Meeting Functional Requirements of the Solution

|  |  |
| --- | --- |
| Criteria | Met |
| Account Registration & Login | See flaskr\app.py Registration and Login Route |
| User Roles & Permissions | See flaskr\app.py Admin Dashboard Route and Dashboard Route |
| Educational Content | See flaskr\templates\infoPages\ |
| Energy Tracker | flaskr\templates\tracker.html and tracker flask route |
| Schedule Consultations | See flaskr\templates\personConsultation.html, flaskr\templates\solarConsultation.html, and correlating flask routes |
| Manage Bookings | See flaskr\app.py Admin Dashboard Route and Dashboard Route |
| Admin & Technician Features | See flaskr\app.py Admin Dashboard |
| Installation Scheduling | See flaskr\app.py installation Route, And flaskr\templates\installation.html |
| Carbon Footprint Calculator | See flaskr\app.py cfc Route, And flaskr\templates\cfpCalculater.html, flaskr\templates\cfp\_calculator\_submit.html |
| Accessibility Features | See flaskr\static\js\script.js |

Functionality

|  |  |
| --- | --- |
| Criteria | Met |
| Flexible data handling | One of the Simplest show cases of Flexible Data handling is flaskr/app.py, particularly by generating a secure example password when a user navigates to the registration page using a GET request. How ever this concept is a backbone of the program and is used more advanced in other routes |
| User account management | User Account Management has been implemented fully with the use of flask Sessons to securely log in users, I have also developed functionality for role-based access, Seen in flaskr\auth.py and flaskr\app.py |
| Neatly organised code | The Best Example Of Neatly Organised Code is in flaskr\auth.py and flaskr\validation.py Where Functions have been clearly laid out with comments and docstrings |
| Comprehensive APIs | flaskr\consultation.py   * Geoapify Geocoding API: (https://api.geoapify.com/v1/geocode/search) used in the address\_coordinates function. * Geoapify Place Details API: (https://api.geoapify.com/v2/place-details) used in the get\_building\_geometry function.   flaskr\tracker.py   * Google Gemini API: Used in the gemini\_format function to process and structure the extracted text data. The genai.Client is initialized and client.models.generate\_content is called. * OCR.space API: Used in the ocr\_process\_file function to extract text from image files. A POST request is made to https://api.ocr.space/parse/image. |
| Track data | The tracker\_upload\_file route in app.py demonstrates flexible data handling when processing uploaded energy bills. It uses OCR and Gemini AI to extract structured data from various bill formats, accommodating different layouts and information. |
| Interconnected modules | The best example of Interconnected modules is in flaskr\consultation.py where multiple modules call other modules, an example of this is solar\_potential where it calls get\_building\_geometry, calculate\_area\_shapely and calculate\_orientation |
| Optimised algorithms | Calculate\_orientation is an example of optimised algorithms seen in flaskr\consultation.py, the function is Clear and concise, Efficient calculations, Direct compass bracket assignment and Error handling to be as quick as possible |
| Normalised data | The best showcase of normalised data is in flaskr\db.py where get\_user\_energy\_data retrieves energy bill data from the database, processes it, and structures it into a normalized format suitable for generating charts and displaying information. |

Code Organisation

|  |  |
| --- | --- |
| Criteria | Met |
| Clear indentation and structure | Seen All Throughout the codebase |
| Organised into functions/classes | Seen All Throughout the codebase |
| Documented with comments | Seen All Throughout the codebase |
| Efficient use of variables/constants | Seen All Throughout the codebase |
| OOP design features and Object-oriented programming principles | flaskr\templates\installation.html uses JavaScript to implement a custom dropdown. The logic for handling the dropdown is encapsulated within a JavaScript class. |
| Recursive algorithms with stopping conditions | Not Needed |
| Global variables minimised | Only Global Variables used are required for library to function correctly, and do not hold risks of the program security such as the logger setup, what needs to be global to allow logging in all functions |
| Avoid nested if clauses | No Nested if clauses are used |
| Named meaningfully | Seen All Throughout the codebase |
| Independent logic pieces | Seen All Throughout the codebase |
| Zero unnecessary repeated code | Seen All Throughout the codebase |
| Appropriate interrelation of parts | Seen All Throughout the codebase, Such as UI and Backend Integration, Component Interaction within UI, Conditional Logic, Validation Layers and Data Flow |
| Top-level consistency | Seen All Throughout the codebase, Via; Shared Base  Template, Centralized CSS, Shared JavaScript, Consistent User Feedback, Standardized Backend Patterns |

|  |  |
| --- | --- |
| Interfacing front-end and back-end efficiently | Seen All Throughout the codebase |
| Neat and clear Organisation | Seen All Throughout the codebase |

User Experience

|  |  |
| --- | --- |
| Criteria | Met |
| User-friendly interface | Seen All Throughout the codebase, Such as assesbility mode, Consistent User Feedback with flask flash, example passwords, Screen Reader Support, Cross Browser Support, Mobile Support |
| Excellent error handling and messages | Seen all Throughout the codebase, Using Python Logger and Flask Flash and flask error handlers, to catch and explain every error, And Try and Else Statements in almost every function catching the errors to display This can been seen specifically in the installation routes where specific errors like value error are caught also see add\_installation\_request in flaskr/db.py |
| Personalised feedback | Flask Flash is used to give feedback to the users, The best example of this is in the register route where feedback is given to the user, with specific information of what went wrong |
| Ease of navigation and input | Navigation Bar: A main navigation bar is set up in base.html and appears on every page to help users move between sections easily.  Content Cycling on Info Pages: Pages like greenEnergy.html use JavaScript to change the content shown when users click an arrow icon. The content comes from a list of sections stored in a JavaScript array.  Standard Forms: There are basic forms for login, registration, and consultations. These use standard HTML <input> elements (e.g., for text, email, passwords, phone numbers, dates, and times). The look is styled with style.css to be clean, with clear borders, helpful placeholders, and visual effects when fields are selected.  Address Autocomplete: On the Solar Consultation page, users can type their postcode, and JavaScript fetches address suggestions using the GetAddress.io API. These suggestions are shown in real time, and selecting one fills out the form automatically. A debounce function is used to reduce the number of API calls.  Conditional Form Fields: On the Installation page, some form fields only appear depending on what product the user selects (e.g., solar panels or EV chargers). This is controlled by JavaScript.  File Upload: The Energy Tracker page includes a file upload feature. The standard file input is hidden, and a styled button is used instead to make it look nicer. When a file is selected, JavaScript shows the file name to the user. |
| Robust accessibility features | Accessibility Features have primarly been implemented in the toggleAccessibilityMode function within script.js, Where by clicking the toggle button, on the nav bar. The page will automatically be loaded into the accessibility mode, automatically on every page till its toggled off.  When enabled, the accessibility mode:  Adds an accessibility-mode class to the <body>.  Disables existing external stylesheets and inline <style> blocks.  Saves and removes most inline style attributes, preserving some for elements like charts.  Injects a new <style> block (#accessibility-styles) with high-contrast settings, increased font size, simplified layout rules, and uses the 'OpenDyslexic' font.  Hides images (<img>) and displays their alt text content within a styled <span> (.alt-text).  Adds descriptive text (.chart-description) after <canvas> elements identified as charts.  Hides certain complex visual elements like .chart-container and .progress-container.  Announces the mode change ("Accessibility mode enabled/disabled") to screen readers using a temporary ARIA live region. |
| Effective validation and sanitisation | Seen All Throughout the codebase, A good example is in flaskr\tracker.py what hold most of the validation for input fields, Done Quickly with regex patterns ( a industry standard ) A Good example of Sanitisation is SQL Injection Prevention, done in db.py on every function what use parameterized queries |
| Consistent branding and aesthetics | Using A Base.html file and a external style sheet, |
| Efficient data visualisation | Base Template Inheritance: Most pages extend base.html, which provides a consistent header, navigation structure, and includes the main stylesheet, Central Stylesheet: The file style.css defines common styles, Reusable CSS Classes: Utility classes like .nixie-font, .hubballi-font, and color classes are applied across different templates, Consistent Component Styling, CSS Variables: Some templates, like personConsultation.html, use CSS variables (:root) to define and reuse specific color schemes within that section. |

Legal and Regulatory Guidelines

|  |  |
| --- | --- |
| Criteria | Met |
| Legal compliance (e.g. GDPR) | Cookie Consent pop-up, Policy/Terms of Service, Explicit Consent, And Security measures like Password hashing can all be found |
| Ethical considerations | Data Privacy and Security, All Data Stored is disclosed to the user and private information like passwords are hashed  As well some Extra considerations have been taken:   * An accessibility mode toggle is implemented in script.js and present in base.html. * Accuracy and Reliability * The accuracy of calculations (See test Log ) * Transparency and Consent |
| Accessibility features | See The “Robust accessibility features” Row above |
| Logically selected fonts and colours | All Colours have been selected to meet WCAGs guidelines, and all fonts are readable with suitable letter spacing, as well as this the accessibility mode removes Colours and using the open dyslexic font |
| System compatibility | The website is useable on all systems, browsers and devices |
| Web standards | Website Designed to meet WCAG |
| Privacy and data policy | Found on the footer of the homepage and dashboard, like most websites, allowing users to quickly find |
| License and intellectual property | See Asset Log |

|  |  |
| --- | --- |
| Terms and conditions | Found on the footer of the homepage and dashboard, like most websites, allowing users to quickly find |
| Multi-platform support | Yes |
| Non-discriminatory design | Yes, With Equity in mind |
| Efficient fallback code | Yes, all errors will be caught in production, using error handlers @app.errorhandler(Exception) means any errors will redirect to home page, not leaving the user on a error screen |

Suitability of Test Data

|  |  |
| --- | --- |
| Criteria | Met |
| Normal data | Done On Every Input Field See example bellow: Example: Test Case 2: User Registration - Normal Data   * Input: Email: user@example.com, Password: P@ssw0rd1234, Password2: P@ssw0rd1234 * Result: User registered successfully (Status: Pass). This tests the system with standard, valid inputs. |
| Erroneous data | Done On Every Input Field See example bellow:  Example: Test Case 2: User Registration - Erroneous Data   * Input: Email: user@example.com, Password: Password, Password2: Password2 * Result: Initially failed to show a user-facing error (Status: Fail), demonstrating testing with invalid data (passwords don't match) leading to a required fix (Configure Flask Flash). |
| Extreme data | Done On Every Input Field See example bellow:  Example: Test Case 2: User Registration - Boundary Data   * Input: Email: user@example.com, Password: P@ssw0rd123, Password2: P@ssw0rd123 * Result: User registered successfully (Status: Pass). This tests the system with data at the edge of defined requirements (e.g., minimum password length). |

Use of Testing to Inform Iterative Development

|  |  |
| --- | --- |
| Criteria | Met |
| Testing of inputs | Extensive testing (approx. 67 tests) focused on user interactions with input fields, buttons, dropdowns, and form submissions, ensuring the UI behaved as expected and captured data correctly. |
| Testing of calculations | Specific tests (7 recorded) were performed to verify the accuracy of backend calculations, such as carbon footprint totals and solar potential estimations, based on defined inputs. |
| Testing of validation | Numerous tests (approx. 28) specifically checked both client-side and server-side validation rules, including required fields, data formats (email, phone, date, time), and logical constraints (e.g., booking times, past dates). |
| Testing of processes | End-to-end user workflows like registration, login, submitting calculations, booking consultations, and retrieving data were tested (approx. 13 tests) to ensure the steps flowed correctly and integrated properly. |
| Evidence of iteration | The log provides significant evidence (33 instances) of a test-fail-fix-retest cycle. Failures were documented, specific fixes were identified and implemented (e.g., adding routes, fixing database queries, adjusting validation, correcting UI elements, improving OCR/AI), and subsequent tests often confirmed the resolution, demonstrating how testing directly informed improvements.  As well as this a grand total of 119 tests were done. |

Quality of Iterative Development Process

|  |  |
| --- | --- |
| Criteria | Met |
| Records of changes | See Change Log and the Task 2 Versions |
| Rationale for changes | See Change Log |
| Versioning method used | Git with a local repository, See the hidden git folder, and the CHANGELOG.md |