

**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**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | OCT | NOV | DEC | JAN | FEB | MAR | APR |
| Research |  | |  |  |  |  |  |
| Requirements |  |  |  |  |  |  |  |
| Database Design |  | First Draft Completed – 19/11/21 |  |  |  |  |  |
| API Design |  | First Draft Completed – 20/11/21 |  |  |  |  |  |
| UI Design |  |  |  | |  |  |  |
| API Implementation |  |  | |  |  |  |  |
| Database Implementation |  |  | |  |  |  |  |
| Front End Implementation |  |  |  | | |  |  |
| Internal Testing (Own) |  |  |  |  |  |  |  |
| External Testing (User) |  |  |  |  |  |  |  |
| Finalise Report |  |  |  |  |  |  |  |

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

**1.2 – 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.

**1.3 – 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.

**1.4 – 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.

**1.5 – Potential ethical and legal issues**

The system will handle user data therefore it will need to conform to the General Data Protection Regulation (**GDPR**). There is no intent to store user passwords in the database as it will be more beneficial for the user to log in via an existing account they have on another platform. This will be achieved via OAuth2.

# **2 – Literature Review**

**2.1 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.

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

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

**2.2 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:

|  |  |
| --- | --- |
| 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).

**SQL VS NOSQL COMPARISSON:**

# **3 – Requirements**

**3.1 Functional Requirements:**

**API:**

|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| FR1 | A GET request sent to the Photo API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR2 | A POST request sent to the Photo API endpoint must create a record of the data on the database and return the json data of that record as well as the 201 HTTP response code. | M |
| FR3 | A PUT request sent to the Photo API endpoint must update an existing record on the database and return the HTTP response code 204. | M |
| FR4 | A DELETE request sent to the User API endpoint must DELETE a record form the database if it exists and return a 204 HTTP response code. | M |
| FR5 | A GET request sent to the User API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR6 | A POST request sent to the User API endpoint must create a record of the data on the database and return the json data of that record as well as the 201 HTTP response code. | M |
| FR7 | A PUT request sent to the User API endpoint must update an existing record on the database and return the HTTP response code 204. | M |
| FR8 | A DELETE request sent to the User API endpoint must DELETE a record form the database if it exists and return a 204 HTTP response code. | M |
| FR9 | A GET request sent to the Country API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR10 | A GET request sent to the Challenge API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR11 | A POST request sent to the Challenge API endpoint must create a record of the data on the database and return the json data of that record as well as the 201 HTTP response code. | M |
| FR12 | A PUT request sent to the Challenge API endpoint must update an existing record on the database and return the HTTP response code 204. | M |
| FR13 | A DELETE request sent to the Challenge API endpoint must DELETE a record form the database if it exists and return a 204 HTTP response code. | M |
| FR14 | A GET request sent to the Post API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR15 | A POST request sent to the Post API endpoint must create a record of the data on the database and return the json data of that record as well as the 201 HTTP response code. | M |
| FR16 | A PUT request sent to the Post API endpoint must update an existing record on the database and return the HTTP response code 204. | M |
| FR17 | A DELETE request sent to the Post API endpoint must DELETE a record form the database if it exists and return a 204 HTTP response code. | M |
| FR18 | A GET request sent to the Badge API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR19 | A POST request sent to the Badge API endpoint must create a record of the data on the database and return the json data of that record as well as the 201 HTTP response code. | M |
| FR20 | A PUT request sent to the Badge API endpoint must update an existing record on the database and return the HTTP response code 204. | M |
| FR21 | A DELETE request sent to the Badge API endpoint must DELETE a record form the database if it exists and return a 204 HTTP response code. | M |
| FR22 | A GET request sent to the ChallengePosts API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR23 | A GET request sent to the UserPosts API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR24 | A GET request sent to the UserChallenges API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR25 | A GET request sent to the UserBadges API endpoint must return the json data for the specified ID as well as the 200 HTTP response code. | M |
| FR26 | Certain information may only be retrieved from the API if an authorization token is provided alongside the initial request. | S |

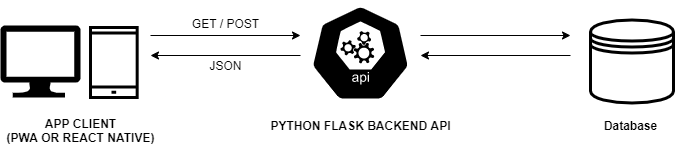
**3.2 Non-Functional Requirements**

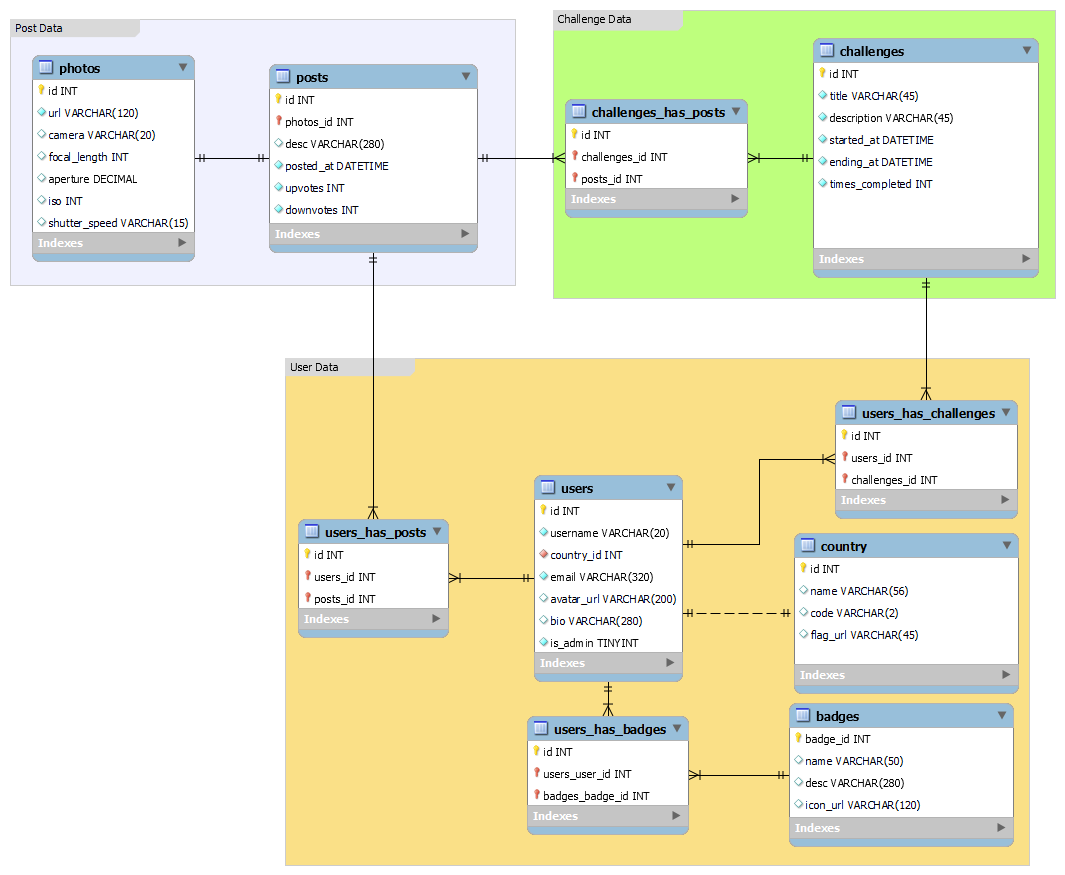
**API:**

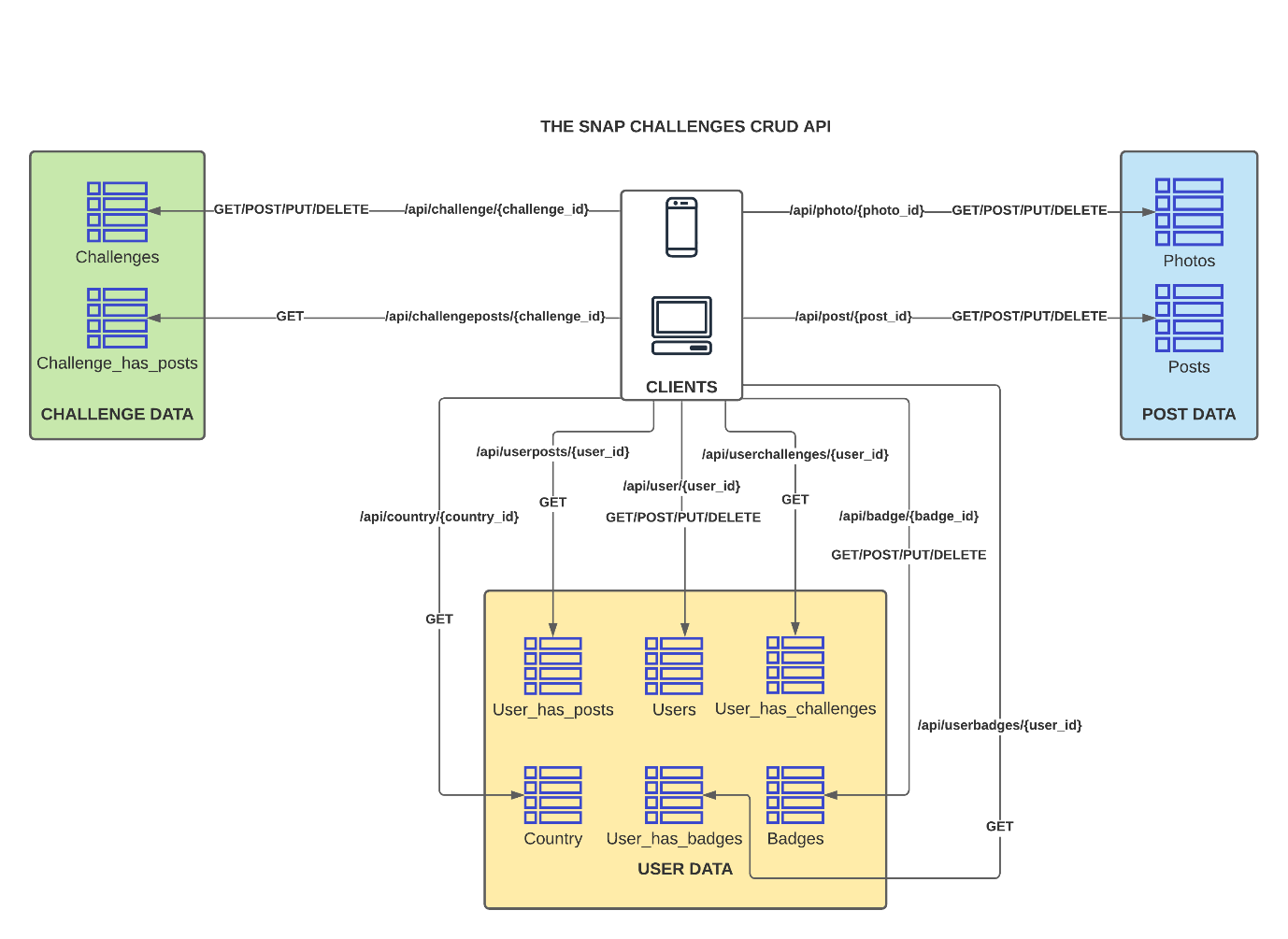
|  |  |  |
| --- | --- | --- |
| **ID** | **DESCRIPTION** | **MoSCoW** |
| NFR1 | Any API call should respond within 3 seconds. | M |

# **4 – Methodology**

# **5 – Design**







# **6 – Implementation**

# **7 – Evaluation**

# **8 – Conclusion**

# **9 – 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].

# **10 – Appendices**