OpinionMarket

Josh Van de Walle

CST-451 Capstone Project Final Architecture & Design

Grand Canyon University

Instructor: Professor Mark Reha

Revision December 4, 2020

Date December 4, 2020

**ABSTRACT**

The moniker OpinionMarket captures the concept of a marketplace of ideas making it the perfect name for an application all about sharing information and thoughts. The application is a web platform for social news and community-driven discussion that will facilitate discourse between individuals based on common interests. OpinionMarket will allow users to join communities of their choice, post content in those communities, comment on content posted by others, upvote or downvote any post or comment, and send direct messages. These features will support the creation of a vast ecosystem of meaningful discussions.

The platform is designed to be easy for anyone to use and helpful for everyone. Whatever a user is interested in, be it a hobby, professional skill, or theoretical physics, they can find or start an online community dedicated to it. Community rules establish user-enforced behavioral regulations that keep communities focused on their topic. Customization features make communities unique, give them character, and allow them to stand out. The wealth of features they provide make OpinionMarket communities an excellent place to look for help, show off accomplishments, or find discussion of just about anything. The application presents an intuitive and friendly user interface that makes it easy to dive-in and start browsing content-rich communities.

|  |
| --- |
| History and Signoff Sheet |

**Change Record**

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Revision Notes** |
| 11/29/2020 | Josh Van de Walle | Initial draft for review/discussion |
| 12/02/2020 | Josh Van de Walle | Updated physical and logical architecture to show ports between React and Spring. Updated risk log. |
| 12/04/2020 | Josh Van de Walle | Updated Security and Operational Support Designs. Added details on Responsive UI design |

|  |
| --- |
| **Overall Instructor Feedback/Comments** |

|  |
| --- |
| **Overall Instructor Feedback/Comments** |

**Integrated Instructor Feedback into Project Documentation**

Yes  No

**TABLE OF CONTENTS**

Design Introduction 4

Detailed High-Level Solution Design 5

Detailed Technical Design 11

Appendix A – Technical Issue and Risk Log 41

Appendix B – References 42

Appendix C – External Resources 43

Design Introduction

The table below lists deliverables that support this document but are not included in it.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Deliverable Acceptance Log | | | | | |
| ID | Deliverable Description | Comments | Evaluator (internal or external as applicable) | Status | Date of Decision |
| 1 | API Design | Design available on Swagger | Josh Van de Walle | Done | 11/27/2020 |
| 2 | Data dictionary | Attached in data\_dictionary.xslx file | Josh Van de Walle | Done | 11/27/20 |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |

Detailed High-Level Solution Design

The table below includes all proof of concept projects undertaken in association with the OpinionMarket project. The table include a description of the project, the rationale for the project, and the results of the project.

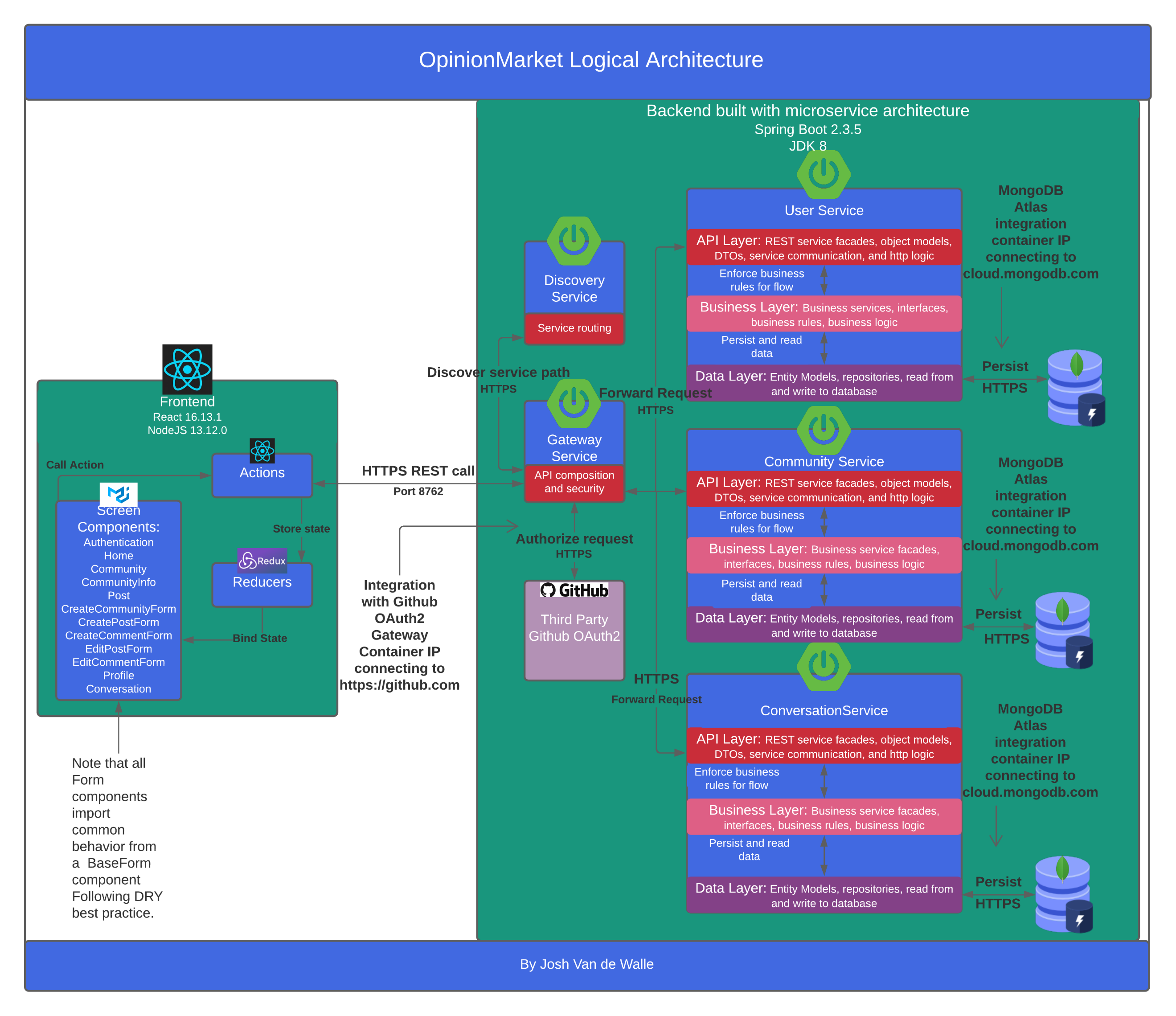
|  |  |  |
| --- | --- | --- |
| Proof of Concepts | |  |
| **Description** | **Rationale** | **Results** |
| 1. Build POC application with React, .NET Core and MongoDB | To determine if the React + .NET Core + MongoDB stack was the right choice for the project | Although this stack is suitable, Spring Boot was chosen over .NET Core because of available resources related to microservice architecture |
| 2 – Build POC application with Express and Mongo | To determine if the MERN stack was the right choice for the project | This stack was not selected for use because Spring Boot was more suitable for the project than Express |
| 3 – Build microservice architecture POC application with Spring Boot and MongoDB | To determine if Spring Boot and MongoDB were suitable backend stack components | Spring Boot and MongoDB were selected as technologies for the backend |
| 4 – Deploy a Spring Boot application that connects to MongoDB Atlas in a Docker Container on AWS | To validate that a Spring Boot application in a Docker container on AWS can connect to MongoDB | The task was successful, and the architecture was validated |
| 5 – Build an OAuth proof of concept application | To validate that OAuth can be used to secure the project’s API | The task was successful, and the architecture was validated |
| 6 – Deploy React application in the cloud | To validate that a React application could be deployed in the cloud | The task was successful, and the architecture was validated |
| 7 – Connect React and Spring Boot applications in the cloud | To validate that REST API communication is possible between React and Spring Boot applications in the cloud | The task was successful, and the architecture was validated. |

The table below includes a list of all the technologies OpinionMarket leverages.

|  |
| --- |
| Hardware and Software Technologies |
| 1 – Spring Boot 2.3.5 |
| 2 – Java 1.8.0 |
| 3 – Maven 3.6.3 |
| 4 – React 16.13.1 |
| 5 – Redux 4.0.5 |
| 6 – Material-UI 4.11.0 |
| 7 – Axios 0.20.0 |
| 8 – React-Router 5.20.0 |
| 9- React-Router-DOM 5.20.0 |
| 10 – React-Images-Upload 1.2.8 |
| 11 – JavaScript ECMAScript 2018 |
| 12 – MongoDB 4.2.10 |
| 13 – MongoDB Atlas |
| 14 – Docker Engine 19.03.13 |
| 15 – Spring Tool Suite 4.7.2.RELEASE |
| 16- Visual Studio Code 1.15.1 |
| 17 – AWS EC2 |
| 18 – AWS Elastic Beanstalk |
| 19 – Amazon CloudWatch |
| 20 - Loggly |
| 21 – Swagger OpenAPI Specification 3.0.3 |
| 22 – Postman 7.34.0 |
| 23 – MongoDB Compass Community 1.21.2 |

**Logical Solution Design:**

OpinionMarket is designed using microservice architecture and the database-per-service pattern. A Zuul API gateway application and a Eureka discovery service discovery application are used to handle API routing and security. Because each service has its own database, the gateway is responsible for API composition in cases where a transaction must be made across multiple databases. On the frontend a React client application communicates with the API gateway via a REST API and displays the user interface. The details of the system’s APIs are documented on Swagger and can be accessed from the link in the Service API Design section of this document. The diagram below illustrates OpinionMarket’s logical architecture.



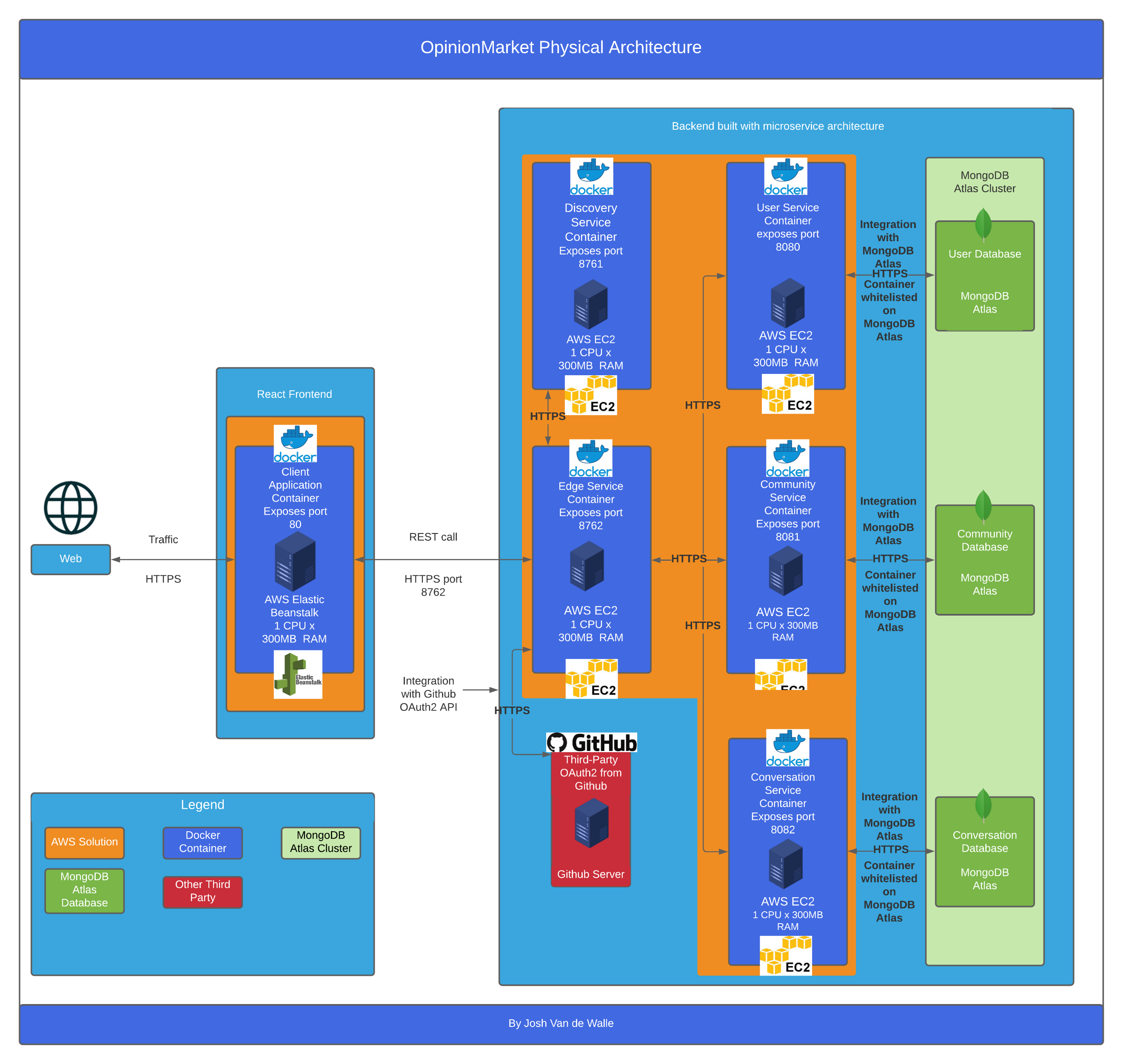
Microservice architecture divides an application into smaller applications to increase the maintainability, technical flexibility, scalability, and fault-tolerance of the overall system (Nemer, 2019). The architecture has several advantages. Changes made to one microservice do not force the entire application to be re-compiled, meaning the code is easier to maintain. Maintainability also benefits from microservices being more simple, from a code standpoint, than monolith applications. This makes a microservice easier for a developer to understand when designing and building an update. Microservices can use different frameworks and databases based on what technology provides the best solution for the problem they are solving. This technical flexibilty means development teams are no longer forced to use technologies unsuitable to the task they are working on. Microservices make scaling up an application easier. Since services are separate it’s easier to scale only the services that must be scaled, improving efficiency. If there is a problem in one microservice the rest of the system is far less likely to be affected than with a monolith architecture because of the code boundaries between microservices.

The data management microservices leverage a variety of design patterns including the layered architecture, Data Transfer Object (DTO), façade, and interceptor patterns. Layered architecture separates concerns within the microservices improving code maintainability. The DTO pattern is the vehicle used for interservice communication and for communicating with the React frontend. The pattern bundles together all relevant information about an API request or response, such as status codes, messages, authorization tokens, and data, in a single object that can be easily exchanged over HTTPS. The Façade pattern is implemented as REST service classes in each service’s API layer and as business service classes in each service’s business layer. These Facades hide the complexity of the business and data access logic beneath them and provide a simple interface to handle incoming requests and database interactions. The Interceptor pattern is used to implement logging in such a way that logging statements need not be repeated allowing the system to adhere to Don’t Repeat Yourself (DRY) best practices.

The React frontend application leverages the Model-View-Update architecture pattern. Model here, refers to the application’s state, stored in the Redux state container. The view architecture component refers to the React components that have state bound to them and are displayed to the user. The Update functions are the API actions that obtain updated state from the backend API. As the logical architecture diagram above illustrates, The MVU pattern creates a unidirectional data flow. The actions update the state, the reducers bind the state to components, and the components call on actions to update the state again in response to user activity.

**Physical Solution Design:**

OpinionMarket’s microservice architecture creates a natural mapping of its logical components onto its physical components. Each data management microservice, the API gateway, and the discovery service will be made into Docker images and deployed in AWS EC2 containers. The React frontend application will similarly be made into a docker image and containerized in AWS elastic beanstalk. A MongoDB Atlas cluster will be used to host the system’s databases in the cloud. The microservices’ domains will be added to Atlas’ access whitelist securing the databases. The diagram below illustrates OpinionMarket’s physical architecture.



Each container will require 1 Central Processing Unit (CPU) and 300 megabytes of Random-Access Memory (RAM). The client application container will expose port 80, the gateway service container will expose port 8762, the discovery service container will expose port 8761, the user service container will expose port 8080, the community service will expose port 8081, and the conversation service will expose port 8082. By having each container expose different ports, the architecture ensures that the OpinionMarket system can be easily run locally on developer’s machines.

Detailed Technical Design

**General Technical Approach:**

The application will be built using a microservice architecture on the backend and a client application on the frontend. To support the microservice architecture, an API gateway service and a discovery service will act as utilities, handling tasks such as security, service discovery, and API composition. Each microservice, and the client application will be containerized and deployed in the cloud. Cloud hosted databases will be leveraged, and the database-per-service architecture pattern will be followed. There will be a total of three data management microservices and three databases. Because a microservice architecture is being used with the database-per-service pattern, transactions are not possible. This problem will be solved by using API composition performed by the gateway service.

The decision to use this architecture was made so that the project could enjoy the benefits of microservices. Microservices improve the scalability, flexibility, maintainability, and fault-tolerance of an application. Since microservices separate the components of an application, the architecture makes it easy to scale only the parts of a system that need to be scaled. Unlike monolith architecture, Microservices allow developers to use different frameworks and technologies in different parts of the same system based on the technology’s suitability for the task at hand. Microservice architecture improves the maintainability of an application because changes to one microservice do not force the entire application to be re-compiled. Maintainability also improves because microservices reduce the complexity of an application’s code, making building and deploying updates easier. If there is a problem in one microservice, the rest of the system is less likely to be affected then with monolith architecture because of the code boundaries between microservices.

**Key Technical Design Decisions:**

The project’s technologies were chosen based on being both industry-relevant and complementary, both to the overall architecture, and each other. The microservices will be written in Java and use the Spring Boot framework. MongoDB has been chosen as the database solution for the application. The client application will be built using React, Redux, and Material-UI. Docker will be used to containerize application components. Amazon Web Services (AWS) will be used to host the application. Github OAuth2 will be leveraged for authorization.

The decision to use Spring Boot was made after evaluating alternatives including .NET Core and Express. Proof of concept applications were built using each framework under consideration. Based on these proofs of concept, Spring Boot was determined to be the most suitable framework for the project due to the ease with which it can be containerized, deployed to clouds, and integrated with MongoDB.

MongoDB was chosen because of its increasing popularity and innovative approach to data persistence. The JSON-like format in which Mongo documents are stored clarifies the structure of an object and eliminates the need for the complex joins often used with relational databases (tutorialspoint, 2020). Mongo can store objects within other objects using embedding, which represents a code object more accurately than relational databases can. In keeping with the theme of high scalability, MongoDB is designed to be easy to scale-out (mongodb, 2020). Mongo does not use a schema. As a result, documents can take whatever form most precisely represents the relevant data, making the application more flexible (mongodb, 2020). Because both microservice architecture and MongoDB increase an application’s scalability and flexibility, it makes sense to use them together.

React was selected for use in the project because of its status as the most popular frontend framework (Simform, 2020). React provides several advantages including the ability to reuse HTML code across the application, a virtual document object model (DOM) that eliminates inefficiencies associated with the actual DOM, and the ability to use Redux for storing data that will be needed across the application (Otuagomah, 2020). Material-UI was chosen as the application’s frontend CSS library because of the responsive components it provides, its extensive collection of beautiful UI elements, and its comprehensive official documentation.

Docker containers provide a consistent and isolated environment for applications to run in (Atkins & Pavlovic, 2020). As a result, productivity increases because developer time can be directed towards development instead of debugging complex environment issues (Atkins & Pavlovic, 2020). Using Docker with microservice architecture is a natural decision for several reasons, including increased application scalability and flexibility. Docker containers clearly define the code boundaries between services, containers can easily be replaced when an update is needed without affecting other containers, and containers can be easily scaled in the cloud (Atkins & Pavlovic, 2020).

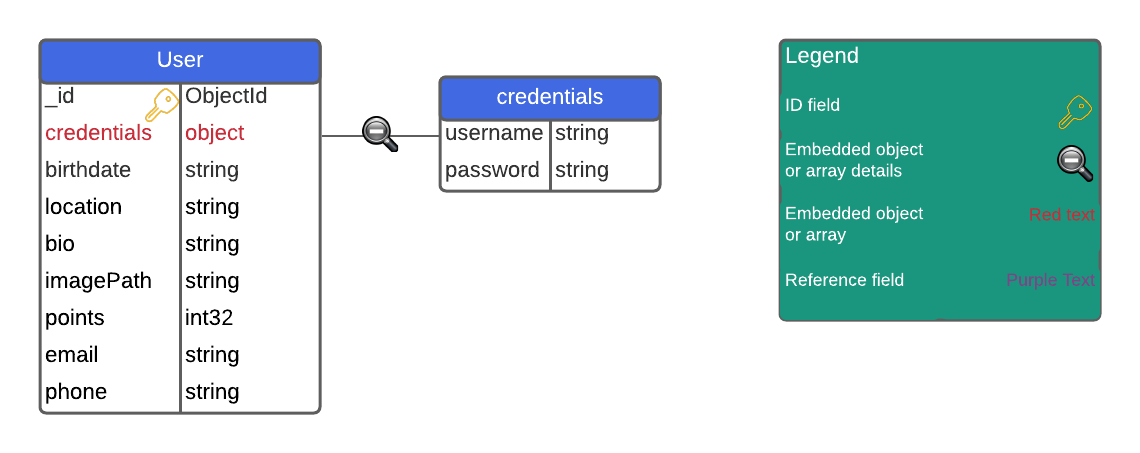
AWS is the cloud solution of choice for microservice architecture according to the *State of Microservices* *2020* report (The Software House, 2020). AWS’s Elastic Cloud Compute (EC2) and Elastic Beanstalk services make deploying Docker containers as painless as possible. OpinionMarket’s backend microservices will be deployed in EC2 containers, while the client application will be deployed in Elastic Beanstalk. Because of its compatibility with Docker and with microservice architecture, AWS was a natural choice for the project’s cloud solution.

The overall stack consists of a collection of technologies, frameworks, and solutions that not only work well together but are relevant in the software industry. The technologies chosen also work well with the overall system’s logical and physical architecture. Because of the time and effort put into technology selection and proof of concepting during the early phases of the Systems Development Lifecycle (SDLC), the risks associated with the technologies were mitigated by the start of the design phase and well before development phase. Because these risks have been addressed in earlier phases of the project, the development phase will not be an exercise in on-the-spot architecture. Instead, development will be guided by a design grounded in firsthand experience with the project technologies.

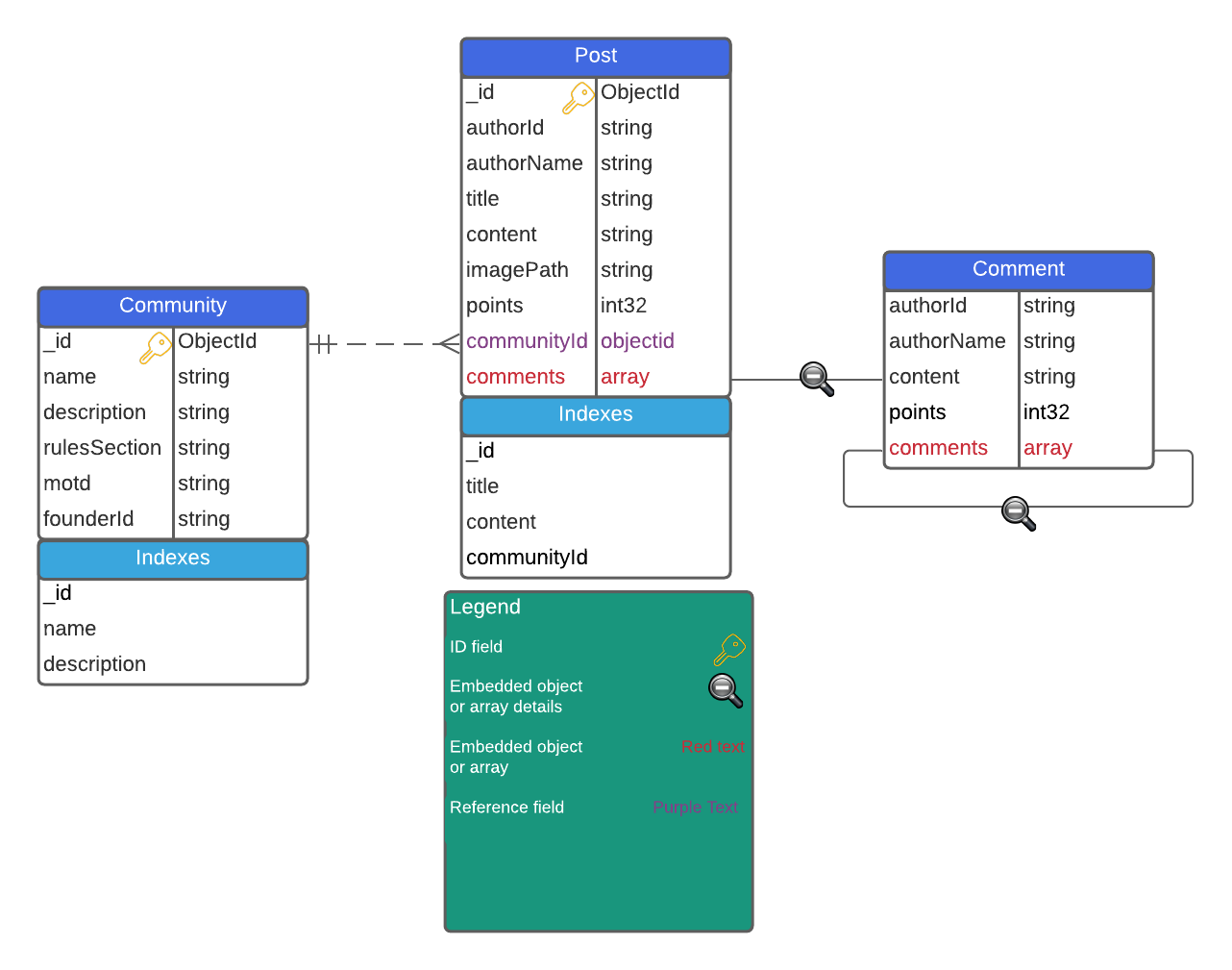
**Database ER Diagram:**

The Entity-Relationship Diagrams in this section depict the design of each database used by the OpinionMarket system. MongoDB is the technology used for all databases. Both normalization (referencing) and denormalization (embedding) have been leveraged in the designs.

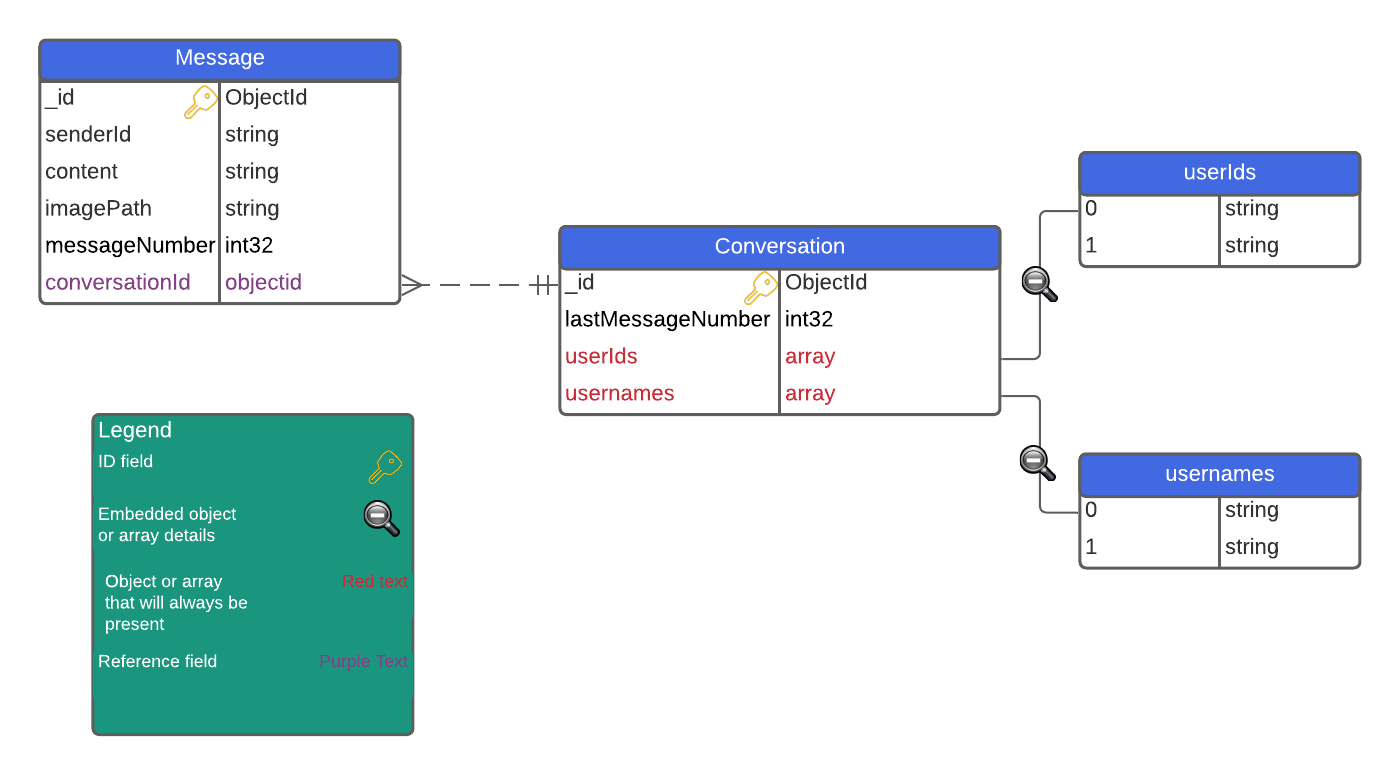
The ER diagram below illustrates the database design of the user service’s database. The credentials field of the User collection contains an embedded object storing the user’s authentication credentials. The decision to use denormalization to depict this relationship was made because there is a one-to-one relationship between users and their credentials.



The following ER diagram depicts the design of the community service’s database. The communityId field of the Post collection is a reference to a community collection document. The decision to implement this relationship using normalization was made because of the potential for the number of posts in a community to grow very large and the fact that not all posts in a community must be displayed when the community is displayed, meaning there will be more write operations than read operations for this field. The comments field of the Post collection contains an embedded array of comment objects. The decsion to use denormalization to represent this relationship was made because, even though the number of comments on a post might grow large, all comments must be displayed whenever the post is displayed, meaning there will be more read operations than write operations for this field. In additon to ID fields and references, the name and description fields of the Community collection togther with the title and content fields of the Post collection are indexed. This is done to support users searchimg communities by name or description and searching posts by title or content.



The following ER diagram depicts the design of the conversation service’s database. The conversationId field of the Message collection is a reference to a conversation collection document. The decision to use normalization to represent this relationship was made because of the potential for the number of message in a conversation to grow very large and the fact that not all messages in a conversation must be displayed at once. The userIds and usernames fields of the Conversation collection contain embedded arrays. The decision to use denormalization to represent this relationship was made because the arrays contain primitive data and are limited in size.



**Database Data Dictionary:**

The OpinionMarket data dictionary describes the contents, format, and structure of the databases. The data dictionary is attached in the data\_dictionary.xslx file.

**Database DDL Scripts:**

MongoDB Compass allows developers to export Mongo Collections as JSON files. Compass also connects to MongoDB Atlas and can import data to a cloud database from a local JSON file. The following JSON files contain example data for each OpinionMarket collection.

The following is the importable sample JSON for the User collection of the User database.

[{

"\_id": {

"$oid": "5fc3cec980b2caa55332d64f"

},

"credentials": {

"username": "JoshV",

"password": "a3gsnn4lv6d2"

},

"birthdate": "1998-06-14",

"location": "Phoenix, AZ",

"bio": "I enjoy programming, software architecture, and sports.",

"imagePath": "",

"points": 9999,

"email": "josh@opinionmarket.com",

"phone": "0000000000"

},{

"\_id": {

"$oid": "5fc3e62237706b4b746f1a8c"

},

"credentials": {

"username": "JoshV2",

"password": "a3gsnn4lv6d2"

},

"birthdate": "1998-07-14",

"location": "San Jose, CA",

"bio": "I enjoy reading",

"imagePath": "public/img/profile/i341567.png",

"points": 99,

"email": "josh2@opinionmarket.com",

"phone": "0000000000"

}]

The following is the importable example JSON for the Community collection of the Community database.

[{

"\_id": {

"$oid": "5fc3d47a8bae84798a036cde"

},

"name": "NFL Fans",

"description": "Sunday, Monday, Thursday, everyday.",

"rulesSection": "1. No abuse to others",

"motd": "Happy Sunday!",

"founderId": "1n4dcj3jckzeiwz"

},{

"\_id": {

"$oid": "5fc3e9b837706b4b746f1a8d"

},

"name": "Programmers",

"description": "StackOverflow has nothing on us!",

"rulesSection": "1. Don't duplicate questions",

"motd": "Roses are red violets are blue, unexpected '}' on line 32",

"founderId": "dc23xcf3dcgcswbju"

}]

The following is the importable example JSON for the post collection of the Community database.

[{

"\_id": {

"$oid": "5fc3eb23d1b4997c89fcc097"

},

"authorId": "5fc3cec980b2caa55332d64f",

"authorName": "JoshV",

"title": "An interesting title",

"content": "Interesting Content",

"points": 34,

"communityId": {

"\"$ref\"": "Community",

"\"$id\"": {

"$oid": "5fc3ed95d1b4997c89fcc099"

}

},

"comments": [

{

"authorID": "qwertyu",

"authorName": "JoshV",

"content": "Interesting content",

"points": 3,

"comments": [

null

]

},

{

"authorId": "asdfgh",

"authorName": "JoshV2",

"content": "less interesting content",

"points": -15,

"comments": [

null

]

}

]

}]

The following is the importable example JSON for the Conversation collection of the Conversation database.

[{

"\_id": {

"$oid": "5fc3f213d1b4997c89fcc09a"

},

"lastMessageNumber": 22,

"userIds": [

"5fc3cec980b2caa55332d64f",

"5fc3e62237706b4b746f1a8c"

],

"usernames": [

"JoshV",

"JoshV2"

]

}]

The following is the importable example JSON for the Message collection of the Conversation database.

[{

"\_id": {

"$oid": "5fc3f498d1b4997c89fcc09b"

},

"senderId": "5fc3e62237706b4b746f1a8c",

"content": "Hello!",

"messageNumber": 21,

"conversationId": {

"\"$ref\"": "Conversation",

"\"$id\"": "5fc3f213d1b4997c89fcc09a"

}

},{

"\_id": {

"$oid": "5fc3f5d437706b4b746f1a8e"

},

"senderId": "5fc3cec980b2caa55332d64f",

"content": "How is it going?",

"messageNumber": 22,

"conversationId": {

"\"$ref\"": "Conversation",

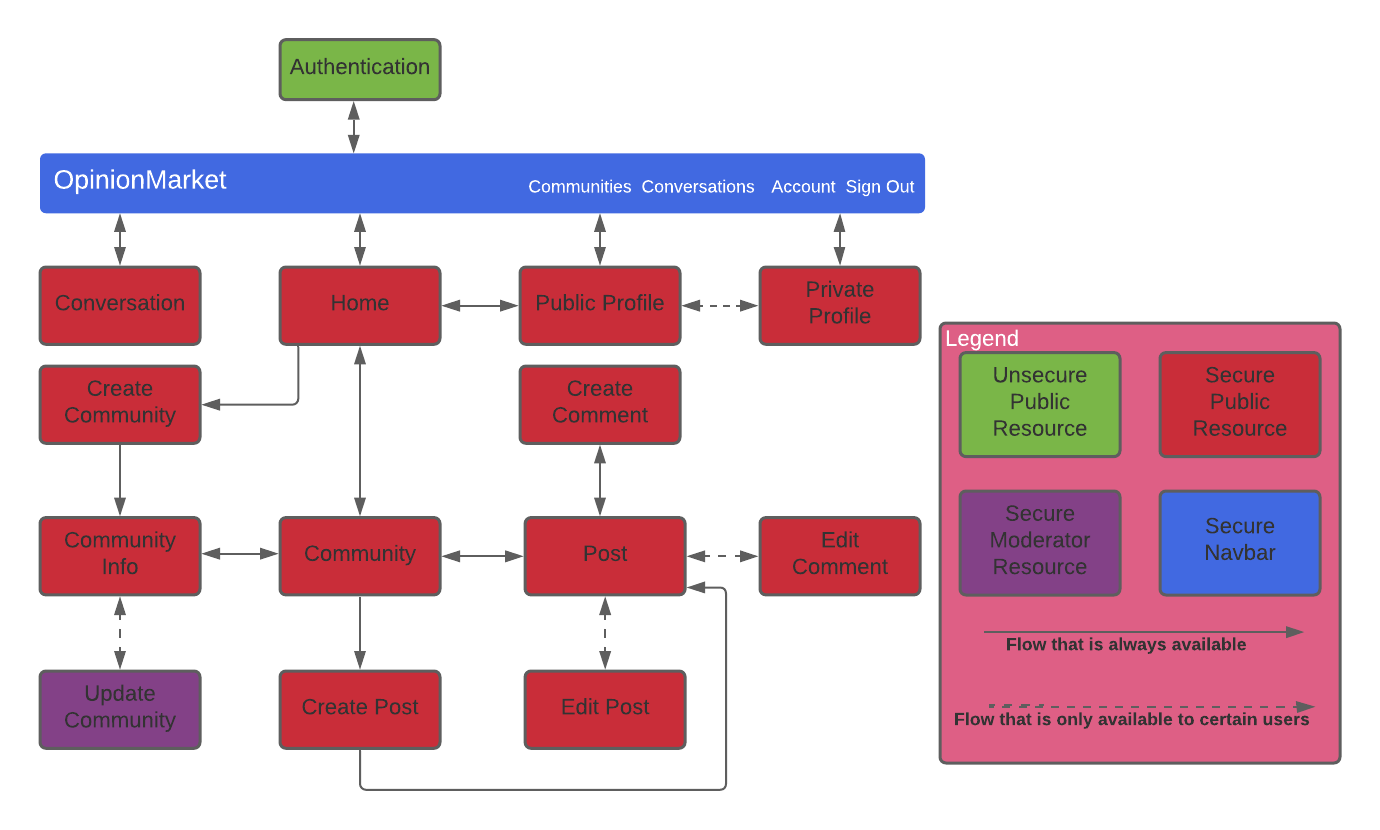
"\"$id\"": "5fc3f213d1b4997c89fcc09a"

}

}]

**Sitemap Diagram:**

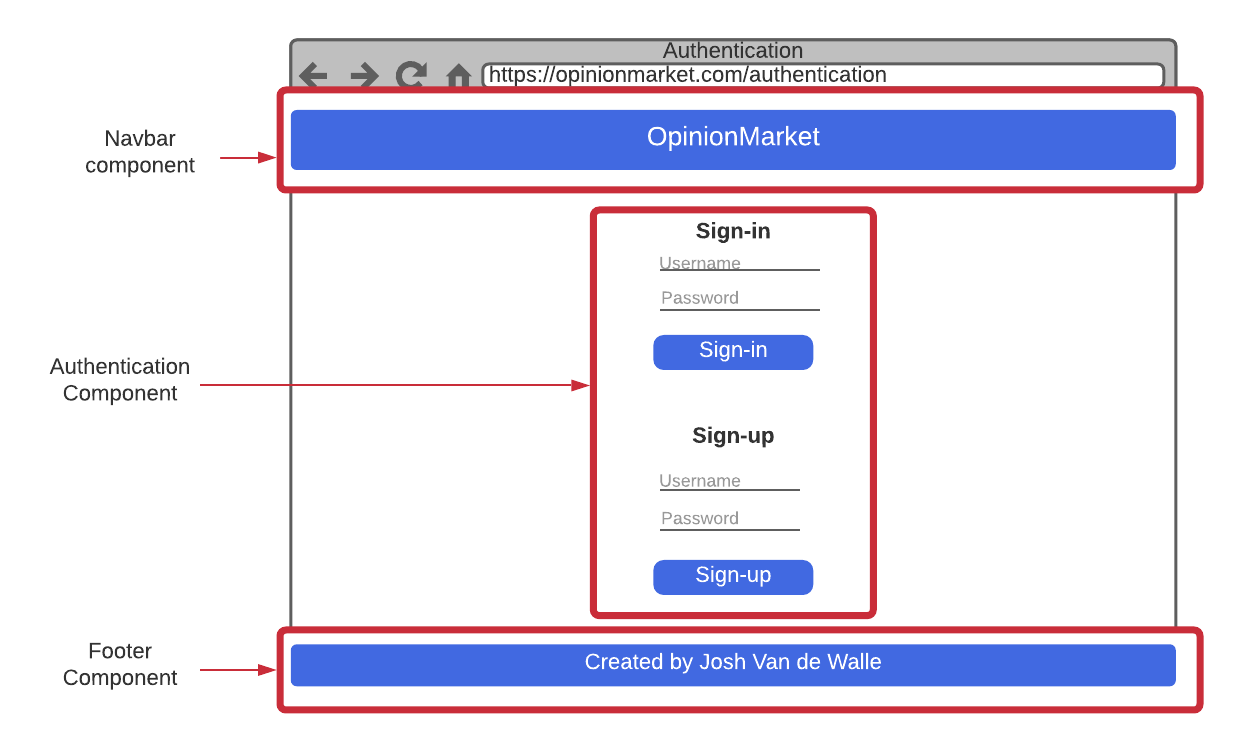
The sitemap below illustrates the screens and flows provided by the client application, as well as the supported navigation between them. The navbar is present on all screens except the Authentication screen.



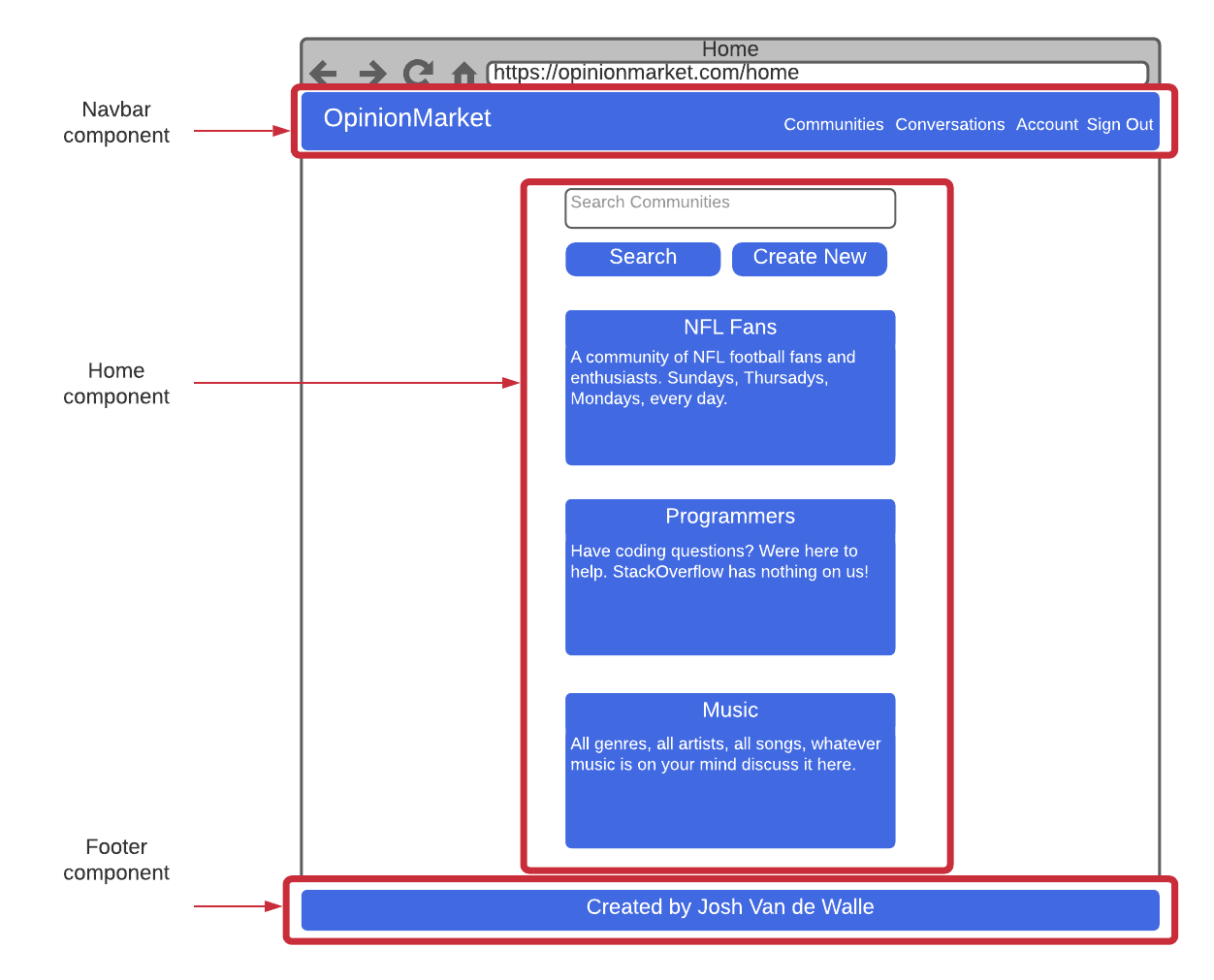
**User Interface Diagrams:**

OpinionMarket is committed to providing an intuitive and friendly user interface across the client application. The user interface will leverage the Material-UI library to support a responsive design. The library provides components that can be used to adapt their contents to various screen sizes. Material-UI components that will be leveraged include the Grid and Container components. The following user interface design diagrams depict the screens provided by the client application along with their underlying React components.

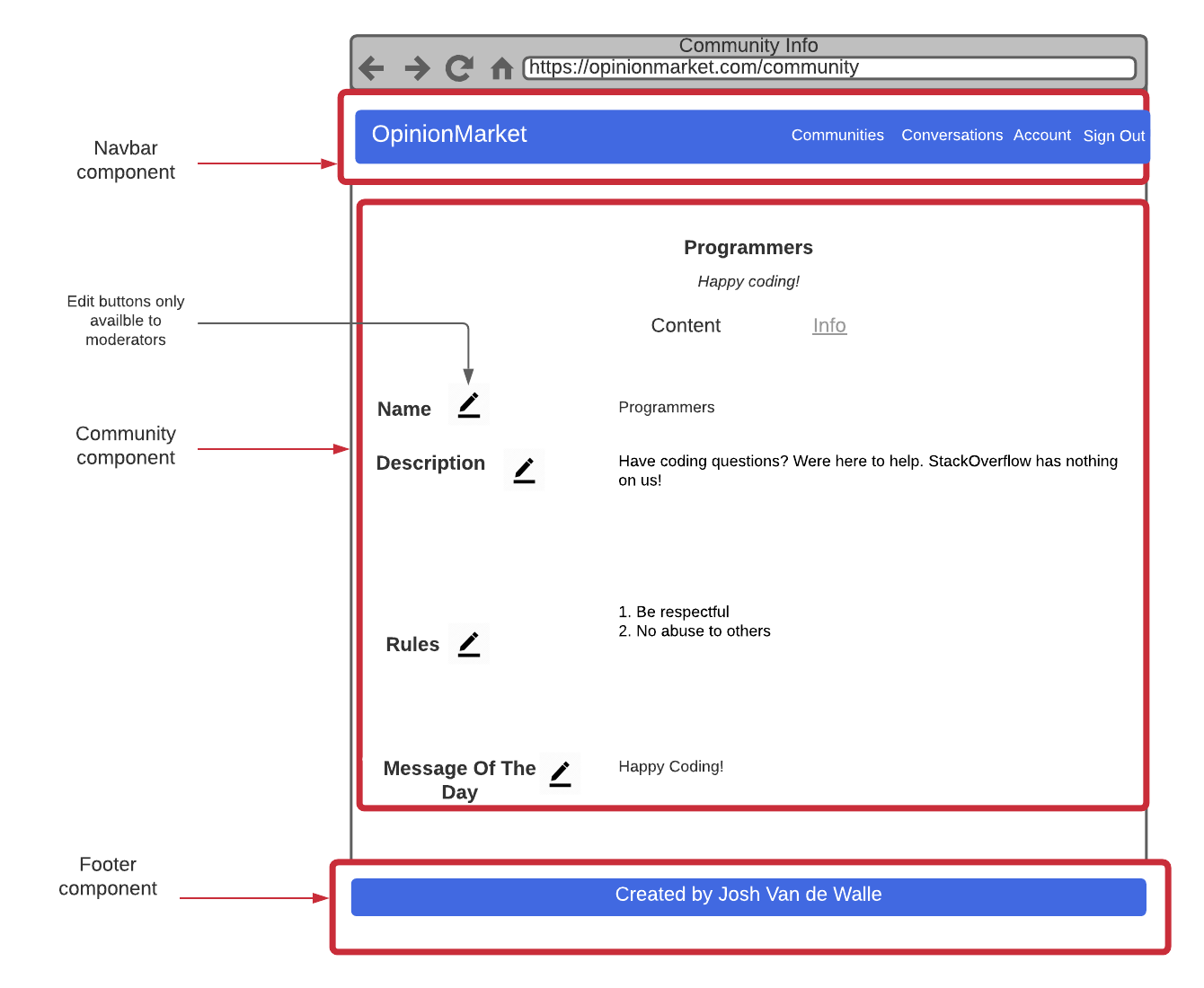
The diagram below shows the planned authentication screen.



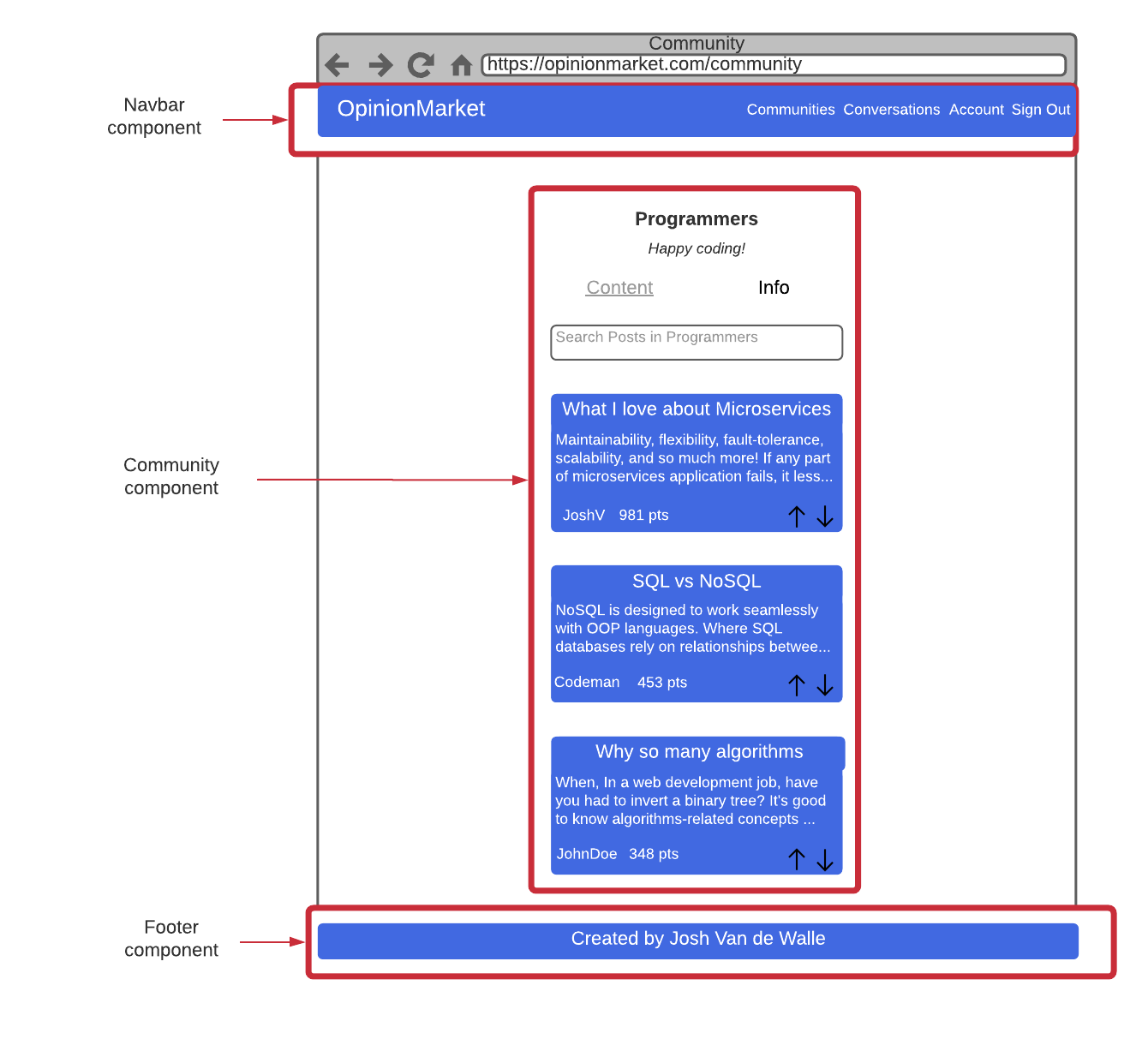
The diagram below shows the planned home screen.



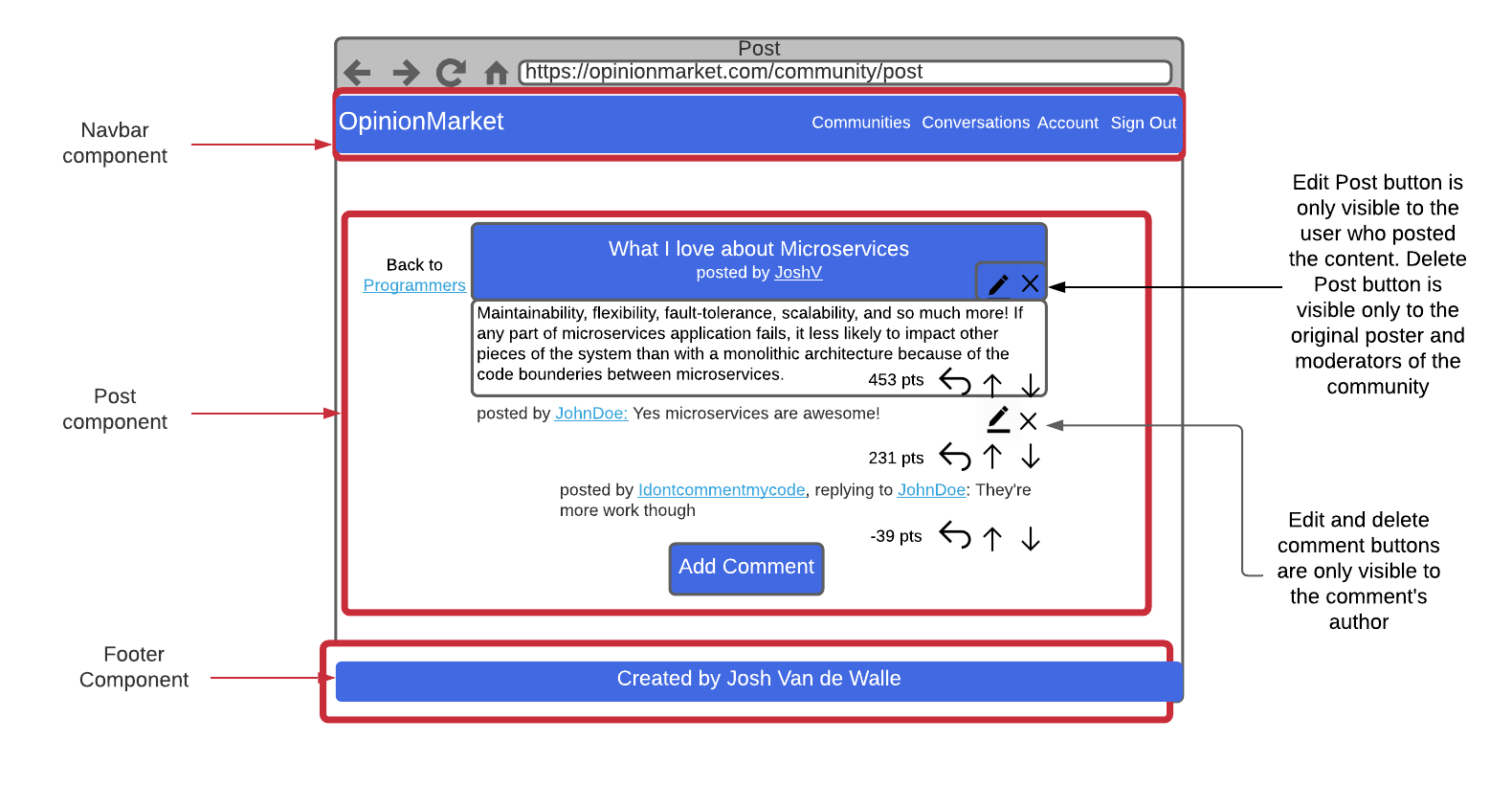
The following diagram shows the planned community info screen.

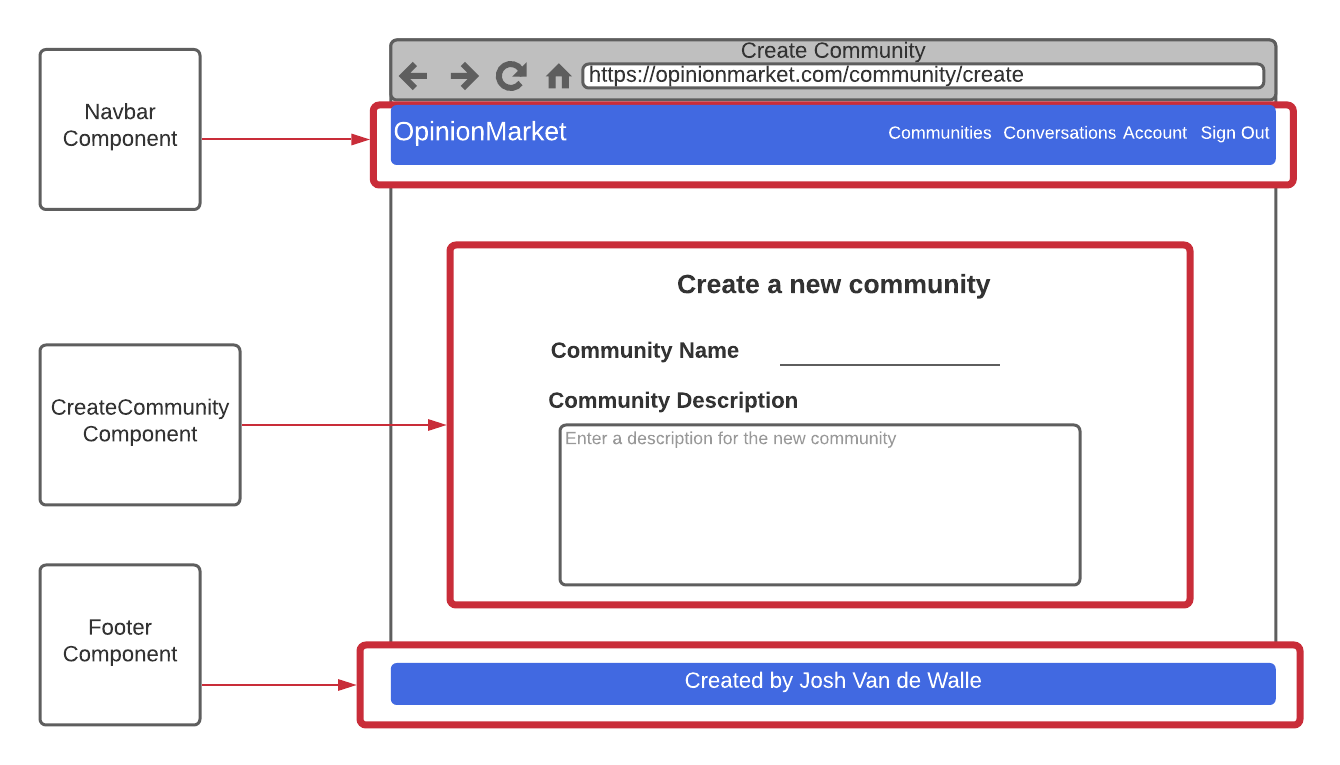


The following diagram shows the planned community screen.



The diagram below shows the planned post screen. Note the edit post button only appears to the user who posted the content. The delete post button only appears to the user who posted the content and the moderators of the community.

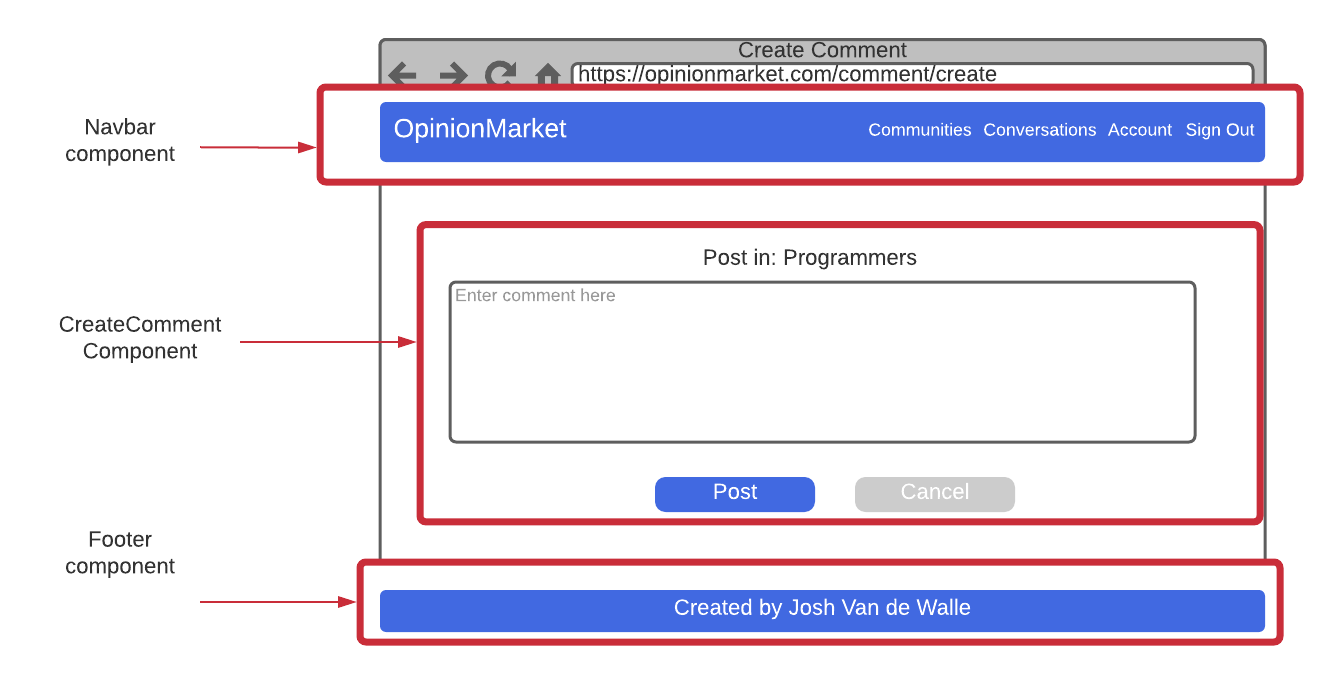
The diagram below shows the planned community creation screen.



The diagram below shows the planned create post screen.



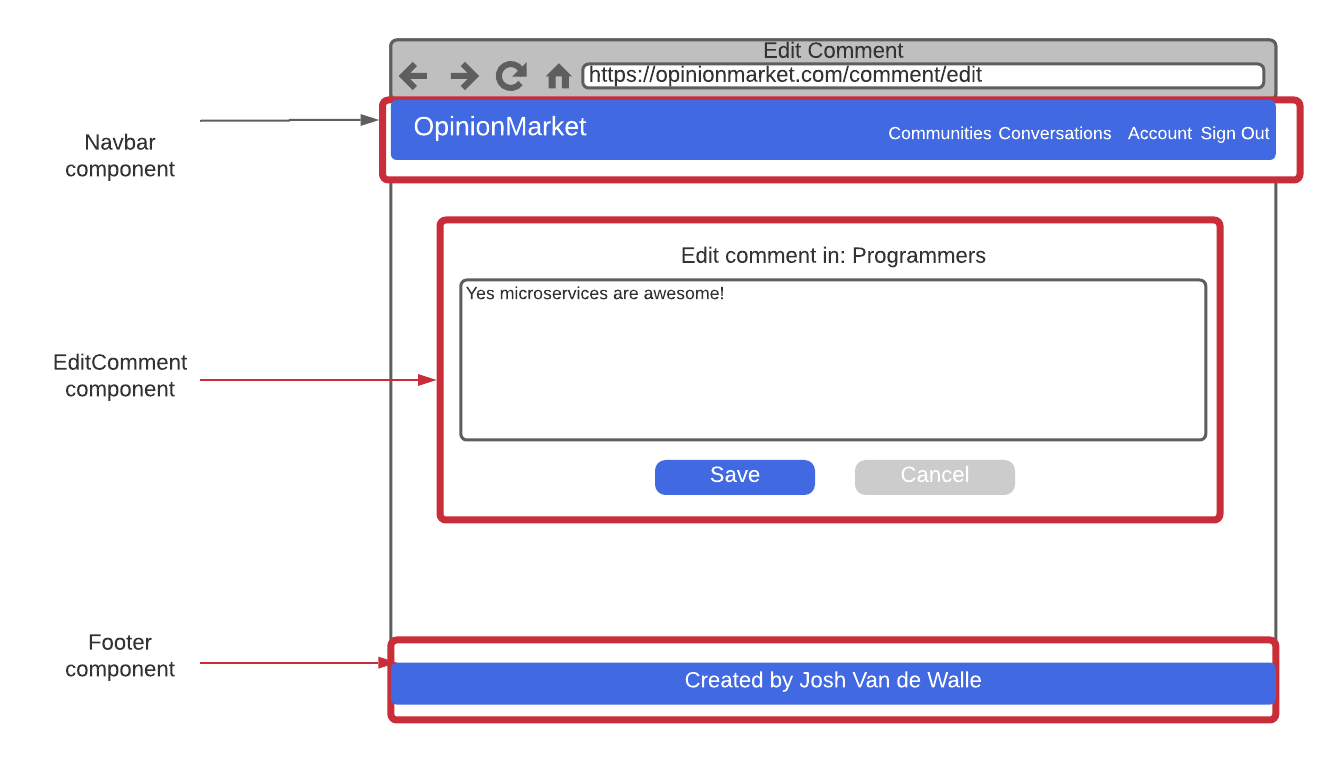
The following diagram shows the planned create comment screen.



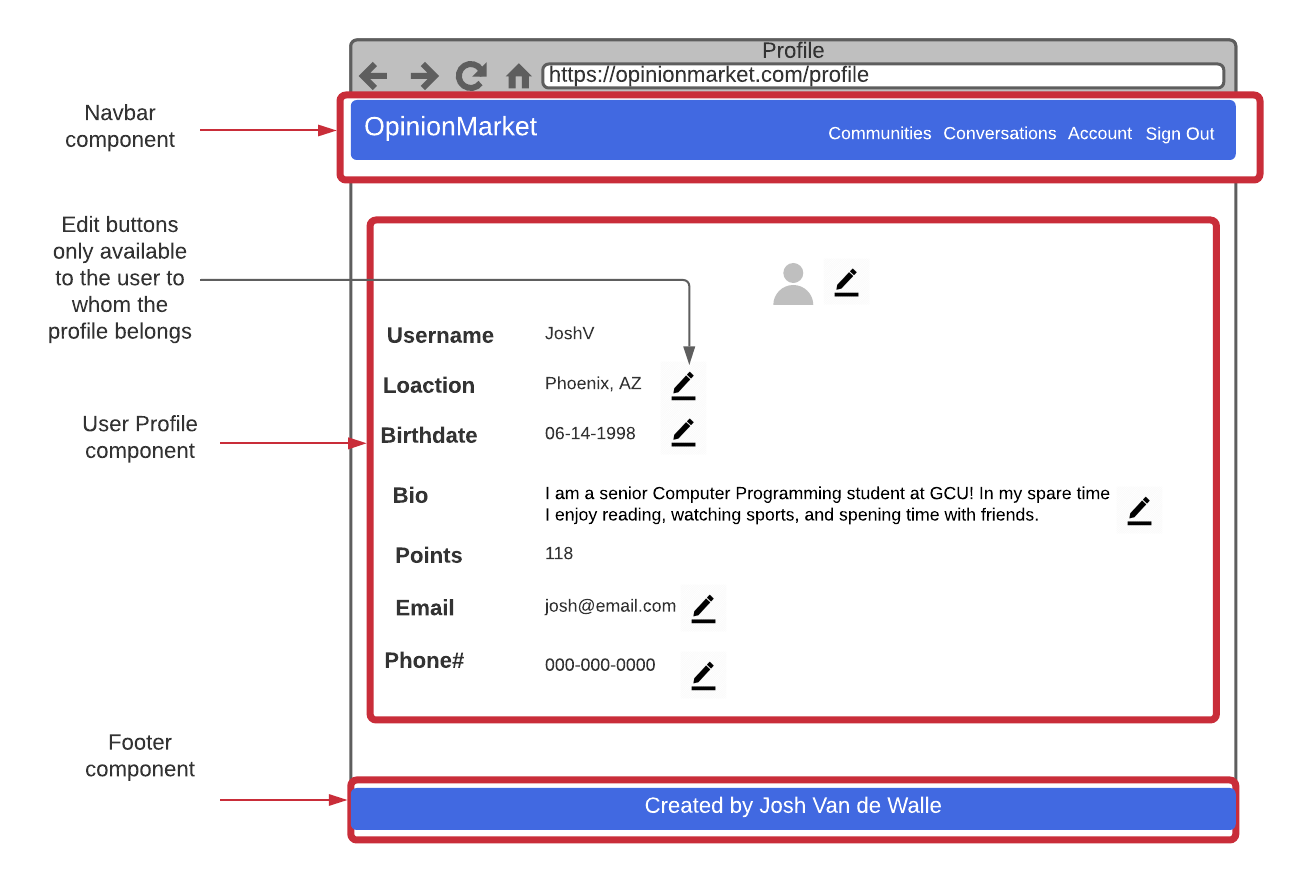
The following diagram shows the planned edit post screen.



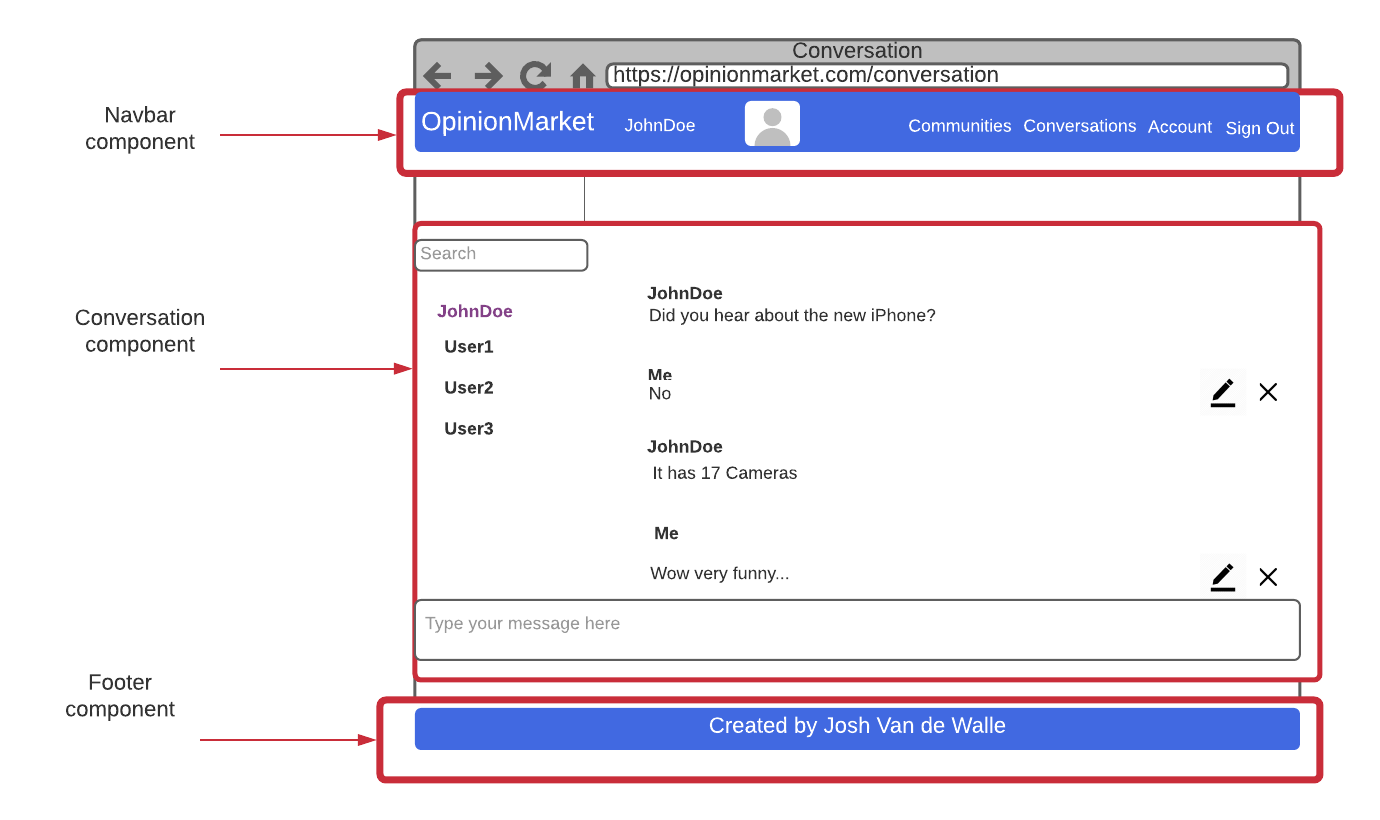
The following diagram shows the planned edit comment screen.



The following diagram shows the planned user profile screen. Note that the edit buttons are only available to the user who owns the profile.

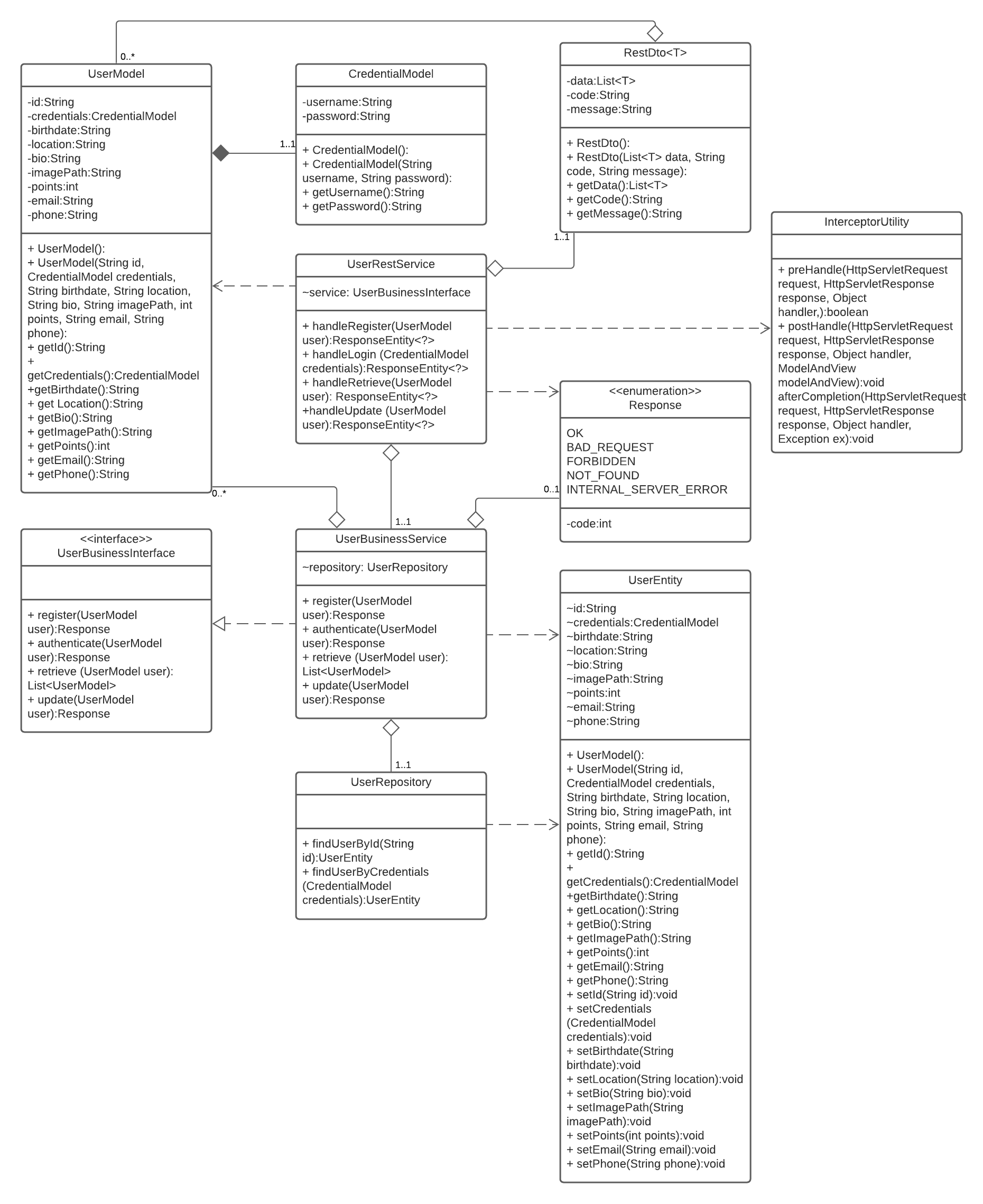


The following diagram shows the planned conversation screen.



**UML Diagrams:**

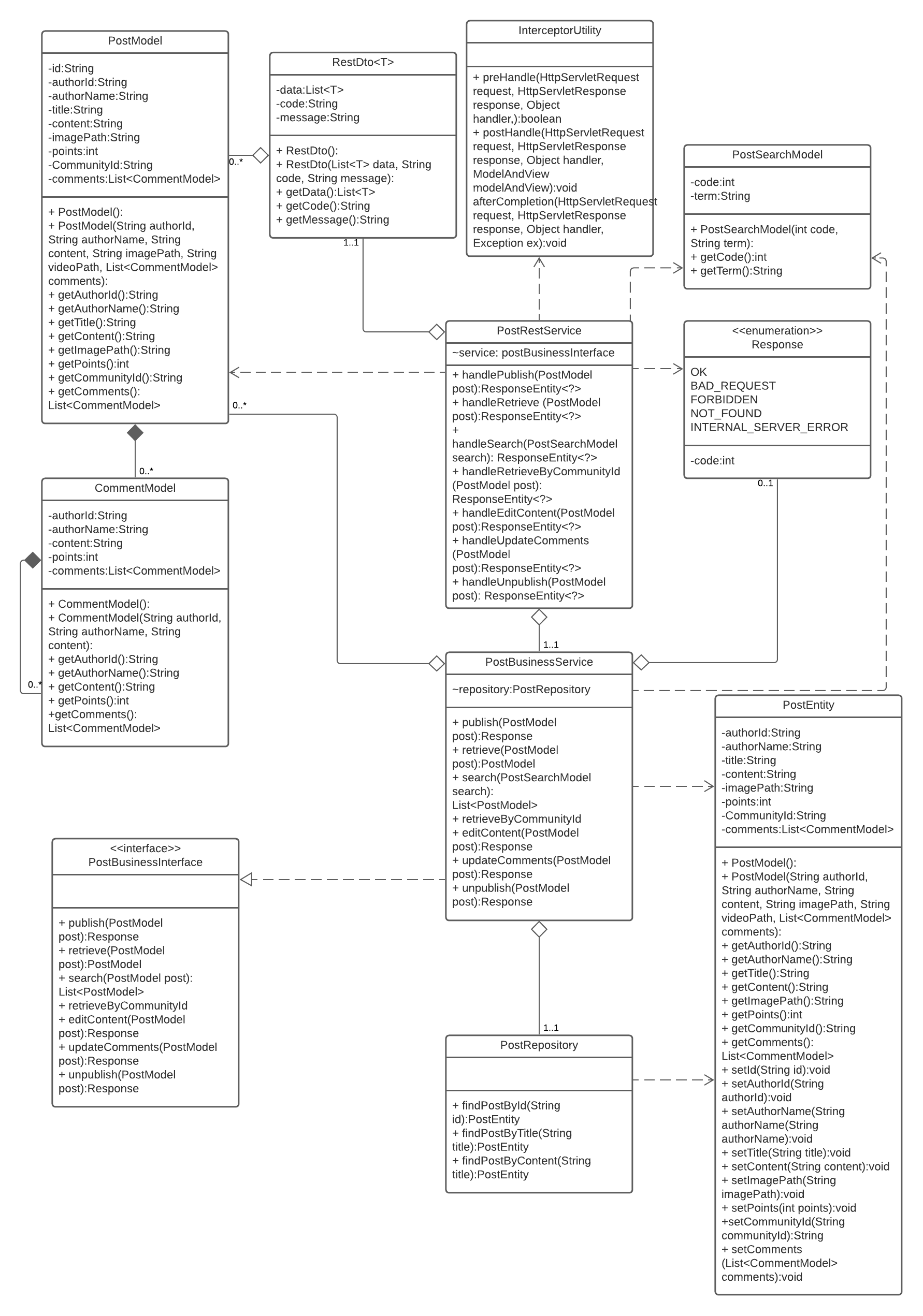
The UML class diagram below depicts the user module of the user service.



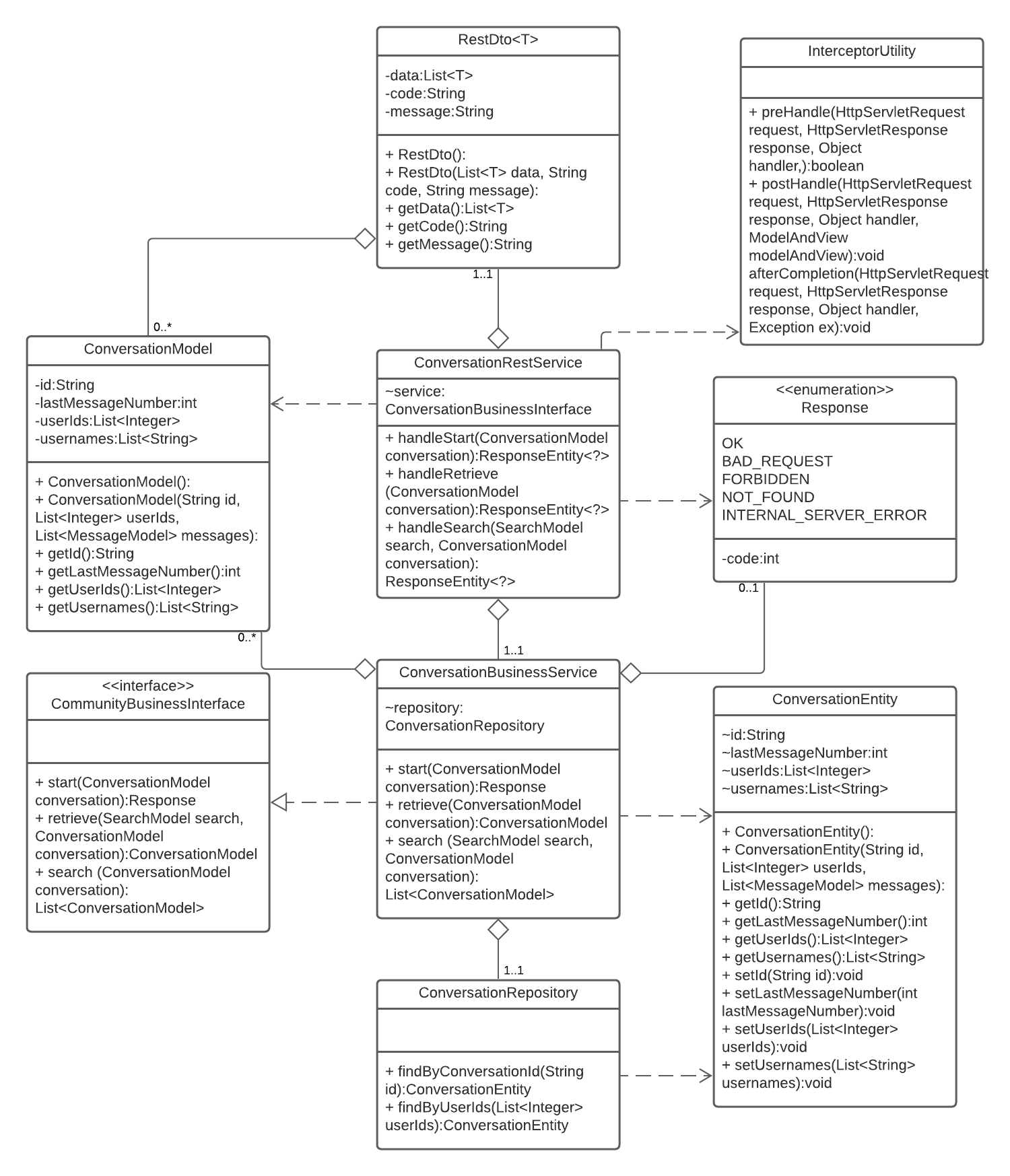
The UML class diagram below depicts the community module of the community service.



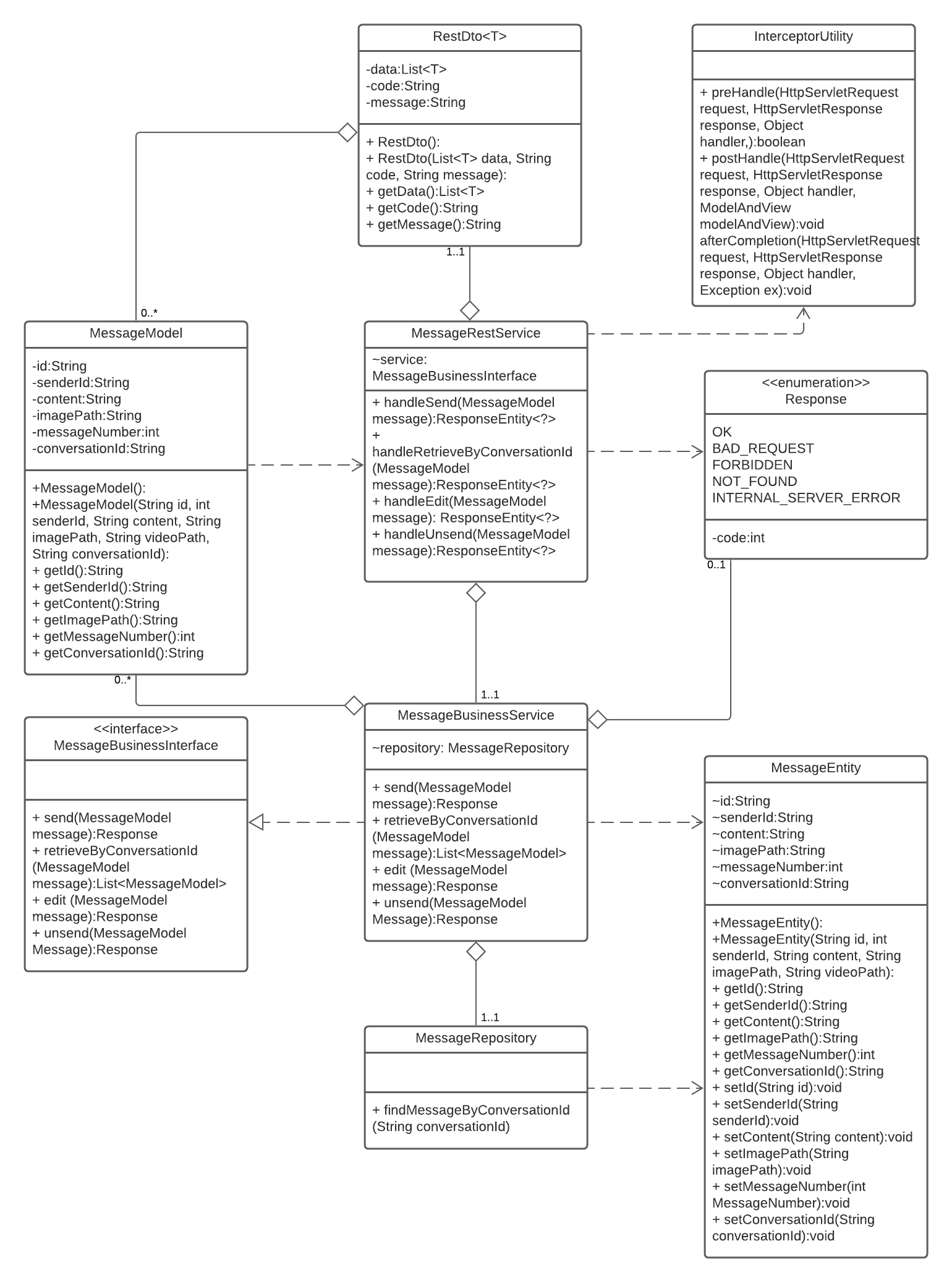
The following UML class diagram depicts the post module of the community service.



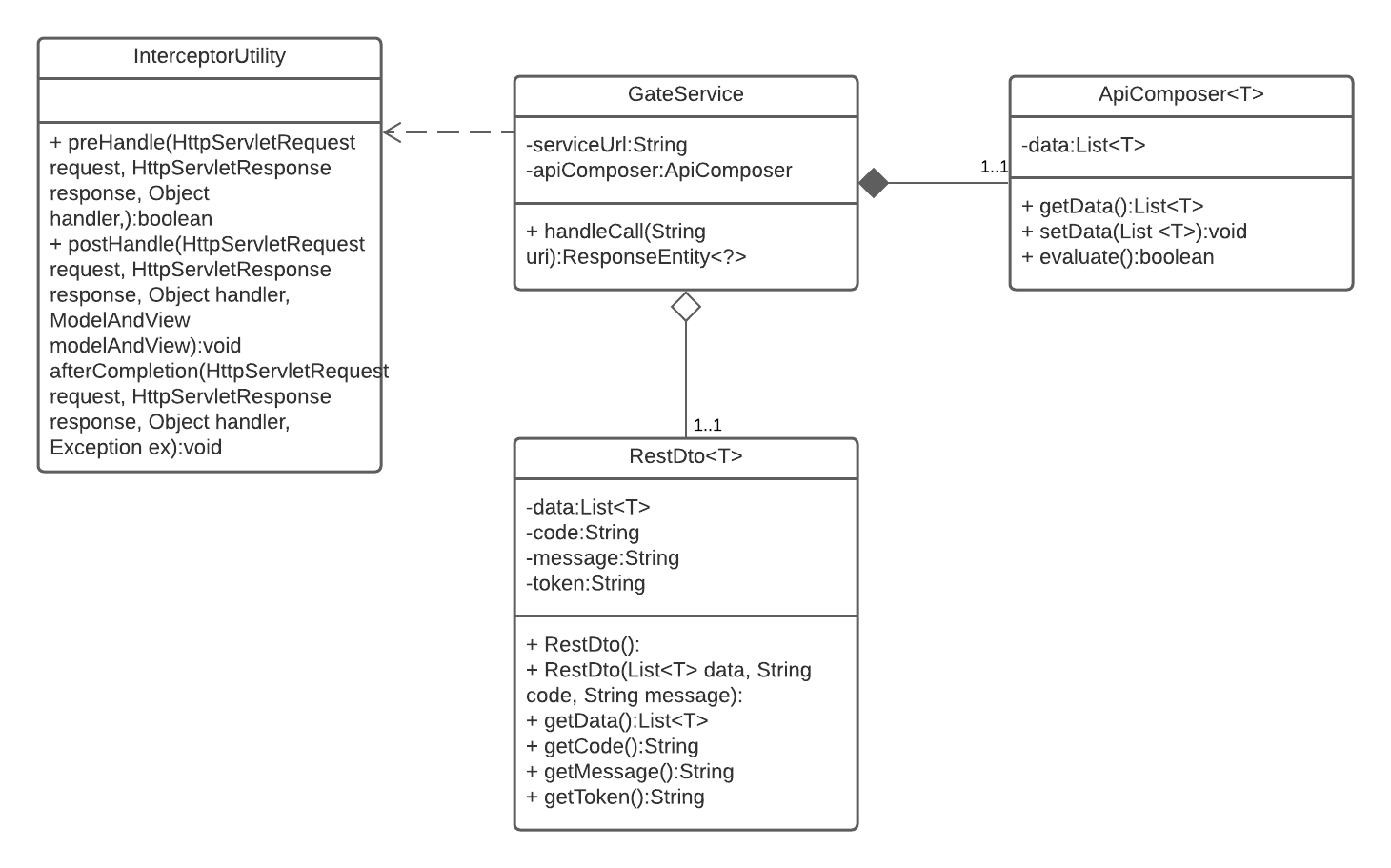
The UML class diagram below depicts the conversation module of the conversation service.



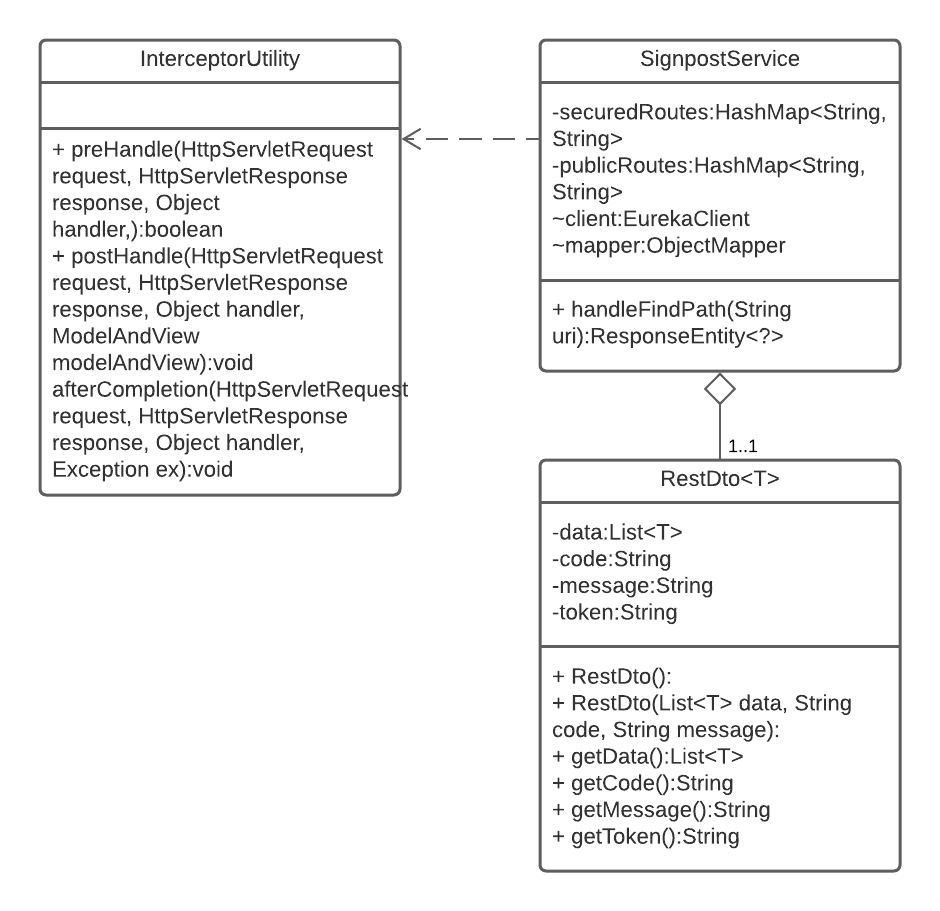
The following UML class diagram depicts the message module of the conversation service.



The following UML class diagram depicts the gateway service’s classes.



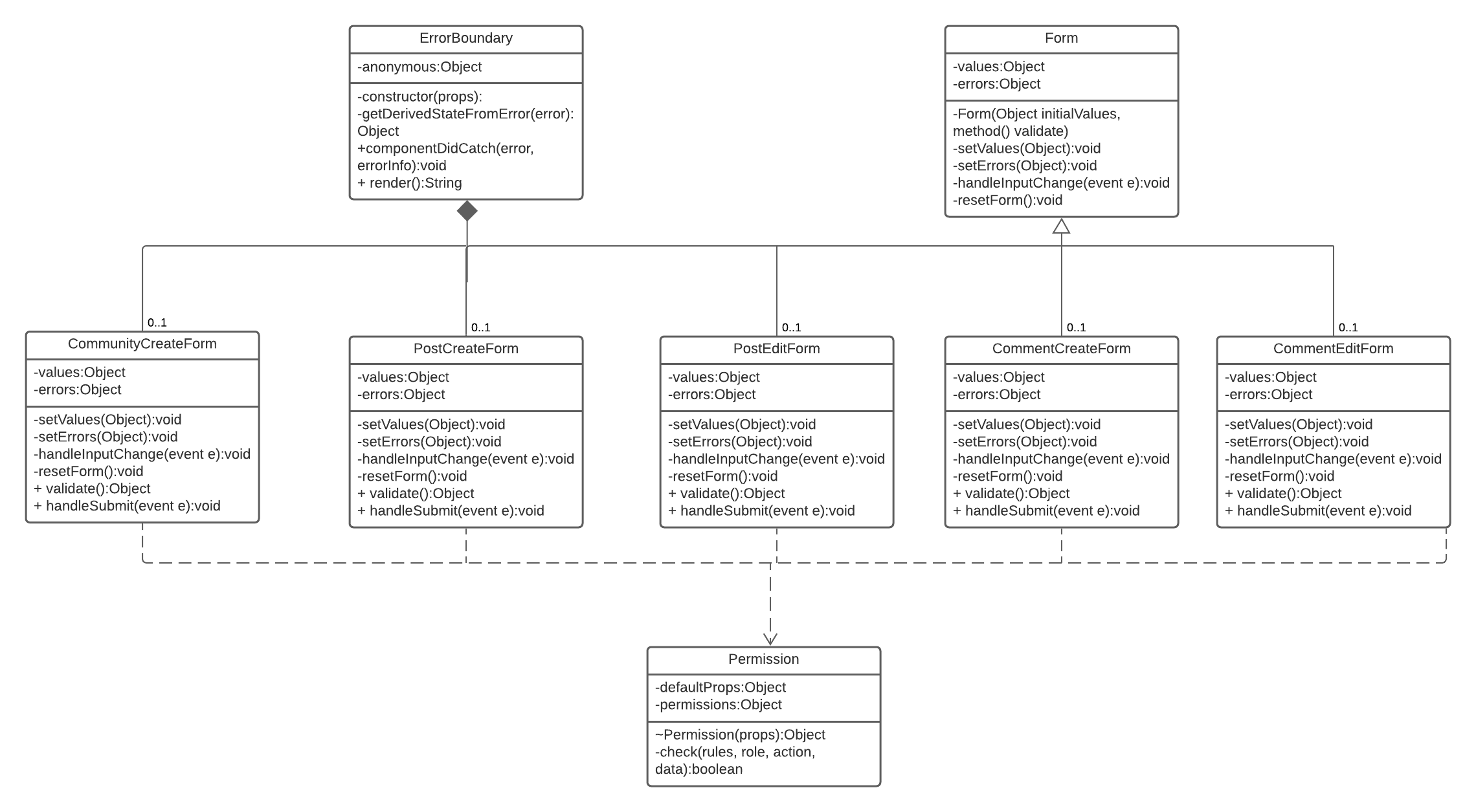
The following UML class diagram depicts the Discovery service’s methods.



The class diagrams that follow depict the state (similar to properties) and effects/behavior (similar to methods) that OpinionMarket’s react components use, as well as the relationships between components. The components are built using the React Hooks convention. Hooks allow functional components to ‘hook’ into React state and lifecycle meaning that functional components can now have state and behavior just like class components (React, 2020). React’s built-in useState hook defines component state that is saved between re-renders along with a set method for the state. The built-in useEffect hook provides a functional component version of lifecycle methods such as componentDidMount() and allows functional components to implement lifecycle behavior. Functional components powered by React Hooks are advantageous because they make reusing stateful logic between components easy and avoid putting unrelated logic in lifecycle methods (React, 2020).

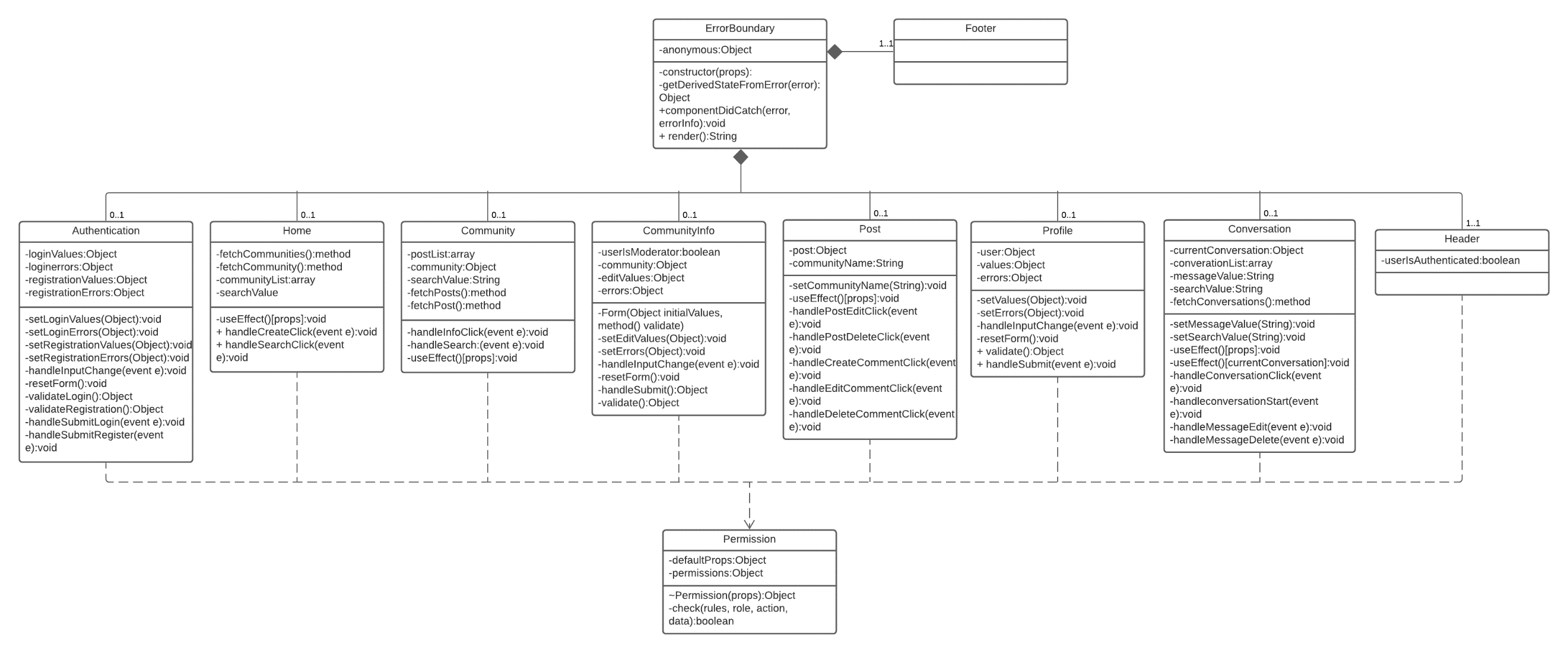
In the diagrams below, state hooks appear as class properties while effect hooks, setter methods for state, and other relevant behavior appear as class methods for each component. Components that import other components to leverage common behavior are considered to have a UML inheritance/generalization relationship with the component they import.

The following UML class diagram depicts the React application’s form components. User interface diagrams depicting the components visually can be found under the User Interface Design section of this document. Navigation between components is outlined in the Sitemap diagram located in the User Interface Design section of this document.

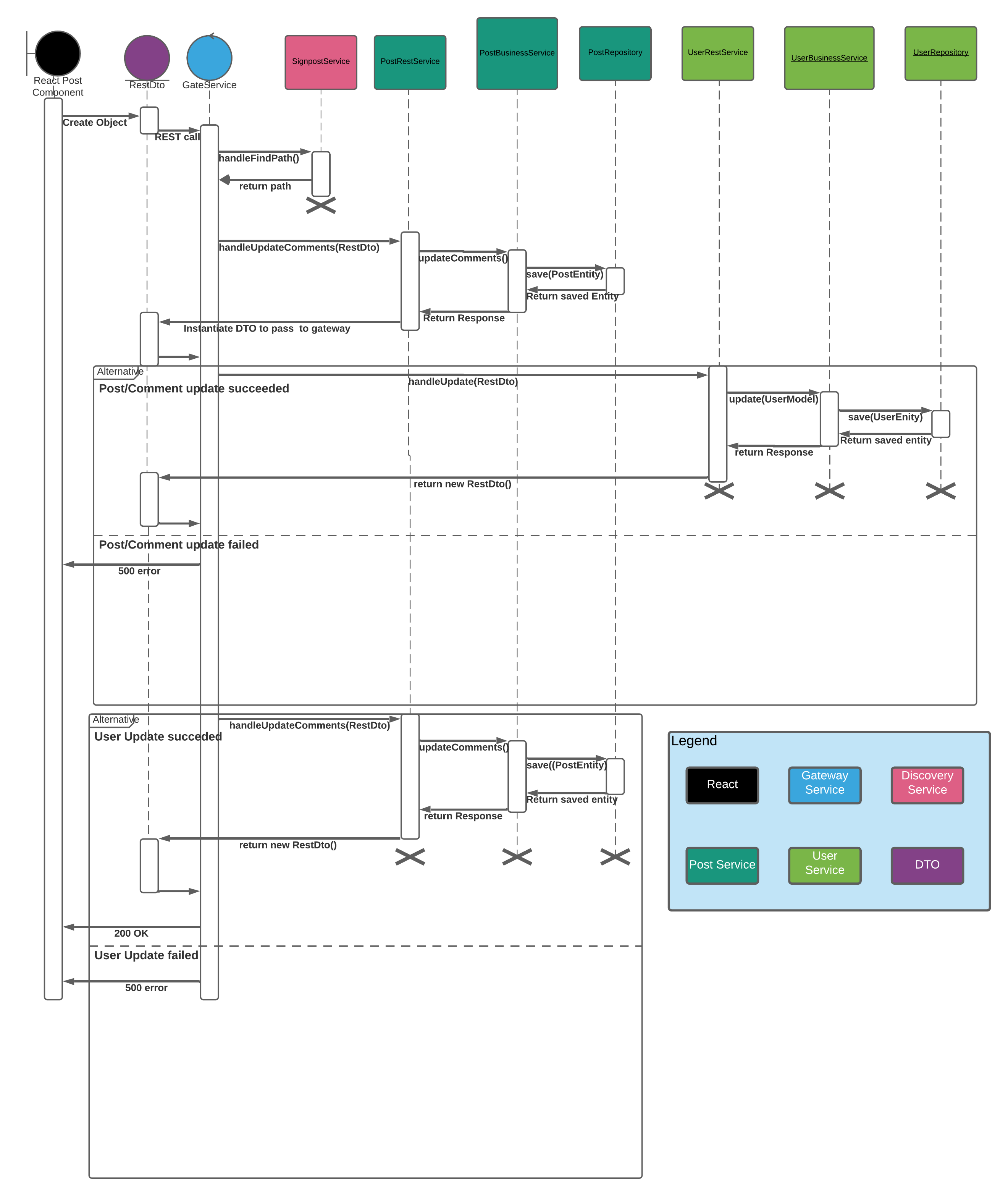


All form components import the Form component that defines common behavior and common state fields that all forms use. This is analogous to the Object-Oriented Programming concept of inheritance and has been depicted as such. Although the concrete form components can inherit from the same component they cannot be merged into one component because they must each implement their own additional behavior for form submission, map to different API actions, and appear different when rendered. All forms depend on the Permission component to evaluate user permissions and a composition relationship exists with the ErrorBoundary component wrapper because if the Wrapper component were deleted, its component subtree would be deleted as well.

The following UML class diagram depicts the non-form components of the client application. All components that will be rendered are considered to have a composition relationship with the ErrorBoundary wrapper because they are rendered as part of its subtree and cannot exist without it. All components except the footer depend on the Permission component to evaluate permissions.



The UML sequence diagram below outlines the project’s most complex in-scope flow. When a user votes on a post or comment the post or comment must be updated with the new point total and the User who authored the post or comment should also have their point total updated. This means that a database transaction is needed across two data management microservices. Since OpinionMarket is designed with a microservice architecture and the database-per-service pattern, a traditional database transaction is impossible. The employed solution is API composition. The gateway service is responsible for ensuring data consistency by creating an in-memory join of database results in cases where multiple databases must be involved in the same operation. The sequence diagram below depicts that if any required database operation fails, the entire operation is cancelled, and if necessary, rolled back. Note this design is not a perfect solution. If the rollback fails data consistency can no longer be maintained.



**Service API Design:**

Swagger was used to document the microservice’s APIs. See https://app.swaggerhub.com/apis/JoshV3742/Capstone/1.0.0 for the swagger documentation

**Security Design**

The security objectives the project must meet are authenticating users of the client application, preventing unauthorized access to the backend APIs, preventing users from performing actions they do not have permission to perform, and keeping users’ Personally Identifiable Information (PII) secure, both in-flight and at rest. Users of the client application will authenticate themselves with a username and password combination. Github’s OAuth2 will be used to secure the backend API. To ensure users can only perform the actions they have permission to perform (e.g. only a moderator of a community can edit the community’s description) the system will leverage role-based access control in the client application and implement business rules in the backend microservices as final security checks. The role-based access control in the react application will be implemented as a component that stores a list of permissions and a list of roles including what permissions each role has. Any permission that has not been explicitly granted to a role is implicitly denied in accordance with the implicit deny access control best practice. Some permissions are static meaning they require no parameters to evaluate while others are dynamic meaning some members of a role may perform the underlying action but others may not, depending on context, (e.g. A user can edit their own profile but not the profile of another user). Password hashing will be performed by PBKDF2 an algorithm that is deterministic (hashing the same password always produces the same hash), cannot be reversed, has high entropy (a small change to the passwords produces a vastly different hash), and is collision resistant (Millington, 2020). This will prevent user’s passwords from falling into the hands of malicious actors.

**NFR’s**

The application has two non-functional requirements (NFRs). The NFRs are included in the project requirements document and in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Issue Key** | **As a/an <actor>** | **I would like to <action>** | **So that <outcome>** |
| CAP-239 | As a system, | As a system, I want to prevent a user who is not logged in from accessing any secured screen (see the sitemap for a comprehensive list of which screens are secured), | so that I can ensure the confidentiality of sensitive information I store |
| CAP-238 | As a system, | I want to display error message #46 if any required database is offline when an attempt is made to access it regardless of the request, | so that I can hide my technologies |

**CAP-239 Solution**

The client app will use the React-Router node package to establish routes to each screen component. By default, however, users will only be able to visit the Authentication screen (Screen designs can be found in the User Interface Design section of this document). This will be accomplished using role-based access control. Every user will have a role and every role will have a list of permissions. Permissions will define which roles may navigate to routes or perform operations. The roles defined will be visitor and user. There is no need for a moderator role because moderators are per community, meaning those permissions are conditional user permissions. Only after a user has successfully logged-in or registered on the Authentication screen will they be given the user role and have permissions to visit any screen. Therefore, only users who have authenticated themselves will be able to access secured screens (secured screens are denoted in the Sitemap Diagram section of this document).

**CAP-238 Solution**

The backend uses the Data Transfer Object (DTO) design pattern to bundle the appropriate data, code, and message that it sends with its HTTPS responses to the client application. If a database exception is thrown in one of the data management microservices, it will send a DTO that includes a code of 500 meaning that there was a server error. When the gateway service receives this code, it will forward it to the client application. When the client app sees a 500 code in an API response, it will throw an exception. React version 16 and above supports using class components as error boundaries. An error boundary is a component that defines the componentDidCatch() lifecycle method. Error boundaries gracefully handle JavaScript errors in their child component tree. All components will be wrapped in an error boundary component called ErrorBoundary. In this way the correct error message will be displayed when a request is made while the database is down, meeting the NFR.

**Operational Support Design:**

The application will use AWS as its cloud solution for both the backend microservices and the frontend client application. The microservices will run in EC2 containers while the frontend application will run in Elastic Beanstalk. Both AWS services are designed to integrate with Amazon CloudWatch. CloudWatch is an AWS tool for monitoring and logging that is used to gain a view of operational health (Amazon, 2020). Another advantage of CloudWatch is that it can easily be used in conjunction with Loggly, a log management tool (Loggly, 2020). Log information will be written using the SLF4J logging framework and Log4J as the logging provider. On the backend, the interceptor design pattern will be leveraged to trace API calls and avoid repetition of logging statements.

A challenge related to logging is that the project currently lacks a solution to aggregate log statements across the different microservices. This is a noted challenge of microservice architecture because debugging can now involve reading multiple sets of logs (Nemer, 2019). The latest on this project risk can be found in the relevant entry in the Issues and Risk Log located in appendix A.

Appendix A – Technical Issue and Risk Log

The project’s risks are listed below

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Issues and Risk Log | | | | | | | | |
| **Issue or Risk** | **Description** | **Project Impact** | **Action Plan/Resolution** | **Owner** | **Importance** | **Date Entered** | **Date to Review** | **Date Resolved** |
| I/R | What is the issue or risk? | How will this impact scope, schedule, and cost? | How do you intend to deal with this issue? | Who manages this issue? |  |  |  |  |
| R | Use of unfamiliar technology, Spring Boot, delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | High | 09/09/20 | 11/29/20 | 10/15/20 |
| R | Use of unfamiliar technology MongoDB delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | High | 09/09/20 | 11/29/20 | 10/15/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar React delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | High | 09/09/20 | 11/29/20 | 10/29/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar technology Redux delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | Medium | 09/09/20 | 11/29/20 | 10/29/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar technology Material-UI delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | Medium | 09/09/20 | 11/29/20 | 10/29/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar technology Docker delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | Medium | 09/09/20 | 11/29/20 | 10/29/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar technology, AWS delays the project | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the technology | Josh Van de Walle | Medium | 10/5/20 | 11/29/20 | 11/05/20 |  |  |  |  |  |  |  |  |
| R | Use of unfamiliar architecture microservices | Features would be pushed out of scope to accommodate the delay | Proof of concepts using the architecture | Josh Van de Walle | Medium | 09/09/20 | 11/29/20 | 10/25/20 |  |  |  |  |  |  |  |  |
| R | CORS must be enabled in Spring boot to communicate with React | A different backend framework would need to be used | Proof of concept solving the problem | Josh Van de Walle | Low | 10/22/20 | 11/29/29 | 10/28/20 |  |  |  |  |  |  |  |  |
| R | Unfamiliarity with early phases of SDLC delays the project | Bad design cripples the project | Researching SDLC | Josh Van de Walle | Low | 09/28/20 | 11/29/20 | 10/5/20 |  |  |  |  |  |  |  |  |
| R | AWS will prove too expensive to use | Another cloud host would have to be selected or the Docker containers would need to be run locally | Careful monitoring of AWS billing and pricing. Using multiple AWS accounts. | Josh Van de Walle | Medium | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | No solution in place to aggregate logs across microservices | More time will be spent debugging possibly pushing features out of scope | Log aggregation options will be explored and evaluated | Josh Van de Walle | Medium | 12/04 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | Code loss | Completed code will be lost due to machine failure | All code will be pushed to Github and, in some cases, to Dockerhub. | Josh Van de Walle | Low | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | Scope creep | The project scope grows to such an extent that in-scope features are delayed | User stories for each sprint will be selected at the beginning of the sprint, and no code will be written that isn’t for a user story included in that sprint. | Josh Van de Walle | Low | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | Inadequate testing | With too little testing, the code is buggy and incomplete. | Test cases for a sprint will be written before the code for that sprint. Code will be rigorously testedto ensure it is bug-free. | Josh Van de Walle | Low | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | Inadequate code comments | Insufficient code comments make the application hard to maintain. | All classes and methods have Javadoc or Javadoc style comments added to them. All lines of code will have some comment about what they do. | Josh Van de Walle | Low | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | Features are not traced between major artifacts such as requirements, architecture document, code module, and test plan. | Lack of traceability masks actual status of project requirements. | A traceability matrix will be maintained and submitted with each code drop. | Josh Van de Walle | Low | 12/02/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |
| R | COVID-19 | Being forced to work from quarantine | Wearing a mask and social distancing | Josh Van de Walle | Medium | 09/27/20 | 1/31/20 | Unresolved |  |  |  |  |  |  |  |  |

Appendix B – References

Amazon. (2020). *Amazon CloudWatch.* Aws.amazon.com. https://aws.amazon.com/cloudwatch/

Atkins, G., & Pavlovic, U. (2020, September 11). *Docker: Top 7 Benefits of Containerization.* Hentsu.com. https://hentsu.com/docker-containers-top-7-benefits/

Loggly. (2020). *Easy AWS Log Analysis.* Loggly.com <https://www.loggly.com/product/aws-log-analysis/>

Millington, S. (2020, September 22). *Hashing a Password in Java.* Baeldung.com https://www.baeldung.com/java-password-hashing

mongoDB. (2020). *Advantages of MongoDB.* Mongodb.com. https://www.mongodb.com/advantages-of-mongodb.

Nemer, J. (2019, November 13). *What are microservices?* Cloud Academy.https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

Otuagomah, D. (2020, June 22). *Comparing the most popular frontend JavaScript frameworks.* Educative.io. https://www.educative.io/blog/compare-popular-frontend-javascript-frameworks

React. (2020). *Introducing Hooks.* Reactjs.org https://reactjs.org/docs/hooks-intro.html

SimForm. (2020). *Best Frontend Frameworks for 2020 Web Development.* simform.com. https://www.simform.com/best-frontend-frameworks/

The Software House. (2020). *The State of Microservices 2020.* Tsh.io. https://tsh.io/state-of-microservices/

tutorialspoint. (2020). *MongoDB – Advantages.* Tutorialspoint.com. https://www.tutorialspoint.com/mongodb/mongodb\_advantages.htm.

tutorialspoint. (2020). *Spring Boot – Logging.* Tutorialspoint.com https://www.tutorialspoint.com/spring\_boot/spring\_boot\_logging.htm

Appendix C – External Resources

|  |  |
| --- | --- |
| **GIT URL:** | https://github.com/JoshVandeWalle/OpinionMarket |
| **Hosting URL:** | Coming Soon. |
| **Swagger URL:** | https://app.swaggerhub.com/apis/JoshV3742/Capstone/1.0.0 |