Architectural Requirements

Debuggers Anonymous

Table of Contents

Design strategy

Our chosen architectural design strategy is **decomposition**. We have opted for this strategy over:

- Quality requirement driven design
- Generation of test cases.

The reason behind this decision is that our project comprises numerous parts and technologies, making an architectural design that breaks it down into smaller components highly advantageous for development and testing purposes.

Decomposition offers several benefits that align with the needs of our project. Firstly, it promotes **modularity** by breaking the system into manageable components. This modularity allows for independent development and testing of individual parts, enabling parallel work and faster progress. Developers can focus on specific components without being overwhelmed by the complexity of the entire system.

Additionally, decomposition enhances **reusability**. By dividing the project into smaller parts, components can be reused in different contexts or future projects. This not only saves development time but also ensures consistency and reduces the likelihood of errors.

Furthermore, the decomposition strategy facilitates **maintenance and troubleshooting**. When an issue arises, it is easier to identify and isolate the problematic component, making debugging more efficient. Modifications and enhancements can be made to specific parts without affecting the entire system, reducing the risk of unintended consequences.

Lastly, decomposition supports **scalability**. As the project grows or user demands change, it is easier to scale specific components rather than scaling the entire system. This flexibility allows for the efficient allocation of resources and the ability to adapt to changing requirements.

Considering the complexity of our project and the need for efficient development and testing, the decomposition strategy emerged as the most suitable choice. By breaking down the system into smaller parts, we can leverage the benefits of modularity, reusability, maintainability, and scalability, ultimately leading to a more robust and manageable architecture.

Architectural Styles

1. Client-Server Style

We have adopted the Client-Server style to divide our project into two distinct parts:

1. Client-facing interface
2. Backend operations handled by the server.

This separation ensures that modifications made to the backend do not impact the frontend, providing ease in development and testing. By splitting the project in this manner, we achieve a clear separation of concerns and facilitate independent updates and enhancements in each component.

The Client-Server style enables **efficient development and testing** by allowing separate teams or individuals to work on the frontend and backend concurrently. Changes made to the backend, such as database updates or business logic modifications, can be implemented without affecting the frontend interface. This decoupling simplifies the development process and reduces the likelihood of unintended consequences or compatibility issues.

Furthermore, the division between client and server makes **scaling** the system more manageable. As user demands increase, resources can be allocated to the server side independently of the frontend. This scalability ensures that the system can efficiently handle a growing user base or increased workload without sacrificing performance or user experience.

2. Structural Style

We have incorporated the Structural style into our architecture to effectively organize and structure our backend. By employing the Structural style, particularly through a **layered architectural design**, we establish a solid framework for managing the system's components and their interactions.

The Structural style enables us to divide the system into multiple layers, each assigned specific responsibilities and functions. This layered approach facilitates a clear separation of concerns, simplifying the comprehension, development, and maintenance of the system. These layers typically include components such as the presentation layer, business logic layer, and data access layer.

Implementing the Structural style offers several notable advantages for our project. Firstly, it promotes **modularity** by encapsulating related functionalities within each layer. This modular design enables independent development and testing of each layer, facilitating parallel work and accelerating overall progress. Additionally, the inherent modularity enhances **reusability**, as components within a layer can be leveraged in different contexts or future projects.

The adoption of the Structural style also **streamlines system maintenance and troubleshooting** efforts. Each layer represents a distinct part of the system, facilitating the identification and isolation of issues when they arise. Modifications or updates can be targeted to a specific layer without impacting the entire system, thereby reducing the risk of unintended side effects.

Overall, the Structural style provides a robust foundation for our architecture, enabling effective organisation, modularity, reusability, and ease of maintenance.

3. Event Driven Style

We have incorporated the Event Driven style, specifically utilising the **Broker architectural design**, as an integral part of our project's architecture. This style facilitates communication and coordination among various system components by employing a publish-subscribe mechanism where components can publish events or messages without having direct knowledge of other components. These events are then routed and delivered to interested subscribers by a central Broker component or an event-driven system.

The Event Driven style offers several advantages such as promoting **loose coupling** between components. Each component can operate independently, reacting to events that are relevant to its functionality. This decoupling allows for flexibility, as components can be added, modified, or replaced without impacting the entire system.

Additionally, the style enhances **scalability**. The publish-subscribe mechanism enables horizontal scaling by distributing events across multiple subscribers or event handlers. This distributed nature facilitates handling a large volume of events and allows the system to adapt to changing demands or increased workloads.

Furthermore, the style supports extensibility and modularity. New components can be easily integrated into the system by subscribing to relevant events. This modularity allows for flexible system composition and encourages the reuse of existing components.

The style also enhances system **reliability and fault tolerance**. Components can react to events asynchronously, reducing the risk of cascading failures and improving system resilience. The decoupled nature of event-driven systems enables components to gracefully handle failures and recover from errors.

Overall, the adoption of the Event Driven style, along with the Broker architectural design, ensures effective event-based communication, promotes loose coupling, scalability, extensibility, and enhances system reliability.

Quality Requirements

1. Correctness

The accuracy of the system in predicting the amount of sunlight that hits the client's house, specifically the solar radiation on their property, is a critical factor. It is expected that the system achieves **a minimum accuracy of 80%**. This level of accuracy is essential because the primary goal of our project is to provide users with reliable information regarding the solar potential of their property. If the system cannot accurately calculate the amount of sunlight, it fails to deliver any value to the customer.

2. Responsiveness

To enhance user experience, the system should be designed to minimise calculation times and ensure responsiveness. Users should **not have to wait for calculations longer than five seconds**. It is crucial to optimise the system's performance to allow customers to quickly access the necessary information they seek. This is particularly important for basic calculations, where reducing the calculation time becomes even more critical.

3. Customisability

The system should offer users the ability to customise their calculations by **adjusting at least five parameters**. Users who recognise the solar potential of their property should have the option to further tailor the calculation to their specific needs. This could involve incorporating additional factors such as specific home appliances and solar products into the calculation. By enabling customisation, our project aims to provide users with a more personalised and accurate assessment of their solar potential.

4. Scalability

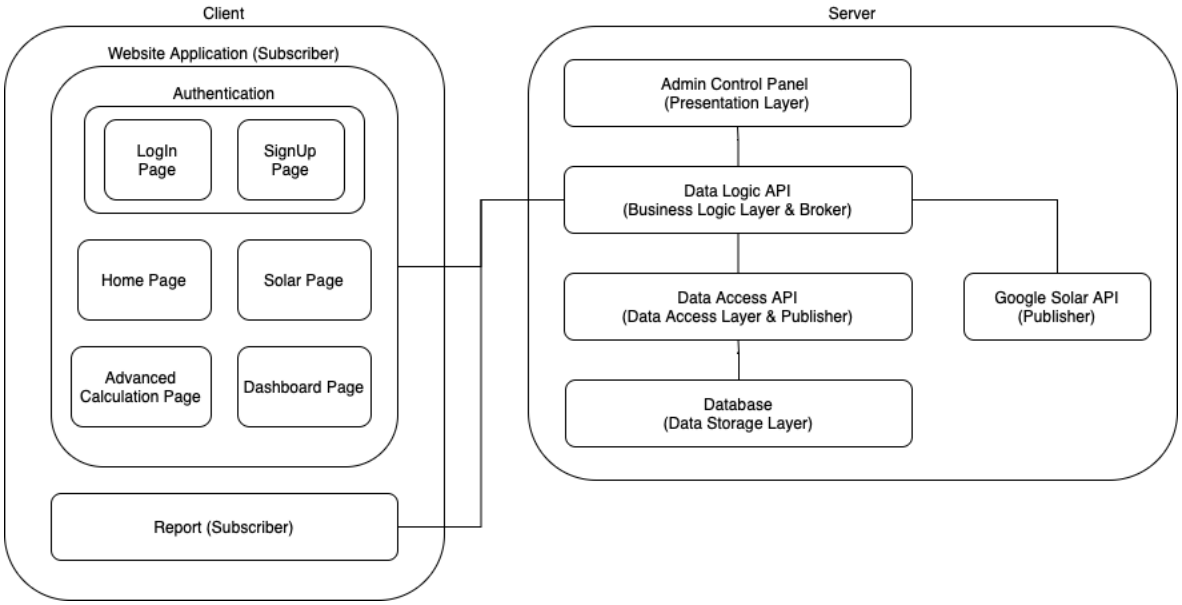
The system must be designed to handle a significant volume of users, with a target capacity of **accommodating ten people per second**. As the project holds value for multiple clients, it is vital to ensure that the system can effectively scale to meet the demands of potential customers. By considering scalability during the system's design phase, we can ensure its ability to serve a growing user base without compromising performance or user experience.

5. Availability

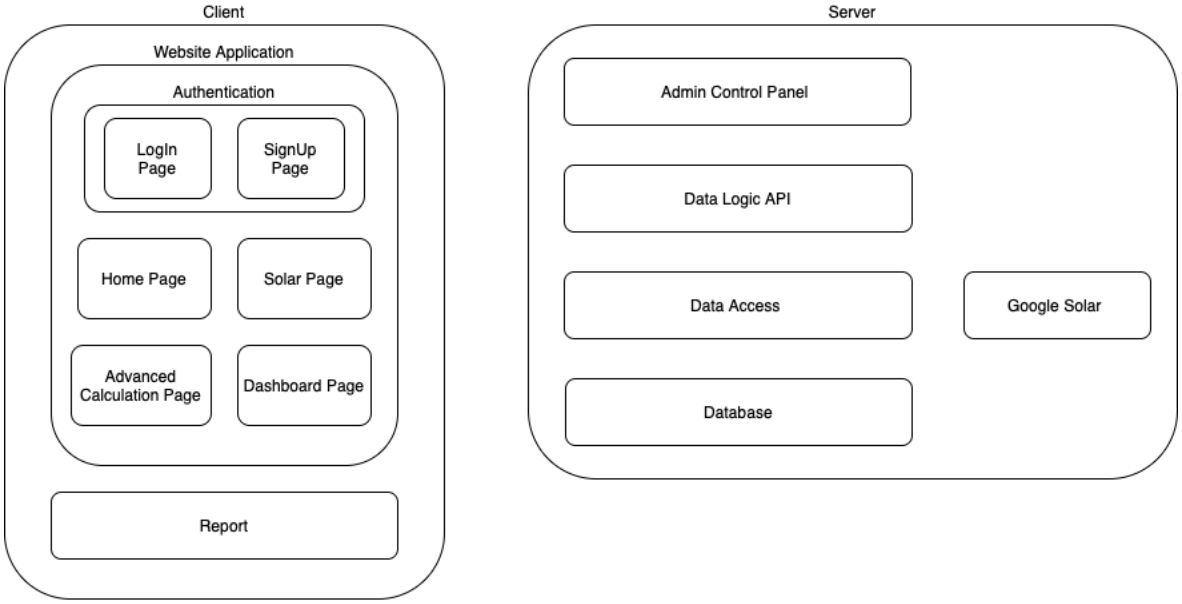
To reach a wide range of users, the system must be **accessible across multiple devices, including computers, tablets, and phones**. The website should be optimised and responsive, allowing users to access and interact with the system seamlessly regardless of the device they are using. Recognising that not all clients will be accessing the system through laptops, it is crucial to make the website available and functional on various devices, accommodating the preferences and needs of different users.

Architectural Design and Pattern

Overview



1. Client-Server pattern



This architectural pattern provides a clear separation between the Client side and the Server side of the project. The Client side encompasses all the pages and systems that the client directly

interacts with, such as the Home Page, Solar Page, and Dashboard Page within the Website application.

On the other hand, the Server side includes various systems that serve the client's needs. This encompasses components like our Data Logic and Data Access API's, the Database and the Admin Control Panel utilised in the project. By distinguishing these two sides, we gain several advantages that contribute to the overall effectiveness and efficiency of our project.

Advantages of the Client side:

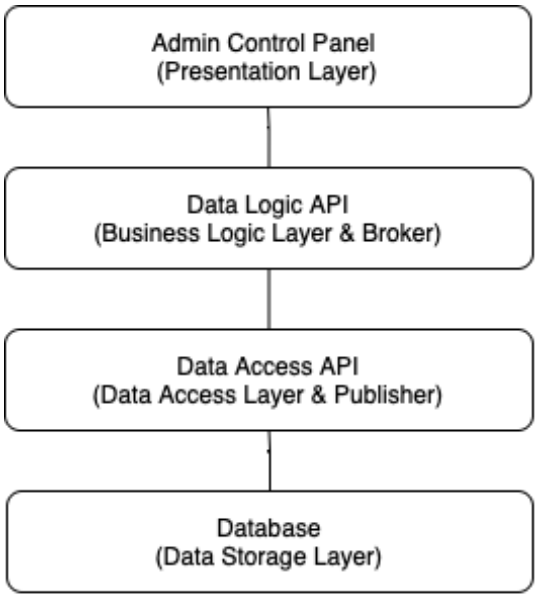
- Responsive and Interactive Features: By concentrating on the Client side, we can prioritise the implementation of responsive design principles, enabling the application to adapt seamlessly to different screen sizes and devices. This ensures that users can access the project from various platforms and enjoy an optimal viewing experience.
- Enhanced Performance: By optimising the Client side components, we can deliver a fast and responsive user interface. This includes efficient client-side rendering, minimising page load times, and optimising the use of resources. Improving performance contributes to a smooth and enjoyable user experience.

Advantages of the Server side:

- Scalability: By focusing on the Server side, we can design our systems to handle increased loads and user demands. Scaling up the Server side components, such as the API, and the database, allows our project to accommodate a growing user base without sacrificing performance or functionality.
- Bug Identification and Maintenance: Separating the Server side components enables easier identification and debugging of issues. By having distinct systems responsible for serving clients, it becomes simpler to pinpoint and resolve any bugs or errors that may arise. Additionally, maintenance and updates to the Server side systems can be performed independently, minimising the impact on the Client side.
- Flexibility in System Updates: With a separate Server side, we can make changes and updates to the backend systems without affecting the Client side interface. This flexibility allows for smoother updates, enhancements, and the integration of new features, ensuring the project remains adaptable to evolving requirements and technologies.

By adopting this Client-Server architectural pattern, we can leverage the advantages of both the Client side and the Server side. This separation allows for focused development, improved user experience, scalability, easier bug identification, maintenance flexibility, and the ability to adapt to future needs.

2. Layered



Our layered architecture is designed to streamline the flow of data from the database to the presentation layer in the Admin Control Panel and vice versa. This architectural approach breaks down the entire process into distinct layers, providing numerous advantages in terms of bug identification, scalability, and maintainability.

The Admin Control Panel serves as the **Presentation layer** in our architecture. It is responsible for presenting relevant data to the admin user and facilitating any necessary changes. By separating the presentation layer, we achieve a clear distinction between user interface elements and the underlying data processing, making it easier to identify and troubleshoot any bugs or issues specific to this layer.

The Data Logic API acts as the **Business Logic Layer**, where requests from the Admin Control Panel are processed. This layer contains the core logic of our application, handling business rules, validations, and orchestrating the appropriate actions. It interacts with the Data Access API, which serves as the **Data Access Layer**, responsible for handling all data requests. The API communicates with the Database, our **Data storage layer**, where the actual data is stored.

By adopting a layered architecture, we gain several advantages:

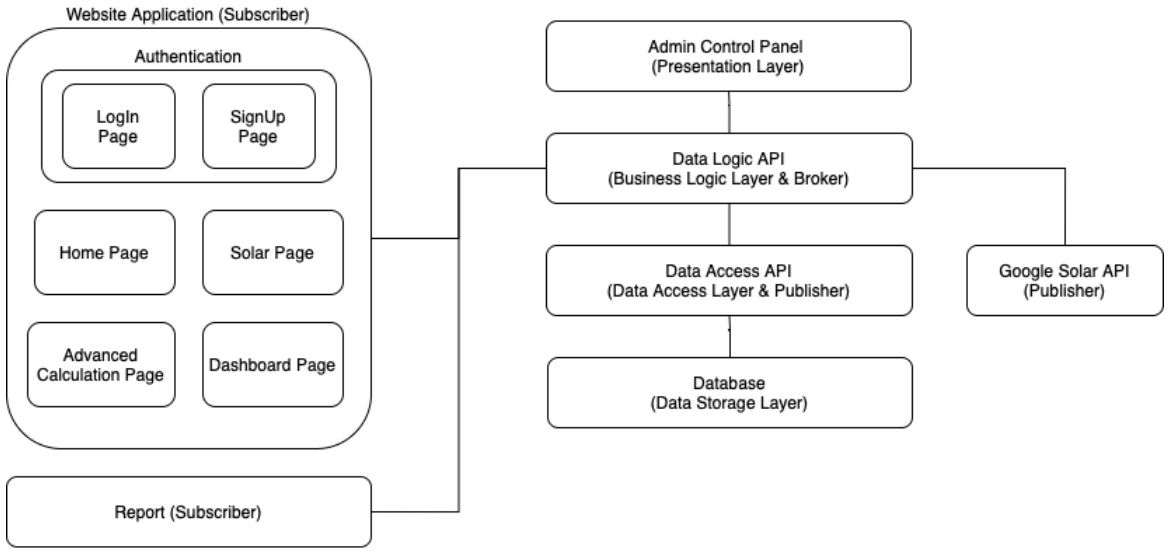
1. **Modular and Separation of Concerns**: The layered approach promotes modular design and separation of concerns. Each layer has specific responsibilities and functionalities, making the codebase more organised and maintainable. Changes and updates can be implemented in a

specific layer without impacting the entire system, improving agility and reducing the risk of unintended consequences.

2. Bug Isolation and Debugging: With the clear separation of layers, identifying and resolving bugs becomes more efficient. Issues can be isolated to a specific layer, allowing developers to focus on the affected layer without unnecessary complexity from other parts of the system. This accelerates bug identification, diagnosis, and resolution.
3. Scalability and Flexibility: Layered architecture supports scalability by enabling individual layers to scale independently. For example, if the application experiences increased demand on the Data Access Layer, it can be scaled separately from other layers to handle the load. This flexibility allows for efficient resource allocation and optimal performance.
4. Reusability and Interchangeability: Layers can be designed to be reusable and interchangeable. The separation of concerns facilitates the reuse of components across different projects or integration with external systems. For instance, the Business Logic Layer and Data Access Layer can be utilised in other applications with minimal modification, promoting code reusability and reducing development effort.
5. Testability: The layered architecture enables easier testing as each layer can be tested independently. Unit tests and integration tests can be designed specifically for each layer, ensuring the reliability and functionality of individual components before integrating them into the overall system.

In summary, our layered architecture offers advantages such as modular design, bug isolation, scalability, flexibility, reusability, and improved testability. By dividing the process into distinct layers, we enhance the maintainability, scalability, and reliability of our application while providing a clear structure for development, debugging, and future enhancements.

3. Broker



The Broker architecture pattern plays a crucial role in facilitating the exchange of data between our database, and the client. It serves as the central connection point through which data flows seamlessly. By adopting the Broker pattern, we can achieve efficient data communication and streamline the development process.

In our architecture, the Website Application and Report components act as **subscribers**, specifying the specific data they require. The Data Logic API, functioning as the **Broker**, handles these requests and coordinates the retrieval of the relevant data from different sources. The Data Access API serves as **publishers**, responsible for providing the necessary data back to the Broker.

The use of the Broker pattern offers several advantages:

1. **Centralised Data Flow**: The Broker pattern provides a centralised and well-defined mechanism for data exchange. It establishes a clear flow of data between the database, and the client, simplifying the development process by establishing a central point where data is called and managed.
2. **Decoupling of Components**: The Broker pattern promotes loose coupling between components. The subscribers (Website Application and Report) do not need to be aware of the data sources or the details of data retrieval. They simply subscribe to the required data, and the Broker handles the retrieval and delivery. This decoupling enhances modularity and flexibility, making it easier to modify or add new data sources without affecting the subscribers.
3. **Scalability and Extensibility**: With the Broker pattern, our architecture can easily scale to accommodate additional subscribers, publishers, or data sources. The decoupled nature of the pattern allows for independent scaling of the components, ensuring optimal performance and efficient resource utilisation.
4. **Simplified Development and Debugging**: The clear flow of data facilitated by the Broker pattern simplifies the development and debugging process. Developers can easily trace the path of data, making it easier to identify and resolve any issues or inconsistencies. This enhanced visibility aids in troubleshooting and maintaining the integrity of the data flow.
5. **Reusability**: The Broker pattern promotes reusability of components. The Broker itself can be utilised in different scenarios where data mediation and coordination are required. Similarly, the publishers and subscribers can be leveraged in other parts of the system or even in separate projects, enhancing code reuse and reducing development effort.

Overall, the adoption of the Broker architecture pattern in our system enhances the efficiency and reliability of data communication. It centralises the data flow, decouples components, enables scalability and extensibility, simplifies development and debugging, and promotes reusability. This pattern serves as a critical component in our architecture, facilitating effective data exchange and enabling smooth integration between our database, and the client.

Architectural Constraints

The architectural constraints imposed by the client were relatively minimal and did not pose significant limitations on our architecture. The client specified a few requirements:

1. Utilize the ASP .NET Framework.
2. Implement Blazor for the front-end.
3. Host the database on Azure.
4. Backend technology should use JavaScript Runtime.

Thankfully, these requirements seamlessly integrated with our initial architectural design and did not hinder its effectiveness. Instead, they harmoniously complemented our existing structure, allowing for a cohesive and coherent architecture.

Technology choices

Website Application

In response to the client's requirement to use Blazor, we incorporated this technology into our project. Fortunately, Blazor seamlessly integrates with our chosen .NET framework, ensuring a smooth and cohesive architecture.

Database

In compliance with the client's request, we adopted the Azure database for our project. The Azure database is hosted on a free tier, specifically tailored for students, providing ample storage capacity and bandwidth to meet the needs of our project. This selection ensures that our architecture can leverage the scalability, reliability, and robust features provided by Azure's database services.

Backend

For our backend we were free to choose any technology as long as it uses JavaScript runtime, we were given recommendation of using NodeJS, Deno or Bun. We opted to use a NodeJS over the newer and faster Deno or Bun, due to the wide support and thorough documentation that NodeJS has, and while Bun and Deno may be faster the lack of documentation and exposure to different situations is an unnecessary risk of possible road blocks.

Data Access API

Once NodeJS had been chosen for our backend we had the option of NestJS or Express.js to use as an API to our database, we decided to choose Express.js to efficiently and effectively connect to the database and easily integrate with our Azure database. another important reason for choosing Express.js is that Microsoft provides extensive documentation on interacting with an azure database in Express.js

Data Logic API

Given the requirement to utilize the .NET framework, we ended up choosing the .NET API for our Data Logic API. The extensive capabilities and compatibility of .NET with Express, Blazor makes it the perfect candidate for serving as the intermediary or "Broker" in our architecture.

Overview

- Website Application & Subscriber - Blazor
- Business Logic Layer & Broker - .NET API
- Data Access Layer & Publisher - Express API
- Data Storage Layer - Azure SQL Database
- Publisher – Google Solar API
- Backend - NodeJS

Realised Architecture