BITS Bug Issue Tracker Solutions

Project Procedure:

Django version 1.11.16 was installed. Version 2 was not considered as it is still in development.
The project was then started using django-admin startproject PROJECTNAME . In this case the project name used was issue-tracker and is situated inthe root directory. Thie issue-tracker project dir contains the settings.py file and the root urls.py file.

The allowed user (in this case cloud9 for local use) was typed into the allowed user in the settings.py file, to allow the file to be run during development. For production on Heroku the allowed user was added.

An env file was established (to hide any secret keys or passwords) and imported into the settings.py file (for use in the local environment during development. It was omitted prior to production).

A git ignore file was then created to be used to omit certain files to the git commit. (Files included "*.sqlite3\n*.pyc\n.~c9\n__pycache__/\nenv.py").

The following procedure was adopted to create apps and varied across apps depending on whether models or forms where required:

Apps were started using the following command, django-admin startapp APPNAME. Thus the accounts app for example was started using django-admin startapp accounts.

Once the app was started, it was added to the settings.py file.
A view was added to the app.views to call a url to that would return a document, a response or call some json type data, in the case of graphs. Initially the url patterns were written into the main (project) urls.py file but eventually it was considered best to create an app specific urls.py file, which then related back to the main root urls file using the include method. For example the accounts urls.py file handles all authorisation and authenticating. In addition the accounts app contains a backends.py file linked to a custom authentication back-end in the settings.py file , enabling login using an email address rather than just a username.

Generally where forms were being used to take data from a user, model and form files were established. Models with fields were set up and then the django forms library was used to set up forms.
Models were made accessible through the admin back-end, by opening up the admin.py file and importing the model class and registering the modelclass.

Once a model was created or updated the makemigrations command was used followed by the migrate command, thus ensuring the relevant db tables were generated or amended.
A superuser (*python3 manage.py createsuperuser)* was set up allowing access to the administration panel. 2 superusers are available on this project.
A run shortcut was estabilished by adding the workspace path and run command to the aliases file. Thus typing run would run the project.
View files were used to derive functions to return the various html pages and forms, to fire urls that would redirect to other vies or html pages, to create instances of users and items.

The template DIRS[] in the settings.py file wa updated to ensure that all template dirs potentially contain templates.

Static and media directories were also set up in the settings.py file to handle media and static files. The context processors section within templates in the settings.py file was modified by adding 'django.template.context_processors. Media'

The Pillow library was installed using sudo pip3 install pillow, thereby adding support to open, manipulate, and save different image file formats.

sudo pip3 install django-vote was used to add upvote functionality.

Django bootstrap forms were used with the library and installed using sudo pip3 install django_forms_bootstrap. {% load bootstrap_tags %} and {{ form | as_bootstrap }} were then used within the html documents.

sudo pip3 install dj-database-url was used to enable connection to a database url in heroku.
sudo pip3 install psycopg2 installed a package allowing connection to a postgresql database.
The database was imported into the application using import dj_database_url in the settings.py file.
A requirements.txt file was installed to the root directory, listing the dependencies and allowing heroku to build the application when deployed to heroku.

Wiring up to the aws cloud server:
Sign in to account.
Use s3 service to create a bucket
Set permissions to public
Create bucket
Click on the properties tab, then select static website hosting
Select use this bucket to host a website
Enter the permissions tab

Enter a the bucket policy in the bucket policy tab
Save and return to the main page of aws
Search for iam to manage who can access the services
Create group. In this case it was called manage-bits-tracker
Create the policy for the group (in this case an existing existing CRUD policy was imported as it has similar permissions, but was edited to be specific for the bits-tracker)
Attach the policy to the group and add user (in this case admin-bits)
Assign group permissions

Wiring django to s3
Install 2 packages to facilitate this (sudo pip3 install django-storages and sudo pip3 install boto3)
Add storages to installed apps in the settings.py file
Update the settings.py file with the aws parameters
Add secret keys to the env.py file (for running locally)
Use (python3 manage.py collectstatic) to run collect static and select yes to upload static files
A custom storages mechanism is set up to facilitate storage of static files in one directory and storage of media files in another directory. Then modify settings.py file to accept custom static and media storage.

run collect static again to upload static files to static directory and delete the other directories with duplicate files

Inspecting the database:
The following django commands were used to navigate the db:
sqlite3 db.sqlite3 (to access the environment).
.tables (to list the tables)
select * from TABLE_NAME; (to view data in a particular table)
.quit (to exit the environment)
A sqliterc file was created to configure headers and columns for neat output, which was useful when viewing the various tables and the data within. Using the commands to navigate the database was useful during development to check that the correct data was being entered and to understand how django manipulates the information within the database.

The filter method used within views was used to return data for use in graphs. Some chaining of filters was necessary to extract multiple data fields in json format.

Template logic is used extensively to reveal buttons or other elements depending on user permissions. For example on the blog page detail view

staff users will see an extra edit button allowing them to edit the blog post. Other authenticated users will not see this button. (also see documentation).

Comments are used within the code pages to describe functions and to annotate sections enabling other programmers to follow and understand what is going on.

The history of the development of the the application is represented by an extensive git commit of each new piece of functionality or amendment.