```text

Title 1: Frontend (React)

• **Expected Inputs:**

* Source: User interaction via web browser.

* Format: Data entered into HTML form fields.

* Data Types and Constraints:

* `username`: String, required, minLength: 3, maxLength: 20.

* `password`: String, required, minLength: 8, must contain at least one uppercase letter, one lowercase letter, one number, and one special character.

* `email`: String, required, must be a valid email format (e.g., using regex).

* `firstName`: String, optional, maxLength: 50.

* `lastName`: String, optional, maxLength: 50.

* Validation: Client-side validation using JavaScript.

• **Expected Outputs:**

* Data Structure: JavaScript objects representing user input.

* Format: JSON (when sending data to the backend API).

* Destination: Backend API endpoints (`/register`, `/login`, `/welcome`).

* Display: Updates to the UI based on backend responses (success/error messages, redirection to welcome board).

• **Data Flow and Integration Points:**

* User enters data in React components (`RegistrationForm`, `LoginForm`).

* React components use `APIClient` to send data to backend API.

* Backend API processes data and returns a response.

* React components update the UI based on the response.

• **Architecture:**

* Component-based architecture using React.

* Uses React Hooks for state management and side effects.

* Utilizes a component library (e.g., Material UI, Ant Design) for UI elements.
```

• **Module/Component Responsibilities:**

* `RegistrationForm`: Handles user registration form, validation, and API call.

* `LoginForm`: Handles user login form, validation, and API call.

* `WelcomeBoard`: Displays welcome message and user-specific information after successful login.

* `AuthContext`: Manages authentication state (token storage, user login status) using React Context API.

* `APIClient`: Handles HTTP requests to the backend API (using `fetch` or `axios`).

• **Design Patterns:**

* Component-Based UI.

* Context API for state management.

• **Communication Protocols:**

* HTTPS for all communication with the backend.

* JSON for request and response bodies.

* Authentication: JWT (JSON Web Token) stored in `localStorage` or `sessionStorage`.

* Headers: `Content-Type: application/json`, `Authorization: Bearer ` (for authenticated requests).

* Error Handling: Display user-friendly error messages based on backend API responses.

• **Data Specifications:**

• **Database Schema:** N/A (Frontend does not directly interact with the database).

• **Data Structures:**

* `RegistrationData`: JavaScript object representing user registration data.

* `LoginData`: JavaScript object representing user login data.

* `UserData`: JavaScript object representing user profile information.

• **Validation Rules:**

* `username`: Required, alphanumeric characters only, minLength: 3, maxLength: 20.

* `password`: Required, minLength: 8, must contain at least one uppercase letter, one lowercase letter, one number, and one special character (regex validation).

* `email`: Required, valid email format (regex validation).

• **Security Specifications:**

* Storing JWT in `localStorage` or `sessionStorage` with appropriate security considerations (e.g., using `sessionStorage` and setting appropriate expiration times).

* Protecting against XSS attacks by sanitizing user input and using appropriate React rendering techniques.

* Using HTTPS for all communication.

• **Runtime Environment and Deployment Details:**

* Runtime Environment: Web browser (Chrome, Firefox, Safari, Edge).

* Dependencies: React, React Router, Axios/Fetch, UI Component Library (e.g., Material UI, Ant Design).

* Deployment: Static files deployed to AWS S3 and served via CloudFront CDN.

Title 2: Backend (Python - e.g., Flask/FastAPI)

• **Expected Inputs:**

* Source: Frontend API requests (JSON format).

* Endpoints: `/register`, `/login`, `/welcome`.

* Data Types and Constraints:

* `/register`:

* `username`: String, required, minLength: 3, maxLength: 20.

* `password`: String, required, minLength: 8.

* `email`: String, required, valid email format.

* `firstName`: String, optional, maxLength: 50.

* `lastName`: String, optional, maxLength: 50.

* `/login`:

* `username`: String, required.

* `password`: String, required.

* `/welcome`:

* JWT token in the `Authorization` header.

• **Expected Outputs:**

* Data Structure: JSON responses.

* Format: JSON.

* Destination: Frontend.

* Responses:

* `/register`: Success/failure message, user ID.

* `/login`: Success/failure message, JWT token.

* `/welcome`: Welcome message, user profile information.

• **Data Flow and Integration Points:**

* Receives requests from the frontend.

* Validates user credentials.

* Interacts with the MongoDB database.

* Generates and manages user sessions/tokens (JWT).

* Sends responses back to the frontend.

• **Architecture:**

* RESTful API using Flask or FastAPI.

* Modular design with separate modules for authentication, user management, and database interaction.

• **Module/Component Responsibilities:**

* `AuthService`: Handles user authentication and authorization (JWT generation, password hashing).

* `UserService`: Manages user data (creation, retrieval, updates).

* `RouteHandlers`: Defines API endpoints and handles request/response logic.

* `DatabaseConnector`: Manages connection to the MongoDB database.

* `Config`: Loads and manages application configuration.

• **Design Patterns:**

* MVC (Model-View-Controller) or similar pattern.

* Repository pattern for database access.

• **Communication Protocols:**

* HTTPS for all communication.

* RESTful API.

* JSON for request and response bodies.

* Authentication: JWT (JSON Web Token).

* Headers: `Content-Type: application/json`, `Authorization: Bearer ` (for authenticated requests).

* Error Handling: Standard HTTP status codes (e.g., 200, 201, 400, 401, 409, 500).

• **Data Specifications:**

• **Database Schema:**

* `users` collection in MongoDB (see Database section for details).

• **Data Structures:**

* `User`: Python object representing a user.

* `RegistrationRequest`: Python object representing registration data.

* `LoginRequest`: Python object representing login data.

• **Validation Rules:**

* Using libraries like `marshmallow` or `pydantic` for input validation.

* `username`: Required, alphanumeric characters only, minLength: 3, maxLength: 20.

* `password`: Required, minLength: 8.

* `email`: Required, valid email format.

• **Security Specifications:**

* Password hashing using `bcrypt`.

* JWT authentication.

* Input validation to prevent injection attacks.

* HTTPS for all communication.

* Rate limiting to prevent brute-force attacks.

• **Runtime Environment and Deployment Details:**

* Runtime Environment: Python 3.8 or higher.

* Dependencies: Flask/FastAPI, pymongo, bcrypt, python-dotenv, PyJWT.

* Deployment: AWS EC2, Lambda with API Gateway, or ECS.

* WSGI server (e.g., Gunicorn, uWSGI).

Title 3: Database (MongoDB)

• **Expected Inputs:**

* Source: Backend API requests (from `UserService` and `AuthService`).

* Data Types and Constraints:

* `username`: String, required, unique, minLength: 3, maxLength: 20.

* `email`: String, required, unique, valid email format.

* `password`: String, required (hashed).

* `firstName`: String, optional, maxLength: 50.

* `lastName`: String, optional, maxLength: 50.

* `createdAt`: Date, default value: current timestamp.

* `updatedAt`: Date, default value: current timestamp.

• **Expected Outputs:**

* Data Structure: MongoDB documents (JSON-like).

* Format: BSON (Binary JSON).

* Destination: MongoDB database.

* Operations:

* Create: New user accounts.

* Read: User authentication, profile retrieval.

* Update: User profile updates.

* Delete: (Potentially) User account deletion.

• **Data Flow and Integration Points:**

* Backend API sends requests to the MongoDB database.

* MongoDB database processes the requests and returns a response.

* Backend API receives the response and sends it back to the frontend.

• **Architecture:**

* NoSQL database.

* Document-oriented database.

• **Module/Component Responsibilities:**

* Stores user data, including credentials, profile information, and session data.

* Provides data persistence.

* Ensures data integrity.

• **Design Patterns:**

* N/A

• **Communication Protocols:**

* MongoDB wire protocol.

• **Data Specifications:**

• **Database Schema:**

* Database Name: `login_app`

* Collection Name: `users`

* Fields:

* `_id`: ObjectId (primary key).

* `username`: String, required, unique, minLength: 3, maxLength: 20.

* `email`: String, required, unique, valid email format (regex: `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`).

* `password`: String, required (hashed using bcrypt), stores the hash.

* `firstName`: String, optional, maxLength: 50.

* `lastName`: String, optional, maxLength: 50.

* `createdAt`: Date, default: `Date.now()`.

* `updatedAt`: Date, default: `Date.now()`.

• **Data Structures:**

* MongoDB documents (BSON).

• **Validation Rules:**

* `username`: Required, unique, alphanumeric characters only, minLength: 3, maxLength: 20 (applied at the database level with a unique index).

* `email`: Required, unique, valid email format (applied at the database level with a unique index and regex validation in the backend).

* `password`: Required (validation of complexity handled in the backend before hashing).

• **Security Specifications:**

* Authentication: MongoDB authentication mechanisms (username/password, x.509).

* Authorization: Role-based access control (RBAC).

* Encryption: Encryption at rest and in transit.

* Network security: Restricting access to the database server.

• **Runtime Environment and Deployment Details:**

* Runtime Environment: MongoDB server.

* Deployment: MongoDB Atlas (cloud-based) or self-managed MongoDB instance on EC2.

* Replication: Replica sets for high availability.

* Sharding: For horizontal scalability (if needed).

* Connection Pooling: Implemented in the backend to efficiently manage database connections.

Title 4: Load Balancer (Google Cloud Load Balancer)

• **Expected Inputs:**

* Source: Incoming traffic from users (HTTPS requests).

* Data Types: HTTP requests.

• **Expected Outputs:**

* Destination: Backend instances (EC2 instances, Lambda functions, or ECS containers) running the Python backend.

* Data Types: HTTP requests routed to healthy backend instances.

• **Data Flow and Integration Points:**

* All incoming traffic from users is directed to the Google Cloud Load Balancer.

* The load balancer distributes the traffic across multiple instances of the backend application.

• **Architecture:**

* Layer 7 (HTTP/HTTPS) load balancing.

* Global load balancing.

• **Module/Component Responsibilities:**

* Distributes traffic evenly across backend servers.

* Health checking backend instances.

* Routing traffic to healthy instances.

* Providing SSL termination.

• **Design Patterns:**

* Load Balancing.

• **Communication Protocols:**

* HTTPS.

• **Data Specifications:**

* N/A (Load Balancer primarily deals with network traffic, not structured data).

• **Security Specifications:**

* SSL termination.

* DDoS protection.

* Integration with Google Cloud Armor for web application firewall (WAF) capabilities.

• **Runtime Environment and Deployment Details:**

* Runtime Environment: Google Cloud Load Balancing service.

* Configuration: Configured using Google Cloud Console or command-line tools (gcloud).

* Health Checks: Configured to monitor the health of backend instances (e.g., HTTP health checks).

Title 5: AWS Cloud Services

• **Expected Inputs:**

* Source: Infrastructure as Code (IaC) tools (e.g., Terraform, CloudFormation), application deployment scripts, monitoring data.

* Data Types: Configuration files, deployment packages, log data, metrics.

• **Expected Outputs:**

* Destination: Deployed application, running infrastructure, monitoring dashboards, alerts.

* Data Types: Running virtual machines (EC2), containerized applications (ECS/EKS), static files (S3), user identities (Cognito/IAM), logs and metrics (CloudWatch).

• **Data Flow and Integration Points:**

* IaC tools provision AWS resources.

* Application deployment scripts deploy the application to AWS resources.

* CloudWatch collects logs and metrics from the application and AWS infrastructure.

• **Architecture:**

* Cloud-based infrastructure.

* Microservices architecture (optional, if using ECS/EKS).

• **Module/Component Responsibilities:**

• **EC2:** Provides virtual servers for running the backend application.

• **ECS/EKS:** Provides container orchestration for managing Docker containers.

• **S3:** Stores static assets (e.g., images, CSS, JavaScript files).

• **IAM:** Controls access to AWS resources.

• **CloudWatch:** Provides monitoring and logging.

• **Design Patterns:**

* Cloud-Native Architecture.

* Infrastructure as Code.

• **Communication Protocols:**

* HTTPS (for accessing AWS services).

* AWS API.

• **Data Specifications:**

* N/A (AWS Cloud Services manage infrastructure and resources, not application-specific data).

• **Security Specifications:**

* IAM roles and policies to control access to AWS resources.

* Security groups to restrict network access to AWS resources.

* Encryption of data at rest and in transit.

* Compliance with industry standards (e.g., PCI DSS, HIPAA).

• **Runtime Environment and Deployment Details:**

* Runtime Environment: AWS cloud environment.

* Deployment: Using IaC tools (Terraform, CloudFormation) and deployment pipelines.

* Monitoring: Using CloudWatch metrics, alarms, and dashboards.

Title 6: Docker

• **Expected Inputs:**

* Source: Application code, dependencies, configuration files.

* Data Types: Source code, package manifests, configuration files.

• **Expected Outputs:**

* Destination: Docker images.

* Data Types: Container images containing the application and its dependencies.

• **Data Flow and Integration Points:**

* Docker builds images from application code and dependencies.

* Docker containers are deployed to AWS ECS/EKS or EC2.

• **Architecture:**

* Containerization.

• **Module/Component Responsibilities:**

* Creating consistent and reproducible environments.

* Simplifying deployment and scaling.

* Isolating applications from each other.

• **Design Patterns:**

* Containerization.

• **Communication Protocols:**

* N/A (Docker primarily deals with packaging and running applications, not communication protocols).

• **Data Specifications:**

* N/A (Docker primarily deals with packaging applications, not application-specific data).

• **Security Specifications:**

* Image scanning for vulnerabilities.

* Limiting container privileges.

* Using a secure base image.

• **Runtime Environment and Deployment Details:**

* Runtime Environment: Docker engine.

* Deployment: AWS ECS/EKS or EC2.

* Orchestration: Docker Compose, Kubernetes (if using EKS).

```
```