Collaborative Note-Taking App

Final Project Group 11
Student 1: Yiyou Pan - CS7319 - Off Campus
Student 2: Clifton Wallace - CS7319 - Off Campus
Student 3: Parker Brown - CS7319 - Off Campus

Project Description

Overview of the Application
Our Collaborative Note-Taking App empowers users to create, share, and edit notes seamlessly. Key features include:

- Secure user login and authentication.
- An dashboard to manage and access notes.
    a. Create, edit and delete notes.
- *(Pub/Sub only)* Real-time updates for shared notes, enabling effortless teamwork and collaboration.

Architectural Solutions

1. **REST Architecture**
    - Core foundation of the app.
    - Supports CRUD operations for notes (Create, Read, Update, Delete).
    - Handles user authentication, note organization, and basic synchronization via HTTP requests.
2. **Pub/Sub Architecture (Building on REST)**
    - Adds real-time collaboration via WebSocket.
    - Publishes updates to subscribed users whenever a shared note is edited.
    - Ensures instant synchronization, enhancing teamwork by delivering live updates on top of REST services.

Note: The Pub/Sub model extends the REST framework, ensuring scalability and compatibility across the app.

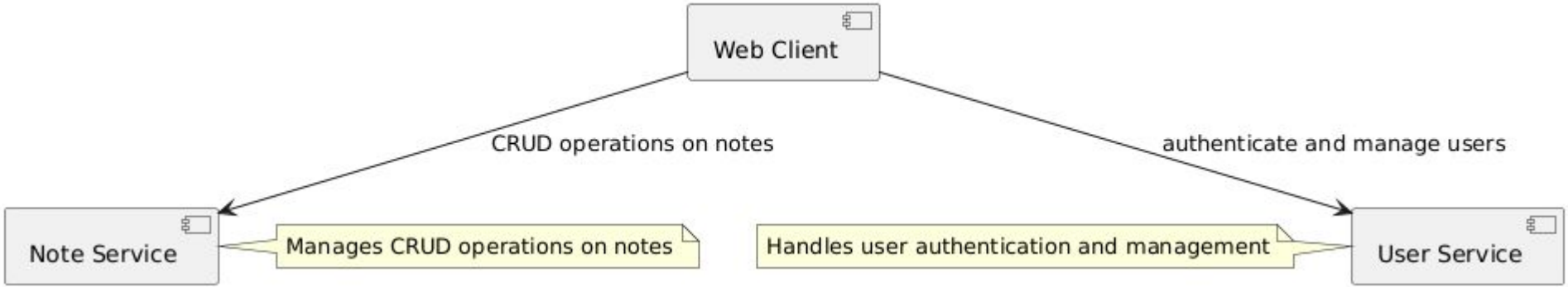
Architecture 1 - REST

Overview
The REST architecture forms the foundational layer of the Collaborative Note-Taking App, implemented using a microservices approach for enhanced scalability and maintainability.

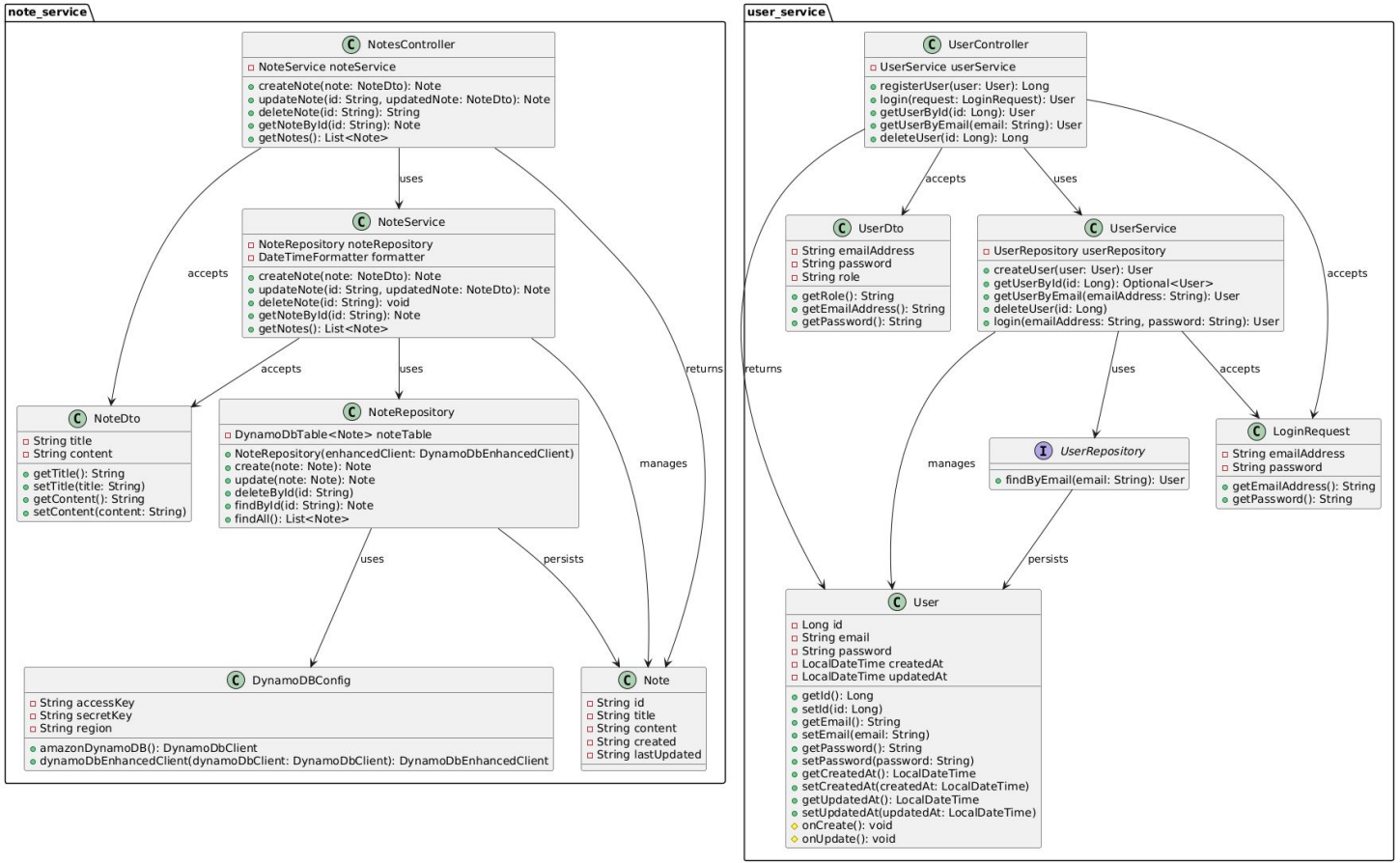
Key Features:

- **Microservices Design:** Each core feature (e.g., notes, users) is encapsulated in an independent service ensuring modularity
- **CRUD Operations:** Supports Create, Read, Update, and Delete actions for notes through standard HTTP methods (GET, POST, PUT, DELETE).
- **User Authentication:** Secure login and user management
- **Note Management:** Organizes user-created notes for easy access.
- **Stateless Communication:** Each request is self-contained, enabling scalability across services.

REST-Component Diagram



REST-Class Diagram



REST-Component Mapping

Component: note_service
Provides core functionality for managing notes, including API endpoints, business logic, data persistence, and configuration for database interactions.

Classes:

NotesController: Manages API endpoints for note operations, handling HTTP requests and responses, and delegating tasks to the service layer.

NoteService**:** Handles business logic for managing notes

NoteRepository: Interacts with the database to perform CRUD operations on notes

NoteDto**:** Data Transfer Object to encapsulate note data

Note**:** Represents the note entity to map to database

DynamoDBConfig: Configuration for connecting to DynamoDB database

Component: user_service
Provides core functionality for managing notes, including API endpoints, business logic, data persistence, and configuration for database interactions.

Classes:

UserController: Manages API endpoints for user operations, handling HTTP requests and responses, and delegating tasks to the service layer.

UserService: Handles business logic for managing users

UserRepository: Interacts with the database to perform CRUD operations on users

LoginRequest: Encapsulates user credentials like username and password.

UserDto: Data Transfer Object to encapsulate user data

User: Represents the user entity to map to database

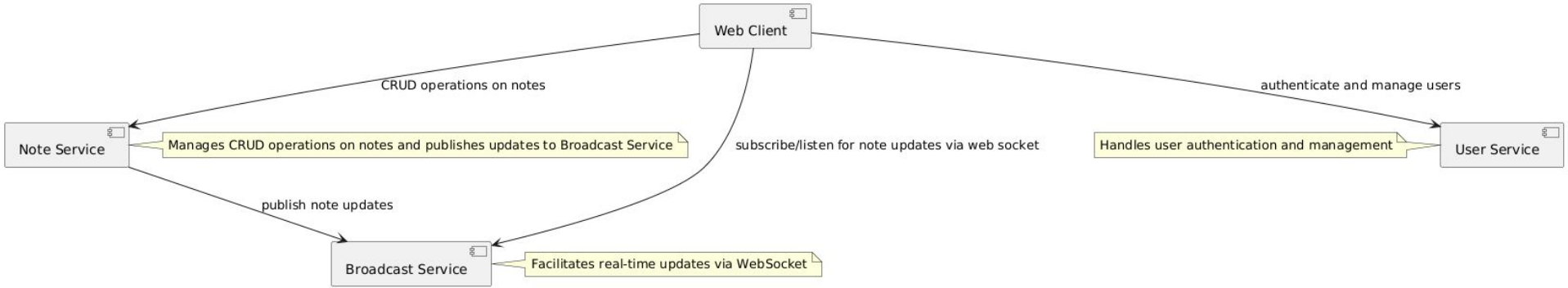
Architecture 2 - Pub/Sub

Overview
Building on the foundational REST architecture, the Pub/Sub model introduces real-time collaboration using AWS Simple Notification Service (SNS), AWS Simple Queue Service (SQS), and WebSocket technology to ensure instant updates across users.

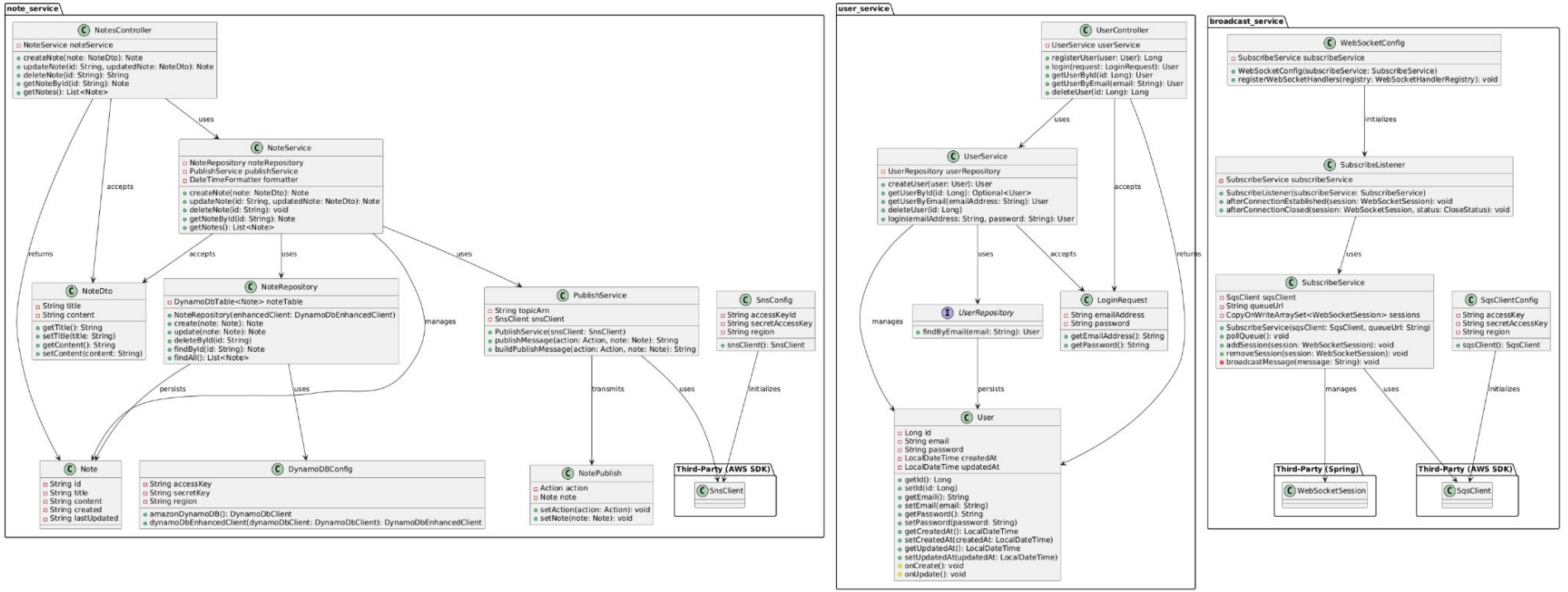
Key Features:

- **Real-Time Messaging:**
  - **SNS:** Publishes note updates to subscribed users.
  - **SQS:** Ensures reliable message delivery with decoupling between services.
  - **WebSockets:** Enables low-latency, bi-directional communication for real-time updates.
- **Integration with REST:** Leverages REST APIs for initial note synchronization, authentication, and other CRUD operations.
- **Efficient Collaboration:** Updates propagate instantly to all connected users when a shared note is edited.

Pub/Sub-Component Diagram



Pub/Sub-Class Diagram



Pub/Sub-Component Mapping

Component: note_service
Provides core functionality for managing notes, including API endpoints, business logic, data persistence, and configuration for database interactions.

Classes:

PublishService: Publishes message updates to Sns

SnsClient: Configuration for connecting to Sns

NotePublish: Data Transfer Object sent to Sns containing note updates

Existing Classes From REST architecture:

NotesController: Manages API endpoints for note operations, handling HTTP requests and responses, and delegating tasks to the service layer.

NoteService: Handles business logic for managing notes

NoteRepository: Interacts with the database to perform CRUD operations on notes

NoteDto: Data Transfer Object to encapsulate note data

Note: Represents the note entity to map to database

DynamoDBConfig: Configuration for connecting to DynamoDB database

Component: user_service
Provides core functionality for managing notes, including API endpoints, business logic, data persistence, and configuration for database interactions.

Classes:

UserController: Manages API endpoints for user operations, handling HTTP requests and responses, and delegating tasks to the service layer.

UserService: Handles business logic for managing users

UserRepository: Interacts with the database to perform CRUD operations on users

LoginRequest: Encapsulates user credentials like username and password.

UserDto: Data Transfer Object to encapsulate user data

User: Represents the user entity to map to database

Pub/Sub-Component Mapping

Component: broadcast_service
Enables clients to connect via WebSocket for receiving real-time updates on notes

Classes:

SubscribeListener: Manages WebSocket sessions, handling client connections, disconnections, and message subscription

SubscribeService**:** Handles the distribution of real-time updates to clients connected via WebSocket.

WebSocketConfig: Configures WebSocket endpoints, connection settings, and listener integration

SqsClientConfig**:** Configures the connection to AWS SQS for handling message queueing and delivery to WebSocket clients

REST Architecture

Pros:

✅ **Scalable:** Handles a large number of users and requests efficiently, making it ideal for growing apps.

✅ **Client Flexibility:** Supports multiple client types (web, mobile, etc.) due to the separation of client and server.

✅ **Simplicity:** Stateless communication ensures simplicity and predictability in API interactions.

✅ **Modularity:** Each part is separate, so we can update or scale parts without affecting the whole app

Cons:

❌ **Not Real-Time:** Users may need to refresh the page to see updates, which conflicts with real-time collaboration goals.

❌ **Managing Related Data:** Stateless design makes it challenging to manage related or dependent data across requests.

Pub/Sub Architecture

Pros:

✅ **Built on REST:** Extends the existing REST architecture, maintaining compatibility with existing APIs while adding real-time capabilities.

✅ **Real-Time Communication:** Enables real time updates using WebSocket, solving REST's limitation for real-time collaboration.

✅ **Scalable Messaging:** Uses AWS SNS and SQS for reliable, scalable message delivery.

Cons:

❌ **Increased Complexity:** Adding WebSocket, SNS, and SQS introduces additional layers and dependencies.

❌ **Resource Intensive:** Requires more resources to deploy and manage the infrastructure for real-time updates.

❌ **Debugging Challenges:** Interactions between REST and Pub/Sub layers make identifying and fixing issues more complex.

Rationale of Selection - Pub/Sub

We chose the Pub/Sub architecture because it best meets the needs of our collaborative note-taking app.

Key Reasons for Choosing Pub/Sub:

- **Real-Time Collaboration:** Pub/Sub, using WebSockets, provides instant updates for shared notes, solving REST's limitation for real-time communication.
- **Reliability:** Built on top of REST, it ensures that if the real-time update feature encounters issues, core functionalities like login and basic CRUD operations remain unaffected.
- **Scalability:** The architecture is designed to handle high user concurrency. For example, the messaging system (using AWS SNS and SQS) scales independently without impacting the app's overall performance.
- **Modularity:** Each component (e.g., REST APIs, real-time updates) operates independently, allowing targeted updates or improvements without affecting the rest of the app.

Why Pub/Sub Over REST Alone?
While REST is great for simplicity and small apps, it lacks the real-time communication needed for effective collaboration. Pub/Sub extends REST's capabilities, making it a better choice for handling real-time updates and ensuring the app grows flexibly and reliably.

Most Importantly…..
Designing and building Pub/Sub was more fun! Real-time collaboration, WebSocket integration, and crafting a dynamic messaging system offered exciting challenges and opportunities to innovate.

Risk Analysis

REST Architecture:

Data loss is a concern if notes fail to save due to server issues, so automatic backups and error recovery are essential. Security is another risk, as user data could be vulnerable to hacking; strong encryption and secure login methods like HTTPS will help mitigate this.Real-time sync failures may occur, preventing updates from appearing instantly for all users, which can be addressed by thorough testing and fallback systems. High traffic could slow down the app, so scalable servers and optimized code are necessary.

Pub/Sub Architecture:

Managing concurrency between the client/server under two architectures can be a challenging undertaking. The client still sends updates via REST, but receives updates using a separate mechanism (e.g. Websockets). Over the course of development, we were faced with synchronization issues because updates from the broadcast service would overwrite any updates typed after the initial update was sent. Any service that is concurrent in nature can introduce unforeseen consequences without extensive testing, thus we'd like to recognize the importance of testing components that may seem fundamentally solid in a REST environment.

Risk Analysis

For REST, studies show that its stateless design makes it easy to use, but it struggles with heavy traffic. In tests, REST systems experienced significant delays when handling over 100 simultaneous requests, with response times increasing by up to 70% compared to lighter loads (Costa et al., 2016). REST's use of a uniform interface also limits its ability to handle complex operations effectively.

Conventional communication techniques are a thing of the past according to Dubey, as the need for responsive techniques become more and more critical. Websockets provide a well-tested and secure mechanism for sending data bidirectionally and without latency concerns, providing a beneficial avenue for our application to communicate with other clients at lightning speed (Dubey, 2023).

References

Costa, B., Pires, P. F., Delicato, F. C., & Merson, P. (2016). Evaluating REST architectures—Approach, tooling and guidelines. Journal of Systems and Software, 112, 156–180. https://doi.org/10.1016/j.jss.2015.09.039

Dubey, A. (n.d.). Enhancing Real Time Communication and Efficiency With Websocket. In International Research Journal of Engineering and Technology. Retrieved November 20, 2024, from https://www.irjet.net/archives/V10/i8/IRJET-V10I8147.pdf

Thank You