THE COMPAMY

Software Architecture and design for Simple tool Rental Application

Prepared for Centralized Architecture Team

CONFIDENTIAL

Version: 0.1

Date: January 19th, 2024

Revision History			
Date	**Version**	**By**	**Description of Change**
Jan 19, 2024	0.1	Driss ELOUEDRHIRI	Initial document.
Feb 2, 2024	0.2	Driss ELOUEDRHIRI	Documentation Released.
Feb 2, 2024	0.3	Driss ELOUEDRHIRI	Completed Project Code. Main business logic is in Java under renting Micro-Services

Table of Contents

SCOPE OF THE DOCUMENT

This document presents the architecture and design for Simple tool Rental Application.

GLOSSARY

The following terms are used in this document:

Term	Definition
API	Application Programming Interface.

APPLICATION REQUIREMENTS

3.1 Functional Requirements

The functional requirements are described as part of the technical assessment document.

3.2 Non-functional System Requirements

The overall System should primarily support continuous scalability and performance to continuous user operations.

The management of costs is also a key factor to consider while designing the infrastructure and components setups.

It shall implement security measures to identify any potential security threats.

APPLICATION ARCHITECTURE

4.1 Architectural decision record

The architecture that provides high scalability, performance, and manageable cost, shall be used toward the system definition.

Architecture	Performance	Cost	Scalability
Monolithic	-	-	-
Microservices	O	O	O
Event Driven	O	-	O
Service Oriented	-	-	-
Serverless	-	O	O

The adapted architecture is Microservices as per the above matrix table.

4.2 System domains definition

The domain definition concludes the following:

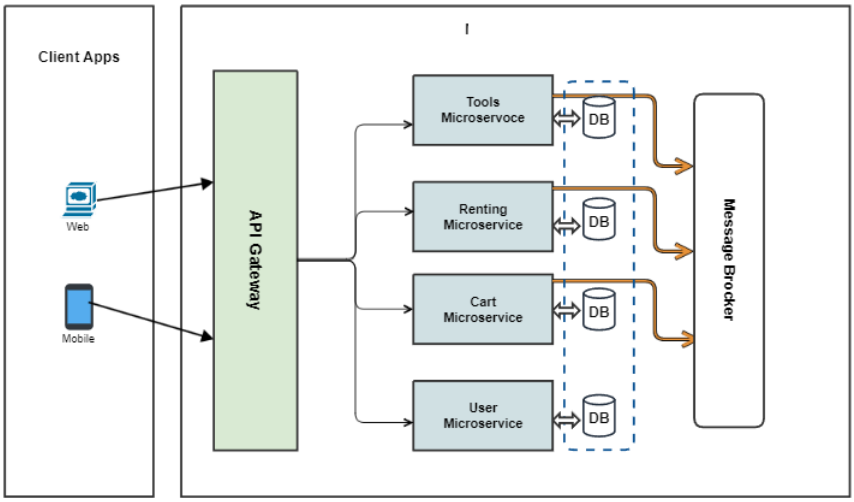
- **Tool domain**
  - Handles information about tools, including tool code, type, brand, daily charges, and availability.
  - Provides functionality for retrieving tool details, checking availability, and updating tool information.
- **Cart domain**
  - Manages customer cart and purchasing session.
- **Rental domain**

- Handles the checkout process, creating rental agreements, and calculating charges.
- Interacts with the Tool and Holiday domains to get tool and holiday information.
- Manages rental agreement data and history.

- **User domain**
  - Manages customer information, including user accounts and discounts.
  - Provides authentication and authorization services.

4.3 Software Architecture

For this system, I shall select microservices architecture as defined per the ADR section.

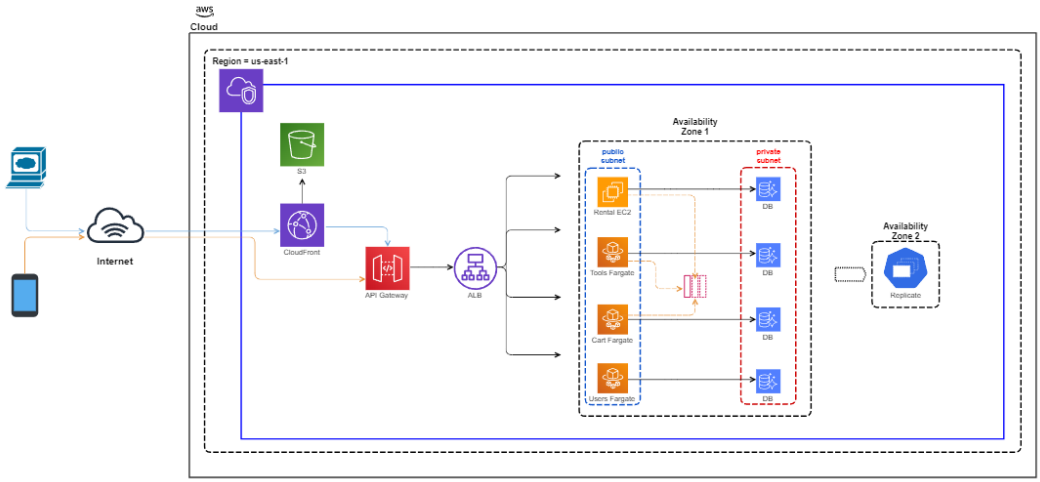


INFRASTRUCTURE

5.1 Infrastructure considerations

- The lean toward the ease of networking communication between microservices and effectively manage the cost and resource optimization, results in the consideration of shared VPC strategy.
- We shall consider a hybrid approach to run services on EC2 that requires fine-tuned control, while keeping stateless services running on Fargate.
- Containerize all microservices to ensure consistency and ease of deployment across both EC2 and Fargate. Use container orchestration tools like Amazon ECS for efficient management.
- Placing microservices in separate subnets to provide isolation and security by controlling the traffic flow between them as microservices need be more restricted in terms of access.
- Distributing microservices across multiple Availability Zones enhances fault tolerance and improves high availability. If one AZ experiences an issue, services in other AZs can continue to operate.
- Adapt a Stateless architectures typically align with the demands of dynamic workload and changing business requirements. Stateless application design can increase flexibility with horizontal scaling and dynamic deployment. This flexibility helps applications handle sudden spikes in traffic, maintain resilience to failures, and optimize cost.

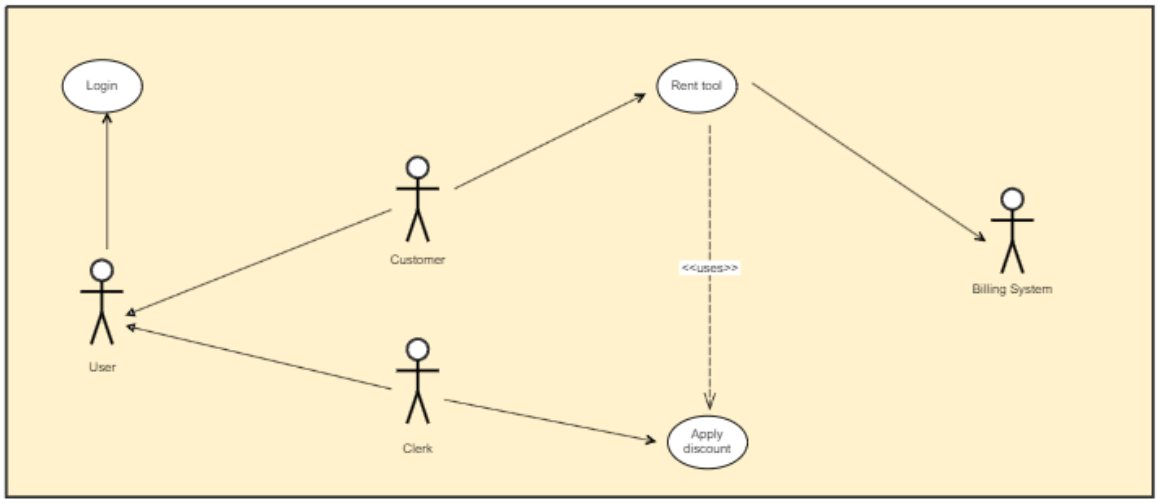
5.2 Infrastructure diagram



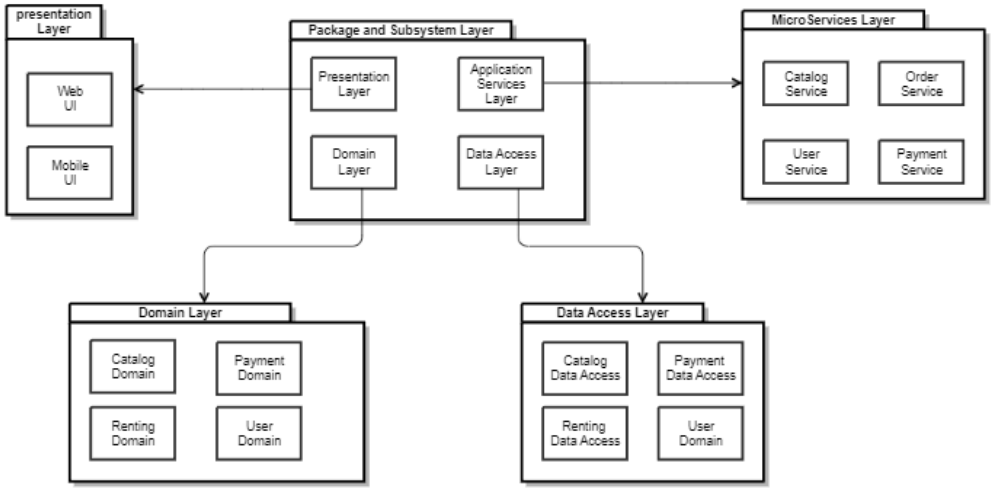
SYSTEM CONSTRAINTS

- The application is a point-of-sale tool for a store, like Home Depot, that rents big tools.
- Customers rent a tool for a specified number of days.
- When a customer checks out a tool, a Rental Agreement is produced.
- The store charges a daily rental fee, whose amount is different for each tool type.
- Some tools are free of charge on weekends or holidays.
- Clerks may give customers a discount that is applied to the total daily charges to reduce the final charge.

USE CASES



PACKAGE AND SUBSYSTEM LAYERING



8.1 Microservices Identification

- **Catalog Service:**
  - Manages tools information, including details, pricing, and availability.
- **Order Service:**
  - Handles order processing, including creating, updating, and managing customer renting orders.
- **User Service:**
  - Manages user accounts, authentication, and authorization.
- **Payment Service:**
  - Deals with payment processing and integration with payment gateways.

8.2 Package and Subsystem Layering

- **Presentation Layer**
  - Handles user interfaces and interactions.
  - Packages: Web UI, Mobile App UI.
- **Application Services Layer**
  - Manages application-specific logic and workflows.
  - Subsystems: Catalog Application Service, Order Application Service, User Application Service, Payment Application Service.
- **Domain Layer**
  - Contains business logic and entities.
  - Subsystems: Catalog Domain, Order Domain, User Domain, Payment Domain.
- **Data Access Layer**
  - Responsible for database interactions.
  - Subsystems: Catalog Data Access, Order Data Access, User Data Access, Payment Data Access.

- **Database Per Service**
  - Each microservice manages its own data store.
- **Database Per Service**
  - Each microservice manages its own data store.
- **Communication protocols**
  - **Synchronous:**
    - HTTP/REST requests from client applications to microservices.
    - Access to list of tools and details
    - See user's details including order lists.
    - Interact with databases synchronously using direct queries.
  - **Asynchronous**:
    - Microservices publish events or messages to a message broker to communicate asynchronously with other microservices. Subscribe to events to react to changes in the system.
    - Implement asynchronous patterns for handling long-running processes, such as by offloading tasks to a message queue or triggering background jobs.
    - Handle background tasks, processing large datasets, or performing time-consuming operations without impacting the responsiveness of the main application.
    - Changes to the database state may trigger events or messages, notifying other microservices of data updates.
    - Asynchronous cache updates shall be triggered when data changes, ensuring that caches are kept up to date.

      Publish events related to security events, such as successful logins or unauthorized access attempts.

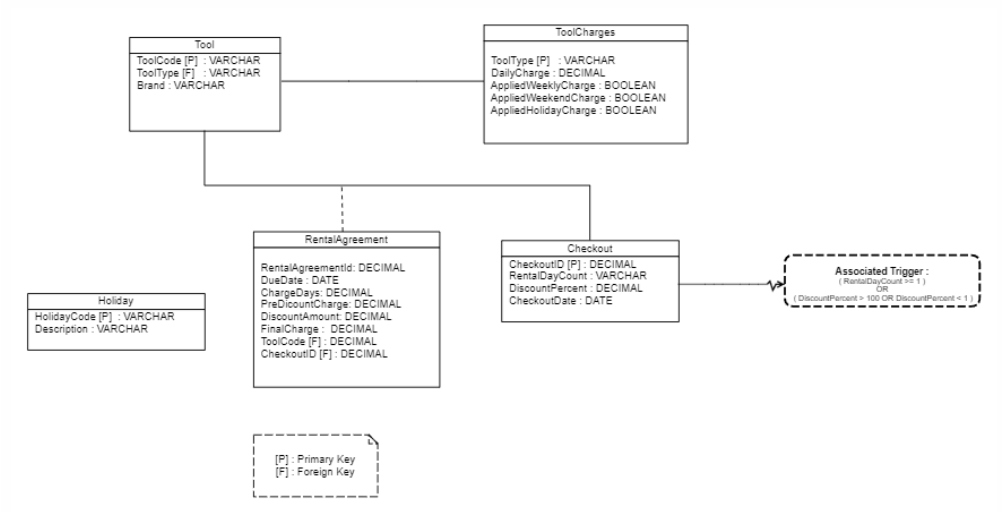
- **Communication protocols**
    - Each microservice incorporates logging and monitoring capabilities.
    - Centralized logging and monitoring tools track the health and performance of individual services.
- **Security and Authorization**
    - Authentication and authorization mechanisms are implemented at the microservices level.
    - Access to sensitive operations, such as placing an order, is restricted based on user roles.
    - Align with FedRAMP certificate requirements.
- **Deployment Strategies**
    - Containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) are employed for efficient deployment and scaling.
- **Scalability and Performance**
    - Each microservice is designed for horizontal scalability, and caching strategies are implemented for performance optimization.
- **Governance and Compliance**
    - Versioning, dependency management, and compliance checks are part of the development and deployment processes.
- Accessibility
    - Design user interfaces (UI) with accessibility in mind. Use semantic HTML, provide alternative text for images, and ensure a logical reading order for screen readers.
    - Ensure that all interactive elements are keyboard accessible and use ARIA (Accessible Rich Internet Applications) roles and attributes appropriately.
    - Design your microservices APIs to be accessible and user-friendly. Use clear and consistent naming conventions and provide comprehensive documentation with examples.

- Consider providing RESTful APIs that adhere to best practices for accessibility, making it easier for developers and applications to consume your services.
- Provide clear and meaningful error messages that are presented in accessible ways. Users with disabilities should be able to understand error messages through alternative means, such as screen readers.
- Stay informed about accessibility standards, such as the Web Content Accessibility Guidelines (WCAG) and strive to comply with them. Adhering to standards helps ensure a high level of accessibility.

- **Confidentiality**
  - Data subject to regulations like GDPR, should implement robust consent management systems. Ensure that users provide explicit consent for the collection and processing of their personal data.

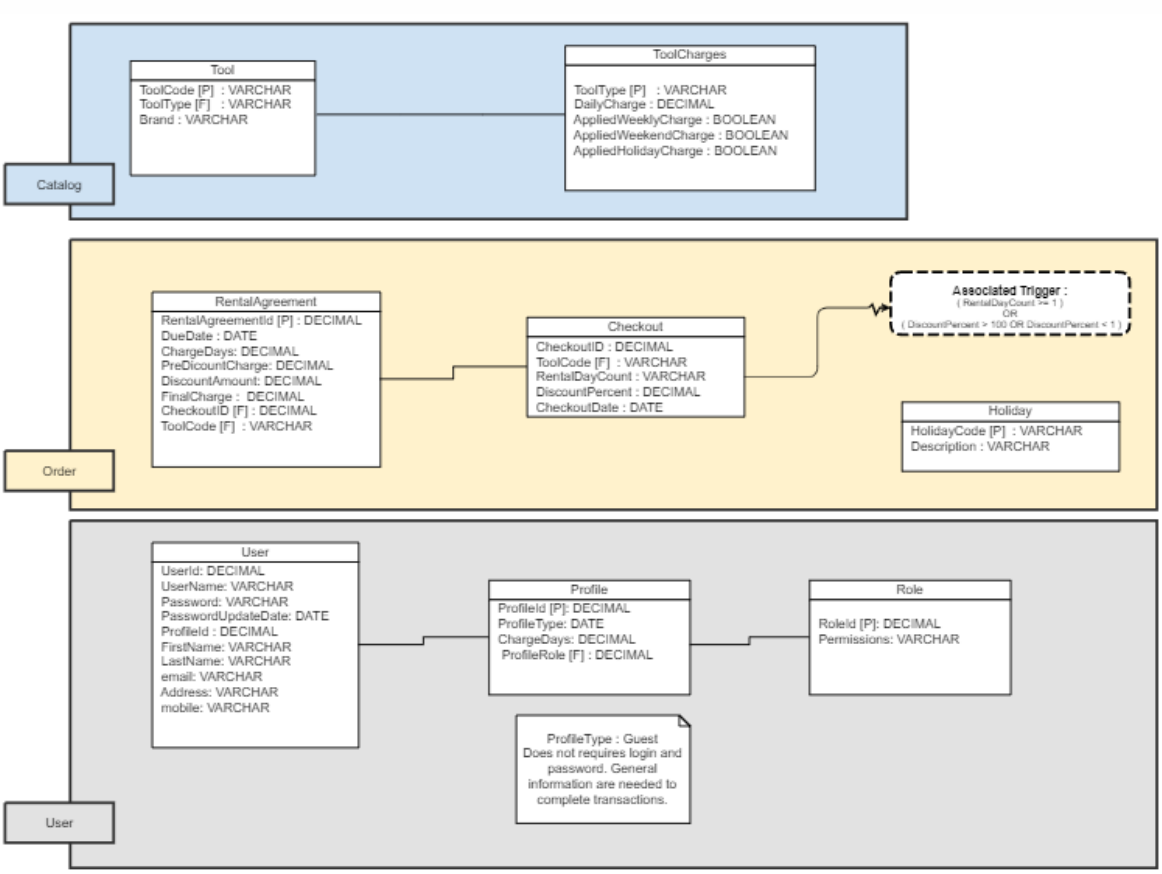
DATA MODEL

9.1 General Schema



9.2 Decomposed Schema

Each microservice is designed to be a self-contained unit of functionality and has its own database. This approach allows teams to work independently on different microservices, promoting scalability, agility, and easier maintenance.



API DEFINITION

10.1 RentController Class

The RentController class is a Spring WebFlux RESTful controller responsible for handling HTTP requests related to renting and generating rental agreements for tools. It utilizes reactive programming with Spring's ReactiveRedisTemplate for asynchronous communication.

10.2 Constructors

RentController(ReactiveRedisTemplate<String, Cart> redisTemplate): Constructor for the RentController class, taking a ReactiveRedisTemplate as a parameter to initialize Redis-related operations.

10.3 Methods

index(): Handles HTTP GET requests to the root ("/") endpoint. It returns a JSON response with the name and version of the Renting API.

- **Method: GET**

  Endpoint: "/"

  Response: JSON string

  checkout(Checkout checkout): Handles HTTP POST requests to the "/checkout" endpoint, where users can request a rental agreement based on tool checkout details. It validates the checkout details, generates a rental agreement, and logs the result.

- **Method: POST**

  Endpoint: "/checkout"

  Request Body: JSON representing a Checkout object

  Response: ResponseEntity containing the rental agreement or error message

  validateCheckout(Checkout checkout): Private method to validate the provided checkout details. It ensures that the rental day count is greater than or equal to 1 and the discount percent is within the range of 0 to 100.

  Parameters: Checkout object

  Throws: IllegalArgumentException if validation fails

  generateRentalAgreement(Checkout checkout): Private method to generate a RentalAgreement based on the provided checkout details. It creates a rental agreement, sets attributes based on checkout, and prints the agreement details to the console.

Parameters: Checkout object

Returns: RentalAgreement object

Endpoints

Root Endpoint

- **Method: GET**

  Endpoint: "/"

  Description: Returns a JSON response with the name and version of the Renting API.

  Checkout Endpoint

- **Method: POST**

  Endpoint: "/checkout"

  Request Body: JSON representing a Checkout object

  Description: Handles checkout requests, validates check-out details, and generates a rental agreement.

  Generate Rental Agreement Endpoint

- **Method: POST**

  Endpoint: "/generateRentalAgreement"

  Request Body: JSON representing a Checkout object

  Description: Generates a rental agreement based on the provided checkout details and prints the details to the console.

10.4 Tools API

- **Get All Tools**

  Endpoint: /tools

  Method: GET

  Description: Fetches all tools from the database.

  Response: JSON array containing tool information.

  Error Response: Status 400 with an error message if there's an issue fetching tools.

- **Get All Tool Charges**

  Endpoint: /toolCharges

  Method: GET

  Description: Fetches all tool charges from the database.

  Response: JSON array containing tool charge information.

  Error Response: Status 400 with an error message if there's an issue fetching tool charges.

- **Get Tool by Tool Code**

  Endpoint: /tool/:toolCode

  Method: GET

  Description: Fetches a specific tool based on the provided tool code.

  Parameters: toolCode - Tool code of the tool to be fetched.

  Response: JSON object containing tool information.

Error Response: Status 400 with an error message if there's an issue fetching the tool.

- **Get Tool Charges by Tool Type**

Endpoint: /toolCharges/:toolType

Method: GET

Description: Fetches tool charges based on the provided tool type.

Parameters: toolType - Tool type for which charges are to be fetched.

Response: JSON array containing tool charge information.

Error Response: Status 400 with an error message if there's an issue fetching tool charges by tool type.