DIJO

Capstone Project - Report

Digital Bullet Journal (DIJO)

Satya Bhasale (46131470)

Jaleel Abdur-Raheem (45805448)

Fabian Prasch (47926347)

Jonathan Tang (46459688)

Siale Taumoefolau (4739059x)

Samantha Tran (45831661)

05/06/2023

Table of Contents

1 Abstract

This report documents the progress architecture and deliverables achieved by Dijo, a digital bullet journal application. From its initial project proposal, the changes made were user-experience based to allow the development team to focus on features that provided more value. The architecture used by the Dijo software system was a service-based approach on the backend and a technical and domain-partitioned approach on the frontend to promote the extensibility and availability of the system. Throughout development, many architectural decisions were made, with each decision being influenced by the quality attributes and how they were affected. A critique and evaluation of how well the quality attributes extensibility, availability and reliability were addressed, making use of examples and load tests. Finally, the report will include reflections on how the team worked together on the project and any weaknesses that were identified throughout.

2 Changes

When implementing Dijo as a minimum viable product (MVP) the majority of the features and architecturally significant requirements (ASRs) were addressed as the proposal intended. However, there were some adjustments made that were non-functional in nature. Some of the changes to proposal and initial agreed plans are discussed in detail below:

2.1 Removal of Shopping Cart

The main adjustment from the MVP was the mechanism of adding digital assets from the store to a shopping cart that could then be used to checkout. We decided against implementing this feature in this way as we believed for an MVP the functionality of the software was more important than the end-user experience of how that functionality was implemented. Since the assets can be purchased from the asset store in a one-click process, the functionality of Dijo's MVP is not compromised. The MVP has implemented a credit system for the purchase of assets so a one-click solution is suitable. In the future when actual real payments such as credit cards are introduced, a shopping basket might make more sense. A shopping basket would not require much change to the code base with the majority of the changes being required in the front-end code. Since extensibility was an ASR, there would be minimal impact in bringing any future asset purchase changes to the system, as the implementation is highly modular with a service-based architecture.

2.2 Lack of Third-Party Account Management System

While not explicitly stated as a requirement in the proposal, Dijo's MVP does not use a third-party account management system like AWS Cognito. The login and account setup of Dijo's MVP uses JWT tokens as part of the Flask application that are verified against a username and password field in the user database table. Using a service like AWS Cognito has the benefit of not having to store a password in Dijo's user database table, which is more vulnerable to unauthorised nefarious actors. In the MVP, implementing AWS Cognito was determined to be unnecessary for the first iteration of the system, since there are no sensitive details stored on Dijo's systems yet like payment information. Adding AWS Cognito in future iterations of the project would require some limited changes to the user database table and the provisioning of a AWS Cognito service to be accessed by the login and register backend endpoints. This is not a significant change and can therefore be implemented in the future with limited negative impacts.

2.3 Notebook Page Auto Saving

Another feature that was not specifically mentioned in the proposal but was addressed differently than originally intended was saving changes to notebook pages. Initially in our group we thought that Dijo should automatically save the notebook periodically, so that the user does not have to be worried about manually saving their progress. As we implemented the front-end, there was more pressure to implement more complicated visual features such as adding images and assets to a notebook and being able to resize and rotate them. It was decided that implementing a CTRL-S saving method would be sufficient for the MVP, with an autosaved version to be implemented in the future. Having built an architecture that is available and reliable using components like load balancers and autoscaling policies, the infrastructure should be able to handle more frequent periodic auto saving. Therefore, the majority of the changes required to implement auto saving will be in the front-end code base without a major impact on other components of the architecture.

3 Architecture and Models

3.1 Backend

3.1.1 Overall Architecture

The backend employs a service-based architecture for deployment. This allows us to extend the backend by easily adding modular services independent of one another, contributing to our extensibility quality attribute. For files that are common between all services, Dockerfiles also copy a core functionality directory alongside the relevant domain directory. Keeping this folder comprehensive limits duplicated code amongst the services, but most importantly slims down the boilerplate needed to create a new service, thereby increasing extensibility.

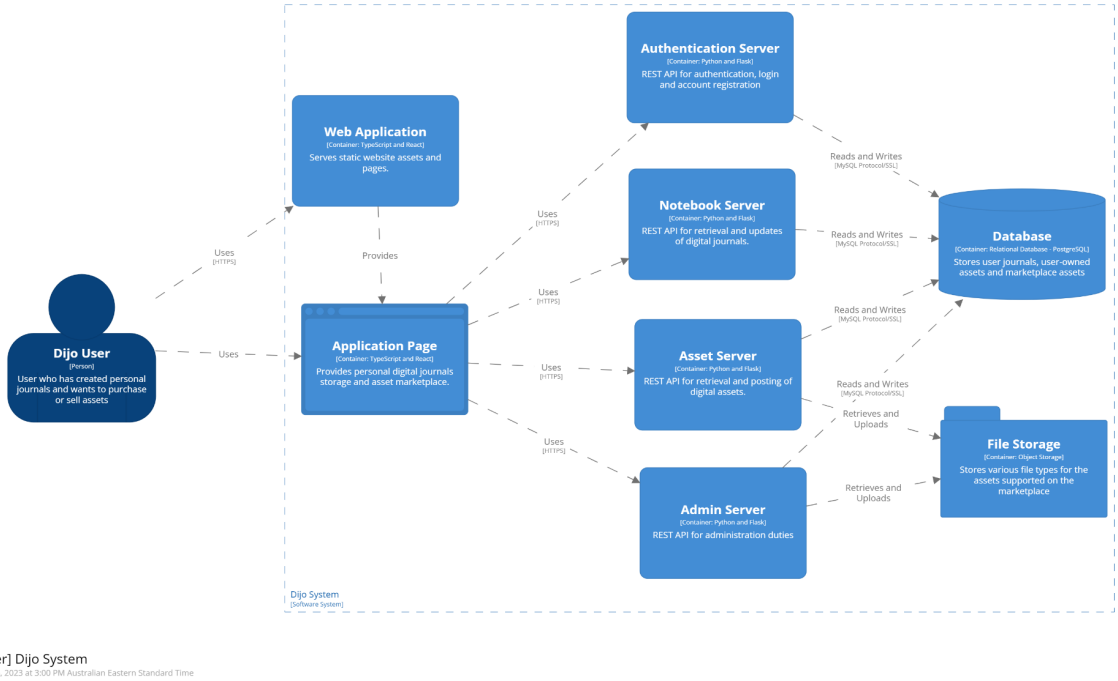

Figure 1: Container Model of the Dijo System

Each service employs a layered architecture to define its custom functionality. For now, this is split up into routes (endpoint controllers that make procedural calls at a very high level) and schemas (split into requests for input validation, and responses for model output manipulation). Technical partitioning at this depth is favourable because it is easier to navigate & manipulate from a developer's perspective. If maintained well, there is a lesser chance for circular dependencies, and the communication between each layer is standardised. With a standardised

communication protocol, it becomes easy to extend new features by following the established pattern, increasing our backend's extensibility.

At the "routes" layer, the endpoints are grouped by path. This made the most sense because we are utilising a RESTful API, where GET, POST, PUT, and DELETE all naturally fall into the same domain. Separating the routes like this makes it easier to navigate to the features we want to look at, as opposed to a non-partitioned "routes" layer with all the endpoints in a single file.

Overall, our backend architecture uses a mix of domain and technical partitioning at different depths to achieve a file structure that is the most readable, exposes the least amount of unnecessary functionality, and encourages extensibility in several dimensions.

3.1.2 Communication Protocol within a Deployable Service

The deployable backend services utilise the facade design pattern to expose core functionality for the user interface, which decouples it from the implementation of the "services" (for lack of a better name, the "services" layer is the one responsible for executing ORM queries with input passed down to it). The facades/routes are designed to be as lightweight as possible, caring only for the flow of data from one "service" to another. In this layer, there are almost no assumptions being made about the "services" layer. This is achieved with the schemas mentioned earlier, which are in fact directly coupled with the "services" for the benefit of flexible function signatures (schemas deserialize input into dictionaries, which are the sole input to service functions making them good for maintainability) and satisfactory input validation/default value loading. All the "routes" layer needs to know is what the expected return type is of the "service" functions. These simplification characteristics are desirable for creating an API that is easily extensible.

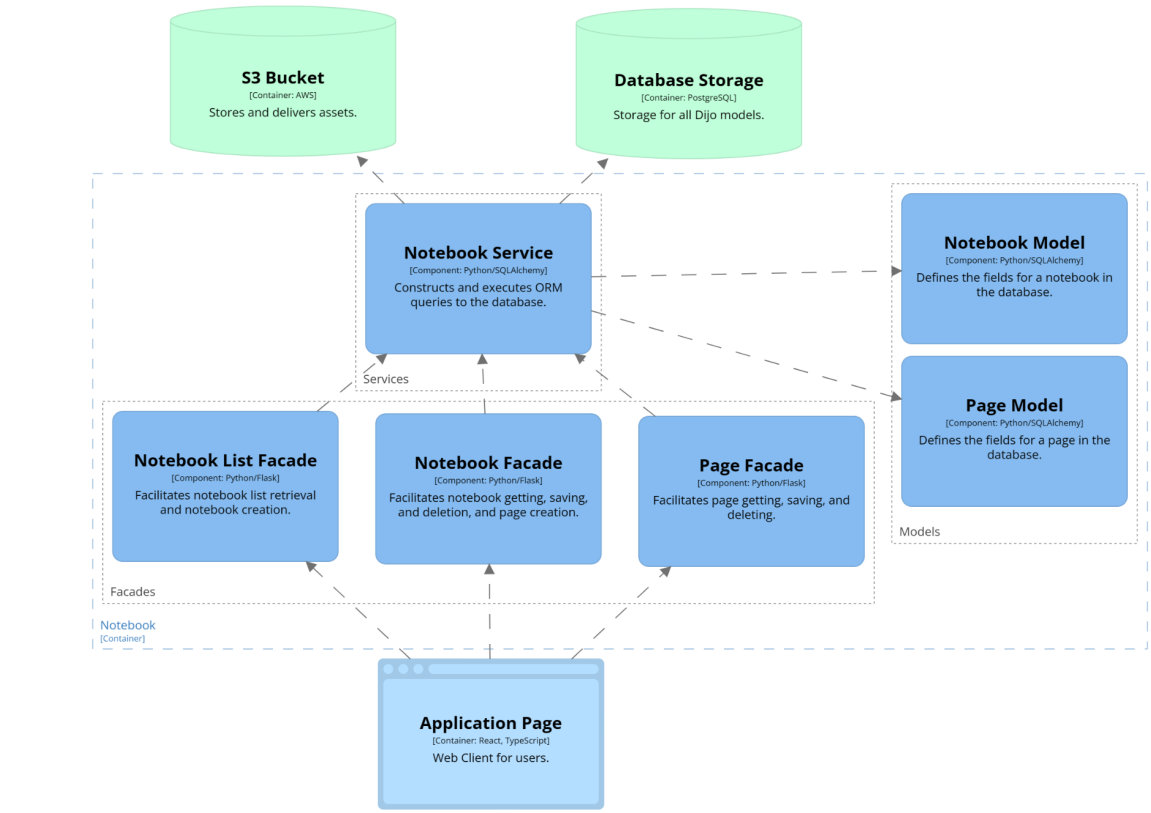

Figure 2: Component Diagram of the Backend Notebook Architecture

Once a schema-deserialized dictionary is passed to a “service” function, the function also operates assuming expected behaviour. We intentionally leave this layer vulnerable to errors like KeyErrors from using the dictionary because we want these errors to follow the upward notification pattern. Assuming a client-friendly error message is thrown from the “services” layer, the “routes” layer catches this and returns it to the user interface, all without needing to know the implementation details of the “service” function. The alternative to this is silent error handling, which leaves parsing the malformed return value back to the “routes” layer. The problem with this approach is that the reason for a malformed value becomes ambiguous, since the point of error is silently erased. In conclusion, we lean into throwing exceptions with specific, client-friendly messages back to the “routes” layer and send them as responses, rather than silently handling errors and introducing ambiguous failure points. This somewhat caters to the reliability of the backend, ensuring messages are accurate and do not expose vulnerabilities like stack traces to clients.

The communication between the “services” layer and the persistence layer is through Flask-SQLAlchemy which provides an ORM interface for making queries. The benefit of using SQLAlchemy is that our backend is mostly relational database-agnostic, allowing us to move away from PostgreSQL if we choose to with minimal overhead. The tables we chose are normalised to reduce the chance of anomalies. In the case of the page table, page content is stored as a JSON string that defines all the components on the page. Storing content as a string saves us from parsing it and constructing new component models, which would increase the computation required on the backend and also increase coupling with the definitions of components on the frontend and backend. This is architecturally significant because it allows our backend to store anything out of the box, allowing the frontend to extend their component definitions freely. It should be noted that if we did decide to load page content as components and store them in their own database tables, the functionality would be an example of a “service” that solely belongs to the deployable marketplace service. This is entirely doable given the architecture/file structure mentioned above (and should have been done to the “services” in the core folder before we ran out of time), giving us the choice to easily implement it in the future.

3.2 Frontend

3.2.1 Overall Architecture

The frontend has followed a technical and domain partitioned architecture to promote segregation based on functionality. The use of this approach in the frontend has allowed for the grouping of reusable components and APIs throughout the application and better separation of specialised components such as individual pages and the canvas. This architecture also promotes maintainability as there is no coupling between separate pages (main components) and only dependencies on small, general components such as buttons and navigation bars. The use of generalised components that act as wrappers allow for consistency in the design and improve the maintainability as updates only need to be made in a single location. Any changes to main components would not require refactoring in others as these components have no coupling between them. Future extensibility is also accounted for as additions can make use of reusable components while maintaining independence from existing components.

The main components of the frontend architecture have been shown in Figure 3. The route manager is extendable and allows for the addition of new pages to the web application. To emphasise the main architecture, minor components have been omitted from the diagram, however would be utilised by the page components.

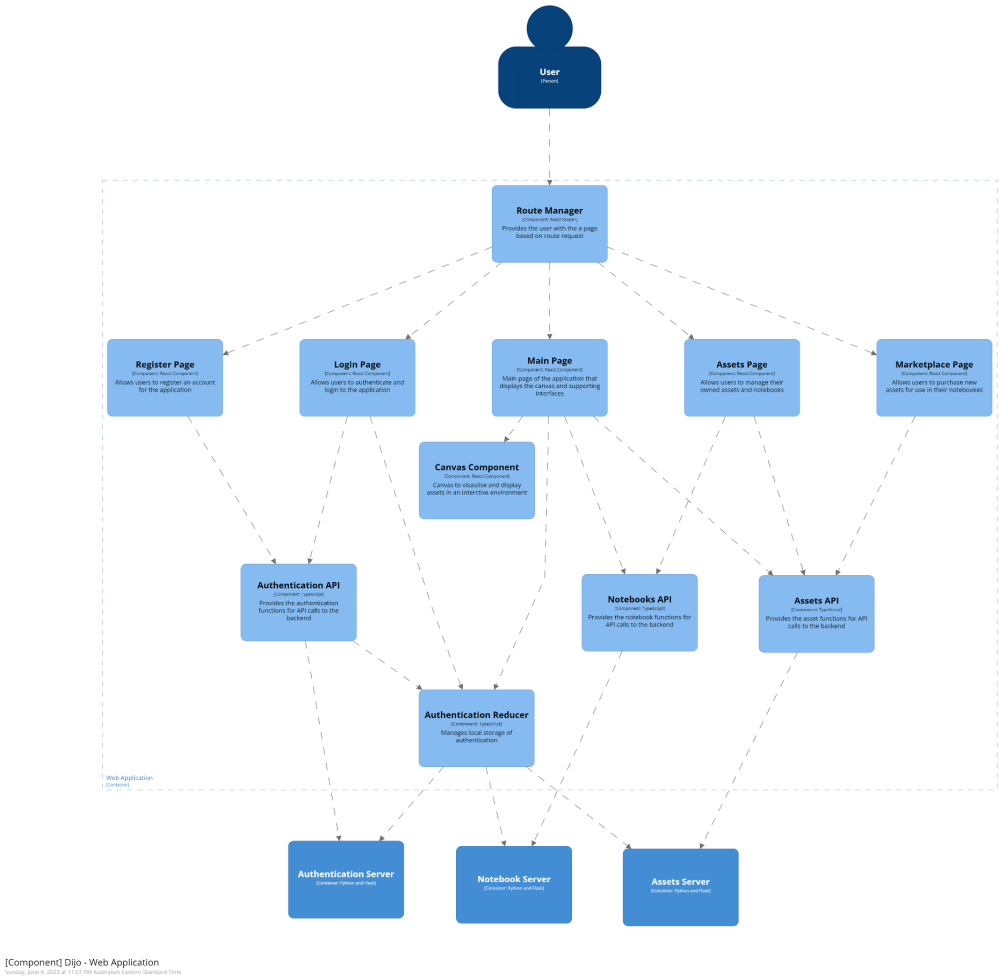

Figure 3: Component Diagram of the Frontend React Architecture

3.2.2 Canvas Architecture

The canvas component on the frontend application follows a plugin-based architecture with a simple interface being the container component. The canvas has been used to handle mouse movements, keyboard inputs, container positioning and saving page contents. The canvas uses a publisher-subscriber pattern when notifying components within the canvas about any updates. This container component abstracts the core functionality observed by canvas components such as grabbing, moving, resizing and rotation. This abstraction allows for the types of assets to be easily extended where the asset itself is responsible for more advanced and specialised features.

Extending the current array of assets available would require the React component of the new asset to be imported as a functional component. This functional component will then be used in a switch statement when parsing the page contents. This function would also need to provide any necessary information that the imported component would need as a prop object. Extension in the

toolbox has not been implemented for the MVP but would be required if an added asset had configuration that the developer would like to have an interface for.

4 Trade-Offs

Throughout the MVP's development lifecycle, several design decisions were made, with consideration of the trade-offs and influence on the desired quality attributes. Some major design decisions are discussed in the following:

4.1 Storing Page Content as a String over an Object

When a user saves their page, the contents of the page must be persistently stored. This design decision is in the context of the format in which the page contents are saved. Initial discussion led to two approaches: the first being a JSON string, the second being page assets and their page related data being stored in tables.

It was decided for page content to be stored as a string in the backend storage. One drawback to this is the neglecting of validation and parsing of the JSON-ified pages. Thus, in the frontend's perspective, incoming pages are not guaranteed to be valid but will rarely be an issue in production due to the frontend managing the creation of said JSON strings.

On the contrary, this approach is beneficial as it promotes separation of concern, keeping digital asset implementation details and definitions in just the frontend. In terms of quality attributes, this approach is far more extensible as adding an asset does not require an additional table to represent the new asset type, as well as additional functionality in the frontend to accommodate for the new asset type. A new asset will solely be a new key within the JSON string, with no implication on pre-existing page content.

Most importantly, it results in less computation in the backend, as it solely queries the 'Page' table and returns the stored string, rather than querying several tables to form a response that describes the page. Similarly, when writing to the database, only the 'Page' table will be written to, in comparison to multiple asset tables. This reduces the workload and time in which tables are locked. One thing to note, however, is that page content strings may contain 10-1000 lines of text, which can potentially cause bloating of the database.

4.2 Front-end Asset Retrieval

This design decision is in the context of how static assets, particularly images in the MVP's context, are retrieved and rendered in the front-end, and stored in the back-end. Initial discussion resulted in two main approaches. Both approaches store the image in an object store (AWS S3 Bucket). However, the first approach has the backend retrieving the image from the object store

and passing it to the frontend as a binary. The second approach involves storing a reference to the S3 object in the database and retrieving the image on the frontend using the stored reference.

Approach two was implemented due the result of a simpler and more manageable method of retrieving data from the backend and database. Additionally, this approach allows the frontend to retrieve more information about assets and provide more flexibility in how it utilises the data. Although, this approach results in the incurring of extra latency on the frontend due to the additional HTTP request to fetch the asset from the object store. By having the frontend be responsible for fetching the image, it decreases the workload on the backend, making the backend servers more available.

Storing assets in an object store and meta data in the backend is more extensive than storing asset binaries within the backend. As stated in the proposal, the application can be extended to include assets such as fonts, page templates and colour palettes. These assets cannot be easily transformed into a binary and will require additional workload on the frontend to convert to web-renderable formats.

In terms of using Amazon S3 as the object store, it must be noted that it is eventually consistent, meaning objects are replicated across services, however, these changes take time to propagate to the replicas. Thus, availability is prioritised over consistency in this scenario.

4.3 Canvas Components Inheriting Minimal Functionality to Support Extensibility

This design decision is in the context of creating a system that is extensible by allowing addition of features and digital assets. Any new component types will need to retain basic functionality such as resizability and transformability. Additionally, in reference to Section 4.1, designs are required to be easily converted from JSON to rendered HTML, and vice versa.

To enforce extensibility, it was decided that each asset will be wrapped by a container that provides transformation functionality. This allows updates to any components related to generic functionality to be propagated to all the other components. Also, the addition of a new component only requires specifying the HTML for the content as well as the properties unique to the component. All of which allow for good design practices as common functionality is re-used rather than duplicated between components, making the codebase neat and maintainable. Additionally, it ensures consistency of the design and how basic component manipulation is handled.

One disadvantage is that development time for the currently supported assets will be increased due to enforcing of generic and abstract functionality. Additionally, updates to the currently

supported fields may not have backward compatibility and will need to be updated on the frontend.

4.4 Service Based Architecture

In the context of delivering an application that is easily extensible and highly available, the system was switched from a technically-partitioned monolithic architecture to a domain-partitioned service-based architecture. DIJO consists of several features that fall into easily identifiable domains (such as authentication, marketplace browsing, notebooking, etc), with some features being used more than others. Additionally, there is an expectation for more features to be implemented in the future.

To reflect these changes on the architecture side, the infrastructure code was changed from only one scalable cluster for the backend service to multiple scalable clusters for each domain. Additionally, the references to tables in the current singular database are lowly-coupled. In the future, it could be beneficial to have separate databases for some domains, which can potentially lower data coupling.

The advantages of such change include increased modularity of components, making the system highly extensible. A more advanced infrastructure which supports availability and scalability. Additionally, the separation of features make for a better unit testing experience. Some disadvantages include increased complexity for a currently small-sized product as well as the loss of distinction between the different layers in the backend service.

4.5 Error Handling in Services Layer

To assist in the delivery of an extensible architecture, the location at which the functionality provided by a service fails has been discussed and decided on. The decision made by the developers involved would be that the service layer of the backend will be responsible for handling errors and abstract this from the routes layer. The alternative that was considered was making propagation errors through the service layer and allow the route layer to handle error management.

The advantage of this change was to minimise the functionality of the route layer and further decouple it with the services layer. This is because the routes layer does not need to be informed about how to handle the error, instead just catch the error and gracefully send the error message. The change does however cause the services layer to increase in complexity which is a tradeoff that was decided would not outweigh its benefits.

4.6 Abstracted Request Parsing and Model Responses

In the context of delivering an easily extensible application, it was decided that the backend will enforce the services layer to catch exceptions and preprocess the messages before they are sent to the routes layer. The service functions deal with many moving parts within the system, and are therefore prone to causing errors.

By moving error catching down to the services layer, this results in the routes layer not requiring conditional statements to accommodate for the return value of service functions being None. This also ensures no confusion is made about the returning of None value. For example, the endpoint for the retrieval of assets assumes that a None value from the associated service function implies the asset is non-existent. However, there are many causes of the None value being returned. In cases where endpoints return specific and presumptuous messages to the user based on a None value, moving these messages to the services layer and throwing them as exceptions will result in the message being closer to the cause of error. As a result, increasing the likelihood that the description is accurate.

One advantage of this design decision is the route layer becoming decoupled with the services layer, due to the removal of assumptions on the return value of service functions. Additionally, the routes layer is slimmed down, allowing for faster creation of endpoint rules and clarity. It also results in more specific and client-friendly error messages that are less verbose and security-compromising.

Whilst the routes layer becomes more lean, the services layer increases in complexity. As the services layer is already verbose due to the ORM calls and conditional statements, adding exception throwing will result in some functions becoming more bloated. However, either way, it is a trade-off between the routes layer becoming more bloated, or the services layer. One thing to note is that status codes will also need to be sent along with the custom error messages as they are closely related and the routes layer is no longer making custom messages.

5 Critique

With reference to the proposal, the 3 main quality attributes relevant to the project include: Availability, Reliability and Extensibility. In terms of availability, the software should always be accessible by users at any time on any platform. If the server were to fail, end users should not notice due to failover or quick recovery. In the context of DIJO, reliability is defined as users viewing up-to-date data and data not being lost. Users should be certain that their request, whether it be the creation of a page or placement of an asset, will be consistently fulfilled without failure. Lastly, extensibility is defined as how well the system's architecture allows for new features to be added. In DIJO's context, developers should be able to add new features and asset types with ease and without any major refactor. The following section will explore and critique how the system enforces these quality attributes.

5.1 Front-end Extensibility

The main extensibility support that the front-end application has to accommodate for is the addition of different types of assets. Functionality for assets include the upload, preview, insertion and use in the notebook. In the general case, the system is capable of extending to allow for different types of assets; however, in the case of assets such as fonts, there is no functionality to support this. The main support for extensibility is provided within the canvas which allows for any type of renderable asset to be displayed and manipulated. The canvas provides extensibility of the assets as anything renderable can be encapsulated by a container. The canvas is ignorant to the implementation of the asset and uses the container as a facade for the asset. The internals of the asset can take the form of any React component such as an image, video, shape, etc.

Updating the current state of the canvas to allow the support for a new asset is simple and only requires importing the react component of the new asset. However, there is currently no support to upload different types of said assets and manipulate the asset via the toolbox. In its current state, functionality for the marketplace and assets would need to be refactored to support the extension of new assets using a plug-in mechanism. The current method of extending the asset support involves importing the asset and adding it as a renderable option, updating the asset upload schematic with the type of asset to upload and if necessary, updating the toolbox interface for more advanced configurations.

Overall, the architecture of the front-end has kept extensibility in mind in some areas but its inability in others have left the system inextensible and requiring manual uploads for different asset types. For the MVP, the architecture was able to achieve the requirements of the default asset types it had set out to achieve but will require extra work on the upload and toolbox configuration to achieve this.

5.2 Backend

Extensibility has been by far the most addressed quality attribute here of the three. The backend contains several extensible dimensions in the file structure (developers can extend more services, routes, services, and schemas), and doing so has been simplified as much as possible. In fact, the Admin service was a late addition to the codebase to deal with some issues that couldn't be fixed through the other endpoints. The time to implement this service locally and define all the resources for AWS deployment was around 40 minutes, which is quite fast considering the other services took the majority of the MVP time allocated to the backend.

However, there are some missed opportunities with the backend. The most obvious improvements would be to refactor the "services" layer in-line with the error handling standard mentioned in 3.1.2, and separate these services out into their dedicated containers if they have one. Doing so would decouple the "routes" layer from the "services" layer, and limit the existence of non-used/privileged source code. This is especially important for the Admin service; it should require quite powerful permissions to use, but the functionality for those powers exists in all of the containers given it is in the core folder.

Reliability has not been explicitly addressed in the backend. We defined reliability as consistently outputting up-to-date information to the client, and never known stale information. This is particularly important in the context of collaborative documentation/scrapbooking and asset purchasing, but these aspects are harder to implement in the backend. The backend does follow a stateless pattern which should allow any instance of a service to respond adequately to a client, but whether the information is up-to-date depends on the ORM queries. Unfortunately not enough attention was given to ensuring atomic requests and locking specific rows for guarantees.

5.3 Database

Ideally we employed a leader-follower replication architecture for the database, but we ran out of time. This would have addressed reliability and availability. Instead, we opted to use AWS RDS Multi A-Z which does backup duplication for a database for us. This does ensure our databases have as little downtime as possible if the primary database goes down, and the documentation states that this event is mostly lossless when dealing with queries queued up for the downed primary database.

We also looked into CQRS for our implementation. Our project could have benefitted from this because our models are quite elementary, and do not expose different sets of information based on the use cases. An example would be displaying an asset in the marketplace and uploading an

asset. There are some fields that are particular for the command like the user id, but the query is more interested in the username.

6 Evaluation

6.1 Canvas Extensibility

To demonstrate the quality attribute of extensibility of the frontend, example documentation of how imported asset types would function has been provided.

The following code snippets are for functionality that would be required to implement such a type in the canvas. As the following example is a basic HTML component, anything that can be rendered using React can make use of the container interface for manipulation in the canvas. This opens up the potential of having assets as raw HTML that can be sold in the marketplace.

```tsx
import React from 'react';
import './Image.css'

interface ImageProps {
    src: string | undefined
}

const Image: React.FC<ImageProps> = (props) => {

    return (
        <img src={props.src} className="image" draggable={false}/>
    )
}

export default Image
```

Figure 4: Code Snippet for and Image Asset for the Frontend Canvas

After importing the asset, the Container component will need to make use of it when parsing the JSON page content. This following code snippet shows how it determines which component to display to the screen.

```
const getComponent = () => {
    switch (component.type) {
        case ComponentTypes.TEXT_TYPE:
            return <Text text={component.text}/>
        case ComponentTypes.IMAGE_TYPE:
            return <Image src={component.src}/>
        case ComponentTypes.RECTANGLE_TYPE:
            return <Rectangle colour={component.colour}/>
        case ComponentTypes.VIDEO_TYPE:
            return <Video src={component.src}/>
    }
}
```

Figure 5: Code Snippet for parsing of a Component Object to correct Component

An example page content as a JSON string is provided below and makes use of the "Type" attribute of the object to determine how the component should be rendered. The canvas component is responsible for the parsing and mapping of each item in the parsed array.

```
export const exampleJson = JSON.parse(`[
    {
        "id": 1,
        "type": "Image",
        "src": "https://${image_url}.jpg",
        "container": {
            "left": 700,
            "top": 500,
            "width": 500,
            "height": 300,
            "rotation": 0,
            "zIndex": 1
        }
    },
]`)
```

Figure 6: Example JSON format of Page Contents

6.2 Functional Requirements and Reliability

A github workflow was created to run automated tests each time a commit was pushed to a branch. The workflow sets up a python environment and runs an automated test suite created using PyTest. This test suite can be found in the folder /code/backend/test. In this test suite, every function in the service layer is tested, allowing us to test the functional requirements.. We tested for both valid calls to those functions as well as error cases. A local test database that is separated

from the main database was utilised. Calls to the S3 bucket were mocked as it is not reasonable to start a S3 bucket every time the unit tests are run.

In addition to the service layer tests, every endpoint is covered by a unit test as well. Only valid calls to the endpoints were tested as the point of the endpoint tests is to make sure that the communication between the service and endpoint layer works and we already have covered all the functionality with the service layer tests.

Not every piece of functionality can be tested adequately using unit tests: uploading assets requires a S3 bucket which does not get covered by the unit tests and functionality like placing assets into a page and rotating those assets also needs to be tested manually; so we also created a test plan. The test plan template that we created is stored in /report. This test plan was executed successfully, with the results are also stored in /report.

6.3 Availability and Reliability Tests

K6 tests were conducted to determine how reliable the system would be when load had been applied to it and whether the servers crashing would make it difficult for the system to recover. The team had identified the main area where reliability was a concern had been the notebook service as this was service that would be the most frequently used and updated.

The first set of K6 tests had the procedure of first adding a page to a notebook, checking that the contents of the page were correct, then updating it and determining if the update had also been successful. The figure below shows how the system was able to handle 10 concurrent users attempting to update the same notebook, with each iteration adding a new page.

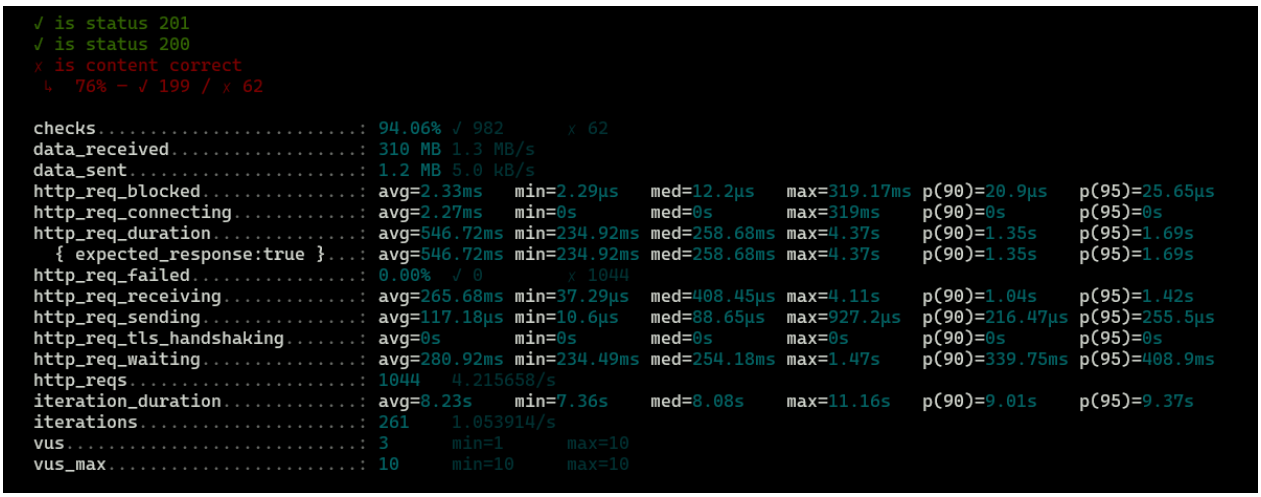

Figure 7: K6 Tests for Adding and Updating Page Contents

The results above had been quite poor, however, in retrospect, these tests had not been indicative of what average and high-load usage would be like. This is because it would be unlikely for new pages to be frequently added to a notebook and multiple users would not update the same notebook. This initial test had allowed us to refine how we evaluated the availability and reliability of the system.

The second set of K6 tests were something more similar to expected usage and would be an area that could be explored in the future regarding collaboration. While still somewhat unrealistic, these sets of tests look at how the system is able to handle 10 concurrent users updating the contents of the same page. This would open the system up to race conditions between the API calls for updating and later retrieval so failures would be expected. As availability was one of the major quality attributes, this test had also looked at how the system would cope over a prolonged period of time rather than scaling up quickly. The results provided in the figure below show results that had been better than the previous set but something that will need to be improved upon after transitioning from the MVP.

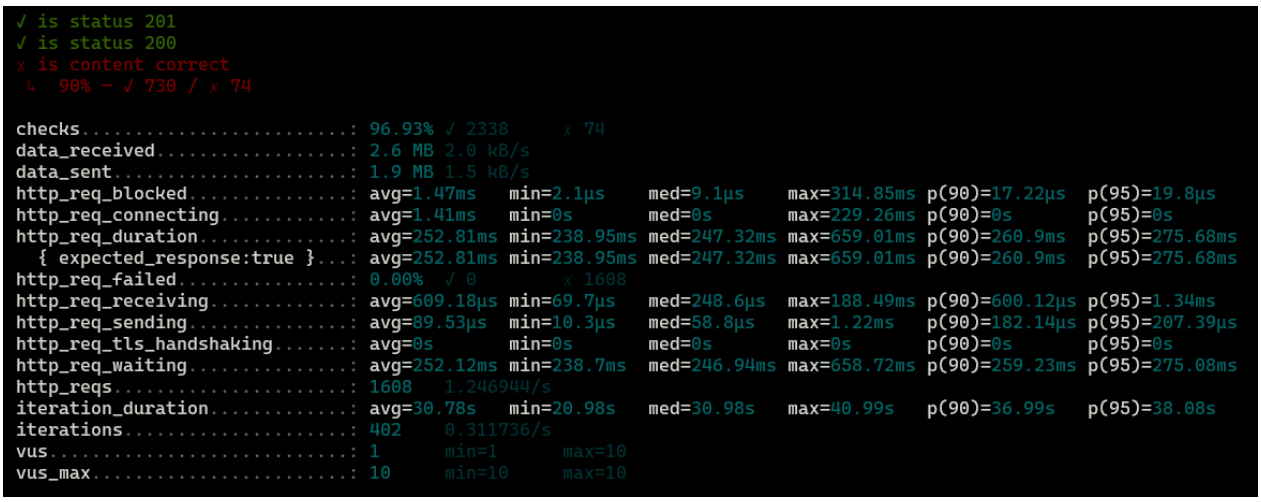

Figure 8: K6 Tests for Concurrent Users Updating the Same Page Contents

7 Reflection

7.1 Front-end Reflection

The work on the frontend had produced an interface that had exceeded the initial expectations that we had first set for ourselves. Although we had been proud of the functionality delivered, our main quality attribute extensibility should have been our main priority.

There were a few problems that arose within the developers working on the frontend however, and areas that we would look to do differently in the future. Collaboratively developing on a shared codebase was something new to us and had become an issue with how each developer structured and formatted their code. In the future, some things that the team working on the front-end would do differently would be to collaborate throughout the initial stages of the project to develop a set of guidelines to maintain consistency. This would improve the maintainability of the codebase as there would be less duplication and a standardised approach to solving similar problems. The use of thorough code reviews would have also eliminated this problem however due to time constraints were not enforced.

In hindsight, we thought we had not architected the front-end with the quality attributes in mind, rather we had focused on producing the functional requirements of the MVP. If we were to do this again, we would instead spend more time architecting the system in such a way that would address our quality attributes, in particular, extensibility. Having not focused on the quality attributes had meant that our MVP was unable to deliver the non-functional requirement to the extent that we would have liked.

7.2 Back-end Reflection

The backend architecture went through several overhauls, but each one prioritised extensibility so the results were ideal every iteration. Especially the switch from a monolithic architecture to a service-based one; partitioning our code into domains meant that functionality related to one domain was not going to severely disrupt the development of another domain's functionality. However, the switch itself suffered from concurrent development. What we learnt here is that we should have designed our service-based architecture from the start. This would have essentially removed the "lock" on the backend during the overhaul. If it weren't for the lock, we would have had more time actively changing the codebase and included more features in the MVP. The drawbacks to this approach is obviously the slow startup of getting the backend running; we would need to implement a reverse proxy for the services and pay more attention to our file structure earlier.

Another reflection on the backend relates to the extensive standardisation we employed throughout the team's process. Changes like the introduction of request/response schemas, modularisation of code, and explicitly defining the purposes of each layer in the services were all in favour of extensibility, but the miscommunication of these standards led to them not always being employed when introduced. Reflecting on the way these standards were implemented, perhaps more effort should have been given explaining their purpose and how to reproduce the same consistency for edge cases encountered throughout the codebase. This would have limited the number of retreads needed to bring all the code up to standard. In the future, the team should use more direct communication between the original implementer of the standard and developers who are now responsible for maintaining it. In the context of this project, peer programming would have supplemented this well.

7.3 Overall Reflection

As a team, some aspects of the project that we thought we had done well was the collaboration between the front end and the back end. Many of the issues that we had encountered during integration were quickly resolved as members in the team were always available and willing to help. This close collaboration had continued into general programming and had allowed us to obtain quick feedback and help debug minor problems.

Areas of the project that we believe caused major issues and setbacks in the completion of the project was how we planned the timeline of the project. Due to each member's individual reasons, development on the project had only commenced within the last 2 weeks before the deadline. This meant that we were unable to thoroughly plan out our work but had been looking for quick and easy solutions. In the future, the team has agreed that we would plan and employ a project timeline that would prevent the need to rush near the deadline.