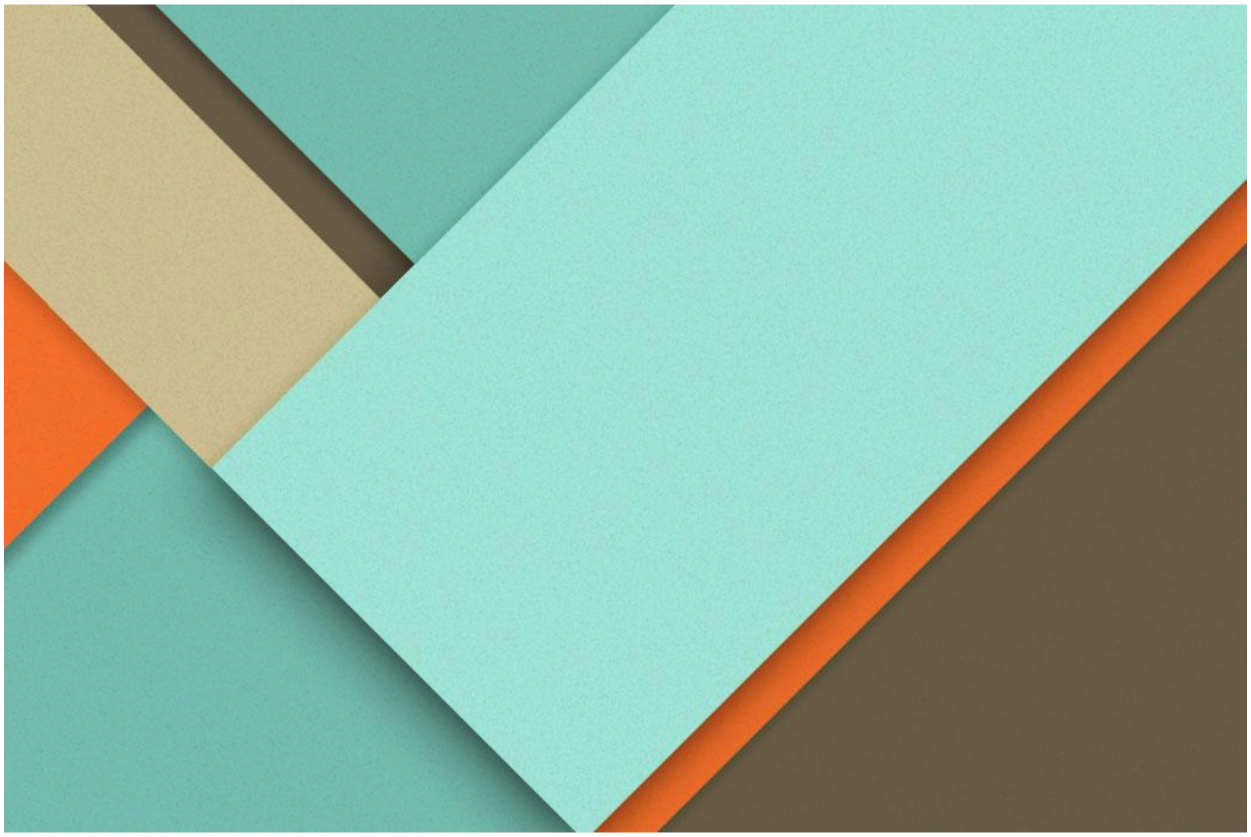


Quiz Booking System

04.07.2024

—

Karthikeyan M
Technical Architect

INDEX

All the services are communicated from frontend through http with enforced headers like HTHS and with JWT token Authentication headers, Backend services will validate the JWT token and send back the response.

Deploy our services across two separate **Private Kubernetes clusters and two regions:**

- The Payment Service will be deployed in a dedicated private cluster. This isolation safeguards sensitive payment data and strengthens overall security.
- **Enhanced Security:** By isolating the Payment Service in a separate,cluster, we minimize the attack surface and potential security risks associated with payment processing.
- **Improved Maintainability:** Separating services simplifies deployment, maintenance, and scaling for each cluster.

The separation of payment processing strengthens security, while independent cluster management simplifies development and maintenance.

Service Routing will be handled in the **Ingress Controller at cluster level**

So two load balancers which will point To 2 k8's cluster with in a region. Same in another region with AWS Global Accelerator acts as a global entry point, routing traffic to the most appropriate ALB across two regions. AWS WAF Rules added on ALB to prevent vulnerable attacks

Improved Scalability: Each load balancer can independently handle the traffic specific to its cluster, ensuring better overall scalability.

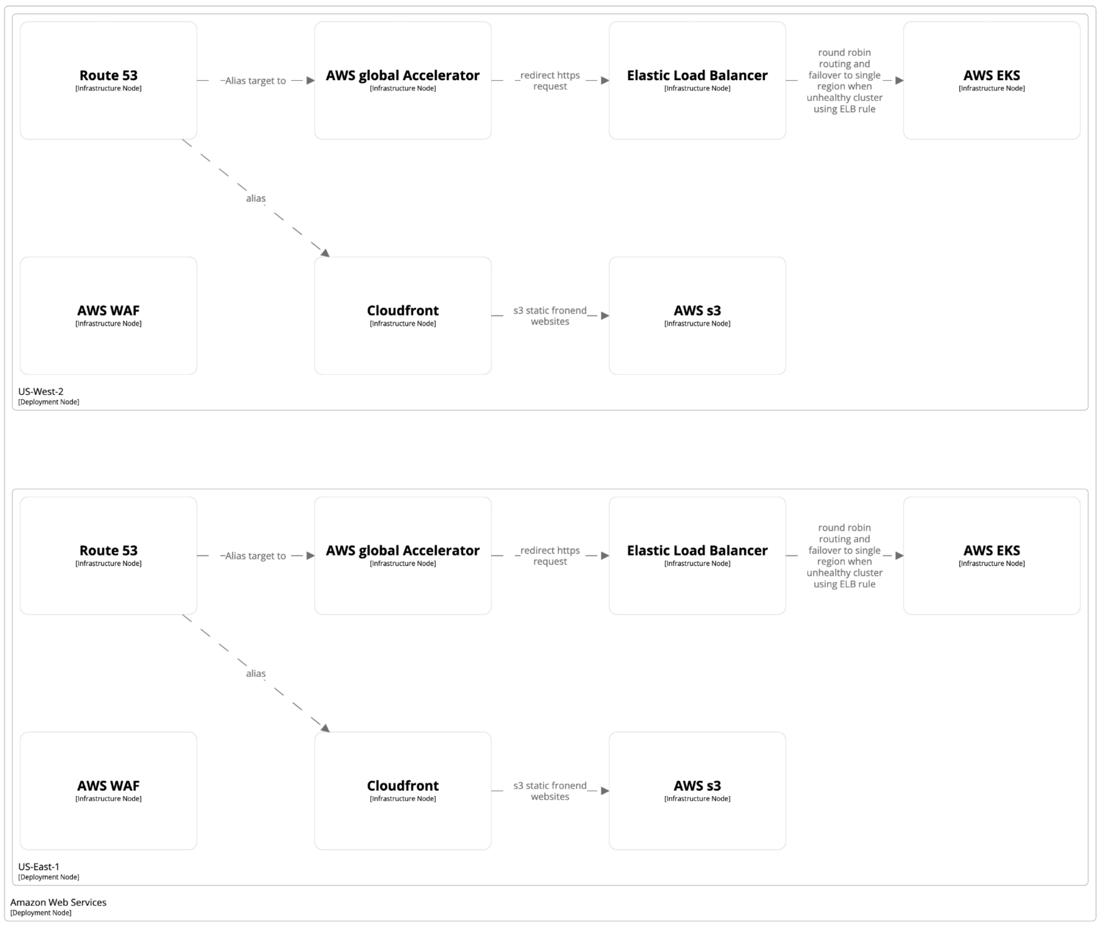
Enhanced Availability: If one cluster or its load balancer experiences an issue, other clusters and their load balancers can continue functioning. This improves fault tolerance.

Simplified Management: Managing a load balancer per cluster simplifies configuration and troubleshooting compared to a single centralized load balancer.

This deployment utilizes 4 Kubernetes clusters strategically placed across different regions. Traffic is load-balanced based on two key factors:

1. **Low Latency:** Users are directed to the cluster that offers the lowest latency, ensuring a faster and more responsive experience.
2. **Cluster Health:** If a cluster becomes unhealthy, the load balancers automatically failover to the healthy cluster, maintaining application availability.

Only the frontend is exposed in public, All the backend resides in EKS private cluster.



User Registration and Authentication

Overview

AWS Cognito for user authentication and stores user credentials securely. Upon user registration, relevant user data is persisted in AWS Cognito.

User Registration

1. **User Registration:** The user registers through the application interface hosted in s3.
2. **AWS Cognito Registration:** User data (email and a unique identifier) is published to a topic in Apache Pulsar/Kafka upon registration.

3. **User Microservice:** A consumer like the User service subscribes to the topic and retrieves the published data.
4. **Database Registration: The User service registers the user in the MySQL database** only with email and unique identifier in the user table. This establishes a connection between user identity and actions like course booking and quiz participation.
5. **AWS SES Email:** AWS Cognito Inbuilt sends an invitation email via SES (Simple Email Service) prompting the user to set up a password.
6. **Password Setup:** The user sets a password through the AWS Cognito interface. The password is stored securely within AWS Cognito and not in the MySQL database.

User Login

1. **Login Form:** The user logs in through a login form integrated with the frontend using AWS Amplify.
2. **Authentication and JWT Token:** AWS Amplify handles user authentication and provides a JSON Web Token (JWT) upon successful login.
3. **JWT Token Storage:** The JWT token is stored in cookies and used for authorization across subdomains and services.
4. **Token Validation:** Each service validates the JWT token using a dedicated AWS API with a shared secret key before granting access to resources.

Additional User Profile Data

Additional user profile details like name, phone number, and country are stored as attributes within AWS Cognito. Since AWS adheres to PII (Personally Identifiable Information) compliance standards, there's no need for additional data masking and storage within the MySQL database.

User profile information

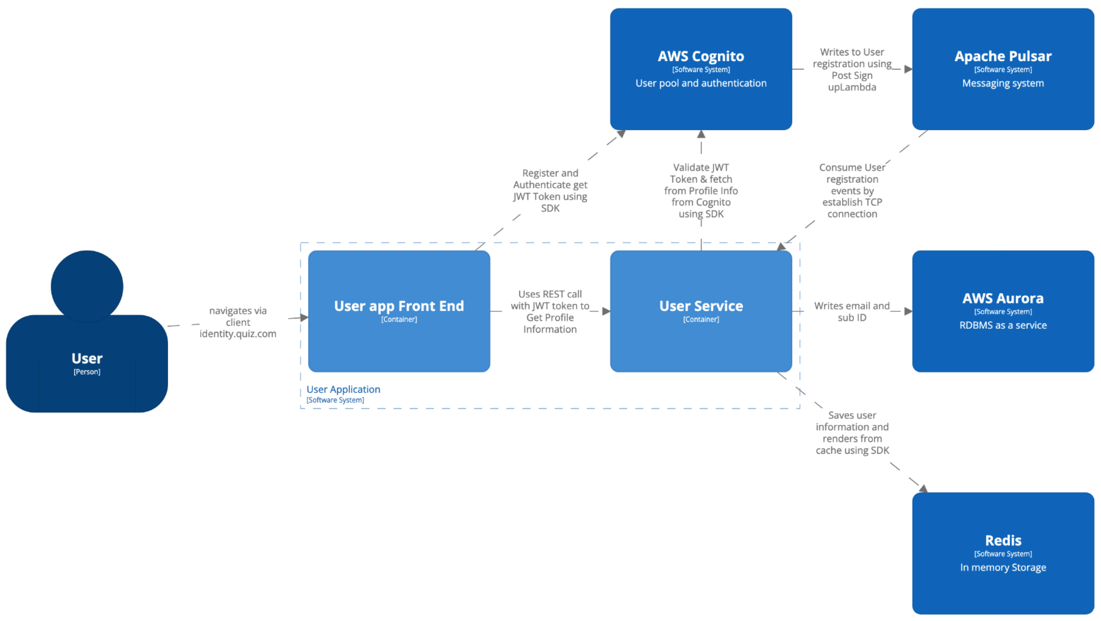
Will be saved in **Redis cache** for faster retrieval and avoid multiple requests to Cognito

Benefits of this Approach

- **Security:** AWS Cognito handles user authentication securely, eliminating the need to store passwords in our database.
- **Centralized Management:** User authentication is centralized in AWS Cognito, simplifying user management.
- **PII Compliance:** Leverage AWS's built-in PII compliance for user data storage.

- **Scalability:** AWS Cognito scales to accommodate a growing user base.

This approach offers a robust and secure user registration and authentication system for the application.


AWS Cognito
[Software System]
User pool and authentication
Writes to User registration using Post Sign upLambda
Apache Pulsar
[Software System]
Messaging system
Register and Authenticate get JWT Token using SDK
Validate JWT Token & fetch from Profile Info from Cognito using SDK
Consume User registration events by establish TCP connection
User
[Person]
navigates via client identity.quiz.com
User app Front End
[Container]
Uses REST call with JWT token to Get Profile Information
User Service
[Container]
Writes email and sub ID
AWS Aurora
[Software System]
RDBMS as a service
User Application
[Software System]
Saves user information and renders from cache using SDK
Redis
[Software System]
In memory Storage

Course Booking Service

Focusing on key functionalities like search, creation, listing, editing, and cart management and price.

Service Components

- **Frontend:**
  - **Technology:** Amazon Simple Storage Service (S3)
  - **Description:** The user interface for interacting with the system is hosted on S3, providing a user-friendly experience for course browsing, search, cart management, and order placement.
-

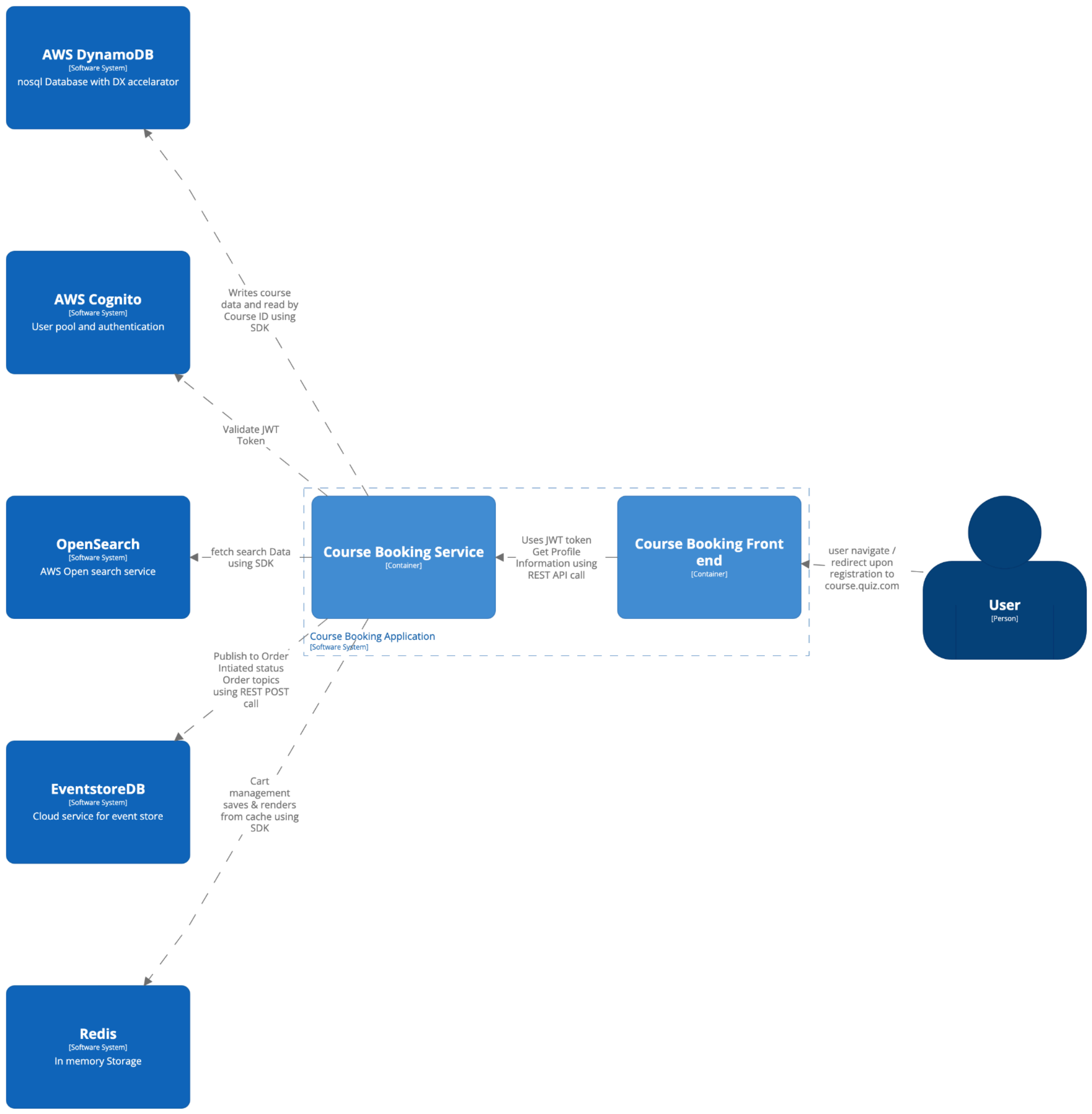
  - **Description:** A dedicated microservice responsible for handling various course-related functionalities:
    - **Course Management:** Handles creation, editing, and listing of courses within the system.
    - **Search:** Interacts with AWS OpenSearch to facilitate course search based on user queries. Retrieved course data is then returned to the front end for display.
    - **Cart Management:** Updates the Redis cache with cart data (e.g., added courses, quantities) for faster retrieval and improved user experience.
    - **Order Initiation:** Upon order placement, the course service saves order details, including user ID, course ID, and initial order status ("Order Initiated"), to the event store database.
- **Database:**
  - **DynamoDB:**
    - **Technology:** NoSQL Database (Key-Value Store)
    - **Description:** Stores course data with high availability and scalability.
      - **DX Accelerator:** Leverages edge caching to enable faster retrieval of course details by ID, improving performance for frequently accessed data.
  - **AWS OpenSearch:**
    - **Technology:** Search Engine
    - **Description:** A dedicated search engine for indexing and searching course data. This allows users to efficiently find relevant courses based on their search queries.
  - **Event Store Database:**
    - **Technology:** (Event Sourcing Solution - e.g., Apache Kafka, Kafka Streams)
    - **Description:** Stores order details with "Order Initiated" status upon successful order placement. This database serves as an append-only log of events, decoupling course booking and order processing workflows.
    - Replay History: We can Replay
- **Redis Cache:**
  - **Technology:** In-Memory Data Store
  - **Description:** Stores cart data (e.g., added courses, quantities) for faster listing and retrieval, improving performance and responsiveness for frequently accessed cart information.

Data Flow

1. **Course Creation/Update:** The course service writes the updated course data to DynamoDB for persistent storage.
2. **Indexing:** The updated course data is automatically indexed by AWS OpenSearch, enabling efficient search functionality based on user queries.
3. **Course Search:** The frontend sends search queries to the course service. The service retrieves results from AWS OpenSearch based on the search criteria and returns them to the frontend for display.
4. **Cart Management:** Cart data (e.g., added courses, quantities) is added or updated in the Redis cache for faster access, optimizing user experience for frequently viewed cart information.
5. **Order Placement:** Upon confirmation of an order, the course service saves order details with user ID, course ID, and the "Order Initiated" status to the event store database which will be projected to Orders table in **MYSQL** database for reading purpose. This triggers the order processing workflow (potentially handled by a separate service).

Benefits of this Approach

- **Scalability:** Both DynamoDB and AWS OpenSearch are highly scalable solutions, allowing the system to accommodate a growing user base with a large course catalog.
- **Performance:** Utilizing DX Accelerator in DynamoDB ensures fast retrieval of frequently accessed course details by ID. Additionally, the Redis cache improves performance for frequently accessed cart data.
- **Search Functionality:** AWS OpenSearch provides a robust and efficient search engine for course discovery, enabling users to easily find relevant courses based on their needs.
- **Event-driven Architecture:** Leveraging an event store database allows for decoupling course booking and order processing workflows, improving system flexibility and scalability.
- **Event Sourcing:** An event store is specifically designed to capture the entire order history as a sequence of events (e.g., "Order Placed," "Payment Received," "Order Shipped"). This simplifies audit trails and replaying events for debugging or data pipelines.
- **Scalability:** Event stores are generally more scalable than relational databases for high-volume data ingestion.


AWS DynamoDB
[Software System]
nosql Database with DX accelarator
AWS Cognito
[Software System]
User pool and authentication
OpenSearch
[Software System]
AWS Open search service
EventstoreDB
[Software System]
Cloud service for event store
Redis
[Software System]
In memory Storage
Writes course data and read by Course ID using SDK
Validate JWT Token
fetch search Data using SDK
Publish to Order Intiated status Order topics using REST POST call
Cart management saves & renders from cache using SDK
Course Booking Service
[Container]
Uses JWT token Get Profile Information using REST API call
Course Booking Front end
[Container]
Course Booking Application
[Software System]
user navigate / redirect upon registration to course.quiz.com
User
[Person]

Payment Service

Once an order is placed, several crucial aspects need to be considered for secure and reliable payment processing using Stripe and Event store DB & MySQL database:

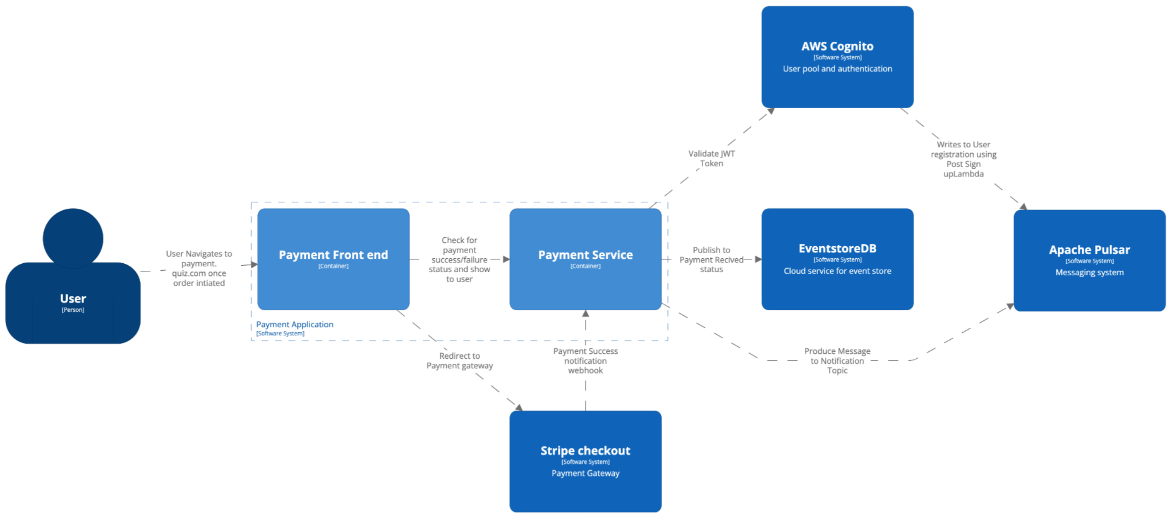
Frontend: Stripe Checkout buttons shown in order placement flow.

- **User Redirection:** Upon order placement, redirect the user to a secure Stripe payment page where they can enter their payment information.

- **Payment Tokenization:** Stripe generates a payment token without storing sensitive credit card details. We dont need to store card data in DB only token, Stripe is PCI complaint.

Order Management:

- **Order Status Update:** Upon successful payment authorization from Stripe to the backend service using webhook notification, update the order status in the Event store database to **"Payment Received"** The endpoint which is configured in stripe will validate the notification's authenticity using Stripe's signing secret in backend service..
- Also payment service produce a Message in Apache pulsar to send out email by notification service to the Customer that the course is booked.
- In case of payment failure place to update the order status to Payment Failed in Event store DB and publish a message in Apache Pulsar to send out email to communicate payment failure reasons to the user and provide options to retry or cancel the order.


AWS Cognito
[Software System]
User pool and authentication
Validate JWT Token
Writes to User registration using Post Sign upLambda
User
[Person]
User Navigates to payment. quiz.com once order intiated
Payment Front end
[Container]
Check for payment success/failure status and show to user
Payment Service
[Container]
Publish to Payment Recived status
EventstoreDB
[Software System]
Cloud service for event store
Apache Pulsar
[Software System]
Messaging system
Payment Application
[Software System]
Redirect to Payment gateway
Payment Success notification webhook
Produce Message to Notification Topic
Stripe checkout
[Software System]
Payment Gateway

Quiz Management Service

Components

- **Frontend:** ReactJS
  - **Description:** The user interface where users take quizzes, view results,
  - **Quiz Service:**

        - **Quiz Management:** Manages quiz creation, editing, Listing, and question bank management.
        - **Grading and Evaluation:** Scores quizzes based on user responses and defined rules.
        - **Result Processing:** Processes quiz results and publishes them to the event store database.
- **Database:**
    - **Question Bank:**

      DynamoDB (NoSQL) with DX accelerator

      Stores quiz questions, answer choices, and scoring logic

      Quiz Metadata:

      Stores quiz metadata (title, description, duration, etc.) in DynamoDB for efficient retrieval.

- **Messaging System:**
    - A high-throughput, low-latency messaging system for communication between services.
    - Users' quiz responses are published as messages to Apache Pulsar topics.
    - The Quiz Service subscribes to these topics, receives responses, and performs grading/evaluation.

- **Event Store Database:**
    - Stores processed quiz results as an append-only log of events. This allows for:
        - **Real-time Result Publishing:** A separate Quiz Result service subscribes to the event store and publishes results immediate display to users.
        - **Result History:** Provides a historical record of quiz results for later analysis or retrieval.

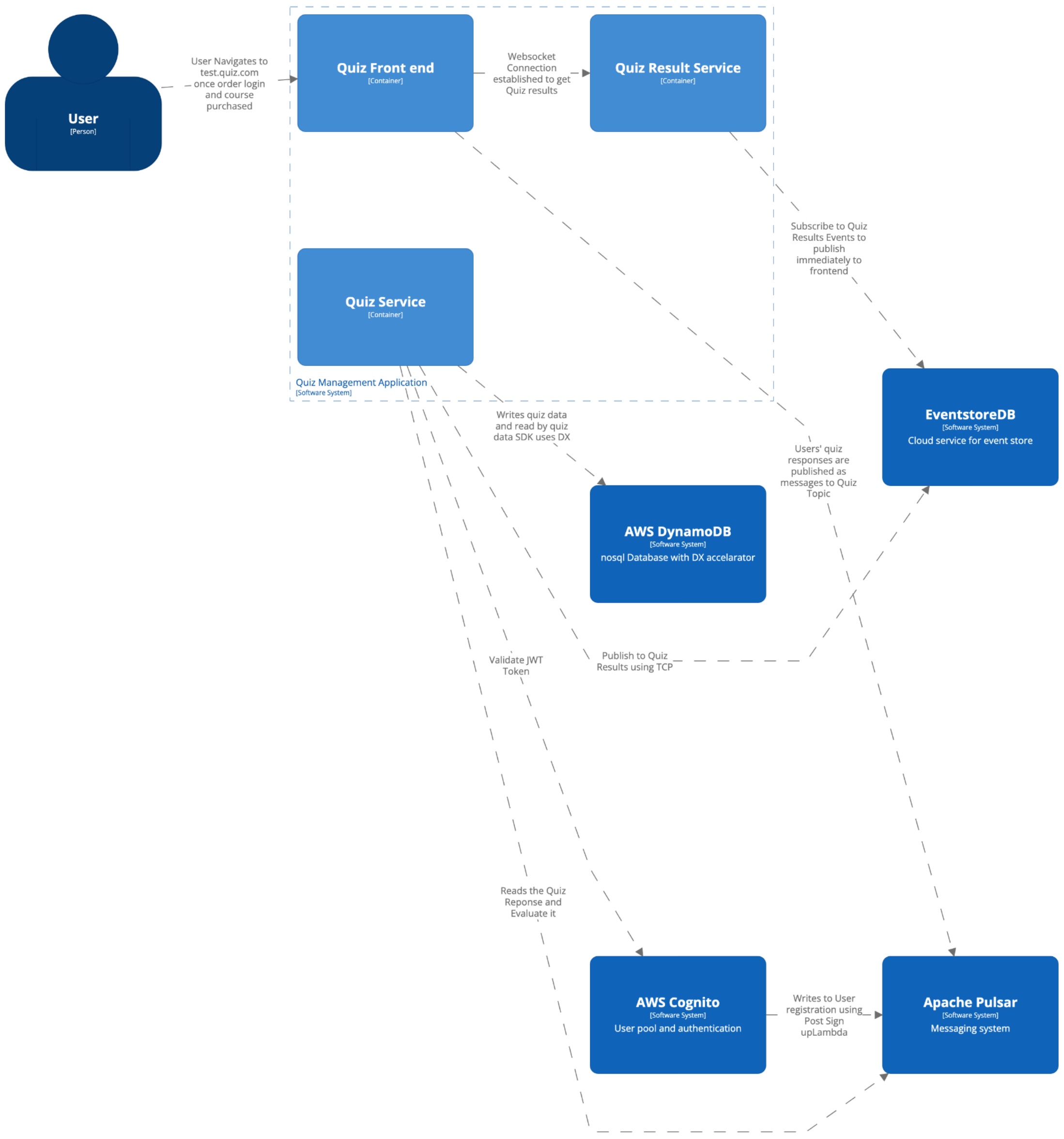
Data Flow

1. **User Takes Quiz:** The user interacts with the quiz through the frontend application.
2. **Quiz Delivery:** The frontend communicates with the Quiz Service to receive the quiz content.
3. **User Submits Responses:** Upon completion, the user submits their quiz responses through the frontend.
4. **Response Publication:** The user's responses are published as messages to relevant Apache Pulsar topics by the frontend.

5. **Quiz Grading:** The Quiz Service, subscribed to the Pulsar topics, receives user responses and performs grading/evaluation.
6. **Result Processing:** The graded quiz results are processed and published as events to the event store database.
7. **Result History:** The event store database persists quiz results for historical reference and potential retrieval.

Benefits

- **Scalability:** Leverages scalable services (DynamoDB, Apache Pulsar) to handle a high volume of users and quizzes.
- **Real-time Results:** Provides immediate feedback to users upon quiz completion through real-time messaging (optional).
- **Decoupled Architecture:** Services communicate through asynchronous messaging (Apache Pulsar), improving fault tolerance and scalability.
- **Event Sourcing:** Utilizes an event store for persistent quiz results, enabling historical analysis and potential for replaying events for debugging or data pipelines.


User
[Person]
User Navigates to test.quiz.com once order login and course purchased
Quiz Front end
[Container]
Websocket Connection established to get Quiz results
Quiz Result Service
[Container]
Quiz Service
[Container]
Quiz Management Application
[Software System]
Subscribe to Quiz Results Events to publish immediately to frontend
EventstoreDB
[Software System]
Cloud service for event store
Writes quiz data and read by quiz data SDK uses DX
Users' quiz responses are published as messages to Quiz Topic
AWS DynamoDB
[Software System]
nosql Database with DX accelarator
Validate JWT Token
Publish to Quiz Results using TCP
Reads the Quiz Reponse and Evaluate it
AWS Cognito
[Software System]
User pool and authentication
Writes to User registration using Post Sign upLambda
Apache Pulsar
[Software System]
Messaging system

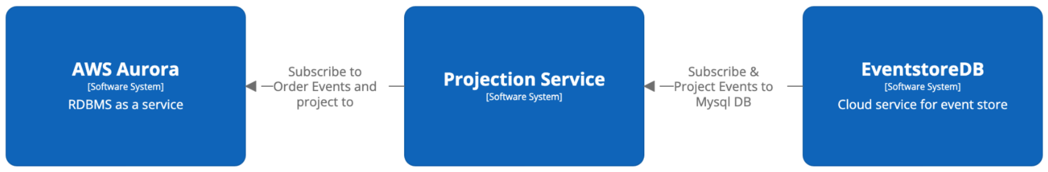
Quiz Result Service

- **Technology:** WebSockets,
- **Description:** Enables real-time delivery of quiz results to the user interface, providing immediate feedback upon quiz completion.

Once result is published to the event store DB this service will subscribe to the events using a TCP connection and it updates results immediately to the Quiz frontend through WebSocket

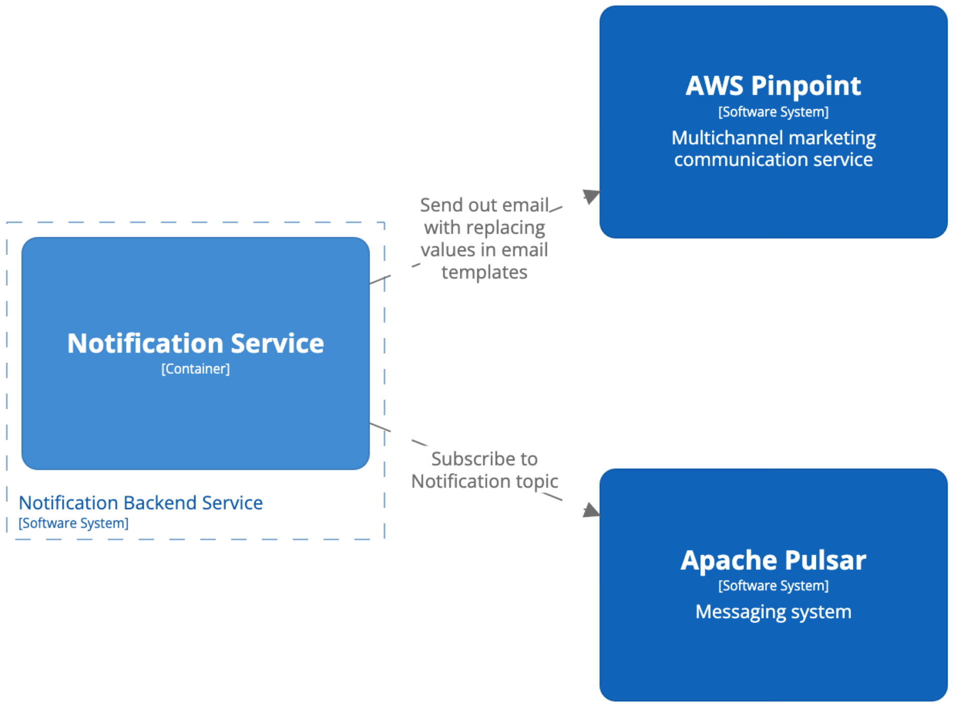
Projection Service

This service Listens to Event store DB subscribe to Order events and update it to Mysql Database.



Notification Service

This notification service listens to Apache Pulsar topics for messages related to notifications. When a message is received, it processes the message data and sends a notification via AWS Pinpoint. Pinpoint's templating engine lets us manage email templates and use dynamic content placeholders. Pinpoint provides detailed analytics on email performance, including opens, clicks, conversions, and unsubscribes.


AWS Pinpoint
[Software System]
Multichannel marketing communication service
Send out email with replacing values in email templates
Notification Service
[Container]
Notification Backend Service
[Software System]
Subscribe to Notification topic
Apache Pulsar
[Software System]
Messaging system

Technologies used and security measures,

- Frontend: React - This is a popular JavaScript library for building user interfaces.
- Content Delivery Network (CDN): Cloudfront - This is a service offered by AWS that caches and delivers content (like images) to users with high performance and low latency.
- Storage: S3 - This is a scalable object storage service offered by AWS for storing various types of data.
- Security:
    - Pre-signed URLs: These are URLs that grant temporary access to specific S3 objects, enhancing security.

    - OWASP validation: This refers to following guidelines from the Open Web Application Security Project (OWASP) for validating user input and preventing security vulnerabilities.
    - AWS WAF: This is a web application firewall offered by AWS that helps protect against common web attacks.
    - Rate limiting: This restricts the number of requests a user can make in a given time frame to prevent denial-of-service (DoS) attacks.
    - Dependabot: This is a tool that automates dependency updates and helps keep software up-to-date and secure.
    - GitGuardian: This is a security tool that scans code repositories for vulnerabilities and creates issues and pull requests to fix them.
- Monitoring:
    - Datadog: This is a monitoring and analytics platform that provides insights into application performance, infrastructure health, and user experience.
    - APM agent: This is an agent that collects application performance metrics for Datadog.
    - RUM (Real User Monitoring): This monitors user experience (performance) from the end-user's perspective.
    - Synthetic monitoring: This simulates user actions to proactively monitor application performance and functionality.
    - SLO monitors and notifications: This sets up monitoring based on Service Level Objectives (SLOs) and sends notifications when breaches occur.

Security Measures

The system incorporates several security measures to protect against potential threats:

- Secure storage of static content (images) using S3 with pre-signed URLs.
- Validation of user input to prevent malicious code injection attacks.
- AWS WAF to filter out common web attacks.
- Rate limiting to prevent DoS attacks.
- Automated dependency updates and vulnerability scanning using Dependabot and GitGuardian.
- Comprehensive monitoring with Datadog for application performance, infrastructure health, and user experience.

Event store DB cloud

EventStoreDB Cloud is a managed cloud service offered by EventStore that allows to easily provision, manage, and run highly available and secure EventStoreDB clusters in various cloud environments.

.

MYSQL Aurora Global Database:

- This offering allows to create a single, logically consistent database that spans across multiple AWS regions.
- A primary cluster resides primary region, housing the writable master instance.
- Up to five read-only secondary clusters can be deployed in different regions, replicating data from the primary cluster for disaster recovery purposes.
- Aurora utilizes MySQL's binary log replication for efficient data synchronization.
- Changes made to the primary database (inserts, updates, deletes) are continuously logged in the binary log file.
- The secondary clusters in other regions continuously read and apply these changes to their own database copies, ensuring they remain up-to-date with the primary.

Failure Recovery:

- In case of a regional outage impacting the primary cluster, can initiate a failover to one of the healthy secondary clusters in another region.
- The chosen secondary cluster is promoted to become the new primary, allowing the application to continue reading and writing data with minimal downtime.

Streamnative

StreamNative is a company that provides a cloud-native event streaming platform built on top of the Apache Pulsar project. Here's a breakdown of what StreamNative offers:

StreamNative Platform:

- **Cloud-Native Design:** StreamNative's platform is architected for cloud environments, offering features like elasticity, scalability, and integration with various cloud-native technologies.

- **Apache Pulsar Compatibility:** The platform is built on and extends Apache Pulsar, a popular open-source event streaming platform. This ensures compatibility with existing Pulsar tooling and ecosystems.

WAF, S3, Cloudfront, Elastic Cache, EKS, ALB, Cognito, DynamoDB, Datadog

USED Architectural patterns

BFF in Conjunction with Event Sourcing Pattern

BFF

The BFF pattern introduces a dedicated service for each frontend client (e.g., web app, mobile app). This service acts as a facade for the backend, offering the following benefits:

- Simplified Frontend Development: The frontend only interacts with a single, well-defined API provided by the BFF. This reduces complexity and allows developers to focus on the user interface.
- Tailored Data Presentation: The BFF can aggregate data from multiple backend services and present it in a format optimized for the specific client. This eliminates data duplication and improves the user experience.
- Enhanced Security: The BFF can act as a gatekeeper, filtering data and exposing only what's necessary for the client. Sensitive data from backend services can be hidden from the frontend.

Microservice BFF Architecture:

Here's how the BFF pattern integrates with a microservice architecture:

1. Frontend Application: This is your user interface, like a React app.
2. BFF Service: This is a dedicated service for each client type (web, mobile, etc.). It interacts with various backend microservices via their APIs.
3. Backend Microservices: These are independent services that handle specific functionalities of the application.
4. API Gateway (Optional): A single entry point can be used to route requests to the appropriate BFF service based on the client type.

Benefits of BFF in Microservices:

- Improved Maintainability: Changes in the backend don't directly impact the frontend as long as the BFF API remains stable.

- Flexibility: Each BFF can evolve independently to cater to the specific needs of its client.
- Improved Client Performance: BFFs can optimize data retrieval and presentation for better user experience.

EVENT Sourcing Pattern

The Event Sourcing pattern is a data persistence approach that focuses on storing a sequence of events that represent all the changes to an application's data. It's a powerful alternative to the traditional method of storing the current state of the data.

Here's a breakdown of the core concepts:

Traditional Data Persistence:

- In the traditional approach, data is stored in a database (relational or NoSQL) as its current state.
- Whenever the data changes, the updated state is saved in the database, overwriting the previous version.
- This method can make it difficult to:
    - Track the history of changes.
    - Replay events to recreate past states.

Event Sourcing:

- Event Sourcing flips the script. Instead of storing the current state, it stores a sequence of events that represent all the changes that have happened to the data.
- Each event is a self-contained unit containing details about the change, like:
    - Timestamp of the event.
    - Type of event (e.g., "ItemAdded", "ItemUpdated").
    - Data associated with the event (e.g., new item details for "ItemAdded").
- Events are typically stored in an append-only database called an event store.

Benefits of Event Sourcing:

- Audit Trail: Provides a complete and immutable history of all changes to the data.
- Replayability: Allows you to replay events to recreate the state of the data at any point in time. This is useful for:

    - Debugging issues.
    - Implementing features like undo/redo functionality.
    - Data analysis based on historical changes.
- Scalability: Event stores are often horizontally scalable, making them suitable for large datasets and high-throughput applications.
- Event-Driven Architecture: Aligns well with event-driven architectures where events trigger actions and updates across various parts of the system.