RentAll

Asher Kirshtein

March 2023

1 Features

1.1 Why is this relevant

Often times people have a lot of things around their house that is collecting dust and are not doing them any good, it is simply taking up space. Most of the time things don't get thrown out because we know it is useful but we still just never get around to using them. If you have an extra car or an old gaming system or anything similar you are now able to use these to make you some money instead of simply letting it waste away.

On the flip side of this if you are looking for something cheap and convenient to rent out near you. You can simply log onto the app and search what you need and in minutes you can be on your way to getting exactly what you need. It is convenient cheap and easy for users. Think about how many times you need a drill or a tool that you never will need again. Why would you spend 100 USD on a drill when you can rent a used one for cheap?

1.2 Who is this for

This app can be great for anyone whether you are an individual who would need to rent something quick or if you need something for weeks months or even years our new app will be great for you.

If you are simply someone who has too many things lying around and want to earn a little bit more cash this app would be great for you. Even if you wanted to start an entire business where you rent out many different things. Even large industries will benefit by having an app where those companies can be more accessible to customers.

1.3 What does the app look like for users

1.3.1 Login and Privacy Settings and Security

On the login I would want to have 1 profile for both renters and people looking to rent for ease of access since logging into something will be annoying. I will also include 2-factor authentication for security since users will be having their cards and information likely saved on the site.

1.3.2 searching

I would want to have a search bar at the top of the page for users to be able to search for what they are looking for and easily find it. Ease of access and simplicity is key.

There are also going to filter so you can get anything within a certain radius of where you live(right now I am also contemplating instead of doing it by radius doing it by how long it will take you to get to where you need to be to pick up the item as they do with google maps) Anything that would be outside of that

radius would need to be shipped over and would be a different filter. I would also want to filter based on the condition and price and rating of the renter.

It is going to include an explore section or a for you section that is specified to the individual user based on past rentals and past searches so that they may be able to find something they didn't know they were able to rent.

Multiple product categories: The app will offer different categories for products, making it easy for users to find what they need. For example, you could have categories such as electronics, sports equipment, tools, etc.

1.3.3 Reviews

We will have reviews under each seller with different star ratings as well as user comments on their profiles

1.3.4 User Profiles

User profiles will include a profile picture of the user along with their products listed in order, you will be able to filter the order by date and price and be able to search within only the user. You will also be able to read their reviews as a seller as well as a renter in order to assure yourself you can trust the other user. I also want users to be able to connect their PayPal to the user in order so they can be paid.

1.3.5 Messaging between users

I would like to include a message board similar to a Facebook messenger or Instagram direct messaging so that users will be able to communicate without actually giving up their phone numbers. I would also want to allow users to be able to call sellers across the platform

1.3.6 Options when looking at a product

Users will be able to read reviews on the product as well as the user renting the product.

we will include Customizable rental periods to will allow owners to set their own rental periods for each product, and renters should be able to customize the rental period based on their needs

I want the user to be able to see multiple pictures that the seller has posted for the user to see. I will include a way for the user to contact the seller to ask about the product.

The user will also be able to read a description of the product and why the user is renting it.

if the user needs the product shipped instead of being able to pick it up themselves it will include a shipping cost as well as estimated delivery to that specific user

We will offer a price comparison feature, allowing renters to compare prices for similar products from different owners

1.3.7 Options when renting a product

Ability to add product pictures as well as descriptions and prices for the product where you can add the condition and reason for renting out the product. I also would want to include a quantity section so that users will be able to add multiple of the same product and be able to rent out many things on the same

tab.

We will allow owners to offer delivery and pickup options, which can be convenient for renters who may not have a vehicle to transport the product

1.3.8 Payment portal

We will have a secure payment portal with many options for users to be able to rent with ease. I want to include Paypal, Venmo, Zelle, Cash option in person, Apple pay, and Google pay. Just as many methods as possible so that users will be able to pay with ease.

We will allow for automatic payments to make the rental process more streamlined and hassle-free for both renters and owners. We will allow users to set up recurring payments so that users will be able to rent out an item multiple times or have it for long periods of time.

1.3.9 If there is an issue

Users will be able to report other users for any issues they have when dealing with other users.

We will provide customer support to help renters and owners with any issues they may encounter during the rental process. This can include a help center, FAQs, and a contact form or chatbot.

1.3.10 Notifications

We will include Notification of product availability, we will send notifications to renters when a product they are interested in becomes available for rent.

We will also include notifications for users to be able to know when someone wants to rent their product.

Users will be able to opt-in for different notification types, they will be able to receive emails, texts, and even notification bubbles on their phone so the notification will be able to go directly where the user wants the notification to go.

1.3.11 Product Tracking

If an item needs to be shipped back and forth Users will be able to see where the product is in transit and will be able to see the status of the product. I also will want to have a tracking number of the product so users will be able to see the tracking of the product through the service the sender sent it.

1.3.12 Analytics and reporting

We will offer analytics and reporting features to help owners track the performance of their rental products and make data-driven decisions

1.3.13 Virtual try-on

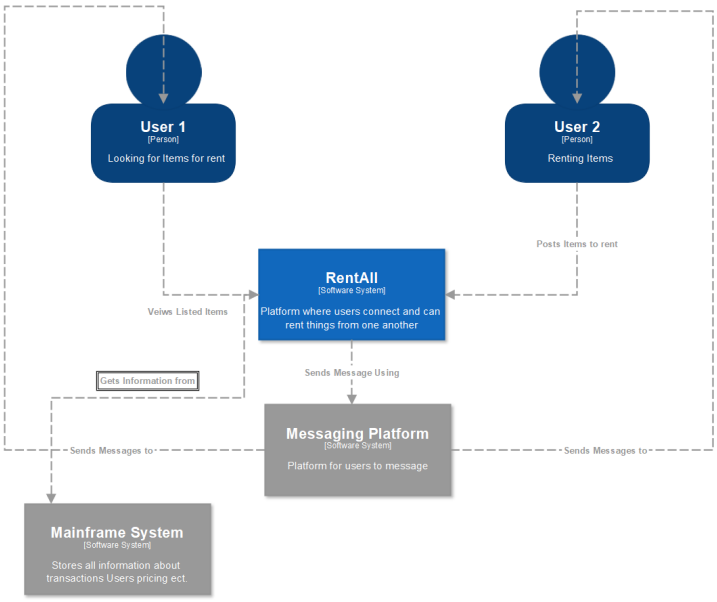
a virtual try-on feature for products such as clothing or accessories, allowing renters to visualize how the product would look or fit before renting it.

1.3.14 Referral program

We will offer a referral program that will allow users to rent and rent out a certain number of products at a discount, the discount will come from us not collecting a fee on the transaction.

2 System Components

2.1 Level 1 Diagram



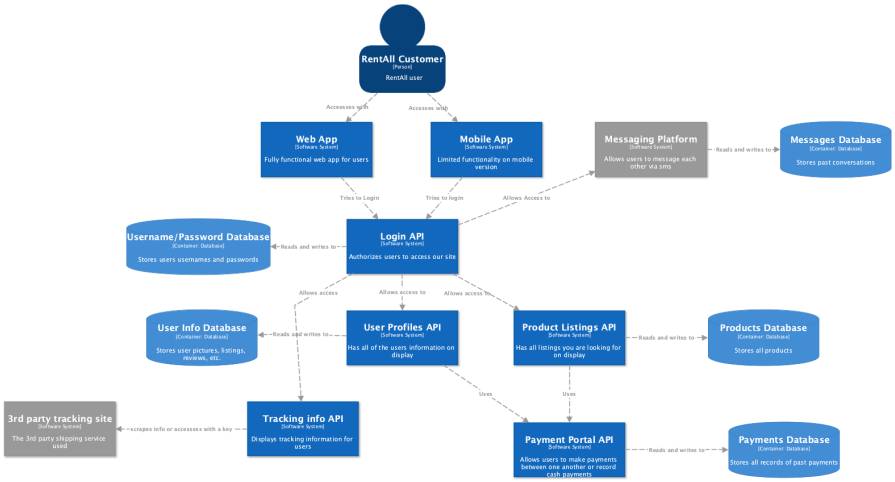
2.2 RentAll

The front-end application where users are able to post listings and view listings posted by others It connects to the backend mainframe which deals with the storage of the information of users, deals with payment between users, login and security, deals with user profiles and ratings etc.

2.3 Messaging Platform

Similar to facebook messenger our messaging platform is going to be slightly separate from the actual app itself. Deciding to message will take users to a differnet app to message each other about the listings and products.

2.4 Level 2 Diagram



2.5 App containers

2.5.1 Login API

When users open our page they will be met with the login page of our app. The login page is going to allow users to log in to their accounts. The usual scenario will be that a user will enter their username and password and then we will pull the information from the database if it matches then the user will be able to move to the next part where they will receive a message for the 2-factor authentication process. Once they enter that pin they will have access to the other features in the app. In the event the user doesn't get a text they are able to have the text resent to them. If the user forgets their password they will be

able to reset their password from the registered email that they have provided.

2.5.2 User Profiles API

Within the app, users are able to view other pages. If you visit a users profile the user database pulls the information and displays all of the features for the user to see.

2.5.3 Products Listing API

When a user looks for a product they can visit the product page and the information on the products available are pulled from the database. The product listings allows user to search and filter products based on a variety of features.

2.5.4 Payment Portal API

when a user decides to make a purchase they are taken to the payment portal which enables users to purchase products through our website. If the user decides to pay with a 3rd party site then they are able to record an external payment. All payments are recorded and stored in their own database

2.5.5 Tracking Info API

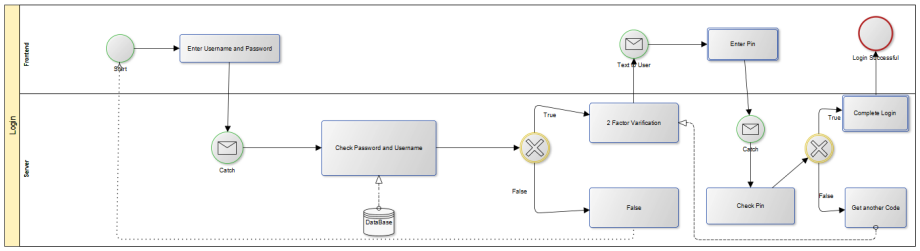
For the tracking info it will be taken from the 3rd party site. It will be accessed with an API key if available but if not the data will be scraped from the other side so the user can view the projection on when their product will arrive and the status of the product.

2.5.6 Messaging API

The messaging platform similar to facebook messenger is going to be its own app that connects to our site and allows users to message each other back and forth. All messages will be stored in a database and pulled as needed.

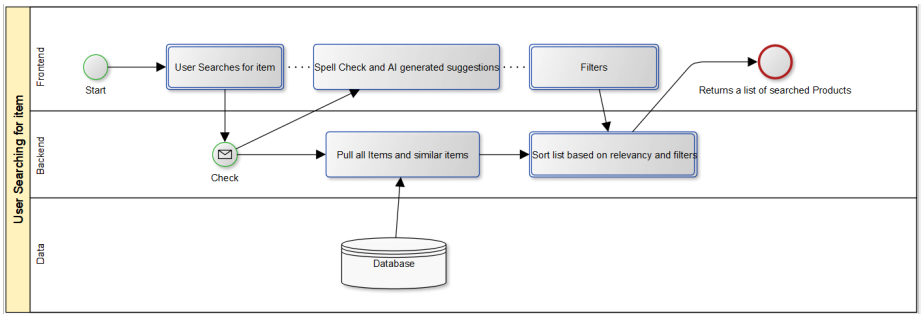
3 WorkFlows

3.1 Login Workflow



On the login the User will be prompted to put in their User name and Password into sign in. When they put in their username and password it is then checked with our database. If it is wrong then they are sent back to the beginning. If it was right then we will send an SMS message to the user with a pin for 2 factor authentication where the user will then have to put in the pin. Once the user has put in the pin if it is wrong then we will send the user another message if it is correct then we will allow the user access to the site.

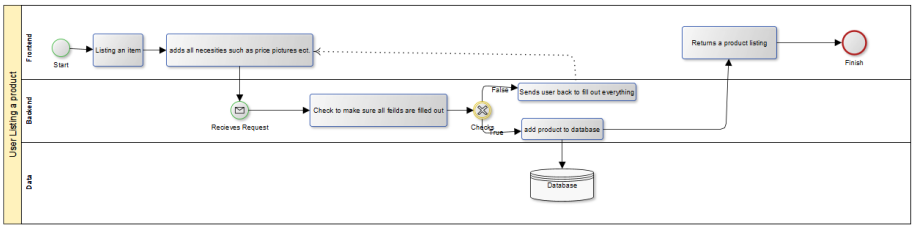
3.2 Searching Workflow



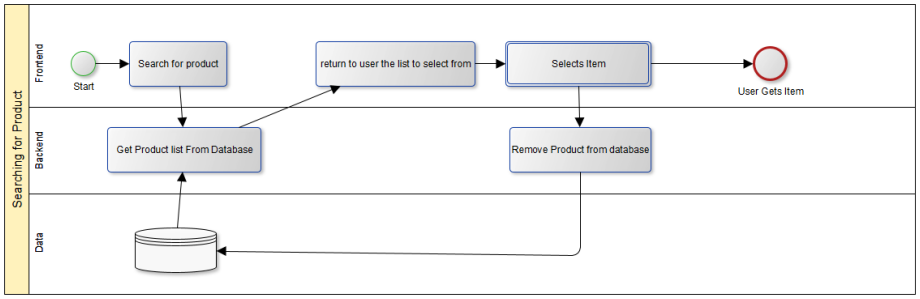
On the search we take the search the user has given us. While they are typing we give them AI generated suggestions as well as spellcheck checking

their spelling. We then take the search and likely will use a trie data structure and iterate down in to get all of the products we need from our database. We then will take all of those and then once we have the list we will check all of the ones against the filters if any and we will only return the ones based on the filtered results

3.3 Adding and Getting Product Workflow



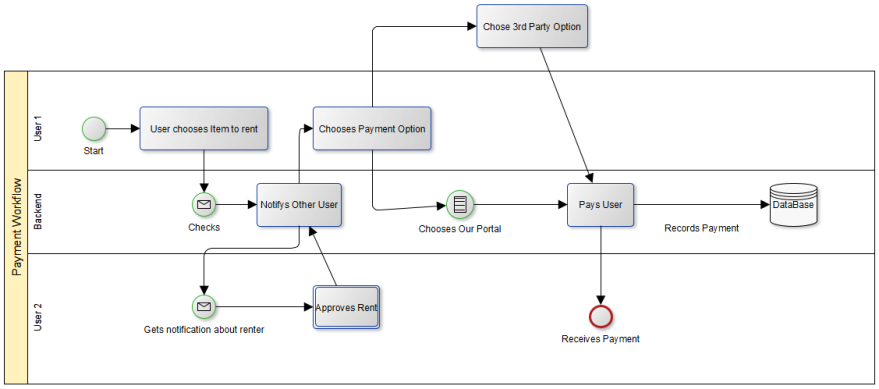
In the workflow above we have a user who wants to list a product. What happens is the user fills all the required feilds for posting an item. that will be images, price time and all of the other bells and whistles I mentioned earlier in features. We then check to make sure everything required was filled out If so we add it to our database and the item is then listed in the user profile page



In this workflow we have what it is like if a user wants to search for a product. the user Searches for a product and then gets the results list from our database.

Then the user looks through the list and selects the product that they want to use. We have the user select it and it then is removed from the database which stores availible items.

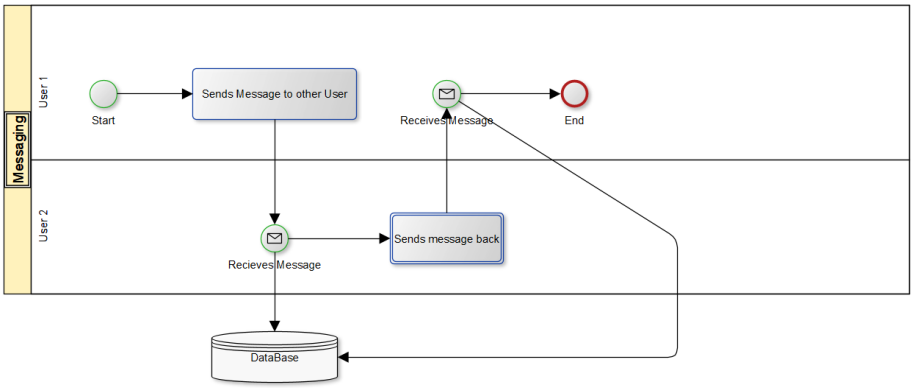
3.4 Payment Workflow



In the payment workflow first a user chooses an item to rent. the other user gets a notification that his item is desired to be rented when he gets that notification he can approve it and then once it is approved he can then get paid. it can go through our own portal or he can choose a 3rd party option. Once the user selects their option we record the payment and the other user gets paid.

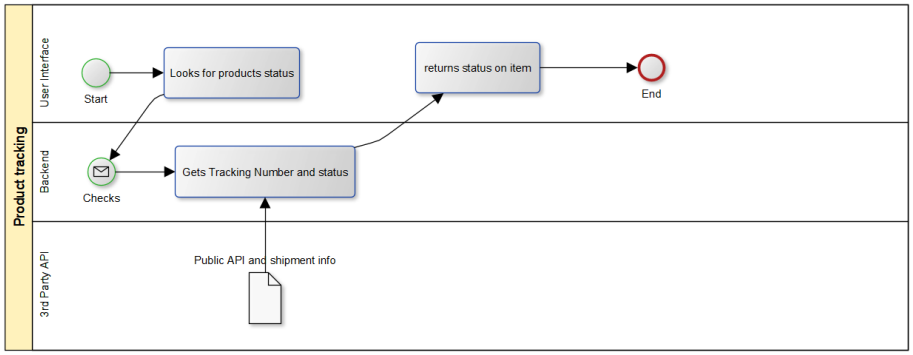
3.5 Messaging Workflow

In our messaging workflow users will be able to send each other messages back and forth as needed. One user will send an sms message to the other user. the other user can then reply and all chats are stored in a database and will be pulled up everytime the user logs into the messaging platform



3.6 Tracking workflow

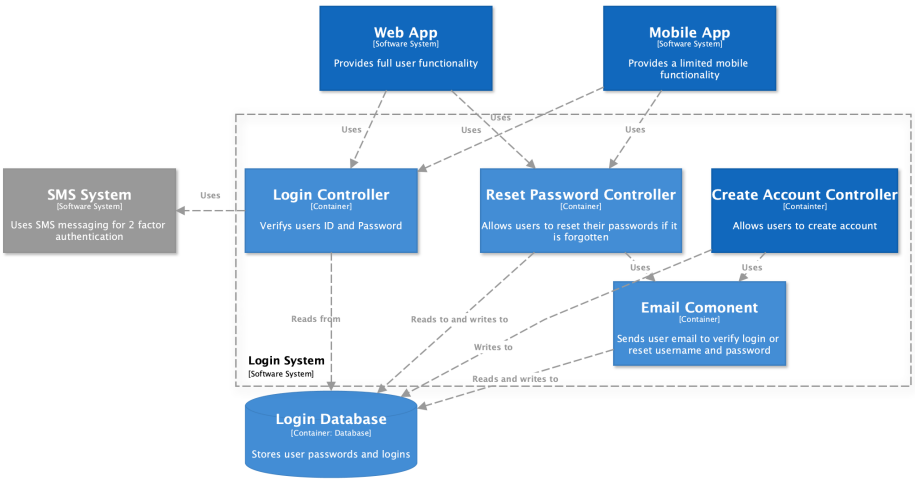
When a user wants to see where their item is in the event it is shipped then the user will check on our site or the 3rd party carrier's site. If he/she checks on our site we recieve the request we process it and we pull the info off of the 3rd party tracking info availible. If no info then we simply return that there isn't any info on the package.



4 Functional and Nonfunctional Requirements

4.1 Diagrams

4.1.1 Login API

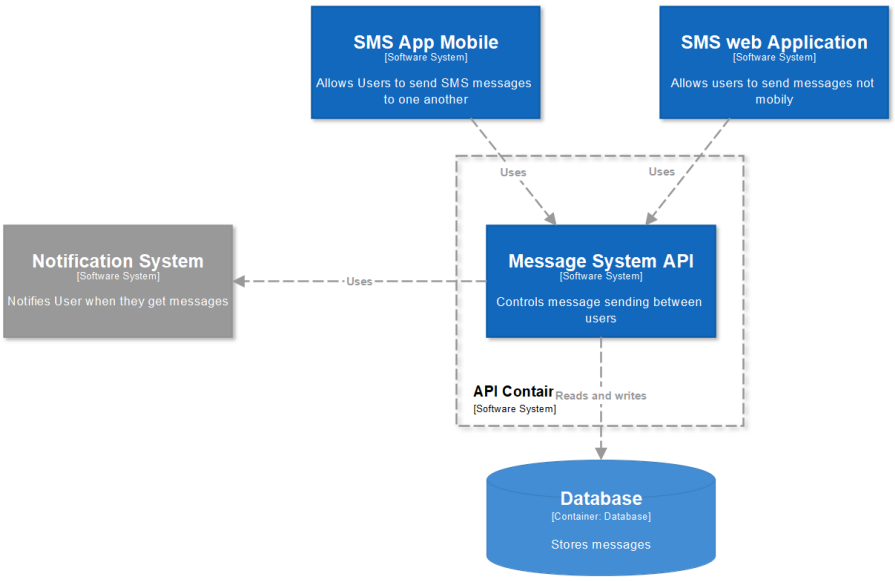


In our login API system, we have a few different options. The standard login will have the user login and when they login the controller will then pull the information from the database to see if the user data is correct. If it is then we will have the SMS system which will send the user a message on their phone with a pin. Then the user will need to input the right pin on the controller. Once the user has done that they are now able to access our site.

In the event that the user wants to reset or create an account, the user will then receive an email from our email component which will ask the user to put in a password. We can call it mean or we can call it Cybersecurity but the user will not be allowed to use one of the previous 5 passwords and it will be checked against our database. when the user changes the password it will be written to

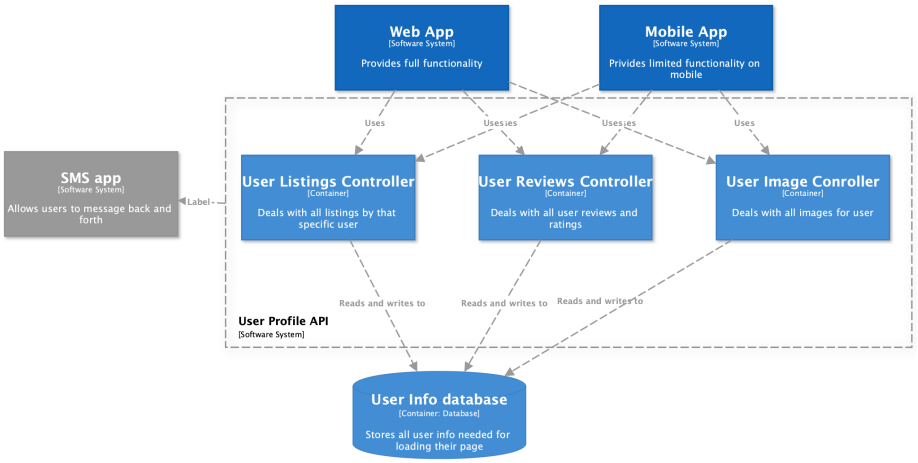
the database.

4.1.2 Messaging Platform API



In our system, we have our messages as a component that can be used on both mobile and web versions by calling the API system which connects to the notifications system and it stores the info into the database, and can pull past calls from that same database. The system is going to be an external app that will take you to as you need.

4.1.3 User Profiles API



For our user profiles controller, our system has 3 different controllers in them.

The User listings controller only deals with the listings that the user has stored and all of the information and organizes it allowing users to filter and allows users to add to their listings as well as remove them.

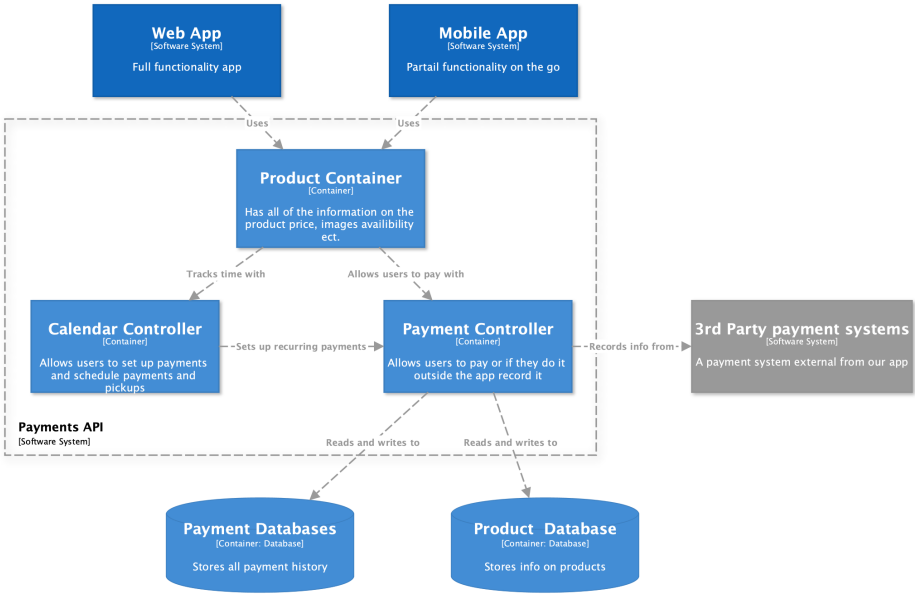
The reviews controller just deals with user reviews and allows other users to read and write to it. It will also allow filtering

The image controller will only deal with the images and match the image to the right products in the listings controller.

All of these are part of the same system and will then read and write to the same database.

The SMS system will be a part of the profile and will allow users to link to their messaging board and allow them to message the user directly.

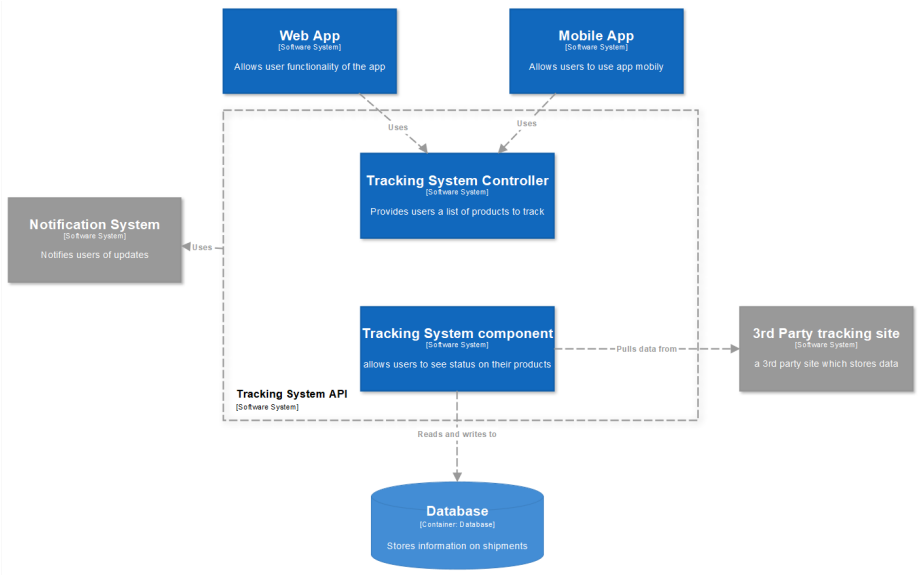
4.1.4 Payment Portal API



Our Payment Portal is going to have some options along with it. We have the Calendar functionality which enables users to set up recurring payments and setting up pickup dates and the time periods the item is rented for.

Once the user selects all of their options for what they want and when they wanted they are then taken to the online payment portal. The portal allows the user to pay from the app itself or go through a 3rd party and record the payment manually. Whatever way the payment is recorded the data is then stored in a database which is accessed only by the payment controller itself.

4.1.5 Tracking System API



In our tracking portion, we have a system that has the web and mobile app connected to our Tracking system's controller which connects to the system API. Our API pulls info available to us based on the updates from the carrier's site for us. Ideally, it will be with a key from the site. However, if there is no key then I will scrape the data needed. For example, UPS has a tracking number and when the item will arrive. We pull that data and use it to show our users the status and expected delivery of their item if it needed to be shipped.

It will update the status and will update the status for the user to see.

4.2 Deployment Architecture

The user-facing web application has been constructed employing ReactJS as the frontend framework with a Java backend. Communication between the frontend and backend is achieved through the use of a RESTful API, which is a standard approach for web application communication. Our chosen database management system is comprised of PostgreSQL, which serves as the primary

data store, with MongoDB being utilized as a supplementary backup. Careful consideration was given to the selection of these technologies to ensure optimal functionality and reliability within the application.

The web application frontend is deployed on a set of load-balanced servers to provide high availability and reduce downtime in case of hardware failure. The backend application, is also deployed on a separate set of nodes, with each service deployed in a separate container using Docker. To ensure coordination between the nodes, we use our own algorithm which is described in depth at the end of the paper.

To connect components, we use messaging protocols such as HTTP. For example, the frontend communicates with the backend using a RESTful API over HTTP, while the backend nodes communicate with each other using TCP for efficient and reliable communication. Messages between components are typically sent using standard message formats such as JSON.

4.3 Component APIs

(Note: On many of the APIs I used fake tokens and card numbers as examples to show how it would look so don't assume every example is a real token or card number)

4.3.1 Login APIs

User registration: When a user creates an account, they'll enter their email, phone number, and password. We'll store their email and hashed password securely in the database. We'll also generate a random code and send it to the user's email address to verify their email. We will use our email system component to send that email.

Endpoint: /user/create

Method: POST

Request: To create a user account, the client must send a POST request to the /user/create endpoint with the following parameters in the request body:

- **email (string):** The user's email address.
- **phone (string):** The user's phone number.
- **password (string):** The user's password.

Response: Upon successful creation of the user account, the API will respond with a status code of 201 (Created) and the following JSON object in the response body:

```
{
"status": "success",
"message": "User created successfully. Please check your email for verification code."
```

```
}
```

Endpoint: /user/verify

Method: GET

Request: To verify the user's email address, the client must send a GET request to the /user/verify endpoint with the following query parameters:

- **email (string):** The user's email address.
- **code (string):** The verification code sent to the user's email.

Response: Upon successful verification of the email, the API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
"status": "success",
"message": "Email verified successfully."
}
```

Endpoint: /user/login

Method: POST

Request: To login, the client must send a POST request to the /user/login endpoint with the following parameters in the request body:

- **email (string):** The user's email address.
- **password (string):** The user's password.

Response: If the login is successful, the API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
"status": "success",
```

```
"message": "User logged in successfully."
}
```

If the email or password is incorrect, the API will respond with a status code of 401 (Unauthorized) and the following JSON object in the response body:

```
{
"status": "error",
"message": "Invalid email or password."
}
```

Email verification: After the user receives the email, they'll click on a verification link that contains the verification code. The link will send a request to the backend server to verify the code. If the code is correct, we'll mark the user's email as verified in the database.

Endpoint: /verify-email

Method: GET

Request: When the user clicks on the verification link, the client must send a GET request to the /verify-email endpoint with the following query parameters:

- **email (string):** The user's email address.
- **code (string):** The verification code sent to the user's email.

Response: Upon successful verification of the email, the API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
```

```
"status": "success",
"message": "Email verified successfully."
}
```

If the verification code is incorrect or has expired, the API will respond with a status code of 401 (Unauthorized) and the following JSON object in the response body:

```
{
"status": "error",
"message": "Invalid or expired verification code."
}
```

If there is an error verifying the email for some other reason, the API will respond with a status code of 500 (Internal Server Error) and the following JSON object in the response body:

```
{
"status": "error",
"message": "An error occurred while verifying the email."
}
```

A login system with 2-factor authentication and SMS confirmation: we need to create a backend server that handles the user authentication process and integrates with a third-party SMS provider to send the confirmation code. Here's an outline of the steps involved:

User registration: When a user signs up, we need to store their details securely in our database. We'll use a hash function to store the user's password so that it can't be easily retrieved if the database is compromised. Additionally, we will have password requirements that include an uppercase letter a special

character a number 8 characters it can't be the users name and it will have to be deemed as a strong password.

Login request: When a user tries to log in, they'll enter their username and password on the client-side. The client will then send a login request to the backend server, which will verify the user's credentials.

First-factor authentication: To verify the user's password, we'll hash the password entered by the user and compare it with the hash stored in the database. If the hashes match, we'll consider the first factor of authentication to be successful.

SMS confirmation: If the first factor of authentication is successful, we'll generate a random code and send it to the user's phone number via SMS. We'll use a third-party SMS provider that we implement. The SMS should include a message that lets the user know that they are receiving a code to confirm their identity.

Second-factor authentication: After the user receives the SMS message, they'll enter the code into the login form on the client-side. The client will then send the code to the backend server, which will compare it to the code that was generated. If the codes match, we'll consider the second factor of authentication to be successful.

Token generation: If both factors of authentication are successful, we'll generate a JSON Web Token (JWT) and send it back to the client. The JWT will contain the user's ID and other relevant information. The client can then use this token to access protected resources on the server.

API: Token Generation

Endpoint: /generate-token

Method: POST

Request: When both factors of authentication are successful, the client must send a POST request to the /generate-token endpoint with the following request body parameters in JSON format:

- **email (string):** The user's email address.
- **password (string):** The user's password.

Response: Upon successful authentication, the API will respond with a status code of 200 (OK) and the following JSON object in the response body:


```
{
"status": "success",
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiZW1haWwiOiJhZG1pbkBleGFt
cGxlLmNvbSIsImlhdCI6MTYxNzM3MDg5MiwiZXhwIjoxNjE3MzczNzkyfQ.7aShcX-_rvmbif_ObItaYW94
ZTJiY7FOmz1deO7VlU",
"message": "Token generated successfully."
}
```


The **token** field contains the generated JWT, which the client can use to access protected resources on the server.

If the authentication fails due to invalid credentials, the API will respond with a status code of 401 (Unauthorized) and the following JSON object in the response body:


```
{
"status": "error",
"message": "Invalid email or password."
}
```


If there is an error generating the token for some other reason, the API will respond with a status code of 500 (Internal Server Error) and the following JSON object in the response body:

```
{
"status": "error",
"message": "An error occurred while generating the token."
}
```

Forgot password: If the user forgets their password, they can request a password reset by entering their email address on the client side. The client will then send a password reset request to the backend server, which will generate a random code and send it to the user's email. The email should include a message that lets the user know that they are receiving a code to reset their password.

Password reset: After the user receives the email, they'll click on a reset link that contains the reset code. The link will send a request to the backend server to verify the code. If the code is correct, we'll mark the reset code as used in the database and send a response back to the client-side to allow the user to enter a new password.

API: Forgot Password

Endpoint: /forgot-password

Method: POST

Request: When the user forgets their password, the client must send a POST request to the /forgot-password endpoint with the following request body parameter in JSON format:

- **email (string):** The user's email address.

Response: Upon receiving a valid email address, the API will send an email to the user's email address with a reset code. The API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
```

```
"status": "success",
"message": "Reset code sent to your email."
}
```

If the email address is not valid or does not exist in the database, the API will respond with a status code of 404 (Not Found) and the following JSON object in the response body:

```
{
"status": "error",
"message": "Invalid email address."
}
```

API: Password Reset

Endpoint: /reset-password

Method: POST

Request: When the user receives the email with the reset code, they must click on a reset link that sends a POST request to the /reset-password endpoint with the following request body parameters in JSON format:

- **email (string):** The user's email address.
- **resetCode (string):** The reset code received in the email.
- **newPassword (string):** The user's new password.

Response: Upon receiving a valid email address and reset code, the API will verify the code and mark it as used in the database. The API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
"status": "success",
```

```
"message": "Password reset successfully."
}
```

If the reset code is not valid or has already been used, the API will respond with a status code of 400 (Bad Request) and the following JSON object in the response body:

```
{
"status": "error",
"message": "Invalid reset code."
}
```

If there is an error resetting the password for some other reason, the API will respond with a status code of 500 (Internal Server Error) and the following JSON object in the response body:

```
{
"status": "error",
"message": "An error occurred while resetting the password."
}
```

4.3.2 Searching API

Collect and preprocess data: We'll start by collecting all the data we want to search and preprocess it by tokenizing, stemming, and removing stop words. The text is broken down into individual words or tokens. Then our system will be reducing each word to its base form by removing suffixes and prefixes. Then it will be removing common words like "the", "and", and "of" that are unlikely to be useful in a search query.

Collect and preprocess data:

We will begin by collecting all the data we want to search and preprocess it by performing the following steps:

1. **Tokenizing:** The text will be broken down into individual words or tokens.

2. **Stemming:** Our system will reduce each word to its base form by removing suffixes and prefixes.

3. **Removing stop words:** We will remove common words like "the", "and", and "of" that are unlikely to be useful in a search query.

Build an inverted index: We'll build an inverted index to map each term in the preprocessed data to the documents that contain it. We'll also store the frequency of each term in each document. The inverted index allows us to quickly find documents that contain a particular term and rank them by their relevance to a search query.

Build an Inverted Index

Request:

```
POST /inverted_index
Content-Type: application/json

{
"data": [
{
"id": "document1",
"text": "This is the text of document 1"
},
{
"id": "document2",
```

```
"text": "This is the text of document 2"
}
]
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
"message": "Inverted index created successfully."
}
```

Description:

This API is responsible for building an inverted index, which maps each term in the preprocessed data to the documents that contain it. It also stores the frequency of each term in each document. The inverted index allows for quick retrieval of documents that contain a particular term and enables ranking of documents by their relevance to a search query.

Handle Search Requests

Request:

```
GET /search?q=query
Content-Type: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
{
"results": [
{
"id": "document1",
"title": "Document 1",
"snippet": "This is the text of document 1"
},
{
"id": "document2",
"title": "Document 2",
"snippet": "This is the text of document 2"
}
]
}
```

Description:

This API is responsible for handling search requests. When a user enters a search query, the client sends a GET request to the backend server with the query. The backend server preprocesses the query and uses the inverted index to retrieve a list of documents that contain the query terms. The documents are ranked by their relevance to the query, and the search results are presented to the user.

Handle Errors

Request:

None

Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
```

```
{
"error": "Invalid request."
}

HTTP/1.1 404 Not Found
Content-Type: application/json

{
"error": "Resource not found."
}
```

Our system will be using a Trie and will iterate down the trie in order to find the necessary information. We do that by:

Traversing the trie: To retrieve information from a trie, start at the root node and traverse the trie using the characters in the search key. If the node does not have a child node that corresponds to the next character in the search key, then the search has failed and you can return a failure result or that no was info with that result.

Checking for termination: Once we have traversed the trie to the end of the search key, you need to check whether the node you have reached represents a complete word or not. If the node represents a complete word, then you can return the information associated with that word. That word might include its definition, synonyms, and antonyms.

Handling partial matches: If the node you have reached does not represent a complete word, but the search key has been fully traversed, then we will return a list of all the words in the trie that start with the search key. This is useful for auto-complete or suggestion features. To do this, we traverse the subtree rooted at the current node and collect all the words that are represented by nodes in

that subtree. Then return this list of words as a result.

Handling wildcards: Our implementation supports the use of wildcards, which allow you to search for words that match a certain pattern. To handle these wildcards, we use a recursive algorithm that traverses all the child nodes of the current node, checking each one to see if it matches the next character in the search key. If the next character is a wildcard, we recursively traverse all the child nodes and collect the words that are represented by nodes in the subtree rooted at each child node.

Handle search requests: When a user enters a search query, the client will send a request to the backend server with the query. The backend server will then use the inverted index to search for documents that match the query. This involves preprocessing the query and using the inverted index to retrieve a list of documents that contain the query terms. We'll then rank the documents by their relevance to the query.

Present search results: Finally, we'll present the search results to the user. Our system displays a list of documents that match the query, along with some summary information like the title and snippet of each document. We may also allow the user to filter or sort the search results based on various criteria.

Request:

- Endpoint: /search
- Method: GET
- Parameters:
  - **query** - the search query entered by the user
  - **sort** (optional) - the sorting criteria for the search results, e.g., relevance, date, etc.

- **filter** (optional) - the filtering criteria for the search results, e.g., by date, category, author, etc.

Response:

- Status Code: 200 OK
- Content Type: application/json
- Body:
  - **results** - an array of objects representing the search results, where each object contains:
    - **title** - the title of the document
    - **snippet** - a short summary of the document content
    - **url** - the URL of the document
    - **date** - the date the document was created or last updated
    - **score** - the relevance score of the document to the search query
    - **category** (optional) - the category of the document
    - **author** (optional) - the author of the document
  - **total** - the total number of documents that match the search query

4.3.3 Messaging API

Secure communication: To ensure secure communication between clients and the messaging platform, HTTPS protocol will be used. We use HTTPS because it is a secure communication protocol that uses SSL/TLS encryption to protect messages from unauthorized access.

Message delivery: To ensure reliable message delivery, the messaging platform will implement message queuing which allows messages to be stored temporarily on the messaging server until they can be delivered to the recipient.

This ensures that messages are not lost or delayed due to network issues or other problems.

Endpoints:

POST /message - Sends a message to the recipient.

Request Parameters:

- **recipient**: The recipient of the message.
- **message**: The content of the message.

Response:

- **200 OK**: The message was successfully queued for delivery.
- **400 Bad Request**: The request was invalid or missing required parameters.
- **500 Internal Server Error**: There was an error on the server side.

POST /message-status - Checks the status of a message.

Request Parameters:

- **message-id**: The ID of the message to check the status of.

Response:

- **200 OK**: The message is still in the queue and has not been delivered yet.
- **404 Not Found**: The message with the given ID could not be found.
- **500 Internal Server Error**: There was an error on the server side.

Message storage: Messages will be stored in a database for easy retrieval by users. The database will be MySQL.

Message searching: To enable easy message searching, you can use a search engine. The search engine can index messages based on various criteria such as

sender, recipient, message content, and time. This enables users to search for messages using natural language queries.

Request: GET /messages/search

Parameters:

- **q**: The search query. Can be a natural language query or a set of keywords.
- **sender**: Optional parameter to search for messages sent by a particular sender.
- **recipient**: Optional parameter to search for messages received by a particular recipient.
- **start**: Optional parameter to specify the start date for the search (in ISO format: YYYY-MM-DD).
- **end**: Optional parameter to specify the end date for the search (in ISO format: YYYY-MM-DD).
- **sort**: Optional parameter to specify the sorting criteria (e.g. by date, by relevance, etc.).
- **limit**: Optional parameter to specify the maximum number of results to return.

Response:

- **200 OK**: The search was successful. Returns a list of messages that match the search criteria. Each message includes the sender, recipient, timestamp, and message content.
- **400 Bad Request**: The search query was invalid or missing required parameters.

- **404 Not Found**: No messages were found that match the search criteria.
- **500 Internal Server Error**: An error occurred on the server while processing the search request.

Message notifications: Real-time notifications can be implemented using WebSockets or Push Notifications. Which enables bidirectional communication between clients and the server, allowing the server to push notifications to clients in real-time. Push Notifications allow the messaging platform to send notifications to clients even when the client is not actively using the platform.

4.3.4 Payment API

Encryption: All payment information transmitted through the API is encrypted using industry-standard encryption protocols to prevent unauthorized access.

Requests:

- **POST /payment** - Sends payment information to the API for processing.
  - **Parameters:** Payment information including credit card number, expiration date, and security code.
  - **Headers:** None.
  - **Request Body:** Encrypted payment information.

Responses:

- **200 OK** - Payment processed successfully.
  - **Headers:** None.
  - **Response Body:** None.
- **401 Unauthorized** - Payment information could not be authenticated.
  - **Headers:** None.

– **Response Body:** Error message indicating that the payment information is invalid.

Tokenization: The Payment API uses tokenization to store payment information securely. Tokenization replaces sensitive payment information, such as credit card numbers, with a unique identifier called a token, which can be used to process payments without exposing the actual payment information. This one's API will be more or less the same as the previous tokenization request above.

Compliance: The Payment API will comply with various regulatory standards, such as PCI-DSS, to ensure the security and privacy of payment information.

Fraud Prevention: The Payment API may incorporate various fraud prevention measures, such as address verification, card verification codes, and other risk assessments, to prevent fraudulent transactions.

Requests:

GET /fraud-prevention

Description: Returns a list of fraud prevention measures implemented by the Payment API.

Response:

HTTP Status Code: 200 OK

Body: A JSON object containing a list of fraud prevention measures, such as address verification, card verification codes, and other risk assessments.

- "fraud-prevention-measures": ["address-verification", "card-verification-code", "risk-assessment"]

[language=json,firstnumber=1]

POST /fraud-prevention/verify-address

Description: Verifies the address of a user during a transaction.

Request:

HTTP Headers:

- **Content-Type:** application/json
- **Authorization:** Bearer ¡access token¿

Body: A JSON object containing the user's address information.

- "name": "John Doe", "address": "123 Main St", "city": "San Francisco", "state": "CA", "zip": "94111", "country": "USA"

Response:

HTTP Status Code: 200 OK

Body: A JSON object containing the result of the address verification.

```
{
"result": "verified"
}
```

POST /fraud-prevention/verify-card

Description: Verifies the card information of a user during a transaction.

Request:

HTTP Headers:

- **Content-Type:** application/json
- **Authorization:** Bearer ¡access token¿

Body: A JSON object containing the user's card information.

```
{
"card-number": "4111111111111111",
```

```
"expiration-date": "12/23",
"cvv": "123"
}
```


Response:

HTTP Status Code: 200 OK

Body: A JSON object containing the result of the card verification.


```
{
"result": "verified"
}
```


3rd Party payments: The user will be able to record a third-party payment and will be able to record how much and how they paid

Outside Platforms: Our system is going to also allow for a paypal use. We will use their API to allow purchases.

Storage: all of these transaction will be stored in a mongoDB database.

4.3.5 Tracking API

If the shipping carrier provides an API, the tracking service can retrieve the package tracking information directly from the API. The API typically requires an authentication key to access the data, which the tracking service can obtain by registering with the shipping carrier. Once authenticated, the tracking service can make requests to the API to retrieve the package tracking information.

If the shipping carrier does not provide an API, the tracking service will use web scraping techniques to extract the tracking information from the carrier's website. The tracking service controller will use web scraping libraries such as Beautiful Soup or Scrapy to automate this process.

Once the tracking service has retrieved the package tracking information, it will store it in a database and provides it to users through a user interface.

The tracking information will include the package status, the location of the package, the estimated delivery time, and any relevant updates or notifications.

Requests:

```
POST /payment HTTP/1.1
Host: example.com
Content-Type: application/json

{
"card_number": "1234 5678 9012 3456",
"expiration_date": "12/23",
"security_code": "123",
"amount": 50.00
}
```

Responses:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
"status": "success",
"message": "Payment processed successfully."
}

HTTP/1.1 401 Unauthorized
Content-Type: application/json

{
"status": "error",
```

"message": "Payment information could not be authenticated."
}

4.4 Descriptions and rigorous Explanations

When storing my data I am going to use a couple of techniques for storing my data efficiently. I am planning on using sharding to make sure my data is stored and can be accessed as quickly as possible.

As TigerTech.net recommends I am not going to allow more than 1 GB of data on each database so therefore I am going to need many databases.

The user profile information will contain The name of the User, the description information, the reviews on that user, the hashes of their listed products, the profile picture, the location of the user, and will be ready for a system to present to the user.

If a user is selected the system will pull the product information from the different databases which will contain the products listed. The product stored will contain the images, and product information and will contain anything along with the product that needs to be displayed.

All of the information will be stored across many databases by using Hash-Based sharding. We will do that by:

4.4.1 Data Sharding

Firstly the system determines the sharding key. This is the attribute or column that will be used to divide the data into smaller subsets. The system will use user ID as the sharding key.

In the next step, our system determines the number of shards that will be used to store the data. Each database will only contain a maximum of 1 GB so in order to maintain that once the system hits 700 MB the system will no

longer add any new information from other users as a buffer. Any new profiles will be hashed to a different Database.

Once the sharding key and number of shards are determined, our system will create the shards. Each shard will have a unique identifier, a number that we get from hashing the user ID, and will be hosted on a database instance. The data from the original dataset is then partitioned based on the sharding key and distributed across the shards.

Now that the shards are created and the sharding strategy is implemented, the next step is to implement a routing layer. This layer acts as a proxy between the application and the shards, and is responsible for directing queries to the appropriate shard based on the sharding key. The routing is similar to adding information. We hash to the database by the user ID and once we have done that we then can pull the data from the database we hashed to and return it to the user.

Payment History is stored within the user profile database and will be accessed the same way as any other information that we pull from their profile.

The way the storage system is going to look is that we are going to have a few managing servers that will be selected by the other servers that will take the request from the client and will then calculate the hash of that information location. It will then go to the correct database in the collection and will make a call for that information off of that server. It will then get the information from that database and return it to the client who made the request.

4.4.2 Backup Storage

cases I am concerned with is 1. a server is down 2. corrupted data. If a server is down and a database can't access that server I am going to make sure to have a solution to such a problem. The strategies I am going to use is going to be to

have redundancy, load balancers, and backup data stored in a low-cost storage service for emergencies. For the redundancy, I am going to have a 2nd database copy for each one in order to pull info from. the same hash is going to take you to 2 what are supposed to be the same tables and will compare them. In order to keep them in sync I will have another hashtable that will take you to a MongoDB database instead of an SQL database. The same hash will take you to both so when you are pulling an item from the main SQL database If there is an issue it will then need to look at the backup storage and once the main database is back online then we will update whichever db was down with the other database.

4.4.3 Load Balancer

For my load balancer, I am going to have 3 major different types of nodes. One node is going to be a leader node, one node is a manager node and another node is a worker node.

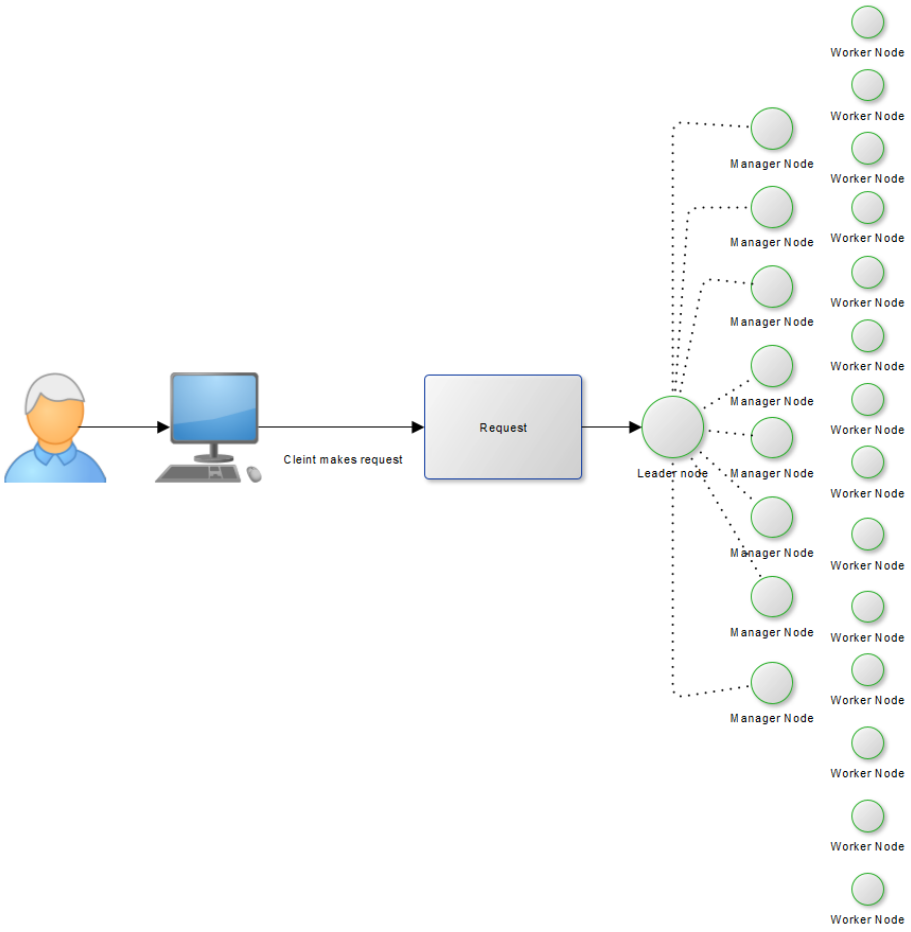
The Leader node's role is to take the request from the client and return it to the client. This node is the only node that receives and sends from the client. This node takes the request from the client and immediately pushes it to a managing node round-robin style.

The Manager's role with the node is to assign it to the right worker. The manager selects an IP address from the list based on the geographical location of the client. This is done by checking the client's IP address against a database of IP address ranges and their associated geographical locations. Once we have the location of where the request is coming from along with the work request given we can assign it to the right worker. So we put it in a queue that holds the work that needs to be done and given back to the client. Then we find a

worker that is currently free and we send it that work. Once we receive that work back from the worker we are then able to take it off the queue of our given work and then we can send it back to the leader who will then send it back to the client.

The Worker's role is to fulfill the request from the manager the worker receives work and then adds the work that needs to get done into a ConcurrentLinkedQueue,it will use a locking mechanism to prevent multiple threads from accessing the same data or resource simultaneously. For example, the Worker component may use a synchronized block to ensure that only one thread at a time can access the ConcurrentLinkedQueue we will do that by putting in a synchrnized block. The node will then deal with as many requests as the machine can handle at a time using the equation: Number of threads = Number of Available Cores * (1 + Wait time / Service time); Waiting time is the time spent waiting for IO bound tasks to complete. Service time is the time spent being busy, say processing the HTTP response, marshaling/unmarshaling, any other transformations etc. Wait time / Service time - this ratio is often called the blocking coefficient. (The equation is from: `https://engineering.zalando.com/posts/2019/04/how-to-set-an-ideal-thread-pool-size.html`) Workers will mostly be hashing to the right database to pull the right information to send back to the individual.

Below is a small visual of how the system will look.



Quick explanation: The User sends a request, the leader gets the request, the leader sends the request to a manager, the manager selects the right worker and sends it the work. The worker does the work and sends it back to the manager who sends it back to the leader who sends it back to the client.

Selecting leader and managers. At startup I am going to make 1 out of every certain number of nodes a manager. I still haven't decided on the exact number but for now we can say 1 in every 100 is going to be a manager. As far

as leaders go we will do a zookeeper election amongst the managers and that is how we decide the leader there.

4.4.4 Fault tolerance

We will start by having an election to determine who our leading node is.

Nodes in the cluster connect to each other and form a group. Each node starts by creating a unique ephemeral node in a designated ZooKeeper directory.

We do this by attempting to create a new sequential node in the election directory. This node will represent the node's candidacy for becoming a leader.

The node that successfully creates the lowest-numbered sequential node becomes the leader. The other nodes then watch this node and wait for it to fail.

If the current leader fails or disconnects from the cluster, the next lowest-numbered node becomes the new leader. The algorithm ensures that only one node becomes the leader at any given time.

If multiple nodes try to create a node with the same sequential number, ZooKeeper uses the node's ID to break the tie. Nodes with lower IDs win the tie.

If a node fails, the ephemeral node it created is automatically deleted by ZooKeeper. This signals to the other nodes that the node is no longer a candidate for leader.

If a node disconnects from the cluster without properly deleting its ephemeral node, ZooKeeper detects this and removes the node from the leader election process. This ensures that the node cannot become a leader in the future.

The leader node periodically sends heartbeats to the other nodes to signal that it is still alive and functioning as the leader. If the leader fails to send a

heartbeat within a certain time window, the other nodes will start a new leader election.

For ZooKeeper leader election, the messaging protocol used is Apache ZooKeeper's own protocol, which is based on the ZooKeeper ensemble's consensus algorithm, known as ZAB (ZooKeeper Atomic Broadcast). ZAB ensures that only one server in the ZooKeeper ensemble is the leader at any given time, and that all the other servers synchronize their state with the leader. The leader election process involves the use of ZAB to achieve consensus among the ensemble members and elect a new leader in the event that the current leader fails or goes offline.

The heartbeat protocol is going to work as follows:

Each node in the system maintains a list of its peers in the cluster.

Each node also maintains a timer that is used to send heartbeat messages to its peers periodically. It will be every 5 seconds.

When the timer expires, the node sends a heartbeat message to each of its peers. The message will be a ping message that requests a response from the recipient.

When a node receives a heartbeat message from one of its peers, it sends a response message back to signal that it is still alive and responsive.

If a node does not receive a heartbeat message from one of its peers within a certain time window we say 20 seconds it assumes that the peer has failed or become unresponsive. We then remove the peer from its list of active peers.

To ensure that the heartbeat messages do not overload the network or the receiving nodes, the messages will be sent at a lower priority level than other types of messages in the system. This allows the system to continue operating smoothly even under heavy loads.

For the Heartbeat as well as the messaging between leader nodes and worker

nodes the message protocol used is going to be TCP because Using TCP ensures that the heartbeat signal is delivered reliably and in the correct order. TCP provides error checking, retransmission of lost packets, and flow control, which helps to prevent congestion and ensure that the data is transmitted at an appropriate rate.

In addition, TCP is widely supported by networking infrastructure and operating systems, making it a standard choice for many network applications. It is also efficient and scalable, making it suitable for use in high-performance and large-scale systems.

Overall, TCP's reliability and wide availability make it a suitable choice for the heartbeat algorithm, where ensuring connectivity and detecting failures in a timely and accurate manner is critical.

Case: Leader fails, then we do a new election and elect a new leader. All the work the leader has received fails and the clients will need to send it again to get their results.

Case: Manager fails, Leader marks a manager failed if it hasn't heard from it in a certain amount of time. If the manager is marked as failed all work out from it will be gone and the leader will resend the work to a different manager to assign the work.

Case: Worker fails, manager will simply send another worker the work and that worker will be ignored.

4.4.5 Brief Recap

So quick recap of the algorithms I am using. I am using a DNS load balancing structure for manager-to-worker load balancing and I am using a round-robin load balancing method from leader to manager. I am using a zookeeper election to get leaders. I am using heart beat to detect faults.

5 Citations

5.1 Helpful Websites I used

1. TigerTech.net: https://support.tigertech.net/mysql-size: :text=A

2. AWS website: https://aws.amazon.com/free/?trk=fce796e8-4ceb-48e0-9767-89f7873fac3dsc$_c hannel = pss_k wcid = AL!4422!3!432339156150!e!!g!!awsef_i d = CjwKCAjw5dqgBhBNEiwA7PryaO_H vxpWGhp1LhpK3xJAtvk80zwmxJwDUXDHklOMREn0M0b_6 W e cQAvD_B wE : G : ss_k wcid = AL!4422!3!432339156150!e!!g!!awsall - free - tier.sort-by = item.additionalFields.SortRankall-free-tier.sort-order = ascawsf.Free$

3.$DNSLoadBalancingexplantaion : https : //avinetworks.com/glossary/dns-load - balancing$

4.$http : //highscalability.com$

5.$https : //www.apache.org$

6.$https : //www.allthingsdistributed.com$

7.$https : //blog.acolyer.org$

5.2 From ChatGPT

5.2.1 1st prompt

My prompt: Come up with an idea for a new internet-scale service

Certainly, here are some more ideas for an internet-scale service:

Virtual language translation service: An online service that provides real-time language translation for individuals and businesses. It could provide translation services for text, voice, and video, as well as connect users with professional translators for more complex translations.

Virtual interior design marketplace: An online marketplace that connects homeowners with interior designers for virtual consultations and design services. It could provide tools for visualizing and testing different design options, as well as connect users with affordable and vetted designers.

Online platform for peer-to-peer car rentals: An online platform that allows car owners to rent out their cars to other individuals for short-term use. It could provide tools for scheduling rentals, insurance coverage, and payment processing, as well as incentivize car owners through competitive rental rates.

Virtual home renovation service: An online service that helps homeowners plan and execute home renovation projects remotely. It could provide tools for visualizing and testing different design options, as well as connect homeowners with professional contractors for more complex renovations.

Online platform for personalized nutrition plans: An online platform that helps users create personalized nutrition plans based on their health goals, preferences, and dietary restrictions. It could provide tools for tracking food intake, calculating calorie intake, and connecting users with registered dietitians for consultation and support.

I hope these ideas inspire you and help you generate even more ideas for your project!

5.2.2 2nd Prompt

My Prompt: what are some features you would want in an app for renting out products

As an AI language model, I don't have personal preferences, but here are some features that could be useful in an app for renting out products:

User-friendly interface: The app should have a simple and intuitive interface that makes it easy for users to navigate and find what they're looking for.

Secure payment system: The app should have a secure payment system that allows users to make transactions without worrying about their financial information being compromised.

Inventory management: The app should allow users to easily manage their inventory and keep track of which items are available for rent.

Scheduling and calendar integration: The app should allow users to schedule rentals and integrate with their calendar so that they can keep track of their rental schedule.

Communication tools: The app should have messaging or chat features that allow renters and owners to communicate with each other to coordinate rental details.

Reviews and ratings: The app should allow users to leave reviews and ratings for both the products and the renters, which can help build trust and confidence in the community.

Insurance coverage: The app should provide insurance coverage to protect renters and owners from any damages or losses that may occur during the rental period.

Notifications: The app should send notifications to users when items become available or when rental requests are accepted or declined.

Geolocation and mapping: The app should use geolocation and mapping technology to allow users to find rental items that are near them.

Integration with social media: The app should integrate with social media platforms to allow users to share their rental experiences and promote the app to their friends and followers.

5.2.3 3rd Prompt

My prompt: What is a good api for me to use if I want to add a search feature in my app with filters similar to ebays search feature

There are several APIs that you can consider using to add a search feature with filters similar to eBay's search feature. Here are a few options:

Elasticsearch: Elasticsearch is an open-source search and analytics engine that you can use to build powerful search features with filters. It's a popular choice for building search features because of its flexibility and scalability. Elasticsearch also has a robust API that you can use to build your search feature.

Algolia: Algolia is a hosted search API that you can use to build fast and relevant search features with filters. Algolia offers several features, such as typo tolerance, geo-search, and multi-language support, that make it easy to build a search feature that meets your specific needs.

Solr: Solr is another open-source search platform that you can use to build a search feature with filters. Solr is built on Apache Lucene, a powerful and popular search engine library. Solr has a rich set of features, including faceting and filtering, that you can use to build your search feature.

Amazon CloudSearch: Amazon CloudSearch is a fully-managed search service that you can use to build a search feature with filters. CloudSearch provides an easy-to-use API that you can use to add search capabilities to your application. It also integrates well with other Amazon Web Services, such as Amazon S3 and Amazon EC2.

Each of these APIs has its own strengths and weaknesses, so you should

choose the one that best meets your specific needs.

5.2.4 4th Prompt

My Prompt: best API for storing a large amount of data

The best API for storing a large amount of data largely depends on the specific requirements of your use case. Here are a few popular APIs for storing large amounts of data:

Amazon S3: Amazon S3 (Simple Storage Service) is a popular cloud-based storage solution that can be used to store and retrieve any amount of data from anywhere on the web. S3 is designed to be highly scalable, durable, and available, making it a great option for storing large amounts of data. S3 also offers a range of features such as versioning, access control, and lifecycle management.

Google Cloud Storage: Google Cloud Storage is a cloud-based storage service that provides object storage for unstructured data. Like S3, it is designed to be highly scalable and durable, and offers features such as versioning, access control, and lifecycle management. Google Cloud Storage also integrates well with other Google Cloud services.

Azure Blob Storage: Azure Blob Storage is a cloud-based storage solution offered by Microsoft Azure. It provides scalable object storage for unstructured data and offers features such as access control, versioning, and lifecycle management. Azure Blob Storage also integrates well with other Azure services.

MongoDB: MongoDB is a popular NoSQL database that can be used to store large amounts of structured and unstructured data. It is designed to be highly scalable and can handle large amounts of data with ease. MongoDB offers features such as automatic sharding, replication, and dynamic queries.

Each of these APIs offers a different set of features and pricing plans, so it's important to evaluate your specific needs and requirements to choose the best

one for your use case.

5.2.5 5th Prompt

My Prompt: Give me a good api for making a messaging platform similar to facebook messenger

There are several APIs that you can consider using to build a messaging platform similar to Facebook Messenger. Here are a few options:

Twilio: Twilio is a cloud communications platform that provides APIs for building messaging, voice, and video applications. Twilio's Programmable Messaging API allows you to build messaging features into your application using SMS, MMS, and chat messaging. Twilio also provides features such as message history, delivery status tracking, and rich media support.

SendBird: SendBird is a chat and messaging API that allows you to build messaging features into your application. SendBird provides a range of features such as chat rooms, channels, message threading, and push notifications. It also offers SDKs for popular programming languages and platforms.

Layer: Layer is a messaging and chat API that provides features such as real-time messaging, push notifications, and conversation history. Layer also offers customizable UI components that can be integrated into your application, making it easy to build a messaging platform with a polished user interface.

Firebase: Firebase is a mobile and web application development platform that provides a range of features, including a real-time messaging API. Firebase's Cloud Messaging API allows you to build messaging features into your application using push notifications. Firebase also offers features such as offline data synchronization and user authentication.

Each of these APIs has its own strengths and weaknesses, so you should choose the one that best meets your specific needs. It's important to consider

factors such as pricing, scalability, and ease of integration when evaluating messaging APIs.

5.2.6 6th prompt

my prompt: Write me a rigirous backend implemetation for a login system with 2 factor authentication that sends a message to via sms to confirm

To implement a login system with 2 factor authentication and SMS confirmation, we need to create a backend server that handles the user authentication process and integrates with a third-party SMS provider to send the confirmation code.

Here's an outline of the steps involved:

User registration: When a user signs up, we need to store their details securely in a database. We'll use a hash function to store the user's password so that it can't be easily retrieved if the database is compromised.

Login request: When a user tries to log in, they'll enter their username and password on the client-side. The client will then send a login request to the backend server, which will verify the user's credentials.

First-factor authentication: To verify the user's password, we'll hash the password entered by the user and compare it with the hash stored in the database. If the hashes match, we'll consider the first factor of authentication to be successful.

SMS confirmation: If the first factor of authentication is successful, we'll generate a random code and send it to the user's phone number via SMS. We'll use a third-party SMS provider such as Twilio to send the message. The SMS should include a message that lets the user know that they are receiving a code to confirm their identity.

Second-factor authentication: After the user receives the SMS message, they'll enter the code into the login form on the client-side. The client will

then send the code to the backend server, which will compare it to the code that was generated. If the codes match, we'll consider the second factor of authentication to be successful.

Token generation: If both factors of authentication are successful, we'll generate a JSON Web Token (JWT) and send it back to the client. The JWT will contain the user's ID and other relevant information. The client can then use this token to access protected resources on the server.

Here's a sample implementation in Python using Flask and the Twilio API:

5.2.7 7th Prompt

My prompt: Tell me an in depth walkthrough of getting information from a trie

Sure, here's an in-depth walkthrough of getting information from a trie:

A trie is a tree-like data structure that is used for efficient string search operations. It is particularly useful for searching for strings in a large collection of data, such as a dictionary or a set of web pages. Here's how you can get information from a trie:

Traverse the trie: To retrieve information from a trie, you need to start at the root node and traverse the trie using the characters in the search key. For example, if you are searching for the string "cat", you would start at the root node and follow the path that corresponds to the letters "c", "a", and "t". If a node does not have a child node that corresponds to the next character in the search key, then the search has failed and you can return a failure result.

Check for termination: Once you have traversed the trie to the end of the search key, you need to check whether the node you have reached represents a complete word or not. If the node represents a complete word, then you can return the information associated with that word. For example, if you are

searching for a dictionary entry for the word "cat", the information associated with that word might include its definition, synonyms, and antonyms.

Handle partial matches: If the node you have reached does not represent a complete word, but the search key has been fully traversed, then you may want to return a list of all the words in the trie that start with the search key. This is useful for auto-complete or suggestion features. To do this, you can traverse the subtree rooted at the current node and collect all the words that are represented by nodes in that subtree. You can then return this list of words as a result.

Handle wildcards: Some trie implementations support the use of wildcards, which allow you to search for words that match a certain pattern. For example, you might search for all words that start with "c" and end with "t". To handle wildcards, you can use a recursive algorithm that traverses all the child nodes of the current node, checking each one to see if it matches the next character in the search key. If the next character is a wildcard, you can recursively traverse all the child nodes and collect the words that are represented by nodes in the subtree rooted at each child node.

Overall, getting information from a trie involves traversing the trie using the characters in the search key, checking for termination to determine whether a complete word has been found, and handling partial matches and wildcards as needed.

5.2.8 8th Prompt

My prompt: Write me a rigirous backend implemetation for a searching system

Sure, here's an explanation of a search system from scratch:

Collect and preprocess data: We'll start by collecting all the data we want to search and preprocess it by tokenizing, stemming, and removing stop words. Tokenization involves breaking down the text into individual words or tokens. Stemming involves reducing each word to its base form by removing suffixes

and prefixes. Stop word removal involves removing common words like "the", "and", and "of" that are unlikely to be useful in a search query.

Build an inverted index: We'll build an inverted index to map each term in the preprocessed data to the documents that contain it. We'll also store the frequency of each term in each document. The inverted index allows us to quickly find documents that contain a particular term and to rank them by their relevance to a search query.

Handle search requests: When a user enters a search query, the client will send a request to the backend server with the query. The backend server will then use the inverted index to search for documents that match the query. This involves preprocessing the query and using the inverted index to retrieve a list of documents that contain the query terms. We'll then rank the documents by their relevance to the query, which is typically done using a scoring function like tf-idf.

Present search results: Finally, we'll present the search results to the user. This typically involves displaying a list of documents that match the query, along with some summary information like the title and snippet of each document. We may also allow the user to filter or sort the search results based on various criteria.

5.2.9 9th Prompt

My Prompt: For these requitements :A complete description of the functionality, and nonfunctional requirements, of each component, and a rigorous explanation of how each component will provide the given functionality while meeting its nonfunctional requirements. You will have to draw on everything you have learned across Intro to Distributed, Advanced Distributed, Parallel Programming, and possibly other courses, to write these explanations. Give me a list of bullet points on what I should cover

response:

Sure, here's a list of bullet points to help you cover the necessary aspects of functionality and nonfunctional requirements for each component:

Functionality:

Clearly define the purpose of each component and its expected functionality. Describe the inputs and outputs of each component, including the expected data types, formats, and sizes. Explain how each component will interact with other components in the system, including communication protocols and data exchange methods. Provide a detailed description of any algorithms, models, or techniques used by each component to perform its designated task. Include any necessary error handling procedures or fallback mechanisms to ensure that the component can handle unexpected situations and provide appropriate feedback.

Nonfunctional Requirements:

Define the nonfunctional requirements that each component must meet, such as performance, scalability, reliability, and security. Specify any resource constraints, such as memory usage, processing time, or network bandwidth. Describe how each component will handle concurrency and parallelism, including any synchronization or locking mechanisms. Discuss any architectural patterns or design principles that will be used to ensure the component meets its nonfunctional requirements. Provide testing strategies and metrics for verifying that each component meets its specified requirements. Overall, your explanations should provide a clear and concise overview of each component's functionality and how it will meet its nonfunctional requirements. It's also important to consider how each component fits into the larger system architecture and how it interacts with other components to achieve the desired end result.

5.2.10 10th Prompt

My Prompt: Write me a list of bullet points as an outling to cover for these requirements:The deployment architecture of each component individually, as well as the system as a whole, should be thoroughly described. (You may find creating c4model deployment diagrams useful both for clarifying your own thoughts and for describing it in the paper) This includes: 1. the software stack on which each component is built (Language? Libraries? Server runtimes?) 2. where each component is deployed, and, if it is on multiple nodes, how the nodes coordinate: what protocols, messages, and algorithms are used 3. how the components discover each other, connect to each other, and with what protocols and messages they communicate with each other

Response:
Here's an outline to cover the requirements for describing the deployment architecture of each component individually, as well as the system as a whole:

I. Introduction

Brief overview of the system architecture Importance of describing deployment architecture II. Software Stack

Description of the software stack used for each component Programming language(s) used Libraries and frameworks used Server runtimes used III. Component Deployment

Description of where each component is deployed If deployed on multiple nodes, how the nodes coordinate Protocols, messages, and algorithms used for coordination IV. Component Communication

How components discover each other How components connect to each other Protocols and messages used for communication V. C4Model Deployment Diagrams

Explanation of C4Model deployment diagrams How C4Model deployment diagrams are useful for describing deployment architecture VI. System Deployment

Description of how the components fit together as a whole system Overall system architecture Protocols and messages used for system communication VII. Conclusion

Recap of the importance of describing deployment architecture Summary of the deployment architecture described in the paper.

5.2.11 11th prompt

ChatGPT: requirements are to thoroughly describe the deployment architecture of each component and the system as a whole. This involves:

Providing a detailed description of the software stack used for each component, including the programming language, libraries, and server runtimes used.

Identifying where each component is deployed, and if it is on multiple nodes, explaining how these nodes coordinate. This includes discussing the protocols, messages, and algorithms used for communication between nodes.

Describing how the components discover and connect to each other, as well as the protocols and messages used for communication between them.

Me: this fulfill all of those requirements:The user-facing web application has been constructed employing ReactJS as the frontend framework with a Java backend. Communication between the frontend and backend is achieved through the use of a RESTful API, which is a standard approach for web application communication. Our chosen database management system is comprised of PostgreSQL, which serves as the primary data store, with MongoDB being utilized as a supplementary backup. Careful consideration was given to the selection of these technologies to ensure optimal functionality and reliability within the application.

The web application frontend is deployed on a set of load-balanced servers to provide high availability and reduce downtime in case of hardware failure. The backend application, is also deployed on a separate set of nodes, with each service deployed in a separate container using Docker. To ensure coordination between the nodes, we use our own algorithm which is described in depth at the end of the paper.

To connect components, we use messaging protocols such as HTTP. For example, the frontend communicates with the backend using a RESTful API over HTTP, while the backend nodes communicate with each other using TCP for efficient and reliable communication. Messages between components are typically sent using standard message formats such as JSON.

*ChatGPT:*Yes, this statement fulfills the requirements by providing a clear description of the software stack for both the frontend and backend components. It also describes how each component is deployed, where they are deployed, and the protocols used for communication. Additionally, it provides information on how the components connect and communicate with each other using messaging protocols such as HTTP and TCP, along with standard message formats such as JSON.

5.2.12 12th Prompt

My prompt: Fully specify the requests and responses for each part of each API. API 1: A Login API show the requests for: When a user creates an account, they'll enter their email, phone number, and password. We'll store their email and hashed password securely in the database. We'll also generate a random code and send it to the user's email address to verify their email. We will use our email system component to send that email

ChatGPT response: API: Login API

Endpoint: /user/create

Request: When a user creates an account, they'll send the following data to this endpoint:

email: the user's email address phone: the user's phone number password: the user's password Response: After successful creation of the user account, the following response will be sent:

status: "success" message: "User created successfully. Please check your email for verification code." Endpoint: /user/verify

Request: To verify the email address of the user, the following data is required:

email: the user's email address code: the verification code sent to the user's email Response: After successful verification of the email, the following response will be sent:

status: "success" message: "Email verified successfully." Endpoint: /user/login

Request: To login, the user needs to send the following data to the endpoint:

email: the user's email address password: the user's password Response: If the login is successful, the following response will be sent:

status: "success" message: "User logged in successfully." If the email or password is incorrect, the following error response will be sent:

status: "error" message: "Invalid email or password."

5.2.13 13th Prompt

My Prompt: fantastic do exactly that again but for this: Email verification: After the user receives the email, they'll click on a verification link that contains the verification code. The link will send a request to the backend server to verify the code. If the code is correct, we'll mark the user's email as verified in the database.

ChatGPT response:

Certainly! Here's the specification in LaTeX format:

API: Email Verification

Endpoint: /verify-email

Method: GET

Request: When the user clicks on the verification link, the client must send a GET request to the /verify-email endpoint with the following query parameters:

- **email (string):** The user's email address.
- **code (string):** The verification code sent to the user's email.

Response: Upon successful verification of the email, the API will respond with a status code of 200 (OK) and the following JSON object in the response body:

```
{
"status": "success",
"message": "Email verified successfully."
}
```

(Note: I Can't believe you are actually reading through all of my logs why would anybody want to read this?)

If the verification code is incorrect or has expired, the API will respond with a status code of 401 (Unauthorized) and the following JSON object in the response body:

```
{
"status": "error",
"message": "Invalid or expired verification code."
}
```

If there is an error verifying the email for some other reason, the API will respond with a status code of 500 (Internal Server Error) and the following JSON object in the response body:

```
{
"status": "error",
"message": "An error occurred while verifying the email."
}
```

5.2.14 14th Prompt

My Prompt:

Does this fulfill all of those requirements:The user-facing web application has been constructed employing ReactJS as the frontend framework with a Java backend. Communication between the frontend and backend is achieved through the use of a RESTful API, which is a standard approach for web application communication. Our chosen database management system is comprised of PostgreSQL, which serves as the primary data store, with MongoDB being utilized as a supplementary backup. Careful consideration was given to the selection of these technologies to ensure optimal functionality and reliability within the application.

The web application frontend is deployed on a set of load-balanced servers to provide high availability and reduce downtime in case of hardware failure. The backend application, is also deployed on a separate set of nodes, with each service deployed in a separate container using Docker. To ensure coordination between the nodes, we use our own algorithm which is described in depth at the end of the paper.

To connect components, we use messaging protocols such as HTTP. For example, the frontend communicates with the backend using a RESTful API

over HTTP, while the backend nodes communicate with each other using TCP for efficient and reliable communication. Messages between components are typically sent using standard message formats such as JSON.

ChatGPT response:

Yes, this statement fulfills the requirements by providing a clear description of the software stack for both the frontend and backend components. It also describes how each component is deployed, where they are deployed, and the protocols used for communication. Additionally, it provides information on how the components connect and communicate with each other using messaging protocols such as HTTP and TCP, along with standard message formats such as JSON.