Backend Design Story

Project Background:

The backend system is designed to support the real-time management and collaboration features for Trello's boards, lists, and cards. The frontend is built using Angular v15, while the backend is developed using Node.js with Express.js, providing RESTful APIs. MongoDB is used as the primary database, ensuring efficient handling of large datasets, while AWS services (including EC2 and S3) are utilized for scalable processing power and storage of static assets.

Team Size:

Development Teams:

- **Full-stack Developers:** 4
    - Frontend-focused: 2
    - Backend-focused: 2
- **Quality Assurance Engineers:** 3
- **DevOps Engineers:** 2

UI/UX & Product Owners:

- **UI Designer:** 1
- **UX Designer:** 1
- **Product Owner:** 2

Business Teams:

- **Business Analysts:** 2
- **Project Managers:** 2

Development Teams intro:

- **Full-stack Developers:**
    - **Frontend-focused:** Responsible for implementing and optimizing the user interface in Angular, ensuring a seamless and responsive user experience across different devices.
    - **Backend-focused:** Developed and maintained RESTful APIs using Node.js and Express.js, focusing on efficient data handling and secure communication with the database.
- **Quality Assurance Engineers:** Conducted comprehensive testing, including unit, integration, and end-to-end testing, to ensure the reliability and security of the application before deployment.
- **DevOps Engineers:** Managed the CI/CD pipeline, automated deployments, and ensured the application's scalability and availability by configuring and monitoring AWS infrastructure.

UI/UX & Product Owners:

- **UI Designer:** Designed the visual components of Trello, ensuring consistency with brand guidelines and creating an intuitive interface for end-users.
- **UX Designer:** Focused on user research and experience design, ensuring that the platform's features met user needs and provided an optimal user journey.
- **Product Owner:** Prioritized features and requirements based on business goals and user feedback, coordinating between the development team and stakeholders to ensure alignment and successful delivery.

Business Teams into:

- **Business Analysts:** Analyzed user requirements and market trends to provide insights and recommendations for new features and enhancements, ensuring the platform remained competitive and user-centric.
- **Project Managers:** Oversaw project timelines, managed resources, and ensured that the project stayed on track to meet deadlines and deliverables, while facilitating communication between teams.

Overall Architecture Design

System Architecture Overview:

The backend system of Trello originally operated on a monolithic architecture, where all components were tightly integrated into a single codebase. As Trello grew in complexity and user base, we recognized the need for a more scalable and maintainable solution. To address these challenges, we transitioned the system to a microservices architecture.

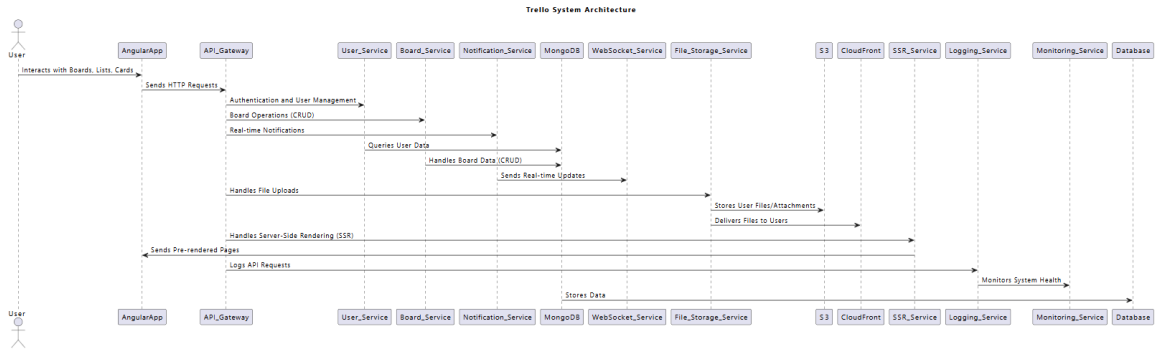
Key Components:

- **Monolithic to Microservices Transition:**
    - **Original Monolithic Architecture:** Initially, Trello's backend was built as a single, unified application where all functionalities, such as user management, board management, and notification services, were bundled together. This architecture, while simple, became increasingly difficult to manage as the platform scaled.
    - **Transition to Microservices:** To improve scalability and maintainability, we refactored the backend into a microservices architecture. Each microservice was designed to handle specific business logic independently, such as user management, board management, and real-time notifications. This decoupling allowed us to scale individual services based on demand without impacting the entire system.
- **Express.js API Gateway:**
    - **Function:** Acts as the centralized entry point for all HTTP requests. It manages routing, authentication, and load balancing across the various microservices, ensuring that requests are efficiently processed and directed to the appropriate service.
    - **Why:** The API Gateway simplifies the overall architecture by centralizing cross-cutting concerns like security and routing. This makes the system easier to maintain and extend, and improves overall performance.
- **Microservices:**
    - **User Management Service:** Handles user authentication, profile management, and role-based access control. This service is crucial for ensuring secure and personalized user experiences.
    - **Board Management Service:** Manages the creation, updating, and deletion of boards, lists, and cards. This service was optimized for performance, particularly in handling large boards with hundreds of cards.

    - **Notification Service:** Manages real-time notifications and updates, ensuring that users are promptly informed of changes and updates to boards they are part of.
- **Data Management with MongoDB:**
    - **Function:** MongoDB remains the primary database, now interacting with each microservice as needed. Data handling is optimized through the use of compound indexes, aggregation pipelines, and sharding to ensure scalability and performance across large datasets.
    - **Why:** MongoDB's flexibility and ability to scale with the microservices architecture is critical for efficiently managing the growing volume of data generated by Trello users. The use of sharding and replication further enhances the system's ability to handle large-scale operations.
- **AWS Services:**
    - **EC2 Instances:** Provide scalable computing resources, enabling each microservice to scale independently based on demand.
    - **S3 for Static Assets:** Used for storing static files like images and documents, ensuring quick and reliable access.
    - **Lambda for Serverless Operations:** Handles specific tasks such as image processing, which benefits from serverless execution, reducing costs and improving efficiency.

Integration with Frontend Technologies:

- **Server-Side Rendering (SSR) with Angular Universal:**
    - **Function:** Enhances SEO and reduces initial load times by pre-rendering Angular pages on the server before delivering them to the client.
    - **Why:** SSR is particularly beneficial for improving user experience by making pages load faster and ensuring that content is more accessible to search engines.



1. User Interaction:

User: The user interacts with Trello through the user interface provided by the Angular application. This interaction includes actions like creating, moving, or deleting cards, updating boards, and managing tasks. The user interface is designed to be intuitive and responsive, allowing real-time collaboration on tasks.

2. Frontend Application:

AngularApp: This is the frontend of the Trello application, built using Angular. It provides the visual interface for users to interact with boards, lists, and cards. Angular handles user input and updates the user interface based on the actions taken by the user.

How it Works: When a user interacts with the Angular app (e.g., dragging a card to a different list), the Angular app processes this input and sends an HTTP request to the backend through the API Gateway. Angular also handles rendering data received from the backend, ensuring the user interface is always up-to-date with the latest information.

3. API Gateway (Express.js)

API_Gateway: The API Gateway is the central entry point for all requests coming from the frontend. It is implemented using Express.js, a Node.js web application framework. The API Gateway handles routing, security (such as authentication), and load balancing. It acts as a

middleman, directing requests to the appropriate microservices based on the URL and HTTP method.

How it Works: When the Angular app sends a request (e.g., to move a card), the API Gateway receives this request. It then routes the request to the correct service (e.g., the Board Service) based on the request's endpoint. The API Gateway also manages authentication by checking the user's JWT (JSON Web Token) to ensure that the request is authorized.

4. Microservices:

User_Service:

- **Function:** Manages user authentication, authorization, and profile management. This service is responsible for ensuring that only authenticated users can access specific features of the application.
- **How it Works:** When a user logs in, the Angular app sends the login credentials to the API Gateway, which forwards them to the User Service. The User Service validates the credentials, generates a JWT if the credentials are correct, and returns it to the frontend. The Angular app then uses this token for subsequent requests to verify the user's identity.

Board_Service:

- **Function:** Manages all operations related to boards, lists, and cards. This includes creating new boards, adding or moving cards, and deleting lists.
- **How it Works:** When the API Gateway receives a request related to boards (e.g., creating a new board), it forwards the request to the Board Service. The Board Service processes the request, interacts with the MongoDB database to store or retrieve board data, and then sends the response back to the API Gateway, which forwards it to the Angular app.

Notification_Service:

- **Function:** Handles real-time notifications and updates. This ensures that all users collaborating on the same board see updates as they happen, without needing to refresh the page.
- **How it Works:** When a user makes a change (e.g., moving a card), the Notification Service sends real-time updates to all other users connected to that board. This is typically done using WebSockets, which provide a continuous connection between the server and the clients for instant data transmission.

5. Data Management (MongoDB):

MongoDB: The primary database used for storing all application data, including user information, boards, lists, cards, and other relevant data. MongoDB is a NoSQL database, which means it stores data in a flexible, document-oriented format (JSON-like).

How it Works: Each microservice interacts with MongoDB to perform CRUD (Create, Read, Update, Delete) operations. For example, when a new card is added to a board, the Board Service sends a request to MongoDB to store this card in the appropriate collection. MongoDB's indexing and sharding capabilities ensure that the database can efficiently handle large amounts of data and scale as needed.

6. File Storage (S3 and CloudFront)

File_Storage_Service:

Function: Manages the uploading, storage, and retrieval of files (e.g., attachments, images) that users may add to their cards.

How it Works: When a user uploads a file, the Angular app sends the file to the API Gateway, which forwards it to the File Storage Service. This service then uploads the file to AWS S3, where it is stored securely. The file's URL is then saved in MongoDB, and the user can access the file via this URL.

S3 (Amazon Simple Storage Service):

Function: A scalable storage solution where all user-uploaded files are stored. S3 ensures that files are highly available and durable, meaning they are always accessible when needed.

How it Works: Files are uploaded to S3 by the File Storage Service. Each file is given a unique URL that can be used to retrieve it. S3 also integrates with CloudFront for efficient content delivery.

7. Server-Side Rendering (SSR)

SSR_Service:

- **Function:** Handles the server-side rendering of Angular pages to improve SEO and reduce initial load times.
- **How it Works:** When a user accesses Trello, the SSR Service generates the HTML for the page on the server and sends it to the client. This pre-rendered HTML allows the page to load faster and makes it easier for search engines to index the content.

8. Logging and Monitoring:

Logging_Service:

- **Function:** Logs all API requests, errors, and other significant events in the system.

- **How it Works:** Every time an API request is made or an error occurs, the Logging Service records this information in a log file or a centralized logging system. This helps developers diagnose issues and monitor system performance.

Flow Summary:

1. **User Interaction:** The user interacts with the frontend (AngularApp), performing various actions such as managing boards and cards.
2. **Request Handling:** The Angular app sends requests to the API Gateway, which routes them to the appropriate microservice (User_Service, Board_Service, etc.).
3. **Data Operations:** Microservices interact with MongoDB to store or retrieve data as needed. For file operations, the File Storage Service interacts with S3.
4. **Real-time Updates:** The Notification Service manages real-time communication, ensuring that all users see updates instantly.
5. **Response:** The processed data is sent back through the API Gateway to the Angular app, which updates the user interface accordingly.
6. **Monitoring:** The Logging and Monitoring services track system health and performance, ensuring that any issues are quickly identified and resolved.

Data Structure Design:

1. User Data Structure

Purpose: Store user-related information, including authentication details, profile information, and user-specific settings.

```
const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  passwordHash: { type: String, required: true },
  role: { type: String, enum: ['User', 'Admin'], default: 'User' },
  profilePictureUrl: { type: String },
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now },
  boards: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Board' }]
});

module.exports = mongoose.model('User', userSchema);
```

username: The user's unique identifier.

email: Email address, which is also unique for each user.

passwordHash: The hashed password for security.

role: Defines user roles, such as 'User' or 'Admin'.

profilePictureUrl: Stores the URL of the user’s profile picture, typically stored in S3.

boards: An array of references to the boards the user is part of, allowing quick access to a user's boards.

2. Card Data Structure

Purpose: Store the tasks or items within a list, including details such as descriptions, attachments, and assigned users.

```
const cardSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String },
  list: { type: mongoose.Schema.Types.ObjectId, ref: 'List',
      required: true },
  board: { type: mongoose.Schema.Types.ObjectId, ref: 'Board',
      required: true },
  position: { type: Number, default: 0 },
  assignedTo: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User'
      }],
  dueDate: { type: Date },
  labels: [{ type: String }],
  attachments: [{ type: String }],
  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment'
      }],
  createdAt: { type: Date, default: Date.now },
  updatedAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Card', cardSchema);
```

Explanation:

title: The name of the card, typically a brief summary of the task or item.

description: A more detailed description of the card's content or task.

list: A reference to the list to which this card belongs.

board: A reference to the board to which this card belongs.

position: Numeric value indicating the order of the card within the list.

assignedTo: An array of users assigned to this card, allowing for task delegation.

dueDate: Optional due date for the card's task.

labels: Array of labels for categorizing or tagging the card.

attachments: URLs pointing to files related to the card, stored in S3.

comments: An array of references to comments made on the card.

RESTful API Design:

Board Management

- **GET /boards/**
  - **Purpose**: Retrieve details of a specific board by board ID.
  - **Response**: Returns the board's details, including its title, description, members, and lists.
- **POST /boards**
  - **Purpose**: Create a new board.
  - **Request Body**: `{ "title": "New Project Board", "description": "Project management board" }`
  - **Response**: Returns the newly created board's details, including a unique board ID.
- **PUT /boards/**
  - **Purpose**: Update an existing board.
  - **Request Body**: `{ "title": "Updated Board Title" }`
  - **Response**: Returns the updated board details.
- **DELETE /boards/**
  - **Purpose**: Delete a board by its ID.
  - **Response**: Returns a success message indicating the board has been deleted.

List Management

- **GET /boards/**
  /lists
  - **Purpose**: Retrieve all lists associated with a specific board.
  - **Response**: Returns an array of lists, each with details such as title and position.

- **POST /boards/**
  /lists
  - **Purpose**: Create a new list within a board.
  - **Request Body**: `{ "title": "To Do" }`
  - **Response**: Returns the newly created list's details, including a unique list ID.
- **PUT /lists/**
  - **Purpose**: Update an existing list.
  - **Request Body**: `{ "title": "Updated List Title" }`
  - **Response**: Returns the updated list details.
- **DELETE /lists/**
  - **Purpose**: Delete a list by its ID.
  - **Response**: Returns a success message indicating the list has been deleted.

Card Management

- **GET /lists/**
  /cards
  - **Purpose**: Retrieve all cards within a specific list.
  - **Response**: Returns an array of cards, each with details such as title, description, and position.
- **POST /lists/**
  /cards
  - **Purpose**: Create a new card within a list.
  - **Request Body**: `{ "title": "New Task", "description": "Task details" }`
  - **Response**: Returns the newly created card's details, including a unique card ID.
- **PUT /cards/**
  - **Purpose**: Update an existing card.
  - **Request Body**: `{ "title": "Updated Task Title", "description": "Updated task details" }`
  - **Response**: Returns the updated card details.
- **DELETE /cards/**
  - **Purpose**: Delete a card by its ID.
  - **Response**: Returns a success message indicating the card has been deleted.

Example API usage:

Creating a New Board:

- **Endpoint:** `POST /boards`
- Request body:

```
{
  "title": "New Project Board",
  "description": "Board for managing new project tasks"
}
```

- Response body:

```
{
  "id": "5f2d1234abc567890",
  "title": "New Project Board",
  "description": "Board for managing new project tasks",
  "owner": "5f2d0987abc123456",
  "members": [],
  "lists": [],
  "createdAt": "2024-09-01T12:00:00Z"
}
```

Fetching all the cards in a list:

Endpoint: GET /lists/5f2d6789abc123456/cards

- **Response body:**

```
[
  {
    "id": "5f2d9876abc123456",
    "title": "Task 1",
    "description": "First task description",
    "position": 1,
    "assignedTo": ["5f2d1234abc567890"],
    "dueDate": "2024-09-10T12:00:00Z"
  },
  {
    "id": "5f2d8765abc123456",
    "title": "Task 2",
    "description": "Second task description",
    "position": 2,
    "assignedTo": ["5f2d0987abc567890"],
    "dueDate": "2024-09-15T12:00:00Z"
  }
]
```