CLINCHAPP
A WEB APPLICATION FOR PRODUCT DISCOVERY AND EXCHANGE

Interactive Multimedia Design

Daniel Skelton | B00594246

Simon Frazer | PSG09

May 2015

Acknowledgements

This project would not have been possible without the guidance and support received from my mentor Simon Fraser; who has supervised each stage of the project. Regular meetings with Simon helped me through critical changes during the development stage of this project which reassured me of my decisions. I would like to thank Simon for being a fantastic mentor who was always on hand to answer any of my questions and showed a keen interest in helping me, and others.

During the course of the project I received help and advice from many people for which I am really grateful. Thank you to Ryan Grieve of rehabstudio, for helping me resolve several important issues in developing this application, for which I greatly appreciate for the time you put into helping me comprehend and learn from my mistakes. Thank you to Neil Mccallion and Paddy Carey of rehabstudio who had provided some early on advice in developing the project to help point me in the right direction.

To George Moore, I would like to say thank you for providing insightful material and lectures that inspired me to improve upon my development skills and knowledge in this project.

Finally I would like to thanks to some of my peers, including Stephen Martin for being a consistent and reliable person who was always on hand to answer any questions in regards to the creation of this report. To Cat Jackson and Aisling Rasdale for their scrupulous note taking and constant reminders of relevant tasks to be carried out.

1. Introduction

Websites like gumtree and eBay provide a service based on products and goods that allow users to exchange items. Services like these often lack social integration and can become cluttered with advertisements. The purpose of this project is to provide an intuitive interface to promote item discovery and exchange via social connections. Users of the service should be able to easily sign up/log in to add items to their profile without any distractions.

1.1 Project Aim & Objectives

To produce a rich web application, acting as a service that allows users to discover and exchange items they own with one another.

Several objectives have been outlined that will form the initial building blocks of the project scope. These are initial objectives and will be subject to change depending on further analysis of methodologies and an in-depth overview of project requirements.

Project objectives at a high-level include:

- Creation of user profiles, requires registration system with server-side auth
- Server provisioning and configuration on Digital Ocean VPS
- Database Schema Design
- Back-end model/view/controller architecture
- Front-end UI/UX design
- Creation of a RESTful API for client side views
- Paper prototyping and sketching from initial project ideas
- High Fidelity Mockups
- Configure Amazon AWS S3 Bucket for data storage
- User testing
- Unit testing
- Modular front-end javascript architecture
- Third party api integration for sending emails (Mandrill API)

These objectives will be divided into smaller low-level development tasks during each of the allocated objective timeframes. Several of these objectives are assumed to take much longer than their counterparts, therefore a detailed project plan will be devised to effectively select a timeframe for each objective. This will outline what can effectively be produced by providing an estimation based on past development experience and discussions with professionals in the industry.

1.2 Project Report Overview

This project seeks to follow several phases that are required to complete the project from its initial inception to deployment. This report will outline each step at a low-level and discuss the underlaying components of each task and how it affects the project. The steps below are a guideline into the core report structure.

1.2.1 Planning

Idea generation and discovery are two of the most important aspects of the planning stage and form the basis of the project. This stage also looks at defining project requirements, user requirements and uses relevant research into the subject to provide a feasibility overview of the project. As discussed in aims & objectives, each objective will be critically placed onto a project plan gantt chart to obtain a rough estimation on how long each task will take. This will also lead to a better understanding of task priority.

1.2.1 Design

The design phase evolves through numerous iteration cycles from beginning to completion, beginning with paper prototyping, where initial sketches and wireframes are developed based on the outcome of the idea generation phase. These preceding iterations are used in the development of an initial user experience design. When critiqued, this is used to develop a refined user experience design which should be a close reflection of the application interface and brand identity. This section will take a detailed overview on the refined user experience design, focusing on how it has been improved from the initial UX design.

1.2.3 Development

The application will be developed in accordance with a selected software development methodology that will seek to improve developer workflow and time management. The refined UX design will be used to develop the interface of the application. Development technologies, tools and frameworks will be outlined and reasons for selecting each will be clearly defined. In addition, Database design and relational database modelling will be outlined that should help in comprehending the over-arching application structure. This section will reveal at a low-level the software development patterns used in both the front and back-end of the application.

1.2.4 QA / Testing

Testing the application to ensure that it has been developed bug free is an important step before releasing the product. This phase involves writing several unit tests of the front and back-end application code to rule out bad quality code. This phase will also focus on several beta users of the application and their responses of the product.

1.2.4 Deployment

As outlined in the objectives, the project will be deployed to an instance on Digital Ocean. Doing so will ensure full ssh and root access to the server. This section will outline the reasons for using an external hosting service as well as accounting for risks, it will also outline the process required in its provisioning.

2. Concept Definition and testing

2.1 Idea generation

This project formed out of 3 initial conceptual ideas. The application, known soon as Clinch, was chosen as one of the most feasible ideas to develop in the available timeframe.

The original proposal arose from the idea of having an application that allows users to swap items with one another, whilst this idea seemed somewhat unique, it has already been implemented before and not with huge success. Later it was decided to alter this idea and allow users to communicate through email about exchanging items through the application interface, a similar model to Gumtree, but still unique in its own right.

To develop these initial ideas a list containing areas of interest was created, subsequently followed by a mind-map to help stimulate the production of ideas. In addition to this, some research was carried out into what problems people had with applications on the web. This was done to gauge whether they had any original ideas for a web application or anything that is currently out there that could be improved upon. The outcome of this resulted in several ideas for a web applications based around social item discovery, which was inline with ideas produced from previous brainstorming sessions.

2.2 Requirements Specification

2.2.1 Assumptions

As the main developer on this project specific assumptions will be made about how the application will function, what environment the application runs in, down to how users will interact with the application. These assumptions are listed below:

- A user will not need to enter credit/debit card information upon registering.
- The application will not take responsibility for users misusing the application. E.g. If a user agrees to exchange an item with a user and does not.
- Users will be able to follow users and receive updates when they add items.
- Users will only be able to add an item to their account if they tag it at least once.
- Users will only be able to add an item to their account if they have at least one image for that item.
- Users will not be able to add an item to their account with the same name as a previous item on their account.
- Users will be able to search the system for items.
- Users will be able to infinite scroll through search items to prevent large database load.

2.2.2 Exclusions

As the requirements are being outlined, it's equally important to define whats not possible and what this project will not be delivering.

- The project will not support IE8 or below
- The application will not support instant messaging between users
- The application will not support video streaming or video uploads
- The application will not support gif image types

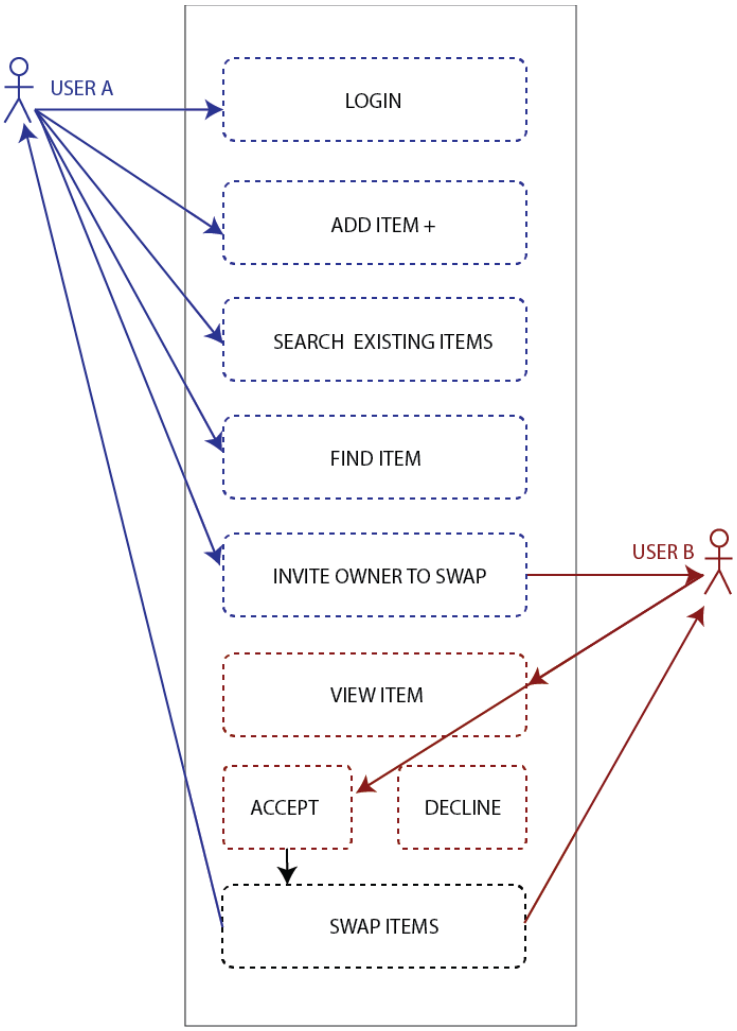
Excluding these from the requirements means that there will be more time to focus on the core functionality of the application than worry about browser bugs with IE8, or instant messaging between users, which is a functionality that can be added to the backlog for implementation at a later date.

2.2.3 Defining Scope: Use Case Diagram

Project scope is defined as the part of a project that involves documenting and listing project goals and deliverables. Defining scope is important as it allows the project to consist of planned features, instead of features being introduced half way through the project, commonly known as scope creep. (jamasoftware.com, 2013)

The main scope of requirements in this project will be based upon how a user may interact with the system. To visualise this, A use diagram has been created to help map out the process that a user may take once they log in. This is shown in the diagram below:

- User A starts off the process by logging into the system.
- User A then decides that they wish to add a new item.
- After user A adds an item, they begin to search for items they are interested in.
- User A finds an item and views the profile of the owner.
- User A chooses to follow user B to receive updates when they add new items.
- User B can view their followers to see their recent followers.
- User B uploads an item that user A is interested in
- User A views item and sends user B a message of interest in item.



2.3 Paper Prototyping

Paper prototyping contributes to the planning stage by introducing some creativity in regards to the application interface. Initial, rough sketches where created to help put ideas to paper. This allowed for a greater clarification of the proposed idea. Areas that were deemed important to begin developing where conceptually sketched, which included a search view, that enables users to search for items uploaded by other users. The search view would enable features such as searching for items within the users current location, and lazy loading of items to provide a better browsing experience, it was assumed that thinking of these ideas early on would be beneficial moving towards refined designs.

Adding items to a users account was an area that requires early thought, users could have the ability to upload multiple images per item. This feature is important as it will allow the user to express their item through several images, and should provide better satisfaction of the service.

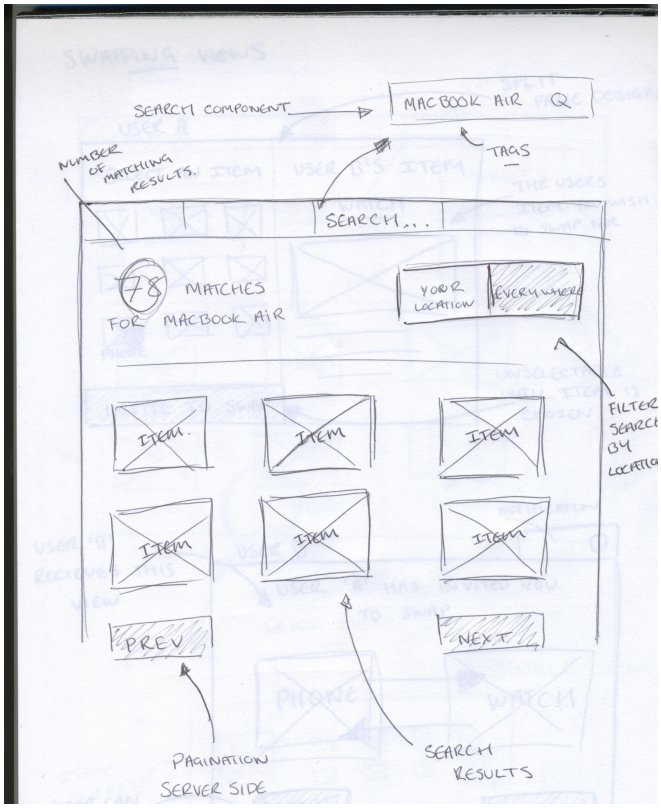


Figure 2: Paper prototyping application views

The purpose of prototyping views for the idea is to identify features that may or may not work before venturing further into the design process. Therefore it was important to prototype several other views to provide a clearer analysis, on still quite a high level, of features that could be included.

As part of this process, a 6-UP was created to help destruct a view into 6 different possible iterations. The purpose of this was to creatively express alternative scenarios that could arise from thinking about different ways of structuring layout. Out of this 6-UP, one iteration stood out that made more sense than the rest. It defined an individual product view and how a user communicates with another user to establish their interest in an item.

Its often much easier to scrap ideas on paper but not so much when you begin to spend hours perfecting a design that may not be effectively expressing the application to its best. All the views that were thought needed at this stage have been prototyped to mitigate with this problem.

A breakdown of the relevant views are as follows:

- Homepage
- Login / Register
- Search
- User Profile
- Add Item view
- Detailed item view

At this early state its not important that these designs remain consistent, thats not the aim from this task, instead its more important to rule out specific ideas that may not work. Identified these can save design and development time in the project.

2.4 Feasibility Testing

Due to the complexity of the project, building a functional prototype of some core features in the project is essential to ensure they could be developed. As the application will allow users to upload items to their profile, a core aspect of the application will involve the creation of a user profiles, therefore a login/registration system needs to be designed in order to facilitate this requirement. In addition to this, users should have the ability to change their avatar and bio.

Ensuring users can upload items was also equally important once the user is logged in. A brief implementation of this was designed to ensure users could have multiple items associated with their account.

It was also important to test the dynamic nature of the application by emulating data that could be returned from a RESTful service. A collection of items as JSON data returned from the server will enable client side rendering to be tested with the use of templates with client side views and models representing the data.

The functional prototype was successful in regards to ensuring these features could be developed as part of the current system. It was an unexpected outcome that developing these features would result in very few issues, although several were brought to light in the development of this prototype, including the complicated mix of client side and server side rendering. In its current form the prototype seemed frail as it relied on exposing global variables to the window when navigating between page views so the client side could fetch items for a particular user.

Ideally global variables should be kept to a bare minimum due to it being a commonly known bad practise as objects defined on the window can easily be manipulated through the browser console. The purpose for the inclusion of objects in the window for the prototype was used for simplicity in prototype to avoid over engineering solutions.

This functional prototype used tools and development practises that would be used in the development of the final application.

Although it was noted that the application would run off a RESTful API in the back-end. Developing it for use in this prototype did not make much sense as its purpose was to outline the feasibility of the project.

As client side rendering will become a large aspect of the application for generating the user interface. The prototype shows how this could effectively be done using client side templating and view logic for rendering the template into the DOM with the appropriate data. Below is an example of a client side view that inherits from a view by the popular backbone framework.

```
/**
    ItemView
    @extends backbone.View
    Representation of a single item in the item list
*/
class ItemView extends Backbone.View {

    constructor(options){
        this.tagName = 'li';
        super(options);
    }

    initialize(){
        this.template = _.template($('#user-items-template').html());
    }

    render(){
        this.$el.html(this.template({ item: this.model.toJSON() }));
        return this;
    }
}
```

Figure 3: Functional Prototype ItemView

In this ItemView the template is defined as 'template' on the itemView, therefore being accessible from any of its methods. The render method is responsible for rendering the template into a DOM node and passing data into the template so it has context to render. The prototype didn't really require this level of architectural design just to render some items, but it was important to design it this way due to it being a strong design pattern that may be used in the developing of the real application.

2.5 Methodologies

Project methodologies define the process that is responsible for maintaining structure in software development. Three main approaches consist of waterfall, agile and prototyping. (noop.nl, 2008)

2.5.1 Waterfall

There are many methodologies that exist for web/software development. The most popular is the waterfall, an old and traditional methodology that evolved from manufacturing. Projects that use this approach tend to have a well documented requirements and are small in size. It follows a standard linear approach from initial design through to end user testing. (seguetech.com, 2013) This methodology could be useful for someone building an application that knows exactly what they need to accomplish, using this methodology on a project like Clinch could become problematic as the project has the possibility of becoming complex at specific stages.

It is established by a linear approach, each task must be fully completed before the next task can begin and is most suitable for projects that have requirements that are unlikely to change.

A big advantage of the waterfall approach is that is simplistic in nature and easy to mange, although some disadvantages include the inability to adopt to changes in requirements due to difficult tasks or client needs.

2.5.2 Agile

The agile methodology is a relatively modern one that originated from the Agile manifesto, usually used by large teams to manage development tasks. Agile uses a range of methods from scrums to extreme-programming and story-driven scenarios. It is focuses on rapid cycles of development tasks normally known as sprints. Sprints are made up of tasks, and are commonly defined by user stories. A large advantage of this is that it is flexible in regards to a change in requirements. (agilemethodology.org, 2015)

Tasks are normally defined by priority level which enables the most critical features in a sprint to be established ensuring the likelihood of success. Scrum boards are used to provide a visual

representation of the tasks outlined in the sprint. Any tasks that are deemed unimportant or is an area for future improvement are put into a back-log.

2.5.3 Prototyping

The prototyping model uses initial requirements to build a quick prototype of the application, it therefore helps to define the project requirements. This kind of model works well with high end user engagement and allows users to put forth their ideas on features they would like to see within the application (istqbexamcertification.com, 2015). Prototyping is often seen as an attractive methodology for complicated systems due to its ability to help define their requirements, although it may lead to increasing the complexity of the project scope by continually moving beyond original plans.

2.5.4 Methodology selection - Agile

An agile approach to software development ensures that tasks are developed in tandem with the project requirements. Although agile is commonly used in team development, it benefits can be seen with a single developer by utilising tools such as scrum boards and issue tracking. Im addition to this, testing becomes much easier as tasks are broken down into testable segments defined by business readable specifications (BDD) Behaviour driven development.

Development tasks will be defined week by week and sorted by level of priority ensuring that each task is given adequate time for completion. Future improvements of the application will be pushed into the project back-log.

3. Design

The application relies on fundamental design patterns that encourages strong cohesion between components. This implies both to the software and UX design where strong design patterns improves the quality of output.

3.1 UX Design Evolution

The initial user experience design evolved from conceptual sketches and ideas that took the initial project ideas from the idea generation phase and began to experiment with layout and composition of core user interface components, such as the navigation. It focused on more on visual aesthetics, colour and layout of page views. It helped define some of the core pages such as the search page, focusing on an old conceptual idea of swapping items, an idea to be dismissed and evolved later in the development phase. This was the first iteration of designing on screen and provided a useful platform to improve upon after critical feedback was obtained.

The feedback form the initial UX design showed a little confusion around swapping items, not the sole reason for the eventual dismissal of this idea but it played a small role. The feedback was used moving forward to develop a refined UX design which used aspects of the initial UX design and drastically improved areas such as colour, layout and typography, thus will hopefully improve user experience.

The refinements to the UX design ultimately lined up with the new iteration of the application which dismisses the idea of swapping items in favour of item exchanging via email communication. The new interface drastically improved the consistency of UI elements and focused on improving space. The application sidebar was removed as the length of the navigation items didn't warrant the need to include one. Instead the top level navigation was included horizontally at the top of the application, which improves application view real estate.

One of the main points of confusion in the initial UX design was the homepage, therefore a detailed overview of the UI /UX on this view is necessary to justify how the changes made positively impacts the application moving forward.

The redesigned homepage incorporates a new colour scheme which is much more appealing than in the initial UX design. It emits more vibrancy and has a playful personality which will be highlighted in the application with a range of smooth animations when routing between views. The main navigation has been re-designed to maximise space in this view. An additional sub navigation is embedded to route between user items, following and followers etc. This gives user the ability to easily manage their items. The image below reveals how the active state of the application should look. It will be expressed via a url to directly navigate to the route. E.g. /profile/items.

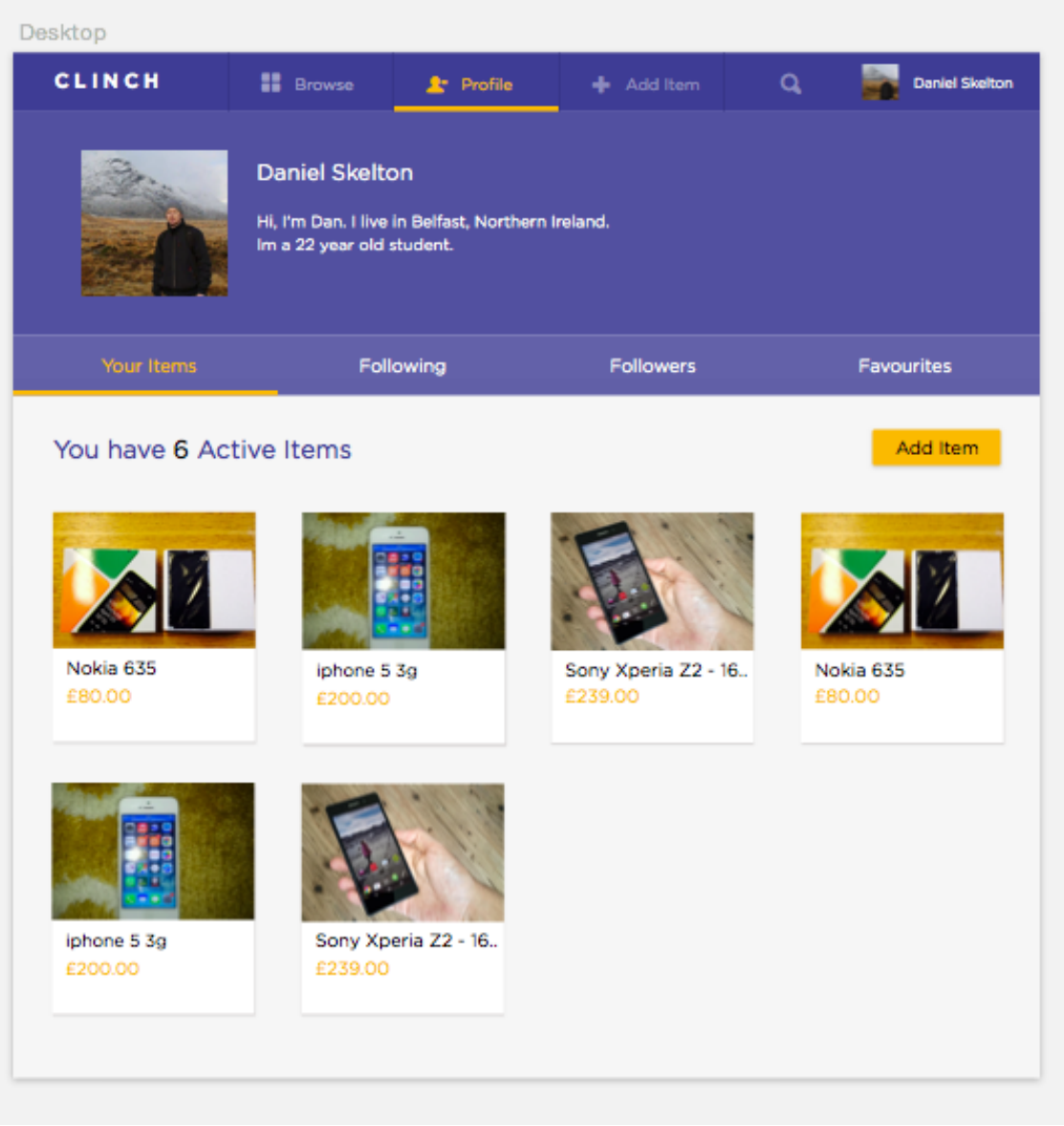


Figure 4: Refined user interface. User profile layout.

Important user profile information is displayed at the top of the view. This includes the user's username, profile picture and a short bio. In its current state the view below shows how the user profile would appear for a logged in user viewing their own profile, well, exactly the same layout will be used for users who view other users profile. It's implementation can be compared to Facebook's user profiles, where when viewing you're own profile specific buttons on the page will appear that do not appear for other users.

Updating the information displayed on your profile should feel natural. Most applications require you to update via a settings area. The top-level navigation reveals the current logged in user, with a small avatar and their name. Clicking on the name will take the user to their settings area where they can update their information including adding a new avatar and adding a cover photo. This will replace the purple background behind the user's avatar and will enable each user profile to have a certain level of uniqueness.

The overall re-design is mostly inspired by Google's material design guidelines, which focuses on how objects are expressed within our 3D environment. It focuses on how ambient light and shadows help define objects to give them context and meaning. Subtle borders and shadows are creating on buttons and item containers to lift them gently off the page.

The design of the items has been inspired by the card component that appears in google's material design guidelines. It can also been seen in their google+ service. As part of their usage definition they state.

"Cards are a convenient means of displaying content composed of different elements. They're also well-suited for showcasing elements whose size or supported actions vary, like photos with captions of variable length."

This best describes the nature of the item component in this application. As consistency of UI elements is important for users, the same item component appears in various states in the application, including when you search for items.

3.2 System Design

This section takes an in-depth look at the over-arching system design, that is, how the application will function as a SPA (Single Page App) and the technology that enables the front and back-end to become decoupled, almost to the point where they can be ran off completely different servers. Though that is not the main goal.

3.2.1 RESTful API

REST is an acronym for the term, representational state transfer. It is a software architecture pattern commonly used in the development of web services. REST runs off the HTTP protocol (HyperText Transfer Protocol) and is established by 6 main constraints (restapitutorial.com, 2014).

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on demand (optional)

The constraints define how a RESTful web service operates. Uniform interface describes URI's (Universal Resource Identifiers) that are used to link to resources, therefore URI's must be constructed a uniform manner to match a resource, these URI's are represented in the form of /resource/:id.

A stateless server in REST is important. This essentially means that all requests sent to the server must contain the necessary state to handle the request, combined with HTTP verbs such as GET, POST, PUT, DELETE and the non-standardised PATCH. The server must not handle or know about the application state, therefore this is left to the client.

In relation to this application. Two main resources will be users and items, the combination of these resources can also be applied and is still considered RESTful. E.g. /users/2/items/5. Which would return the item with id 5 of the user with id 2.

The back-end of the application will be built as a RESTful API, written in python. Building the application this way will a substantial amount of work, but enables the user to receive a truly 'native' web application feel that is truly fast and responsive. When logged into the application, the server should never return any HTML when switching between states (I will refer to pages as 'states' as a more accurate description of what they are). What happens in this case is the front-end requests the data it needs from the api, then builds the view from the data received.

The data for the application is exposed via the api to allow other services, applications and devices to use this information, providing they use the correct credentials. For example: The application could also be built as a mobile app alongside this application. As the API allows for the data to be 'decoupled' If you like from the application logic, this means the application has the potential to scale at a much faster rate.

3.2.2 Development Environment

Vagrant has been used in combination with VirtualBox to provide a virtualised environment for installing project dependencies. This is done to replicate the production environment as much as possible. Nginx has been chosen as an HTTP server due to it being more lightweight and its ability to handle more simultaneous connections. The database will also be installed from within the virtual machine, again do so provides practise for deploying the application into a real environment.

3.3 Logic Design

This area focuses on the the architecture patterns used on both the server-side and client-side of the application. It outlines some of the common architectural design patterns used in web development today as well as introducing a new emerging pattern in front-end development.

3.3.1 Examining Existing Software Architecture Patterns

MVC

Many developers advocate different architecture patterns for developing applications, one of the more popular patterns is the MVC pattern, (Model-View-Controller) which promotes a separation of concerns whereby each part of the system deals with something different to other parts. Many variations of the MVC pattern exist such as:

- MVA (Model View Adapter)
- MVP (Model View Presenter)
- MVVM (Model View ViewModel)

Although a common trait seems to be removing the controller from the equation and substituting it with with another component. In MVP, the presenter takes the controller position, whereby all presentation logic is pushed to the presenter. The pattern has been widely implemented in languages with success such as .NET. (codinghorror.com, 2008)

In traditional MVC, the model is a representation of the data and does nothing else. Views display the model data and sends user actions to the controller, which in turn requests the data from the model and returns it back to the View to be rendered onto the page. The controller acts as the middle man in this paradigm communicating with the View and Model.

Client-Server Model

The Client-Server model is another architecture pattern that can be useful for prototyping applications but provides very little of the way of separating concerns when programming. (webopedia.com, 2014) Take a PHP application for example, when the server receives an HTTP request, in a client-server approach the PHP code will execute, perhaps run an SQL query, then inject pieces of HTML alongside that logic and return the response to the user.

Three-Tier Pattern

An implementation of the client-server approach that helps decouple the architecture is the multi-tier architecture or commonly known as the Three-tier architecture pattern. This pattern is split into 3 common tiers. Presentation tier, Logic Tier and the Data Tier. Often these tiers are completely separated sometimes onto separate servers so they are physically separated.

3.3.2 Server-side architecture

An MVC architecture will be followed throughout the back-end of the application to adhere to a separation of concerns between view layer and business logic.

In order to improve performance and speed of the application. All user interface rendering has been moved from the server onto the client. There are many reasons why this option has been chosen. One of the most important being speed. Since all rendering is performed on the client, the server only needs to 'bootstrapped', in this case meaning that it only needs to serve one html page, navigating through the application should be noticeably smoother due to the browser not needed to download and parse page resources for every page request.

Views should be responsible for returning multiple facets of content. In the homepage, login and registration pages they should be responsible for returning HTML, although for the API the views should return JSON. Regardless of this, the views are still performing their role within the application and still remain adhere to a separation of concerns.

Models will be used to define the database schema and will contain several methods for saving the models to the database. Database migrations will be performed (manually) when changes to a models file occurs, this means that the database will be automatically updated to reflect the current database schema. Database migrations are performed by using a command to generate a migration file of changes applied since the last migration file was created, and then apply it via another command.

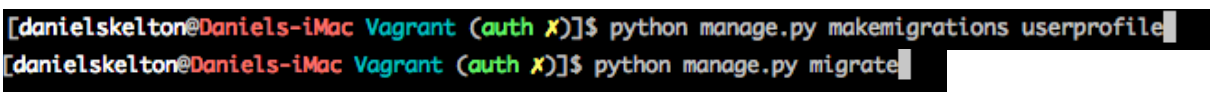


Figure 5: Database migration on user profile class in model

3.3.3 Modular javascript architecture

The modularity of the front-end is extremely important within this application as it is responsible for a large proportion of the application business logic. For example, client side routing. The only server-side routes the application will render is the homepage, login and registration pages. Once a user logs in, the client will handle routing.

With the introduction of ES6 (Ecmascript 6) and ES7 there has been a lot of changes to Javascript. One of the most useful features that will be widely used within my application is the introduction of a module system. This gives the opportunity to greatly decouple components in this application by importing and exporting modules. Javascript modules can be used in ES5 today using tools such as browserify and require.js.

Browserify is a tool that lets you 'require' modules in the same syntax as Node.js, it is widely used as it can allow for a phenomenon called 'Isomorphic Javascript', meaning you can run the same code on the server and the client. Again, why is this an exciting area for the web development community? One of the most important aspects known of is progressive enhancement. Here the server can render any route a user visits via the server, then the client can take over and subsequently render the rest of the routes.

Using this module system won't necessarily make the functionality of this project's code any better, but it will allow for a greater separation of concerns and improves productivity as the code is not in one large javascript file.

The directory structure of the application will be separated into multiple directories for various modules and components instead of one large javascript file, modules will then be imported and compiled into a main minified file at build time. This means the project will be easier to maintain due to the separation of these modules.

The main benefit of this approach is code re-usability, generalising components to be non-specific gives the ability for them to be re-used through the application. Examples of this will be shown throughout this report.

3.3.4 Component based user interface

The introduction of web components is described as the future of web development based on encapsulated and interoperable custom elements that extend HTML itself. (polymer-project.org 2015). A strong example of this is Google's Polymer library. Creating re-usable small components allows for an increase in code reusability, this also means code can be tested as stand alone components. This is interesting as it could greatly improve testing.

React, a user interface library developed by Facebook, is based on a component model albeit with a slightly different implementation. React doesn't render to the shadow DOM like Polymer, instead it renders to a Virtual DOM,written in javascript (facebook.github.io, 2015). There are a number of reasons why this could greatly improved the performance of this application.

1. React uses an algorithm that diffs the virtual DOM with your current DOM tree, to compute the minimal amount of DOM manipulations needed to bring your UI up-to-date. (This has been compared to how services like git performs diffs to compare commits)
2. React gives you stateful components, this means that a component always knows its own state within the application.
3. React has an imperative API for updating the DOM, you should never update the DOM with jQuery as at that point you've broke everything React stands for.

3.3.5 Flux architecture over MVC convention

React is often described as the V in the MVC, being only responsible for the view layer in an application, this means that application models and controllers still need implemented. Facebook recently announced a new architecture for building client-side web applications that complements React's composable view components by utilising a unidirectional data flow. Its called Flux.

Experience gained from client side MVC revels that models and views often become too coupled as the application beings to grow. For example: A view could trigger a model to update which could trigger other views to respond and update their UI.

Flux proposes unidirectional data flow (facebook.github.io/flux, 2015). Meaning that components do not trigger actions on models directly, all data propagates downwards throughout an application.

Relevant components can subscribe to stores, so that can have a call to action when stores receive data from the API. This is often referred to as a publish/subscribe pattern and promotes loose coupling of components.

Two important aspects of Flux are actions and stores, stores are used to represent data. They trigger and pass data it to all listening components. It is in these stores that I will store my business logic for interacting with the REST api. When a components needs data, it uses actions to interact with the stores. However, MVC is also a viable architecture pattern on the front-end and can be used alongside React too. Models and Controllers don't need to be written inside any sort of framework and can easily be replicated using regular Javascript using the revealing module pattern. (addyosmani.com, 2015)

3.4 Data Design

3.4.1 Relational Database modelling

This application will use a relational database as Django does not support NoSQL. NOSQL databases such as MongoDB use a different technique where it uses multi-dimensional data types such as arrays to store nested relationships.

MySQL was chosen over the originally intended Postgresql due to past experience. Most of the interaction with the database will be achieved through Django's ORM. (Object relational mapper) This is a unified API that acts as a facade / abstraction to perform SQL statements behind the scenes of the framework. The question may be asked, why is this needed?

The ability to perform SQL statements via an ORM means allows for greater portability of the codebase. Again, this is why developers love implementing abstractions. The abstraction here means that developers can use the API of the ORM that will never change, even if you swap out the database type. In this scenario, normally you would have to switch the SQL statements to interact with the database you are using. Its likely that these statements are also used all over an application. Its immediately apparent why using the ORM is such as good idea.

When you specify a model in Django, that model represents a table in a database, you can add methods on the model for interacting, formatting the data. It also provides database migrations, which is Django's way of propagating changes you make to your models(adding a field, deleting a model etc) into your database schema. This will be useful when developing the system if updates are needed to tables.

3.4.2 Entity Relationships

Modelling a basic relationship diagram was important as it will help in demonstrating how the database will be constructed. A basic model was created to demonstrate the relationship between the tables. Each of the Many-to-Many relationships, (categories, tags etc) will be represented by intermediary tables in the database.

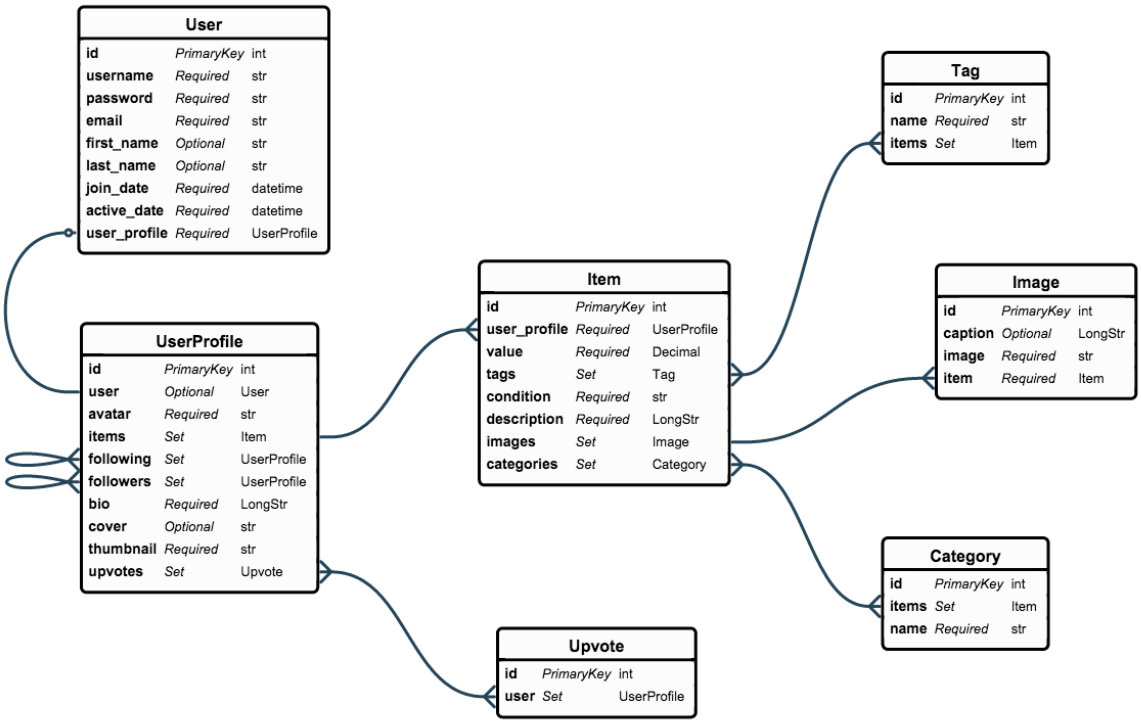


Figure: 6: Revised entity relationship diagram

The entity relationship displays the relationships between 7 entities, these will form the core tables within the database. Datatypes have also been included in the diagram. An overview of the 7 database tables as defined in the ER diagram is shown below:

- User
- UserProfile
- Item
- Image
- Category
- Tag
- Upvote

A relationship between a user and a user profile is 1:1, when a user registers for the application, a new user profile will be created for that user.

A User has many required fields such as username, password and email which are defined on the user model when they register. Other fields are optional such as first_name and last_name. The relationship between a User and an Item is Many-to-one whereby a user can add many items although each item must belong to one user.

Tags and Categories have a Many-to-many relationship with an Item whereby an item can have multiple tags and a single tag can be used by multiple items. This relationship is essential for how the search functionality will work.

The diagram shows that the database will consist of 4 junctions tables sitting as the bridge between many to many relationships. These include:

- Item_tag
- Item_category
- Userprofile_following
- Userprofile_followers

These many to many relationships gives a strong indication as to how the application will function. They express the issue of enabling multiple tags to be permitted for each uploaded user item and describe how a user has a many to many relationship with its own entity.

4. Implementation

4.1 Technology / tool selection

This application relies on a number of technologies to power the front and back-end of the application. These technologies, especially on the front-end, will become common place in the web development industry within the coming months/years. This section focuses on some of the tools and technologies in the application, but also discusses the build tools that are not evident in the application despite playing a vital role in all development phases.

4.1.1 Front-end build system

The front-end technology stack features a front-end build system powered by Gulp. Gulp is a javascript task runner that will run compilations of code, as well as minification, bundling and linting. It is a command line tool built on node.js and is massively popular in the web development community along with other build systems such as Grunt. Gulp is used for compiling the Sass code in my project to CSS. It is also used for compiling ES6 javascript and React to ES5 using a javascript compiler called Babel (babeljs.io, 2015). Gulp is configured to watch for changes in file directories and automatically compile when it notices a file has been changed. It utilises the power of node steams and its pipeline, making the build process extremely fast as it doesn't write intermediary files to disk. For example, multiple operations can be performed on a file such as listing, then minification, then concatenation with other files, in the end you could specify a directory to write the file to.

The role of a build system was important within this application to run ES6 code through the Babel compiler, another use case was that it allowed the code to become easily portable. Front-end project dependencies are specified through the package.json file, meaning that once the build system was created just one command was needed to install the dependencies and start using the system. Therefore the ability to move the project to various machines became trivial.

The command would look somewhat like the following to install these dependencies.

"npm install && gulp build"

NPM (Node package manager) installs the dependencies into a folder called node_modules, its often good practise to add this direct to .gitignore and install dependencies via package.json every time a repo is cloned.

Using a tool called browserify, libraries such as jQuery, can be imported into the project. This is done using the same syntax as node. If a path is not specified, it is automatically resolved from the node_modules directory.

4.1.2 Sass (CSS extension language)

Sass is used and compiled by Gulp to CSS on build time. This means that maintaining application code becomes much easier with the ability to programmatically create functions and mixins that will compile to CSS. Variables can also be created and referenced throughout the application, that makes it much easier to change a variable and have the change be reflected everywhere its referenced. The Sass code is also split into small modules and imported into a base file. This improves modularity and becomes much easier to maintain. An example of this from the project is shown below:

```
//COMPONENTS

@import "./components/loader";
@import "./components/searchBar";
@import "./components/buttons";
@import "./components/item";


//LAYOUTS

@import "./layout/search";
@import "./layout/itemDetail";
```

Figure: 7: Component imports improving modularity

The benefits of using a CSS pre-processor like Sass in this project has improved developer workflow with the ability to ability to perform inheritance through the use of *@extend*. This is just one of the many Sass features that was used during the development of this project. It allowed components such as buttons to be derived from a base button class, therefore inheriting all its stylistic properties.

4.1.2 React (Javascript view layer)

React.js is a component based user-interface library, written in javascript and developed by Facebook. It is used throughout the application to build *stateful* components that represent the UI.

Emphasis on stateful, React's core principal is that these components should always have an awareness of their own state. In React you should never update the DOM manually or retrieve data from the DOM because the DOM is not the 'source of truth' and is subject to manipulation / deletion. (andrewray.me, 2015)

React can be compared with various web component libraries like Google's Polymer, in that it allows you to create encapsulated and re-usable components, but the implementation is vastly different in the underlying architecture of these two libraries.

When rendering a React component you never render to the real browser DOM, but instead a virtual DOM implemented in javascript. An algorithm, O(n), is used to perform a diff between the virtual DOM and the current DOM tree in the browser to find the minimal DOM operations needed to bring the UI up to date (facebook.github.io, 2015). This diff can be compared to how Git uses diffs for code comparison, although no comparison is made between the two underlying algorithms here.

React also helps prevent against DOM layout thrashing, a phenomena that forces the DOM to perform a 'repaint / reflow' that kills browser performance. As react only updates the DOM according to whats changed it becomes much easier to build a fluid web application that is responsive.

Common practise in the web development industry regarding front-end development was to usually separate your view logic into separate template files, that are parsed by a templating engine, such as Handlebars. Whilst this is still better than 'jQuery soup / spaghetti code' and promotes a separation of concerns, it adds another layer of abstraction into the mix and leaves you as a developer having to succumb to the limitations of the API.

Take for example Handlebars or any templating language, they have a reliance on primitive abstractions such as {{#each}} to loop over objects, but often you need to jump outside of the templating engine back into javascript to write a custom 'helper' as they call it, which is a function written in javascript that enables the templating engine to use, because you could not perform the operation directly within the tempting language itself. Its significantly underpowered compared to javascript. Pete Hunt, a software developer at Facebook who actively worked on the React codebase gave a great talk on this in his presentation titled "React - Rethinking best practises". (Pete Hunt, 2013)

Getting to the point, React expresses the power of javascript to build the UI in a much more expressive and functional manner and does not use a templating engine, instead uses a dedicated stateful component. This is the reason React was chosen for this application, it provides much more power and performance.

Below is a good example of a stateful component. It features the searchList component in this application.

```
    render(){

        if(this.props.loading && !this.props.data.length){
            return <Loader />
        }

        if(!this.props.query.tags){
            return null;
        }

        if(!this.props.data.length) return <h1>{this._suffixCount(this.props.count)}</h1>;

        let {data, count, loading} = this.props;
        let canLoadMore = data.length < count;

        return (
            <div className="searchList">
                <h1>{this._suffixCount(this.props.count)}</h1>
                    <div className="results">
                        {data.map(item => <UserItem item={item} key={item.id} />)}
                    </div>
                {loading && <Loader />}
                {canLoadMore && !loading ? this.canLoadMore() : null}
            </div>
        )
    }
});
```

Figure: 8: Stateful react component: SearchList

In react, ‘Props’ are immutable data structures passed from parent to child components and ‘State’ is data defined on a component. This example shows the stateful nature of this component and its ability to render various states depending on various conditional statements. A really great part the react render method, arguably, is that it looks like if contains HTML, but there is no string concatenation.

It is in fact not HTML. Its called JSX and was built to express their virtual DOM components, using this requires a pre-build step that converts the JSX to regular javascript objects. Hence, the reason for wrapping the HTML in a return statement is that its just regular javascript in operation, meaning its incredibly fast.

The example also expresses how individual user items are rendered. Inside a template language this would normally be executed with an each statement, although in react its a common pattern to map over some data and return a child component passing in the data as props.

4.1.2 Django (Web Framework, Python)

The server-side of this application needed to be in a language that was not only familiar but one that enhanced productivity. There decision to use Python came from an analysis between it and PHP. Whilst PHP itself was not the intriguing factor, the Laravel framework written in PHP has grown rapidly in popularity with a lot of admiration in the PHP community. The temptation to use this new framework was quite high because of this. Having said that, it was a new framework to learn with the downsides of having to spend ‘x’ amount of time getting to grips with its API / general syntax. Python offered a web framework known as Django, which wasn’t an unfamiliar framework, resulting in an assumed higher productivity rate. It also provides great exception handling and is an overall strong, stable and well proven language for web applications.

Django is a high-level web application framework and is used to power many popular websites such as Instagram, Pinterest and Disqus. This framework was chosen to power Clinch on the server as it has excellent community support and documentation. It also provides a solid MVC structure enforcing a strong separation of concerns. The strong

templating language in Django allows the developer to extend from base templates to perform inheritance and is extremely expressive.

Django was used to build the back-end rest API along with the popular Django rest framework (built by Tom Christe) for this application. The API is built by serialising data from the models and building endpoints that serve it. The serialised data is in the form of JSON (Javascript Object Notation) as it is a widely supported, lightweight and well known data-interchange format for the web. In this REST framework, the process of serialising data from models is performed inside of serialisers. These allow complex data such as query sets and models instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serialisers also provide deserilization, allowing parsed data to be converted back into complex types.

Below is an example of a serialiser being used in this application to serialise a user follower.

```
class FollowsSerializer(serializers.ModelSerializer):

    username = serializers.CharField(source='user.username', read_only=True)
    email = serializers.CharField(source='user.email', read_only=True)

    follow_status = serializers.SerializerMethodField('is_following')

    def is_following(self, user):
        request = self.context.get('request', None)
        count = request.user.profile.follows.filter(followers__follows=user).count()
        return count > 0

    class Meta:
        model = UserProfile
        fields = (
            'id',
            'avatar',
            'thumbnails',
            'bio',
            'follow_status',
            'username',
            'email',
        )
```

Figure: 9: User Follower Serialiser of Profile Class

In this example you can see inside the Meta class is a reference to the *UserProfile* model of another user. The fields tuple specifies what model fields to return, it is worth pointing out here that 'following' inside fields is not included, as that would probably result a race condition or infinite loop in the application.

Custom functions can be created inside serialisers to perform operations such as the is_following function above. This function queries the database to check if the current signed in user follows this user by analysing the length of returned dataset from the query.

Creating serialisers allow for a resource to be serialised, but for this application many to many relationships need nested serialisers to represent a nested JSON output inside the API.
This process is performed by using one of the fields shown in the figure above as an instance of another serialiser instead of a string, where inside that serialiser could have references to other serialisers. The pattern continues in that fashion where you can have as many nested serialisers as you wish depending on the relationships within an application.

The API should be responsible for processing values to determine whether they are "truthy" or "falsy" values, therefore the front-end will be able to process the result and act on it accordingly. E.g If a user does not have a profile cover, the API should return false, instead of an error or empty string. (Appendix 4.1.2)

As stated in the entity relationship diagram, items are represented as a many to one relationship with a user. Therefore it needs a nested representation in this resource, inside items also reveals the nested images resource. Having the application return JSON like this from an API allows the front-end to query the REST API and build the interface without the server returning HTML. This means once you are logged in there is no need to fetch resources such as CSS and Javascript files because the page will never be reloaded. The constant interchange between the front-end and the server via the REST api allows for this to happen. The effects of this should be a noticeably smoother, native like experience and a much more responsive application.

Interacting with the database can be done in numerous ways within a Django application, but the most common and preferred way is through its ORM, but if need be, raw SQL queries can always be performed instead of using the ORM.

4.2 Notable challenges

Despite personal experience surrounding the system architecture, the most significant challenge within this project is what holds the application together. The RESTful API, it is the bridge between the front and back-end and is the glue that holds the application together.

The API itself is a large area to cover as a notable challenge in the application, although it was. This section will focus on specific aspects of the rest API that proved challenging because of several reasons such as, limited knowledge in the area and limited documentation on performing complex tasks. It will also discuss aspects of the front-end that proved challenging, with a focus on client side routing and its importance for end user experience.

4.2.1 Creating a new item via the rest API with M:N relationships

There is often a simple association made between a resource in REST and its relation to a single database table, however in complex systems resources may contain other resources as well as single values. When creating a new item, users have the ability to add multiple tags to the item so they can be found in search results, in addition users have the ability to upload multiple images of an item. This already needs to perform several SQL statements, in which they all need linked back to the item, as they are tags and images for a particular item.

The complication arose when creating a new item become more complex than incepted. When creating the item, a custom create method needed to be introduced to over-ride its normal behaviour to upload each item image and tag to the system before it was used on the item.

Of course, this problem persisted into PUT and DELETE requests to the server. When updating the an item, each of current tags on an item will need to be deleted and re-added, when deleting an item, each item image and tag will need to be removed as well as the item resource.

Looking back at this, I can understand why this needs to happen and was naive to think that the framework would take care of this issue automatically. The process seems slightly tedious but in reality its a simple solution that needs implemented in this type of system.

The image below shows how this problem was solved by over-riding the create method on the item serialiser. The tags and images are extracted from the data in the XHR request, they are then looped over to create new resources of tags and images. Subsequently they are then saved and added to the current item instance being processed.

```
47
48      def create(self, validated_data):
49
50          current_tags = Tag.objects.all()
51
52          imgs = self.context['request'].data.getlist('images')
53
54          print validated_data
55
56          tagList = []
57
58          for x in current_tags:
59              tagList.append(x.name)
60
61          tags = validated_data.pop('tags')
62          images = validated_data.pop('images')
63
64          instance = Item.objects.create(**validated_data)
65
66          for key, tag in enumerate(tags):
67              if not tag in tagList:
68                  instance.tags.add(Tag.objects.create(name=tag))
69              else:
70                  instance.tags.add( Tag.objects.filter(name=tag).first() )
71
72          for img in imgs:
73              i = ItemImage.objects.create(name=img)
74              i.save()
75              instance.images.add(i)
76
77          return instance
78
```

Figure: 10: Custom create method in item serialiser

In order to create a new item, the process beings with the front-end of the application which takes in user input and prepares it to be sent in an XHR request as a FormData object with an enctype of multipart-formdata. This has been done to handle file uploads as sending a JSON string means the API has no means of decoding the image.

As the component is stateful, there is no need to extract the values from the DOM when the form is submitted, as the component will already know each form field value. A new instance of the formData object is created and assigned to ‘fd’. Each field is programatically appended into the FromData object. (Appendix 4.2.1)

You can see the fields that have many to many relationships are added a little differently to standard fields, such as tags and images. These are looped over and added into the FormData object one by one. Once this is done, the component calls an action to be performed passing

in the FormData. This is turn calls a method on the store, which calls a method on a separate class for handling XHR's, then transmits the received data back to the view. Therefore the view is always receiving a one way flow of data and are guaranteed to always be up-to-date. This unidirectional flow of data is shown below.

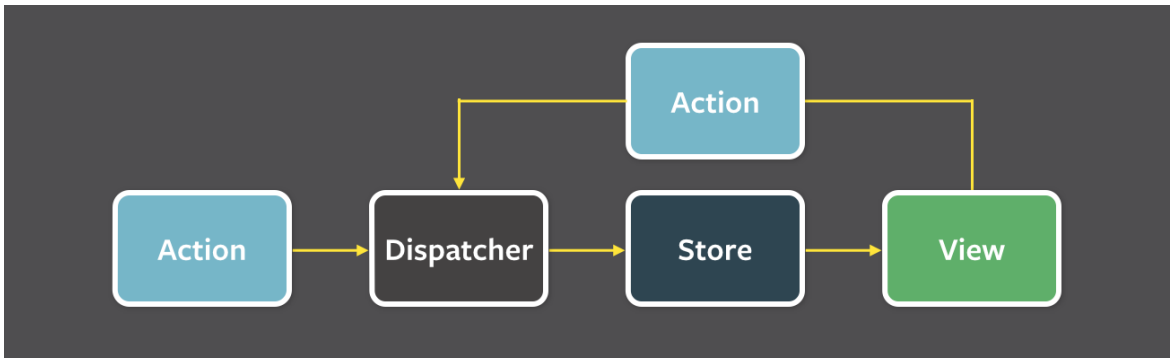


Figure: 11: Facebook flux architecture flow

This is a good example of how the architecture of this application works. If views any form of data, methods are called on actions via the view, which calls a centralised dispatcher passing through an action type, to the store, where it emits data for all listening views/components. This architecture pattern seems to be taking off in the front-end development community and there is still a lot of discussion around its implementation, it is also just a pattern and not a framework or library of any kind.

4.2.2 Client side routing, resource fetching and dynamic rendering of 404 views

Client side routing was extremely important in this singe page app as it allows a user to save application state. The application flow/ process of this is as follows.

- User navigates to route (eg. http://clinchapp.com/#/profile/items)
  - Router performs a lookup to see if route matches any routes defined
  - If a route is matched, the route triggers an associated view component to render (eg. ItemListView)
  - If listView is defined within a parent route (eg. http://clinchapp.com/#/profile) then the profile component is also matched and will be rendered.

If a route is triggered that should only render 1 resource, e.g a single user Profile, the front-end needs to send a request to retrieve the user, if the rest API returns false or an empty array then no user has been found, therefore a dynamic 404 view should be rendered.

Routing was a notable challenge within this project due to the nature of the project being a web application. The great thing about the web is it has URL's, you can link to direct resources, this does not happen in native applications. You have to manually navigate to the state you want. The Spotify web application has a web client and a native desktop client, in the web client you could save a url in browser history then navigate back to it. E.g. http://spotify.com/playlist/playlist However native web applications do not do this because they do not have a URL.

The main problem with this in Clinch was that when user's navigate to another user's profile. The url has to contain an id. E.g. /people/4. As the id is needed to be parsed from the url to send an XHR request to the server so it could return the correct person from the REST API. However, this presented a problem where the URLS did not convey much meaning for a user. Granted, they may know what ID the other user was, but surely a user would rather type. /people/<username> to lookup another users profile. The latter provided a problem, the client doesn't magically know what ID that username maps to, to subsequently perform a request to the back-end. Two ways this problem could be mitigated where:

- Perform the request to the server by passing in the username instead of the id.
- Perform two requests to the server.

The downside to the first option was that it violates the RESTful architecture. Single resources should be received via their ID and not any other field to ensure uniqueness. However, the upside being that it only needed to perform one request for the resource.

The second option was a bit more complicated, but did not violate REST principals. Two requests needed to be made to the server, the first being a small lookup to find the ID of a user with username 'x'. Once this was returned, the client can immediately use that ID to perform another request to get the correct resource. This option was the chosen option, as it meant that other clients such as native mobile apps could still use the REST api, but don't have to perform the first request as they do not have URL's to parse a username from.

Below shows an example of these two requests through the network panel and there responses respectfully.

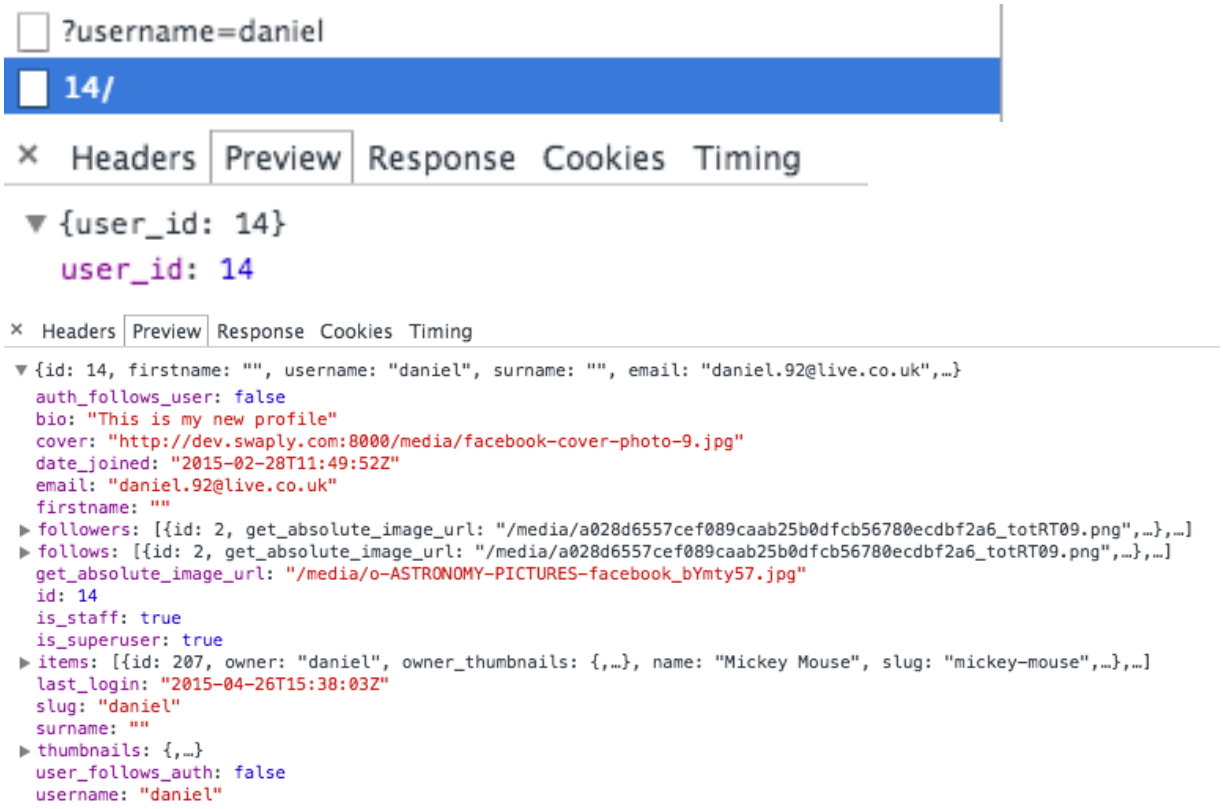


Figure: 11: API user lookup with subsequent RESTful call

The first request, as discussed, is the user lookup being performed. Directly after the request succeeded a RESTful call is made to retrieve the user form the ID.

4.3 Notable achievements

One of the main achievements in this application was generating the user interface based on 100% javascript. It's built on a component based architecture that is decoupled, therefore components can be, and have been re-used across the application. For example, the same item component is used to render items inside a user's profile, inside their home feed and inside search results. Other examples of this include a tags component, where tags need to be replicated across various views in the application.

Developing this application has resulted in a lot more work that it would have been if it was developed as a regular server-rendered application, the entire codebase has been carefully and logically considered throughout.

The search component is made up of around 4 or 5 components, including the searchBar, searchList, searchSideBar and LoadMore components. They work in tandem to provide a page view that is capable of searching every item uploaded by a user via their item tag. Items are then loaded into the view as a paginated data set. As the stateless principal in REST recommends, the client maintains control over the application state and keeps track of the current paginated result set. When users scroll to the bottom of the page, a new request is sent to the server to retrieve the next set of results, if this is successful the results are lazy loaded onto the page in a similar fashion to twitter. However, if the initial request to the server returns only 1 data set, the client knows that it doesn't need to initiate a request to the server when the user reaches the bottom of the page as the client already has all the data.

Multiple components work alongside this view to let the user know data-fetching is happening, this includes a Loader component. While the application is loading, it is also impossible to try and initiate a new request until it has finished, therefore preventing a user from misusing the application by spamming the load more button or constantly scrolling to the bottom of the page. These potential mis-uses of the view have been carefully considered and these careful preventative measures have been put in place to ensure the search view gives an enjoyable and successful experience to a user.

4.3.1 Social Activity Feed based on followers

Creating an application feed was not part of the original defined requirements, although the addition to this does reinforce the social aspect of the application. A major accomplishment in the application was the creation of this view. When users follow other users, any items that user adds will appear in the users activity feed, ordered latest item added. This is possible as when items are added they are given a timestamp, this can be seen on the item model. A query is made to retrieve all the items of the users that you follow, ordered by the date they uploaded the item.

The popular library moment.js is then used to parse the date format into a relative time that is more meaningful to the user. This means the user feed items will appear as 'Added 10 minutes ago' etc. Again, this provides a really useful social tool for users, as they can see the latest items of the people they follow before anyone else who does not follow them, and thus we have a deep need for the followers/following functionality. In addition to the functionality of the feed, the design of the feed is matched as closely as possible to the original feed design as part of the UX/UI design process.

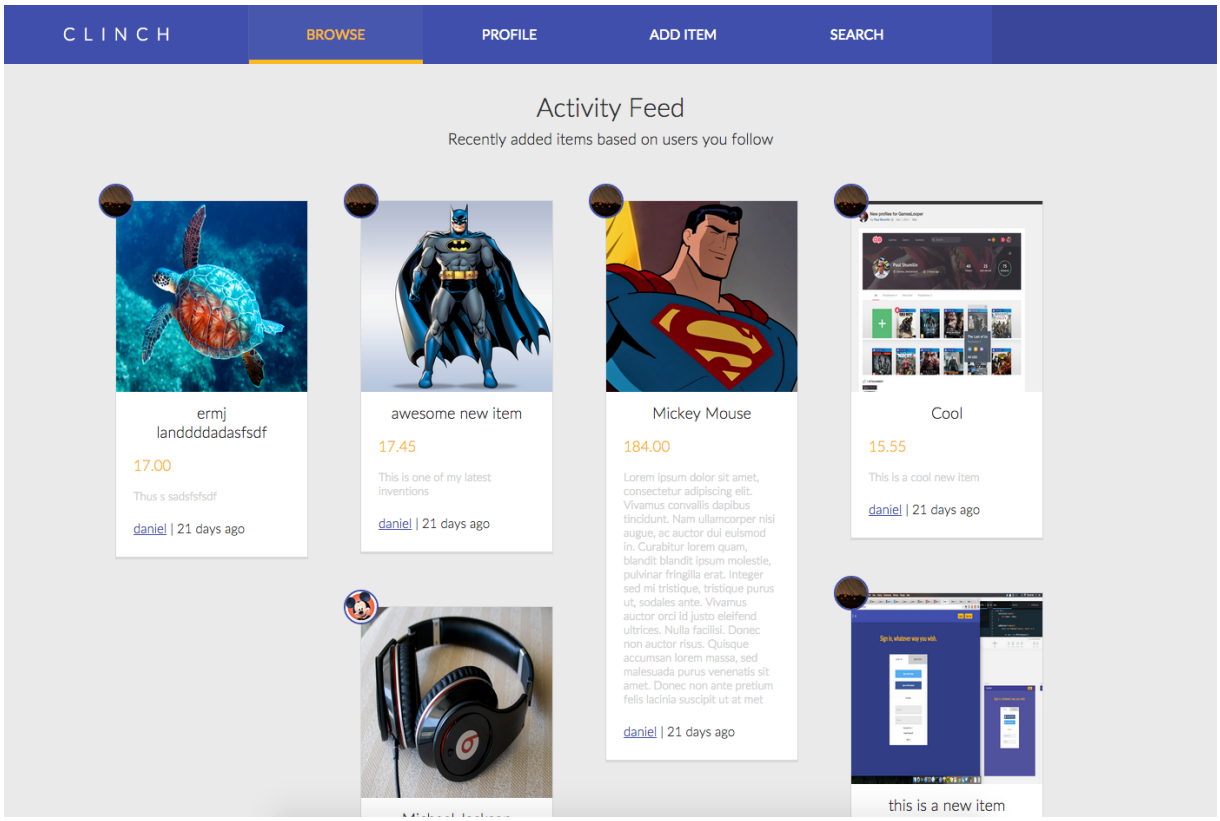


Figure:12 : Application activity feed with dummy data

This feed does contain a few technical additions with the introduction of a user avatar in combination with the item. The feed will also live update when users add items, with the list being automatically re-sorted. Using react means that the previous and current DOM trees are computed to check for differences, thus re-sorted the list will not force the DOM to update the list if it doesn't need to.

4.3.2 Back-end REST service

Other than the front-end, a lot of logic needed to implement the API has been considered a major achievement due to the time and effort required to make it functional and usable. The positive aspects of creating this API meant that there was no need to rely on using an external API to power the application, and therefore by bypassing issues such as rate limiting and temporary downtime of an API, although these issues could be greatly mitigated by implementing a cache. Nevertheless, since the API has been created solely for the purpose of this application, there is no need to implement these features (cache) unless the applications user base gains significant traction.

The achievement was made in building the structure of the API in a scalable fashion with the separation of resources and views. Below is an example of the directory structure that relate to several of the application endpoints. URL's are mapped via a regular expression and call methods on views when they are matched, views in turn use the serialisers to serialiser the data, therefore request information via the resource model. The the response is returned to the client.

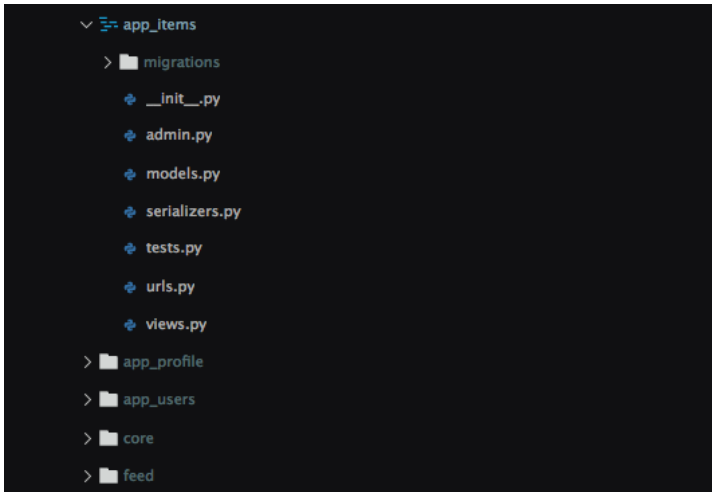


Figure:13 : Back-end API directory structure

4.3.3 ECMAScript 6

A significant risk in the project was developing the application in ES6, the next iteration of Javascript. The reason for doing so was not to benefit the project but to expand understanding of the language, this project was a perfect platform for doing so. In any case, the project has benefited from using ES6 as it has made performing the process of OOP (Object orientated programming) much easier by providing a better API for performing inheritance.

Classes have also been introduced to the language, although they are not native classes, merely syntactic sugar around javascript's prototypal inheritance model. An example where classes have been used in this application can be seen in the API class. The purpose of this class was to encapsulate all the logic for making XHR's to the back-end API. In ES6 native promise objects are introduced to the language, since this is the case there is no need to incorporate jQuery AJAX method. It was decided that using the native XMLHttpRequest would be much more sufficient. Each method in the API then returns a new promise object.

```
12  import {toQueryString} from "./utils";
13
14  class API {
15      constructor(token){
16          this.token = token;
17      }
18
19      addNewItem(formData){
20          return new Promise((resolve, reject) => {
21
22              var req = new XMLHttpRequest();
23              req.open('POST', '/api/items/', true);
24              req.setRequestHeader("X-CSRFToken", this.token);
25
26              req.onload = () => {
27                  if(req.status == 200){
28                      resolve( JSON.parse(req.response) );
29                  } else {
30                      reject( new Error(req.statusText) );
31                  }
32              }
33              req.send(formData);
34          });
35      }
```

Figure:14 : Front-end API Class for connecting with REST API

This is an an extract from a screenshot of the API class detailing the request sent to the items endpoint / resource. We can check if the request was successful by looking at the status code returned, then its possible to resolve or reject the data. JSON parse is used to parse the JSON into a javascript object, this is something thats normally done in the internals of jQuery AJAX call.

In relation to the Flux architecture pattern, stores are responsible for maintaining a reference to the applications data. Therefore they don't necessarily have to hold the logic for fetching the data itself.

The role of fetching data was solely the responsibility of the API class. Stores were used to call methods on the API then trigger events on listening components. The reason this is noted as a challenge in project is due to the architecture pattern. Application stores were made to be as trivial as simple as possible without logic duplication. Below is an example of how a stores calls methods on the API class and triggers the response to all listening components.

```
1  import ProfileActions from "../actions/profileActions";
2  import API from "../api";
3
4  export default Reflux.createStore({
5      listenables: [ProfileActions],
6      init(){},
7
8       async onGetProfile(user_id){
9         let response = await API.getProfileData(user_id);
10        this.trigger(response);
11      },
12
13      async onGetUserProfile(username){
14        let {user_id} = await API.lookup('users', username);
15        let resp = await API.getProfileData(user_id);
16        this.trigger(resp);
17      }
18 });
```

Figure:14 : Application store asynchronous requests

At first this seems quite standard. The ProfileActions is imported along with an instance of the API class, although something immediately seems odd with the syntax of the methods with an introduction to async functions and the 'await' keyword. The functions look like they

performing synchronous operations, but they are in fact not. The execution of any javascript code is delayed until the promise returns form the API is fulfilled. This is part of the ES7 specification but can be used with next generation compilers such as Babel.

5. Testing

This section will focus on a range of testing methods including an evaluation of the functional aspects of the project in relation to the outlined requirements specification. It will also discuss the usability testing carried out as a means of examining how users responded to the application. In an attempt to ensure cross-browser consistency, compatibility testing will be performed on various browsers and mobile devices to ensure that the application functions as expected.

5.1 Software testing methods

Software testing methods are normally assorted into two mains types of testing, black box testing and white-box testing. This is commonly known as the box approach.

5.1.1 Black box testing

Black-box testing refers to the analysis of software functionality as opposed to the underlying internal structures that make the system work. Its commonly referred to as functional testing (softwaretestingfundamentals.com, 2010). The main purpose of black-box testing is to check whether the intended software is meeting targets set by the requirements document/spec. In this test case, black box testers are aware of how the system should operate and what the expected results should be, but remain unconcerned about how the application performs tasks. Black box testing also encompasses non-functional testing, where performance, usability and maintainability are analysed.

5.1.2 White box testing

The testing approach known as white-box testing is the opposite of black-box testing where its main purpose is to test how the system performs tasks. These tests are known as unit tests, they are ran against small, encapsulated pieces of code to ensure its return value or outcome is what is expected by the developer. This testing approach ensures that early bugs with the application are found before any integration with new code happens.

The benefits of this testing approach within an application is that it ensures a consistent and high quality output of software that is mostly free of bugs. Other areas of white box testing

include integration and regression testing. Integration testing is commonly carried out after unit tests and is responsible for testing grouped pieces of software that have passed unit tests, in essence it looks at how these pieces of software interact with one another. Regression testing is carried out when new aspects of the system has been developed or patched as a means of uncovering new faults. (softwaretestingfundamentals.com, 2010)

5.1.3 Black-box testing, based on requirements specification

The requirements specification outlines the user stories that focus on key aspects of the applications functionality. An account of whether the application has passed each stage of the specification is displayed below.

Blackbox functional testing against requirement specification

Requirement	Expected Outcome	Actual Outcome	Status
#1 A user can register for an account by providing their email and password	The user should be able to register for an account using their email and password and subsequently receive an activation email.	When a user registers for an account with their password and email address, they are sent an activation email and subquently able to log in.	**PASSING**
#2 A user can reset their password if they have forgotten it	The user should be able to visit a password-reset page where they can enter their email to receive a password reset link.	When a user enters their email into the password reset form, the system email the user with a link to reset their password.	**PASSING**
#3 A user must verify their password by entering it twice upon registering	The user should be able promoted with a message if the passwords are different.	When both passwords match, the user is able to register. When both passwords do not match, a notification is given to the user to ensure they match	**PASSING**

Requirement	Expected Outcome	Actual Outcome	Status
#4 A user can login to the application by providing their username and password	The user should be able to login to the application with the username and password used upon registering	After registering, the user is able to login to the application after entering their username and password	**PASSING**
#5 A user can logout of the application	The user should be able to logout of the application and be redirected to the homepage	A user can logout of the application and is redirected to the homepage.	**PASSING**
#6 A user can upload an avatar to use as their profile image	The user should be be able to upload a new avatar to their profile or change an existing avatar.	when a user navigates to the settings page, they can change their profile information including their avatar.	**PASSING**
#7 A user can view another users profile when logged in	When a user is logged in, they should be able to directly access another users profile.	When the user logs in, items that appear in search results link to user profiles. Users can link to profiles by entering their username in the url.	**PASSING**
#8 A user can click on an uploaded item to see a detailed item view	When a user clicks on an item anywhere in the application, they should be directed to a detailed item view.	A user can link to a detailed item view to review all the images and tags relating to that item.	**PASSING**
#9 A user can opt out of having their items searchable via the search feature.	When a user uploads an item, they should be able to specify whether they wish it to appear within search results.	Users cannot at this present time decide whether they wish for their items to appear in search results.	**FAILING**
#10 A user can view the items of all their followers in a list	A user should be able to have a home or feed area where they can view the recent items uploaded by the users they follow.	When the user logs into the application, they can view a list of items uploaded by the people they follow in the feed view.	**PASSING**

After analysing some of the key functionality within the application. Most of the functional requirements are met. Requirement #9 was not met due to time limitations, but has been added as a future improvement of the application and was put into the project back-log.

5.4 User survey analysis

As part of testing this application is was vitally important to carry out a user survey to gauge the success of the application and its internal features. This enabled critical data to be gathered that could be used if future development of the application was to continue.

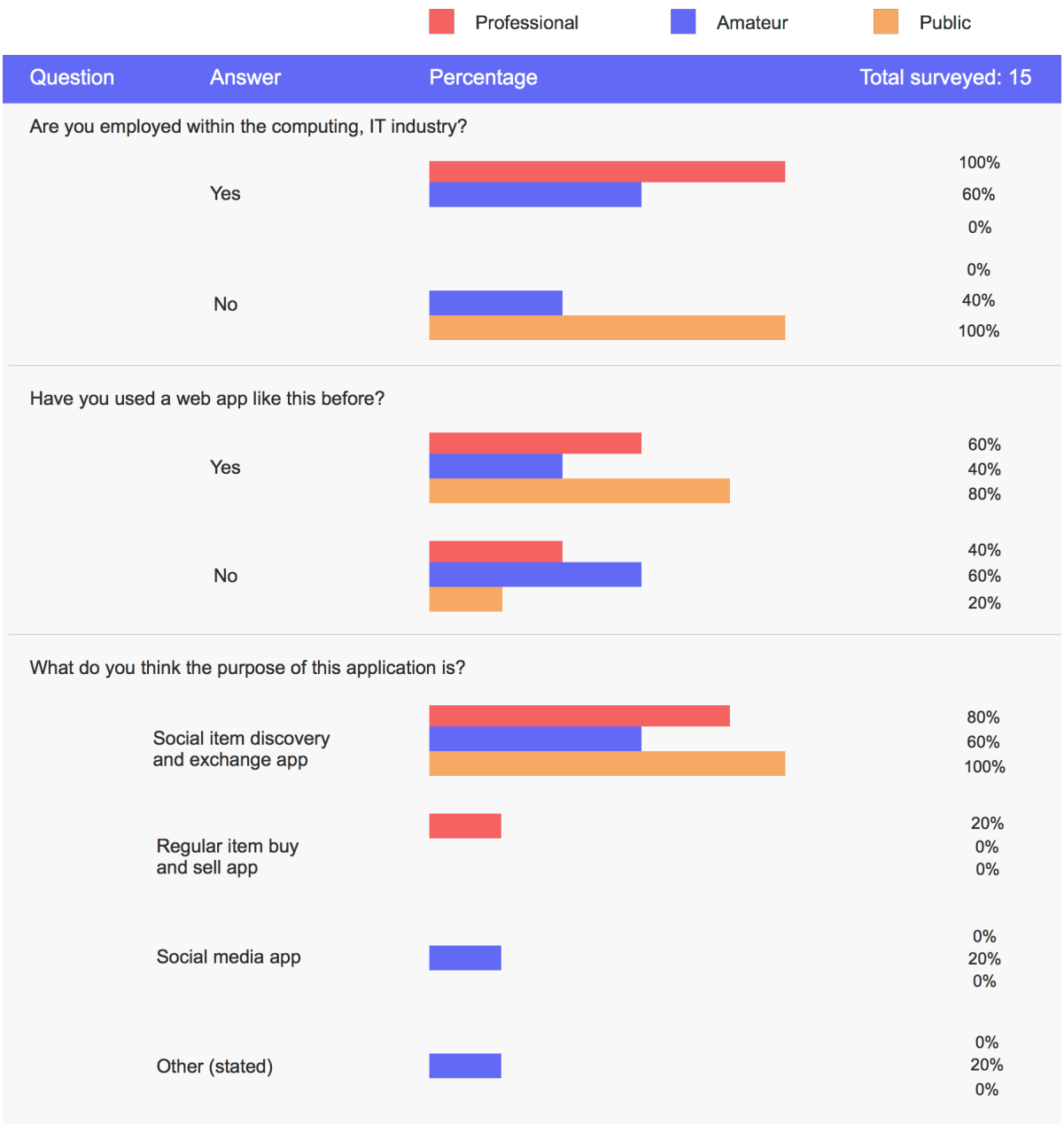


Figure:15 : User survey analysis

This graph represents a survey of 15 participants, 5 of whom are in the professional IT industry, 5 range from Amateur web development hobbyists and 5 members of the public non-relating to the industry. The survey showed a correlation between the professionals having a more critical view of the application than the general public. This was to be expected as their viewpoint may be biased due to their industry experience, though overall the figures where quite impressive.

Participants answered a range of question types from yes-no questions to more complicated questions based on usability. They were able to give their opinion on features that could be integrated into the application to provide a greater user experience, but conveyed the core features where there and have been well executed.

100% of all participants described the application to be in the range of usable - very usable. In hindsight this was great to discover a selected number of users had no issues in navigating through the application.

6. Deployment

The application was deployed to a VPS server on Digital Ocean. One of the main reasons for doing so was that commands needed to be ran as a root user to provision the server with everything the application needs. In this case

- MySQL database
- Python
- Django
- Nginx (Proxy Http server)
- Gunicorn (WSGI)
- Supervisor

Inside the development environment Vagrant was being used to replicate a production environment running on an instance of Ubuntu 14.04 with the above installed. This allowed the application to be tested locally in the same conditions as the production server without putting it on a real server. However, this application had not been bootstrapped to provision the server with a deployment script. Partly due to not being overly experienced in DevOps and lack of time, however some initial research was carried out into deploying the application using Vagrant and Docker.

Vagrant

Vagrant, combined with VirtualBox, enables you to run operating systems inside virtual machines. It provides a mechanism for mounting folders inside your machine to the virtual machine, this means you can work on an application and its changes are persisted into guest OS.

Deploying an application via Vagrant can be complicated, provisioning tools such as Chef and Puppet help to automate this process, although every time you deploy an application, you are in essence deploying a whole machine, which can take some time.

Docker

Docker allows you to deploy applications without the need for shipping a whole virtual machine. It acts somewhat like virtual machine except it has more portability as it only runs processes on top of the virtual machine, it is not a virtual machine itself.

7. Evaluation

7.1 User survey

Participants provided several features that could be useful in improving the application's functionality. Each of these features have been analysed with a response and outcome of each provided.

7.1.1 Feature addition 1, professional participants stated:

Adding a search features for users in the system would be beneficial if you wish to find your friend.

Response to feedback:

Users can find their friends if they know their username by entering if directly into the URL, although this feature would benefit as it gives the user the ability to search for multiple users with the same name.

Outcome:

This feature was added to future feature/enhancement list.

7.1.2 Feature addition 2, public participants stated:

The social media sign up is quick and easy. There could be room for social media platform integration when you add new items to your profile to further widen the audience reach.

Response to feedback:

This would be a great feature, enabling users to optionally send a tweet when they add a new item could attract more users to the application.

Outcome:

This feature was added to future feature/enhancement list.

7.1.3 Feature addition 3, Amateur participants stated:

This application is great. There should be a mobile web application for this. I think I would use this application even more then.

Response to feedback:

The application has been architected in a fashion that will make integration into mobile platform as painless as possible. You can expect to see Clinch as a mobile app soon.

Outcome:

This feature was added to future feature/enhancement list.

7.2 Project Outcomes

The intended goal of this application was originally to provide a service for social item discovery and exchange. This section looks at the overall success of the application for providing this service.

7.3 Methodology

The project has been following an Agile methodology during its development lifecycle to ensure a rapid progression of tasks thats inline with the original project plan. An agile approach has ensured that tasks are broken down into small parts, therefore increasing the viability of completing the task within the sprint. A project scrum board has been used to view each task within a given sprint.

The image below shows a scrum-board which lays out the development tasks for a given week. Issues that were not viable or too large where either broken down into smaller tasks or added into the back-log. Using this development approach was vital to the success of the project as it ensured a steady pace of development by focusing on critical tasks.

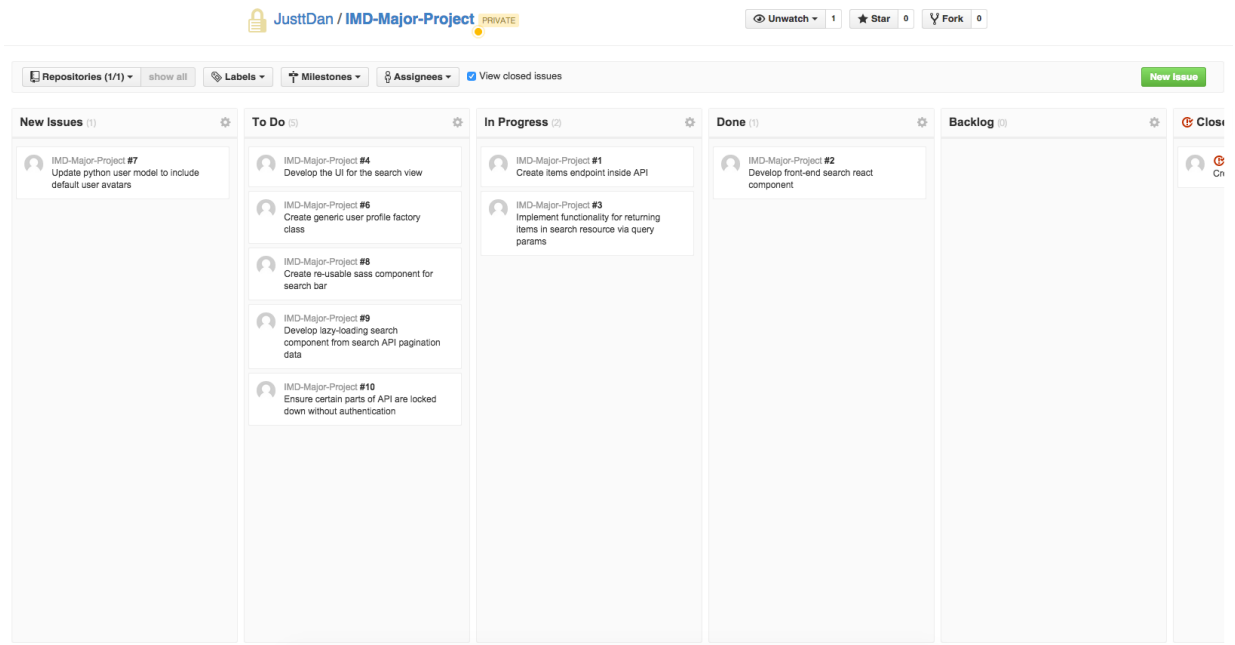


Figure:16 : Development Scrum-board - Github

Working with the agile methodology was at times challenging as it involved a steady development workflow by establishing rigorous deadlines on a weekly basis, however, the workflow has helped the project progress with great success by keeping track of project issues.

7.4 Project Plan

The project plan was drawn out and established at the beginning stages of the project and helped provide strong timescale estimations for each task, based on research and feedback from professionals in the industry. In addition, the plan provided project milestones for each task, this was important as it helped identify if one of the tasks was taking too long.

In reflection, the development timescales provided for certain tasks where a little too long, aspects such as user profile registration and server provisioning where performed way ahead of time. Although this was a slight oversight, getting some of the development tasks done earlier enabled for more critical aspects of the development to begin which improved the success of the project.

The project plan, (Appendix 1.1) has took all aspects relating to the development of the project into consideration including research, design and testing. Focusing on the design phase, the project plan outlines that most of the design would have been finished around mid December. However due to the nature of this projects methodology, critical changes in design persisted into January and February as minor changes in development has been changed.

Therefore, following the project plan throughout the course of project has been challenging, but it has been a vital resource to refer to during this project.

8. Conclusion

This project has seen several design and development iterations and been surveyed and used as a beta product by participants inside and outside the industry. The final aspect of this report sees its own summary and provides a high level review of the project and an evaluation of my own performance throughout. Lastly it discusses several future enhancements that have been defined by myself as the creator of this application and by survey feedback.

8.1 Report Summary

Each section of this report has clearly outlined the processes, workflow and technologies used to make application function. It defines areas that proved challenging and outlines how problems where overcome during the course of the project. The design phase focuses on internal system, logic and user experience design and discusses all aspects of their implementation within this application. It is in essence the pinnacle of my own accomplishments and reveals how the application progressed from its initial inception.

8.2 Project reflection

The project features a functional web application that allows users to upload items, follow other users and much more. It is a fun and lively application that serves its purpose well in the promotion of item discovery and exchange. In reflection the project meets the majority of core requirements and has been built in a scalable manner allowing for future work on the project if needed. As stated in section 1.1, the main goal of this application was.

“To produce a rich web application, acting as a service that allows users to discover and exchange items they own with one another.”

The project has successfully fulfilled this requirement while becoming much more than a simple service, but a fun and usable application that integrates aspects of social engineering to promote items. With a little extra work its not hard to believe that a web application like this could have commercial success.

8.3 Self review

Throughout the course of this project I have continually challenged myself in all areas, however I strongly believe I am stronger at front-end development, Javascript in particular. The code shown throughout this project is 100% my own, original code that makes use of future trends in Javascript, however the underlaying architecture of the application what I am particularly proud of. I do not wish to write code that merely just works, instead I have spent time and effort into ensuring that the application has a strong foundation of which to grow and maintain. The back-end of the application has also been developed in a considerate manner to allow the application to be maintainable. The approach taken with the back-end API has been difficult and my limits where stretched to accommodate for this, however I will always stretch my abilities in whatever I do, I certainly have surprised myself in what I have accomplished within this project.

8.4 Future improvements/enhancements

Section 7.1 discusses future improvements to the application that have been outlined by participants of the user survey. This application has a lot of fun areas to explore therefore future improvements are always going to identified. The most critical future improvement of the application is to develop in-app messaging. The current functionality in the application when users wish to contact one another is to send an email to the user through the interface of the application, this uses the Mandrill API for email delivery.

An aspect of the application that could be drastically improved is the follower suggestion section that appears when users sign unto the application that have no followers, the current implementation extracts the last 5 users to sign up to the application and suggests them, however it would be more beneficial if an algorithm was designed to return the 5 closest users to a given users location, therefore these users would have more relevance to a new user. The current implementation of this in hindsight is acceptable as the new user just needs to follow some users to get started. They can always unfollow.

References

Karl Wiegers . 2013. *Defining Project Scope: Managing Scope Creep*. [ONLINE] Available at: https://www.jamasoftware.com/blog/defining-project-scope-managing-scope-creep/. [Accessed 16 March 15].

Jurgen Appelo. 2008. *The Definitive List of Software Development Methodologies*. [ONLINE] Available at: http://noop.nl/2008/07/the-definitive-list-of-software-development-methodologies.html. [Accessed 18 March 15].

Agile Methodology. 2008. *Agile Metholody, What Is Agile?*. [ONLINE] Available at: http://agilemethodology.org/. [Accessed 20 March 15].

Agile Methodology. 2013. *Waterfall vs. Agile: Which is the Right Development Methodology for Your Project?*. [ONLINE] Available at: http://www.seguetech.com/blog/2013/07/05/waterfall-vs-agile-right-development-methodology. [Accessed 28 March 15].

ISTQB EXAM CERTIFICATION. 2013. Waterfall vs. Agile: Which is the Right Development Methodology for Your Project?. [ONLINE] Available at: http://istqbexamcertification.com/what-is-prototype-model-advantages-disadvantages-and-when-to-use-it/. [Accessed 28 March 15].

Rest API tutorial. 2013. *What Is REST?*. [ONLINE] Available at: http://www.restapitutorial.com/lessons/whatisrest.html. [Accessed 01 April 15].

Brian Mulloy. 2011. *Teach a dog to REST*. [ONLINE] Available at: https://vimeo.com/ 1 17785736. [Accessed 18 April 15].

Jeff Atwood. 2008. *Understanding Model-View-Controller.* [ONLINE] Available at: http://blog.codinghorror.com/understanding-model-view-controller/. [Accessed 04 April 15].

Google Polymer. 2008. *Welcome to the future*. [ONLINE] Available at: https://www.polymer-project.org/0.5/. [Accessed 04 April 15].

Facebook React. 2015. *A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES*. [ONLINE] Available at: https://facebook.github.io/react/. [Accessed 07 April 15].

Facebook Flux. 2015. *APPLICATION ARCHITECTURE FOR BUILDING USER INTERFACES*. [ONLINE] Available at: https://facebook.github.io/flux/. [Accessed 08 April 15].

Addy Osmani. 2015. *The revealing module pattern*. [ONLINE] Available at: http://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript. [Accessed 08 April 15].

Babel JS. 2015. *A detailed overview of ECMAScript 6 features*. [ONLINE] Available at: http://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript. [Accessed 08 April 15].

Andrew Ray. 2014. *ReactJS For Stupid People*. [ONLINE] Available at: http://blog.andrewray.me/reactjs-for-stupid-people/. [Accessed 12 April 15].

Facebook. 2014. *Reconciliation*. [ONLINE] Available at: https://facebook.github.io/react/docs/reconciliation.html. [Accessed 12 April 15].

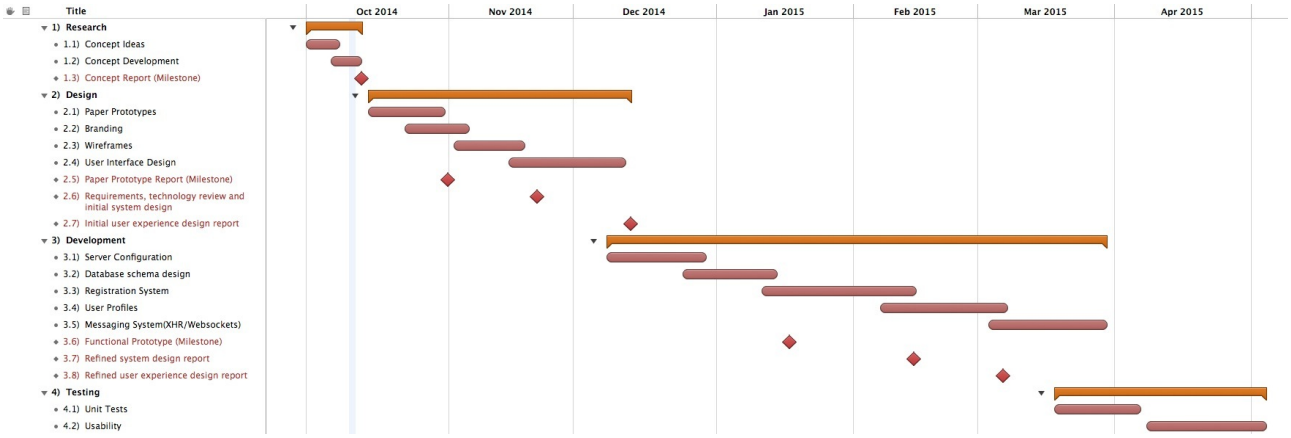
Pete Hunt. 2013. *React: Rethinking best practices*. [ONLINE] Available at: https://www.youtube.com/watch?v=x7cQ3mrcKaY. [Accessed 15 April 15].

Software Testing Fundamentals. 2013. *White Box Testing*. [ONLINE] Available at: http://softwaretestingfundamentals.com/white-box-testing/. [Accessed 15 April 15].

Software Testing Fundamentals. 2013. *Black Box Testing*. [ONLINE] Available at: http://softwaretestingfundamentals.com/black-box-testing/. [Accessed 15 April 15].

Appendix

1.1 Planning



2.2 Requirements specification

#01

Type
Functional

Description
A user can register for an account by providing their email and password.

Rationale
Users will need to be registered and logged in to upload items to their profile.

Fit Criterion
Users will be subsequently able to login to the application and have access to their profile page.

Dependencies
N/A

#02

Type
Functional

Description
Users can easily reset their passwords if they have forgotten it.

Rationale
Users frequently use different passwords they might forget when signing up to new services they are not familiar with.

Fit Criterion
A reset link will be sent to the user's email allowing them to reset their password by entering a new password twice for confirmation.

Dependencies
#01 - A user must have an account to reset their password.

#03

Type
Functional

Description
A user must verify their password by entering it twice upon registering.

Rationale
To ensure the intended password is the correct one, two passwords will need to be entered to check for equality.

Fit Criterion
A user will be able to successfully register an account without any form errors appearing.

Dependencies
#01

#04

Type
Functional

Description
A user can log into the application by providing their email and password.

Rationale
In order to use the features in the application a user will have to log in.

Fit Criterion
When a user logs in they will be redirected to their default profile page.

Dependencies
#01, A user needs to be registered in order to log in

#05

Type
Functional

Description
A user can log out of the application

Rationale
Users may wish to be logged out of the application when they are not using it.

Fit Criterion
When a user presses the log out button, they will be logged out and redirected back to the homepage.

Dependencies
#04, A user must be logged in to be able to log out.

#06

Type
Functional

Description
A user can upload an avatar to use as their profile image.

Rationale
A user needs to identify and personalise their account.

Fit Criterion
A user's profile photo will update with the image uploaded by the user, the profile image link will be saved to the database.

Dependencies
#04, A user must be logged in to change their profile photo.

#07

Type
Functional

Description
A user can upload a new item they wish to swap to their profile.

Rationale
In order for the user to engage in a swap they must have at least one swappable item.

Fit Criterion
A success message will appear confirming the addition of a new item. The item will then appear on the users profile page.

Dependencies
#04, A user must be logged in upload an item.

#08

Type
Functional

Description
A user can click on an uploaded item to see a detailed item view.

Rationale
A user may wish to see the full details of the item in their profile.

Fit Criterion
The user will be navigated to the individual item view where they can view the full item details, E.g Tags, item value etc.

Dependencies
#04, A user must be logged in to access this view

#09

Type
Functional

Description
A user can search for items via item tags.

Rationale
To help users discover items on the system, a user can search for tags to find items with that tag.

Fit Criterion
A list of items will appear that are tagged with the users search term.

Dependencies
#04, A user must be logged in in order the use the search feature.

#10

Type
Functional

Description
A user can invite the owner of an item to a swap.

Rationale
An invitation must be sent to the owner of an item so they can decline or accept the invitation.

Fit Criterion
A confirmation prompt will appear confirming the invitation has been sent to the items owner.

Dependencies
#04, A user must be logged in to sent invitations.

#11

Type
Functional

Description
A user can delete their account.

Rationale
A user must have the ability to delete their account if they no longer wish to use it.

Fit Criterion
The user will be logged out and redirected to the homepage. The user will no longer be able to login to the application using their email / password.

Dependencies
#01, The user must be registered in order to de-register.
#04, The user must be logged in to delete their account.

#12

Type
Functional

Description
A user can view the profile of another user.

Rationale
A user may wish to view items by a particular user.

Fit Criterion
A user can navigate to another users profile.

Dependencies
#04, The user must be logged in to view other users profiles.

#17

Type
Functional

Description
A user can swap an item with another user.

Rationale
A user should be able to swap an item with another user if both items are of the same value.

Fit Criterion
Confirmation email is sent to both parties confirming the swap, Items will be removed from both users profile to prevent other users from sending invites for that item.

Dependencies
#04, The user must be logged in to swap items with other users.

#18

Type
Functional

Description
A user can view each individual items gallery

Rationale
When a user uploads an item, they are encouraged to upload more than one, so other users can view them in a gallery.

Fit Criterion
The user can view the items gallery from the individual item view.

Dependencies
#04, The user must be logged in to view an individual items gallery.
#08, An individual item view must exist for each item.

#19

Type
Functional

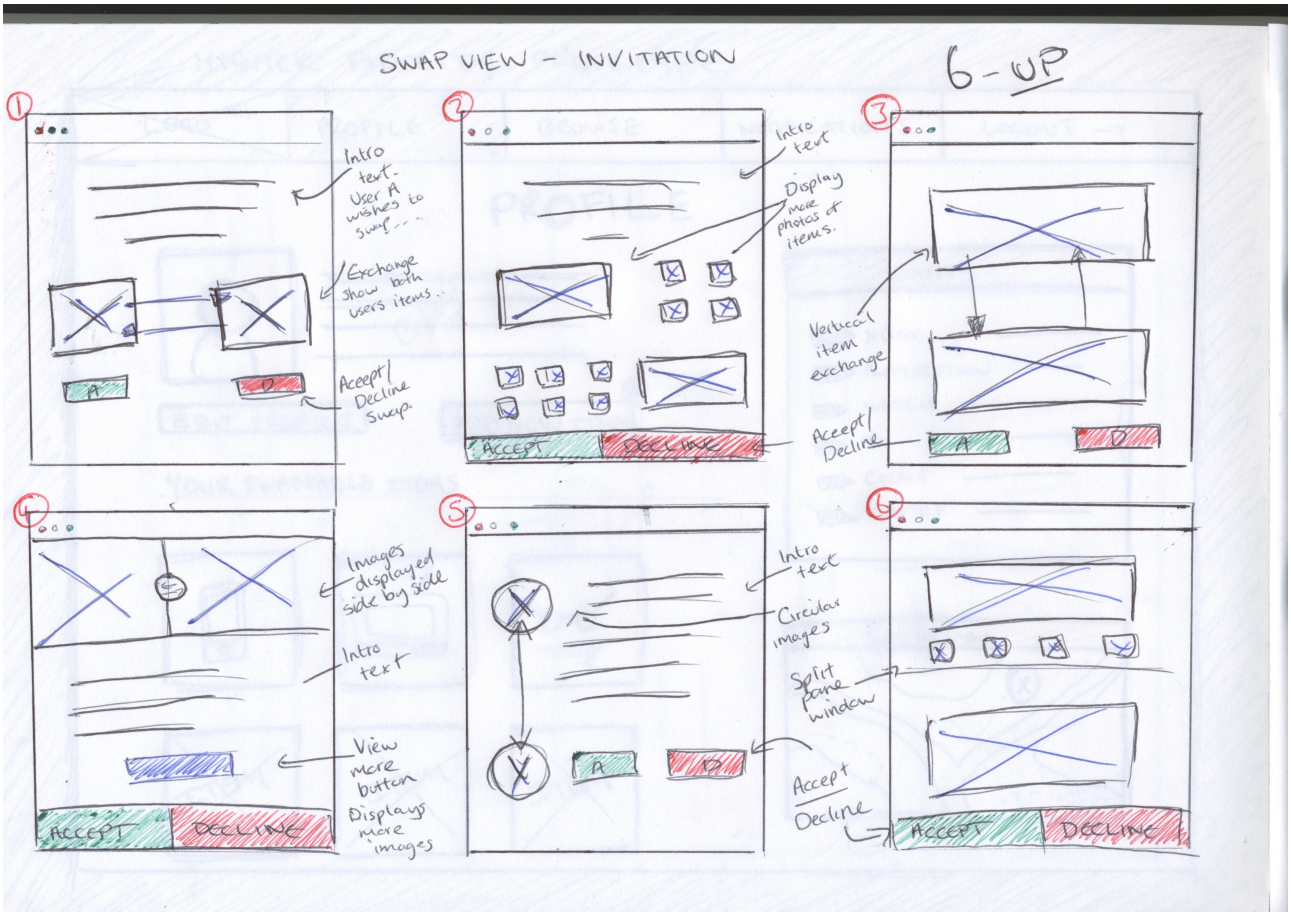
Description
A user can opt out of having their items searchable by the search feature

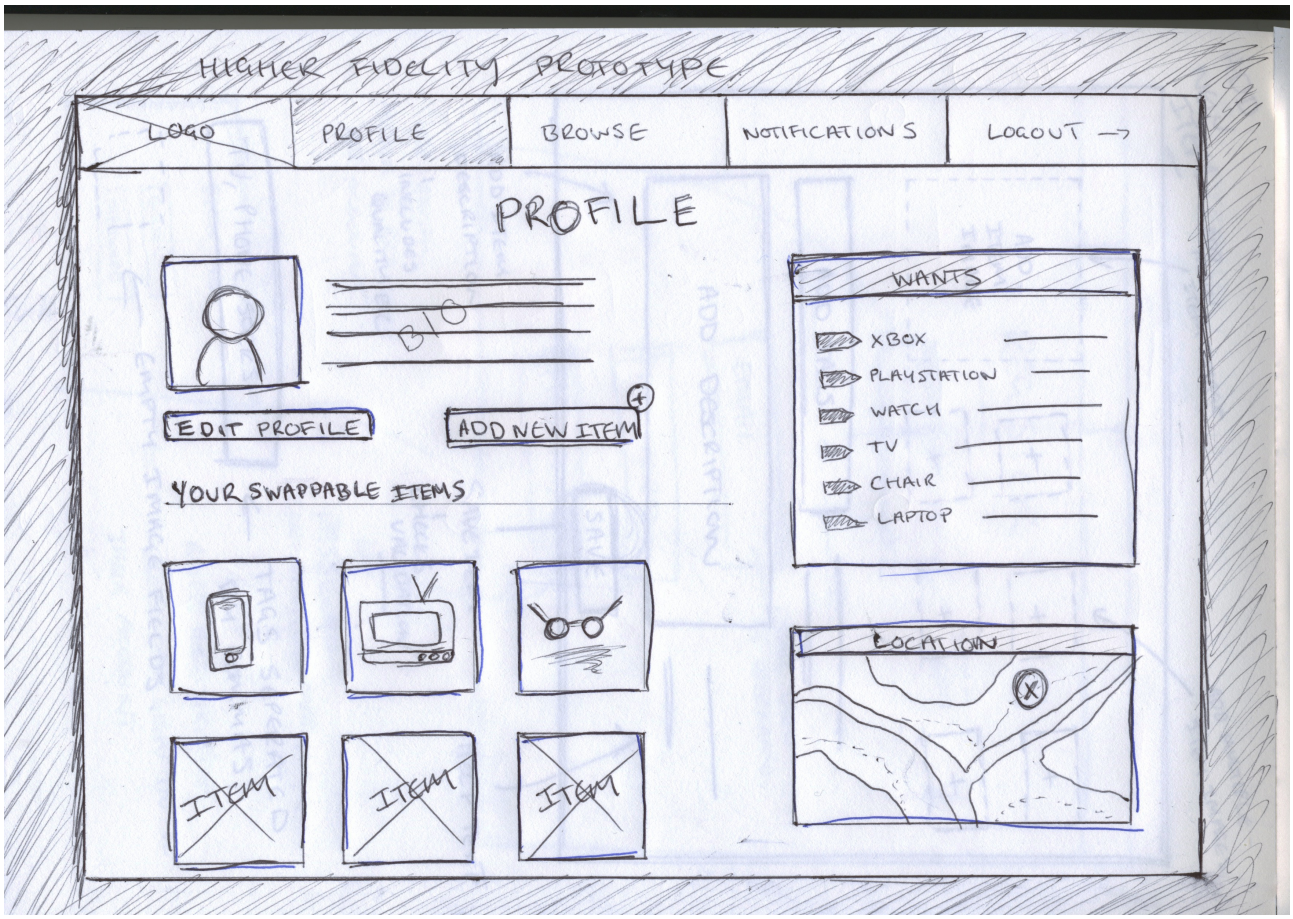
Rationale
A user may wish to opt out of the search and only share items with people they know.

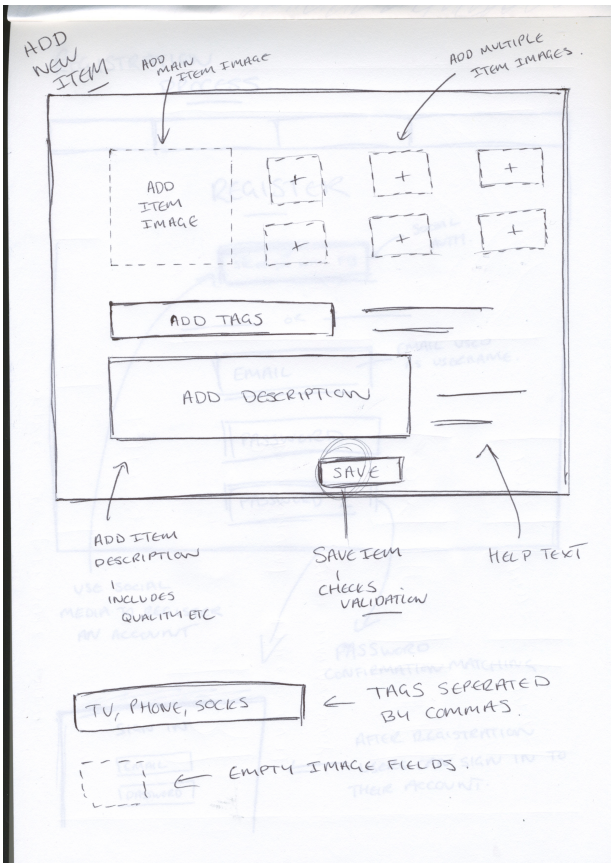
Fit Criterion
A users items will not appear in the search results.

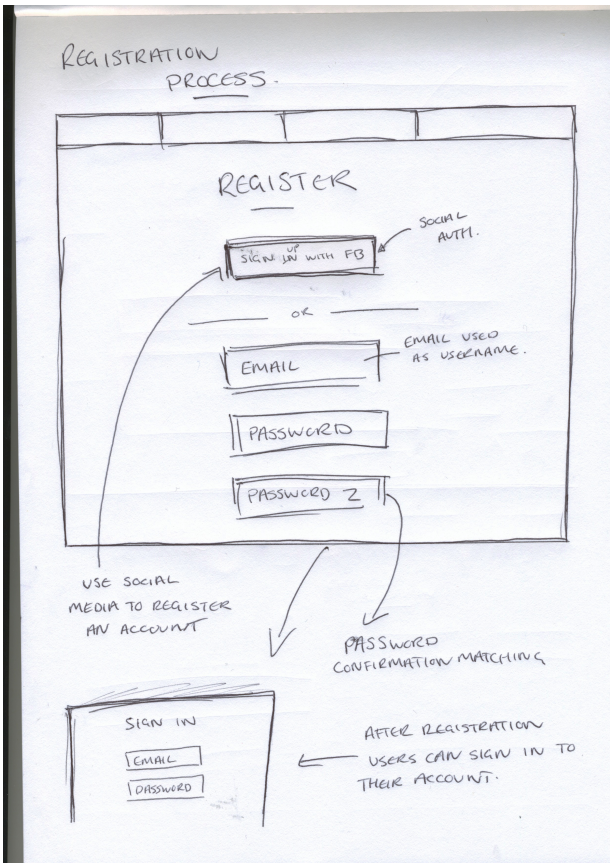
Dependencies
#04, The user must be logged in.

2.3 Paper Prototyping

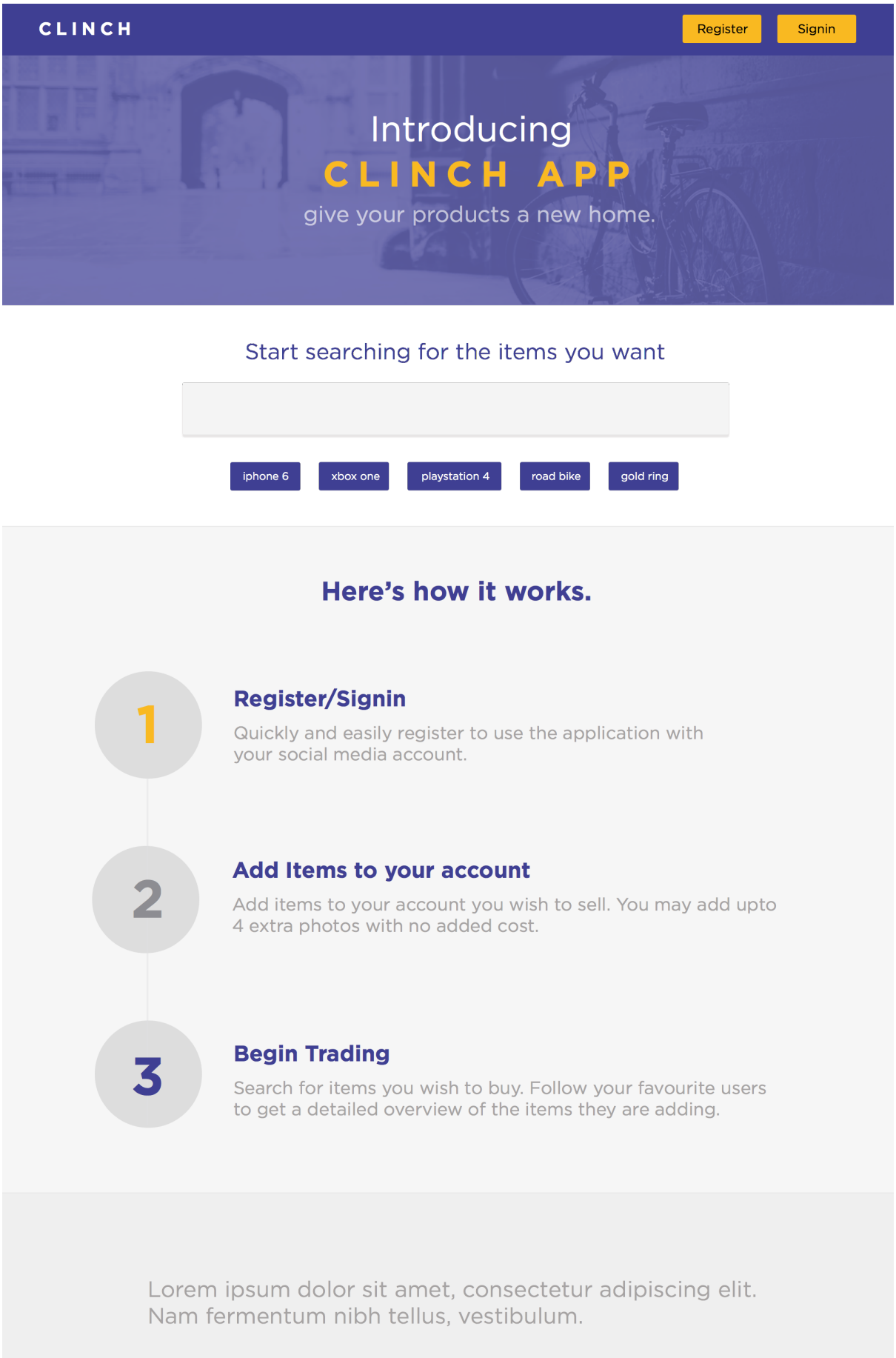



HIGHER FIDELITY PROTOTYPE
LOGO
PROFILE
BROWSE
NOTIFICATIONS
LOGOUT →
PROFILE
BIO
EDIT PROFILE
ADD NEW ITEM
YOUR SWAPPABLE ITEMS
ITEM
ITEM
ITEM
WANTS
XBOX
PLAYSTATION
WATCH
TV
CHAIR
LAPTOP
LOCATION


ADD NEW ITEM
ADD MAIN ITEM IMAGE
ADD MULTIPLE ITEM IMAGES.
ADD ITEM IMAGE
ADD TAGS
ADD DESCRIPTION
SAVE
ADD ITEM DESCRIPTION
INCLUDES QUALITY ETC
SAVE ITEM
CHECKS VALIDATION
HELP TEXT
TV, PHONE, SOCKS
TAGS SEPERATED BY COMMAS.
EMPTY IMAGE FIELDS.


REGISTRATION PROCESS.
REGISTER
SIGN UP SIGN IN WITH FB
SOCIAL AUTH.
OR
EMAIL
EMAIL USED AS USERNAME.
PASSWORD
PASSWORD 2
USE SOCIAL MEDIA TO REGISTER AN ACCOUNT
PASSWORD CONFIRMATION MATCHING
SIGN IN
EMAIL
PASSWORD
AFTER REGISTRATION USERS CAN SIGN IN TO THEIR ACCOUNT.

3. Design


CLINCH
Register
Signin
Introducing
CLINCH APP
give your products a new home.
Start searching for the items you want
iphone 6
xbox one
playstation 4
road bike
gold ring
Here's how it works.
1
Register/Signin
Quickly and easily register to use the application with your social media account.
2
Add Items to your account
Add items to your account you wish to sell. You may add upto 4 extra photos with no added cost.
3
Begin Trading
Search for items you wish to buy. Follow your favourite users to get a detailed overview of the items they are adding.
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nam fermentum nibh tellus, vestibulum.

CLINCH | Browse | Profile | Add Item | Daniel Skelton

Activity Feed

Based on users you follow

Nokia 635

£80.00

I've had this phone for 2 years and its never failed me. It features an 8mp camera and is in superb condition. Please contact me.

mike783 | 16 hours ago

Nike Blazer Hi Su...

£30.00

Worn once but still in very good condition.

rebecca0x | 19 hours ago

18ct White gold t...

£395.00

Engagement ring only worn for a few weeks. In fantastic condition.

sarahs47 | 1 day ago

iphone 5 3g

£200.00

Great mobile phone seeking new owner due to an upgrade. This phone has a small crack in the bottom left corner. Could be easily repaired.

markBolton | 1 day ago

Canon Eosv camera

£35.00

jamesmay1 | 1 day ago

CLINCH

Register Signin

Phone

104

PRICE

MIN MAX

LOCATION

BELFAST

iphone 5 3g
£200.00

Nokia 635
£80.00

Sony Xperia Z2 - 16..
£239.00

iphone 5 3g
£200.00

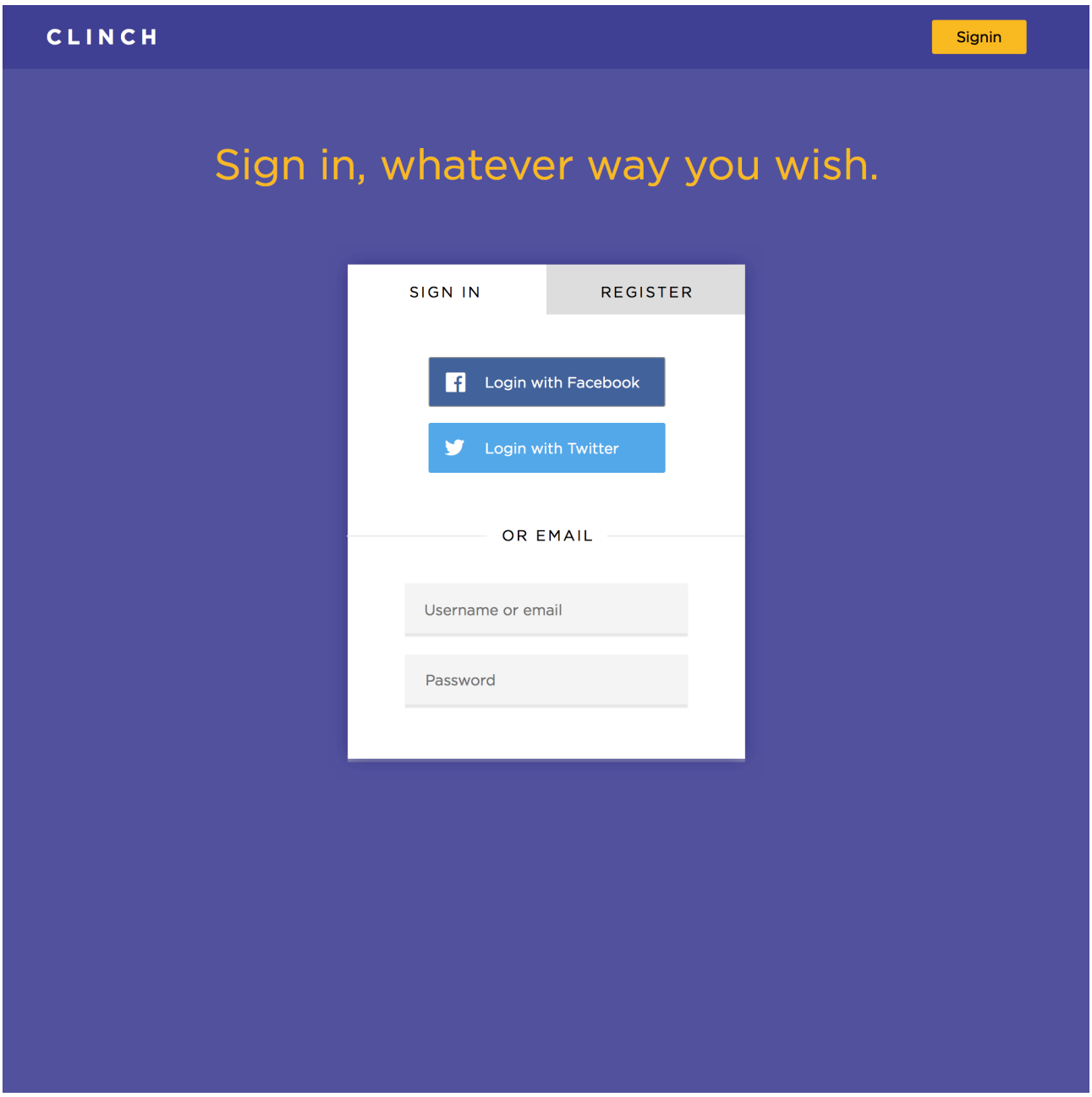
Nokia 635
£80.00

Sony Xperia Z2 - 16..
£239.00

iphone 5 3g
£200.00

Nokia 635
£80.00

Sony Xperia Z2 - 16..
£239.00


CLINCH
Signin
Sign in, whatever way you wish.
SIGN IN
REGISTER
Login with Facebook
Login with Twitter
OR EMAIL
Username or email
Password

CLINCH | Browse | Profile | Add Item | Daniel Skelton

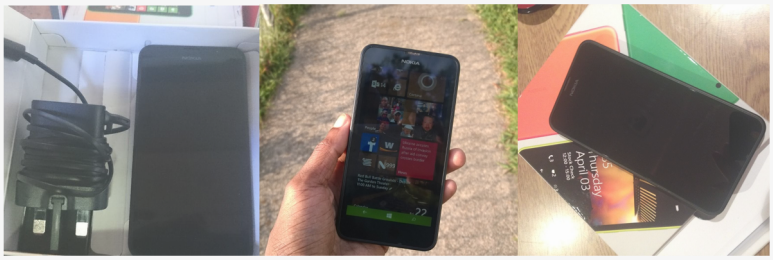
View item owner

Nokia 635

£80.00

I've had this phone for 2 years and its never failed me. It features an 8mp camera and is in superb condition. Please contact me.

phone nokia black



Contact **mike783**

Hi, I am interested in talking to you about purchasing this phone ...

419

Send

* The owner will respond to you via email. Please check your inbox for replies.

CLINCH | Browse | Profile | Add Item | Daniel Skelton

Add a new item

1 Add details about your item

Name

Value

Tags

Seperate tags with commas. E.g. iphone, phone

Description

2 Upload images of your item.

Upload image

We will use this photograph to display your item in search results.

Upload image

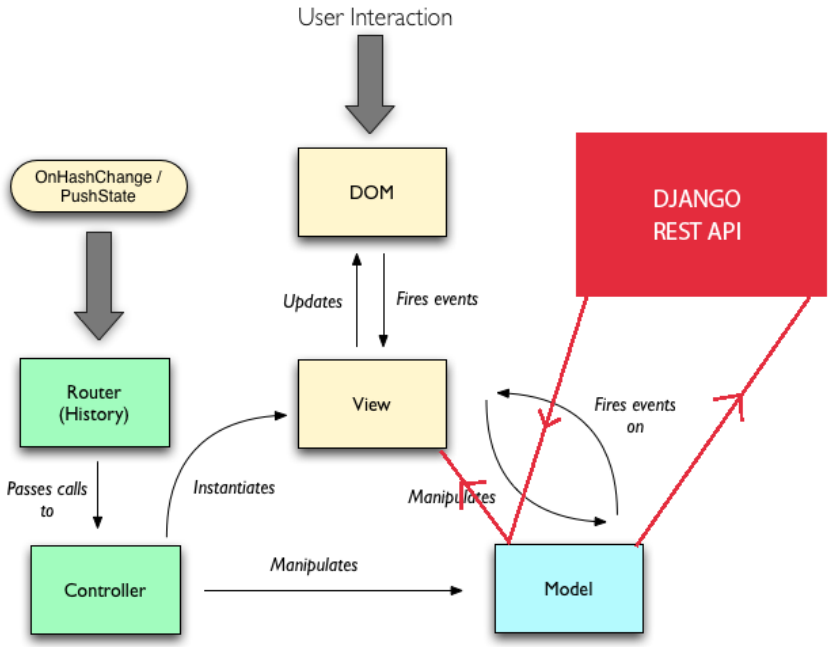
Upload image

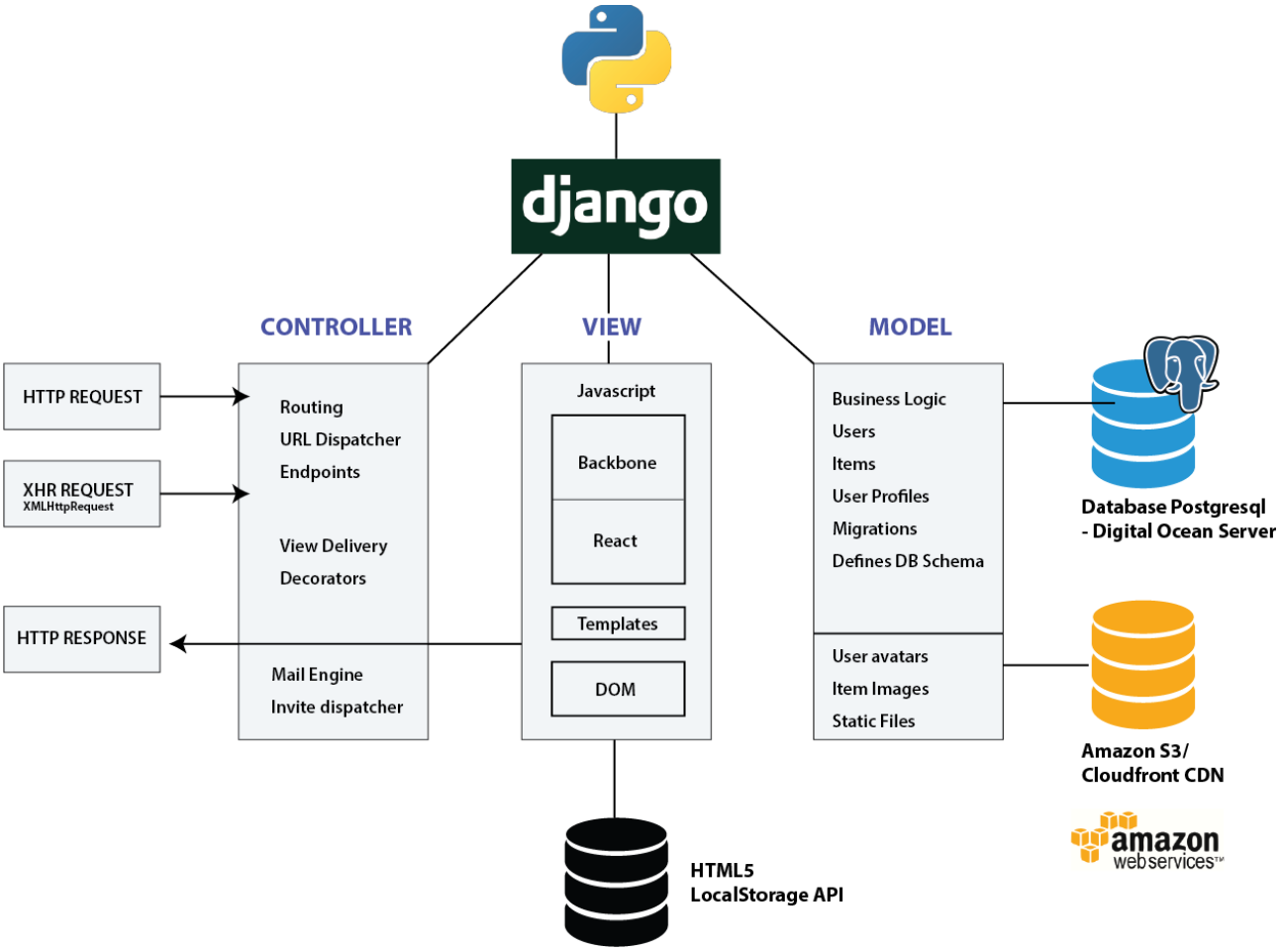
Upload image

Upload image

Upload Item

3.2 System Design





4. Implementation

4.1.2 Example Json Output

```
GET /api/users/
```


```
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, HEAD, OPTIONS

{
    "count": 6,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 2,
            "firstname": "",
            "username": "matthew",
            "surname": "",
            "email": "daniel.92@live.co.uk",
            "is_superuser": false,
            "is_staff": false,
            "last_login": "2015-04-02T21:09:31Z",
            "date_joined": "2015-02-28T12:28:05Z",
            "bio": "Hello. I am matthew. THIS SITE IS AWESOME",
            "cover": "http://dev.swaply.com:8000/media/o-ASTRONOMY-PICTURES-facebook.jpg",
            "slug": "matthew",
            "get_absolute_image_url": "/media/a028d6557cef089caab25b0dfcb56780ecdbf2a6_totRT09.png",
            "thumbnails": {
                "avatar_200": "/media/CACHE/images/a028d6557cef089caab25b0dfcb56780ecdbf2a6_totRT09/fffd5e56060aa049fb1b4c298691bf6e.jpg",
                "avatar": "/media/a028d6557cef089caab25b0dfcb56780ecdbf2a6_totRT09.png"
            },
            "items": [
                {
                    "id": 226,
                    "owner": "matthew",
                    "owner_thumbnails": {
                        "avatar_200": "/media/CACHE/images/a028d6557cef089caab25b0dfcb56780ecdbf2a6_totRT09/fffd5e56060aa049fb1b4c298691bf6e.jpg",
                        "avatar": "/media/a028d6557cef089caab25b0dfcb56780ecdbf2a6_totRT09.png"
                    },
                    "name": "Michael Jackson",
                    "slug": "michael-jackson",
                    "value": "175.00",
                    "created": "2015-03-24T14:46:24Z",
                    "images": [
                        {
                            "id": 162,
                            "name": "http://dev.swaply.com:8000/media/17ma5fcqwpbjjjpg_PY4X7xJ.jpg",
                            "sizes": {
                                "s": "/media/CACHE/images/17ma5fcqwpbjjjpg_PY4X7xJ/621d5f46b0d97e9c223fc2a9a38dcedf.jpg",
```


4.2.1 Client side item creation

```
46          let fd = new FormData();
47
48          fd.append('owner', 1);
49          fd.append('name', this.state.name);
50          fd.append('value', this.state.value);
51          fd.append('description', this.state.description);
52
53          tags.forEach(tag => fd.append('tags', tag));
54
55          Array.from(images.files).forEach(file => fd.append('images', file));
56
57          ItemActions.add(fd);
58
59      },
60
```

4.2.2 Client-side application routes

```
let routes = (

    <Route name="app" path="/" handler={App}>

      <Route name="profile" handler={ ProfileFactory.create(1) } path="profile">
          <Route name="items" path="/profile" handler={UserItemList} />
          <Route name="following" handler={ Relation('follows') }  />
          <Route name="followers" handler={ Relation('followers') }  />
          <DefaultRoute handler={UserItemList} />
      </Route>

      <Route name="item" path="/profile/items/:name" handler={ItemDetail} />

      <Route name="settings" path="/settings" handler={Settings} />


      <Route name="people" handler={ ProfileFactory.create(2) } path="/people/:username">
        <Route name="userItems" path="items" handler={UserItemList} />

        <Route name="userFollowing" path="following" handler={ Relation('follows') }  />
        <Route name="userFollowers" path="followers" handler={ Relation('followers') }  />

        <DefaultRoute handler={UserItemList}  />
      </Route>


      //implement an api lookup instead of query strings.
      <Route name="userItem" path="/people/:username/items/:name" handler={ItemDetail} />

      <Route name="feed" handler={Feed} />
      <Route name="new" handler={NewItem}  />
      <Route name="search" handler={Search} />

      <DefaultRoute handler={Feed} />
      <Redirect from="/" to="feed" />
```

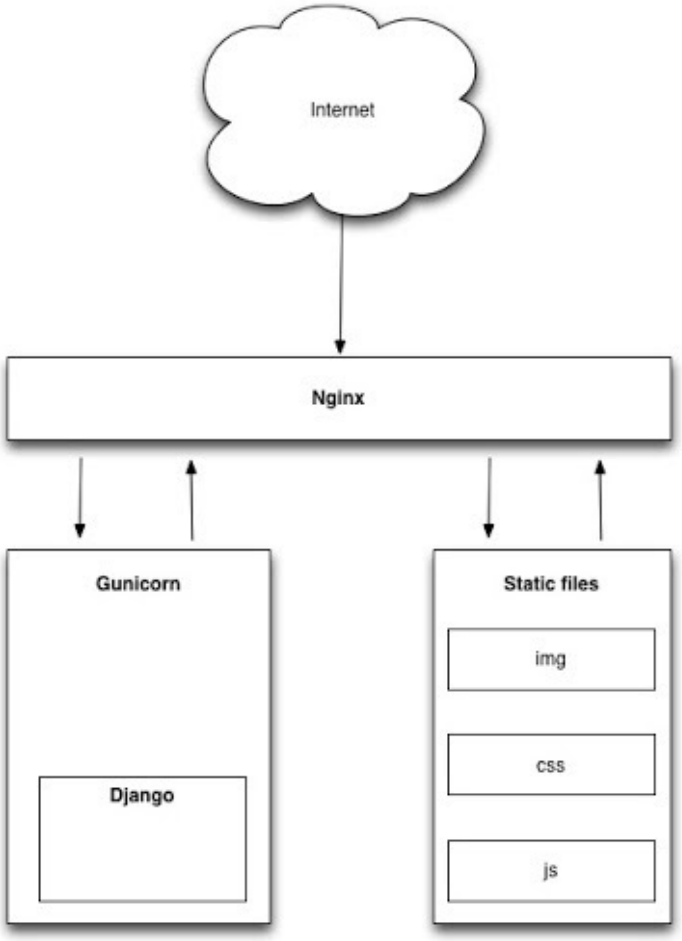
4.3 Search Component

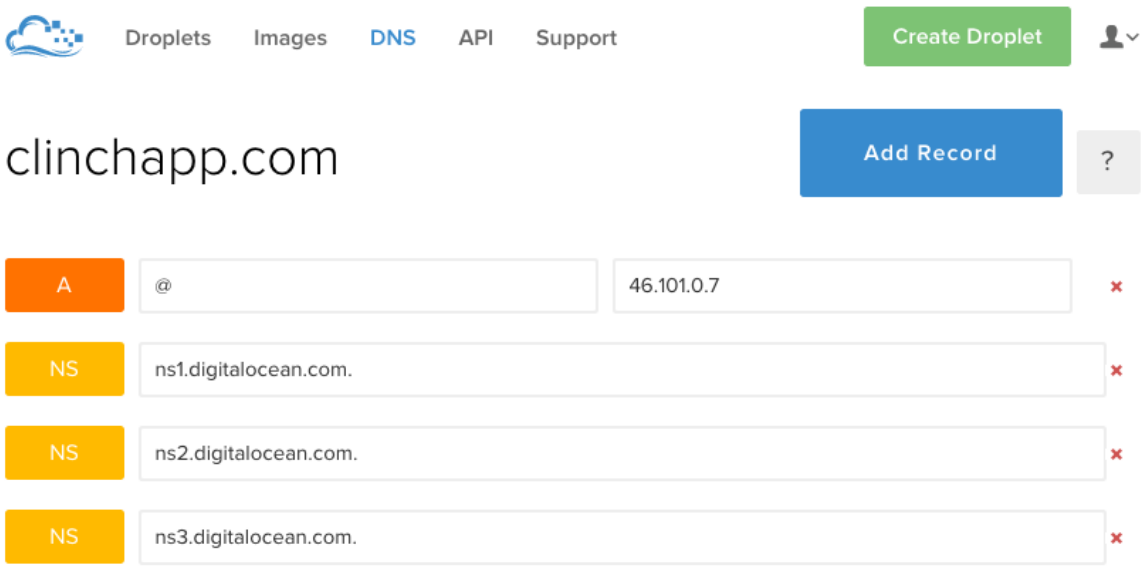
```
/*
  The wrapper for searchbar and searchList,
  recieves tags via props.
  search.jsx needs moved into an interface directory or page.
*/

import SearchActions from "../../actions/searchActions";
import SearchStore from "../../stores/searchStore";
import LoadMore from "../../stores/loadMoreStore";
import {toQueryString} from "../../utils/";
import SearchBar from "./SearchBar";
import SearchList from "./searchList";
import SearchSideBar from "./searchSideBar";

export default React.createClass({

  mixins: [
    Reflux.listenTo(SearchStore, 'onSearch'),
    Reflux.listenTo(LoadMore, 'onLoadMore')
  ],

  getInitialState() {
      return {
        response: [],
        data: [],
        pageState: 1,
        loading:false
      };
    },

  onLoadMore(response){

    //if a trigger uses a Loadmore true, with the response then that can be used
    //instead of two funcs

    if(response.loading){
      this.setState({loading:true});
      console.log('loading...');
      return;
    }

    console.log('loading done...', response);

    this.setState({
      data: this.state.data.concat(response.results),
      response: response,
      pageState: this.state.pageState + 1,
      loading:false
    });
  },
```

4.3.2 Noteable achievements: Backend REST API: User View

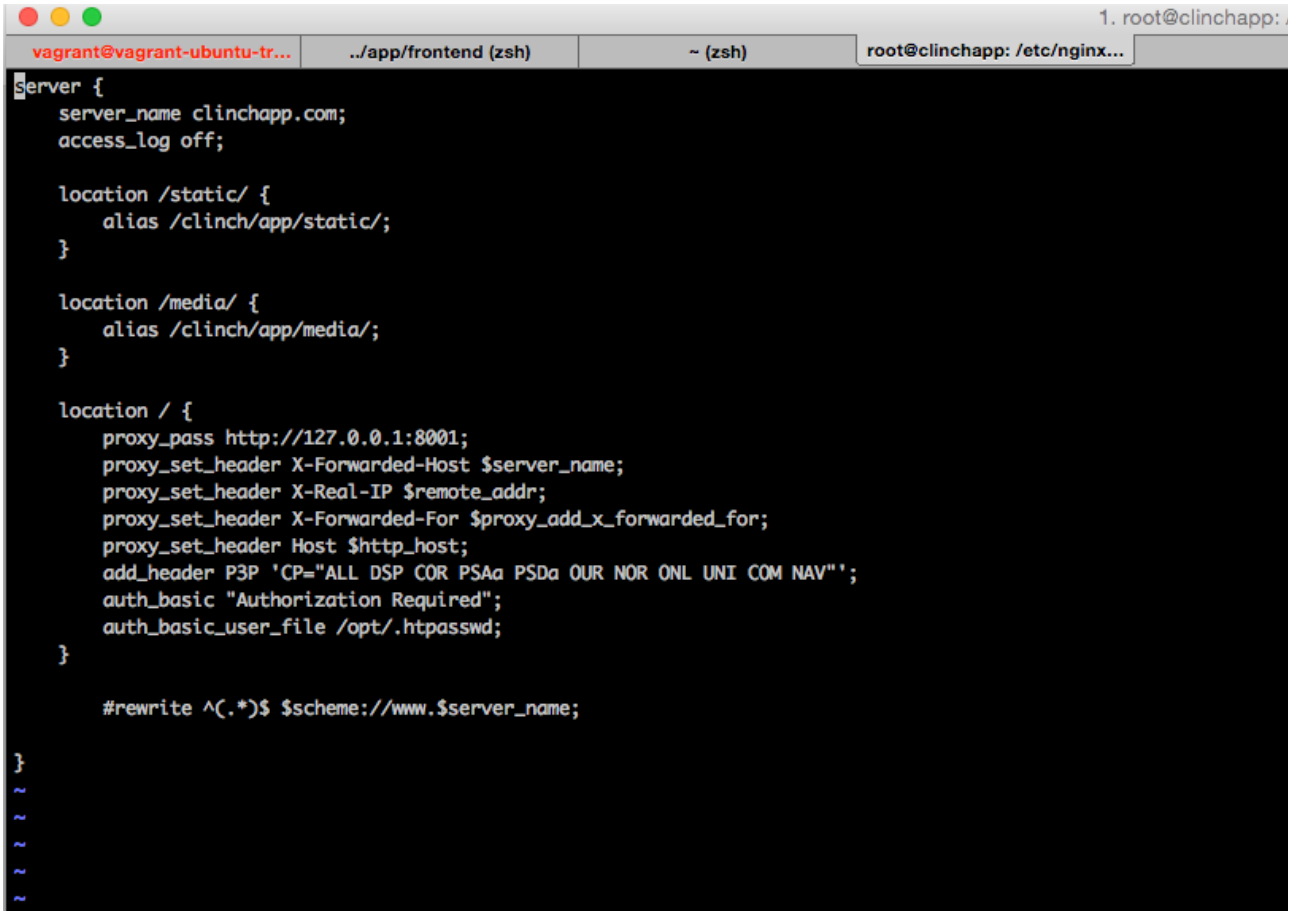
```python
class UserViewSet(viewsets.ModelViewSet):
    queryset = UserProfile.objects.all()
    serializer_class = UserSerializer
    permission_classes = (IsAuthenticated,)
    parser_classes = (FormParser, MultiPartParser,)
    paginate_by = 20

    def list(self, request):

        qs = UserProfile.objects.all()

        #Logic to retrieve a user ID based on a username, only need for web based clients
        #This allows me to preserve good urls for users

        user = request.QUERY_PARAMS.get('username', None)
        if user is not None:
            qs = qs.filter(user__username=user).values('pk')
            if qs:
                qs = qs[0]
                user_id = qs.get('pk', None)
                return Response({"user_id": user_id})
            return Response({"error: user not found"}, status=status.HTTP_404_NOT_FOUND)

        #Paginate the queryset and return response as normal if no query parameters are found

        page = self.paginate_queryset(qs)
        serializer = self.get_pagination_serializer(page)

        return Response(serializer.data)


    def update(self, request, *args, **kwargs):
        profile = UserProfile.objects.get(user=request.user)

        serializer = UserSerializer(profile, data=request.data, context={'request': request})
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            print serializer.errors

        return Response({"error":"erorr updating this record"})
```

6. Deployment

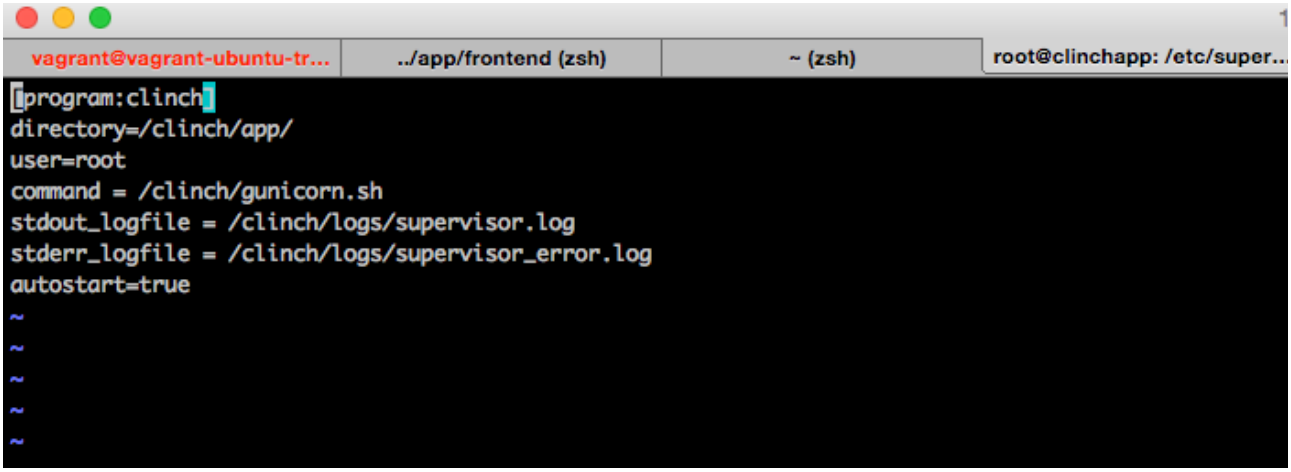




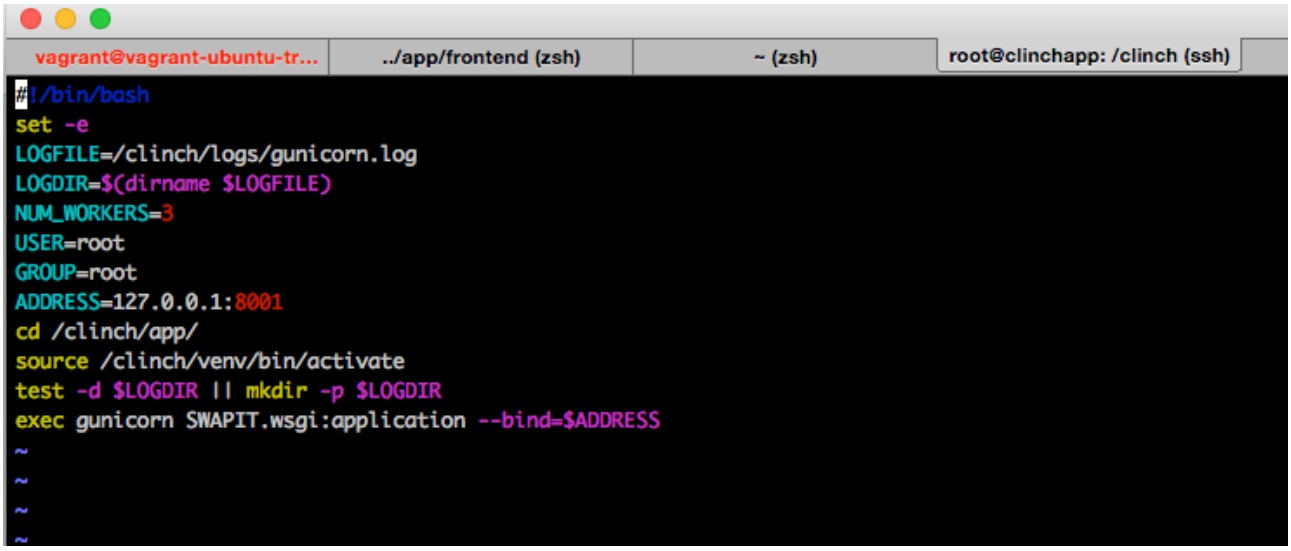
Nginx HTTP Proxy server configuration



Supervisor process, automatically starts the application after reboots



Shell script called by supervisor to start the application



Application running on port 80 reverse proxy to nginx on port 8001