Project Overview

Readers' Haven-Online Bookstore

Objective:

The project is an online bookstore management system that incorporates features like book catalog management, user sessions, shopping cart functionality, and interaction with a PostgreSQL database. The application is designed in Java and utilizes a relational database for storing book and user information. Additionally, it employs Redis for caching purposes

Key Components:

1-Session Management:

Utilize Redis to securely store and manage user session data.

Implement mechanisms for session expiration and renewal to enhance user interactions.

Ensure secure handling and storage of session identifiers to prevent unauthorized access.

2-Caching Strategy:

Identify critical data, such as popular books, search results, and user-specific information, for caching.

Implement caching mechanisms to reduce database load and enhance response times.

Develop a strategy for cache invalidation and refreshing to maintain data consistency and accuracy.

Component Overview

1-Class Overview :

1.1-Entities:

A-Authors:

Represents authors in the bookstore.

Attributes: authorId, name, books.

B-Books:

Represents books in the bookstore.

Attributes: bookId, title, author, price, quantityInStock.

C-Cart:

Represents items in the shopping cart.

Attributes: cartId, bookId, quantity.

D-Customers:

Represents customers of the bookstore.

Attributes: customerId, name, email, password, address.

E-Inventory:

Represents inventory entries for books.

Attributes: inventoryId, bookId, quantityInStock.

F-Orders:

Represents orders placed by customers.

Attributes: orderId, customer_id, order_date, status, items.

1.2-Session Management:

A-LoginRequest:

Represents user login requests.

Attributes: email, password.

B-UserSession:

Represents a user session storing session attributes.

Attributes: sessionData.

C-UserSessionManager:

Manages user sessions, including creation, retrieval, and updating.

2-DAO Layer Overview:

A-AuthorsRepository:

Manages database interactions for authors.

B-BooksRepository:

Manages database interactions for books.

C-CartRepository:

Manages database interactions for shopping cart entries.

D-CustomersRepository:

Manages database interactions for customers.

E-InventoryRepository:

Manages database interactions for inventory entries.

F-OrdersRepository:

Manages database interactions for orders.

3-Service Layer Overview:

A-AuthorsService:

Manages author-related business logic.

B-BookService:

Manages book-related business logic, including caching strategies.

C-CartService:

Manages shopping cart-related business logic.

D-CustomerService:

Manages customer-related business logic.

E-InventoryService:

Manages inventory-related business logic.

F-OrderService:

Manages order-related business logic.

4-Controller Layer Overview:

A-AuthorsController:

Responsibility:

Manages operations related to authors in the bookstore.

Endpoints:

POST /api/authors: Creates a new author.

GET /api/authors: Retrieves a list of all authors.

GET /api/authors/{id}: Retrieves an author by ID.

PUT /api/authors/{id}: Updates an existing author.

DELETE /api/authors/{id}: Deletes an author.

B-CartController:

Responsibility:

Manages operations related to the shopping cart.

Endpoints:

POST /api/cart: Adds an item to the shopping cart.

GET /api/cart: Retrieves all items in the shopping cart.

GET /api/cart/getByCustomerId/{customerId}: Retrieves cart items by customer ID.

PUT /api/cart/item/{id}: Updates an item in the shopping cart.

DELETE /api/cart/item/{id}: Deletes an item from the shopping cart.

C-CustomersController:

Responsibility:

Manages operations related to customers.

Endpoints:

POST /api/customers: Creates a new customer.

GET /api/customers: Retrieves a list of all customers.

GET /api/customers/{id}: Retrieves a customer by ID.

PUT /api/customers/{id}: Updates an existing customer.

DELETE /api/customers/{id}: Deletes a customer.

D-InventoryController:

Responsibility:

Manages operations related to inventory.

Endpoints:

POST /api/inventory: Adds an item to the inventory.

GET /api/inventory: Retrieves all items in the inventory.

GET /api/inventory/{id}: Retrieves an inventory item by ID.

PUT /api/inventory/{id}: Updates an inventory item.

DELETE /api/inventory/{id}: Deletes an inventory item.

E-LoginController:

Responsibility:

Manages user authentication and session handling.

Endpoints:

POST /api/login: Authenticates a user and creates a session.

GET /api/getUserById/{id}: Retrieves user information by ID.

F-OrdersController:

Responsibility:

Manages operations related to orders.

Endpoints:

POST /api/orders: Creates a new order.

GET /api/orders: Retrieves a list of all orders.

GET /api/orders/getByCustomerId/{customerId}: Retrieves orders by customer ID.

GET /api/orders/{id}: Retrieves an order by ID.

PUT /api/orders/{id}: Updates an existing order.

DELETE /api/orders/{id}: Deletes an order.

Scaling and Maintaining Redis :

Scaling and maintaining Redis components within an infrastructure involves addressing the challenges related to performance, availability, and data consistency. Below are detailed notes on scaling and maintaining Redis components:

Scaling Redis:

1. Vertical Scaling (Scaling Up):
   Description: Increasing the resources (CPU, RAM) of a single Redis instance.
   Pros: Simplicity, easier to manage initially.
   Cons: Limited scalability, potential performance bottlenecks.

2. Horizontal Scaling (Scaling Out):
   Description: Distributing data and load across multiple Redis instances (sharding).
   Pros: Improved scalability, better load distribution.
   Cons: Complexity in managing distributed systems.

3. Redis Cluster:
   Description: Redis Cluster is a built-in solution for distributed Redis.
   Pros: Automatic sharding, high availability.
   Cons: Requires careful planning, not suitable for all use cases.

4. Partitioning:
   Description: Dividing the dataset into smaller partitions.
   Pros: Efficient use of resources, improved parallelism.
   Cons: Complexity in managing partitioning strategy.

5. Replication:
   Description: Creating replicas of Redis instances to improve fault tolerance.
   Pros: Improved availability, data redundancy.
   Cons: Increased memory consumption, potential network overhead.

Maintenance Considerations:

1. Monitoring:
Regularly monitor Redis instances for performance, memory usage, and potential issues.
Utilize monitoring tools (e.g., Redis Sentinel, third-party monitoring solutions) to track key metrics.

2. Backups:
Implement a robust backup strategy to prevent data loss.
Schedule regular backups and test the restoration process.

3. Security:
Regularly update Redis to the latest stable version to patch security vulnerabilities.
Implement secure configurations, including authentication and encryption.

4. Data Persistence:
Choose an appropriate persistence strategy based on your requirements (RDB snapshots, AOF logs, or a combination).
Regularly check and optimize the persistence settings.

5. Capacity Planning:
Regularly review and adjust the capacity of Redis instances based on usage patterns.
Use performance testing to identify bottlenecks and plan for future growth.

6. High Availability:
Deploy Redis in a high-availability setup using Redis Sentinel or other clustering solutions.
Regularly test failover scenarios to ensure quick recovery.

7. Connection Pooling:
Implement connection pooling to efficiently manage and reuse connections to Redis.
Adjust connection pool settings based on the expected load.

8. Resource Management:
Regularly review and optimize resource usage, including memory, CPU, and network bandwidth.
Implement proper resource isolation and allocation.

9. Automated Deployment:
Use automation tools (e.g., Ansible, Terraform) for deploying and managing Redis instances.
Enable easy scaling and configuration changes through automation.

10. Documentation:
Maintain comprehensive documentation for Redis configurations, deployment procedures, and maintenance tasks.
Document troubleshooting steps and common issues.