

**Snap Challenges**

Photography Gamified App

**JACOB ALLEN**

**19003931**

**UXCFXK-30-3**

**Digital Systems Project**

**Text

Description automatically generated**

# **Table of Contents**

# **Abstract**

# **Acknowledgements**

# **Table of Figures**

# **Table of Tables**

# **1 – Introduction**

**1.1 – Project Timeline (Gantt Chart)**

**2021**

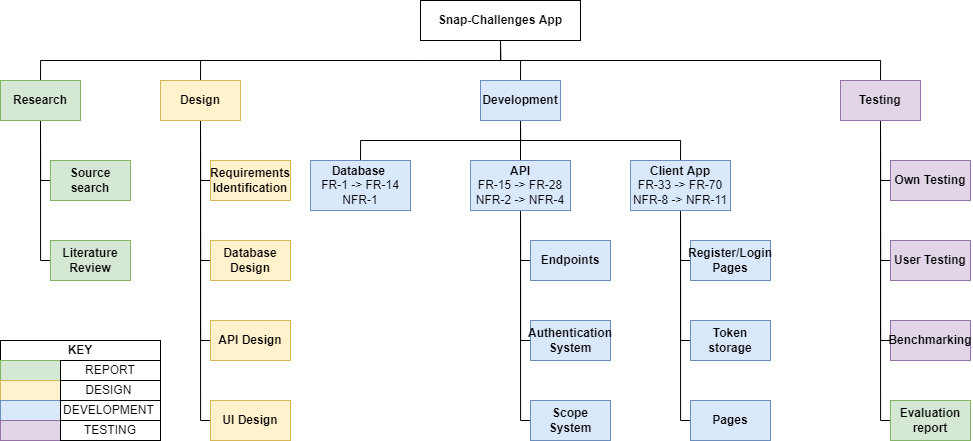
**2022**

Table 1- Gantt Chart

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | OCT | NOV | DEC | JAN | FEB | MAR | APR |
| Research |  | |  |  |  |  |  |
| Requirements |  |  |  |  |  |  |  |
| Database Design |  | **Completed – 19/11/21** |  |  |  |  |  |
| API Design |  | **Completed – 20/11/21** |  |  |  |  |  |
| UI Design |  |  | **Completed – 15/12/21** |  |  |  |  |
| API Implementation |  | **Completed - 03/01/22**  FR-15 -> FR-28  NFR-2 -> NFR-4 | |  |  |  |  |
| Database Implementation |  | **Completed - 03/12/21**  FR-1 -> FR-14  NFR-1 | |  |  |  |  |
| Front End Implementation |  |  | **Completed all must requirements – 03/03/22**  FR-33 -> FR-71  NFR-8 -> NFR-11 | | |  |  |
| Internal Testing (Own) |  |  |  |  |  |  |  |
| External Testing (User) |  |  |  |  |  |  |  |
| Finalise Report |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
| **Legend** | |
| Report Tasks |  |
| Design Tasks |  |
| Development Tasks |  |
| Testing Tasks |  |

**1.2 – Hierarchical breakdown**



**1.3 – The problem**

There are many existing popular photography sharing platforms, Instagram, 500px and Flicker to name a few. However, none of these sites offer an engaging way to get more images shared and often do not include detailed information about the photographs on the platform. Snap Challenges will attempt to resolve the gaps left by these sites by offering an engaging and informative photo sharing experience.

Photographers often experience burn out from their art, this can be for a variety of reasons, such as:

* Not being able to reach new locations.
* Not experimenting with styles outside of their comfort zone.
* High cost of new gear.
* A lack of engagement with the content they share.

Whilst Snap Challenges will not claim to be able to entirely eliminate these burn out factors, Snap challenges will attempt to mitigate this via the use of gamification. The app will provide users with challenges created by users and admins. These challenges will be regular and achievable yet challenging. This approach should result in more engagement with the platform than a typical photography sharing platform.

**1.4 – Why it’s important**

One of the main ways to improve at photography is to just get out there in the field, and practice. Snap Challenges gives the user a reason to get out and do this practice. This will hopefully lead to an improvement in their photography which benefits the whole community as we get to see better photos.

The app will show camera settings such as:

* Make
* Model
* Aperture
* Shutter Speed
* ISO
* Focal Length

Providing users with this level of detail about a photograph provides a good insight in to how the photo was taken. This will mean that the community can collaboratively improve their photography simply by observing others work.

**1.5 – Scope**

Snap-Challenges will be a social media like experience focused on gamifying photography via the use of challenges and badges. The app will feature a database, API, CDN, and a cross platform front end. The app will include several pages, a login system, a challenges page, a profile page, a badges page, and a post page. The app should be easy to use and pick up, yet remain engaging.

**1.6 – Aims and Objectives**

AIMS:

1. To develop a platform independent app to gamify photography via the use of challenges.
2. To minimise burn out amongst photographers using these challenges to engage them with the app and their hobby.
3. To educate the users of the app by providing information about the camera settings used to achieve the resulting photo.

OBJECTIVES:

1. Research gamification benefits and drawbacks, and other times it has been used in the photography field.
2. Research React Native in comparison to Progressive Web Apps (PWAs).
3. Research data storage solutions MYSQL vs NOSQL.
4. Design and develop the database schema.
5. Design and develop the Python Flask API.
6. Design and develop the front-end client as either a React Native app or PWA.
7. Test the system from both a technical and user perspective.
8. Produce a report to summarise findings, research, implementation, and testing stages of development.

**1.7 – Potential ethical and legal issues**

The system will handle user data. Passwords will be stored in the database, this could be a security concern, to minimise risk passwords will be hashed using the SHA256 algorithm and salted. This will be done via werkzeug.security a python module which is already a requirement of flask so using it doesn’t increase the number of dependencies used. The only person who will know the plain text password will be the user.

# **2 – Literature Review**

**2.1 Gamification**

(Hamari and Huotari, 2012) Define gamification as “a process of enhancing a service with affordances for gameful experiences in order to support user’s overall value creation.” This implies that utilising gamification can offer higher engagement levels from users of a service and therefore making each user more valuable to that service. This definition is backed up by a survey completed by (Wang, et al, 2017) who state, “With regard to users’ perceptions, participants found that the gamified HCS (Kpoprally) provided a better engagement experience compared to the non-gamified one.”

**What roles does gamification have to play in an app?**

Gamification can be applied via a wide variety of unique approaches, some of which have likely yet to be conceptualized. Examples of such gamification practices are:

* Rewards
* Challenges / Quests
* Achievements
* Leaderboards
* Progress Indicators

A journal article (Hence, et al, 2017) states “badges, leaderboards, and performance graphs also seemed to contribute to an increase in perceived task mean-ingfulness.” This shows that the tried-and-true methods of gamification work to ensure the user has a more engaging and positive outlook on the app in general.

Examples of such gamification can be seen throughout a wide range of applications both web and mobile, in both massively popular and small niche apps, some of which can be seen in figure 1.

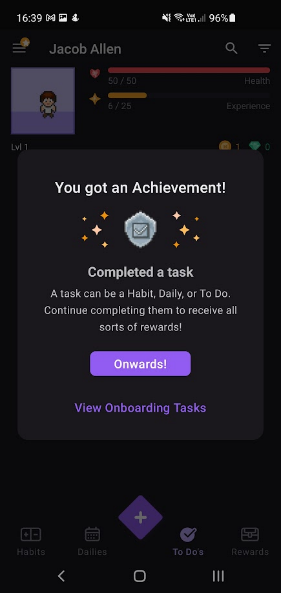
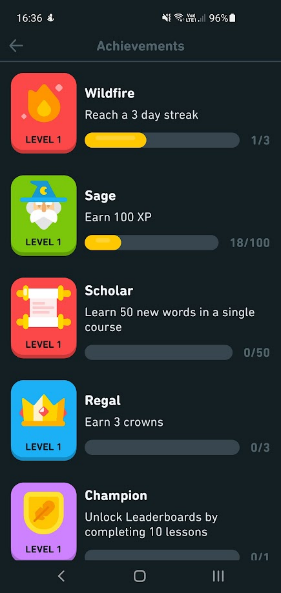


Figure 1 - Examples of Gamification in Duolingo and Habitica

**Gamification’s previous use in photography:**

There have been a few previous use cases of gamification in the photography field. One of which is PhotoTrip a website for helping tourists find less considered cultural/interesting sites, whose creators state “In the gamified version of PhotoTrip, we adopt three most common gamification strategies—points, badges, and leaderboards (PBL). Points are provided as scores, corresponding to the measure of each quest the user is involved in and badges are awarded whenever certain thresholds are met.”

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2 - Gamification being used via use of a voting system in PhotoTrip

(Bujari, et al, 2016).

The implementation of gamification in this website is a starting point however it is quite basic only utilising three gamification techniques. And whilst gamification might aid its user’s engagement, it isn’t the only factor, and unfortunately this website lacks other engaging factors such as visual design. The app feels like it was designed with minimal thought towards aesthetic and a full focus on functionality. Also, the website appears to be designed purely for desktop use, which although is fine, reduces its usability by the very demographic it tries to serve, as tourists would likely want to spontaneously check the site during their trip not only use it for preplanning.

Another use of Gamification in relation to photography was the Narva application which was used to find the location and vantage point from which historical photographs were taken as shown in figure 3.

A picture containing grass, outdoor, sky, residential

Description automatically generated

Figure 3 - An example of image positioning with Narva (Liestøl, 2018)

The Narva app’s approach to gamification was not to follow traditional implementations such as badges, quests, e.t.c. But instead, was to use traditional games such as Hot & Cold and Jigsaw and to modify their rules to fit the context of the app as is shown by the following quotes from the conference paper:

* “The project or quest in the Hot & Cold game is comparable to that of the rephotographer who seeks to discover the original photograph’s vantage point by comparing the current view with the old photograph.”
* “the relationship between individual photos, and the space they record and depict, as well as their positions, quickly pointed to the jigsaw puzzle”

(Liestøl, 2018)

However, later in the paper it is made clear that this approach wasn’t as beneficial as they initially thought. They state, “Instead of searching among traditional analogue games for defining rules comparable to and shared by the activity in question (photography), we could have adapted more superficial strategies for the implementation of game elements, using extrinsic rewards like achievements and badges”. This is likely due to the superficial nature of most humans, we see value in gaining something that is ours to keep and to show off, in this case badges or achievements. We see little motivation to do something we are not getting any value or reward out of.

**Final Findings:**

There are many varied and unique approaches to gamification, and it can be applied in a wide range of different forms, however, some forms are tried and tested and are almost guaranteed to be more engaging. These are badges/achievements, challenges, and leaderboards. This is likely due to a human desire to be rewarded and compete, people are more likely to stick with something if there is a clear and well-defined goal at the end. Completing a task or earning an achievement provides a boost of dopamine, the effect of which is “well known for their strong responses to rewards and their critical role in positive motivation.” (Bromberg-Martin, 2010)

**2.2 HTTP Protocol**

The system will partially consist of a backend API, this will be developed from scratch using Python and Flask. To successfully develop this an understanding of the HTTP Protocol will be required. In particular, an understanding of the HTTP verbs will be required.

These HTTP Verbs are defined by the HTTP Protocol (Request for comments, 1999) and the definitions are as follows:

**GET:**

“The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.”

**POST:**

“The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.”

**PUT:**

“The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the origin server MUST inform the user agent via the 201 (Created) response. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request. If the resource could not be created or modified with the Request-URI, an appropriate error response SHOULD be given that reflects the nature of the problem. The recipient of the entity MUST NOT ignore any Content-\* (e.g. Content-Range) headers that it does not understand or implement and MUST return a 501 (Not Implemented) response in such cases.”

**DELETE:**

“The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server SHOULD NOT indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location.”

The API the system will use will be a CRUD style REST API. This means that each of the CRUD operations can be mapped to different HTTP verbs as follows.

Table 2 - CRUD to HTTP Mappings

|  |  |
| --- | --- |
| CRUD | HTTP |
| CREATE | POST |
| READ | GET |
| UPDATE | PUT |
| DELETE | DELETE |

These mappings fit the definitions of the HTTP verbs provided above.

**2.3 Database Choice**

There are many different options to choose from when it comes to databases. The main comparison to look into will be between SQL vs NOSQL based databases.

**SQL:**

SQL stands for Structured Query Language and acts as the main interface between the database and the client. It was initially conceived in 1974 by IBM Researchers and was first standardized in 1986 Most SQL based database software are relational database management systems (RDBMS). These databases consist of records, fields, relations and derived relvars. These can also be described using SQL terms:

Table 3 - RDBMS vs SQL Terms

|  |  |
| --- | --- |
| Relation DB Term | SQL |
| Record | Row |
| Field | Column |
| Relation | Table |
| Derived Relvar | View |

Groff (2002) States “SQL has become the standard database management language across a broad range of computer systems and application areas, including mainframes, workstations, personal computers, OLTP systems, client/server systems, data warehousing, and the Internet.” This shows the importance of SQLs role in the relational database.

SQL is hugely successful due to its early support from IBM, throughout the early 80s IBM pushed for the mainstream adoption of SQL, via the use of a commercialized product, SQL/Data System which was announced in 1981. In 1983 IBM announced a version of this system to run on their mainframe operating system VM/CMS. Later in same year they announced DB2, which began shipping in 1985, DB2 ran on IBM’s MVS operating system, which was used by large data centres. Then in 1986 the first SQL standard ANSI SQL1 is ratified. In 1987 ISO follows and ratifies ISO SQL1. However, despite the existence of these standards many versions of SQL have small variations between each other.

**NOSQL:**

NOSQL is a “Next Generation Database Management Systems mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable.

The original intention has been modern web-scale database management systems. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge amount of data and more. So the misleading term "nosql" (the community now translates it mostly with "not only sql") should be seen as an alias to something like the definition above.” (NOSQL, 2009).

NOSQL DBs address some of the limitations of more traditional relational databases, mainly scalability, complexity, complex querying, and feature bloat.

**Scalability:**

Traditional relational databases can be hard to scale due to their design. The only ways to scale relational databases are to upgrade the hardware it is running on or distribute the database across multiple servers. However, “relational databases aren't designed to function with data partitioning, so distributing their functionality is a chore” (Leavitt, 2010)

**Complexity:**

Traditional DBs require all data to be put into tables however, not all data can be fit into tables. Because this data doesn’t fit well with the relational DB structure, it can be hard to force unsuitable data to work with this format.

**Complex Querying:**

Traditional relational DBs utilise SQL which although very strong for querying structured data, it struggles with other types as it is not designed to handle that type of data. Utilising SQL also requires a significant amount of code making it more complex. “SQL can entail large amounts of complex code and doesn't work well with modern, agile development” (Leavitt, 2010).

**Feature bloat:**

“Relational data-bases offer a big feature set and data integrity. But NoSQL proponents say database users often don't need all the features, as well as the cost and complexity they add.” (Leavitt, 2010).

**Final Verdict:**

The data that snap-challenges will be handling is very structured and lends itself well to and SQL based server as the data can nicely fit into tables. For this reason, snap-challenges will use MYSQL for its database server. However, if demand requires it a hybrid database architecture could be used by integrating a in-memory based key value store such as Redis as a cache. “Redis is an in-memory remote database that offers high performance, replication, and a unique data model to produce a platform for solving problems.” (Carlson, 2013). Redis will only be integrated into snap-challenges if demand requires it as having an in-memory cache is redundant and adds extra complexity if the app still performs well when utilising the standard MYSQL server.

**2.4 React Native vs Progressive Web App**

**What is React Native?**

“React Native is a framework for building native mobile apps in JavaScript using the React JavaScript library; React Native code compiles to real native components.” (Dabit, 2019)

React native supports many different platforms, some officially (Android and iOS), others via community-built tools which can be seen on the React Native web page covering Out-of-Tree Platforms (React Native (a), 2022) some of which include Windows Desktop, macOS, Android TV. On top of that react native also supports the web using react-native-web, and Desktop applications using Electron.

React native is open source however it is owned and maintained by Meta (previously known as Facebook). This does raise concerns given Meta’s history and less than trustworthy public opinion. On the react native web site they state “Facebook released React Native in 2015 and has been maintaining it ever since. In 2018, React Native had the 2nd highest number of contributors for any repository in GitHub.” (React Native (b), 2022).

**Benefits of React Native**

The main advantage of React Native is that it is faster than most other cross platform solutions this is because “React Native renders using its host platform’s standard rendering APIs” (Eisenman, 2017) which differs from other applications that typically render HTML, JavaScript, and CSS instead of utilising the built-in components for the OS. This ensures a native feel when using an app developed with React Native.

This native approach has also allowed for lots of community development to provide solutions for easily communicating with existing features of the host device. An example of this is the expo-camera package (Expo, 2022) which provides easy access to the devices camera and is supported on Android, iOS, and the Web. Being able to access device feature like this is not possible with most other cross platform development tools, due to their HTML rendering style approach.

Another advantage React Native has is its familiarity, it is based on React which is one of the most popular web development tools, this means developers are already aware of most of the concepts used in React Native and most can easily adjust to using react native in a small period of time.

**Drawbacks of React Native**

React Native is not perfect and there are definitely a few different disadvantages of using it. The main disadvantage is that it adds a layer of complexity to a project, which inevitably makes it harder to debug. The use of JavaScript can also be problematic as it is a dynamically typed language and strong variable declarations are not required, this also makes debugging harder, however this can also be mitigated by using TypeScript which is also officially supported by react native (React Native (c), 2022).

Another potential drawback is identified in the Learning React Native Book (Eisenman, 2017) which states, “when updates are released for the host platform—say, a new suite of APIs in a new version of Android—there will be a lag before they are fully supported in React Native”. This could lead to delays in support for the new update which could be displeasing for users.

**What is a Progressive Web App?**

“Progressive Web App (PWA) is a new generation of Web application designed to provide native app-like browsing experiences even when a browser is offline.” (Kim, et al, 2018) PWAs can be installed by the click of a button from the web browser as shown in figure 4.

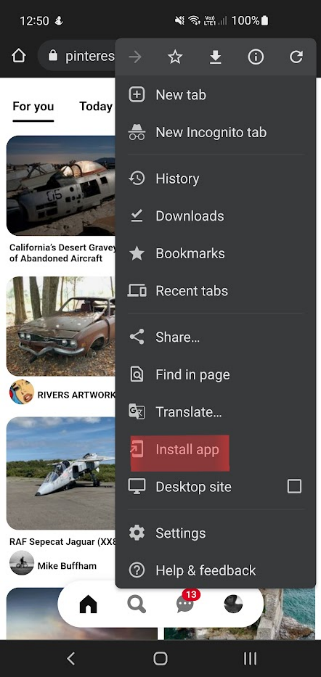


Figure 4 – PWA Install

A PWA can be created from any HTML code served via HTTPS and they can be implemented by including a web manifest, icon, and service worker with your app.

**Benefits of PWAs**

PWAs can be implemented from an existing website with extreme easy, this shortens development time.

PWAs are also supported quite heavily by Google, and they can be listed on the app store directly meaning the user doesn’t have to directly visit a website to download the app. Whilst also providing additional benefits such as those mentioned in Google’s official article on listing a PWA on their store “Google Play also offers app ratings and reviews, giving users insight into your PWA before installing it.” (Google, 2021).

**Drawbacks of PWAs**

However, whilst PWAs may be widely supported by Google, Apple have a much more restrictive policy and do not allow them on the app store stating “Your app should include features, content, and UI that elevate it beyond a repackaged website. If your app is not particularly useful, unique, or “app-like,” it doesn’t belong on the App Store.” (Apple, 2021).

Another disadvantage of PWAs is that they don’t have native access to device features. You cannot access hardware such as the camera from a PWA, this limits the functionality of the app and constrains the developer.

PWAs do however let you send HTML5 notifications, unfortunately this also has a downside as recently these notifications have been associated with scams and phishing attempts some of which are shown in figure 5.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 5 - Scam and Phishing Attempts (Kim, et al, 2018)

**Final Verdict**

Both React Native and Progressive Web Apps are suitable candidates for developing Snap Challenges, however the extra versatility offered by React Native is clearly heavily beneficial. Being able to access the device hardware is crucial for a photography-based app, after all it is going to need access the device’s camera.

**2.5 Existing Solutions**

There are a few different existing apps the main ones being GuruShots and ViewBug. However, these apps both seem to suffer from similar flaws. The main flaw being and extreme focus on a pay to win style of gamification and at times the apps almost feel like they solely exist as a quick cash and data grab scheme. Figure 4 shows some examples of features where these apps ask for payment:

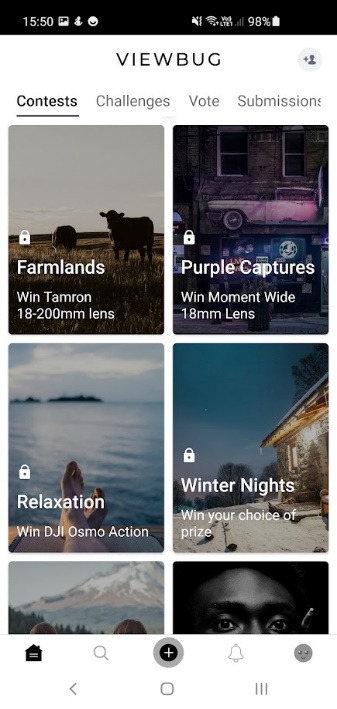
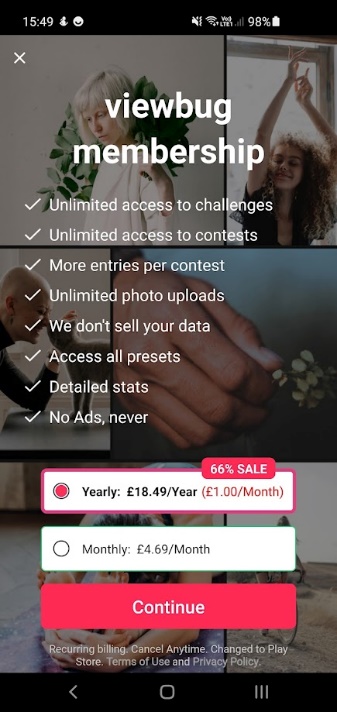
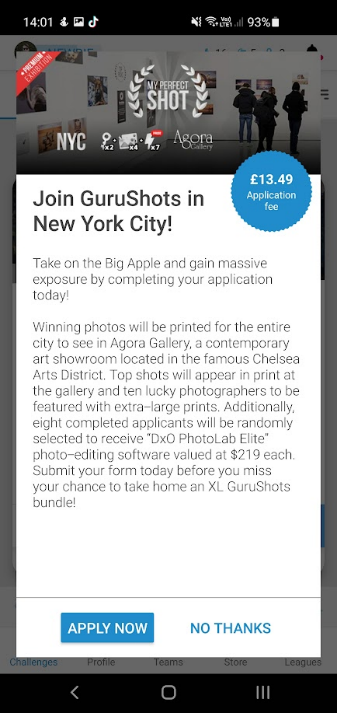
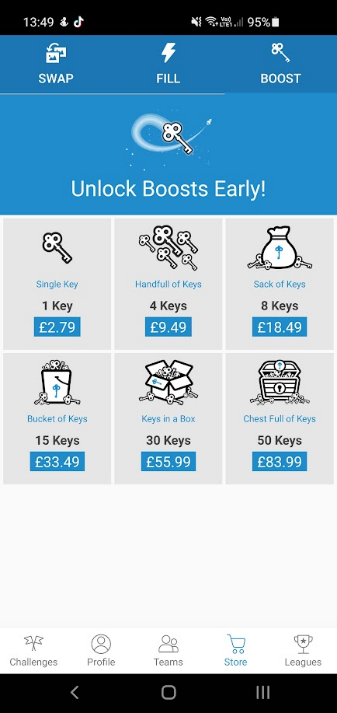


Figure 6 - Examples of Gurushots (left 2 images) and ViewBug (right 2 images) asking for payment.

Overall, Gurushots is less obtrusive with its microtransactions and membership, however ViewBug is almost unusable without a membership which costs 18.49 GBP per year. The rightmost image in figure 4 shows different contests, the small padlock above the contest name indicates that it requires this membership. As you can see there are no contents in the screenshot which can be freely accessed, with a total of 21 free contests out of a total of 78 available contests at the time of writing. This means that roughly 73% of contests at the time of writing required membership. This does somewhat make sense as these contests do offer expensive prizes, however this practice is misleading as ads for ViewBug such as the one shown in figure 5 do not show any of these locked challenges and do not mention the membership cost.

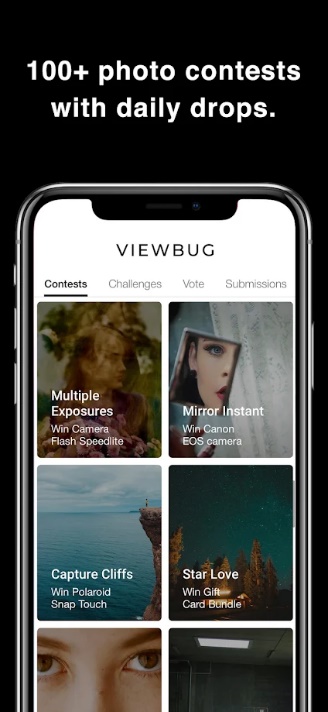


Figure 7 - ViewBug ads shown on Google Play (Google Play, 2013)

Both Gurushots and ViewBug do list camera metadata alongside images. However, this is very hidden, deep in menus, of which most users wouldn’t think to ever look through. This means it is likely ignored by all but the most curious of users and therefore is almost redundant to include.

On top of that public opinion on these apps seems to mostly be mixed. One user on the r/photography subreddit posted the question “Your Take on Apps Like GuruShots and Other Photo Challenge Apps” (Anon, 2018) to which another user submitted the following response

“

I like the 'idea' of gurushots and I like that you can see your ranking after the challenge is over.

But it encourages people to just vote on every image they see to vote more/faster. (Or to vote on sh\*\*\*y images because, maybe, they're less likely to get other votes)

I do think that truly 'great' images rise to the top, but I know when I decide to 'grind it out' and vote, I'm just voting yes as fast as possible - because NOT voting doesn't do s\*\*t for me. Scrolling past an image is a waste of effort.

If it was a tinder-like yes/no vote, and you got 'rewarded' for each one, I'd be more thoughtful with my votes.

” (Anon, 2018)

From this quote it is clear to see that photographers do enjoy the concept of a challenge-based photography app; however, it is also clear that they feel Gurushot’s implementation requires too much dedication in order too succeed.

# **3 – Requirements**

**3.1 MoSCoW:**

MoSCoW is a technique used to prioritise the requirements needed to reach a launch ready product. This is done by splitting the requirements into four separate categories, these are as follows:

* **Must have (M)** – must be included in the launch, without these components the product will not function as intended. If these are not met the launch should be seen as a failure.
* **Should have (S)** – requirements critical to the success of the system but not crucial for a minimum viable product.
* **Could have (C)** – Less critical requirements which are seen as a nice to have.
* **Won’t have (W)** – the least critical requirements, which are not seen as worth while in meeting at the current time. However, they may be worth returning to in the future.

These definitions have been adapted from “The project manager's guide to mastering agile” by (Cobb, 2015).

**3.2 Functional Requirements:**

Functional Requirements are the requirements used to clarify the functionality of the product. These requirements have been split into 4 sub sections – Database, API, CDN, and the React Native App.

**DATABASE:**

Table 4 - DB Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-1 | A Badges Table | S |
| FR-2 | A Challenges Table | M |
| FR-3 | A Challenges Has Posts Table | M |
| FR-4 | A Countries Table | M |
| FR-5 | A Photos Table | M |
| FR-6 | A Posts Table | M |
| FR-7 | A Users Table | M |
| FR-8 | A User Has Badges Table | S |
| FR-9 | A User Has Challenges Table | M |
| FR-10 | A User Has Posts Table | M |
| FR-11 | The Countries table should be auto filled with data on database initialisation. | S |
| FR-12 | Data stored in the database could be regularly backed up. | C |
| FR-13 | SQL Views to allow administrators to view data all on one table. | W |
| FR-14 | Redis in memory cache | W |

**API:**

Table 5 - API Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-15 | **Register Endpoint** – Handles a new user POST Request sent to the API.  Passwords sent to this endpoint must be hashed and salted with sha256 encryption.  **Returns:**   * 201 and JSON containing user data if completed successfully. * 400 and relevant message if the user already exists or no data was provided to the endpoint. | M |
| FR-16 | **Login Endpoint** – Handles a get request sent to it and returns a JSON Web Token (JWT) if the login is valid.  **Returns:**   * 200 and the normal JWT as well as a refresh token JWT if completed successfully. * 401 and relevant message if the user can’t be verified. | M |
| FR-17 | **Refresh** **Token** – Will be called whenever the JWT is due to expire, will return a new token. A refresh token will need to be given.  **Returns:**   * 200 and a new JWT if completed successfully. * 401 if the refresh token given is invalid. | S |
| FR-18 | **Users Endpoint** – Used for data about the user. Accepts GET/PUT and DELETE requests.  **GET returns:**   * 200 and user data if successful. * 404 if the user data can’t be found.   **PUT returns:**   * 204 and user data if successful * 404 if user not found.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-19 | **Users Has Challenges Endpoint** – Used for handling data involving the challenges users have joined. Could be used for showing a list of challenges the user has joined. Accepts GET/POST/DELETE requests.  **GET returns:**   * 200 and user challenge data if successful. * 404 if the user challenge data can’t be found.   **POST returns:**   * 201 and user challenge data if successful. * 400 if the user challenge data isn’t provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | C |
| FR-20 | **Users Has Posts** – Used for handling data involving the posts users have made. Accepts GET/POST/DELETE requests.  **GET returns:**   * 200 and user post data if successful. * 404 if the user post data can’t be found.   **POST returns:**   * 201 and user post data if successful. * 400 if the user post data isn’t provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-21 | **Users Has Badges** – Used for handling data involving the badges users have been granted. Accepts GET/POST/DELETE requests.  **GET returns:**   * 200 and user badge data if successful. * 404 if the user badge data can’t be found.   **POST returns:**   * 201 and user badge data if successful. * 400 if the user badge data isn’t given.   **DELETE returns:**   * 204 if successful * 404 if not found. | S |
| FR-22 | **Photos Endpoint** – Used for handling data about photos specified posts. Accepts GET/POST/DELETE requests.  **GET returns:**   * 200 and photo data if successful. * 404 if photo data can’t be found.   **POST returns:**   * 201 and photo data if successful. * 400 if the no photo data provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-23 | **Posts Endpoint** – Used for handling post data. Accepts GET/POST/PUT/DELETE requests.  **GET returns:**   * 200 and post data if successful. * 404 if post data can’t be found.   **POST returns:**   * 201 and post data if successful. * 400 if the no post data provided.   **PUT returns:**   * 204 and updated post data if successful. * 404 if can’t find a post to update. * 400 if required data not provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-24 | **Challenge endpoint** – Used for handling challenge data. Accepts GET/POST/PUT/DELETE requests.  **GET returns:**   * 200 and challenge data if successful. * 404 if challenge data can’t be found.   **POST returns:**   * 201 and challenge data if successful. * 400 if the no challenge data provided.   **PUT returns:**   * 204 and updated challenge data if successful. * 404 if can’t find a challenge to update. * 400 if required data not provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-25 | **Challenge Has Posts endpoint** – Used for handling data related to posts which belong to challenges. Accepts GET/POST/DELETE requests.  **GET returns:**   * 200 and challenge post data if successful. * 404 if challenge post data can’t be found.   **POST returns:**   * 201 and challenge post data if successful. * 400 if the no challenge post data provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | M |
| FR-26 | **Countries Endpoint –** Handles data about countries. Accepts GET/POST requests.  **GET returns:**   * 200 and country data if successful. * 404 if country data can’t be found.   **POST returns:**   * 201 and country data if successful. * 400 if the no country data provided. | M |
| FR-27 | **Badges Endpoint –** Handles data about badges. Accepts GET/POST/PUT/DELETE requests.  **GET returns:**   * 200 and badge data if successful. * 404 if badge data can’t be found.   **POST returns:**   * 201 and badge data if successful. * 400 if the no badge data provided.   **PUT returns:**   * 204 and updated badge data if successful. * 404 if can’t find a badge to update. * 400 if required data not provided.   **DELETE returns:**   * 204 if successful * 404 if not found. | S |
| FR-28 | **A scope system -** certain users have unrestricted access, others are limited by scopes, for example admins, mods may have access to endpoints that users wouldn’t. | C |

**CDN:**

Table 6 - CDN Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-29 | Static assets stored on the server. | M |
| FR-30 | Static assets available via a URL. | M |
| FR-31 | CDN should load assets from a server local to the user’s geo location in order to increase loading speeds. | W |
| FR-32 | An upload system that allows images to be uploaded to the cdn. | M |

**React Native App:**

Table 7 - React Native App Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-33 | A login page. | M |
| FR-34 | A sign-up page. | M |
| FR-35 | A challenges page. | M |
| FR-36 | A profile page. | M |
| FR-37 | A badges page. | S |
| FR-38 | A post page. | M |
| FR-39 | The app should support both light and dark themes and should determine which to used based on the users pre-set system preference. | S |

**Login Page:**

Table 8 - Login Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-40 | A login button which when clicked will call the API to validate the login. | M |
| FR-41 | A sign-up button which will redirect the user to the signup page. | M |
| FR-42 | A message should be displayed if login has failed. | S |
| FR-43 | If login is successful the token returned must be stored in the local storage, this will allow logins to be skipped if the token hasn’t expired. | S |

**Sign-up page:**

Table 9 - Sign-up Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-44 | A sign-up button which when clicked calls the register api endpoint with data provided in textboxes above by the user. | M |
| FR-45 | A date picker which utilises the android default date picker – on web it will just show a text input. | M |
| FR-46 | A dropdown list for selecting countries. | M |
| FR-47 | After successful sign-up the user will be redirected to the login page. | M |
| FR-48 | Input validation to warn users if their input is incorrect. This is done on the API side anyway, but a warning message on the client side would be a nice to have. | C |

**Challenges Page:**

Table 10 - Challenges Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-49 | A list of active challenges (those whose end date is in the future) will be displayed to the user pulled from the API. | M |
| FR-50 | The user will be able to click on a challenge and see all the posts for that given challenge. | M |
| FR-51 | The user should be able to click on a join button next to a challenge and be able to upload a post to said challenge. | M |
| FR-52 | A timer should be displayed next to each challenge to indicate how much time is left on the challenge. | S |
| FR-53 | The user could bookmark a challenge in order to indicate that they are interested in it but don’t immediately have an image to post for it. | C |
| FR-54 | The number of posts for each challenge could be shown on the card for each challenge. | C |

**Profile Page:**

Table 11 - Profile Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-55 | User data will be displayed, including avatar, name, country flag, and bio. | M |
| FR-56 | A selection of recent badges will be displayed on the profile. | S |
| FR-57 | Users could be able to pick which badges are displayed. | C |
| FR-58 | Number of posts shown. | M |
| FR-59 | Total kudos received shown. | M |
| FR-60 | All of the users’ posts should be shown. | M |
| FR-61 | User join date could be shown. | C |
| FR-62 | When a post is clicked it should take you to the correct post page. | M |
| FR-63 | When a badge is clicked details about it should be shown. | S |

**Badges Page:**

Table 12 - Badges Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-64 | A list of all badges available should be shown. | S |
| FR-65 | Badges the user has achieved should be shown fully whilst locked ones should be slightly less opaque. | S |
| FR-66 | Progress towards each badge could be shown. | C |

**Post Page:**

Table 13 - Post Page Functional Requirements

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR-67 | The image must be shown in its original aspect ratio. Apart from strange non-standard aspect ratios, such as incredibly tall images. | M |
| FR-68 | Camera settings should be loaded from the post data. | M |
| FR-69 | Kudos and Downvotes should be shown. | M |
| FR-70 | Time posted should be shown. | S |
| FR-71 | A menu that will allow the user to edit or delete the post if it is their own. | S |

**3.3 Non-Functional Requirements**

**Database:**

Table 14 - Database NFRs

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR-1 | An SQL query to the database should be executed within a second. | S |

**API:**

Table 15 - API NFRs

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR-2 | Any API call should respond within 3 seconds. | M |
| NFR-3 | The API will be accessible from an external network. | M |
| NFR-4 | Hosted on a Virtual Private Server (VPS) so the API will be accessible almost all the time. | C |

**CDN:**

Table 16 - CDN NFRs

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR-5 | The CDN should return content within 3 seconds on average (larger images may take longer due to their larger file sizes). | S |
| NFR-6 | Hosted on a Virtual Private Server (VPS) so the API will be accessible almost all the time. | C |
| NFR-7 | The CDN will be accessible from an external network. | M |

**React Native App:**

Table 17 - React Native App NFRs

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR-8 | Pages should load on average within 3 seconds on a 4g throttled connection. | M |
| NFR-9 | A text scale slider could be made available to allow people who struggle to read smaller font sizes to have a good experience on the app. | C |
| NFR-10 | A colour theme selector could be made to support people who struggle to read with certain colours as backdrops. | C |

**Post Page:**

Table 18 - Post Page NFRs

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR-11 | Camera info such as focal length must be clearly visible on the post page if it is available. | M |

# **4 – Methodology**

The project will utilise one of the many agile style methodologies to allow for flexible development and periodic testing.

The agile methodology focusses on the following values which are defined in the Manifesto for Agile Software Development (Beck, et al, 2001):

“

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

“

These values are upheld to ensure projects are completed on time and in a functional capacity. This is done by following the 12 agile principles defined by the Agile Alliance (Agile Alliance, 2015)

“

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

“

Agile itself has many different methodologies which follow the values and principles stated above. A few examples of these would be:

* Extreme Programming
* SCRUM
* Feature Driven Development
* Pragmatic Programming

The chosen methodology for this project is Feature Driven Development (FDD)

FDD consists of the following five stages:

Diagram

Description automatically generated

Figure 8 - Feature Driven Development (Felsing, Palmer, 2002)

Using A FDD approach ensures functionality of the app is completed by focussing on the features. More important features are prioritized as can be seen above in the requirements.

# **5 – Design**

**5.1 – High Level Design (HLD):**

High level design focuses on providing an overview of the entire system, mapping components, services and their relationships in the system.

**5.1.1 – System Architecture:**

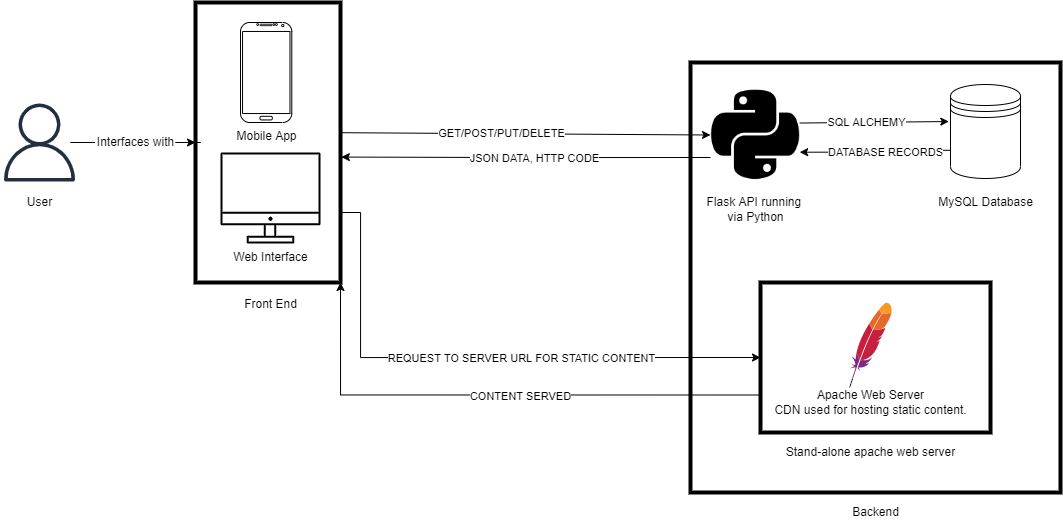


Figure 9 - System Architecture Diagram

Figure 9 shows the general architecture of the Snap Challenges system. The system can be broken down into two main components, a front-end and a back-end, these components can then be subdivided down into smaller parts. The front-end will consist of the client apps which will be built utilising react native and expo to allow for one single codebase for multiple different platforms. The back-end can be broken down into 3 core components, A database for storing information, an API to access the data and an Apache web server to host the static content such as images.

**5.1.2 – Database Design:**

Diagram

Description automatically generated

Figure 10 - Entity Relationship Diagram

Figure 10 shows the database tables and how they are linked together and dependant on each other. The tables are grouped into three separate sections. These are Post data, Challenge data and User data.

* Post data contains all the information relevant to posts on the app as well as information about the photo for the post.
* Challenge data stores all the information about challenges such ad challenge info and what posts have been submitted for each specified challenge.
* Finally, User data holds all info about the users of the system, what posts they’ve made, what badges they’ve earnt, what challenges they’ve joined and information about their country.

**5.1.3 – API Design**

Diagram

Description automatically generated with medium confidence

Figure 11 - API Design Diagram

Figure 11 shows the design diagram for the API layer. It consists of multiple different endpoints to be developed in Flask and follows a CRUD style approach based off research conducted in the Literature Review. The clients will interact with the API in a way that is obfuscated to them simply interacting with it via button presses and interactions on the app.

**5.1.4 – User Interface Design:**

Graphical user interface, application, Teams

Description automatically generated

Figure 12 - User Interface Design Composites

Figure 12 shows a general overview of the composites for each page of the app. In total there 6 pages.

Graphical user interface

Description automatically generated with medium confidence

Figure 13 - Colour Schemes

Snap Challenges should support both light and dark modes as stated in the requirements (FR-39), for this reason separate colour schemes had to be considered for each setting. The app will use 4 separate colours:

* A backdrop colour.
* A text colour.
* A contrasting colour.
* A alternate contrasting colour.

After deliberation It was decided that the colour schemes shown in Figure 13 would be used.

These are as follows:

1. Dark Mode
   1. Backdrop: #1E1E1E
   2. Text: #FFFFFF
   3. Contrast 1: #9500FF
   4. Contrast 2: #AE24FF
2. Light Mode
   1. Backdrop: #FFFFFF
   2. Text: #1E1E1E
   3. Contrast 1: #9500FF
   4. Contrast 2: #AE24FF

These colour schemes allow for an interchangeable design language as they maintain the same contrast colours, and simply invert the backdrop and text colours.

A picture containing shape

Description automatically generatedFigure 14 is the composite for the login page, it is the first page and therefore acts as an introduction to the app. For this reason it will display the Logo of the app as well as a brief text description, both large and visible.

Figure 14 - Login Page



Figure 15 - Post Page

Figure 15 is the initial design for the post page. The post page will be able to show any image of any aspect ratio (within reason) inside the image box to meet FR-67. This design clearly shows the user info about the post and camera settings, kudos and downvotes are clearly visible aswell as a description.

A picture containing polygon

Description automatically generatedPolygon

Description automatically generated

Figure 16 – Profile and Challenge Pages

Figure 16 shows both the profile and challenge pages, they have been grouped as they are very similar to one another. The top of the pages shows info about the user/challenge and the bottom shows a global component image grid. In react native a global component is one which in reused multiple times throughout the project.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 17 - Challenge List Page

Figure 17 shows the challenge list page, this page will get and display all currently active challenges in a list, displaying info such as time left on a challenge card. When clicked a card will go to the challenges page as shown in figure 16. There will also be a join button on the card which will allow users to participate in the challenge by uploading their own image.

Circle

Description automatically generated with low confidence

Figure 18 - Badge Page

Figure 18 is the badge page it will display all possible badges distinguishing between those a user has unlocked and those that they are yet to unlock, this will engage users by giving them target to aim for.

**5.2 – Low Level Design**

**5.3 – Test Design**

Table 19 - Test Design

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| X | N/A | System is in Y state. | System outputs Z | System outputted Z | Pass |

Table 19 shows how Snap-Challenges will be tested. Each test case will test a specific capability of the app and may often relate to some of the requirements stated in the requirements section of this report, if this is the case requirement ID numbers will also be provided. Each test will specify any preconditions that are present at the point of the test. For example, if a user is logged in. We then specify the result that we expected to see and the actual result. If these match the test is marked as passed, if they don’t the case is marked as failed.

# **6 – Implementation**

**6.1 - Introduction**

**6.2 – Database**

**6.2.1 – SQLite consideration**

After careful consideration and research conducted in the literature review the database technology chosen was MySQL this was due to the tabular style of the data needed to be stored. During the early stages of development, SQLite was also considered however, was not picked as “SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process.” (SQLite, 2022). This would’ve been problematic for Snap-Challenges as data needs to be shared via multiple clients therefore having a local datastore would not be an option.

**6.2.2 – SQLAlchemy & DB Models**

The database communicates with the API using the SQL alchemy library for Flask which “aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks.” (Flask-SQLAlchemy, 2021). SQL Alchemy allows for SQL commands to be obfuscated and run via python functions. Database tables are interpreted as Model Classes an example of which can be seen bellow:



Each model class represents a database table, each model class also includes a \_\_repr\_\_ method which prints out a text representation of the object. There is also a serialize function which returns the object as a dictionary (JSON Format). There are model classes for each table of data stored on the db.

**6.2.3 – Changes to the database structure**

During development of these model classes, it was found that the system could benefit from more information in comparison to that of the original database design, seen in figure 10.

Graphical user interface, text, application

Description automatically generated

Figure 19 - Updated DB ER Diagram

Figure 19 shows the updated ER which includes extra info. There are no new tables however, the photos, users, and countries tables have all been slightly modified.

The photos table now has two new columns to store information about the location and date a shot was taken. This data is optional and can be NULL.

The users table now stores a wide array of new information used to handle the login system. The reason behind this drastic change was due to the decision to switch from using Google Authentication, to a custom-built authentication system, this was done to allow for JSON web token security on the API. Usernames and Passwords are now stored in the database, the passwords are not stored as plain text for security reasons and are hashed and salted using the SHA256 algorithm before being stored on the DB. This is done for security reason so even if the database was breached user passwords wouldn’t be exposed. Info about the date registered and last login is also stored just to allow for better tracking of user’s engagement with the app.

Finally, a column was removed from the countries table as it was redundant to store a flag URL on the database as a URL could be formed using just the ISO code as the flag images are stored on the server using these codes as filenames.

**6.3 – API**

**6.3.1 – API V1**

The initial implementation of the API was developed utilising flask restful which is “an extension for Flask that adds support for quickly building REST APIs.” (Flask-RESTful, 2020). This initially allowed for a much simpler implementation of the API, it allowed for user resource classes which can then be marshalled by resource fields which can be used to validate data sent to the API and return relevant error messages. This simply was initially beneficial and allowed for a first implementation of the API to be completed within a very short period of time. However, this simplicity also became an issue that prevented the API from being secured utilising JSON web token (JWTs). In order to implement JWTs with flask restful another extension would need to be implemented Flask-JWT-Extended which “not only adds support for using JSON Web Tokens (JWT) to Flask for protecting routes, but also many helpful (and optional) features built in to make working with JSON Web Tokens easier.” (vimalloc, 2022). Whilst this was certainly an option, the decision was made to not utilise this extension as it added a further layer of complexity to the system. However, this did mean that flask restful also couldn’t be used. As a result a second implementation of the API was required. This implementation uses flasks default @app.route decorator to handle the HTTP verbs for each URL.

**6.3.2 – Tokenization**

As previously mentioned, the API is secured with JSON Web Token (JWT) authentication, this caused many issues with the implementation of the API, some of which have already been mentioned.

One of the biggest issues was that the original design didn’t take into account the use of these JWTs, the original design didn’t include any API endpoints for handling tokens. Handling tokens requires one for registering the user, one for when a user logs in and is given a token, and a final one for refreshing a token when it expires.

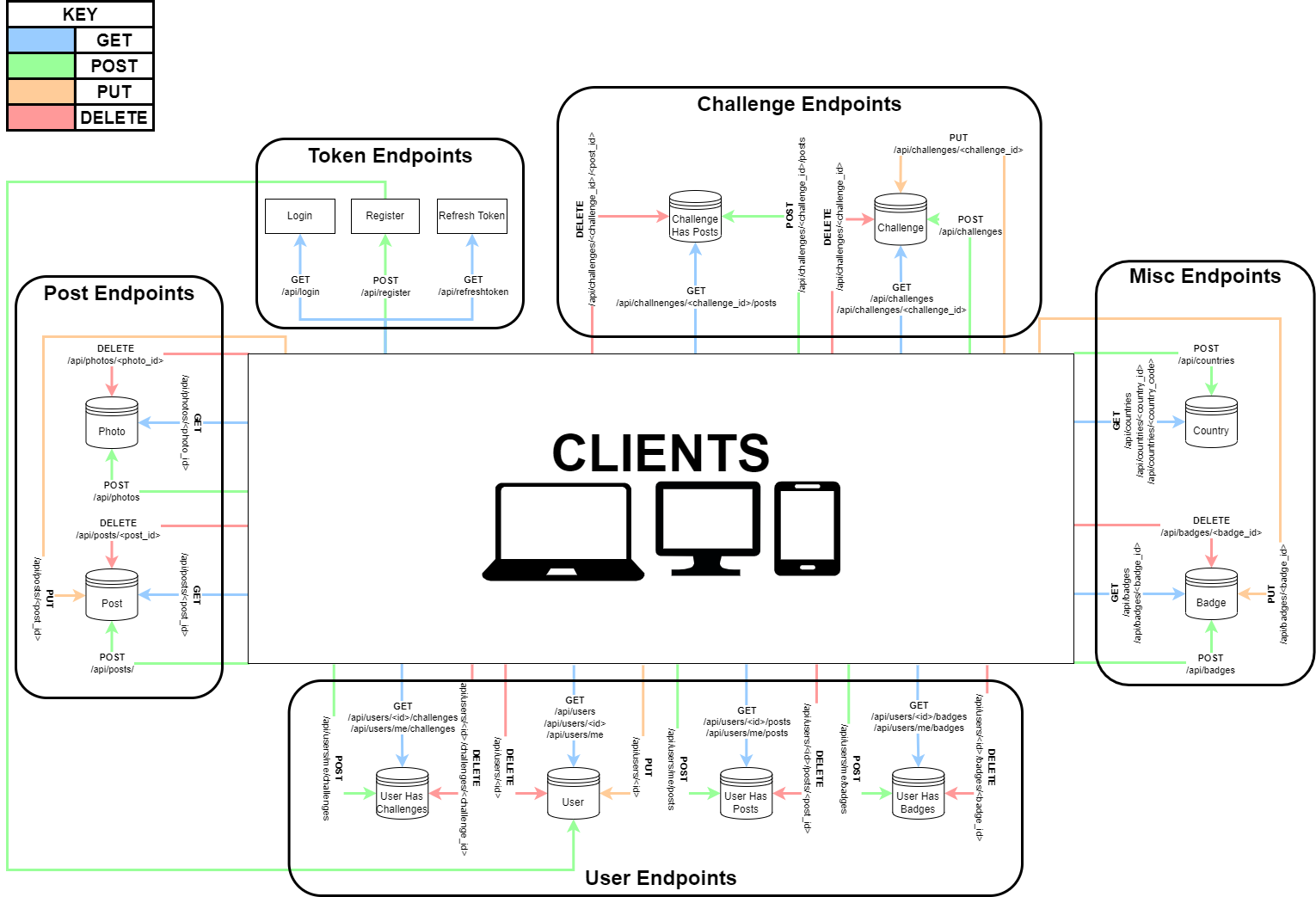


Figure 20 - Updated API Design Diagram

Figure 20 shows the updated API design which includes these new endpoints.

Tokens are generated on account login with the following route:



The JWTs are created using the PyJWT library which is installed via the python pip package manager. This library is used for encoding and decoding these JWTs.

When a login request is sent to the API via the api/login endpoint the login is first validated, if it is a valid login two JWTs are generated, one for use for validation and one for refreshing and getting a new token. The token used for validation is given the user scope which has access to the whole API, due to this it is given a shorter expiration time, this is done for security reasons as if the token is exposed it can essentially be used to falsify being the account holder and retrieve information form the API. The refresh token has a much longer expiration time as it is restricted to only being able to access one API endpoint, the refresh endpoint, this token’s whole purpose is to simply be used to generate a new validation token once the current validation token expires. On top of this basic scope information the tokens also contain the user’s public id which is used to validate which account is related to the token.

To validate tokens the following python decorator was created:



This decorator will check if a request to the decorated endpoint function contains the x-access-token header. If a token is present it is decoded by the PyJWT package to verify it is a token that was created by Snap Challenges, this is done by using the apps secret key to decrypt the token. After the token is decoded, we can access the info stored inside it, such as the user’s public id and scope. The scope is used to validate the token given is a user token and not a refresh token. The public id is used to obtain the current user’s data from the database.

**6.3.3 – CORS Prefilght**

When developing the client app, a problem was discovered with the API, whenever a HTTP request was sent to any API endpoint the JavaScript would fail and return the error, “Access to x has been blocked by cors policy”. This was initially a surprise as when testing the API in python and via Postman there were no reported errors. However, after some research, it was discovered that this error was being caused by the JavaScript fetch method which first sends an OPTIONS request to the endpoint. This OPTIONS request is essentially a verification handshake process to ensure the security of the API and is sent out by default by most devices. A simple diagram of this request is shown below in figure 21.

Diagram

Description automatically generated

Figure 21 – An OPTIONS pre-flight request (Hossain, 2014)

After a bit more research it was discovered that there are multiple ways of resolving this conundrum, the ones considered were, disabling CORS, handling these requests on the API side, or using the Flask-CORS python package to handle this.

Disabling CORS would’ve been the easiest method to resolve the issue, it would’ve been as simple as specifying the mode as no-cors in the fetch request, whilst this would’ve resolved our issue it was sort of a non-solution as doing so simply avoids the larger problem, CORS requests are sent for security reasons to verify origins and headers sent to the API. So “fixing” it by disabling the CORS doesn’t resolve the larger issue and therefore was not the solution implemented.

Flask-CORS describes itself as “A Flask extension for handling Cross Origin Resource Sharing (CORS), making cross-origin AJAX possible.

This package has a simple philosophy: when you want to enable CORS, you wish to enable it for all use cases on a domain. This means no mucking around with different allowed headers, methods, etc.” (Flask-CORS, 2022). Whilst this certainly would’ve been a valid an potentially easier solution this was also not implemented as it was a bit over complex for the use case, we only needed to allow all origins and some extras headers such as x-access-token.

That left one option implementing our own solution to handle these OPTIONS requests. This was quite simple and only required 6 lines of code extra.



This function uses flasks after\_request decorator to run this function after each request, the code inside the function handles the CORS pre-flight by generating a response with the headers and origins we wish to allow to be used to communicate with the API.

**6.4 – Apache2 Web Server**

In the architecture of Snap-Challenges the web server plays a small but crucial role which is to host static content such as the images uploaded on the app, they are stored on the webserver and accessible via a URL from any network location, not only localhost, this is crucial in order for users to view each other’s images. The server is also capable of handling image uploads directly to it, which will be utilised whenever a user uploads an image on the client, this is done via a PHP script which accepts a base64 encoded image, validates it’s in JPEG or PNG format and if so, stores the image on the server.

Development of this backend architecture went smoothly with only one minor problem arising during development. Thankfully it was a problem that had already been identified during the API development stage, The CORS check. To solve the issue was very similar in PHP as to JavaScript and just ensued ensuring the OPTIONS request gets the expected values returned.

**6.5 – Client App**

**6.6 - Completion of Aims**

**6.7 – Summary**

# **7 – Testing**

**7.1 – Introduction:**

In this section the project will be tested by both the developer and users. The developer will complete more rigorous internal testing of the different requirements implemented, whilst users will try out the app and offer feedback and suggestions for the app.

**7.2 – Internal Testing:**

**Database:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| 1 – Validating existence of DB tables after initialisation. | FR-1 -> FR-10 | The database is setup. | All tables will exist on the DB | As expected. | Pass |
| 2 – Validating that the countries table is prefilled with country data. | FR-11 | Test 1 has passed, and the countries table exists. | All country data given in the given countries.json is stored as a record on the countries table. | As expected. | Pass |

**API:**

The API has been tested using a free tool named postman.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| 3 – Valid user data is passed to the register API endpoint. | FR-15 | The database is setup.  The API is running. | Request is received and processed by the API server; HTTP Status code 201 returned alongside user data in JSON format. A record on the DB is also created. | As expected. | Pass |
| 4 – Empty data is sent to register API endpoint. | FR-15 | The database is setup.  The API is running. | Request is received and processed by the API server; HTTP Status code 400 returned alongside a “No data provided” message. | As expected. | Pass |
| 5 – An existing username is sent to the register API endpoint. | FR-15 | The database is setup.  The API is running.  A test user exists. | Request is received and processed by the API server; HTTP Status code 400 returned alongside a “User already exists.” message. | As expected. | Pass |
| 6 – Valid data sent to the login API endpoint. | FR-16 | The database is setup.  The API is running.  A test exists. | Request is received and processed by the API server; HTTP Status code 200 returned alongside tokens in JSON format. | As expected | Pass |
| 7 – Invalid login information is sent to the login endpoint e.g. an incorrect password. | FR-16 | The database is setup.  The API is running. | HTTP STATUS code 401 returned alongside a “could not verify” message. | As expected. | Pass |
| 8 – Valid GET request for an existing user is sent to the API. | FR-18 | The database is setup.  The API is running.  The test user exists. | HTTP Code 200 and user data returned. | As expected. | Pass |
| 9 – A GET request for a non-existent user is sent. | FR-18 | The database is setup.  The API is running. | HTTP Code 404 and cannot be found message returned. | As expected. | Pass |
| 10 – A valid PUT request is sent to the users endpoint. | FR-18 | The database is setup.  The API is running.  A test user exists. | HTTP code 204 returned and db record updated. | As expected. | Pass |
| 11 – A PUT request with no data is sent to users endpoint. | FR-18 | The database is setup.  The API is running. | HTTP code 400 and no data provided message returned. | As expected. | Pass |
| 12 - A PUT request for a non existent user is sent to the API | FR-18 | The database is setup.  The API is running. | HTTP code 404 and user not found message returned. | As expected. | Pass |
| 13 – A valid DELETE request is sent to the users endpoint. | FR-18 | The database is setup.  The API is running.  A test user exists. | HTTP code 204 returned and the record is removed from the db. | As expected. | Pass |
| 14 – A DELETE request is sent to users endpoint for a non-existent user. | FR-18 | The database is setup.  The API is running. | HTTP code 404 and user not found message returned. | As expected. | Pass |
| 15 – A valid GET request is sent to the Users has Posts endpoint | FR-20 | The database is setup.  The API is running.  A test user exists.  A test post exists. | HTTP code 200 and user post data returned. | As expected | Pass |
| 16 – An Invalid GET request is sent to the endpoint Users has Posts endpoint for a user that doesn’t exist. | FR-20 | The database is setup.  The API is running. | HTTP Code 404 and an error message stating that the data can’t be found. | An empty posts array is returned. Likely caused by a forgotten check to verify the user given exists. | Fail |
| 17 – A valid POST request sent to the User Has Posts Endpoint. | FR-20 | The database is setup.  The API is running.  A test user exists.  A test post exists. | HTTP Code 201 and user post data returned. | As expected. | Pass |
| 18 – An Invalid POST request containing no post\_id is sent. | FR-20 | The database is setup.  The API is running.  A test user exists. | HTTP Code 400 is returned. | Code 500 (Internal server error) is returned, a check is missing to validate that the post\_id exists. | Fail |
| 19 – A valid GET request is sent to the Photos endpoint. | FR-22 | The database is setup.  The API is running.  A test photo exists. | HTTP code 200 and photo data is returned. | As expected. | Pass |
| 20 – An invalid get request for a photo that doesn’t exist is sent. | FR-22 | The database is setup.  The API is running. | HTTP code 404 returned alongside an error message. | As expected. | Pass |
| 21 – A valid POST request is sent to the Photos endpoint. | FR-22 | The database is setup.  The API is running. | HTTP code 201 and photo data returned. | As expected. | Pass |
| 22 – Invalid POST request sent to photos endpoint containing no data. | FR-22 | The database is setup.  The API is running. | HTTP Code 400 returned and error message. | As expected. | Pass |
| 23 – Valid DELETE request sent to photos endpoint. | FR-22 | The database is setup.  The API is running.  Test photo is in the system. | HTTP code 204 returned. | As expected. | Pass |
| 24 – Invalid DELETE request sent for a photo id which doesn’t exist. | FR-22 | The database is setup.  The API is running. | HTTP code 404 returned. | As expected. | Pass |
| 25 – Valid GET request sent to the Posts endpoint. | FR-23 | The database is setup.  The API is running.  A post record exists. | HTTP code 200 and post data returned. | As expected. | Pass |
| 26 – Invalid GET request sent to posts endpoint for record that doesn’t exist. | FR-23 | The database is setup.  The API is running. | HTTP code 404 and error message. | As expected. | Pass |
| 27 – Valid POST request sent to the posts endpoint. | FR-23 | The database is setup.  The API is running.  A post exists. | HTTP code 201 and post data returned. | As expected. | Pass |
| 28 – Invalid POST request sent to posts endpoint with no post data. | FR-23 | The database is setup.  The API is running. | HTTP code 400 and error message returned. | As expected. | Pass |
| 29 – Valid PUT request sent to posts endpoint. | FR-23 | The database is setup.  The API is running.  A post record exists. | HTTP code 204, record on db is updated. | As expected. | Pass |
| 30 – Invalid PUT request sent to the posts endpoint for a post id that doesn’t exist. | FR-23 | The database is setup.  The API is running. | HTTP code 404 and error message returned. | As expected. | Pass |
| 31 – Invalid PUT request containing no data is sent to the posts endpoint. | FR-23 | The database is setup.  The API is running.  A post record exists. | HTTP code 400, and error message returned. | As expected. | Pass |
| 32 – Valid DELETE request sent to the posts endpoint. | FR-23 | The database is setup.  The API is running.  A post record exists. | HTTP code 204 returned. | As expected. | Pass |
| 33 – Invalid DELETE request sent to posts endpoint for a post id that doesn’t exist. | FR-23 | The database is setup.  The API is running. | HTTP code 404 and an error message returned. | As expected. | Pass |
| 34 – Valid GET request sent to the challenges endpoint. | FR-24 | The database is setup.  The API is running.  A challenge record exists. | HTTP code 200 and challenge data returned. | As expected. | Pass |
| 35 – Invalid GET request sent to the challenges endpoint for a challenge id that doesn’t exist. | FR-24 | The database is setup.  The API is running. | HTTP code 404 and error message returned. | Code 500, a validation check to verify the id given exists is missing. | Fail |
| 36 – Valid POST request sent to the Challenges endpoint. | Fr-24 | The database is setup.  The API is running. | HTTP code 201 and challenge data returned. | As expected. | Pass |
| 37 – Invalid empty POST request sent to the challenges endpoint. | FR-24 | The database is setup.  The API is running. | HTTP code 400 and error message returned. | As expected. | Pass |
| 38 – Valid PUT request sent to the challenges endpoint. | FR-24 | The database is setup.  The API is running.  A challenge record exists. | HTTP code 204 returned. | As expected. | Pass |
| 39 – Invalid PUT request sent to challenges endpoint for challenge id that doesn’t exist. | FR-24 | The database is setup.  The API is running. | HTTP code 404 and error message returned. | Code 500, a validation check to ensure the id given is valid is missing. | Fail |
| 40 – Invalid PUT request sent to challenge endpoint containing no data. | FR-24 | The database is setup.  The API is running.  A challenge record exists. | HTTP code 400 and error message returned. | Code 204, returned, this is a result of a change to the API since the requirements were defined, PUT requests now populate required empty fields with the previous data. So there was nbo change in the data stored. | Fail |
| 41 – Valid DELETE request sent to the challenge endpoint. From a non admin user account. | FR-24 | The database is setup.  The API is running.  A challenge record exists. | HTTP code 401 and unauthorized message. | As expected. | Pass |
| 42 – Valid DELETE request sent to the challenge endpoint. From an admin user account. | FR-24 | The database is setup.  The API is running.  A challenge record exists. | HTTP code 204. | As expected. | Pass |
| 43 – Invalid delete request sent to the challenges endpoint for a challenge id that doesn’t exist. | FR-24 | The database is setup.  The API is running. | HTTP code 404 and error message. | Code 500, validation of id missing. | Fail |
| 44 – Valid GET request sent to the challenge has posts endpoint. | FR-25 | The database is setup.  The API is running.  A challenge post exists. | Code 200 and challenge post data. | As expected. | Pass |
| 45 – Invalid GET request sent to challenge has posts endpoint for a challenge id that doesn’t exist. | FR-25 | The database is setup.  The API is running. | Code 404 and error message. | Code 200 and empty post data returned. Missing validation check on the challenge id. | Fail |
| 46 – Valid POST request sent to the challenge has posts endpoint. | FR-25 | The database is setup.  The API is running.  A challenge exists.  A post exists. | Code 201 and challenge post data returned. | As expected. | Pass |
| 47 – Invalid POST request containing no data sent to the challenge has posts endpoint. | FR-25 | The database is setup.  The API is running. | Code 400 and error message returned. | As expected. | Pass |
| 48 – Valid DELETE request sent to the challenge has posts endpoint. | FR-25 | The database is setup.  The API is running.  A challenge post exists. | Code 204 returned. | Code 500, The values used to get the record to delete are incorrect, this should be changed to use the challenge post id instead of two separate ids. | Fail |
| 49 – Invalid DELETE request sent to the challenge has posts endpoint. | FR-25 | The database is setup.  The API is running. | Code 404 returned. | Code 500, missing validation check on id. | Fail |
| 50 – Valid GET request sent to the countries endpoint. | FR-26 | The database is setup.  The API is running.  The country table is populated. | Code 200 and country data returned. | As expected. | Pass |
| 51 – Invalid GET request sent to countries endpoint for country id that doesn’t exist. | FR-26 | The database is setup.  The API is running. | Code 404 and error message returned. | Code 500, missing validation check on the id. | Fail |
| 52 – Valid POST request sent to countries endpoint. | FR-26 | The database is setup.  The API is running. | Code 201 and country data returned. | As expected. | Pass |
| 53 – Invalid POST request containing to data sent to countries endpoint. | FR-26 | The database is setup.  The API is running. | Code 400 and error message returned. | As expected. | Pass |
| 54 – Most API endpoints require the user to be authorized via a JWT sent alongside their request in the X-Access-Token header, if this is not present the API should return 401 unauthorized. | N/A | The database is setup.  The API is running. | Code 401 and unauthorized message returned. | As expected. | Pass |

**CDN:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| 55 – The CDN server should store static files used by the clients, accessible via a URL. | FR-29, FR-30 | The CDN is running.  The static files are stored on the CDN. | Files will be accessible via URL. | As expected. | Pass |

**React Native App:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| 56 – All required pages are present. | FR-33, FR-34, FR-35, FR-36, FR-38 | N/A | A login, signup, challenges, profile, and post page exists. | All pages present. | Pass |
| 57 – Dark mode is enabled. | FR-39 | N/A | The app loads in a dark theme. | As expected. | Pass |
| 58 – Light mode is enabled. | FR-39 | N/A | The app loads in a light theme. | As expected. | Pass |

**Login Page:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test case** | **Requirements Tested** | **Preconditions** | **Expected Result** | **Actual Result** | **Passed?** |
| 59 – A login button is present which when clicked logs the user in. | FR-40 | The username field is filled.  The password field is filled.  The API is running.  An account exists on the database with the given details. | The user is logged in and navigated to the challenges page. | As expected. | Pass |
| 60 – When clicked the signup button on the welcome page redirects users to the signup page. | FR-41 | None | User is navigated to signup page. | As expected. | Pass |
| 61 – After login the token is stored in local storage. | FR-43 | The valid login has taken place. | User’s token is stored in the devices local storage. | As expected. | Pass |
| 62 – When the signup button on the signup page is clicked a call to the API register endpoint is sent and the user account is created. | FR-44 | All signup fields have been populated with valid data. | API call made, user account created and record stored on DB. | As expected. |  |

**7.3 – User Testing:**

**7.4 – Conclusion:**

* API needs better validation for some endpoints to verify ids.

# **8 – Evaluation**

# **9 – Conclusion**

# **10 – References**

* Request for Comments (1999) *2616 Hypertext Transfer Protocol -- HTTP/1.1* [online]. IETF Datatracker. (no place): RFC Editor. Available from: <https://datatracker.ietf.org/doc/html/rfc2616>
* Groff, J. R. and Weinberg, P. N. (2002) *SQL : The Complete Reference.* New York: McGraw-Hill Professional. Available at: <https://search-ebscohost-com.ezproxy.uwe.ac.uk/login.aspx?direct=true&db=nlebk&AN=80523&site=ehost-live> [Accessed: 21 December 2021].
* NOSQL (2009) *NOSQL Database.* Available From: <https://hostingdata.co.uk/nosql-database/> [Accessed: 22 December 2021].
* Leavitt, N. (2010) Will NoSQL Databases Live Up to Their Promise?. *Computer* [Online]. 43 (2), pp. 12-14. [Accessed 27 December 2021].
* Carlson, J. (2013) *Redis in Action* [online] New York: Manning Publications Co. [Accessed 28 December 2021]
* Hamari, J., Huotari, K. (2012) 16th International Academic Mindtrek Conference [online], Tampere, Finland, 3-5 October 2012. Association for Computing Machinery. Available From: <https://dl-acm-org.ezproxy.uwe.ac.uk/doi/10.1145/2393132.2393137> [Accessed 05 January 2022]
* Wang, X., et al (2017) Examining the Effectiveness of Gamification in Human Computation. *International Journal of Human–Computer Interaction* [online] 33 (10), pp.813-821. [Accessed 11 January 2022]
* Hence, J., et al (2017) How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction. *Computers in Human Behavior* [online] 69 (no part/issue), pp.271-380. [Accessed 12 January 2022]
* Bujari, A., et al (2016) Using gamification to discover cultural heritage locations from geo-tagged photos. *Personal and Ubiquitous Computing* [online]. 21 (no part/issue), pp.235-252. [Accessed 16 January 2022]
* Liestøl, G. (2018). The Photo Positioning Puzzle : Creating Engaging Applications for Historical Photographs by Combining Mobile Augmented Reality and Gamification. In: 2018 3rd Digital Heritage International Congress (DigitalHERITAGE) held jointly with 2018 24th International Conference on Virtual Systems & Multimedia (VSMM 2018). [online] Digital Heritage International Congress (DigitalHeritage). Available from: [https://ieeexplore-ieee-org.ezproxy.uwe.ac.uk/document/8810038/](https://ieeexplore-ieee-org.ezproxy.uwe.ac.uk/document/8810038/%20) [Accessed 16 January 2022].
* Bromberg-Martin, E., S., et al (2010) Dopamine in Motivational Control: Rewarding, Aversive, and Alerting. *Neuron* [online]. 68(5), pp.815-834. [Accessed 16 January 2022]
* Google Play (2013) ViewBug - Photography. Available from: [https://play.google.com/store/apps/details?id=com.viewbug.viewbug&hl=en\_GB&gl=US](https://play.google.com/store/apps/details?id=com.viewbug.viewbug&hl=en_GB&gl=US%20) [Accessed 16 January 2022].
* Anon (2018) r/photography - Your Take on Apps Like GuruShots and Other Photo Challenge Apps. Available from: [https://www.reddit.com/r/photography/comments/9vnjvm/your\_take\_on\_apps\_like\_gurushots\_and\_other\_photo/](https://www.reddit.com/r/photography/comments/9vnjvm/your_take_on_apps_like_gurushots_and_other_photo/%20) [Accessed 16 January 2022].
* Eisenman, B. (2017) *Learning React Native* [online] 2nd Edition. California: O’reilly Media, Inc. [Accessed 21 January 2022]
* React Native (b) (2022) React Native · Learn once, write anywhere. Available from: <https://reactnative.dev/> [Accessed 23 January 2022].
* Dabit, N. (2019) *React Native in Action* [online] New York: Manning Publications [Accessed 23 January 2022]
* ‌React Native (a) (2022) Out-of-Tree Platforms – React Native. Available From: <https://reactnative.dev/docs/out-of-tree-platforms> [Accessed 23 January 2022]
* Expo (2022) Camera - Expo Documentation. Available from: [https://docs.expo.dev/versions/latest/sdk/camera/](https://docs.expo.dev/versions/latest/sdk/camera/%20) [Accessed 23 January 2022].
* React Native (c) (2022) Using Typescript – React Native. Available From: <https://reactnative.dev/docs/typescript> [Accessed 23 January 2022].
* Kim, H., et al (2018) *CCS: Computer and Communications Security 18* [online], Toronto, ON, Canada, 15-19 October 2018. Association of Computing Machinery. Available from: <https://dl-acm-org.ezproxy.uwe.ac.uk/doi/10.1145/3243734.3243867> [Accessed 23 January 2022]
* Google (2021) List your Progressive Web App in Google Play. Available from: <https://chromeos.dev/en/publish/pwa-in-play> [Accessed 23 January 2022].
* Apple (2021) App Store Review Guidelines - Apple Developer. Available from: <https://developer.apple.com/app-store/review/guidelines/> [Accessed 23 January 2022].
* Cobb, C. G. (2015) *The project manager's guide to mastering agile: principles and practices for an adaptive approach* [online] Hoboken, New Jersey: Wiley. [Accessed 09 Feb 2022]
* Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martrin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2001) Manifesto for Agile Software Development. Available from: <https://agilemanifesto.org/> [Accessed 9 March 2022].
* Agile Alliance (2015) 12 Principles Behind the Agile Manifesto | Agile Alliance. Available from: https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/ [Accessed 9 March 2022].
* Felsing, J. M., Palmer, S. R. (2002) *A Practical Guide to Feature-Driven Development.* 1st Edition. Upper Sadle River, NJ, USA: Prentice Hall.
* SQLite (2022) About SQLite. Available from: <https://www.sqlite.org/about.html> [Accessed 23 March 2022].
* Flask-SQLAlchemy (2021) Flask-SQLAlchemy — Flask-SQLAlchemy Documentation (2.x). Available from: https://flask-sqlalchemy.palletsprojects.com/en/2.x/# [Accessed 23 March 2022].
* Flask-RESTful (2020) Flask-RESTful — Flask-RESTful 0.3.8 documentation. Available from: https://flask-restful.readthedocs.io/en/latest/ [Accessed 24 March 2022].
* vimalloc (2022) vimalloc/flask-jwt-extended: An open source Flask extension that provides JWT support (with batteries included)! Available from: https://github.com/vimalloc/flask-jwt-extended [Accessed 24 March 2022].
* Hossain, M. (2014) *CORS in Action: Creating and consuming cross-origin APIs* [online]. New York: Manning Publications. [Accessed 24 March 2022]
* ‌ Flask-CORS (2022) Flask-CORS — Flask-Cors 3.0.10 documentation. Available from: <https://flask-cors.readthedocs.io/en/latest/> [Accessed 30 March 2022].

# **11 – Appendices**