Stream Processing and Analytics

Case Study: Inventory Management System(IMS) using AKKA

Abstract

This case study demonstrates the implementation of a concurrent inventory management system using the Akka actor model in Scala 3. The system showcases how actor-based programming can be leveraged to create a robust, scalable solution for managing inventory in real-time applications such as e-commerce platforms. The study presents a simple yet functional inventory management system that allows for adding items, removing items, and viewing the current inventory state. By utilizing Akka's actor model, the system ensures thread-safe operations on shared inventory data, making it suitable for high-concurrency environments.

Key aspects of the implementation include:

1. The use of typed actors to represent the inventory system, ensuring type safety at compile-time.
2. Asynchronous message passing for all inventory operations, demonstrating Akka's non-blocking communication model.
3. The application of the ask pattern to handle responses from the actor system in a user-friendly console interface.
4. Proper handling of concurrent operations and potential race conditions through actor encapsulation.

Architecture

1. Actor System Structure:

- The system is built around a single root actor, InventoryActor, which manages the entire inventory state.
- The ActorSystem is created in the main InventoryDemo object, serving as the entry point and runtime environment for the actor.

2. Message-Driven Communication:

- All interactions with the inventory are performed through message passing.
- Defined message types:
    - Commands: AddItem, RemoveItem, GetInventory
    - Responses: ItemAdded, ItemRemoved, InventoryState, OperationFailed

3. State Management:

- The inventory state is encapsulated within the InventoryActor.
- State is represented as an immutable Map[String, InventoryItem].
- Each operation creates a new state, ensuring thread-safety.

4. Concurrency Model:

- Akka's actor model inherently handles concurrency.
- All inventory operations are processed sequentially within the actor, eliminating race conditions.

5. Asynchronous Operations:

- The system uses Akka's ask pattern for non-blocking communication with the actor.
- Responses are handled asynchronously using Scala's Future and onComplete callbacks.

6. Scalability:

- The actor-based design allows for easy scaling by distributing actors across multiple nodes (not implemented in this basic version but inherently supported by Akka).

7. Fault Tolerance:

- Akka's supervision strategy can be implemented to handle failures and restart actors if necessary (not explicitly shown in this basic implementation).

8. Separation of Concerns:

- The actor logic (inventory management) is separated from the user interface logic (console interaction).
- This separation allows for easy replacement of the UI layer without affecting the core business logic.

9. Typed Actor System:

- The use of Akka Typed ensures compile-time type safety for messages.

10. Dependency Injection:

- Scala 3's given instances are used to provide necessary dependencies like ActorSystem, Scheduler, and ExecutionContext.

11. Extensibility:

- The system can be easily extended to include more complex operations or additional actors for different aspects of inventory management.

12. Logging and Monitoring:

- While not explicitly implemented, Akka provides built-in logging capabilities that can be easily integrated.

13. Configuration:

- Akka's configuration system (using HOCON) can be utilized for more complex setups.

Use Cases

1. E-commerce Platforms:

- Managing product inventory across multiple warehouses
- Handling concurrent order processing and inventory updates
- Real-time stock level updates for customers

2. Retail Point of Sale (POS) Systems:

- Managing inventory across multiple store locations
- Handling concurrent sales transactions and inventory updates
- Real-time synchronization between in-store and online inventory

Other use cases:

Warehouse, Supply Chain, Airline Reservation, Event Ticketing , Rental Services

Inventory Management System (IMS)

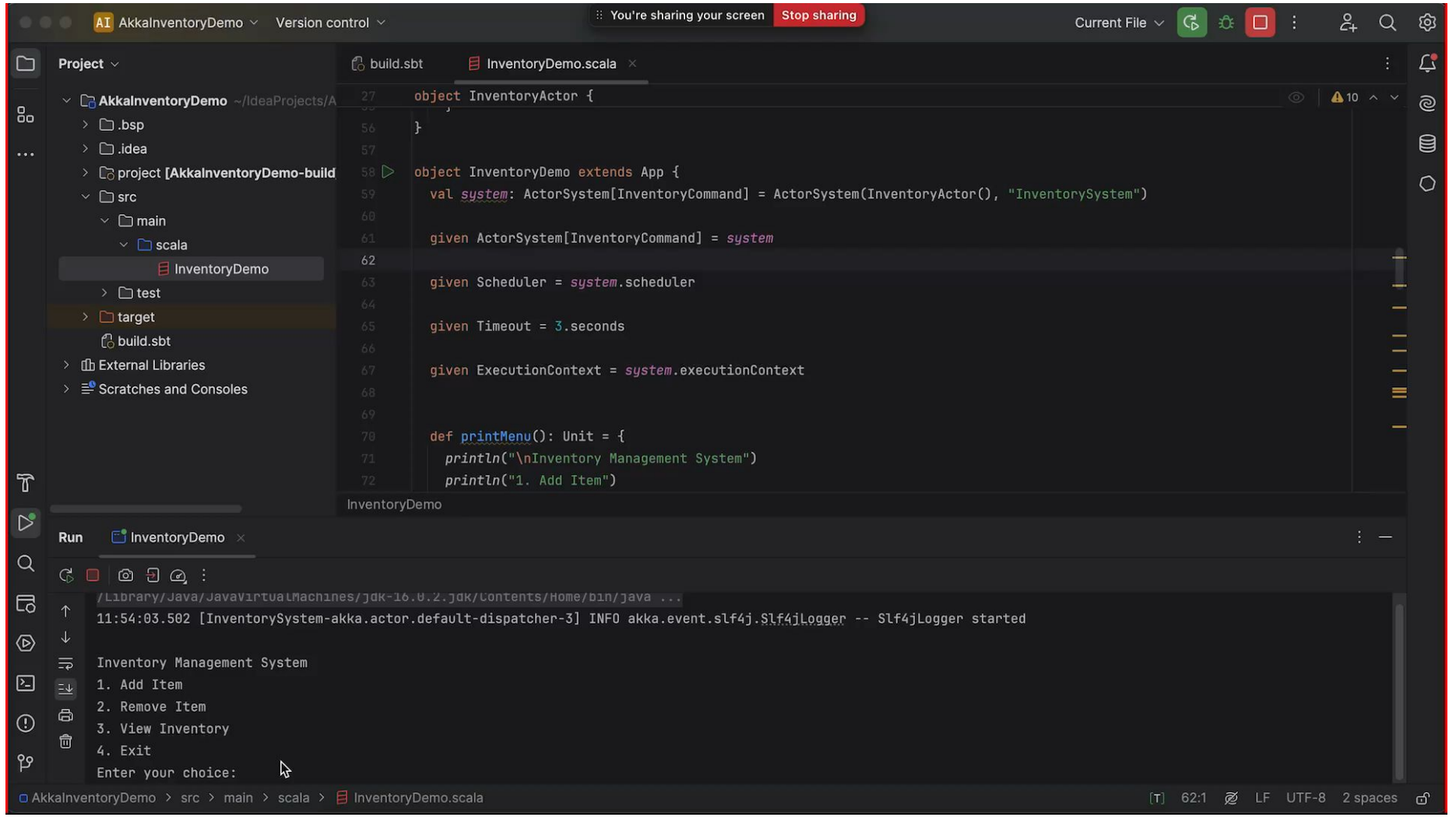
Environment Setup

- Install Scala
- Install Intellij IDEA
- Create sbt project in IDE.
- Modify build.sbt

build.sbt × InventoryDemo.scala

```
1   ThisBuild / version := "0.1.0-SNAPSHOT"
2
3   ThisBuild / scalaVersion := "3.3.3"
4
5   lazy val root = (project in file("."))
6     .settings(
7       name := "AkkaInventoryDemo",
8       libraryDependencies ++= Seq(
9         "com.typesafe.akka" %% "akka-actor-typed" % "2.8.5",
10        "ch.qos.logback" % "logback-classic" % "1.5.6"
11      )
12    )
```

Working Demo



Step By Step Demo

- Adding first item in inventory

```
/Library/Java/JavaVirtualMachines/jdk-16.0.2.jdk/Contents/Home/bin/java ...
11:54:03.502 [InventorySystem-akka.actor.default-dispatcher-3] INFO akka.event.slf4j.Slf4jLogger -- Slf4jLogger started

Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 1
Enter item ID: 1
Enter item name: Pencil
Enter quantity: 2
Press Enter to continue...
Added: InventoryItem(1,Pencil,2)
```

- Viewing first item

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 3
Press Enter to continue...
Current Inventory:
1: Pencil - Quantity: 2
```

- Adding second item

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 1
Enter item ID: 2
Enter item name: Books
Enter quantity: 3
Press Enter to continue...
Added: InventoryItem(2,Books,3)
```

- Viewing all items

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 3
Press Enter to continue...
Current Inventory:
1: Pencil - Quantity: 2
2: Books - Quantity: 3
```

- Removing 1 quantity from the first item

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 2
Enter item ID: 1
Enter quantity to remove: 1
Press Enter to continue...
Removed 1 of item 1
```

- Removing 1 more quantity from the first item

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 2
Enter item ID: 1
Enter quantity to remove: 1
Press Enter to continue...
Removed 1 of item 1
```

- Viewing items after removal

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 3
Press Enter to continue...
Current Inventory:
1: Pencil - Quantity: 0
2: Books - Quantity: 3
```

- Exiting

```
Inventory Management System
1. Add Item
2. Remove Item
3. View Inventory
4. Exit
Enter your choice: 4
Exiting...
11:56:31.900 [InventorySystem-akka.actor.default-dispatcher-3] INFO akka.actor.CoordinatedShutdown -- Running CoordinatedShutdown with reason [ActorSystemTerminateRea

Process finished with exit code 0
```

Conclusion

We have implemented a simple inventory management system using the Akka actor model. The system allows users to add, remove, and view inventory items. The use of Akka actors provides a good foundation for building a concurrent and fault-tolerant system.

Inferences

1. **Actor-based architecture**: The system uses Akka actors to manage the inventory, which allows for concurrent and fault-tolerant processing of inventory operations.
2. **Domain-driven design**: The code uses domain-specific models (e.g., InventoryItem, InventoryCommand, InventoryResponse) to represent the business domain, making it easier to understand and maintain the code.
3. **Asynchronous processing**: The system uses asynchronous processing to handle inventory operations, which allows for non-blocking and efficient processing of requests.
4. **Error handling**: The code includes basic error handling mechanisms, such as returning error messages to the user when an operation fails.
5. **Simple user interface**: The system provides a simple text-based user interface to interact with the inventory management system.

Potential areas for improvement

1. **Persistence**: The system does not persist the inventory data, which means that the data will be lost when the system is restarted.
2. **Validation**: The system does not perform thorough validation of user input, which could lead to errors or security vulnerabilities.
3. **Security**: The system does not implement any security mechanisms, such as authentication or authorization, to protect the inventory data.
4. **Scalability**: The system may not be designed to handle large volumes of inventory data or a large number of concurrent users.

Overall, the built system provides a good foundation for building an inventory management system, but it may require additional features and improvements to make it more robust and efficient.