Data modelling and generating SQL

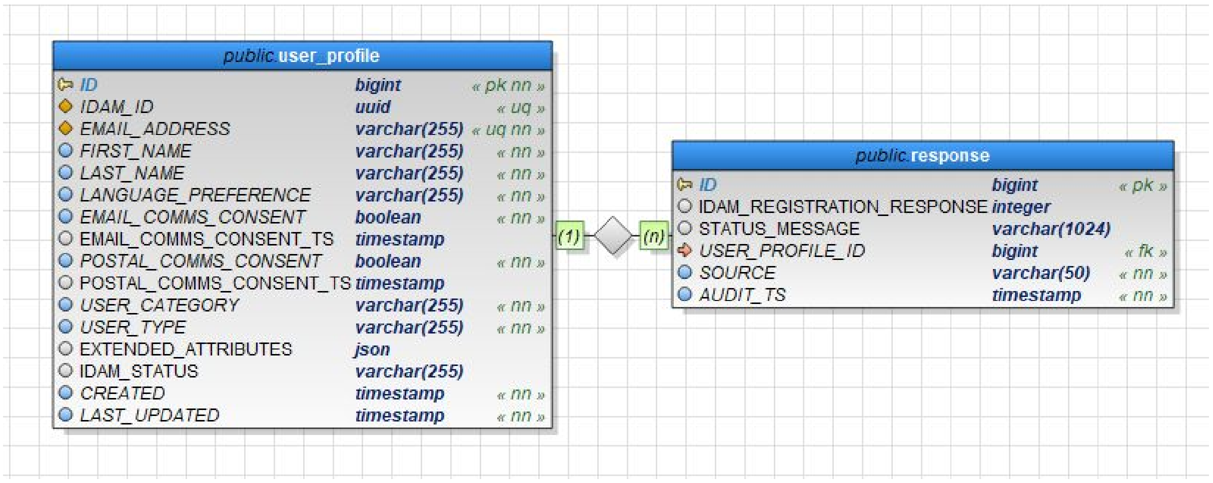
Introduction

This ticket, User Profile Data Model Finalisation, was created because the project needed clear documentation of the database structure that had been refined and clarified over time. The team and client could then access this and have a clear visualisation of the tables, columns, constraints, keys, and relationships. The purpose of spending time fine tuning this visual representation is primarily to avoid the issues that often arise as a consequence of premature coding. Analogously it is easier to amend a blueprint than a building.

The Development Process

The word finalisation is a keyword in certain respects, indicating the agile approach to the software development lifecycle (SDLC) that we use. This is an adaptive approach, meaning we adjust according to changing requirements as they arise. So rather than this work- requirement analysis and planning- being something that was completed in its entirety before development work had begun, it was an iterative process as with everything else. Being tasked with this offered me an insight into the earlier stages of the SDLC as well as development and testing. The acceptance criteria of the user story asked that I, "adhere to the design put in the confluence with all the mentioned Defaults/Validations". Confluence being a collaboration wiki tool used by all teams here, contain everything from technical requirements for individual tickets to high level overviews of various services.

My first step was to familiarise myself with the design in Confluence and to acquire the appropriate tool. I used pgModeler on the recommendation of my tech lead as our project was using postgreSQL[1]. I found pgModeler to be quite intuitive and well documented so I was able to put together the model (see Figure 1) quite quickly following the tables, column names, and data types detailed in the design. The key symbol denotes a primary key, the arrow a foreign key, the orange diamonds represent a unique constraint, blue circles denote a not-null constraint- and there are also useful acronyms down the right hand side referencing the same.



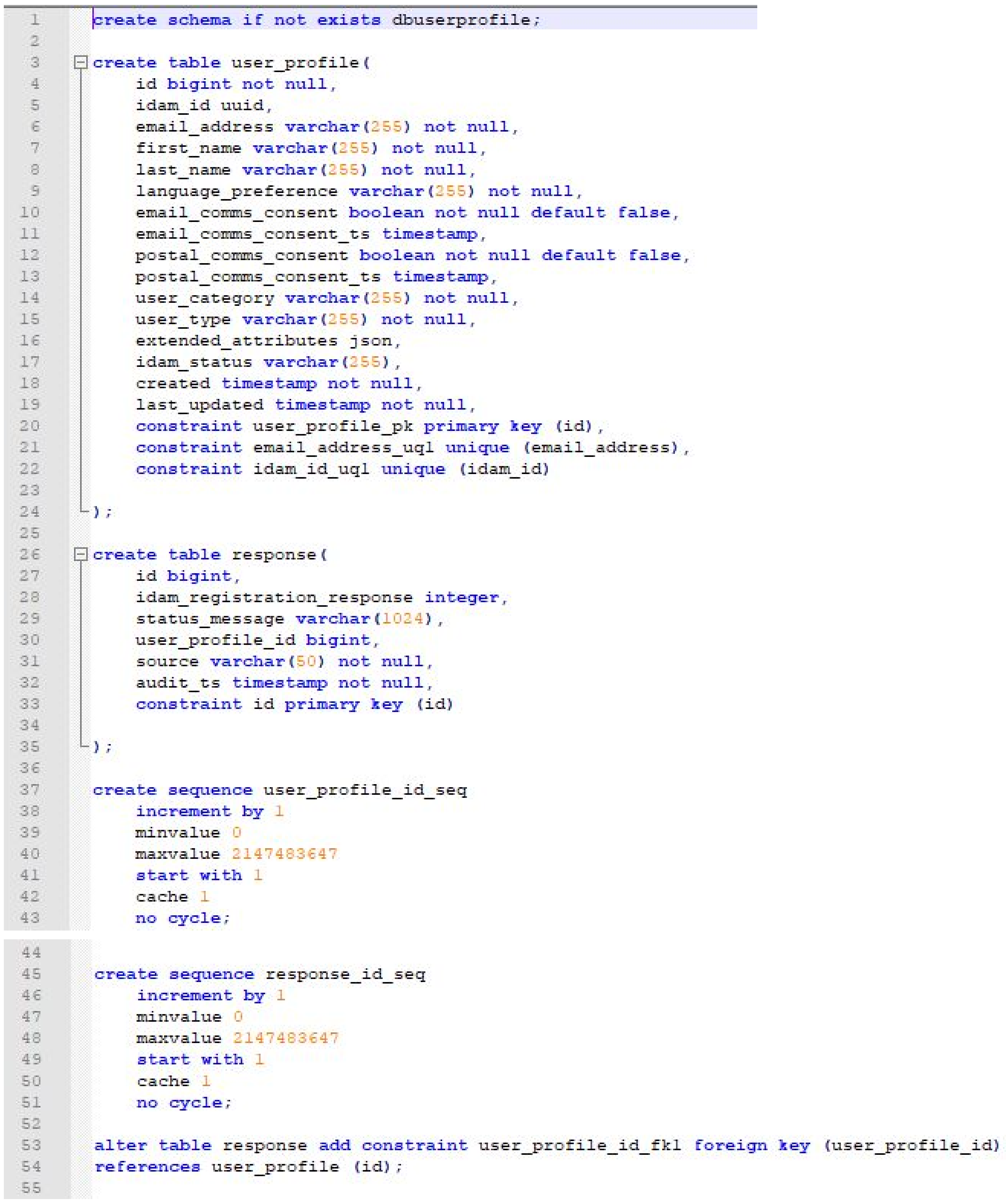
public user_profile		
ID	bigint	« pk nn »
IDAM_ID	uuid	« uq »
EMAIL_ADDRESS	varchar(255)	« uq nn »
FIRST_NAME	varchar(255)	« nn »
LAST_NAME	varchar(255)	« nn »
LANGUAGE_PREFERENCE	varchar(255)	« nn »
EMAIL_COMMS_CONSENT	boolean	« nn »
EMAIL_COMMS_CONSENT_TS	timestamp	
POSTAL_COMMS_CONSENT	boolean	« nn »
POSTAL_COMMS_CONSENT_TS	timestamp	
USER_CATEGORY	varchar(255)	« nn »
USER_TYPE	varchar(255)	« nn »
EXTENDED_ATTRIBUTES	json	
IDAM_STATUS	varchar(255)	
CREATED	timestamp	« nn »
LAST_UPDATED	timestamp	« nn »

public response		
ID	bigint	« pk »
IDAM_REGISTRATION_RESPONSE	integer	
STATUS_MESSAGE	varchar(1024)	
USER_PROFILE_ID	bigint	« fk »
SOURCE	varchar(50)	« nn »
AUDIT_TS	timestamp	« nn »

Figure 1 – design translated to visual model

By each column name there's also data type details, and the one-to-many relationship between the two tables is also illustrated.
My next goal was to generate the SQL (see figure 2) and embed the updated schema within the project.

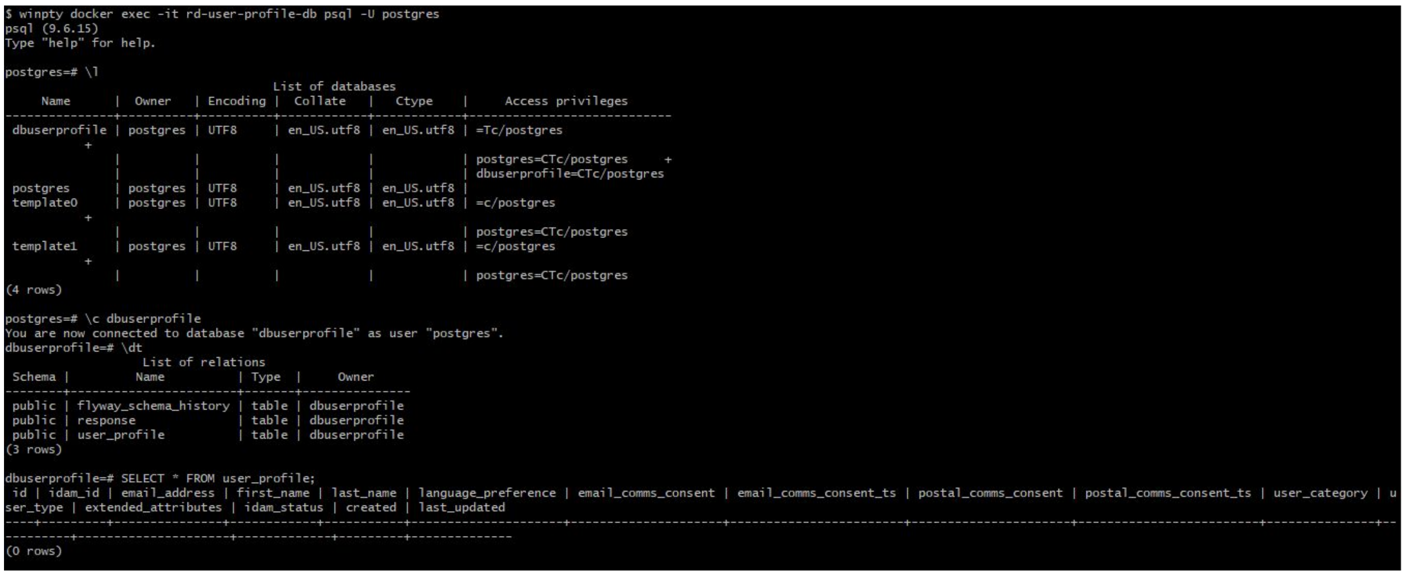
[1] PostgreSQL is a popular, well supported, database management system that supports SQL for relational queries.



```sql
create schema if not exists dbuserprofile;

create table user_profile(
    id bigint not null,
    idam_id uuid,
    email_address varchar(255) not null,
    first_name varchar(255) not null,
    last_name varchar(255) not null,
    language_preference varchar(255) not null,
    email_comms_consent boolean not null default false,
    email_comms_consent_ts timestamp,
    postal_comms_consent boolean not null default false,
    postal_comms_consent_ts timestamp,
    user_category varchar(255) not null,
    user_type varchar(255) not null,
    extended_attributes json,
    idam_status varchar(255),
    created timestamp not null,
    last_updated timestamp not null,
    constraint user_profile_pk primary key (id),
    constraint email_address_uq1 unique (email_address),
    constraint idam_id_uq1 unique (idam_id)

);

create table response(
    id bigint,
    idam_registration_response integer,
    status_message varchar(1024),
    user_profile_id bigint,
    source varchar(50) not null,
    audit_ts timestamp not null,
    constraint id primary key (id)

);

create sequence user_profile_id_seq
    increment by 1
    minvalue 0
    maxvalue 2147483647
    start with 1
    cache 1
    no cycle;

create sequence response_id_seq
    increment by 1
    minvalue 0
    maxvalue 2147483647
    start with 1
    cache 1
    no cycle;

alter table response add constraint user_profile_id_fk1 foreign key (user_profile_id)
references user_profile (id);
```

Figure 2 – the data model as SQL

Before putting in a pull request I needed to test the changes locally. To do this I ran the Docker container and then in git bash connected to the database (see figure 3).

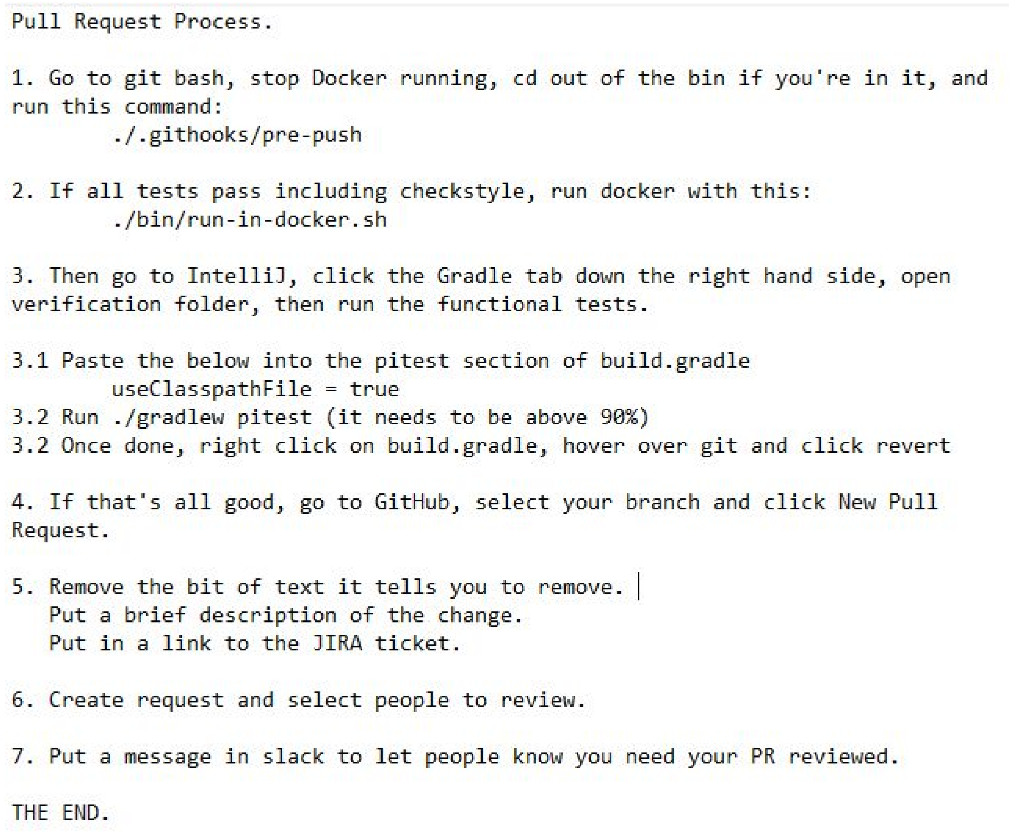


```
$ winpty docker exec -it rd-user-profile-db psql -U postgres
psql (9.6.15)
Type "help" for help.

postgres=# \l
                                  List of databases
     Name      |  Owner   | Encoding |  Collate   |   Ctype    |     Access privileges
---------------+----------+----------+------------+------------+-----------------------------
 dbuserprofile | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =Tc/postgres
          +
               |          |          |            |            | postgres=CTc/postgres     +
               |          |          |            |            | dbuserprofile=CTc/postgres
 postgres      | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0     | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
          +
               |          |          |            |            | postgres=CTc/postgres
 template1     | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres
          +
               |          |          |            |            | postgres=CTc/postgres
(4 rows)

postgres=# \c dbuserprofile
You are now connected to database "dbuserprofile" as user "postgres".
dbuserprofile=# \dt
                List of relations
 Schema |         Name          | Type  |     Owner
--------+-----------------------+-------+---------------
 public | flyway_schema_history | table | dbuserprofile
 public | response              | table | dbuserprofile
 public | user_profile          | table | dbuserprofile
(3 rows)

dbuserprofile=# SELECT * FROM user_profile;
 id | idam_id | email_address | first_name | last_name | language_preference | email_comms_consent | email_comms_consent_ts | postal_comms_consent | postal_comms_consent_ts | user_category | u
ser_type | extended_attributes | idam_status | created | last_updated
----+---------+---------------+------------+-----------+---------------------+---------------------+------------------------+----------------------+-------------------------+---------------+--
---------+---------------------+-------------+---------+--------------
(0 rows)
```

Figure 3 – Connecting to the database locally

The first line in Figure 4 shows the command to make a connection to postgreSQL. Then after listing the databases I connect to the relevant one, and after listing the tables, the last command shown is a CRUD statement which asks to show all things in the user profile table. The table shows all of the columns corresponding to the updated model.

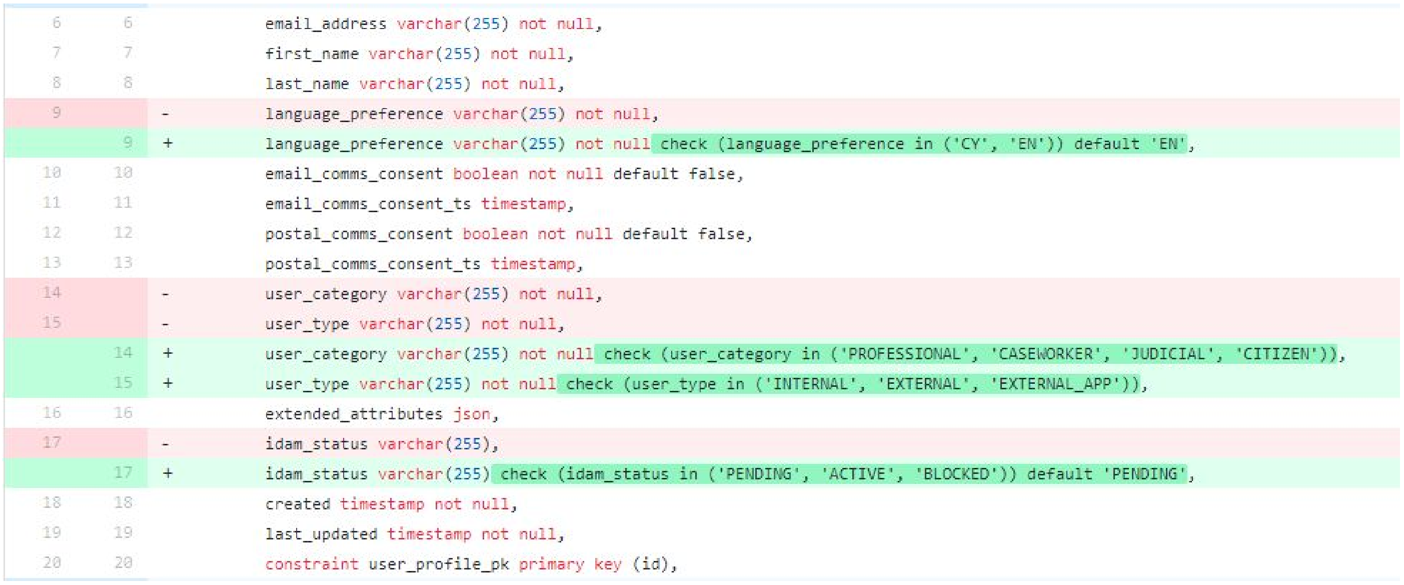
This user story provided me with exposure to the team approach to version control. For every ticket, or user story, the developer takes a branch from master, names it according to the agreed upon naming convention, and makes sure to fetch/pull from master regularly to ensure the codebase on their local machine is up to date. This practice was fairly straightforward to follow and familiar from my prior training. Then when it came to merging my completed work with master I learned there were a number of steps to be taken in order to do this. I therefore documented the process (figure 4), so as to have a reference for this process until such time it was committed to memory- and to help other new joiners.



```
Pull Request Process.

1. Go to git bash, stop Docker running, cd out of the bin if you're in it, and
run this command:
        ./.githooks/pre-push

2. If all tests pass including checkstyle, run docker with this:
        ./bin/run-in-docker.sh

3. Then go to IntelliJ, click the Gradle tab down the right hand side, open
verification folder, then run the functional tests.

3.1 Paste the below into the pitest section of build.gradle
        useClasspathFile = true
3.2 Run ./gradlew pitest (it needs to be above 90%)
3.2 Once done, right click on build.gradle, hover over git and click revert

4. If that's all good, go to GitHub, select your branch and click New Pull
Request.

5. Remove the bit of text it tells you to remove.
   Put a brief description of the change.
   Put in a link to the JIRA ticket.

6. Create request and select people to review.

7. Put a message in slack to let people know you need your PR reviewed.

THE END.
```

Figure 4 – Pull request process documentation.

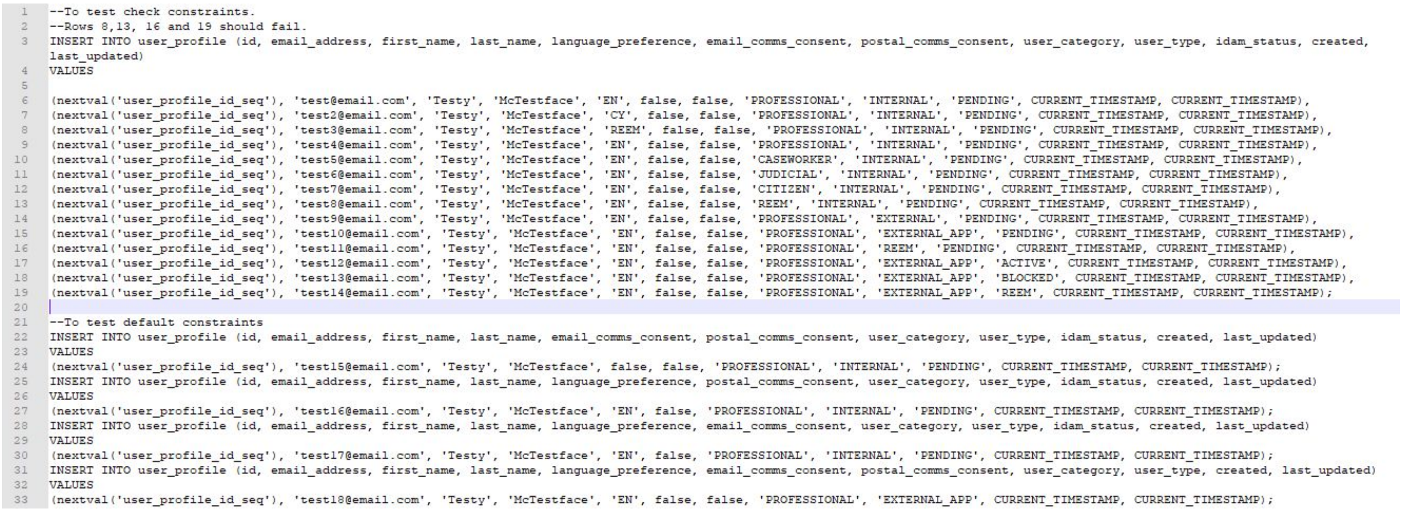
This is a truncated version as often this would prove to be an iterative process: step 1 would likely flag up Checkstyle[2] issues to attend to, and ultimately when you reached the end of the series of steps in Figure 5, you might find the build had failed due to issues raised by SonarQube[3] which would need to be addressed, or perhaps the build might pass but your reviewers have attached comments offering suggestions for improvements, as was the case for me.
The process of working on this user story involved working with others as well as alone. While the development work for this user story was largely autonomous, occasionally discussing it with a team mate working on related tickets, it is always particularly invaluable to liaise with QA. Our QA colleague alerted me to missing check constraints and default enum values which required me to revisit the SQL. After making the necessary changes (See figure 5), I updated him via the JIRA.



```
            email_address varchar(255) not null,
            first_name varchar(255) not null,
            last_name varchar(255) not null,
 -          language_preference varchar(255) not null,
 +          language_preference varchar(255) not null check (language_preference in ('CY', 'EN')) default 'EN',
            email_comms_consent boolean not null default false,
            email_comms_consent_ts timestamp,
            postal_comms_consent boolean not null default false,
            postal_comms_consent_ts timestamp,
 -          user_category varchar(255) not null,
 -          user_type varchar(255) not null,
 +          user_category varchar(255) not null check (user_category in ('PROFESSIONAL', 'CASEWORKER', 'JUDICIAL', 'CITIZEN')),
 +          user_type varchar(255) not null check (user_type in ('INTERNAL', 'EXTERNAL', 'EXTERNAL_APP')),
            extended_attributes json,
 -          idam_status varchar(255),
 +          idam_status varchar(255) check (idam_status in ('PENDING', 'ACTIVE', 'BLOCKED')) default 'PENDING',
            created timestamp not null,
            last_updated timestamp not null,
            constraint user_profile_pk primary key (id),
```

Figure 5 – Amended SQL script.

Our QA colleague and I then discussed how these constraints and default values could be comprehensively tested, which proved to be a really interesting and useful insight into the QA role. Working autonomously again, I wrote the SQL statements in figure 6, with predictions as to which statements are expected to fail and why these expectations fulfil the requirements of the acceptance criteria.



```
1  --To test check constraints.
2  --Rows 8,13, 16 and 19 should fail.
3  INSERT INTO user_profile (id, email_address, first_name, last_name, language_preference, email_comms_consent, postal_comms_consent, user_category, user_type, idam_status, created,
   last_updated)
4  VALUES
5
6  (nextval('user_profile_id_seq'), 'test@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
7  (nextval('user_profile_id_seq'), 'test2@email.com', 'Testy', 'McTestface', 'CY', false, false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
8  (nextval('user_profile_id_seq'), 'test3@email.com', 'Testy', 'McTestface', 'REEM', false, false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
9  (nextval('user_profile_id_seq'), 'test4@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
10 (nextval('user_profile_id_seq'), 'test5@email.com', 'Testy', 'McTestface', 'EN', false, false, 'CASEWORKER', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
11 (nextval('user_profile_id_seq'), 'test6@email.com', 'Testy', 'McTestface', 'EN', false, false, 'JUDICIAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
12 (nextval('user_profile_id_seq'), 'test7@email.com', 'Testy', 'McTestface', 'EN', false, false, 'CITIZEN', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
13 (nextval('user_profile_id_seq'), 'test8@email.com', 'Testy', 'McTestface', 'EN', false, false, 'REEM', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
14 (nextval('user_profile_id_seq'), 'test9@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
15 (nextval('user_profile_id_seq'), 'test10@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL_APP', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
16 (nextval('user_profile_id_seq'), 'test11@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'REEM', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
17 (nextval('user_profile_id_seq'), 'test12@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL_APP', 'ACTIVE', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
18 (nextval('user_profile_id_seq'), 'test13@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL_APP', 'BLOCKED', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP),
19 (nextval('user_profile_id_seq'), 'test14@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL_APP', 'REEM', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
20
21 --To test default constraints
22 INSERT INTO user_profile (id, email_address, first_name, last_name, email_comms_consent, postal_comms_consent, user_category, user_type, idam_status, created, last_updated)
23 VALUES
24 (nextval('user_profile_id_seq'), 'test15@email.com', 'Testy', 'McTestface', false, false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
25 INSERT INTO user_profile (id, email_address, first_name, last_name, language_preference, postal_comms_consent, user_category, user_type, idam_status, created, last_updated)
26 VALUES
27 (nextval('user_profile_id_seq'), 'test16@email.com', 'Testy', 'McTestface', 'EN', false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
28 INSERT INTO user_profile (id, email_address, first_name, last_name, language_preference, email_comms_consent, user_category, user_type, idam_status, created, last_updated)
29 VALUES
30 (nextval('user_profile_id_seq'), 'test17@email.com', 'Testy', 'McTestface', 'EN', false, 'PROFESSIONAL', 'INTERNAL', 'PENDING', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
31 INSERT INTO user_profile (id, email_address, first_name, last_name, language_preference, email_comms_consent, postal_comms_consent, user_category, user_type, created, last_updated)
32 VALUES
33 (nextval('user_profile_id_seq'), 'test18@email.com', 'Testy', 'McTestface', 'EN', false, false, 'PROFESSIONAL', 'EXTERNAL_APP', CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
```

Figure 6 – SQL statements to test constraints and default values.

[2] Checkstyle is a code analysis tool for ensuring Java source code complies with good code presentation practices that improve the quality, readability, re-usability of the code.
[3] SonarQube is a code analysis tool that automatically inspects and reviews code, checking for bugs, security vulnerabilities and other potential issues.

When I was able to successfully test my amended SQL, I updated my pull request and ultimately received the signoff from our QA colleague as well.

Reflective Statement

This project challenged me to dust off foundational knowledge of SQL from boot camp training with Makers Academy and build on it. While SQL is quite easy to get started with, creating simple tables and writing select queries is straight-forward enough, there's a lot more you can do with it to add specificity and detail to the both your tables and queries. I don't feel it was particularly difficult to meet this challenge as Google yields plenty of resources on SQL and PostgreSQL, rather it was a reminder there's always more to learn.

Another challenge was interpreting the acceptance criteria of the user story correctly. In this case my ticket directed me to reference a table in a Confluence page. In hindsight I might have saved some time by discussing it with our Business Analyst to ensure my understanding as a first step on receiving the user story.

The process of completing the work for this user story, and the latter challenge above, has improved my process by forcing me to think more about the development process as it relates to the steps both preceding and following the actual coding. Collaborating with the Business Analyst to clarify the details of the user story and with the QA to understand how the work is tested only improves the flow of the overall process, and if I were to do this task again I'd make sure to do this in the first instance.

As a result of this work the team and client were able to benefit from an updated data model, I grew my understanding of data modelling, database building and querying considerably, and my colleague was able to press forward with his related tickets and work on an impact assessment on existing endpoints.