` and `mock_users.json`), one for users and one for campaigns then generates the activity for all of the users.

4.2.2.4. DBrefs (linking entities)

A challenge was also represented by linking entities (e.g. linking donations to their corresponding campaign). Given the way MongoDB works, Many-To-One relationships were not permitted. After reading the documentation for Spring MongoDB modules, the fact that it has an annotation called DBref was discovered, that serializes that member object as the collection name and the id, therefore allowing to maintain the relationships in the code rather than in the database program.

Due to the way HATEOAS works, using DBRef means that the entities are represented by links rather than embedding the actual data which means cleaner code on the front end.

4.2.2.5. Special fields and views for User / Campaign to make retrieval faster

Once activity was generated, another issue was noticed: getting the value for the current pledges of a campaign. This meant that getting the campaign from the API was needed, followed by a search for all the donations associated with that campaign id. The same was applying for the pairs of data consisting of *Campaigns* and *Like count*, and *Users* and *Donation count*.

Fortunately, the custom handlers could be used to specify some logic after an entity was persisted. This involved calculating the values above each time a Donation or Like was persisted to the database. In order to make the retrieval of objects easier, the inline views were used.

Below there is a snippet of the a *Like* object in the response generated by the URI of `/api/v1/likes/search/findByCampaignId` before adding the inline view:

```
like: {
        givenOn: "2017-04-01T20:49:00",
        _links: {
                self: {
                        href:
        "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0"              },
                like: {
                        href:
        "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0{?projection}",
                        templated: true
                },
                user: {
                        href:
        "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0/user"
                },
                campaign: {
                        href:
        "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0/campaign"
                }
        }
}
```

To speed up the data retrieval, the user metadata was embedded into the Like via an inline view:


```
like: {
       givenOn: "2017-04-01T20:49:00",
       user: {
              firstname: "Melissa",
              lastname: "Bowman",
              email: "mbowmanb@google.ru",
              wantsToReceiveEmail: false,
              memberSince: "2017-02-02T20:49:00",
              role: "BENEFACTOR",
              profilePicture: null,
              totalDonated: 3713
       },
       _links: {
              self: {
                     href:
       "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0"              },
              like: {
                     href:
       "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0{?projection}",
                     templated: true
              },
              user: {
                     href:
       "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0/user"
              },
              campaign: {
                     href:
       "http://xpressweb.site:8080/api/v1/likes/58ff9c62a6842211b66c4cc0/campaign"
              }
       }
}
```


Without this modification being made in the back end, the front end would have needed to initiate a separate request to the back end for each *Like* of a *Campaign* in order to determine the name of the user that gave it. This would have resulted in a significantly higher amount of requests to the back end, hindering the performance of the web application, especially in a scenario of multiple users being online at the same time. For example, for a campaign having 1000 likes, the front end would have needed to make 1000 requests to retrieve the details of the associated users. On a page with 8 campaigns displayed, the front end would have needed to make 8 x 1000 = 8000 requests for each client using the application.

Another issue that was noticed during the implementation of statistics was that because each resource was represented by a link, another request was needed to be initiated for each item to get the data about that entity. Spring allows setting a view for each entity. This allowed specifying some key values along with the link to the resource. Using this approach, when displaying statistics some information can be included about the *Campaigns* involved, including the link to each campaigns that allows taking the end user to the page containing detailed information about that campaign.

4.2.2.6. Tuning the statistics system

An issue that was noticed with the statistics system is that it would take a lot of time to display them, and waiting 10 seconds or more for a response is not acceptable. Therefore, an investigation was done into the cause of the high response time. For each request to an endpoint from the statistics category, for each of the entities, objects are created in memory that store the data retrieved from the database. This is

an expensive process, since it requires allocating memory for each entity. Using the `mockDataLoader` and the `activityGenerator`, since there are 1000 users, each giving 10 donations per day for 1000 days, that would equate to 1,000 x 1,000 = 1,000,000 donation objects. Each donation object is stored using the double data type, which stores the donated amount and the date when it was donated on. Knowing that a double data type is represented in memory using 8 bytes and assuming the date is stored using 24 bytes, a donation object needs 8 + 24 = 32 bytes. This means that each request to a statistics endpoint that retrieves the average donations per user, 32 x 10^6 bytes = 32 MB are retrieved from the database and stored in memory. After the response to the request is sent, the application needs to clean the memory. This type of heavy workload triggers the garbage collector to run more frequently, causing a degrade in performance.

The Spring framework offers a service similar to the cron jobs in Linux, that enables a method to run at a set interval. By using this, an in-memory storage for statistics was created for the data retrieved from MongoDB, which refreshes at a preset interval. This allowed the statistics to run on objects that were already stored in memory without the need to create them. As they were not being created, the most time consuming piece of code handling statistics could be eliminated. The response time dropped from a couple of seconds to a couple of milliseconds and significantly increased the overall performance.

In the future a ”refresh now” button could be implemented (in the back office of the admin committee) that triggers the refresh, rather than waiting for the automatic refresh to occur.

4.2.2.7. Embedding entity links to statistics

One issue that was noticed when working on the statistics service was that it was only setting the names of the users or campaigns that were being displayed in the graphs and the link to the underlying entity was missing. Since HATEOAS provides an `EntityLinks` Bean, that service could be injected that allows fetching the link for each of the entities. This approach optimised the way in which the JSON response is structured for the front end in order to make the rendering of the entities easier.

4.2.3. Critical sections

In order to understand the critical sections of the system, the data flow through the system needs to be presented first, together with the processes that are triggered when application is started.

4.2.3.1. Startup

During the startup of the application, an instance of Apache Tomcat is launched:

```
2017-04-23 10:54:22.840  INFO 9804 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat initialized with port(s): 8080 (http)
2017-04-23 10:54:22.854  INFO 9804 --- [main] o.apache.catalina.core.StandardService   :
Starting service Tomcat
2017-04-23 10:54:22.855  INFO 9804 --- [main] org.apache.catalina.core.StandardEngine  :
Starting Servlet Engine: Apache Tomcat/8.5.11
2017-04-23 10:54:22.989  INFO 9804 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]
: Initializing Spring embedded WebApplicationContext
2017-04-23 10:54:22.990  INFO 9804 --- [ost-startStop-1] o.s.web.context.ContextLoader
: Root WebApplicationContext: initialization completed in 2452 ms
2017-04-23 10:54:23.239  INFO 9804 --- [ost-startStop-1]
o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
```

```
2017-04-23 10:54:23.246  INFO 9804 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to:
[/*]
2017-04-23 10:54:23.256  INFO 9804 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-04-23 10:54:23.257  INFO 9804 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to:
[/*]
2017-04-23 10:54:23.257  INFO 9804 --- [ost-startStop-1]
o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
```

After starting, the instance loads the application onto the embedded Tomcat server and loads up the filters for the URLs. The HTTP server is the most critical part of the application, as communicating with clients is a core part of the functionality. The next step is loading the Persistence Layer. Since Spring manages the connections to the database system, starting and configuring the driver are done automatically. It firstly connects to the MongoDB instance and then determines the type of setup (if it is clustered or not). This is especially helpful when working with a cluster, as the topology is discoverable, which means that only one server from that topology needs to be known, as it offers details about the rest of the cluster. A proof of concept for a cluster configuration is illustrated in Section 5.3.

```
2017-04-23 10:54:23.869  INFO 9804 --- [main] org.mongodb.driver.cluster               :
Cluster created with settings {hosts=[localhost:27017], mode=MULTIPLE,
requiredClusterType=UNKNOWN, serverSelectionTimeout='30000 ms', maxWaitQueueSize=500}
2017-04-23 10:54:23.869  INFO 9804 --- [main] org.mongodb.driver.cluster               :
Adding discovered server localhost:27017 to client view of cluster
2017-04-23 10:54:23.991  INFO 9804 --- [localhost:27017] org.mongodb.driver.connection
: Opened connection [connectionId{localValue:1, serverValue:1}] to localhost:27017
2017-04-23 10:54:23.994  INFO 9804 --- [localhost:27017] org.mongodb.driver.cluster
: Monitor thread successfully connected to server with description
ServerDescription{address=localhost:27017, type=STANDALONE, state=CONNECTED, ok=true,
version=ServerVersion{versionList=[3, 4, 1]}, minWireVersion=0, maxWireVersion=5,
maxDocumentSize=16777216, roundTripTimeNanos=966826}
2017-04-23 10:54:23.995  INFO 9804 --- [localhost:27017] org.mongodb.driver.cluster
: Discovered cluster type of STANDALONE
2017-04-23 10:54:24.407  INFO 9804 --- [main] org.mongodb.driver.connection            :
Opened connection [connectionId{localValue:2, serverValue:2}] to localhost:27017
```

The next step in the startup process is the loading of the in-memory statistics cache. As stated above, this involves getting all the objects from the database and storing them into memory to be processed by statistics requests.

```
2017-04-23 10:54:24.623  INFO 9804 --- [main] c.x.s.StatisticsInMemoryStorage          :
Warming up statistics cache
2017-04-23 10:56:44.988  INFO 9804 --- [main] c.x.s.StatisticsInMemoryStorage          :
Statistics have been refreshed!
```

Once all the core services are loaded up, a scanning for controllers is started. The user-defined controllers are loaded first, followed by another scan of the repository for custom entities. Other repositories for these entities are then generated.

```
2017-04-23 10:56:46.228  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/loadMockData]}" onto public void
com.xpressstarter.controller.MockDataController.loadMockData()
2017-04-23 10:56:46.233  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/api/v1/statistics/avgdonation],methods=[GET]}" onto public
java.util.List<com.xpressstarter.statistics.Statistical>
com.xpressstarter.controller.StatisticsController.getAverageDonationPerCategory()
2017-04-23 10:56:46.234  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/api/v1/statistics/topdonatingusers],methods=[GET]}" onto public
java.util.List<com.xpressstarter.statistics.Statistical>
com.xpressstarter.controller.StatisticsController.getTopDonatingUsers(int)
```

```
2017-04-23 10:56:46.234  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/api/v1/statistics/getnearlyfunded],methods=[GET]}" onto public
java.util.List<com.xpressstarter.statistics.Statistical>
com.xpressstarter.controller.StatisticsController.getNearlyFundedCampaigns(int) throws
java.io.IOException
2017-04-23 10:56:46.235  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/api/v1/statistics/gettopcampaigns],methods=[GET]}" onto public
java.util.List<com.xpressstarter.statistics.Statistical>
com.xpressstarter.controller.StatisticsController.getTopCampaigns(int,int) throws
java.io.IOException
2017-04-23 10:56:46.236  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/getBeans],methods=[GET]}" onto public java.lang.String[]
com.xpressstarter.controller.TestController.getBeans()
2017-04-23 10:56:46.237  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/getLinks],methods=[GET]}" onto public org.springframework.hateoas.Link
com.xpressstarter.controller.TestController.getLink()
2017-04-23 10:56:46.237  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/getFBPrincipal],methods=[GET]}" onto public byte[]
com.xpressstarter.controller.TestController.getPrincipal()
2017-04-23 10:56:46.239  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/error]}" onto public
org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.Htt
pServletRequest)
2017-04-23 10:56:46.240  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/error],produces=[text/html]}" onto public
org.springframework.web.servlet.ModelAndView
org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http
.HttpServletRequest,javax.servlet.http.HttpServletResponse)
```

Now, for Spring Social and Spring HATEOAS it reads the configuration files, and generates the endpoints:

```
2017-04-23 10:56:46.258  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/connect/{providerId}],methods=[POST]}" onto public
org.springframework.web.servlet.view.RedirectView
org.springframework.social.connect.web.ConnectController.connect(java.lang.String,org.spring
framework.web.context.request.NativeWebRequest)
2017-04-23 10:56:46.259  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/connect/{providerId}],methods=[GET],params=[oauth_token]}" onto public
org.springframework.web.servlet.view.RedirectView
org.springframework.social.connect.web.ConnectController.oauth1Callback(java.lang.String,org
.springframework.web.context.request.NativeWebRequest)
2017-04-23 10:56:46.259  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/connect/{providerId}],methods=[GET]}" onto public java.lang.String
org.springframework.social.connect.web.ConnectController.connectionStatus(java.lang.String,o
rg.springframework.web.context.request.NativeWebRequest,org.springframework.ui.Model)
2017-04-23 10:56:46.260  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/connect],methods=[GET]}" onto public java.lang.String
org.springframework.social.connect.web.ConnectController.connectionStatus(org.springframewor
k.web.context.request.NativeWebRequest,org.springframework.ui.Model)
2017-04-23 10:56:46.260  INFO 9804 --- [main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/connect/{providerId}],methods=[GET],params=[code]}" onto public
org.springframework.web.servlet.view.RedirectView
org.springframework.social.connect.web.ConnectController.oauth2Callback(java.lang.String,org
.springframework.web.context.request.NativeWebRequest)

2017-04-23 10:56:46.830  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped "{[/api/v1/{repository}/{id}],methods=[GET],produces=[application/hal+json ||
application/json || application/*+json;charset=UTF-8]}" onto public
org.springframework.http.ResponseEntity<org.springframework.hateoas.Resource<?>>
org.springframework.data.rest.webmvc.RepositoryEntityController.getItemResource(org.springfr
amework.data.rest.webmvc.RootResourceInformation,java.io.Serializable,org.springframework.da
ta.rest.webmvc.PersistentEntityResourceAssembler,org.springframework.http.HttpHeaders)
throws org.springframework.web.HttpRequestMethodNotSupportedException
2017-04-23 10:56:46.831  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped "{[/api/v1/{repository}/{id}],methods=[PUT],produces=[application/hal+json ||
application/json || application/*+json;charset=UTF-8]}" onto public
org.springframework.http.ResponseEntity<? extends
org.springframework.hateoas.ResourceSupport>
```

```
org.springframework.data.rest.webmvc.RepositoryEntityController.putItemResource(org.springfr
amework.data.rest.webmvc.RootResourceInformation,org.springframework.data.rest.webmvc.Persis
tentEntityResource,java.io.Serializable,org.springframework.data.rest.webmvc.PersistentEntit
yResourceAssembler,org.springframework.data.rest.webmvc.support.ETag,java.lang.String)
throws org.springframework.web.HttpRequestMethodNotSupportedException
2017-04-23 10:56:46.832  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped "{[/api/v1/{repository}/{id}],methods=[HEAD],produces=[application/hal+json ||
application/json || application/*+json;charset=UTF-8]}" onto public
org.springframework.http.ResponseEntity<?>
org.springframework.data.rest.webmvc.RepositoryEntityController.headForItemResource(org.spri
ngframework.data.rest.webmvc.RootResourceInformation,java.io.Serializable,org.springframewor
k.data.rest.webmvc.PersistentEntityResourceAssembler) throws
org.springframework.web.HttpRequestMethodNotSupportedException
2017-04-23 10:56:46.834  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped "{[/api/v1/{repository}],methods=[POST],produces=[application/hal+json ||
application/json || application/*+json;charset=UTF-8]}" onto public
org.springframework.http.ResponseEntity<org.springframework.hateoas.ResourceSupport>
org.springframework.data.rest.webmvc.RepositoryEntityController.postCollectionResource(org.s
pringframework.data.rest.webmvc.RootResourceInformation,org.springframework.data.rest.webmvc
.PersistentEntityResource,org.springframework.data.rest.webmvc.PersistentEntityResourceAssem
bler,java.lang.String) throws org.springframework.web.HttpRequestMethodNotSupportedException
2017-04-23 10:56:46.835  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped
"{[/api/v1/{repository}],methods=[GET],produces=[application/x-spring-data-compact+json ||
text/uri-list]}" onto public org.springframework.hateoas.Resources<?>
org.springframework.data.rest.webmvc.RepositoryEntityController.getCollectionResourceCompact
(org.springframework.data.rest.webmvc.RootResourceInformation,org.springframework.data.rest.
webmvc.support.DefaultedPageable,org.springframework.data.domain.Sort,org.springframework.da
ta.rest.webmvc.PersistentEntityResourceAssembler) throws
org.springframework.data.rest.webmvc.ResourceNotFoundException,org.springframework.web.HttpR
equestMethodNotSupportedException
2017-04-23 10:56:46.843  INFO 9804 --- [main] o.s.d.r.w.RepositoryRestHandlerMapping   :
Mapped "{[/api/v1/{repository}],methods=[OPTIONS],produces=[application/hal+json ||
application/json || application/*+json;charset=UTF-8]}" onto public
org.springframework.http.ResponseEntity<?>
org.springframework.data.rest.webmvc.RepositoryEntityController.optionsForCollectionResource
(org.springframework.data.rest.webmvc.RootResourceInformation)
```

While these components are not user defined and come bundled with Spring, they are still crucial to the application. After all the steps illustrated above are completed, the application is loaded and can start to receive requests.

4.2.3.2. Data Flow

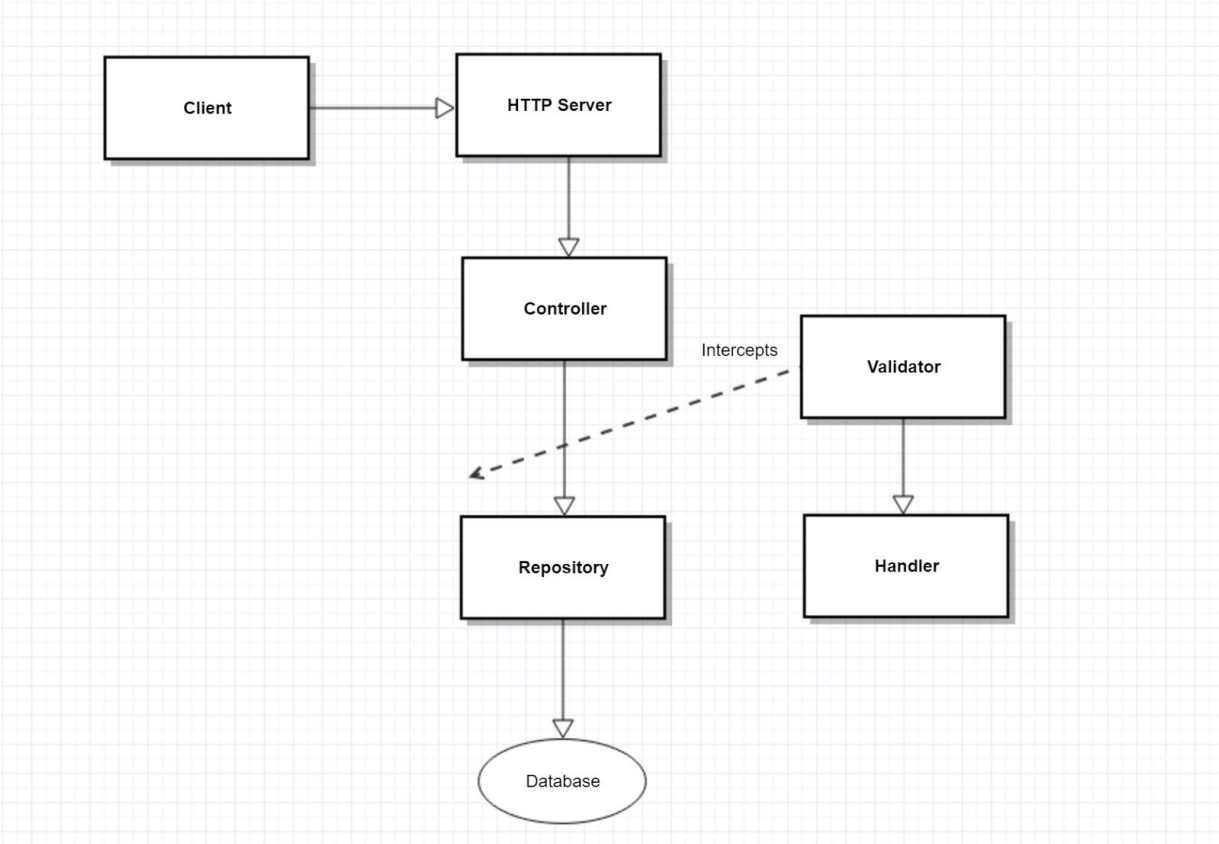


Fig. 4.2.3.2.1: Data flow in the back end of the application

Once a request is made, the associated controller is called. If the request is a POST or a PUT, the entity is deserialized and sent to the persistence layer to be stored in the database. Here is where the validator intercepts the requests and checks if all the member variables of the entity follow the constraints that are defined. An example of this can be observed in the *Donation* entity class, where beside the Hibernate annotations for persistence, validator annotations can be found, such as `NotNull` or `Min`. If one of these constraints is violated, an error message is sent back and the entity is not persisted.

In the example below, a constraint is defined for every field that states that it cannot be null (except for the `donatedOn` field which is set via the handler) and that the donation amount must always be positive. This is applied to all the actions that create or change an entity:

```
public class Donation {
	@Id
	private String id;
	@Indexed
	@DBRef
	@NotNull
	private User user;
	@NotNull
	@Min(value=0L)
	private Double amount;
	private LocalDateTime donatedOn;
	@Indexed
	@DBRef
	@NotNull
	private Campaign campaign;
	@NotNull
	private DonationStatus status;
	…
}
```

A snippet of the donation handler is copied below:

```
@HandleBeforeCreate
public void validateAndCreate(Donation donation){
	donation.setDonatedOn(LocalDateTime.now());
}
```

The `HandleBeforeCreate` annotation dispatches to the handler the instruction to run the code illustrated below before the entity is persisted:

```
@HandleAfterCreate
@HandleAfterSave
public void recalculatePledges(Donation donation){
	Campaign campaign = donation.getCampaign();
	List<Donation> campaignDonations = dRep.findByCampaignId(campaign.getId());
	campaign.setCurrent(campaignDonations.stream().mapToDouble(x ->x.getAmount()).sum());
	List<Donation> userDonations = dRep.findByUserId(donation.getUser().getId());
	User user=donation.getUser();
	user.setTotalDonated(userDonations.stream().mapToDouble(x -> x.getAmount()).sum());
		uRep.save(user);
		cRep.save(campaign);
	}
```

In this scenario, the code illustrated above recalculates the values for the amounts pledged per *User* and per *Campaign* after a Donation is added or modified.

4.2.4. Conclusions

Using the tools and technologies described above, a completely functional prototype for a crowdfunding web platform was developed. The back end allows for CRUD operations validating data and running statistics. Since MongoDB can be used as a distributed database, it allows the platform to be scalable. Standardising the REST API through HATEOAS allows integration with third party services and makes the front end development platform-independent. Using Hibernate as an ORM framework makes the platform future-proof. Because the persistence layer is developed with high-level APIs, the database system could be replaced at any time with any other newer, “state of the art”, more performant solution such as CassandraDB that offers an API closer to MySQL and is also used in high activity environments. The Spring Framework is supported and continuously developed by an active open-source community which further enables the platform to be future-proof through the ability to add all the newly implemented features that will always be backwards compatible with our implementation.

5. Testing and evaluation

This chapter describes how the system was evaluated in order to check if it met the key objectives outlined in Section 1.2 and the functional requirements set in Section 3.1 . The usability and attractiveness of the front end microservice was evaluated using SUS (Simple User Survey), the back end performance and scalability was evaluated using JMeter and the functionality (correctness of the CRUD operations as well as data manipulation) of the back end was evaluated using unit tests. After the front end usability and attractiveness evaluation, this chapter also contains brief statistics of the demographics of the respondents to the survey.

5.1. Front end usability and attractiveness evaluation

The usability and attractiveness of the designed Graphical User Interface and front end implementation was tested through distributing the Questionnaire in Appendix 2 to potential users.

The survey respondents were selected to confirm with the chosen positioning of the platform among socially responsible conscious millennials with a potential interest in startups and business ideas. For this reason, the survey was distributed via social media channels where the primarily users are university students. The sampling method used was convenience sampling. The following channels were used to reach respondents:

- social media groups for students;
- web development forums.

5.1.1. Part 1: User Interface

Question 1:

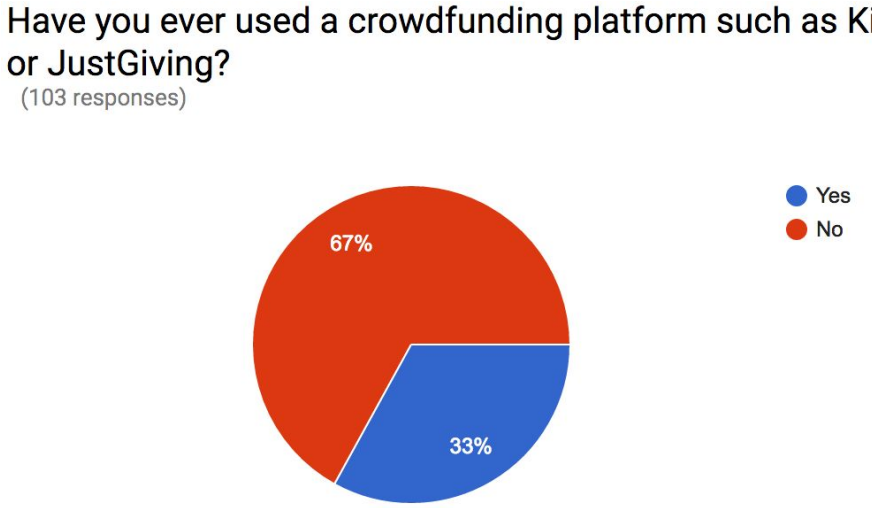


Fig. 5.1.1.1: Question 1 of front end evaluation questionnaire

The analysis indicates that an overall low usage score for crowdfunding platforms. A substantial number of respondents (67%) admitted they never used a crowdfunding platform. When considering the demographic data (illustrated in *Fig. 5.1.1.1*), it looks like the young population that is only now starting to discover crowdfunding platforms for civic projects.

Question 2:

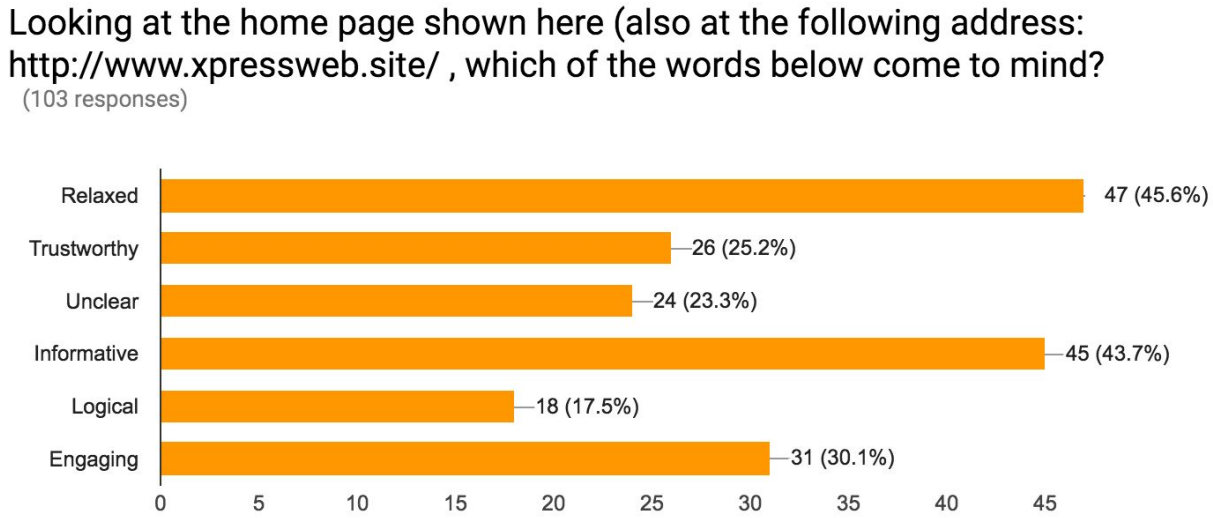


Fig. 5.1.1.2: Question 2 of front end evaluation questionnaire

This question aimed to identify simple associations made with the proposed design of the platform. Around half of the respondents consider the page has a relaxing effect (45.6%), it is informative (43.7%), and engaging (30.1%). On the other hand, less than a fifth of the respondents think the page is logical (17.5%) and less than a quarter said the page is overall unclear (23.3%) – which may indicate the structure of the page and the content could be modified to be more accessible and comprehensible. This, in turn, may make the page look more trustworthy, as for now only 25.2% of respondents consider it as such.

Question 3:

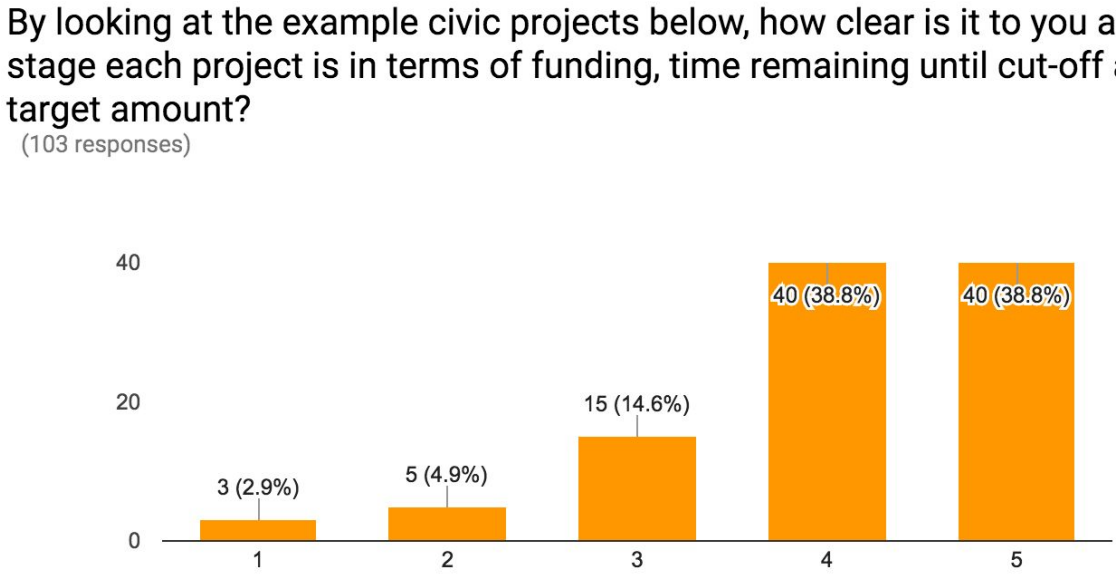


Fig. 5.1.1.3: Question 3 of front end evaluation questionnaire

In the figure illustrated above, 1 corresponds to the response *Very Unclear* and 5 corresponds to the response *Very Clear*.

Having a clear and slick user experience is a crucial requirement for the civic crowdfunding platform I proposed. My analysis of the current scene of crowdfunding websites indicates that the way projects are presented is not always clear and consistent. Thus, crucial information on business ideas can be lost in the process. My main target is to ensure a smooth experience for the end user and provide an objective overview of the advertised projects.

Question 4:

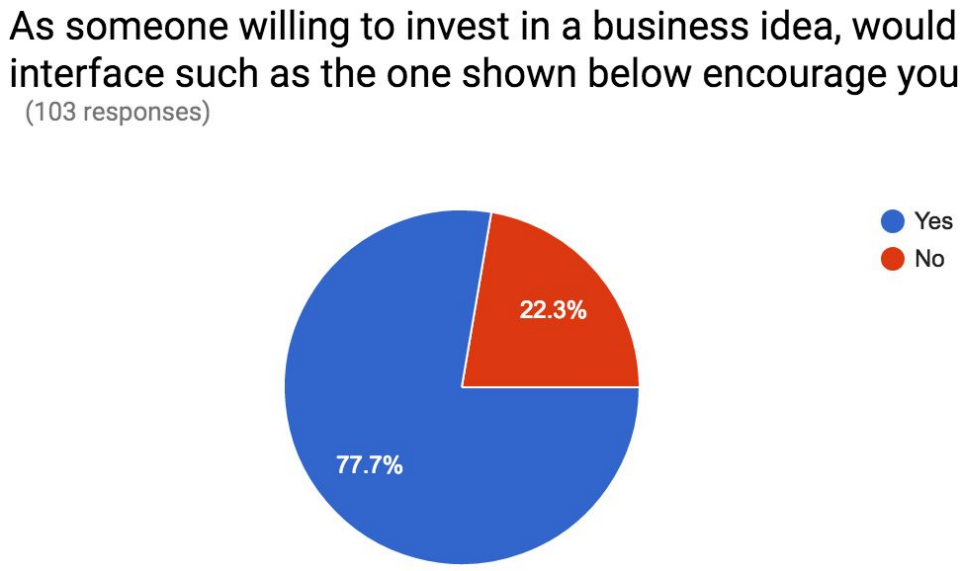


Fig. 5.1.1.4: Question 1 of front end evaluation questionnaire

This question looks at the potential for user conversion. A high percentage of respondents, over 75%, admitted to potentially wanting to invest in a business idea presented on the platform. The question emphasized the end user experience of the platform as a factor for deciding to invest, rather than the content in the business idea presented. This question does not look at the reasons why the respondents opted to potentially invest. However, the results from the other questions indicate that

respondents' willingness to invest may be related to the clear interface and a relaxed tone of the overall platform. Also, the fact that each project idea has a clear overview of its stage, as well as its community support is likely to make sceptical users to convert.

Question 5: Please let us know any other thoughts regarding the User Interface

In this question, respondents submitted their own comments about any other areas of the overall interface and their user experience.

Positive feedback included:

- "more info" - It looks like some respondents would like more detailed information on each project. However, the data provided for each business idea was limited in the questionnaire. Once the actual platform is in place, users will be able to see more information as well as other type of particular insights that will only be generated once a higher number of business ideas are submitted;
- "clear interface/ very clear, condensed information"- a significant number of respondents agreed on the clear layout of the interface and the availability of information. This confirms and informs my design going forward in the project.

Other responses included:

- "Works fine on mobile too";
- "on point, intuitive";
- "Good and structured way to give a brief overview of the projects and their status.";
- "Catchy images";
- "I find this interface very user friendly and clear";
- "They seem engaging, especially since they have children in the main picture, they raise your interest more.".

Negative feedback included:

- "The key information does not seem to have been highlighted.";
- "Would look better in a simplified version".

Overall, this question confirmed the interface design choices and allowed gathering constructive feedback from the survey respondents. The general iterations based on this question will be minor, and later the changes made as a result of this will be specified.

5.1.2. Part 2: Trust

Question 6:

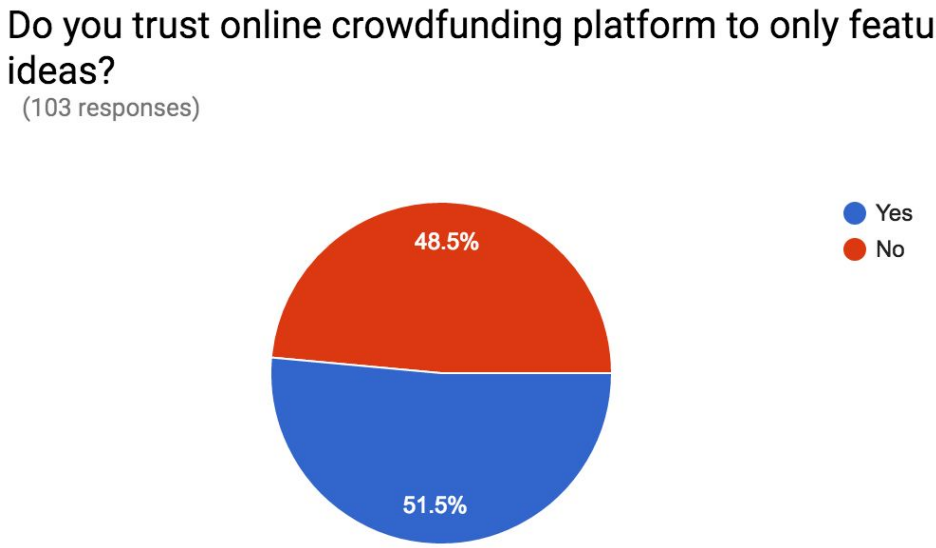


Fig. 5.1.2.1: Question 5 of front end evaluation questionnaire

The first question from the trust section looked at user trust levels towards crowdfunding platforms in general (not specific to our platform), by enquiring about their belief in the platform's ability to only showcase legit business ideas. The survey respondents were almost equally split for this question, equalling to 48.5% not trusting these platforms to showcase good ideas, and 51.5% putting their trust into them. This divide may be a beneficial factor for the current platform - XPressStarter, as a significant proportion of the less trusting audience may switch to a platform that ensures they are only shown the best and brightest civic project ideas.

Question 7:

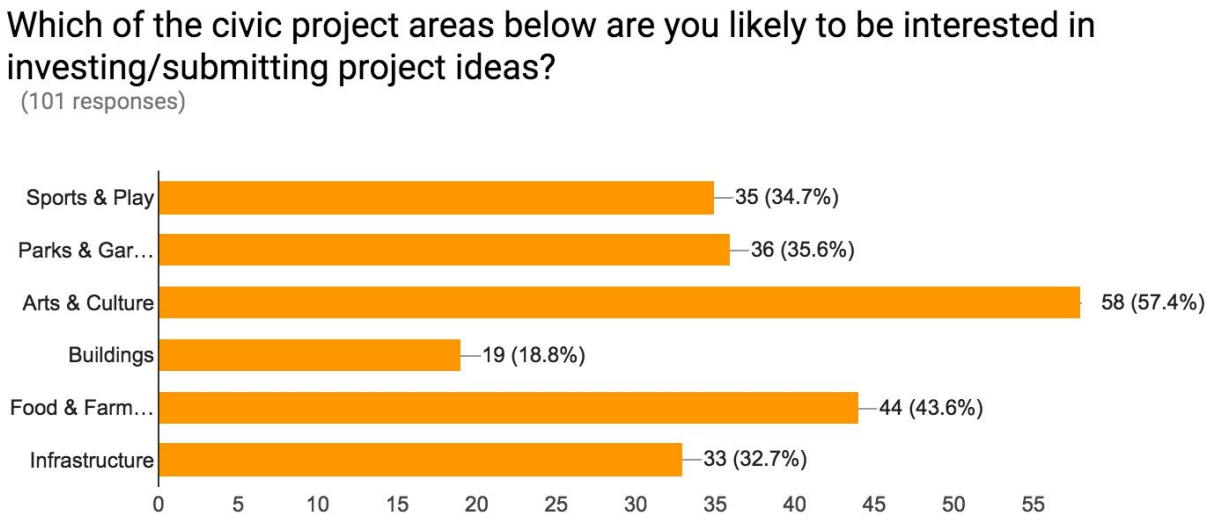


Fig. 5.1.2.2: Question 6 of front end evaluation questionnaire

The options presented in this question are categories already planned to be part of the civic platform. Particularly, arts and culture seems to be the most popular theme among the respondents, with 57.4% of respondents interested in this, closely followed by Food & Farm with another 43.6%. These results will help in prioritising the exposure of various categories of projects within the website, potentially using the main page to highlight the projects from the Arts area better.
Then, parks & gardens, sports & play, and infrastructure seem to attract almost an equal amount of respondents' interest – 35.6%, 34.7%, and respectively 32.7% of people showing interest in these areas. The least amount of interest was shown in the buildings theme, with less than a fifth of respondents choosing this option.

Question 8:

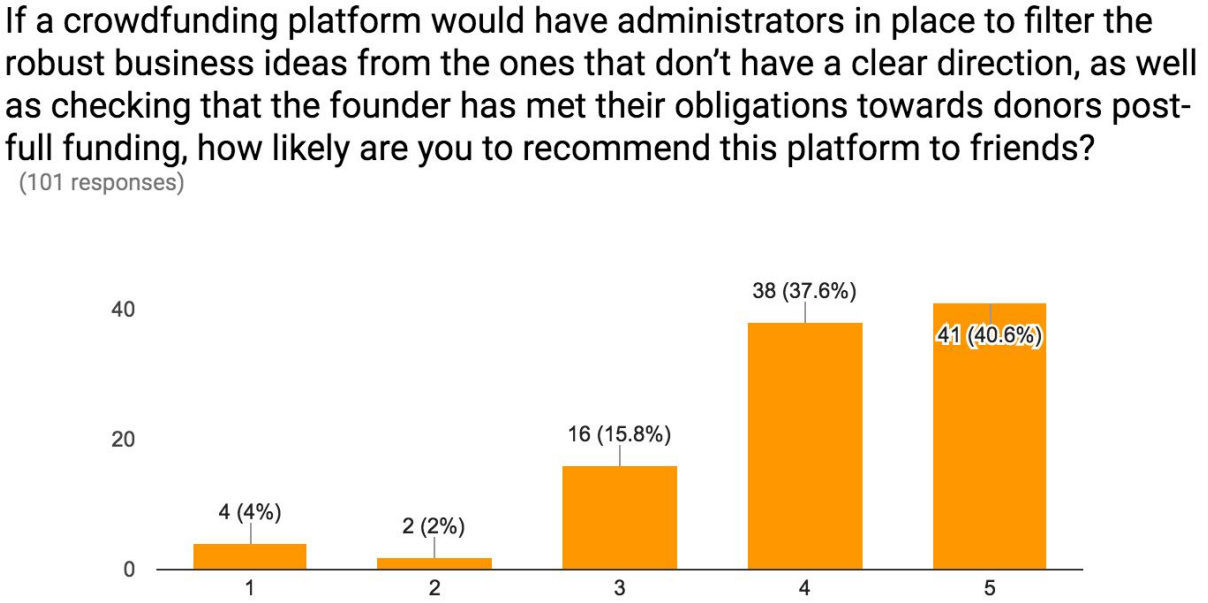


Fig. 5.1.2.3: Question 6 of front end evaluation questionnaire

In the figure illustrated above, 1 corresponds to the response *Very Unlikely* and 5 corresponds to the response *Very Likely*.

The results collected for question 8 indicate a higher likelihood of recommending the platform to friends and family if a network administrator was in place. While this

question is not necessarily linked to the direct user design, it nonetheless confirms the assumption that the admin offering would encourage more users to join the platform as either founders or benefactors.

5.2. Questionnaire demographics

A series of demographic questions were included in the Questionnaire in Appendix 2 to portray the degree to which the platform is matching the expectations of the target audience. This section presents a selection of those, prioritizing on those that meet the segmentation process for the platform best.

As XpressStarter is primarily targeting young individuals with a desire to do something for overall social benefit, it was unsurprising to see the majority of respondents are in the 18-24 age category. This is correlated with the results from the next question, which looks at occupations.

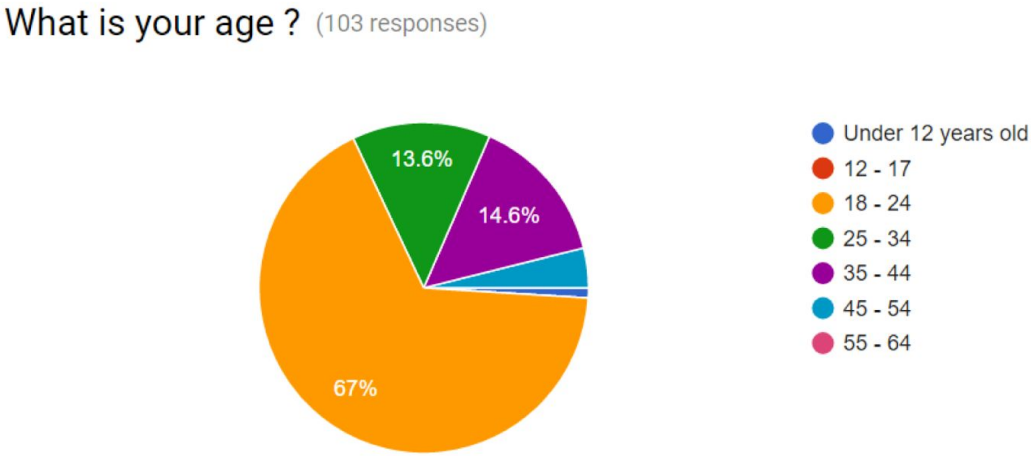


Fig. 5.2.1: The age of the questionnaire audience

A large proportion of respondents were students, since the survey was distributed on student forums and social media. While students are less likely to have the disposable income to invest in civic business ideas, they could be potential founders. Targeting this audience via the survey indicates that there is a pool of talent that is eager to engage in a new type of civic platform.

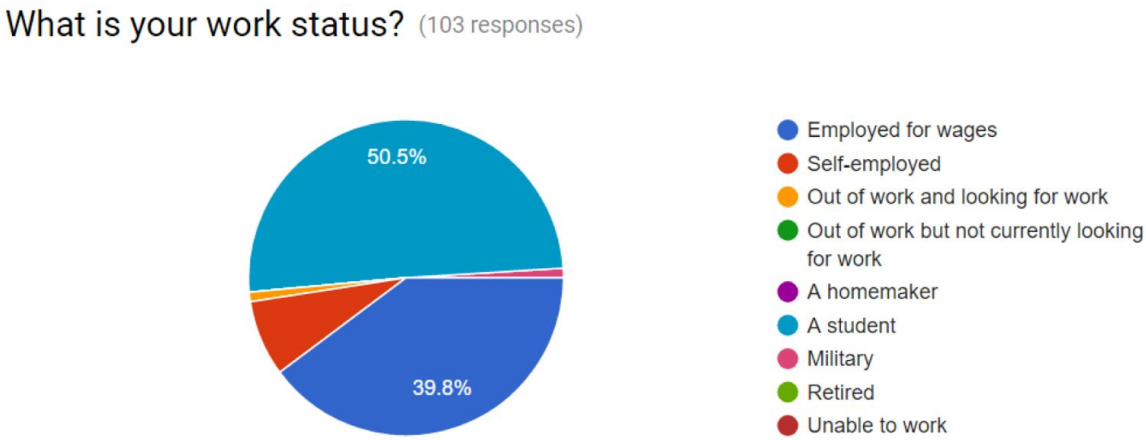
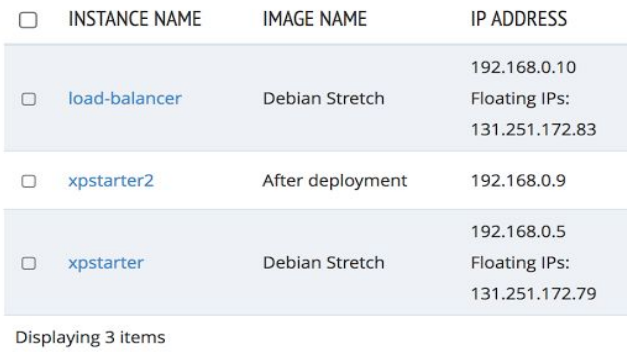


Fig. 5.2.2: The work status of the questionnaire audience

5.3. Back end performance and scalability testing

5.3.1. Introduction

The purpose of this test was to see if the application is horizontally scalable. Using the OpenStack account provided, a basic environment of 2 application servers was set up: **`xpstarter`** and **`xpstarter2`** and a load balancer, named **`load-balancer`**. The summary setup in OpenStack of these instances is shown in the image below.



	INSTANCE NAME	IMAGE NAME	IP ADDRESS
☐	load-balancer	Debian Stretch	192.168.0.10 Floating IPs: 131.251.172.83
☐	xpstarter2	After deployment	192.168.0.9
☐	xpstarter	Debian Stretch	192.168.0.5 Floating IPs: 131.251.172.79

Displaying 3 items

Fig. 5.3.1.1: The OpenStack instances of the application

To demonstrate that the application is horizontally scalable, two instances were needed to show that they can receive simultaneous requests, ensuring high availability (reliability) and increased performance, as the processing power doubles through allowing the instances to split the requests among them using the load balancer.

The load balancer runs an nginx server that balances the load between the application servers, proxying the requests to them at the same time. Both instances needed to connect to the same database in order to ensure that they retrieve the same data.

5.3.2. Database

If more than one back end server is used, a way to make sure both servers can access the same data was needed. MongoDB offers the possibility of using a distributed database system and configure it as a cluster, this ensuring the horizontal scaling of the application.

The full reasoning behind using MongoDB as the back end database is outlined in 4.2.1.2: Brief justification of technologies used - MongoDB .

MongoDB clusters require a *Configuration Server*, a *Query Resolver* and at least one *Shard Server*. The *Configuration Server* stores metadata related to the data stored in the database. As the information is stored into clusters, this is needed map the data to its location in the cluster. The *Query Resolver* uses the *Configuration Server* to act as an interface between clients and the cluster, routing queries to where the data is located. The data sent through the *Query Resolver* is stored on the *Shard Server*, which splits the actual data. The following diagram illustrates the components that are part of the MongoDB cluster:

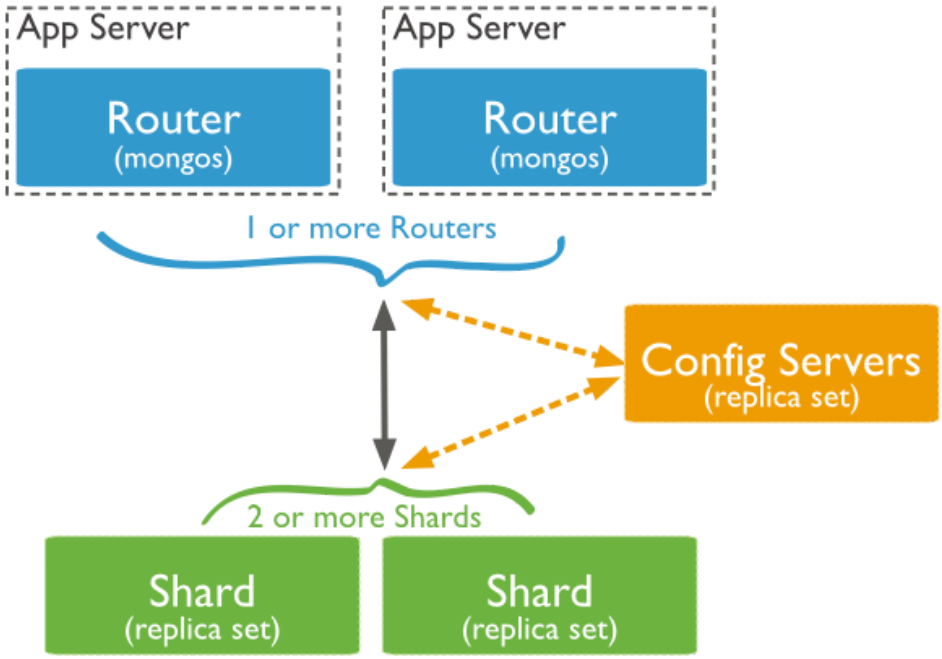


Fig. 5.3.2.1: The components of a MongoDB cluster

A MongoDB cluster configuration can handle multiple instances of its components (Configuration Servers, Query Resolvers and Shard Servers) to make the application scalable, ensuring that there is no single point of failure. Metadata is replicated between *Configuration Servers*, allowing different *Query Resolvers* to use different *Configuration Servers*, increasing performance and reliability. This way, if a *Configuration Server* becomes unavailable, another one can be used.

In the particular scenario of our test, each application server contains a Sharding server, and the *Query Resolver* and *Configuration Server* were setup on the load-balancer machine. The cluster creation process followed the guidelines of the official MongoDB documentation ([46] MongoDB Documentation). The output of the `sh.status()` command, that outputs the status of the MongoDB sharded cluster is shown below:

```
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("58eff1c78b1c93f722f19a29")
}
  shards:
        {  "_id" : "shard0000",  "host" : "192.168.0.9:27017" }
        {  "_id" : "shard0001",  "host" : "192.168.0.5:27017" }
  active mongoses:
        "3.2.11" : 1
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "XpressStarter",  "primary" : "shard0000",  "partitioned" : false }
```

In the configuration files of the back end application the *Load Balancer* was used as the MongoDB server, which meant that both servers were using the *Query Resolver*.

As it can seen in the output of the command `sh.status()`, the database is stored on `shard0000`, which is one of the servers. Partitioning the database over the servers falls outside the scope of this test, so it was considered safe to be left in its current state. In a production environment the database would be partitioned and with a replica set. Fortunately, Hibernate knows how to work with MongoDB Cluster, so no code had to be written for it to work. As it can be observed from the output below generated through running our back end application jar file on **`xpstarter1`**, the sharded cluster was successfully discovered by Java and used by our back end application, pointing to the IP address of the instance where the query resolver resides (`192.168.0.10`):

```
2017-04-25 20:51:15.671 INFO 6505 --- [192.168.0.10:27017] org.mongodb.driver.cluster
:Monitor thread successfully connected to server with description
ServerDescription{address=192.168.0.10:27017, type=SHARD_ROUTER, state=CONNECT
ED, ok=true, version=ServerVersion{versionList=[3, 2, 11]}, minWireVersion=0,
maxWireVersion=4, maxDocumentSize=16777216, roundTripTimeNanos=365896}

2017-04-25 20:51:15.672 INFO 6505 --- [192.168.0.10:27017] org.mongodb.driver.cluster
:Discovered cluster type of SHARDED
```

As outlined in the highlighted sections of the output above, our application has detected that it has connected to a *Query Resolver* and discovered that the cluster is of type `SHARDED` and not a single instance as an ordinary MongoDB connection.

5.3.3. HTTP Requests

Having a shared database, a way to split requests between the web servers was needed. Using nginx,a weighted load-balancer was configured between the two application servers. Since **`xpstarter2`** was holding the database, it would receive all the reads, so I needed to make sure it receives half as much requests as the other server, to evenly spread the load. Nginx had to listen on both ports (80 - the port on which the front end runs and 8080 - the port on which the back end runs). A list of upstream servers was configured using weights in order to distribute the requests in a way such that for each 3 requests the **`xpstarter1`** receives 2 requests and **`xpstarter2`** receives 1 request.

The front end load-balancer configuration is outlined below:

```
debian@load-balancer:~$ cat /etc/nginx/conf.d/load-balancer.conf
# Define which servers to include in the load balancing scheme.
# It's best to use the servers' private IPs for better performance and security.
# You can find the private IPs at your UpCloud Control Panel Network section.

upstream front end {
   server 192.168.0.5 weight=2;
   server 192.168.0.9 weight=1;
}

# This server accepts all traffic to port 80 and passes it to the upstream.
# Notice that the upstream name and the proxy_pass need to match.

server {
   listen 80;
   location / {
      proxy_pass http://front end;
   }
}
```

The back end load-balancer configuration is outlined below:

```
debian@load-balancer:~$ cat /etc/nginx/conf.d/back end-load-balancer.conf
# Define which servers to include in the load balancing scheme.
# It's best to use the servers' private IPs for better performance and security.
# You can find the private IPs at your UpCloud Control Panel Network section.

upstream back end {
   server 192.168.0.5:8080 weight=2;
   server 192.168.0.9:8080 weight=1;
}

# This server accepts all traffic to port 80 and passes it to the upstream.
# Notice that the upstream name and the proxy_pass need to match.

server {
   listen 8080;
   location / {
      proxy_pass http://back end;
   }
}
```

The configuration files pasted above show the weights of the upstream servers used in the load balancing process (weight 2 for **`xpstarter1`** and weight 1 for **`xpstarter2`**). In both configuration files, nginx is configured to listen on ports 80 and 8080 respectively and then proxy the requests to one of the 2 upstream servers (**`xpstarter1`** or **`xpstarter2`**).

5.3.4. Testing the load balancer

First thing after configuring the load balancer, some testing is required. For a simple test I just issued a `wget` command to make sure I was reaching the front end:

```
debian@load-balancer:~$ wget http://localhost
--2017-04-25 20:42:04--  http://localhost/
Resolving localhost (localhost)... 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 15530 (15K) [text/html]
Saving to: 'index.html.1'
index.html.1        100%[===================>]  15.17K  --.-KB/s    in 0s
2017-04-25 20:42:05 (443 MB/s) - 'index.html.1' saved [15530/15530]
```

The output of the `wget` command shown above demonstrates that it was executed successfully and proves that the `index.html` file from the front end can be retrieved from the load balancer, therefore proving that the requests are reaching one of the application servers.

Everything is setup. Now a JMeter test plan was set up to make sure that the server can handle a theoretical load. I used the `htop` utility to see the resource usage on the three servers. It has a nice and colorful interface as opposed to the top utility.

5.3.5. Testing using JMeter

JMeter uses test plans that it applies for a number of threads (users).

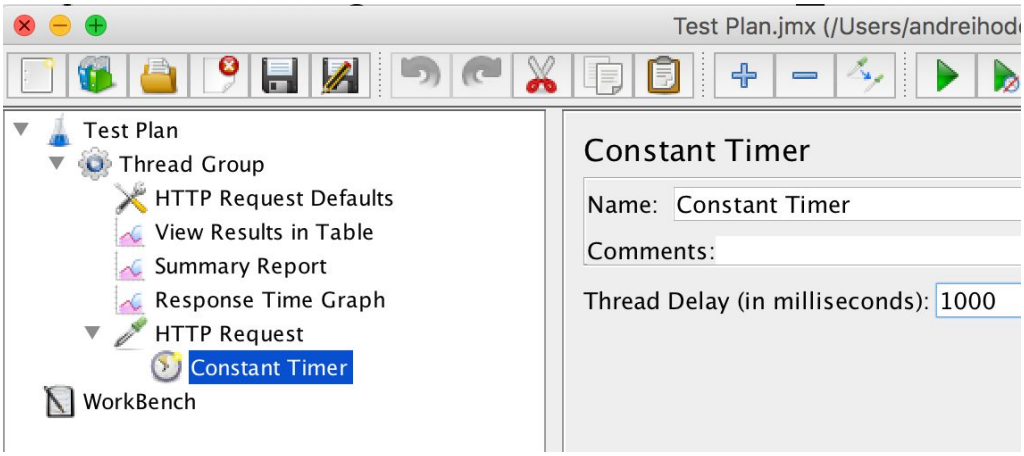


Fig. 5.3.5.1: JMeter timer configuration

The test plan is comprised of different actions and actions used in the test. For this test the back end was queried for campaigns. I started with 10,100,1000 users with a second between requests (this time is applied per thread).

5.3.5.1. Writing a test plan

In order to test how the back end would respond to a certain number of users, a test plan was created that initiates a request that displays the first 5 campaigns. This link would be accessed by a user who accesses the web platform and checks the list of active campaigns. A thread group has been created. The default options (headers) for an HTTP request were configured and the HTTP request was added in the thread group.

Below there is a screenshot that shows how the HTTP requests for the thread were configured:

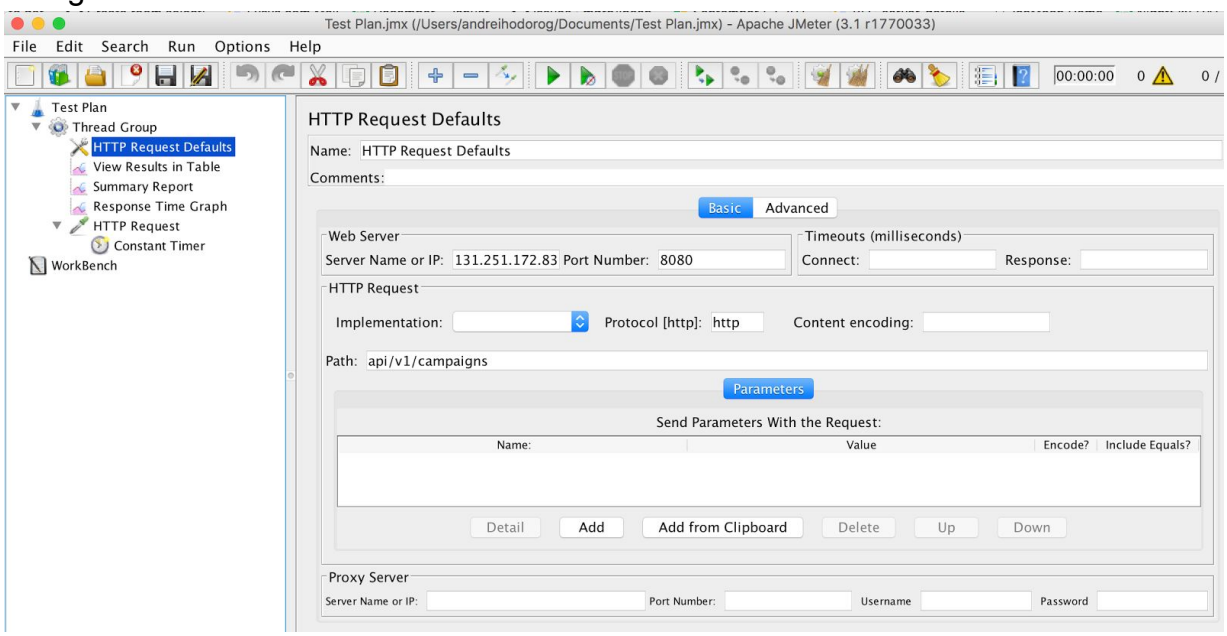


Fig. 5.3.5.1.1: HTTP Requests configuration

Below there is a screenshot that shows the thread group, with the setup of threads that loop forever until the test is stopped.

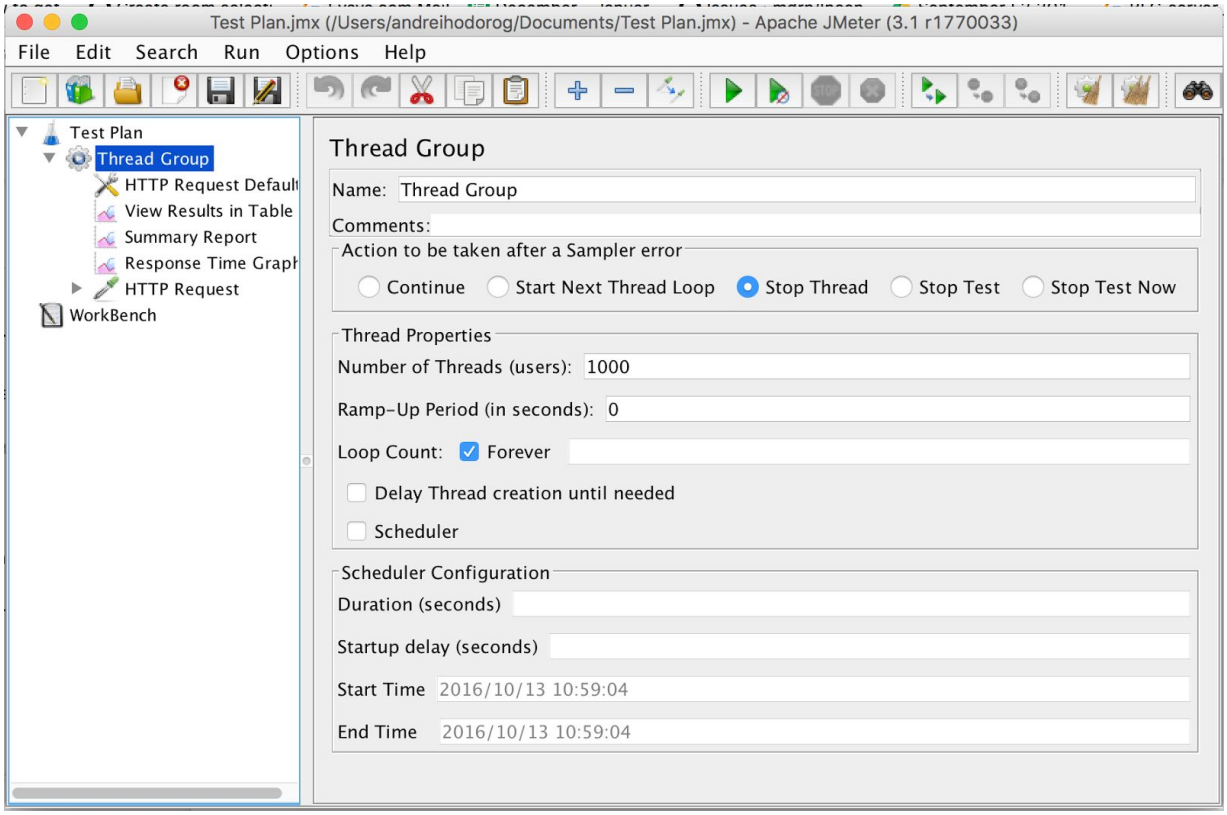


Fig. 5.3.5.1.2: JMeter thread group

In order to see the results of our tests, 3 different views were added that can help us to see the results in different formats. After conducting the tests, the summary report view was considered the most appropriate, as it shows an average response time per request, as well as a minimum and a maximum response time.

The screenshot below shows the table header for the summary report.

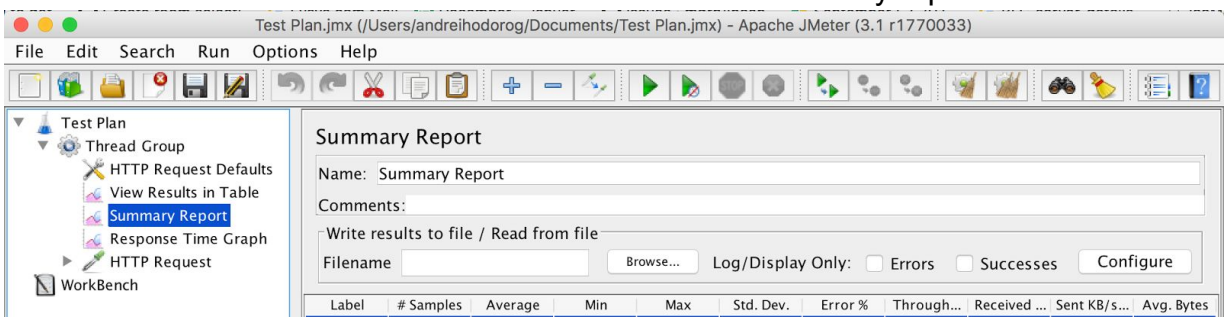


Fig. 5.3.5.1.3: JMeter summary report

5.3.5.2. Emulation of 10 Users (44 ms average)

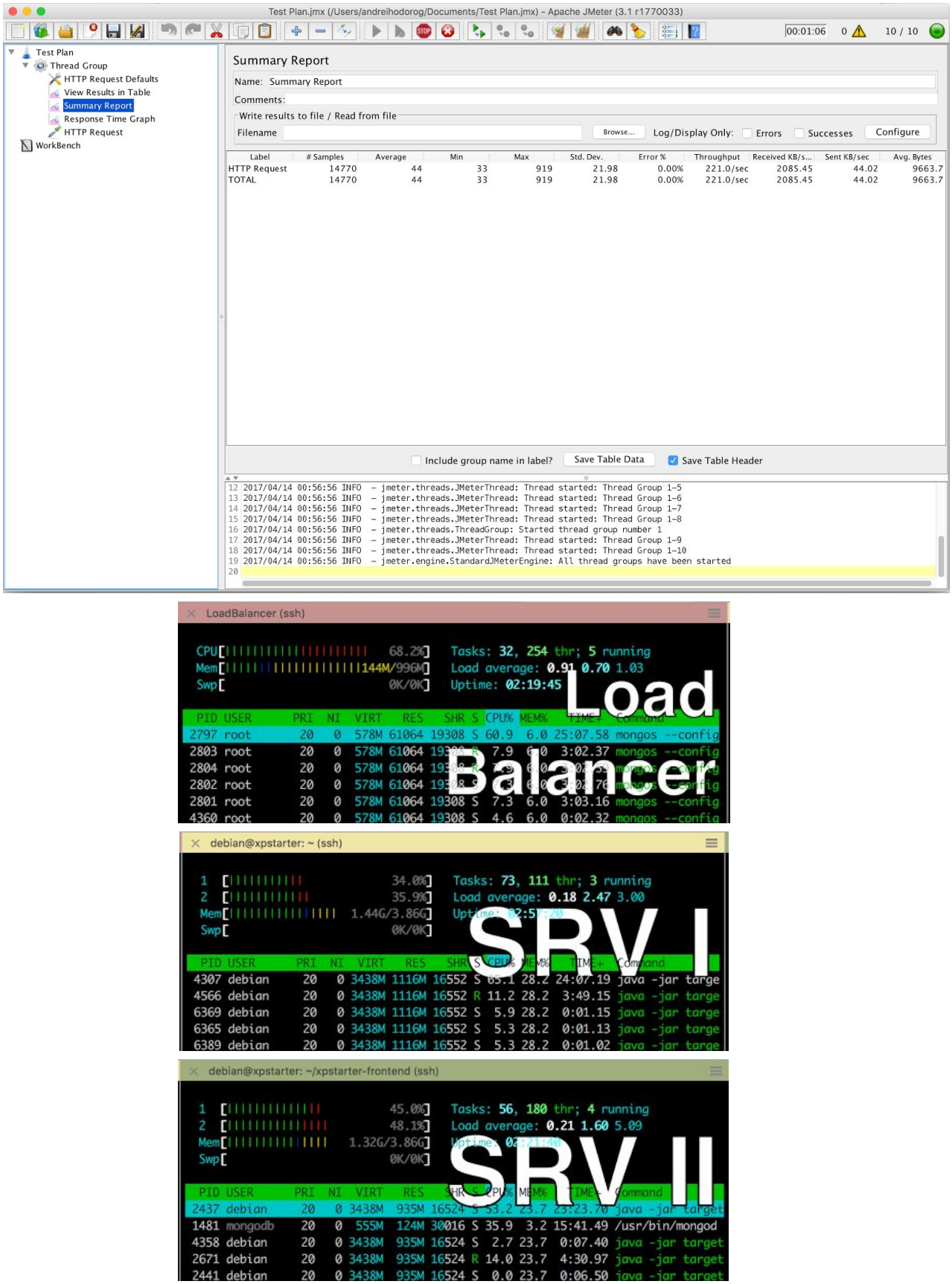


Fig. 5.3.5.2: HTOP utility output - JMeter Emulation of 10 users and HTOP statistics

As it can be observed from the figures above, for the activity generated by 10 users, both on server 1 (SRV1) and server 2 (SRV2), the load per cores varies between 30% and 40%. The servers are not overloaded with this amount of users. The fact that on

the load balancer the load is 70% can be observed (please note that the load balancer has only one core). From the summary report generated by JMeter, after 14770 requests, the average response time is 44 milliseconds.

5.3.5.3. Emulation of 100 Users (144ms average)

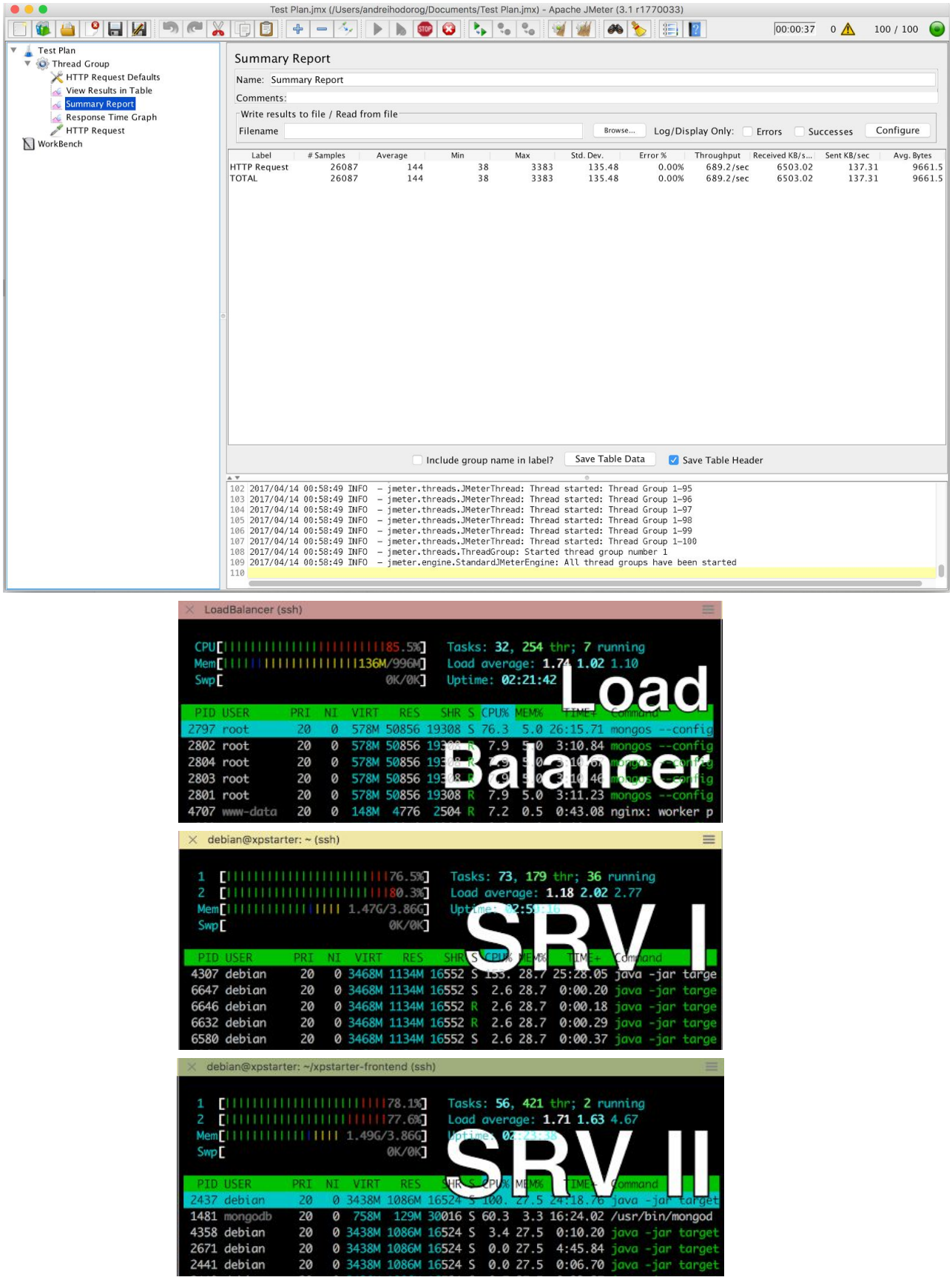
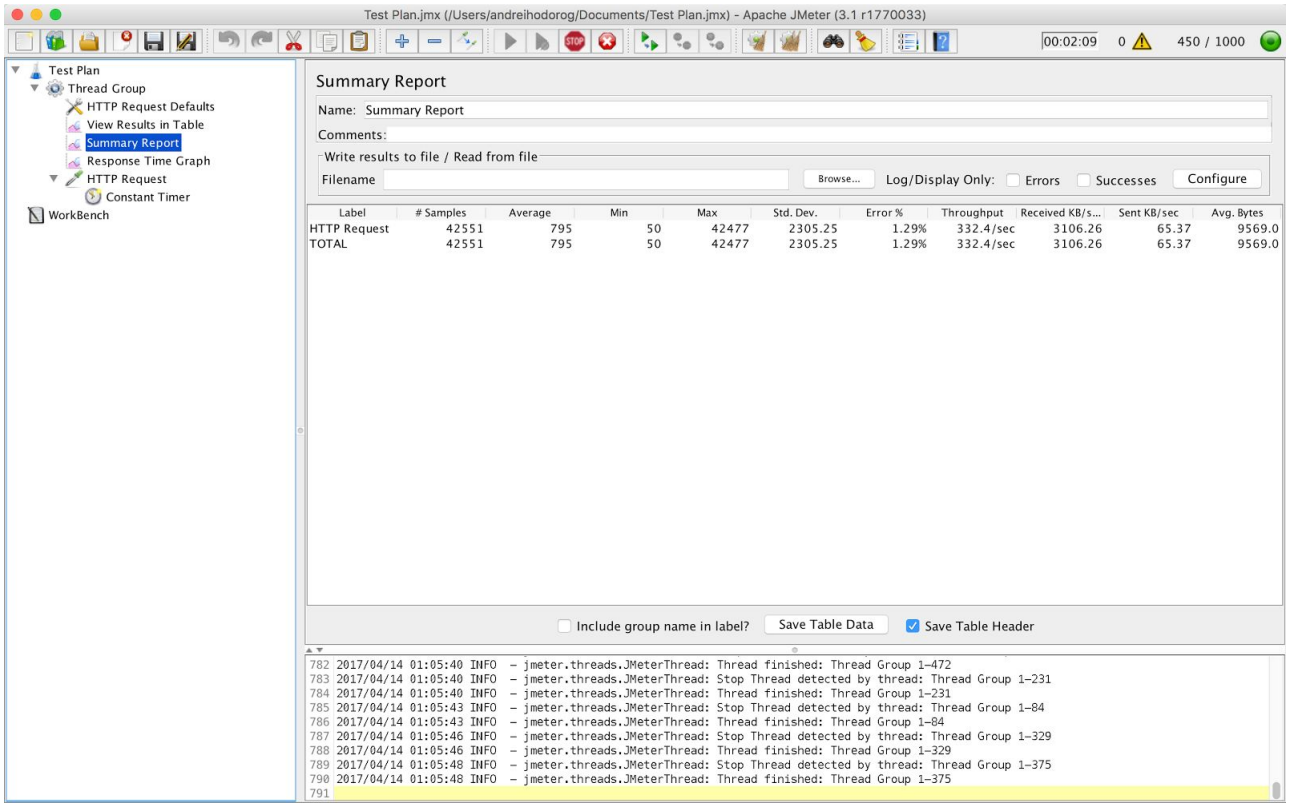


Fig. 5.3.5.3: HTOP utility output - JMeter Emulation of 100 users and HTOP statistics

As it can be observed from the figures above, for the activity generated by 100 users, both on server 1 (SRV1) and server 2 (SRV2), the load per cores varies between 75% and 80%. Such a load is ideal, as a typical load of a server handling traffic efficiently needs to average at 70% ([47] Ben Yemini, 2014), as the resources of a server need to be used efficiently, while leaving room for surges in activity. The fact that on the load balancer the load is 85% can be observed. From the summary report generated by JMeter, after 26087 requests, the average response time is 144 milliseconds, which still acceptable, as any response time under 200 milliseconds, which is considered to be “instant” and does not impact the user experience in any way ([48] Klaus Enzenhofer, 2016).

5.3.5.4. Emulation of 1000 Users (795 ms)



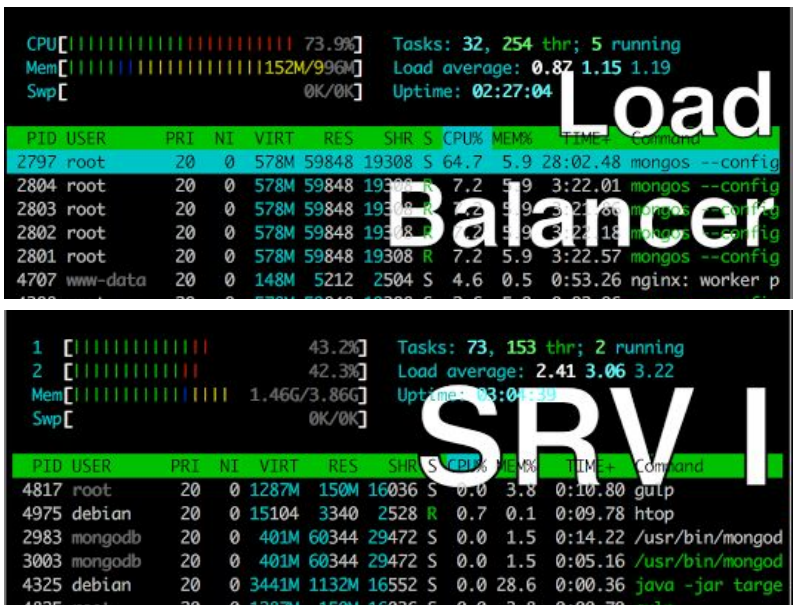


Fig. 5.3.5.4: HTOP utility output - JMeter Emulation of 1000 users and HTOP statistics

As it can be observed from the figures above, for the activity generated by 1000 users, both on server 1 (SRV1) and server 2 (SRV2), the load per cores varies between 43% and 52%. The servers are not overloaded with this amount of users. The fact that the load balancer experiences surges between 70% and 100% can be observed.

The undervolted processor in my laptop was having a hard time running 1000 threads, so in the upper right corner of the JMeter screen it can be seen that only 450 threads are running. The rest of the threads crashed. This is the reason why an error percentage is present, but as the test ran, the percent was going down. It can be seen that for 517 Threads, accessing the link continuously, an average time of 795 ms was achieved.

Since the load on the servers is below what it was with 100 users, the data generated should not be considered an accurate representation of an activity of 1000 simultaneous users.

5.3.6. Conclusions

The reason of this test was to prove that the application is horizontally scalable. For the database scalability was achieved through clustering and sharding. MongoDB was easy to configure as a cluster and the application recognized the cluster setup. The back end and front end services were installed on two servers to simulate a multi-server setup. Using nginx as a loadbalancer split the requests between the servers. Running test with JMeter showed that the load on the server hosting nginx and MongoDB query resolver was manageable as it handled 10/100/1000 requests per second without needing more than 1 core. For safety reasons in a production environment, a more capable machine should be used. Once the system was load balanced some tests with jMeter needed to be run to see what would the load be like on the servers. The test was in three stages, 10 Users, 100 Users, 1000 Users. The test plan did not include a delay, so it was simulating users constantly refreshing the page. The 10 users test averaged around 44ms response time, which is quite good. As it can be observed from the screen shots, the load was around 30%. With 100 users, the average time rose to around 144ms and the load on the servers reached around 70%. Simulating 1000 users was hard to do because jMeter would not keep the 1000 threads alive. The average request took around 700ms with some errors. Looking at the servers, the load is around 40% on each server. Since the load was higher with 100 users, an issue might be jMeter itself not being able to keep threads alive or the lack of processing power of my laptop. It can be seen in the snapshots that the thread count was 571/1000. This result may be ignored as it does not offer any insight. An average response time of 144ms for 100 users on a two server setup with 2 cores and 4GB RAM (which is acceptable). Since the simulated users were constantly refreshing, this load could be produced by more than 100 users, but it is advisable to take into account the worst case scenario, which was demonstrated with

jMeter. To sum up, the application and services are horizontally scalable and the current setup of a single core load balancer and 2 dual-core application servers is fit for 100 very active users.

5.4. Back end jUnit testing

jUnit is a testing framework for Java. It allows testing of various portions of code to make sure that any changes made do not affect other features. The same with performance there are micro-tests, where testing can be done at method level, meso-tests where an entire class is tested or macro tests, where the whole system is tested. It relies on a known output for the given input, and if the known output does not match the output retrieved, the test fails and warns the user.

This framework is usually used with Mockito and Hamcrest, two other frameworks that extend the functionality of jUnit. Mockito allows the mocking of different components while Hamcrest offers some syntactical sugar to make the code more readable.

Tests need to be carefully crafted to ensure that the code coverage is high, reducing the possibility that crucial parts are compromised by a change without the user noticing. They are also the basis for a software development technique called test-driven development. It implies that test are written before the code. This offers some insight into design flaws even before beginning writing code. If code is hard to test, it is usually badly written, as writing tests beforehand forces the developer to split the software into testable modules, leading to loosely-coupled code.

Using jUnit with Spring framework offers some challenges, as the application needs to be fully loaded to test all the features. Spring's dependency and context injection only works when the whole application is loaded. Fortunately the annotation `@SpringBootTest` tells jUnit how to run the tests for this scenario.

To properly test the features of the back end, making HTTP calls was necessary and using MockMvc, allowed executing requests to the REST controllers.

On the scale of Richardson Maturity Level, the REST API uses HTTP verbs to specify various actions (GET for retrieval, POST for addition, PUT for modification, DELETE for deletion). In a less mature REST API level, these actions would have different URIs (`/api/v1/campaigns/<id>/delete`). Since the main role of the back end is to provide the data through the REST API, in the following sections, all the tests involve requests to the REST API routes to test all the components.

5.4.1. Testing if the REST Controllers were generated

A basic test that was needed was to check if the REST controllers are created and accessible. This is helpful as it can signal errors in configuration.

```
@Test
public void verifyIfAllRestControllersAreGenerated() throws Exception {
mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/campaigns").accept(MediaType.APPLICATION
_JSON))
        .andExpect(status().is(200));

mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/likes").accept(MediaType.APPLICATION_JSO
N))
        .andExpect(status().is(200));
```

```
mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/donations").accept(MediaType.APPLICATION
_JSON))
        .andExpect(status().is(200));

mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/users").accept(MediaType.APPLICATION_JSO
N))
        .andExpect(status().is(200));
}
```

Testing this involved doing a GET request to each of the endpoints URIs (*Campaigns*, *Likes*, *Donations*, *Users*) and checking that the return code is `200 OK`. MockMVC performs a request built by the `MockMvcRequestBuilder` class and then expects a `200` return code.

5.4.2. Repository testing

Another test that had to be done, was persisting an entity via the REST API. In order to test the addition of a user to the database through a POST request, a mock user was created, then serialized as JSON using the Spring provided `ObjectMapper`. `MockMvc` performs a POST request and sends the serialized object. The type of data must be specified in the `contentType` header value. A HTTP return code of 201 `CREATED` is expected and if not received the test fails. The user object is then searched in the repository and deleted after it is found.

```
@Test
public void verifyUserPost() throws Exception {
        User testUser = new User("Test","User","testUserPost@test.com"
                ,"ksdhfisd",false,LocalDateTime.now(),Role.ADMIN);
        mockMvc.perform(MockMvcRequestBuilders.post("/api/v1/users").contentType(MediaType.AP
PLICATION_JSON)
                        .content(om.writeValueAsString(testUser))
                        .accept(MediaType.APPLICATION_JSON))
                        .andExpect(status().is(201));
                uRep.delete(uRep.findByEmail("testUserPost@test.com"));
}
```

The same test had to be performed for campaigns as well. A challenge was providing the links to the user objects, as HATEOAS requires them instead of serialized versions. First the user object is created and saved in the repository, then a campaign object is created. Before posting, the serialized object in JSON format is stripped of two fields, specifically the beneficiary and `approvedBy` fields and then added again with links to the user resource. To get the link, the EntityLinks class provided by Spring is used. It generates the dynamical link to the resource. A POST request the same as before is executed with the campaign object as payload. Again a 201 `CREATED` return code is expected. After that a cleanup of the object used is performed.

```
@Test
        public void verifyCampaignPost() throws Exception {
                User testUser = new
User("Test","User","testCampaignPost@test.com","ksdhfisd",false,LocalDateTime.now(),Role.ADM
IN);
                testUser=uRep.save(testUser);
                Campaign testCampaign = new Campaign("TestCampaign", "This is a test
Campaign", testUser, 250.0, 125.5,
                LocalDateTime.now(), LocalDateTime.now().plusDays(50), CampaignCategory.ARTS,
true, testUser);
                testUser=uRep.findByEmail("test@test.com");
                testCampaign.setLikeCount(0);
                testCampaign.setIsApproved(true);
                String content=om.writeValueAsString(testCampaign);
                JSONObject jobj = new JSONObject(content);
```

```
                jobj.remove("beneficiary");
                jobj.remove("approvedBy");
                jobj.put("beneficiary",
links.linkToSingleResource(User.class,testUser.getId()).getHref());
                jobj.put("approvedBy",
links.linkToSingleResource(User.class,testUser.getId()).getHref());
                content=jobj.toString();

mockMvc.perform(MockMvcRequestBuilders.post("/api/v1/campaigns").contentType(MediaType.APPLI
CATION_JSON)
                        .content(content)
                                .accept(MediaType.APPLICATION_JSON))
                                .andExpect(status().is(201));
                cRep.delete(cRep.findByName("TestCampaign"));
                uRep.delete(testUser);
        }
```

The same was done for *Donations*. This time a mock user and a mock campaign were used, and after the test removed.

```
        @Test
        public void verifyDonationPost() throws Exception {
                User testUser = new User("Test","User","testDonationPost@test.com"
,"ksdhfisd",false,LocalDateTime.now(),Role.ADMIN);
                testUser=uRep.save(testUser);
                Campaign testCampaign = new Campaign("TestCampaignD", "This is a test
Campaign", testUser, 250.0, 125.5,
                        LocalDateTime.now(), LocalDateTime.now().plusDays(50),
CampaignCategory.ARTS, true, testUser);
                testCampaign.setApprovedBy(testUser);
                testCampaign.setBeneficiary(testUser);
                testCampaign.setLikeCount(0);
                testCampaign.setIsApproved(true);
                testCampaign=cRep.save(testCampaign);
                Donation testDonation=new Donation();
                testDonation.setAmount(100.0);
                testDonation.setStatusOK();
                String content=om.writeValueAsString(testDonation);
                JSONObject jobj = new JSONObject(content);
                jobj.remove("user");
                jobj.remove("campaign");
                jobj.put("user",
links.linkToSingleResource(User.class,testUser.getId()).getHref());
                jobj.put("campaign",
links.linkToSingleResource(Campaign.class,testCampaign.getId()).getHref());
                content=jobj.toString();

mockMvc.perform(MockMvcRequestBuilders.post("/api/v1/donations").contentType(MediaType.APPLI
CATION_JSON)
                        .content(content)
                                .accept(MediaType.APPLICATION_JSON))
                                .andExpect(status().is(201));
                cRep.delete(cRep.findByName("TestCampaignD"));
                uRep.delete(testUser);
                dRep.delete(dRep.findByUserIdAndCampaignId(testUser.getId(),
testCampaign.getId()));
        }
```

5.4.3. Handlers testing

Another crucial part of the back end is the handlers. The user handler makes sure that when a user is created the `memberSince` date is set to the current time on the server and not the value that was submitted. This is to make sure that data is consistent, and not dependent on client side code. A mock user is created with a member since date that is not today. The test then executes a POST command on the API and then retrieves the user Object. Then it compares the date set in the user object with the actual value set in the object. If they are the same the test fails.

Another important functionality of the handler is making sure duplicate users are not allowed. It is based on the email address as the unique element. Adding the same user again should result in a 400 response code. If the code is anything other than 400, the test fails.

```
	@Test
	public void testUserHandler() throws Exception{
		LocalDateTime sentTime=LocalDateTime.of(2017,03,21,21,18);
		User testUser=new
User("Test","User","testUserHandler@test.com","ksdhfisd",false,LocalDateTime.of(2017,03,21,2
1,18),Role.ADMIN);
		String content=om.writeValueAsString(testUser);

mockMvc.perform(MockMvcRequestBuilders.post("/api/v1/users").contentType(MediaType.APPLICATI
ON_JSON)
			.content(content)
				.accept(MediaType.APPLICATION_JSON))
				.andExpect(status().is(201));
//test if memberSince is changed to the current date rather than the one provided
assertNotEquals(uRep.findByEmail("testUserHandler@test.com").getMemberSince(),sentTime);

//test if an error is returned when adding the same user
mockMvc.perform(MockMvcRequestBuilders.post("/api/v1/users").contentType(MediaType.APPLICATI
ON_JSON)
			.content(content)
				.accept(MediaType.APPLICATION_JSON))
				.andExpect(status().is(400));
		uRep.delete(uRep.findByEmail("testUserHandler@test.com"));
```

5.5. Conclusions on testing and evaluation

The front end evaluation conducted through the method of Simple Usability Scale proves the fact that, subject to minor adjustments that could easily be performed, the front end of the prototype meets the non-functional requirement of being attractive for potential investors and presents the key information about the campaigns posted in a clear and concise way. The target audience of the distributed questionnaire was mainly represented by the young generation eager to engage in this new type of civic platform.

The web platform was tested using JMeter, two application servers and a load balancer. All these instances were configured using the OpenStack account provided by Cardiff University. The application was tested with 10, 100 and 1000 users. The medium response time was the metric used to measure performance.

To share data between the two application servers, MongoDB was configured in cluster mode that shards the data between two servers. This configuration requires a dedicated *Configuration Server* and *Query Resolver*. Using this mode, both servers were able to read and write the same data, at the same time from the cluster.

Due to the limited resources of the OpenStack, only 5 cores and 10 GB of RAM were allowed to be used. Therefore, the *Query Resolver* and the *Configuration Server* needed to be run on the same machine as the Load Balancer. Having only 5 cores, they were distributed in a way such that each application server made use of 2 cores and the load balancer was allocated only one core.

For the Load Balancer, the decision was made to use nginx as a proxy server to forward requests to both application servers. The tests revealed that even when using

a single core, the load balancer performed at an acceptable level. During the test that involved 100 users, the load was at 80%.

An important part of testing is integrated unit tests, for which jUnit was used. This allows testing the new code before deploying it into production (to the servers), ensuring that the existing functionality is not broken by the newly added code (features).

These tests demonstrated the fact that the web platform is horizontally scalable and performed as expected even under heavy load.

6. Future work

The needs of end users are constantly changing and the systems already available on the market are continuously expanding through introducing new features and applying optimisation techniques to enhance the user experience. In order to enhance the project functionality and make it more suitable for the use in a real world scenario, a set improvements that match the optional specifications outlined in Section 3.1.2 could be made. The application needs to constantly evolve and meet user expectations, as well as integrated with popular external, third party services.

6.1. Additional features

6.1.1. Authentication and authorisation system

Since the implemented system is currently in a prototype stage of the development, the primary focus of the development was on the project viability and scalability. In order to showcase the core features (related to the mandatory functional requirements outlined in Section 3.1.1) and demonstrate the proof of concept of the application, a mock authentication system was used.

The tools that were chosen as well as the design choices made allow the implementation of an authentication and authorisation system at a later stage of the development process.

This system will rely on tokens that are generated by the back end every time a user authenticates successfully. The token will be included in every request sent from the front end on behalf of the user. This would allow the back end to identify which user executes the request, by storing a map of the tokens and the username in the database that updates every time a new token for that user is generated. This token will also allow to authorise different permissions depending on the user type (for example, only admin committees will be allowed to approve campaigns).

6.1.2. Campaign recommendations

To help campaigns to reach their target audience, a data mining algorithm could be implemented to recommend similar campaigns to users who have already liked / pledged amounts to other campaigns. A system of tags / keywords could be implemented to assign different tags to campaigns in order to better categorise them based on their content and desired purpose. A clustering algorithm could be used to

group similar campaigns based on keywords / tags. When a user accesses a campaigns, a recommended list would appear as well. Currently, that list is constructed solely based on campaigns in the same category sorted by the number of *Likes*.

Another recommendation that could be made to the users visualising a campaign would be represented by other campaigns campaigns that were pledged the users who donated on that particular campaign. This could be accomplished by the use of A-Priori algorithm to mine association rules between campaigns.

6.1.3. Campaign advanced search

A more refined search could be implemented, allowing users to select different filters based on more fields of the Campaign object. Examples of search could include: campaigns initiated by a particular beneficiary, campaigns initiated in a particular location, campaigns that have a particular percentage of money pledged towards the reach of the target amount. This would allow users to be more specific about the campaigns they chose. The filters applied could also be saved in the local storage of the client browers for guest users and in session storage for logged in users, in order to persist their search preferences.

6.1.4. Campaign sponsorship

A system could be implemented to allow beneficiaries to make certain campaigns more visible through the payment of a fee that allows the campaign to be more visible in a “featured” section of the web platform. This “featured” status would be limited to a preset time period (such as 7 or 30 days) and could be applied on the newly added campaigns, as well as the campaigns that are already active and have already reached their target amount partially. This would allow beneficiaries to increase their chances of reaching the target goal within the planned timeframe.

6.1.5. Heatmaps for user behavior analysis

The activity of the visitors of the XpressStarter platform (especially mouse movements and how much time they spend looking at a particular area of the Graphcal User Interface) could be analysed through the use of Heatmaps. In the background, Javascript is used to associate the coordinates of the cursor on the screen and timestamps to generate graphical traces of mouse movements.

This way, the administrators of the platforms would be able to see which are the areas of the Graphical User Interface that represent a particular interest for the visitors of the website and make some approximate deductions both of the areas that represent a particular interest for the users and the areas that they tend to avoid, so that they could be improved visually. The use of this approach could significantly improve the conversion rate of visitors into investors.

There are both commercial and open source solutions available on the market that allow the integration of Heatmaps as a tool for user behaviour analysis. One of the most popular dedicated usability analysis tools is UsabilityTools (http://www.usabilitytools.com). HotJar (http://www.hotjar.com) is another more comprehensive, all-in-one solution, that offers advanced tools such as real-time

recordings of user sessions and conversion funnels. A well known open-source alternative that would offer basic functionality is Heatmaps.js (https://www.patrick-wied.at/static/heatmapjs). I have personally contributed to a similar project (called iTrackr) as part of a Hackathon in 2013: https://github.com/sabinmarcu/iTrackr.

6.2. Integration with external services

On top of the additional implemented features, there could still be added a couple of integrations that may gather additional potential users for XpressStarter project. When it comes to crowdfunding, the first problem that the initiators of the projects are likely to face is publicity. The key for successfully completing a crowdfunding campaign is by having a large exposure on the Internet. The backers represent only a small subset of campaign's visitors. Getting a large number of visitors is the number one priority.

The fastest and the most efficient way to promote a campaign is through social media. If a campaign presents original and interesting ideas, the shares count can bring a significant amount of new visitors. To encourage the visitors to share their favourite campaigns, the inclusion of social features like authentication and sharing using social media accounts is planned.

In order to reduce the burden of creating a new user account on the XpressStarter web platform, authentication with social media accounts using the oAuth protocol provided by Facebook / Twitter / Google's API could be implemented. This would require the user to grant permisssions to XpressStarter to access their public profile information and their email address, gathering the key information necessary to create a profile on our web platform.

One step further, in order to encourage backers to contribute instantly, a payment method widely available in the world needed to be integrated. Paypal is the most popular and accessible service, which supports a wide variety of credit cards.

6.3. Future-proof scalability with REDIS

In the future, when the number of concurrent visitors for XpressStarter will increase, an additional in memory caching layer can be added. REDIS and Memcached are the main in-memory databases that perform predictable caching in a distributed cluster. The memory caching server is strictly related to the database queries performed on the back end.

REDIS is a database stored in the memory of a system that offers very high speeds of reading and writing. It differentiates from the conventional databases through being a non-relational database, just like MongoDB and Memcached, falling into the category of NoSQL databases. It offers 5 main different types of data structures to manipulate data: Strings, Hashes, Lists, Sets, Sorted sets, in which data from MongoDB could be translated and stored with the purpose of caching metadata used for statistics. This would represent a significant improvement to the performance of the application, since deserializing objects from the database into memory is an expensive operation and should only be done when the whole object is necessary. Caching metadata or responses in REDIS would decrease load times and reduce load on the application servers.

REDIS for high traffic web services is not running as a standalone instance, but the different instances of REDIS are connected in a cluster. When planning a caching service, the goal is to develop a distributed caching service with REDIS.

REDIS uses two models to achieve distributed environments, Sentinel and Cluster. Sentinel needs at least three machines and another at least two for the actual data. The sentinels monitor the REDIS servers and by using a quorum-based system they vote if they can see the active master. If the number of sentinels that do not see the server as active are above the quorum threshold, the standby server is promoted to active.

Cluster mode also works in groups of 3, splitting the keyspace between the active servers, also known as masters. An equal number of servers is then assigned as read slaves, that are read-only and replicate one of the masters. They are known as replicas. Should one of the masters go down, the replica is promoted to master and the cluster is functional. When the server comes back up, it is demoted to a slave, and replicates the new master. The cluster setup also works with a quorum. Our hosting / infrastructure provider cannot be expected to have 100% uptime for all the machines. In case of a 0.01% chance of failure, the setup must be able to redirect our traffic to the rest of our up and running servers and then replicate the data when the servers become available again.

Through storing the metadata of the entities in REDIS, the data retrieval would be much faster than it would be if it were to be retrieved from MongoDB. Another way REDIS could be integrated with our microservice would be through storing data that does not change frequently (such as the highest rated campaigns), eliminating the need of extracting that information from MongoDB.

6.4. Conclusions of future work

All the future work elements listed above are achievable through the design choices of the application that allows meeting user expectations and providing a reliable service through constant development and feature implementation.

7. Conclusions

In conclusion, a fully working prototype of a modern crowdfunding platform was developed. The implementation is clean, the code is human readable, reusable and new functionalities can be added over time with minimal code additions.

Every process that could be automatized was automated and the tools used do not add a complexity overhead for the production, so that end users and developers do not suffer any drawback.

As open source libraries have been used that benefit from a high level of popularity, that are actively maintained by the community, the fact that the application is both modern and future-proof is assured. Moreover, through the design choices made the platform is also horizontally scalable, which means that in the event that more processing power is needed, more machines can be added to support the additional load and increase performance.

Several challenges were faced when designing the database, in order to ensure that is stores the data in the most efficient and safe manner. This resulted in a 3NF compliant data structure.

The application is composed by two separate microservices, which will allow us to scale the platform more predictably, allowing both the back end and the front end to be platform independent, as a benefit for the developer. To allow the code to be loosely-coupled, a general standardised API was used. The most popular way to achieve this is by using a REST API over HTTP. In order to facilitate the integration with third party services, HATEOAS was used, that offers a standardised way to present the API routes, that are self-documented.

The front end is built using modern technologies, ensuring a modern look and responsive feel by being designed in a mobile-first manner. The front end resources were optimised by bundling and minimising all our source code and even fonts and images, so no matter what device the end user owns, the website is fully functional in all circumstances. The development workflow is straightforward to follow, Gulp and its integrated modules allow any change in code to take effect automatically. On the production, the new changes can be deployed with a single command in Terminal.

For the back end implementation, the Spring framework was selected due. It is highly popular because of its modularity, allowing new features to be added simply by adding modules and configuring them. Using a framework enabled the code to be more readable, as it makes it easier for the developer to follow certain coding conventions. Using the HATEOAS module allowed us to have generated endpoints for the entities.

Testing was done to prove that the application was scalable and to validate my design choices. The tests were done using JMeter emulating parallel requests to a “high availability” configuration of the application that was set up together with a load balancer on the OpenStack account provided by Cardiff University. Unit tests were setup to ensure that all the new features developed during the implementation process were not affecting the functionality of the features that were already implemented.

All of the requirements listed in the mandatory specification in Section 3.1.1 have already been implemented and the requirements listed in the optional specification in Section 3.1.2 are currently under ongoing development. The priorities are represented by the integration with social platforms such as Facebook and Google. The development process for this features has already started and the concept is outlined in Section 3.2.2.

Since efficiency and performance represent concerns of utmost importance for us, new technologies are being researched to improve the performance of the scalability of our application beyond what has already been achieved. Technologies like REDIS could improve our caching mechanism and user experience through minimising load times.

Another important concern is represented by user experience and providing users with the ability to express themselves. A system of *Likes* has already been implemented. To help the users to interact with the platform in a more active way, a system that allows the user to comment on campaigns is being researched. It is believed so far that the Disqus API would represent the best option, since it also offers integration with the most popular social platforms natively.

Through the design choices made, the prototype is efficient in handling user requests and storing user data. Adding new features is facilitated through the modular structure

of the application. The front end usability and attractiveness evaluation outlined in Section 5.1 proves the fact that the Graphical User Interface is modern, responsive and cross-platform compatible, attractive enough to attract potential investors and emanates a feeling of trustworthiness.

When the project initially started, my experience in the process of designing and developing an application was narrowly focused on implementing features and gathering requirements from the clients. In earlier, smaller projects that I have been involved in, I tended to neglect important steps in the software development lifecycle such as application design and thoroughly testing. During the development of this platform, I encountered challenges that are briefly outlined in Section 3.1, and detailed in sections 4.1.4 and 4.2.2 that I overcame through a comprehensive process of research and testing.

In my opinion, all the key objectives highlighted in Section 1.2 have been met. Being scalable and modern through the design choices made, the project has the potential to fulfill all the optional specifications highlighted in Section 3.1.2 and perform in a real life scenario. Bespoke features beyond the optional specifications that the client might wish to be implemented or pilot them in the future could be easily added at a later stage on top of the existing architecture.

As a developer, this project continues to represent an enriching experience, as it has given me insight into all the stages of the software development lifecycle, as well as time management, as I had to carefully prioritise all the requirements and make design decisions quickly and carefully at the same time in order to deliver a quality prototype in time.

8. Reflection on Learning

In reflection to the project, I consider that I have successfully achieved the core objectives that I have established at the start and also commenced the implementation of optional specifications. However, there are certain areas in which I could have performed better. Although the suitability of the design choices made for the project usability, performance and scalability were confirmed by the testing and evaluation performed in Section 5, a more thorough research could have been done, as well as a SWOT analysis with the alternatives.

8.1. Complexity estimation

Given my limited experience with Java and the fact that Java SE is not geared towards web development, I expected the back end implementation to be more tedious that it actually proved to be. The choice of the Spring Framework helped me through the generation of boilerplate code and facilitated feature implementation through modules. For example, in the Campaign object, when serializing the data to JSON, adding a percentage of pledged amount vs. total amount required without storing the data in memory was achieved through `@JSONProperty("Percentage")`.

When the HATEOAS module was applied, it generated dynamic controllers based on repositories. I initially thought that adding custom business logic when data is created or modified (by a client initiating a POST or PUT request) would be simple, but it later proved to be more difficult, as it needed custom handlers built for each entity, as well as carefully crafted data flow.

Once the back end had been developed up to a state where it could start to be integrated with the front end, I initially thought that the complexity of the remaining tasks, given my extensive experience on front end implementation was going to be trivial. However, I realised the fact that the structure of some JSON objects returned by the back end in the default format of HATEOAS was not optimal for rendering on the front end. Therefore, I needed to integrate the Express JS library that could pre-process the JSON data returned by the back end and also translate the API routes being called in a more user-friendly format.

8.2. Continuous integration vs. sequential development of the microservices

The process of developing the back end before the front end has lead to some technical challenges that became evident only after the start of the front end development. The issue described in Section 4.2.2.5 is a typical example of an integration problem that could have been prevented. Developing the 2 microservices at the same time could have offered a better insight into the requirements (both in terms of functionality and API design) and performance of the application.

8.3. Time and effort allocation

While I generally felt comfortable with the technical elements of the project, I also needed to spend a significant period of time doing research that justified my design and technology choices. This has occupied a large portion of my available time. Reflecting back on this, I believe I could have better focused my research on the specific areas that concerned the scope of my prototype and leave the research into the later functionalities for later stages.

8.4. The consideration of commercial platforms

After having used a series of popular open-source platforms for the development of my project, I now consider that I could have also done more research into the commercial platforms available on the market that could have offered features, libraries or modules beneficial for my project that the open source platforms do not have built in. Due to the financial constraints given by the need of purchasing a license for some of them, such as Oracle or JBoss Enterprise (a commercial application server alternative to the open-source Tomcat), the decision was made to use open source software for implementation, eliminating the need for additional costs.

8.5. Requirements prioritisation and problem-solving

Linking with the points discussed in Section 1.8. Key Challenges, being able to prioritise the main requirements needing to be implemented was a main challenge for me. I needed to firstly evaluate the criteria without which the platform could not function in the desired state. Similarly, I gained important problem-solving skills by responding to any issues coming up in the development of the platform and the expansion of the agreed requirements.

8.6. Full software development lifecycle exposure

Having a background as a professional web developer, stepping into areas of the design stage of the software development lifecycle has been a truly beneficial learning experience for me. Only having been involved in the implementation stage until recently, this project gave me the opportunity to go through the entire software development lifecycle, following a Waterfall methodology with certain Agile insertions. Therefore, I went through all stages of requirements gathering, design, implementation and testing. I have also gained an understanding of the need to create a scalable product from the start, and the discipline required in selecting the appropriate technologies that will support this aim later on.

8.7. Enhancing my skills and employability

Being exposed to technologies covering the full stack of software development (both back end and front end), as well as managing the project myself, increases my prowess as a developer and enables me to pursue a career as a full stack developer and later as a lead developer and / or project or product manager.

My project management skills have also been enhanced through the extensive use of tools such as Git and Github, as described in Section 3.5. However, I realised the fact that the feedback process could have been enhanced by offering the client access to the Github issue tracker. This would have enabled him to provide me with real time feedback regarding issues / errors discovered, feature requests and enhancements that could be applied to the system.

8.8. Reflection conclusions

Reflecting on all the previously mentioned concerns, I believe that the reflection process has helped me become a more reflective practitioner in future projects, both in my academic and in my professional life that will enhance the quality of my workmanship.

9. Appendices

Appendix 1: Full back end UML entities diagram

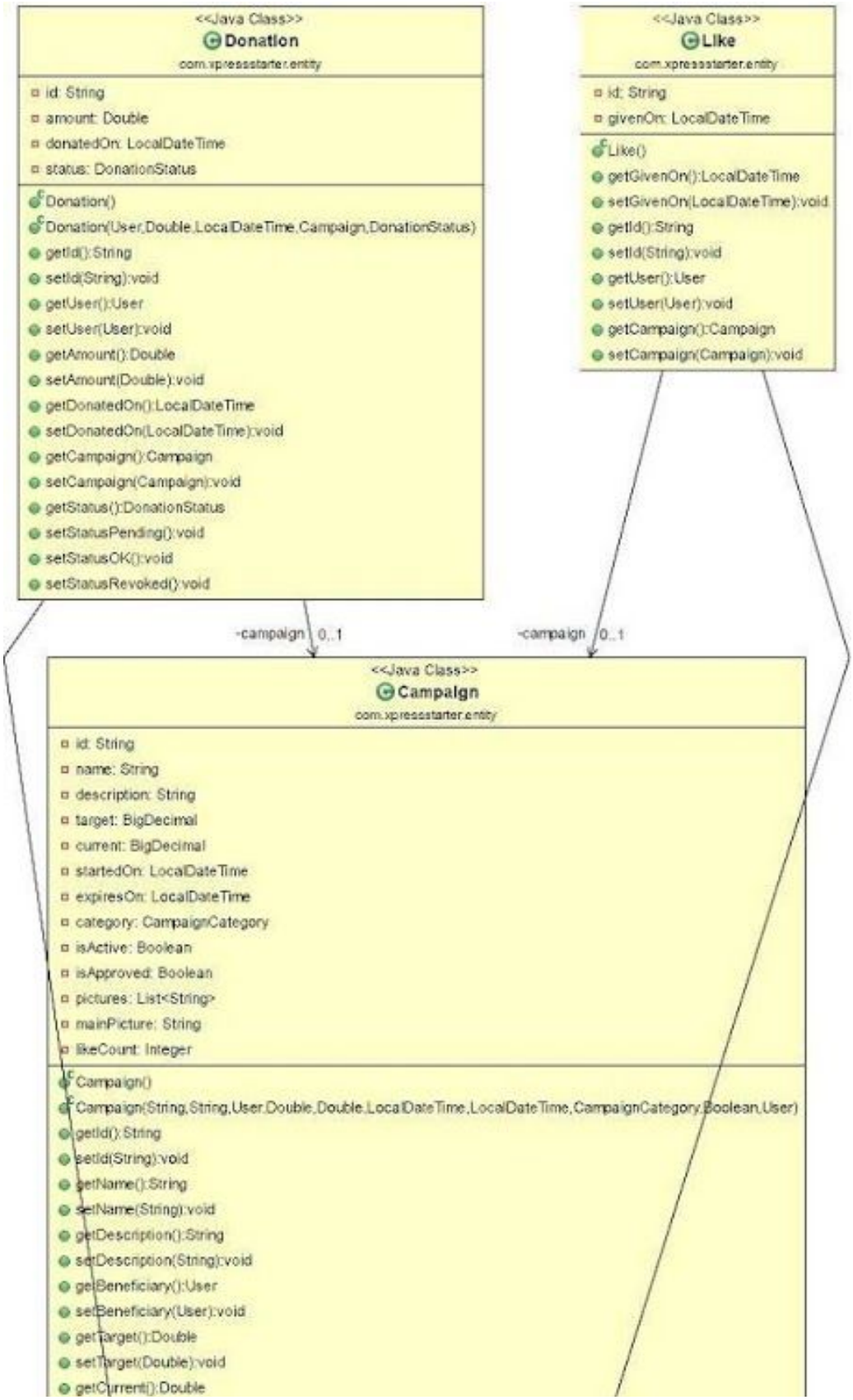


Fig. 9.1: Full back end UML entities diagram - Part 1

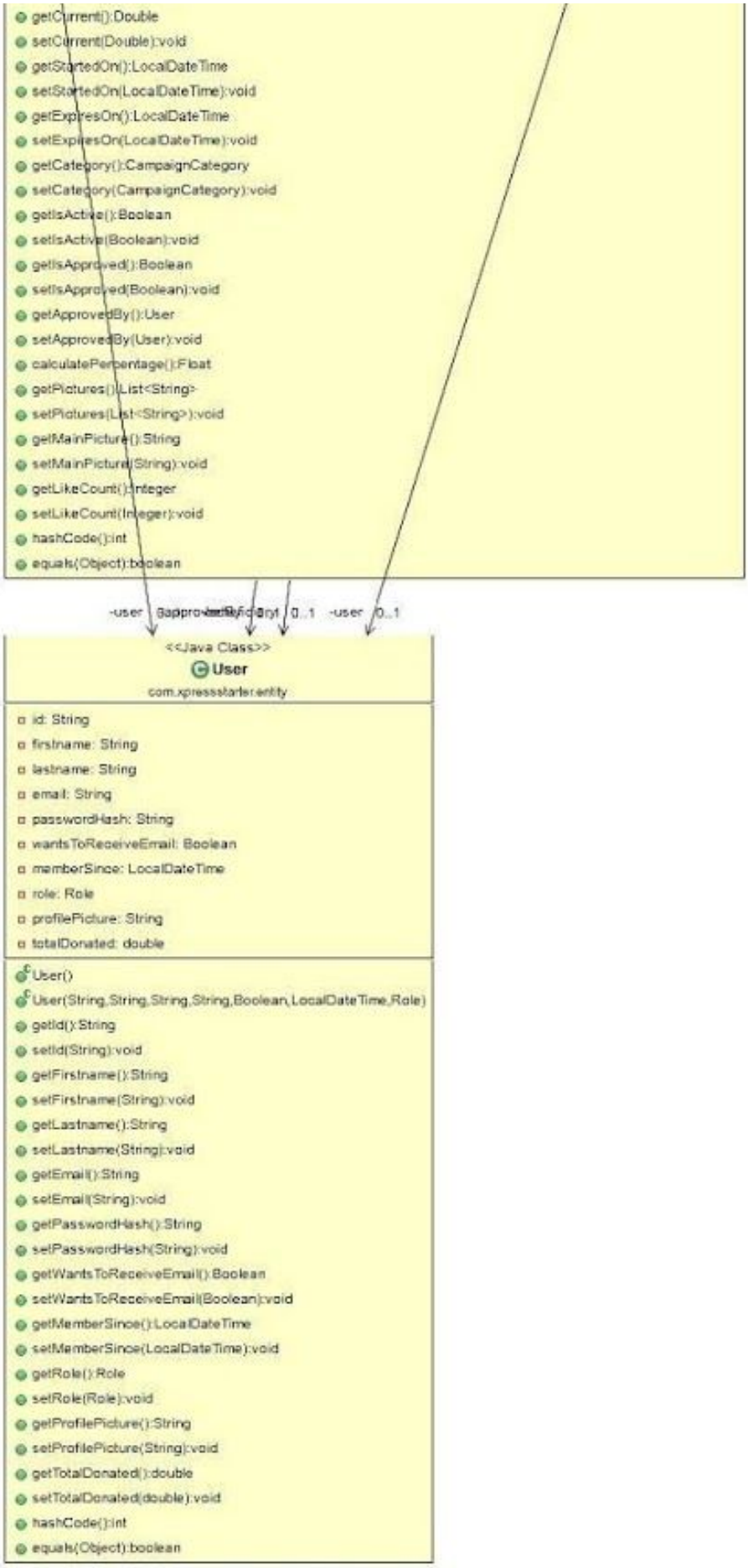


Fig. 9.1: Full back end UML entities diagram - Part 2

Appendix 2: Questionnaire for the evaluation the front end interface design, attractiveness, trust and demographics of the target audience

The following questionnaire was presented to potential users (both beneficiaries and benefactors) for the evaluation of the front end interface design, attractiveness, trust and demographics of the target audience. This was split into 3 parts: *Design*, *Trust* and *Demographics*. The instructions given to the users are highlighted in *Italics* and the answer options are highlighted in **bold** and separated by hyphen (**–**). The *Analysis justification* was not provided to the users in order not to overload them or bias the responses.

1. Have you ever used a crowdfunding platform such as Kickstarters?

Yes – No

The following questions will walk you through a platform aimed at promoting civic projects and matching creators with potential donors. The data collected in this questionnaire will only be used in the evaluation of the front end for my final year project, for academic purposes, and will not be shared with any other third parties. As you go through the questions, please think of how usable and easy to navigate this site is for you.

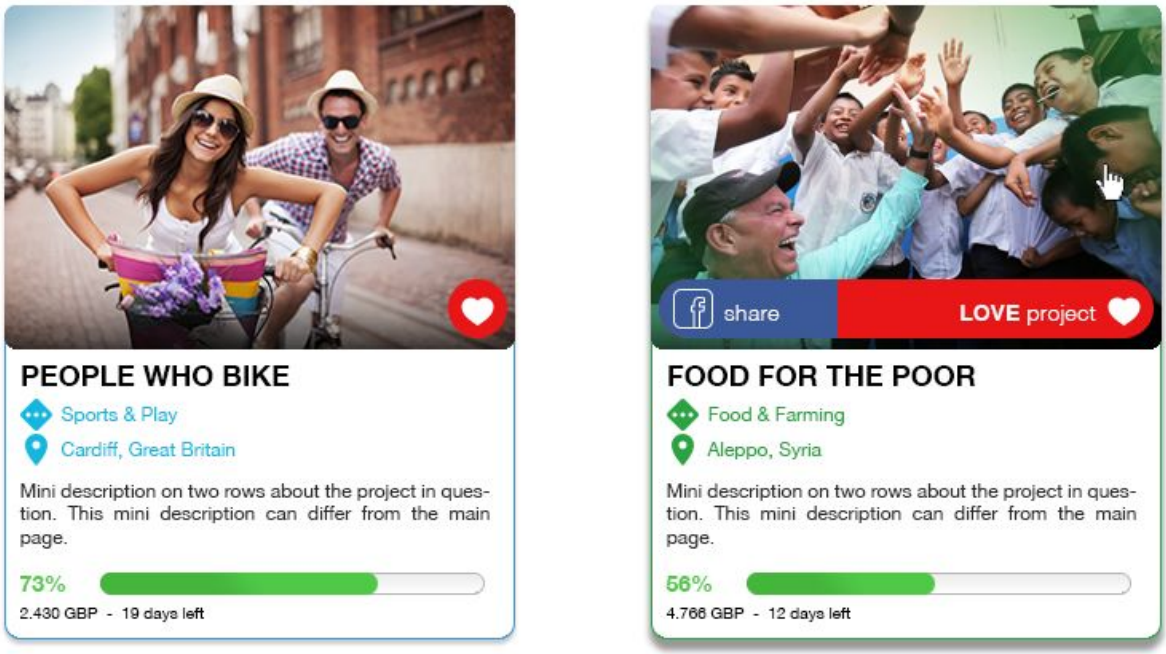
Part 1. Design

2. Looking at the home page shown here (also at the following address: http://www.xpressweb.site/ , which of the words below come to mind? (More than one option can be selected)

Relaxed – Trustworthy – Unclear – Informative – Crowded – Logical – Engaging

Analysis justification: This question will be used for sentiment analysis, looking at the associations made by potential users on the platform. I aim to present the results as a diagram of overall user perception

3. By looking at the example civic projects below, how clear is it to you at what stage each project is in terms of funding, time remaining until cut-off and target amount?

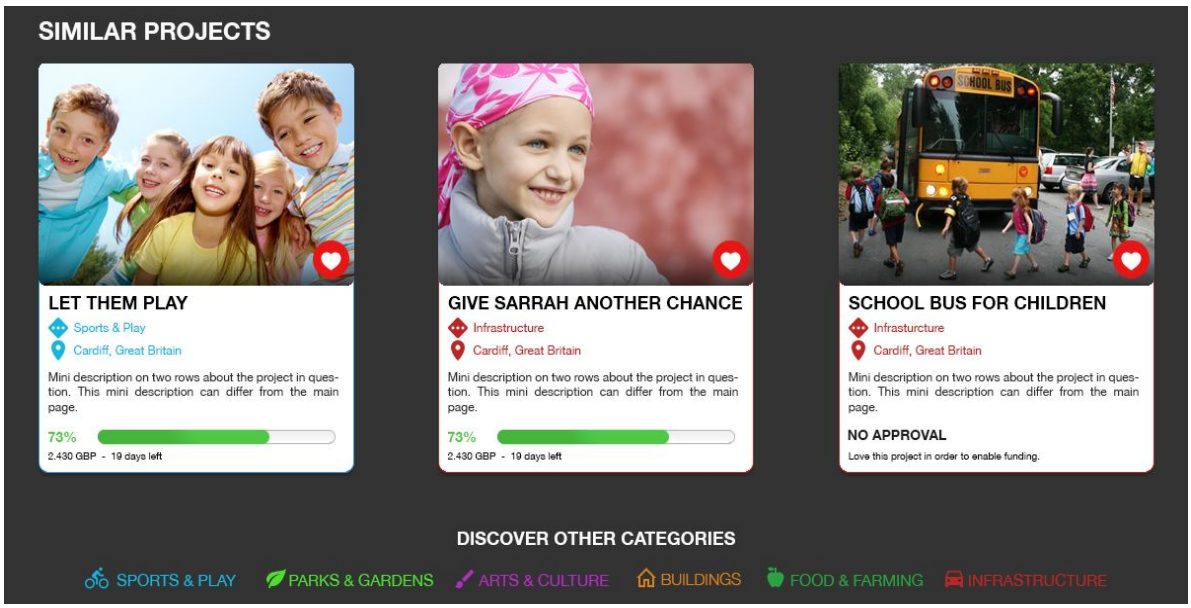


Clear – Relatively clear – Relatively unclear – Unclear

Analysis justification: This question will provide insight on the effectiveness of the structure proposed. An overall “Clear” score for this question is aimed, otherwise this may indicate an ineffective layout.

4. As someone willing to invest in a business idea, would the view of an interface such as the one shown below encourage you to invest?

Yes – No – Maybe



Analysis justification: This question will give me insight into the correlation between website layout and actual user engagement. A high score on this will justify further

developments and going beyond the Proof of Concept stage

Part 2. Trust

5. Do you trust online crowdfunding platform to only feature promising business ideas of a high standard?

Yes – No

Analysis justification: This question is building on my project positioning, filling the gap of trustworthy crowdfunding websites that only advertise legit business ideas. I aim to use the data collected for this question in my product positioning later on and flag the importance of coming up with a website that is valuing high quality content.

6. Which of the civic project areas below are you likely to be interested in investing/submitting project ideas?

Sports & Play – Parks & Gardens – Arts & Culture – Buildings – Food & Farming – Infrastructure

Analysis justification: This question has all the current proposed project categories as answer options. The data behind this will inform perhaps a later focus of the website on a specific category, as well as giving me a better indication of the categories that are likely to attract more business ideas/more funding.

7. If a crowdfunding platform would have administrators in place to filter the robust business ideas from the ones that do not have a clear direction, as well as checking that the founder has met their obligations towards donors after the business idea has been fully funded, how likely are you to recommend this platform to friends?

Extremely likely – Relatively likely – Relatively unlikely – Extremely unlikely

Appendix 3: Regulatory compliance of crowdfunding platforms in the UK

XpressStarter is a crowdfunding platform (later referred to as CFP) which falls under the category of donation-based crowdfunding, a structure that is dedicated for social benevolent enterprises where investors pledge money relying on trust, without necessarily expecting a material return, but a return in the form of benefit for the society or local community they live in. As with any type of CFP, there are a number of regulatory and other legal issues that need to be considered to run the platform in a real world environment. The laws governing these regulations have a high degree of complexity. This section outlines just the key (and minimum) areas in which a CFP needs to be compliant in order to function legally.

Compliance with the code of conduct of UKCFA

Firstly, all CFPs running in the UK should be aware of the existence of the UK Crowdfunding Association (UKCFA), which is the self-regulatory body made up of leading crowdfunding businesses that has the aim to provide clarity and consumer protection for the whole industry. A Code of Conduct has been released ([49] UKFCA, 2017) that aims to protect the growing number of investors and fundraisers in the UK.

Financial regulations

There are two key issues that need to be considered in relation to XpressStarter complying with the financial services regulation: whether it requires authorisation and whether the financial promotion regime applies.

Since XpressStarter is a donation-based platform, it normally falls outside the scope of FCA (Financial Conduct Authority) authorisation. However, issues might arise depending on how payments are processed (please see the *Payment Processing* section below) or if any reward that is offered to the benefactors may be considered a return on investment (for example, a profit share from a successful creative enterprise).

If the beneficiaries that post their campaign pitches on XpressStarter give a background to their business and/or project, this might be classed as a *financial promotion* (an invitation or inducement to engage in investment activity, such as subscribing for shares). If this definition were to be taken into account strictly, all the beneficiaries posting campaigns on XpressStarter must either fall within an exemption (including communication to certain sophisticated investors or high net worth individuals) or have the pitching content of the campaigns they post approved by an authorised person. In most of the cases, XpressStarter will need to act as the authorising entity. This would lead again to the need of XpressStarter to be FCA authorised. However, strict FCA rules govern the content of these promotions to ensure they are clear, fair and not misleading. This means that the benefits of

investing must be carefully considered along with the risks, and the content should be appropriate for the expected target audience of the promotion. This can be particularly challenging for social media channels that restrict the number of available characters (such as Twitter). However, some guidance published by FCA reaffirms that the regulatory requirements are media neutral. XpressStarter should also be careful that any campaign listed on the platform does not take the shape of investment advice. FCA GC 14/6 ([50] Financial Conduct Authority, 2017) list some examples of good practice in this area.

Payments processing

Another key consideration for a CFP such as XpressStarter is related to how the processing of payments made by the benefactors will take place. The Payment Services Regulations 2009 (PSR), set out a number of 'payment services' that require an entity to be FCA authorised. A common issue is where investments / donations / payments come through the own bank accounts of the CFPs as this could be deemed to be a money remittance service under the PSR. There are exemptions, including where a platform acts as a commercial agent for either payer or payee to conclude or negotiate the sale or purchase of goods or services. However, this is not a comfortable fit given that CFPs are rarely involved in the sale or purchase of actual 'goods or services'.

In order to avoid all the issues that might arise from falling under the restrictions of PSR and the need to be FCA authorised, XpressStarter should use third party online processing providers (like WePay, PayPal and Stripe) to process payments on its behalf. This is listed as an optional functional requirement in Section 3.1.2.

Data protection law compliance

Since XpressStarter is a CFP that processes personal data of users (names and addresses, phone numbers, email addresses), it has statutory obligations in regards to the way it collects, uses, stores and shares this data. Since it will need to comply with Data Protection Act 1998 (DPA) and other legislation, XpressStarter will need to (among other aspects):

- register with the Information Commissioner's Office (ICO). A CFP processing personal data will almost certainly be a "data controller" under the DPA, in which case it will need to register with ICO. Failure to do so is a criminal offence;
- establish and maintain a robust privacy and cookies policy, and make sure that the policy is easily accessible on one of the pages of XpressStarter;
- if any third party processes data on XpressStarter's behalf (acting as the "data processor"), XpressStarted needs to enter into an agreement to ensure that the third party is contractually obliged to process the data in accordance with the statutory requirements;
- ensure that no personal data is transferred outside the EEA unless certain preconditions are satisfied. This rule applies to personal data held on servers

outside the EEA, so care must be taken when selecting any cloud providers or other data processors;
- put in place adequate security measures to mitigate the risk of users' data being accidentally or deliberately compromised;
- comply with cookie legislation by giving clear information about the purpose of cookies used on the platform (a simple reference to cookies in the privacy policy is not sufficient) and obtaining users' consent before cookies are placed on their machines. Market practice in the UK is to obtain implied consent through a prominent cookie notice on the website.

The full Data Protection legislation applied in the UK falling under DPA 1998 can be found on the UK Government website ([51] The UK Government).

Final thought on regulatory compliance

While all the laws and regulations that may apply to XpressStarter might seem overwhelming, potential pitfalls can be avoided by seeking professional advice during the early stages of the setup. It is important to note that it is a criminal offence for a CFP to act without all the required authorisations, punishable by a prison term of up to two years, unlimited fines and potentially a liability to compensate for investor losses. Therefore, all the regulatory compliance will need to be ensured before launching the system live to the general public.

10. References

10.1. Introduction

[1] Hunter, G.L. and Garnefeld, I., 2008. *When does consumer empowerment lead to satisfied customers? Some mediating and moderating effects of the empowerment-satisfaction link. Journal of Research for Consumers, (15)*, p.1.

[2] Telerik Developer Network, 2017. *Javascript In 2017 - Libraries And Frameworks*. [ONLINE] Available at: http://developer.telerik.com/topics/web-development/javascript-2017-libraries-frameworks/. [Accessed 20 March 2017]

10.2. Background

[3] Aldrich, H.E., 2014, August. The democratization of entrepreneurship? Hackers, makerspaces, and crowdfunding. In *Presentation for Academy of Management Annual Meeting, Philadelphia, PA*.

[4] Harding, R., 2004. *Social enterprise: the new economic engine?*. *Business Strategy Review*, *15*(4), pp.39-43.

[5] Fast Company, 2014. *Pay For Your City: Crowdfunding For Civic Projects Is Unusually Successful* [ONLINE] Available at: https://www.fastcompany.com/3031412/pay-for-your-city-crowdfunding-for-civic-projects-is-unusually-successful [Accessed 31 January 2017]

[6] Matsuo, Y. and Yamamoto, H., 2009, April. *Community gravity: measuring bidirectional effects by trust and rating on online social networks. In Proceedings of the 18th international conference on World wide web* (pp. 751-760). ACM.

10.3. Specification and design

[7] E. Codd, Wikipedia, 2017. *Third normal form* [ONLINE] Available at: https://en.wikipedia.org/wiki/Third_normal_form [Accessed 5 May 2017]

[8] RestCase, 2015. *REST APIs and their Gain Added Importance on the Rise in Application Integration Design* [ONLINE] Available at: http://blog.restcase.com/rest-apis-and-their-gain-added-importance-on-the-rise-in-application-integration-design/ [Accessed 4 Apr. 2017].

[9] Time Inc, 2017. *Here's why Amazon's cloud suffered a meltdown this week*. [ONLINE] Available at: http://fortune.com/2017/03/02/amazon-cloud-outage/

[10] Webber R., 2013. *Analysis of JSON use cases compared to XML.* [ONLINE] Available at: https://blogs.oracle.com/xmlorb/entry/analysis_of_json_use_cases [Accessed 4 Apr. 2017].

[11] Tony Thomas, 2012. *Medialoot - Skeuomorphic Design: What it is, Who uses it, and Why You Need to Know*. [ONLINE] Available at: https://medialoot.com/blog/skeuomorphic-design/. [Accessed 2 April 2017].

[12] Keith Bryant, 2012. *designmodo – Rounded Corners and Why They Are Here to Stay*. [ONLINE] Available at: https://designmodo.com/rounded-corners/. [Accessed 2 April 2017].

[13] Carrie Cousins, 2016. *designmodo - 11 Web Design Trends for 2016*. [ONLINE] Available at: https://designmodo.com/web-design-trends-2016/. [Accessed 2 April 2017].

[14] Carrie Cousins, 2017. *design shack - 7 Web Design Trends to Watch Out for in 2017*. [ONLINE] Available at: https://designshack.net/articles/inspiration/web-design-trends-2017/. [Accessed 2 April 2017].

[15] Stacey Kole, 2013. *Webdesignerdepot - Serif vs. Sans: the final battle*. [ONLINE] Available at: http://www.webdesignerdepot.com/2013/03/serif-vs-sans-the-final-battle/. [Accessed 2 April 2017].

[16] Karol K., 2015. *Adobe - Motion in Web Design the Smart Way*. [ONLINE] Available at: https://blogs.adobe.com/creativecloud/motion-in-web-design-the-smart-way/. [Accessed 2 April 2017].

10.4. Front end implementation

[17] NewStack, 2016. *JavaScript Popularity Surpasses Java, PHP in the Stack Overflow Developer Survey* [ONLINE] Available at: https://thenewstack.io/javascript-popularity-surpasses-java-php-stack-overflow-developer-survey/ [Accessed 4 Apr. 2017].

[18] RisingStack, 2016. *Node.js Examples - What Companies Use Node for in 2016* [ONLINE] Available at: https://blog.risingstack.com/node-js-examples-what-companies-use-node-for/. [Accessed 4 Apr. 2017].

[19] NPM Inc., 2017. *The Node Package Manager* [ONLINE] Available at: https://www.npmjs.com/. [Accessed 4 Apr. 2017].

[20] Ryan Dahl, 2012. *Original Presentation on NodeJS.* [ONLINE] Available at: https://www.youtube.com/watch?v=ztspvPYybIY. [Accessed 4 Apr. 2017].

[21] NodeJS Foundation, 2017. *NodeJS API Documentation*. [ONLINE] Available at: https://nodejs.org/api/. [Accessed 4 Apr. 2017].

[22] Mozilla Developer Network, 2017. *Javascript Reference.* [ONLINE] Available at: https://developer.mozilla.org/en-US/docs/ [Accessed 4 Apr. 2017].

[23] StrongLoop, IBM, 2017. *Express - Node.js web application framework* [ONLINE] Available at: https://expressjs.com/. [Accessed 4 Apr. 2017].

[24] *The official Jade Documentation.* [ONLINE] Available at: https://pugjs.org/api/reference.html [Accessed 6 Apr. 2017].

[25] Hampton C., Natalie W., Chris E., 2017. *The official Sass Documentation.* [ONLINE] Available at: http://sass-lang.com/guide [Accessed 5 Apr. 2017].

[26] Ashley Nolan, 2015. *The State of Front-End Tooling – 2015* [ONLINE] Available at: https://ashleynolan.co.uk/blog/frontend-tooling-survey-2015-results. [Accessed 4 May 2017].

[27] Github, 2017. *The official Gulp Documentation.* [ONLINE] Available at: https://github.com/gulpjs/gulp/blob/master/docs/API.md [Accessed 5 Apr. 2017].

[28] Github, 2017. *The official Bootstrap repository* [ONLINE] Available at: https://github.com/twbs/bootstrap [Accessed 5 May 2017].

[29] Mark Otto, 2017. *The official Bootstrap Documentation.* [ONLINE] Available at: https://v4-alpha.getbootstrap.com/getting-started/ [Accessed 6 Apr. 2017].

[30] Search Engine Journal, 2016. *75% of Internet Use Will Be Mobile in 2017.* [ONLINE] Available at: https://www.searchenginejournal.com/75-internet-use-will-mobile-2017-report/177433/ [Accessed 6 Apr. 2017].

[31] Schlueter, Isaac Z., 2013. *Forget CommonJS. It's dead. We are server side JavaScript.* [ONLINE] Available at: https://github.com/nodejs/node-v0.x-archive/issues/5132#issuecomment-15432598 [Accessed 5 May 2017].

[32] Gregor Martynus, 2017. *The official Moment JS Documentation.* [ONLINE] Available at: https://momentjs.com/docs/ [Accessed 4 Apr. 2017].

[33] Github, 2017. *The official D3 Documentation.* [ONLINE] Available at: https://github.com/d3/d3/wiki [Accessed 5 Apr. 2017].

[34] HTML5Rocks, 2012. *Introduction to Javascript SourceMaps.* [ONLINE] Available at: https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/ [Accessed 5 Apr. 2017].

[35] *Official Nodemon website*. [ONLINE] Available at: https://nodemon.io/ [Accessed 5 Apr. 2017].

[36] *The official Browsersync Documentation.* [ONLINE] Available at: https://www.browsersync.io/docs [Accessed 6 Apr. 2017].

[37] Anton Kovalyov, 2017. *The official JsHint documentation.* [ONLINE] Available at: http://jshint.com/docs/ [Accessed 5 Apr. 2017].

[38] Twitter, 2017. *The official Bower Documentation.* [ONLINE] Available at: https://bower.io/ [Accessed 5 Apr. 2017].

[39] Microsoft, 2012. *Performance Implications of Bundling and Minification on Web Browsing.* [ONLINE] Available at: https://blogs.msdn.microsoft.com/henrikn/2012/06/16/performance-implications-of-bundling-and-minification-on-web-browsing/.

10.5. Back end implementation

[40] TheServerSide, Kurt Marko, 2017. *Best programming languages for enterprise development.* [ONLINE] Available at: http://www.theserverside.com/feature/Best-programming-languages-for-enterprise-development [Accessed 4 Apr. 2017].

[41] The PHP Group, 2017. *PHP: Backward Incompatible Changes.* [ONLINE] Available at: http://php.net/manual/en/migration54.incompatible.php [Accessed 4 Apr. 2017].

[42] Brewer E., 2000. *Towards Robust Distributed System. Symposium on Principles of Distributed Computing (PODC)*

[43] Boyarsky J, Selikoff S., 2015. *OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808* (pp 40)

[44] Simon Maple, 2017. *Java Web Frameworks Index: March 2017.* [ONLINE] Available at: https://zeroturnaround.com/rebellabs/java-web-frameworks-index-march-2017/ [Accessed 4 May 2017].

[45] Martin Fowler, 2010. *The Richardson Maturity Model.* [ONLINE] Available at: https://martinfowler.com/articles/richardsonMaturityModel.html [Accessed 4 May 2017].

10.6. Testing and evaluation

[46] MongoDB Documentation, 2017. Sharded Cluster Components. [ONLINE] Available at: https://docs.mongodb.com/manual/core/sharded-cluster-components/ [Accessed 4 May 2017].

[47] Ben Yemini, 2014. *Is there an optimal CPU utilisation?* [ONLINE] Available at: https://turbonomic.com/blog/on-turbonomic/optimal-cpu-utilization-depends/ [Accessed 4 May 2017].

[48] Klaus Enzenhofer, 2016. *What is the expected response time to deliver good user experience?* [ONLINE] Available at: https://www.dynatrace.com/blog/what-is-the-expected-response-time-for-a-good-user-experience/ [Accessed 4 May 2017].

10.7. Regulatory compliance

[49] UKFCA, 2017. *Code of conduct.* [ONLINE] Available at: http://www.ukcfa.org.uk/code-of-practice-2 [Accessed 4 May 2017].

[50] Financial Conduct Authority, 2017. *GC14/6 Social media and customer communications: The FCA's supervisory approach to financial promotions in social media* [ONLINE] Available at: https://www.fca.org.uk/publications/guidance-consultations/gc14-6-social-media-and-customer-communications-fca%E2%80%99s [Accessed 4 May 2017].

[51] The UK Government, 1998. *The Data Protection Act* [ONLINE] Available at: http://www.legislation.gov.uk/ukpga/1998/29/contents [Accessed 4 May 2017].