CS Capstone Design Document

November 24, 2019

Peerist Web App

Prepared for

Peerist

Graham Barber
Sebastian Benjamin

Prepared by

Group 65
Peerist Development Group

Christopher Hoskins
Darius Moseley
Michael Chan
Jorge Manzo

Abstract

This document will discuss the technical implementations of the Peerist web application that what will be agreed upon with our client. User interface plans and approaches to achieving an easy to use user experience are defined, along with how this minimal interface will relate to GraphQL and Docker. Data management with PostgreSQL, interfacing with Hasura, and querying with GraphQL are defined. The process of serverless rendering with Next.js integration and Zeit Now is discussed. Last defined is our approach with Docker and Docker Compose for setting up development environments, along with a small section about using Digital Ocean for staging in the future.

Contents

1 Introductions 2

1.1 Purpose . . . 2

1.2 Scope . . . 2

1.3 Context . . . 2

1.4 Summary . . . 2

2 Glossary 2

3 Viewpoint: User Interface 3

3.1 Stakeholders . . . 3

3.2 Design Description . . . 3

3.3 Design Concern . . . 3

3.4 Design Element . . . 3

3.5 Relationships . . . 4

4 Viewpoint: Data Management 5

4.1 Design Description . . . 5

4.2 Database Design Element . . . 5

4.3 Querying Design Element . . . 5

5 Viewpoint: Server-less Rendering on React.js 6

5.1 Design Description . . . 6

5.2 Design Concerns . . . 6

5.3 Design Element . . . 6

6 Viewpoint: Composition Design 6

6.1 Design Description . . . 6

6.2 Design Concern . . . 6

6.3 Containerization Design Element . . . 7

6.4 Orchestration Design Element . . . 7

6.5 Container Networking Design Element . . . 8

6.6 Staging Design Element . . . 8

6.7 Design Relationship . . . 8

7 Conclusion 8

Appendix A: Gantt Chart 9

1 Introductions

This project is a web application for academic paper review.

1.1 Purpose

This document will detail the design views and viewpoints of Peerist. It will also describe some of the development tools used to implement Peerist's requirements.

1.2 Scope

Peerist will be implemented as a database-enabled dynamic website. Development will occur in January 2020 through May 2020. The main end product that will be delivered is a functioning web application prototype. It will consist of a PostgreSQL database, Hasura/GraphQL, and Next.js/React. To assist in development, Docker and Docker-Compose will be used by the development team.

1.3 Context

The aim of Peerist is to create a tool for collaboration in the space of academic research papers. Peerist will provide an online platform to make it easier for users to provide feedback for academic works in a way that is modern and pleasant. This service will also enforce private peer-reviewing once segments are composed into larger components.

1.4 Summary

2 Glossary

Term	Definition
Segment	A limited body of text to be reviewed. Segments can be independently versioned, and grouped into larger works. Segments will have a maximum word count. Segments can be reviewed by Peers or Circles.
Paper	Papers are made of segments. Papers are independently versioned. Papers can only be reviewed by Circles.
Circle	Circles are groups of users that share a specific relationship. These could be members of a university's department, members of a project team, or users that have otherwise selected each other. It is recommended that members in a circle have at some point seen each other in person.
Peers	Peers are groups of people that share a more general connection than Circles. It is more likely that members within a peer group have not met each other. Peers can include people with similar jobs or research fields. Peers are selected using algorithms provided by the client.
Feedback Prompt	The Feedback Prompt is the topic of review. Prompts are pointed and have accompanying possibilities for edits. E.g. the Feedback Prompt "Word Choice" would allow users to select predetermined responses on diction, jargon, and term comprehension.
Version	Each paper and segment will be independently versioned each time they are updated. The version stream will allow the user to view changes made over time and the review results for each version.

3 VIEWPOINT: USER INTERFACE

3.1 Stakeholders

Primary stakeholders include Peerist members Graham Barber and Sebastian Benjamin, and the Peerist Development Group (PDG). The PDG is the primary designer for versions up to 1.0 of Peerist. The PDG is responsible to Peerist to develop the proof of concept alongside their desired design. An end goal is to develop a working system that Peerist can then optimize to run on cloud servers that are scalable for mass use.

3.2 Design Description

This section relates to the user experience of the Peerist service. Peerist should be intuitive and memorable for the user to use. A goal will be efficiency and fast load times.

3.3 Design Concern

While Peerist is designed as an academic service, it should not take a master's degree to operate. Some functionality, such as version control for papers and segments, should be familiar and similar to major existing services. Access to primary functionality such as submitting and reviewing papers/segments should be navigable from any page. Users should feel comfortable editing papers/segments within the web page, without having to use other applications such as Microsoft Word. A first time user should be able to quickly perform primary application functions. JavaScript (JS) and CSS are arguably the most important components for the front-end of a website. A framework that can effectively and efficiently manage the two languages is needed so that Peerist is a responsive service. The framework will provide many features such as dynamic scaling built in, saving hours of development time. The framework will need to be able to create a user experience similar to that of the current Peerist website model and our current mock-up. A design concern for Peerist Foundation is having a design that can be adapted to cloud computing services and server-less lambdas. A PDG concern is using a well-established and documented framework for a proof of concept. This view will be satisfied when we have integrated a framework with built in JS and CSS functionality that can create the established style and UI for Peerist.

3.4 Design Element

As the stakeholder end-goal is a proof of concept, we will not perform an extensive user study for validation of our user design. User design will be evaluated by all the stakeholders involved. Additionally, we will conduct persona walk-throughs to understand how our primary audience for Peerist might interact with the software. This psuedo-blackbox testing will help to uncover any errors in functionality and experience. A utilitarian, modern, and minimalist design philosophy will help users navigate Peerist and use its core functionality. Elements should only be on the web page if they assist the user in their current task or are essential to navigation. As an example, the only elements that should be present on a segment review page are the navigation bar, the segment, and directed evaluation components.(Figure 1) To simplify use of CSS and JavaScript across Peerist, and to create a consistent user experience, we will use a React component such as Rebass.js.

3.5 Relationships

The front-end will rely heavily on GraphQL for data retrieval. The front-end will have to pull a majority of the content it displays from our database. Interactions include pulling/updating segments and papers, version control, user information, and segment/paper review. A priority for the front-end will be creating dynamic containers that can format data coming from the database. Any static files (HTML/CSS/Media) will be served through our docker container.

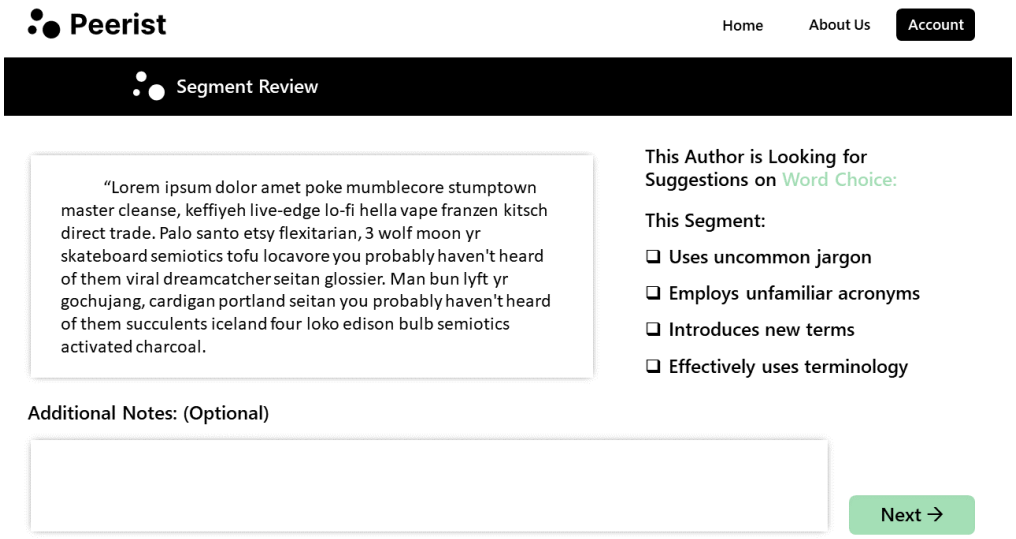

Fig. 1. A Mock-Up of A Segment review Page

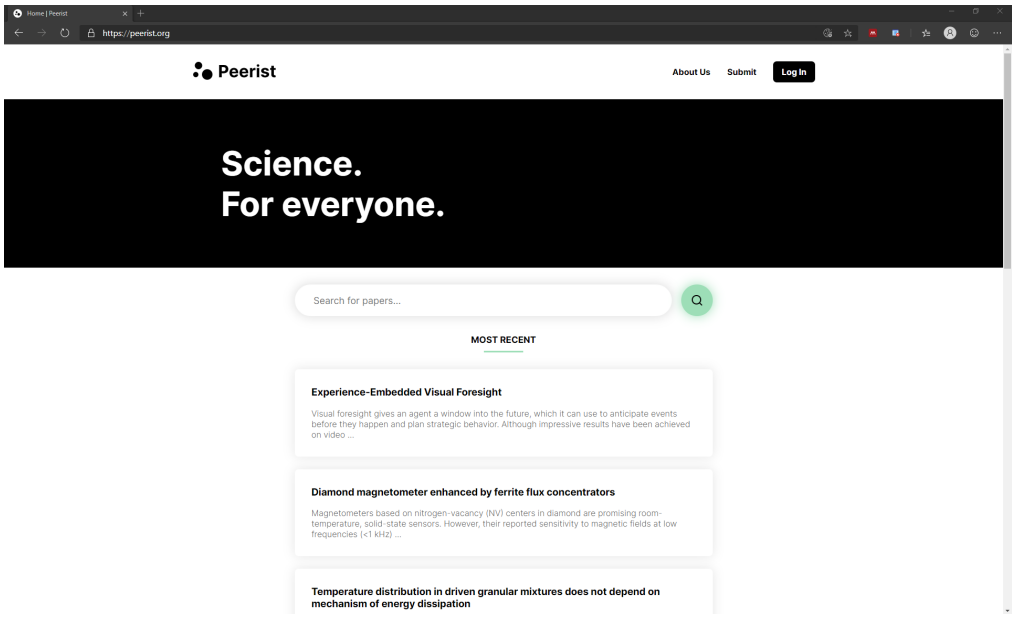

Fig. 2. The current "dummy" homepage for Peerist

4 Viewpoint: Data Management

4.1 Design Description

PostgreSQL will serve as the database for the Peerist application. The data that Peerist deals with is almost entirely relational, which is what makes PostgreSQL a great choice. To handle the querying of the PostgreSQL database we will use GraphQL combined with Hasura's GraphQL engine. GraphQL as a query language for Peerist is the better choice over a REST architectural pattern because of our situation of dealing with relational data and that it will allow for us to reduce the size of API responses to be as minimal as possible.

4.2 Database Design Element

The data that Peerist will deal with is entirely relational. PostgreSQL will allow us to specify relationships between the data that will allow us to fetch the data in different ways. The data entities that Peerist will contain include: users, segments, papers, circles, and feedback. Users will own segments, circles, and feedback. Papers will contain segments. Circles contain users and papers. Feedback will be owned by either a segment or a paper. A more in-depth view of the relationships between the entities in Peerist can be seen below in Figure 3.

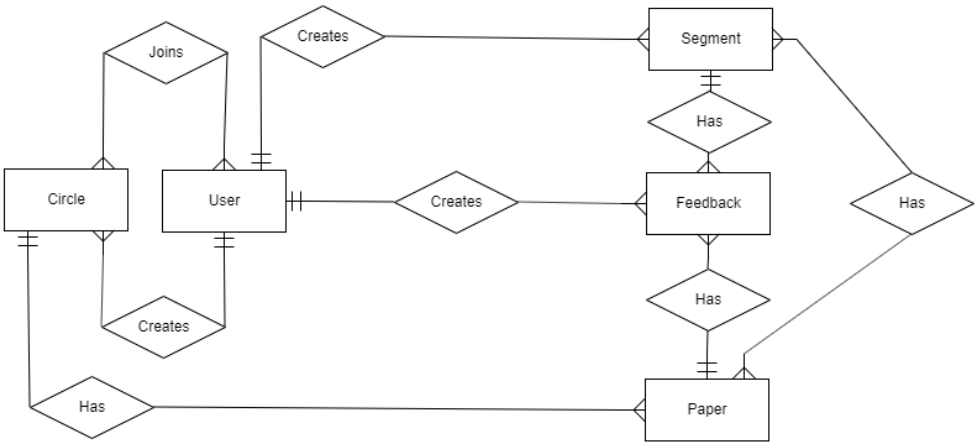

Fig. 3. Peerist Entity Relationship Diagram

4.3 Querying Design Element

GraphQL will allow us to query the database for the exact data we need in multiple ways (due to the relational characteristic) in a lightweight fashion. The API we will use to interface over the PostgreSQL database will be Hasura's GraphQL engine. Hasura's GraphQL engine will allow us to retrieve data from the PostgreSQL database using GraphQL queries. GraphQL allows for the client to specify the response format, and for the server to specify which data is available. GraphQL is best used in situations where the client needs flexible responses to reduce multiple HTTP requests and queries. This allows us to make use of the relational characteristic of our data when making queries. Theoretically, this will reduce the number of HTTP requests that we will need to send from the client.

5 Viewpoint: Server-less Rendering on React.js

5.1 Design Description

This section relates to the interactions between React.js, Next.js, server-less rendering, and server-less lambdas to improve performance for the client. Peerist should be a smooth experience with quick and frequent load times and updates to data.

5.2 Design Concerns

Users can perform various actions, such as logging in, updating account settings, uploading segments, combining segments into papers, interacting with their circles, and trading feedback. User experience suffers if website navigation and large operations like combining segments into papers perform slowly. Overall, users should be able to go between website pages near instantaneously and interact with the user interface seamlessly. The exchange of data between UI and server should be quick and efficient.

5.3 Design Element

These interactions will be handled using Node.js. Through Node.js, any interaction that accesses the database will initiate a GraphQL query on the PostgreSQL database. The interface that manages all user interactions will be created through React.js and rendered server-side using Next.js. Next.js renders server-lessly by providing a server-less lambda to Hasura. Lambda functions allow for the modification of data within the lambda function itself. Data should properly cycle between Hasura and the lambda and experience updates when necessary. Furthermore, server-side rendering speeds up rendering by sending a fully rendered page that was pre-rendered on the server to the client, rather than taking time to render on the client. Next.js also improves performance through automatic static page optimization, page-based routing, automatic code-splitting, and optimized page pre-fetching.

6 Viewpoint: Composition Design

6.1 Design Description

Docker provides a separation between applications and the operating system they are running on. This allows easy execution of apps across many different operating systems. Docker provides several images, applications already built and ready to be instantiated into containers, that we can use for Hasura and PostgreSQL.

6.2 Design Concern

By using Docker, it will improve and ease the development of the Peerist web application, as well as satisfy the requirement that PDG uses Docker.

This concern relates to maintaining contained essential service applications for the development team. Using Docker we address the design concern of keeping our application independent of the developer's operating system.

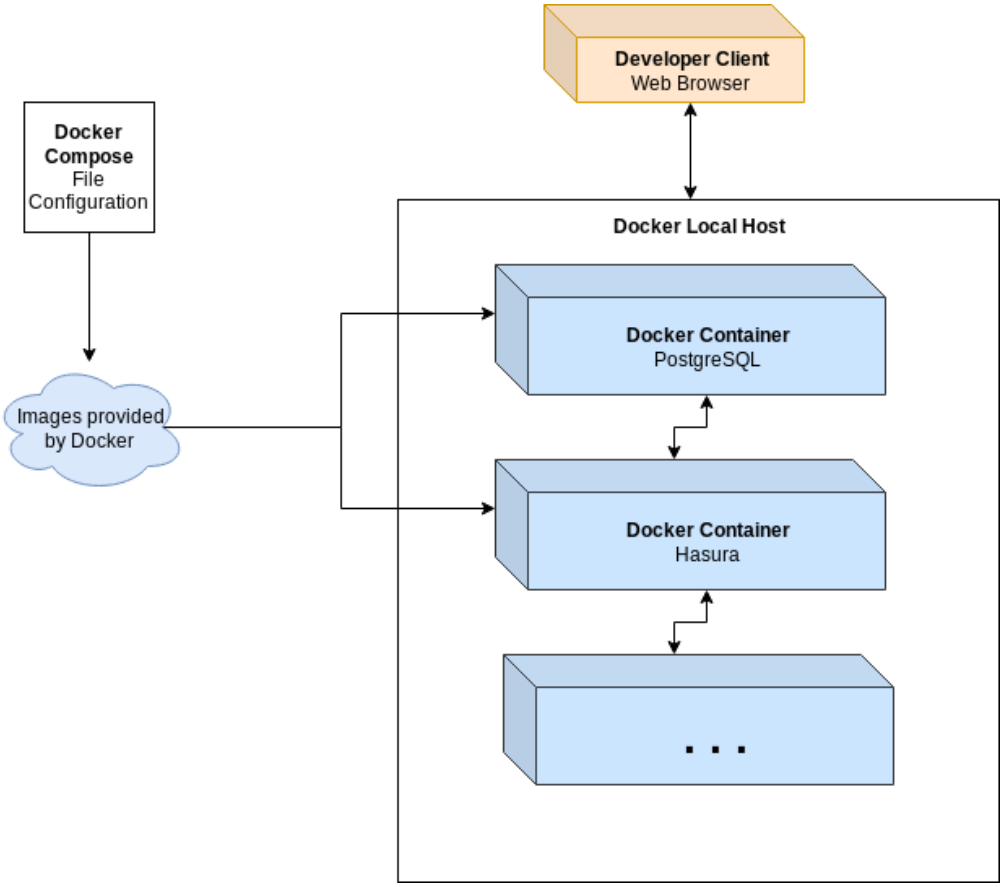

Fig. 4. UML Deployment Diagram for Development Environments

6.3 Containerization Design Element

Essential services are defined for Docker using a file called a Dockerfile. In each Dockerfile, we define commands to create an image for an individual service. According to Docker best practices, each application should have one concern. In our case, we would have one Dockerfile for each of the essential applications we use for development if we were not to use Docker Compose.

Inside the Dockerfile, we specify which image we will be basing the application from. For example, Docker provides an image for Hasura which we can configure as needed. Within this file we also specify any environment variable definitions, any other extra packages, and finally the default commands of what the container should run when it is created.

6.4 Orchestration Design Element

Once these Dockerfiles are distributed to our development team, each team member can build these into images. Our development team can build, start and stop all images from our docker files using the Docker Compose service. Docker Compose will uses a YML file to define how to build the images, start start containers from images, for the entire application.

For our project, our team will actually be relying on an already configured Docker Compose file provided by Hasura. Their configuration already is setup to pull Hasura and PostgreSQL images from Docker Hub, a service that provides

ready-made images. Our client has indicated that these two are the main services that we should use Docker for.

Docker Compose saves time for developers by putting control of the entire application's services into one place. These files can then be added to our source control for easy replication of our development environments.

6.5 Container Networking Design Element

According to Docker's Compose documentation for Networking, Docker Compose allows us to define a network for our containers. Using this network, each container we create is added to that network, and other containers can now communicate with them.

Further, Docker Compose will expose the GraphQL endpoint to other containers as well as the local host. It also exposes the useful Hasura web console too via a similar endpoint. This is a method of allowing each application within a container to communicate with other containerized applications. In our project however, Hasura has already provided a configuration of Docker Compose with important ports already available.

6.6 Staging Design Element

Towards the end of this project, our client would like us to use Digital Ocean for staging parts of our project. Digital Ocean is a cloud compute service which allows developers to purchase virtual machines running on remote servers.

Our project would move its Hasura and PostgreSQL services from local development environments with Docker to Digital Ocean. Digital Ocean's droplet service provides excellent compute resources at a low price. Fortunately, because we are using Docker Compose, setup time for Hasura and PostgreSQL is minimal. By moving these two services to remote machines, it will allow us to test our application in a more production-like environment.

6.7 Design Relationship

By using Docker and Docker Compose our development team can define what applications will be used by docker, how they are stopped and started, and how they communicate with each other. Because Hasura already provides a Docker Compose file for both Hasura and PostgreSQL, it will make setup of these two core services simpler for our team's development. Not included in 4 is a service such as Node.js. The service provided by Zeit Now will be used by our team to take our React.js components and turn them into browser ready code.

7 Conclusion

The Peerist Development Group will use minimal user interface design approaches to build a user experience that is approachable by anyone. By taking advantage of PostgreSQL with Hasura and efficient GraphQL queries, our server-side rendering with Next.js will deliver this experience. Zeit's Now service will be used to pre-process any database interaction as needed in a cost effective manner. Lastly, the Peerist Development Group will take advantage of well contained and easy to use Docker and Docker Compose services to start and stop their development.

Appendix A

Gantt Chart

This Gantt Chart represents our projected progress for the development period.

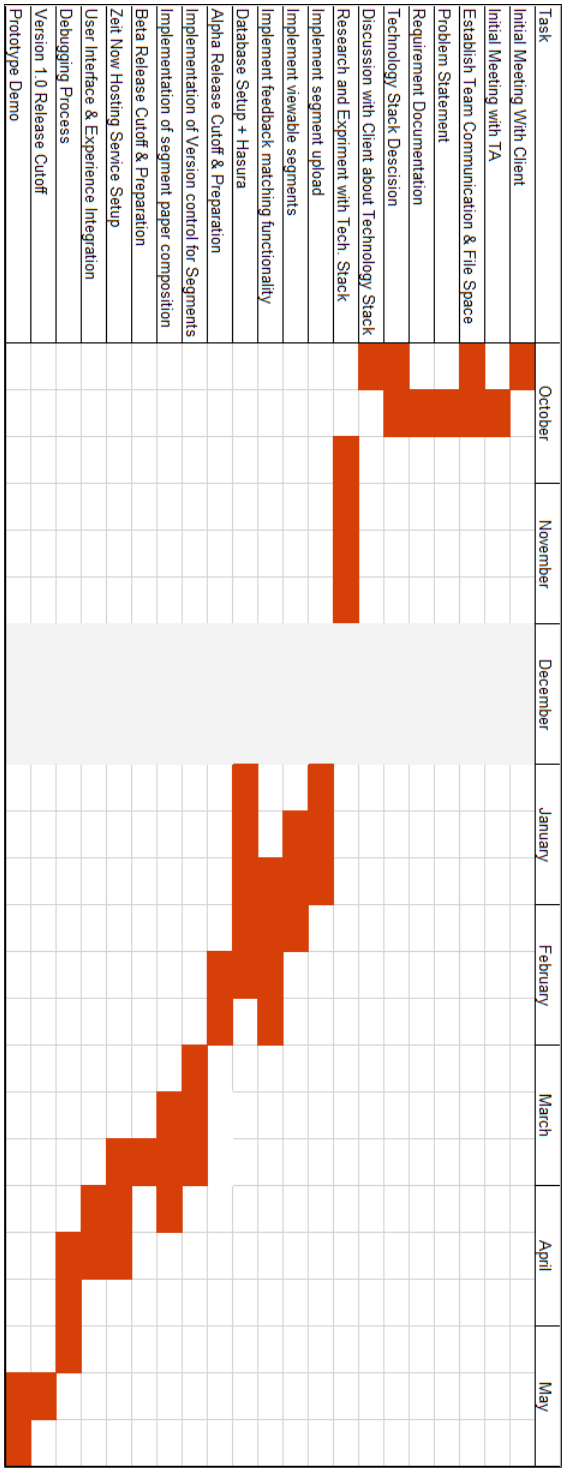

Fig. 5. Gantt Chart