The implementation process began with re-constructing the HTML template by Creative Tim[?] to the different sections mentioned in chapter 4.2, JavaScript going along with the HTML and finally the Django back-end.

0.0.1 HTML

Redesigning the HTML code required changing the CSS, HTML components and removing unwanted JavaScript code. This brought about various bugs and unexpected changes in the overall HTML which crashed a few elements of the code such as Tables. With the use of Google Chrome and Console the debugging was completed and successful HTML pages were created for all three different sections as mentioned in Chapter 4.2. However, since pages are going to be repeated(Dashboard, Viewform, Profile, etc) only one HTML code was required and repeated depending on which panel you are logged in as. Table 1 shows what components were created into their relative files by redesigning the public licensed template. Each of these were combined with CSS and JavaScript from Chapter 4.2.2

Page	**HTML Components**
Login	POST Form(Email, Password)
Register	POST Form(Firstname, Lastname, Email, Password1, Password2, DOB)
Password Change	POST Form(Current Password, Password1, Password2)
Profile Page	POST Form(Email;Disabled, Firstname, Lastname, DOB), TextArea(PublicData)
Dashboard	Statistics, Table(ID, Description, Date, View)
View Form	Text Fields(User Details, Circumstance), Table(Module Details), Table(Uploaded Files), POST Form(Files)
Units	GET Form(Number of modules)
New Form	POST Form(Circumstance, Module Details, Files)
Public Data	TextArea(PublicData;Disabled)

Table 1: HTML Components & Relative Pages

0.0.2 JavaScript

After completion of the HTML design, it was important to add JavaScript in order to allow smooth control of the website. Along side the JavaScript received with the template by Creative Tim[?] such as **Bootstrap**[?] other scripts were manually added.

Tables would load numerous entries for the *Secretary Panel* and the *Scrutiny Panel*, this caused the page to be extensively long without the ability to filter out or search for a specific form, user or find pending forms. In order to achieve efficiency and sort out the entries in the table the use of **DataTables**[?] came in handy. It automatically completed the following;

- Added a search field - Allows the user to search for data in the entries
- Limited entries shown and added more pages - Allows efficiency and speed
- Sort by ascending or descending - Useful when searching for *Pending* forms

The above are a requirement as being able to sort the forms, users and finding exactly something specific is an important factor which can be useful when a form needs to be reviewed or assessed.

Another important public script used was **jsPDF**[?], as per the requirements, the secretary needs to be able to print and download the forms. PDF would be a perfect format to download these files as any browser, or PDF compatible viewer can be used to view the form. This JavaScript code converts the content inside a specific *HTML container* and copies that data to a PDF which can then be downloaded by the user, in our case, the secretary. When the implementation of this was complete, the data from the form would be saved simply as text and would loose its colours and format. This would scramble up all the data and make it highly unreadable. To overcome this bug, **Html2Canvas** was used.

Html2Canvas[?] is a MIT licensed script which creates an image for everything on the current web page or a specific HTML container with the CSS (including colour and design). This is then copied onto the PDF created by jsPDF as mentioned above and made available for download. This allows the form to be printed exactly the same way it would be seen when viewing it on the web-page.

Lastly, the secretary can also directly print the form while viewing it. All modern browsers have the ability to print directly what is being displayed on the current web page and with a simple line of code(*window.print()*), a button on the view form html allows the browser to detect a print command being sent.

0.0.3 Django

The main part of the implementation is combining both the **HTML** and **JavaScript** to work along with **Django** being the main controller. In the implementation, Django 2.2[?] which is the latest version was used. The first step was simply to combine the HTML, JavaScript, CSS and Django together into a folder system and define it on Django's settings. Within seconds the entire HTML was accessible on a Django running web server. The three sections below explain the process of coding in Django, the difficulties faces and lastly a small summary on back-end development.

Authentication & Database

After combining all the above, the authentication system was first to be built. As stated in Chapter 4.4, **All-Auth**[?] was implemented into the Django system by importing the relevant packages. However, in order to avoid users creating multiple accounts with multiple emails, a manual **validation** was coded which only allows the student to register with their University Email Address (**sheffield.ac.uk**) only. All HTML files for All-Auth were overwritten by the HTML design already created in order to have a continuous site. At this point, all data being registered from All-Auth was bring stored into a **SQLite database** which was inbuilt with Django. Creating the database schema seen in **??** was extremely simple with the **Model, View and Controller system** which Django follows. The database was ready in seconds after defining it in Models.

New Form

At this point, the authentication system was ready as well as the database in which all the data would be stored. In order to retrieve data from the database there needed to be some data already inserted and so in order to do that the "Create new form" page had to be completed with back-end code. Different students may have different number of modules which are affected by their circumstance and so the first step would be to ask them to fill in how many modules are affected by their circumstance. This would send in a GET request with the parameter '**units**' and its **value** as an integer. The form for creating a new extenuating circumstance would then get this integer from the GET link and produce those many 'input fields' for the modules section of the form. The creation of a HTML POST form is made extremely easy with Django's **forms.py** system where you simply define the fields of a form such as text input or email and it will automatically present the form on the template where it is called. On successful submit, the student is notified, redirected to dashboard and the form data is saved into the database. The student is also notified if the form was not submitted fully and if it should be re-submitted.

Dashboard

Student: Once the form has been submitted, the student can see it on the list of forms submitted. The dashboard provides statistics based on the current status of the all the forms submitted. These statistics are counted based on the status of each module under each form. If all modules are approved, the form will be '**entirely approved**' and this will be visible to the student under the statistics.

The same goes for pending and rejected. However, if the modules are set to different statuses such as 'rejected' and 'approved' under one form, the overall result is '**partially approved**' or '**partially pending**' if the module has one or more pending modules. The student can see exactly which module has been rejected, approved or still pending. All data is retrieved from the database as follows mathematical calculations where each module is added onto the statistics based on the other modules within the same form. The calculations as well as the display of each form was completed easily with the use of **FOR** loops.

Secretary & Scrutiny Panel: If the secretary or someone from the scrutiny panel logs into the system, the dashboard follows the same format as that of the students but with access to all forms submitted by students and their public data. The statistics of the forms are also similar to that of the students allowing the secretary or the scrutiny panel to know exactly how many pending forms are there which need to be updated.

View Form

Student: The dashboard links each of the forms created to a view form page. This page retrieves data from the database based on the GET parameters(form ID) received when requesting the page. Interlinked data from the database is then retrieved based on the **Form ID** such as the *User* who created it, the *Files* uploaded under the form and lastly the *Modules* affected. Each of these is stored in different tables as they share a one to many relationship. Apart from simply displaying the form data already submitted, there is an option triggered by the secretary which creates a new **file up-loader** at the top of the page allowing the student to upload more files. The file up-loader submits the new files as a POST form and saves based on the form ID.

Secretary Panel: This page follows the same format as the student view form with an exception of allowing the secretary have control over editing the final status and action for each module and requesting more files. Each module is displayed in a table with individual buttons bringing up a '**pop-up**' also known as '**Modal**' in HTML5. The modal pop-ups then allow the secretary to change values for the 'Action' and 'Status' of the form. **HTML drop-down** allows us to display text as the displayed options while the back-end gets a '**value**' which is stored into the database as an integer. This same value is automatically selected when displaying the current selection as it retrieves the integer from the database. A **switch-case toggle** was designed to allow the secretary to request more files from the student. When the switch is toggled, the database value for '**requestfiles**' is switched. This then displays the additional files up-loader as stated previously. The switch is called by a POST submit button with a '**name**' attribute to differentiate between the different POST forms. Lastly, for ease of access, a 'textarea' was created allowing direct change of the students public data without having to open multiple tabs and make changes. This again uses a POST form with a different 'name' attribute. Django '**views.py**' controls all the data received from the POST request and cleans the data ready to be saved into the database with the '**save()**' function.

Scrutiny Panel: Since the scrutiny panel does not control the status directly on the portal, the view form page simply retrieves data from the database and displayed it with the use of HTML components such as p, a, h1, h2, h3, label, and many others.

Profile Page

The profile page simply displays the current users information. If the user is a student they can view their public data in a **disabled** '**textarea**', this allows scrolling of long paragraphs without having to extend the page size. Users cannot change their email address as this would bring about issues of multiple sign ups with different emails hence the **validation of only sheffield.ac.uk emails**. A simple POST form on the page retrieves the values current in the database and allows changing them after cleaning the new data. **This includes: firstname; lastname; dob only.**

Student Record

Every user when registered is provided with a **8 digit random key** which can is used as a **GET parameter** to access their public record. Only users who have the **KEY** can access these records making it secure. This random key is stored in the database under '**User Profile**' and has a **unique schema** which makes sure that two users do not have the same KEY. The public data again is simply displayed with a '**textarea**' HTML component which is coded as mobile first. The purpose of using a random key is to **avoid pattern detection**(For example, using an email instead of a random key) and **unauthorised access.**

Secretary Panel: Since only the secretary can give out the direct link to access the student's record, they also have access to **regenerating the key**. This is a **safety feature** to avoid unauthorised access. The regeneration uses the same principle of using **Python's Random package** to generate an **8 digit key** and replace it in the database.

Email Notifications

Email notification were created with the use of **Gmail's SMTP services**. Django has an inbuilt system which makes mailing and bulk mailing extremely easy. The SMTP details of the sender email can be set in the settings.py page which directly links to the **Django mailer** and with a simple *sendMail* function we can send mail. To make debugging and later changes to the email text being sent, each message is stored as a TXT file which is called in the *sendMail* function when required. For example, when a new user registers, a confirmation email is sent to the user using the *sendMail* function and uses the data from **NewUser.txt** as the main message. Email notifications have been implemented for the following instances;

- New user registration
- Password reset and confirmation of password change
- Successful creation of extenuating circumstance form
  - Email is sent to both Student and Secretary
- Updated status of the form (Rejected/Approved)
- Requested additional files
  - Email is sent to the Student and once uploaded to the Secretary