ABEN HUB
INTEGRATING OPTIMALLY

SOFTWARE ARCHITECTURE PROPOSAL

PREPARED BY

Arsalan Khan

Table of Contents

1. Introduction

" Any intelligent fool can make things bigger and more complex. It takes a touch of genius - and a lot of courage - to move in the opposite direction."
E. F. Schumacher

Software architecture is the general organization of a program and of its components' relationship to one another. It is an understanding of the program that is agreed upon by the software developers.

A good software architecture is important for ensuring that a program can be developed easily, and runs smoothly without issues. Poor or no architecture makes it slower and more expensive to add new features.

In this document I will go through the process of assessing our program and propose an architecture that I believe to be the most suitable for this application.

2. The Program

When developing an architecture for a program, there's a few things that need to be considered first: the non-functional requirements, the functional requirements, and the program flow.

2.1. Non-functional Requirements

The non-functional requirements are all the qualities our program must maintain.

The following are our non-functional requirements:

- Security: user info, most importantly payment info
- Scalability: able to add more features with little effort
- Capacity: able to store lots of user info such as projects, able to store lots of info about different tools, sources of energy, meteorological data, etc.
- Usability: easy to use
- Performance: how fast does the system respond to a user's actions
- Compatibility: needs to be compatible with iOS, Android, and the web.

2.2. Functional Requirements

The functional requirements are all the functionalities our program's users should have access to.

The following are our functional requirements:

- Sign up and log in
    - Choose preferred plan/subscription on sign up
- Create a project
- Receive satellite view of area when address is entered
- Highlight preferred area to be covered by units
- Enter preferred energy requirements (optional)
- Enter preferred budget (optional)

- Results page where you receive best choices for type(s) of units and number of units
- View previous projects
- All this info is stored for a user

2.3. Program Flow

Taking the functional requirements into consideration, we can create a simple diagram to assess the relationship between the user and the system (see fig. 1).

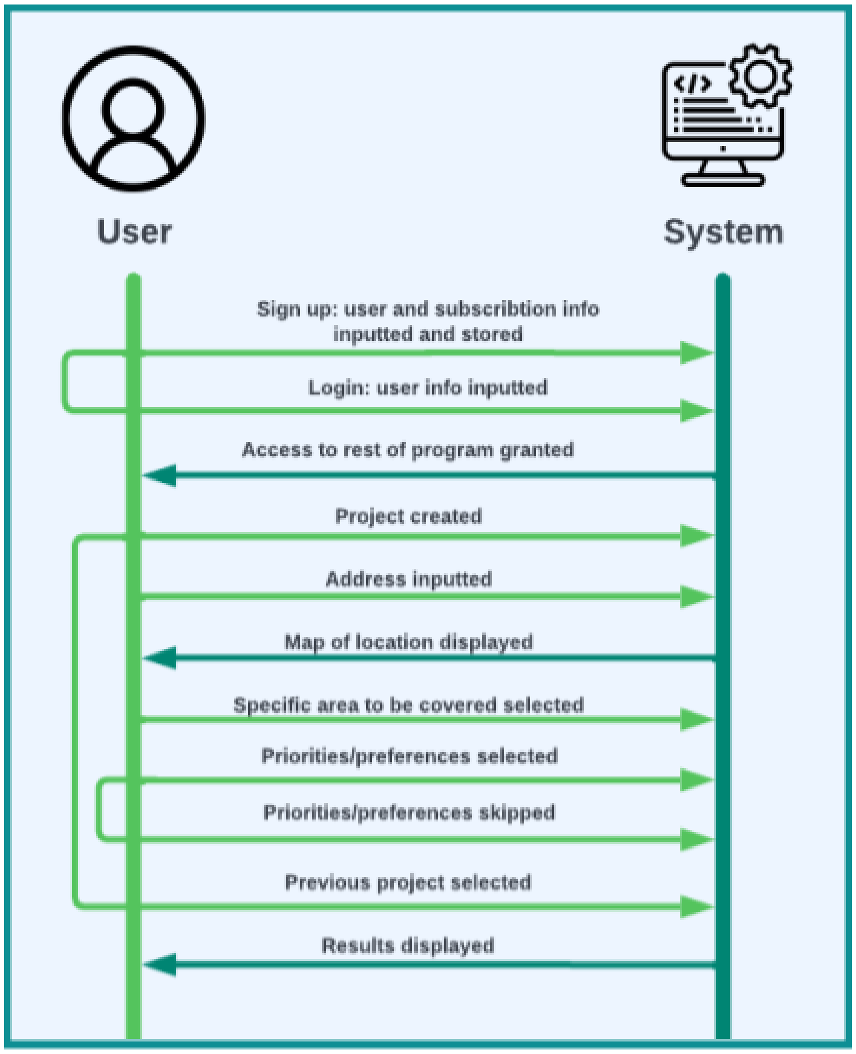
User
System
Sign up: user and subscribtion info inputted and stored
Login: user info inputted
Access to rest of program granted
Project created
Address inputted
Map of location displayed
Specific area to be covered selected
Priorities/preferences selected
Priorities/preferences skipped
Previous project selected
Results displayed

Figure 1: Program flow

Using this diagram and the non-functional requirements we can choose an appropriate architecture for our software.

3. The Architectures

3.1. n-Tiered/ n-Layered Architecture

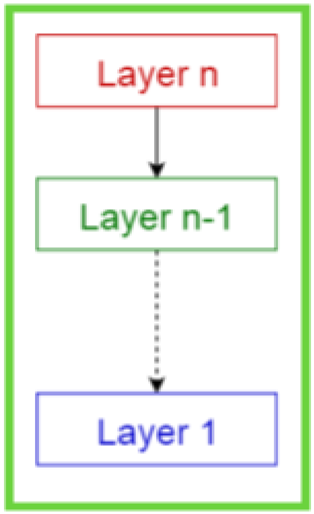

Figure 2: Layered Architecture

In a layered architecture the components are split into multiple layers. Each layer provides services to the next higher layer and each layer is only allowed to use the layer directly below it (see fig. 2).

This organizes the components of the program very nicely and prevents any unnecessary relationships between different components. Because of this simplicity and effectiveness, layered architecture is a very popular option that usually gets the job done for simple projects. I do not expect this to be a small project so I don't suggest using only a layered architecture.

Although we will not be using a strictly layered architecture, I wanted to introduce you to the concept of layering as it can be used within many other architectures where you can introduce layering into the pre-existing components of the architecture.

3.2. Monolithic Architecture

A monolithic architecture is one where all of the program's components and functions are tightly coupled into one large codebase. All the database calls (data interface), logic (business logic) and figuring out how to display that calculated data (presentation) are all mixed in, in a chaotic way in that single codebase.

We can improve the monolith by introducing layers to the codebase so that the different components are more organized. We could have the presentation, logic, and data interface in separate layers. Our architecture would then look like figure 3.

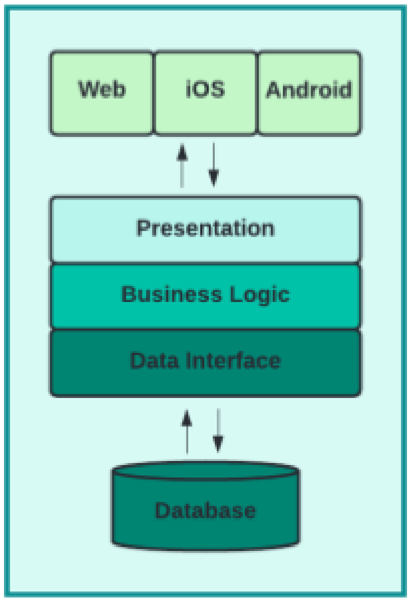

Figure 3: Monolithic Architecture

An advantage of this architecture is that it is very easy to learn and implement compared to other architectures. This is especially advantageous for smaller teams since they only have a few components to worry about. Thus, depending on team size and budget, this could be an option. I don't believe this to be a very strong advantage as the benefits gained from other more complicated architectures could far outweigh the time saved from sticking to this simple architecture.

Another advantage is that because all the components are separated in different layers and have minimal relations to other layers, testing becomes easier since each layer can be tested separately.

A disadvantage of this architecture is that an error in one layer affects the whole program because they all operate together as one unit; removing a layer would make the program incomplete and not functional.

Another disadvantage is that scalability is difficult because once the layers are decided and implemented, it becomes difficult to introduce other layers. Because of this, the specific layers need to be well planned out so that no new layers need to be introduced in the future.

Since all the components are coupled in one codebase, another issue with scalability is that the more features you add, the more complicated the relationships between different parts of the code becomes and the more error prone the program becomes. Because of this, as the program grows, maintenance would become more and more difficult and a single error could cause the entire program to crash.

3.3. Microservices Architecture + API Gateway

The microservices architecture evolved from layered monolithic architectures to solve the difficulties with making changes to and expanding the program. It consists of service components called microservices that represent a single-purpose function or an independent portion of a large application. This means that each microservice consists of code only relevant to one specific functionality.

For our app, some of the microservices we could have are:

- One to get weather data
- One for map data
- One for user functionality (login/sign up, preferences)
- One for projects
- One for info about different modules/tools
- One or more for the different types of calculations to be made

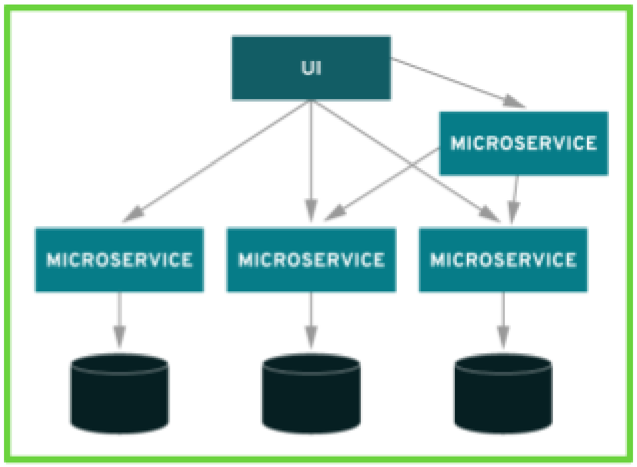

Figure 4: Microservices Architecture

Each microservice only interacts with the microservices that it needs information from. Additionally, rather than having the program connected to a single database, these microservices usually communicate with their own databases. This avoids the issue of coupling from the monolithic architecture where everything is connected together.

One of the biggest advantages of the microservices architecture is that it is very easy to scale. To add a new feature, one simply needs to create a microservice for that feature without having to disrupt the rest of the program.

The independence of each microservice brings many advantages. Because each microservice is independent, a new developer would only need to understand how a specific microservice works rather than knowing how the whole program works as they would in a monolith where everything is interconnected. Similarly, testing would become easier since each microservice could be tested separately (though this is also an advantage of the layers in our monolith). Another advantage of the independence of microservices is that if one microservice fails, the whole program does not fail since the microservice is not connected to everything else; there would still be partial success.

Because in a microservice the program is broken into multiple small components rather than in one whole codebase like in a monolith, one disadvantage is that a monolith runs faster than a microservice architecture.

We could further improve our architecture by introducing something called an API gateway. Like the name suggests, this component acts as a gateway between the UI/clients and the microservices. The API gateway decides which microservice to call upon a client’s request.

With an API gateway, the front end developers do not need to be concerned with the implementation of all of the microservices. Instead, they just need to know how to communicate with the API gateway. One disadvantage is that if the API gateway goes down, the rest of the program goes down. This should not be much of an issue since the API gateway does not contain any complicated calculations or methods.

With the microservices architecture and API gateway, our final architecture would then look something like figure 5.

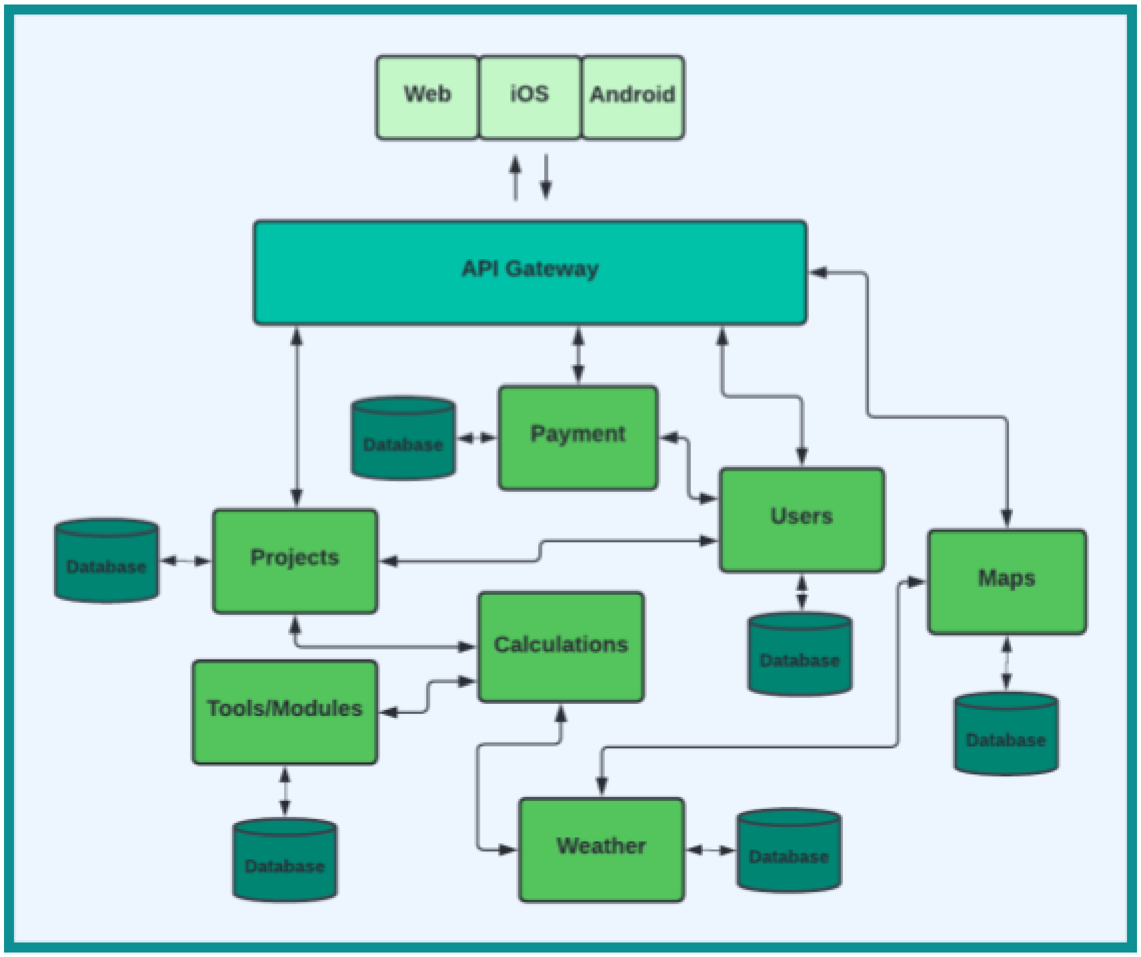

Figure 5: ABen Hub Microservices Architecture

I created this diagram according to my current understanding of the program. I do not know how exactly the calculations function so if possible, it would be best to divide that section into multiple microservices too.

This diagram brings up another (not too significant) disadvantage; it is difficult to plan and organize the microservices for a program and time needs to be spent to assess the interactions of each microservice.

4. Potential Tech Stack

4.1. Front-end

The foundation for a web app is very different from the foundation of a mobile app so we would need to use different frameworks for each.

For web apps, a popular front-end framework is ReactJS. Having experience using ReactJS myself, I believe it is a good choice as it is easy to use and gets the job done.

If we wanted to create an Android app, the Java or Kotlin languages can be used. If we wanted to create an iOS app, the Swift language can be used. Since we want to create both, it would be much more convenient to use a cross-platform framework. React Native is a front-end cross-platform framework that can be used to create both Android and iOS apps.

Another reason I decided to pick ReactJS and React Native is that they are both similar, so even if we would be writing the web and mobile apps, similarly, a lot of the logic and structure is transferable.

4.2. Back-end

For retrieving climate data, the CDS API could be used along with Python. More information on the CDS API could be found here.

Since our calculations are done in the GAMS environment, the APIs available for use with GAMS are a limiting factor for choosing a back-end language. GAMS has APIs for C#, C++, Java, Python, Matlab, and R. Out of these options, the most reliable for software development would be Python. More information on the GAMS APIs can be found here.

Since all of the calculations are handled within the GAMS environment, there would be a lot of information shared between GAMS and Python.

Because of this, we would probably require a separate microservice that handles the communication between GAMS and Python.

For a back-end framework in Python, the two most popular choices are Flask and Django. Both of these can be used to implement our microservices architecture and both are commonly used so there shouldn't be any difficulties whether we choose one or the other. Because Flask is simply easier to learn and use, I decided to pick Flask for our back-end framework.

For handling payment and transactions, there are many different APIs to choose from and any of them should work fine. The most popular API is Stripe, so I decided we should stick to the most popular option. More information on the Stripe API for Python could be found here.

For our map data, Nearmap is a good option since it provides high resolution, good quality, and up to date images of locations. Nearmap provides multiple APIs depending on what exactly we are looking for. For an interactive map where you can zoom in and pan around, the Nearmap Coverage + Tile APIs can be used. For a simple JPEG image centered at an address, the Nearmap Image API can be used. For More information on the different Nearmap APIs can be found here.

One problem is that the Nearmap APIs communicate with Javascript. Though inconvenient, it is possible to have JavaScript communicate with Python so that we can communicate with the Nearmap APIs through Javascript and then communicate with our Python back-end. Some examples I found of this sort of communication were here. This would add an extra layer for 'translation', so similar to our GAMS-to-Python situation, we would likely need another microservice dedicated to translating between Javascript and Python. I would encourage the software team to find an alternative solution to this problem if possible.

An API gateway can be written by ourselves but since all requests from the front-end go through the API gateway, it can be used to monitor how many requests are coming, how long they are taking, or can be used to implement security measures to decide which users have access to which services. If this is something of interest, an open-source API gateway implementation can be used. One popular option compatible with Python is the Amazon API Gateway. I do not know much about the different API gateways, so I encourage further discussing this topic with the software team. More information on the Amazon API Gateway can be found here.

Since we would have multiple types of data related to one another such as projects, user info, and user preferences, a relational database would make most sense. A relational database is one where you have multiple tables that are related to one another through IDs such as a user ID linked with project IDs in another table to link certain projects with a user. I do not have much knowledge of nor do I have much experience with databases, so I will choose the most popular option which is MySQL (I do not think it would make much of a difference which database you choose as long as it is relational).

4.3. The Final Tech Stack

For easy access, I have listed below the technologies required to develop our app:

- Front-end framework: ReactJS and React Native
- Back-end framework: Flask
- Connecting GAMS to Python back-end: GAMS Python API (more info)
- Climate API: CDS API (more info)
- Secure Payment: Stripe (more info)
- Map API: Nearmap Coverage API + Tile API (more info) or Nearmap Image API (more info)
    - Javascript/NodeJS for using the API and then communicating with Python back-end
- API Gateway: Amazon API Gateway (more info)
- Database: MySQL

5. Conclusion

5.1. Revisiting Our Requirements

Considering our final microservices architecture and the proposed tech stack, all of the functional requirements should be covered.

Now let's revisit the non-functional requirements mentioned earlier in the document:

- Security: Security measures need to be taken in order to keep user information and payment information private. The payment information should be taken care of through the payment API and the user information should be taken care of through the API gateway.
- Scalability: Using a microservices architecture ensures that our project will be scalable.
- Capacity: This should be taken care of by the database.
- Usability: This depends on good design by the UI/UX and front-end developers but is not difficult to achieve.
- Performance: The microservices architecture will not perform faster than something like a monolith, but it will still get the job done. I expect the difference in performance to be negligible.
- Compatibility: With our front-end frameworks, users will be able to access ABen Hub through the web, and through Android and iOS apps.

5.2. Next steps

Since developing a tech stack takes as much time and research as developing a software architecture, and since the majority of my time was spent on the architecture, I believe my proposed tech stack should not be taken as the final solution. Instead, I suggest sharing my proposal with the software developers in the future to either confirm or modify it.

Before the front-end is developed, a UI/UX diagram needs to be developed to be used as reference for the front-end developers later.

The front-end can be developed at any point; either before, during, or after the back-end is done and then any buttons can be connected to the API gateway once the back-end is done.

For the back-end, we can develop it one microservice at a time. The beauty of microservices is that you can work on these functionalities one at a time and still have a functional result at the end that you can test.

The order for completing the microservices is up to the developers but I will give my suggested order; after each step, test thoroughly to make sure everything is working as intended:

1. Complete the calculations microservice that is already in development
2. Complete the communication microservice between GAMS and Python
3. Complete the tools/modules microservice and ensure the correct data is stored/fetched
4. Complete the weather/climate API and use it to feed info to test the calculations
5. Complete the projects microservice and ensure projects are created and stored properly
6. Complete the users microservice and ensure projects info can be linked with each user
7. Complete the API gateway and connect the back-end to the front-end
8. Complete the maps microservice and ensure everything displays as required in the front-end
9. Complete the payment microservice and make sure all info is secure

I highly suggest that you encourage the software team to properly name files and objects, comment the code thoroughly, and strictly follow the software architecture. This is very important as although the developer may understand the code they are writing, newer developers in the future would not understand a thing the previous developers wrote unless the code was written clearly with names that make sense and comments that explain what is going on.

Finally, this plan would definitely change over time so in addition to programming, some time would need to be spent on planning and ensuring that any updates to the program still obey the chosen architecture. Depending on how you look at it, this may be advantageous or disadvantageous. It could be disadvantageous if time is limited. Otherwise, I believe careful planning to be very advantageous for the long run.

6. Resources

- https://www.perforce.com/blog/alm/what-are-non-functional-requirements-examples
- https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013
- https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1181/Arch_Design_Activity/Blackboard.pdf
- https://get.oreilly.com/rs/107-FMS-070/images/Software-Architecture-Patterns.pdf
- https://youtu.be/1vjOv_f9L8I
- https://youtu.be/qYhRvH9tJKw
- https://stackoverflow.com/questions/45661006/what-is-the-difference-between-monolith-and-n-layer
- https://www.techtarget.com/whatis/definition/monolithic-architecture
- https://dev.to/zachgoll/introduction-to-software-architecture-monolithic-vs-layered-vs-microservices-452#:~:text=A%20monolithic%20and%20microservices%20architecture,monolithic%20app%20or%20single%20microservice
- https://www.baeldung.com/cs/layered-architecture#:~:text=Advantages%20and%20Disadvantages&text=The%20framework%20is%20simple%20and,Cost%20overheads%20are%20fairly%20low
- https://martinfowler.com/architecture/
- https://www.integrate.io/blog/the-sql-vs-nosql-difference/#:~:text=SQL%20databases%20are%20vertically%20scalable,data%20like%20documents%20or%20JSON
- https://www.npmjs.com/package/gams2js
- https://confluence.ecmwf.int/display/CKB/Climate+Data+Store+%28CDS%29+documentation
- https://blog.logrocket.com/complete-guide-react-native-web/#:~:text=Conclusion-,Can%20React%20Native%20be%20used%20for%20web%20and%20mobile%3F,browser%20using%20standard%20web%20technologies.
- https://medium.com/@HolmesLaurence/integrating-node-and-python-6b8454bfc272

- https://www.monocubed.com/blog/top-python-frameworks/
- https://www.quora.com/How-is-nearmap-better-than-Google-Earth
- https://docs.nearmap.com/display/ND/NEARMAP+APIS