Team: Elite

Maithilee Nargide

24203087

Shubham Limkar

24202802

Synopsis:

The **Distributed Event Management System** is a highly scalable, distributed platform designed to facilitate the seamless organization and participation in events. The system caters to both event organizers and attendees, providing a comprehensive solution for managing events, handling ticket sales, and engaging users through notifications and updates. By leveraging modern cloud technologies, the system ensures high availability, efficiency, and flexibility to meet the dynamic demands of event management.

The Application Domain

The application domain of the Distributed Event Management System lies in Event Management and Ticketing Services. This domain focuses on providing services for the creation, management, and participation in events, ranging from conferences, workshops, and seminars to concerts, exhibitions, and meetups.

Key aspects of this domain include:

- Efficient management of events. Their details, schedules, and participant registration.
- Hassle-free ticket booking and validation processes.
- Enhanced user engagement through notifications, reminders, and updates in real time.

The system caters to event organizers who require robust tools for managing events and participants and attendees seeking a user-friendly platform to discover, register for, and receive updates about events.

What will the application do?

What Will the Application Do?

The Distributed Event Management System will offer various key functionalities:

- The Event Registration enables event organisers to create, update and manage events. Allow participants t view available events and also register. It stores event details and participant registrations.
- Ticketing facilitates ticket purchase and reservation for events. It tracks ticket availability and provide organizers with insights into ticket sales and attendance.
- User engagement send notifications to users regarding upcoming events. The communication between the backend and frontend is enabled using REST APIs.
- The front-end is a responsive user interface built with React.js to ensure an intuitive user experience. It displays event details, allow ticket purchase, and show notifications.
- The backend services are a set of modular backend services built using flask to handle business logic and API requests. Containerizes backend services are orchestrated using Kubernetes for scalability and fault tolerance. By integrating these functionalities, the application ensures reliable event management.

Technology Stack

1. **Flask:** Flask serves as the backend framework for building RESTful APIs. It provides the foundation for handling HTTP requests, managing real-time communication, and implementing business logic.
   Flask is:
   - Lightweight and Simple: Flask's minimalistic design makes it easy to develop APIs quickly and flexibly.
   - Extensibility: Flask's extensive library of plugins allows integration with authentication systems, databases, and middleware.
   - Rich Ecosystem: It provides support for extensions like Flask-Cors, and Flask-RESTful, making it ideal for our needs.
2. **Amazon DynamoDB**: Amazon DynamoDB is the NoSQL database used to store event data, ticket information, and notifications.
   Amazon DynamoDB is:
   - Scalability: DynamoDB automatically scales to handle large amounts of data and traffic without manual intervention.
   - Low Latency: Its fast read/write operations ensure the system remains highly responsive.
   - Managed Service: As a fully managed database, DynamoDB eliminates the need for database administration, reducing operational overhead.
3. **Docker**: Docker is used to containerize backend services, ensuring consistent execution across different environments (development, testing, and production).
   Docker is:
   - Portable: Docker ensures the application behaves identically regardless of where it is deployed, eliminating environment-specific issues.
   - Isolation: Each service runs in its container, preventing dependency conflicts and enabling independent updates.
   - Ease of Deployment: Containerized applications can be deployed quickly, streamlining the CI/CD process.
4. **Amazon Elastic Kubernetes Service (EKS):** EKS orchestrates the containerized backend services, ensuring scalability, fault tolerance, and high availability.
   Amazon EKS:
   - It simplifies Kubernetes management, reducing the need for manual configuration and maintenance.
   - EKS automatically scales pods and nodes based on workload, ensuring optimal resource utilization.
   - EKS integrates natively with other AWS services like IAM, VPC, and CloudWatch, making it easier to manage and monitor the system.
5. **Amazon Elastic Container Registry (ECR)**: Amazon ECR is the Docker image repository used to store container images for backend services.
   Amazon ECR:
   - ECR integrates directly with Amazon EKS, enabling efficient image deployment.
   - ECR provides image scanning to detect vulnerabilities, ensuring secure containerized applications.
   - As a managed service, ECR eliminates the need to maintain a separate Docker registry.
6. **React.js**: React.js is used to build the frontend interface, allowing users to interact with the system dynamically and intuitively.
   React.js:
   - Component-Based Architecture: React's modular design allows us to create reusable UI components, simplifying development and maintenance.
   - Dynamic and Interactive: React efficiently updates and renders UI components in response to user actions or data changes.
7. **Axios:** Handles HTTP requests to communicate with backend APIs for fetching and sending data.
   Axios: Provides an easy-to-use API for making asynchronous calls, with built-in support for error handling and interceptors.

8. **Python:** Core programming language used for implementing the backend services.
   We used python as it is known for its simplicity and vast ecosystem of libraries, especially for interacting with AWS and building APIs.
9. **AWS S3 and CloudFront:** AWS S3 is used to host the frontend static files (JS), while CloudFront is used to distribute these files globally with low latency.
   AWS S3 and CloudFront:
   - Scalability: S3 scales automatically to handle a large number of requests.
   - Global Distribution: CloudFront reduces latency by caching content closer to users.
   - Cost-Effective: S3 and CloudFront provide affordable storage and delivery options for static content.

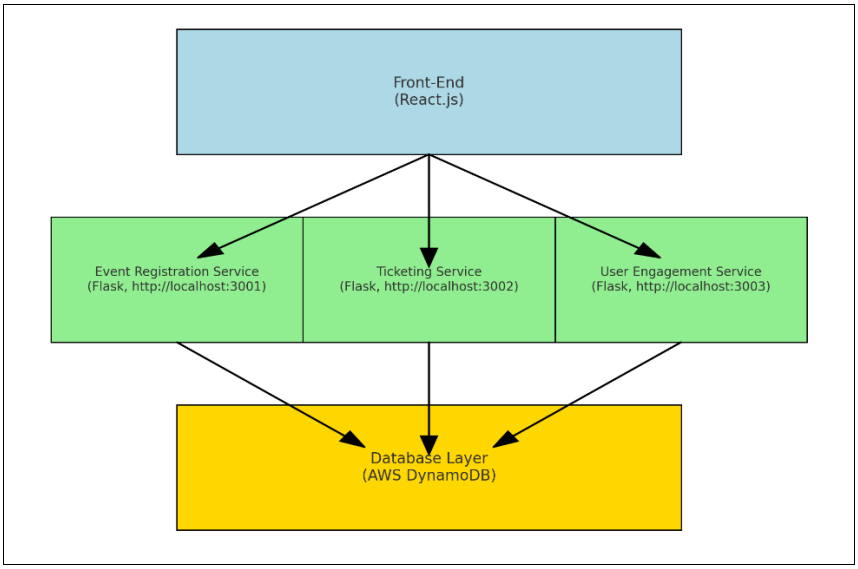
The chosen technologies are tailored to meet the requirements of the Distributed Event Management System such as scalability, performance, reliability and ease of development.

System Overview

Main components of your system:

1. **Event Registration Service**:
   - Manages event details and participant registrations.
   - Stores event details and user registrations in a DynamoDB table.
   - Exposes APIs for event-related operations (POST /events, GET /events).
2. **Ticketing Service**:
   - Handles ticket sales and validation.
   - Stores ticket-related data in DynamoDB.
   - Provides APIs for ticket-related operations (POST /tickets, GET /tickets).
3. **User Engagement Service**:
   - Sends notifications to users.
   - Leverages RESTful APIs for bidirectional communication with the frontend.
   - Stores notification logs in DynamoDB.
4. **Frontend Interface**:
   - Built using React.js to provide a dynamic and responsive user experience.
   - Interacts with backend services via REST APIs.
5. **Amazon DynamoDB**:
   - Serves as the primary NoSQL database for all backend services, storing event, ticket, and notification data.
6. **Docker**:
   - Containerizes each backend service for consistent behaviour across environments.

System Architecture Diagram



Working of system based on the diagram

1. **Frontend**:

   - Technology Used: React.js
   - Hosted locally on: http://localhost:3000
   - Users access the React-based frontend, which is hosted locally during development or can be hosted on a service like AWS EC2 for deployment.
   - The frontend interacts with backend services using REST APIs for functionalities like event registration, ticket management, and notification handling.
   - Users can:
     - View event details.
     - Purchase tickets.
     - View notifications related to their activities

2. **Backend Services**:
   The backend services are implemented using Flask and are containerized using Docker for ease of deployment. Each service has a specific domain of responsibility:

   a. Event Registration Service (http://localhost:3001)
   - Responsibilities: Handles event-related operations such as adding new events, fetching event details, and listing all events.
   - Database Integration: Communicates with the Events Table in DynamoDB to store and retrieve event data.
   - API Endpoints: Includes routes for adding events, fetching event details, and listing all events.

   b. Ticketing Service (http://localhost:3002)
   - Responsibilities: Manages ticket-related operations such as purchasing tickets, validating tickets, and listing tickets.
   - Database Integration: Communicates with the Tickets Table in DynamoDB to store ticket data and ensure availability.
   - API Endpoints: Includes routes for adding tickets, fetching ticket details, and listing all tickets.

   c. User Engagement Service (http://localhost:3003)
   - Responsibilities: Provides user notifications related to events or ticketing activities.
   - Database Integration: Communicates with the Notifications Table in DynamoDB to store and retrieve notification data.
   - API Endpoints: Includes routes for adding notifications and fetching notifications for users.

3. **Database (DynamoDB)**:

    - **Technology Used:** AWS DynamoDB
    - Each backend service has its own DynamoDB table:
      - Events Table: Stores event data, including EventID, OrganizerID, EventDetails, MaxCapacity, and RegisteredUsers.
      - Tickets Table: Maintains ticket data, including TicketID, EventID, UserID, and Status.
      - Notifications Table: Logs notification data such as UserID, Timestamp, and Message.

4. **Containerization and Orchestration**:

    - Each backend service is packaged into a Docker container for consistency across environments.
    - These containers are deployed and managed by AWS EKS, which ensures automatic scaling, rolling updates, and fault tolerance.

How our system is designed to support scalability and fault tolerance

Scalability:

1. **Backend Services**:

    - Deployed on AWS EKS, which automatically scales pods based on workload.
    - DynamoDB scales read/write capacity automatically to handle high traffic.

2. **Frontend**:

    - Hosted on AWS S3, which can handle virtually unlimited concurrent requests.
    - CloudFront provides global caching for faster content delivery.

Fault Tolerance:

1. **EKS**:

    - Ensures high availability by distributing pods across multiple nodes in a cluster.
    - Automatically restarts failed pods and replaces unhealthy nodes.

2. **DynamoDB**:

    - Provides multi-AZ (Availability Zone) replication for high availability.
    - Ensures data durability with automatic backups and replication.

3. **Docker**:

    - Containers isolate services, preventing issues in one service from affecting others.
    - Enables rapid recovery by redeploying containers from stored images.

The Distributed Event Management System is designed with modularity, scalability, and fault tolerance as its core principles. By leveraging modern cloud technologies like EKS, DynamoDB, and S3, the system ensures high availability, low latency, and a seamless user experience.

Contributions

Member 1: Shubham Limkar | 24202802

Tasks:

1. **Setting Up the React Application**:
   - Initialize the React project using create-react-app.
   - Install necessary libraries like axios, react-router-dom.
2. **Develop Frontend Components**:
   - Implement EventList.js to display available events fetched from the Event Registration Service.
   - Create TicketPurchase.js to allow users to purchase tickets via the Ticketing Service API.
   - Build Notifications.js to handle real-time notifications using polling.
3. **Integrate APIs**:
   - Use Axios to connect the frontend components to backend APIs (/events, /tickets, /notify).
   - Ensure proper error handling and loading indicators.
4. **Build and Deploy Frontend**:
   - Create a production build of the React application using npm run build.
   - Deploy the frontend to **AWS S3** and configure **CloudFront** for global distribution.

Member 2: Maithilee Nargide | 24203087

Tasks:

1. **Set Up Backend Environment**:
   - Create and configure three backend services (Event Registration, Ticketing, and User Engagement) using flask.
2. **Develop REST APIs**:

   Implement APIs for:
   - Event Registration: POST /events, GET /events.
   - Ticketing: POST /tickets, GET /tickets.
   - User Engagement: POST /notify, GET /notifications.
3. **Database Integration**:

   Configure Amazon DynamoDB tables for each service:
   - Events Table: Store event details and user registrations.
   - Tickets Table: Track ticket purchases and availability.
   - Notifications Table: Store user notifications and timestamps.
4. **Containerization**:
   - Write Dockerfiles for each backend service to containerize them.
   - Test containers locally to ensure they function correctly on Postman.
5. **Deploy Backend Services**:
   - Push Docker images to **Amazon ECR**.
   - Deploy services to **AWS EKS** using Kubernetes manifests.

Shared Responsibilities

Both members collaborate on System Architecture and Design, System Testing and Debugging, Documentation and Report.

Reflections

Key challenges faced:

1. **Integration of Backend and Frontend**:

   Challenge: Ensuring seamless communication between the React.js frontend and Flask backend via REST APIs. Handling cross-origin issues by implementing CORS in the backend services.

   - **Solution**: Regular testing using tools like Postman for APIs and debugging helped identify and resolve integration issues early.
2. **Data Flow Management:**
   - Ensuring proper mapping between frontend actions (e.g., ticket purchases, event viewing) and backend data processing.
   - Debugging API calls to ensure accurate data is fetched and displayed.
3. **DynamoDB Configuration:**
   - Properly setting up and testing the DynamoDB tables (Events, Tickets, Notifications) for efficient storage and retrieval.
   - Handling errors like "Unable to locate credentials" during backend integration with AWS DynamoDB.
4. **Deployment Challenges:**
   - Deploying the backend services locally and globally using AWS EC2 while ensuring scalability and accessibility.
   - Managing React app deployment and ensuring all endpoints point to the correct backend service URLs.

What would we have done differently if we could start again?

1. **Early Planning of Architecture**: If starting again, more time would have been spent planning the system architecture and defining clear data flows. This would have minimized rework during the integration phase.
2. **Adopting WebSockets Early:** Integrating WebSockets at the beginning for real-time communication would provide a more interactive user experience, especially for notifications and event updates. Libraries like Flask-SocketIO (for Python) could have been used for implementing WebSocket-based communication.
3. **Exploration of Alternative Technologies**: Evaluating other backend frameworks (e.g., FastAPI for Python) or database solutions (e.g., PostgreSQL) might have provided additional insights or benefits. Using WebSockets for real-time communication.
4. Conducting comprehensive load testing earlier in the project would have revealed scalability issues sooner.

What we learnt about the technologies used?

1. **Flask:**
   - Flask's lightweight and modular architecture made it easy to build RESTful APIs for event registration, ticketing, and user engagement.
   - Integration with AWS DynamoDB using Boto3 was straightforward and allowed us to manage data efficiently.
   - Flask's flexibility allowed for easy implementation of additional features like CORS and error handling.

2. **React.js**:
    - React is highly efficient for building interactive and responsive user interfaces with reusable components.
    - It simplifies managing application state using hooks like useState and useEffect, which helped us fetch data from the backend and display it dynamically.
    - Routing with libraries like react-router-dom makes navigation between pages intuitive and manageable.
3. **Amazon DynamoDB**:
    - DynamoDB provided a fully managed, serverless NoSQL database that scaled automatically with traffic.
    - DynamoDB's partition key and sort key model made data retrieval highly efficient for our use cases (e.g., querying tickets by EventID)
4. **Docker**:
    - Docker streamlined the development and deployment process by packaging each backend service with its dependencies, ensuring consistency across environments.
    - Docker Compose simplified managing multiple services, enabling us to start all services together and define their networking.

1. **Amazon EKS**:

    - Benefits: Automates the orchestration and scaling of containerized applications; integrates seamlessly with other AWS services.
    - Limitations: Steep learning curve; managing Kubernetes configurations is complex.

Limitations of the System

1. **Eventual Consistency**: DynamoDB's eventual consistency model might lead to temporary delays in data visibility during high traffic.
2. **Real-Time Communication**: The system fails in real time communication which can be implemented if used WebSockets.
3. **Dependency on AWS Services**: The system heavily relies on AWS, which may limit portability to other cloud providers.

Benefits of the System

1. **Scalability**: Designed to handle increased traffic and users dynamically with Kubernetes and DynamoDB.
2. **Cost Efficiency**: The use of serverless and managed services (e.g., DynamoDB, EKS) reduces operational overhead.
3. **Reliability**: Built-in fault tolerance and high availability across all AWS services ensure minimal downtime.