Architectural Specification

Helix

August 2024

Contents

1 Introduction

This is the outline for the architectural blueprint for the Smart Inventory system, a comprehensive web-based application designed to streamline and automate inventory management processes for businesses of various types, including restaurants and laboratories. Building upon the functionalities outlined in the requirement specification.

2 Objectives

The primary objective of this document is to guide stakeholders involved in the development of the Smart Inventory System. It aims to define the architectural framework, constraints and technologies used to successfully implement the desired features and functionalities. This document details the system architecture that will ensure:

- Reliability
- Scalability
- Security
- Usability
- Maintainability

3 Quality Requirements

3.1 Security

The system must ensure data protection through end-to-end encryption, secure user authentication, and role-based access control to prevent unauthorized access and maintain data confidentiality. The following subsections detail the tactics we will use to achieve this quality requirement and how we will quantify their implementation.

3.1.1 Authentication

Tactics

- Implement strong password policies
- Provide login attempt awareness

Quantification

- Password strength: Minimum 12 characters, including uppercase, lowercase, numbers, and special characters
- Login attempt notification: 100% of suspicious login attempts reported to admin or user

3.1.2 Access Control

Tactics

- Implement role-based access control (RBAC)
- Map system functions to specific roles

Quantification

- RBAC coverage: 100% of system components (application features, APIs, data access)
- Function mapping: 80% of system functions mapped to specific roles

3.1.3 API Security

Tactics

- Implement API gateway
- Authenticate and authorize all API calls

Quantification

- API gateway coverage: 100% of API calls routed through a single entry point
- API call security: 100% of API calls authenticated and authorized before reaching services

3.1.4 Secure User Authentication

Tactics

- Use secure session management

Quantification

- Session security: 100% of user sessions have proper timeout mechanisms

3.1.5 Audit and Monitoring

Tactics

- Implement comprehensive logging
- Conduct regular security audits

Quantification

- Logging coverage: 100% of security-related events logged
- Audit frequency: Quarterly security audits with 100% of identified vulnerabilities addressed within 30 days

3.2 Reliability

The system should be reliable in terms of accurately tracking inventory levels, generating timely alerts, order placement and generating accurate reports.

3.2.1 Alert Management

Tactics

- Implement real-time monitoring of inventory levels
- Develop efficient alert triggering mechanisms

Quantification

- Alert Generation Speed: Generate and deliver inventory alerts within 60 seconds of triggering conditions being met
- Alert Accuracy: Achieve 95% accuracy in alert generation (false positives/negatives ¡ 5%)

3.2.2 Order Processing

Tactics

- Implement asynchronous processing for non-critical order components
- Utilize caching mechanisms to improve response times

Quantification

- Processing Speed: Process 95% of orders within 5 minutes of submission during normal operation

3.2.3 Reporting

Tactics

- Implement efficient data aggregation techniques
- Utilize background processing for report generation

Quantification

- Report Timeliness: Generate 100% of scheduled reports within their defined time-frames
- Report Accuracy: Ensure 95% accuracy in all generated reports

3.2.4 System Stability

Tactics

- Implement robust error handling and recovery mechanisms

Quantification

- Mean Time Between Failures (MTBF): Achieve an MTBF of at least 720 hours (30 days) for critical system components

3.3 Usability

The web interface should be user-friendly and intuitive for administrators, inventory controllers, and end users to perform their respective tasks efficiently.

3.3.1 Task Completion

Tactics

- Implement intuitive user interface design
- Provide clear and concise instructions for each task
- Utilize progressive disclosure to simplify complex tasks

Quantification

- First-time Users: Achieve a 90% success rate for first-time users completing core tasks without assistance
- Experienced Users: Maintain a 98% success rate for experienced users

3.3.2 Task Efficiency

Tactics

- Optimize user interface for common tasks

Quantification

- New User Task Completion Times:
  - Inventory lookup: less than 30 seconds
  - Order placement: less than 2 minutes
  - Generate basic report: less than 90 seconds
- Experienced User Task Completion: 30% faster than new users for all tasks

3.3.3 User Satisfaction

Tactics

- Conduct regular user surveys and feedback sessions
- Implement user-requested features and improvements

Quantification

- Satisfaction Score: Achieve an average satisfaction score of 4.2 or higher on a 5-point scale through user surveys

3.3.4 Learnability

Tactics

- Provide interactive tutorials and tool tips
- Implement a comprehensive help system
- Design intuitive navigation and information architecture

Quantification

- New User Proficiency: 80% of new users should be able to use all basic functions without referencing help documentation after a 30-minute on boarding session

3.3.5 Error Prevention and Handling

Tactics

- Implement input validation and error prevention mechanisms
- Use confirmation dialog for critical actions

Quantification

- Error Rate: Maintain a user error rate below 5% for all critical tasks (e.g., inventory updates, order processing)

3.3.6 Navigation Efficiency

Tactics

- Implement a clear and consistent navigation structure
- Optimize the search functionality

Quantification

- Feature Accessibility: Enable users to access any main feature within 4 clicks from the home screen
- Search Efficiency: Implement a search function that returns relevant results for 95% of queries within 5 seconds

3.3.7 Customization

Tactics

- Implement user-configurable dashboards
- Allow customization of reports and views
- Provide user-specific shortcuts and favorites

Quantification

- Dashboard Customization: Allow users to customize their dashboard, saving at least 10% of frequently used functions for quick access

3.3.8 UI Consistency

Tactics

- Develop and adhere to a comprehensive style guide
- Implement reusable UI components
- Conduct regular UI audits

Quantification

- Design Consistency: Maintain 95% consistency in design elements (colors, fonts, button styles) across all pages
- Terminology Consistency: Ensure 95% consistency in terminology and labeling throughout the application

3.4 Scalability

The system should be designed to handle growth in inventory, users, and transactions, ensuring that performance remains optimal as the business expands.

3.4.1 User Concurrency

Tactics

- Implement efficient load balancing techniques
- Utilize caching mechanisms to reduce server load
- Employ horizontal scaling for web and application servers

Quantification

- Response Time: Maintain response times under 10 seconds for 95% of requests with 1000 concurrent users
- Scalability: Scale to support 5000 concurrent users within 24 hours notice

3.4.2 Transaction Processing

Tactics

- Implement asynchronous processing for non-critical transactions
- Optimize database indexes for common transactions

Quantification

- Processing Capacity: Handle a minimum of 100 transactions per second (TPS) during peak hours
- Success Rate: Maintain a 90% success rate for all transactions under full load

3.4.3 Data Volume

Tactics

- Implement data partitioning for large datasets
- Employ NoSQL databases for handling large volumes of unstructured data

Quantification

- Growth Capacity: Handle annual growth of 50% in inventory items for 5 consecutive years without major architecture changes

3.4.4 API Scalability

Tactics

- Implement API rate limiting and throttling
- Utilize caching for frequently requested API resources

Quantification

- Request Handling: Handle 1000 API requests per second with a 90% success rate

3.4.5 Elastic Scaling

Tactics

- Implement auto-scaling groups for dynamic resource allocation
- Use serverless architectures for highly scalable components

Quantification

- Scaling Speed: Automatically scale resources up or down based on demand, with scaling events completed within 5 minutes

3.4.6 Database Scalability

Tactics

- Implement efficient indexing strategies
- Use caching layers to reduce database load

Quantification

- Write Performance: Handle 1000 write operations per second with 95% success rate

3.5 Maintainability

The system should be designed and implemented in a way that facilitates easy maintenance, updates, and extensions. The following subsections detail the tactics we will use to achieve this quality requirement and how we will quantify their implementation.

3.5.1 Overall Maintainability Metrics

Tactics

- Implement best practices for code organization and structure
- Utilize design patterns that promote maintainability

Quantification

- Bug Fix Efficiency: Bug fixes shall not require more than two hours of additional time due to code maintainability issues
- Feature Implementation Efficiency: Implementation of new features shall not incur more than two hours of overhead time attributable to code maintainability challenges

3.5.2 Comprehensive Documentation

Tactics

- Develop and maintain up-to-date system architecture documentation
- Encourage and enforce inline code comments for complex algorithms and non-obvious design decisions

Quantification

- Documentation Coverage: Achieve 95% documentation coverage for all system components and APIs
- Documentation Freshness: Ensure all documentation is reviewed and updated at least once before every demo

3.5.3 Modular Design

Tactics

- Develop the system using loosely coupled modules with clearly defined interfaces
- Utilize dependency injection to minimize direct dependencies between components

Quantification

- Dependency Reduction: Reduce direct dependencies between components by 50% through the use of dependency injection

3.5.4 Code Quality Assurance

Tactics

- Adhere to consistent coding standards and style guides across the project
- Implement automated code quality checks and linting tools in the development pipeline
- Conduct regular code reviews to ensure adherence to best practices

Quantification

- Code Standard Compliance: Achieve 98% compliance with defined coding standards
- Code Review Coverage: Ensure 100% of new code is reviewed by at least one other developer before merging

3.5.5 Test Coverage

Tactics

- Implement comprehensive unit testing for all modules
- Develop integration tests for inter-module interactions

Quantification

- Unit Test Coverage: Achieve 85% code coverage for unit tests
- Integration Test Coverage: Ensure 85% of module interactions are covered by integration tests

3.5.6 Version Control and Release Management

Tactics

- Automate the build and deployment process

Quantification

- Build Success Rate: Achieve a 85% success rate for automated builds
- Deployment Time: Reduce deployment time to less than 30 minutes for major releases

4 Architectural Strategy

For our smart inventory application, user experience is paramount to its success. We recognize that the efficiency and ease with which users interact with our system directly impacts its adoption and effectiveness in managing inventory.

To this end, we've chosen to structure our architectural design around our quality requirements, with a particular emphasis on user needs. This approach ensures that every aspect of our application, from the user interface to the backend processes, is optimized for the end-user's experience.

By prioritizing quality requirements derived from user stories and client feedback, we're creating a robust foundation for our system architecture.

This user-centric, quality-driven architectural approach will result in a smart inventory application that not only meets technical specifications but also delivers a superior experience for inventory managers, warehouse staff, and other stakeholders.

5 Architectural Styles

5.1 MVVM

5.1.1 Description

The front-end of the Smart Inventory system will implement the MVVM architectural pattern, separating the user interface (View) from the business logic and data (Model) through a mediating ViewModel.

5.1.2 Quality requirements addressed

- Maintainability: Clear separation of concerns makes it easier to update and maintain different aspects of the front-end independently. MVVM provides a clear structure for organizing code, making it easier for developers to understand and maintain the system.

- Usability: The ViewModel can format data and handle user interactions in a way that enhances the user experience. By separating UI logic from business logic, MVVM can lead to more responsive user interfaces. The ViewModel can also ensure consistent data presentation across different views.
- Scalability: The separation allows for easier addition of new features and views without significantly impacting existing code.
- Reliability: The clear structure makes it easier to implement error handling and recovery mechanisms.

5.2 Event Driven

5.2.1 Description

The Smart Inventory system will incorporate event-driven architecture to enable real-time responsiveness and asynchronous processing. Key events, such as low stock levels or expiring products, will trigger appropriate actions, such as generating alerts or initiating automatic order placement.

5.2.2 Quality requirements addressed

- Reliability: EDA allows for asynchronous processing, ensuring that critical events are handled promptly and reliably, even under high load. Failures in one component are less likely to cascade through the entire system and events can be persisted, allowing for replay and recovery in case of failures.
- Scalability: The event-driven approach enables the system to scale efficiently by processing events in parallel and distributing the workload across multiple components. Events can be distributed across multiple consumers, enabling horizontal scaling.
- Maintainability: Decoupling event producers from event consumers allows for easier updates and modifications to system behavior. Components are typically more modular and focused on specific tasks. Clear separation between event producers and consumers.
- Usability (indirectly): Real-time event processing can lead to more responsive system behavior, potentially improving user experience.

5.3 Serverless

5.3.1 Description

The Smart Inventory system can benefit from a serverless architecture by leveraging cloud-based services to handle backend logic and infrastructure management. This approach eliminates the need to provision and manage servers, reducing operational overhead and costs. Here's how serverless architecture can be implemented in the Smart Inventory system:

API Gateway: A single API gateway will serve as the entry point for all external requests to the system. This gateway can handle authentication, authorization, and routing of requests to appropriate serverless functions.

Lambda Functions: Core functionalities of the system, like processing inventory updates, generating reports, or fulfilling stock requests, can be implemented as serverless functions. These functions are event-driven and only execute when triggered by specific events (e.g., an API call, a database update).

Database: A cloud-based database service can be used to store inventory data, user information, and other system data. Serverless architectures often integrate seamlessly with managed database services offered by cloud providers.

5.3.2 Quality requirements addressed

- Scalability: Resources automatically scale up or down based on demand. Eliminates the need for manual capacity planning and server management.
- Maintainability: Less time spent on server maintenance and updates. Many serverless platforms provide managed services that handle common tasks.
- Security: Smaller, more focused code units can reduce the overall attack surface. Cloud providers handle security patches for the underlying infrastructure. Easier to implement principle of least privilege with function-level permissions.

5.4 Service Orientated

5.4.1 Description

The system will be designed using SOA principles, breaking down the functionality into discrete, loosely coupled services. Each service will encapsulate a specific business capability, such as inventory management, order placement, or reporting. Services will communicate through well-defined interfaces, enabling flexibility and re-usability.

5.4.2 Quality requirements addressed

- Scalability: Services can be scaled independently, allowing the system to handle increased load and adapt to changing business needs. Workload can be distributed across multiple service instances.
- Maintainability: SOA promotes loose coupling, making it easier to update or replace individual services without affecting the entire system. Breaking down complex systems into manageable services simplifies maintenance.

- Reliability: Services can be designed with fault tolerance and redundancy, ensuring the system remains operational even if individual components fail. Critical services can be implemented with redundancy for high availability.
- Security: Each service can implement its own security measures, allowing for granular access control and data protection. Access can be controlled at the individual service level.

6 Architectural Constraints

A core architectural constraint for this project is the requirement to utilize only open-source libraries. This decision fosters an open-source development philosophy for the project itself. This constraint might actually provide greater advantages than disadvantages. Here's why this constraint is important:

- Transparency and Collaboration:
  - Open-source libraries offer the advantage of publicly available code.
  - This transparency allows for deeper understanding, easier debugging, and potential contributions from the broader developer community.
- Alignment with Open-Source Project Goals:
  - By relying solely on open-source libraries, we ensure compatibility with an open-source project philosophy.
  - This means the project itself can potentially be released as open-source, allowing others to benefit from and contribute to its development.
- Long-Term Sustainability:
  - Open-source libraries often have active communities and ongoing maintenance.
  - This reduces the risk of relying on proprietary libraries that might become unsupported or unavailable in the future.

While this constraint might limit the available options for libraries and greater capabilities, the benefits of transparency, collaboration, and long-term sustainability contribute to a robust and maintainable project foundation.

7 Front-end Technology Choices

7.1 Introduction

For our Smart Inventory system's front-end development, we have chosen Angular with TypeScript. This decision aligns with our project's quality requirements. Angular, provides a robust framework for building complex, scalable

web applications, offering a component-based architecture and powerful data binding capabilities. TypeScript enhances our development process by adding static typing, which improves code quality and reduces runtime errors.

Together, Angular and TypeScript form a powerful combination that aligns well with the Model-View-ViewModel (MVVM) architectural pattern, facilitating a clear separation of concerns. This choice enables us to create a dynamic, responsive, and user-friendly interface while maintaining code organization and scalability. The following sections will explore how these technologies address our specific quality requirements and contribute to a robust front-end for the Smart Inventory system.

7.2 Angular and Typescript relation to MVVM

Angular and Typescript work together to enhance the implementation of the MVVM (Model-View-ViewModel) architecture. Angular with TypeScript in MVVM Architecture:

Model:

- Angular implements the Model through services and data models.
- TypeScript enhances this by providing interfaces and classes for precise data structure definition.
- Services, powered by TypeScript, handle data retrieval, manipulation, and business logic with strong typing.
- Generic types in TypeScript enable the creation of reusable, type-safe data models.
- This combination ensures robust and well-defined data handling in the Model layer.

View:

- In Angular, the View is represented by component templates (HTML).
- These templates define how data is displayed to the user.
- While TypeScript doesn't directly impact the View, it enhances component interaction by allowing definition of strict prop types.
- This ensures that correct data types are passed from the ViewModel to the View, reducing render-time errors.

ViewModel:

- Angular fulfills the ViewModel role through component classes.
- These classes prepare and manage data for the View and handle user interactions.
- TypeScript's strong typing helps in creating more robust ViewModels:
  - Method signatures and return types are clearly defined.
  - This reduces errors in data manipulation and presentation logic.
- TypeScript interfaces can define contracts between the ViewModel and the View, ensuring consistency.

7.3 Angular's alignment with our quality requirements

Usability:

- Angular's component-based architecture allows for the creation of reusable UI elements, enhancing consistency across the application.
- Its powerful data binding capabilities (both one-way and two-way) enable reactive and dynamic user interfaces.
- Angular excels at building Single-Page Applications (SPAs) that provide a responsive and user-friendly experience
- Angular's router module facilitates seamless navigation in SPAs, contributing to a fluid user experience.
- Built-in form validation and handling make it easier to create user-friendly input interfaces.

Maintainability:

- The clear separation of concerns in Angular (Model-View-ViewModel) makes the code-base more organized and easier to understand.
- Dependency injection in Angular promotes loose coupling between components, making it easier to modify or replace parts of the application.
- Typescript, which Angular uses, adds strong typing, improving code quality and making refactoring easier.
- Angular's CLI (Command Line Interface) automates many development tasks, maintaining consistent project structure and code quality.

Scalability:

- Angular's modular architecture allows for lazy loading of modules, improving performance as the application grows.
- Its change detection mechanism is highly optimized, ensuring efficient updates even with large amounts of data.
- RxJS integration in Angular provides powerful tools for handling complex asynchronous operations and data streams.

Open-Source:

- Angular is a mature open-source framework with a large and active community.
- The large ecosystem includes numerous third-party libraries and tools, expanding Angular's capabilities.
- Extensive documentation and community resources make it easier for developers to learn and solve problems.

7.4 Typescript alignment with our quality requirements

Security:

- Static typing helps prevent injection attacks by ensuring proper data types.
- Strict null checks reduce null pointer exceptions, a common security vulnerability.
- Compile-time checks can catch usage of deprecated or insecure APIs.
- Custom type guards allow for runtime type checking, adding an extra layer of security.

Maintainability:

- Strong typing makes refactoring safer and easier.
- Object-oriented features support better code organization and reusability.

Reliability:

- Static typing reduces runtime errors, leading to more stable applications.
- Compile-time checks catch potential issues before they reach production.
- Interfaces ensure consistent data structures across the application.

Scalability:

- Module system supports better code organization as projects grow.
- Enhanced tooling support aids in managing larger codebases effectively.

8 Back-end Technology Choices

8.1 Database: AWS DynamoDB

8.1.1 Introduction

DynamoDB aligns well with the quality requirements and architectural choices for the Smart Inventory system. Its strengths in security, reliability, and scalability, combined with its suitability for event-driven and serverless architectures, make it a strong choice.

8.1.2 DynamoDB relation to event-driven, and serverless architecture

Event-driven architecture:

- DynamoDB Streams can be used to capture data modifications and trigger Lambda functions or other event-driven processes.

Serverless architecture:

- DynamoDB is a perfect fit for serverless architectures, as it requires no server management.
- It integrates seamlessly with AWS Lambda for serverless compute.

8.1.3 DynamoDB alignment with our quality requirements

Security:

- DynamoDB offers fine-grained access control through AWS IAM.
- Data in transit is encrypted using SSL/TLS.

Reliability:

- DynamoDB provides built-in fault tolerance and high availability.
- It automatically replicates data across multiple Availability Zones.

Scalability:

- DynamoDB offers seamless scaling to handle growing data volumes.
- It provides auto-scaling capabilities for read and write capacity.

Maintainability:

- Being a fully managed service, DynamoDB reduces operational overhead.
- It requires minimal maintenance, as AWS handles patching, backups, and scaling.
- The schema-less nature allows for easy updates to data models without downtime.

8.2 API: AWS Lambda Functions and API Gateway

AWS API Gateway and Lambda functions together form a powerful combination that aligns well with the Smart Inventory system's requirements. They provide a scalable, maintainable, and secure foundation for building serverless, event-driven, and service-oriented architectures.

8.2.1 API Gateway and Lambda relation to event-driven, serverless, and service-oriented architectures

Event-driven architecture:

- API Gateway can trigger Lambda functions based on HTTP requests, acting as an event source.
- Lambda functions can be configured to respond to various AWS service events, facilitating event-driven workflows.

Serverless architecture:

- API Gateway provides a fully managed service for creating, deploying, and managing APIs.
- Lambda functions run code without provisioning or managing servers, embodying serverless.

Service-oriented architecture:

- API Gateway allows for the creation of multiple, independent API endpoints, each representing a distinct service.
- Lambda functions can be designed to handle specific business logic, aligning with the principle of single responsibility in service-oriented design.

8.2.2 API Gateway and Lambda alignment with our quality requirements

Security:

- API Gateway provides built-in DDoS protection and the ability to integrate with AWS WAF.
- Lambda executes functions in isolated environments, enhancing security.
- Both services integrate with IAM for fine-grained access control.

Reliability:

- API Gateway offers high availability and automatic scaling.
- Lambda automatically replicates functions across multiple Availability Zones.
- API Gateway provides request throttling to prevent overload.

Scalability:

- API Gateway can handle any number of API calls, automatically scaling to meet demand.
- Lambda automatically scales the number of function executions based on incoming requests.

Maintainability:

- API Gateway's interface allows for easy API management and versioning.
- Lambda's focus on single-purpose functions promotes code modularity and easier updates.
- Both services are fully managed, reducing operational overhead.

Usability:

- API Gateway provides tools for API documentation and testing.
- Lambda supports multiple programming languages, offering flexibility in development.
- Integration between these services is streamlined, simplifying the development process.

This combination of API Gateway and Lambda functions provides a robust foundation for the Smart Inventory system, supporting its architectural requirements while addressing key quality concerns.

8.3 AWS Step Functions

8.4 Communication and notifications: AWS SES and SNS

9 Standards(More in coding standards)

9.1 Prettier

Prettier is our code formatter used to maintain our linter rules to uphold our coding standards.

9.2 ESlint

For code quality and consistency, we integrate ESLint into our development process. ESLint is a widely-used static code analysis tool that helps identify and fix common programming errors and enforce coding standards. With ESLint, we ensure our code-base remains clean, readable, and maintainable.

10 Testing

10.1 Cypress

For the Smart Inventory project, we've chosen Cypress for both component and end-to-end testing of our front-end. This decision aligns with our commitment to quality and user-centered design principles.

Cypress is a powerful, modern testing tool that excels in testing Angular applications. Its seamless integration with TypeScript enhances our ability to write type-safe, robust tests that mirror our development environment.

Cypress offers several key advantages for our project:

- Real-time testing: Cypress runs tests in the same run loop as our application, providing real-time feedback and making debugging more efficient.
- Comprehensive testing: It allows for both component testing and end-to-end testing, ensuring we can verify individual UI components as well as complete user flows.
- Automatic waiting: Cypress automatically waits for elements to become available, reducing test flakiness and improving reliability.
- Time travel debugging: This feature allows us to see exactly what happened at each step of our tests, crucial for maintaining complex UI interactions.
- Network stubbing: We can easily mock API responses, allowing us to test various scenarios without depending on backend services.

Cypress is an excellent tool for testing the user interface (UI) of the Smart Inventory web application. Following User-centered design principles, Cypress allows manual and automated testing to ensure the UI is intuitive and functions as expected for various user roles. By using Cypress, we ensure our Angular-based UI is thoroughly tested, intuitive, and functions as expected for all user roles. This approach supports our quality requirements, particularly in terms of usability and reliability. Moreover, Cypress's clear syntax and extensive documentation contribute to the maintainability of our test suite, aligning with our overall project goals.

10.2 AWS Built in API tester

For testing our Smart Inventory system's back-end APIs, we've chosen to utilize AWS built-in API testing capabilities. This decision aligns with our serverless architecture and complements our use of AWS services like Lambda, API Gateway, and Step Functions. AWS provides robust testing tools that integrate seamlessly with our chosen back-end technologies:

- API Gateway Test Invocations: Allows us to test individual API endpoints directly within the AWS console, ensuring each route functions correctly before deployment.
- Lambda Console Tests: Enables us to test Lambda functions in isolation, using sample event payloads to verify correct behavior under various conditions.
- Step Functions Test Executions: Provides a visual interface to test and debug complex workflows, ensuring each step of our business processes functions as expected.
- IAM Policy Simulator: Helps verify that our security configurations are correct, supporting our security requirements.

These built-in testing capabilities offer several advantages for our project:

- Seamless Integration: Tests run in the same environment as our production code, reducing discrepancies between test and live scenarios.
- Scalability Testing: We can simulate various load conditions to ensure our serverless architecture scales as expected.
- Cost-Effective: Utilizing AWS's built-in tools eliminates the need for additional third-party testing solutions, aligning with our efficiency goals.
- Comprehensive Coverage: From individual functions to complete API workflows, we can ensure all aspects of our back-end are thoroughly tested.

By leveraging these AWS testing capabilities, we ensure our back-end services are reliable, secure, and perform as expected. This approach supports our quality requirements, particularly in terms of reliability and scalability. Furthermore, the integration with our chosen AWS services enhances the maintainability of our testing process, aligning with our overall project architecture and goals.

10.3 Postman

11 Miscellaneous

11.1 Authorisation and Authentication: AWS Cognito

For the Smart Inventory system's user authentication and authorization, we've chosen AWS Cognito. This decision aligns with our serverless architecture and integrates seamlessly with our other AWS services. It also integrates well with our service orientated design as it is a service that is included.

AWS Cognito is a fully managed identity platform that provides several key features crucial for our project:

- User Pools: Allows us to create and manage scalable user directories, supporting secure sign-up and sign-in functionality.
- Identity Pools: Enables us to grant temporary, limited-privilege AWS credentials to users, enhancing security.

AWS Cognito offers several advantages that align with our project's requirements:

- Scalability: Automatically scales to handle user authentication for millions of users.
- Security: Implements industry-standard authentication protocols and encryption methods.
- Compliance: Helps meet various compliance requirements, crucial for handling sensitive inventory data.
- Seamless Integration: Works smoothly with other AWS services like API Gateway and Lambda.
- Cost-Effective: Pay-as-you-go pricing model aligns with our serverless architecture.

By utilizing AWS Cognito, we ensure robust, scalable, and secure user authentication and authorization for the Smart Inventory system. This choice supports our quality requirements, particularly in terms of security, scalability, and reliability. The seamless integration with our chosen AWS services enhances the

overall system architecture, contributing to maintainability and usability. Cognito's features allow us to implement role-based access control, ensuring users have appropriate permissions within the inventory management system, which is crucial for maintaining data integrity and security.

11.2 Development and Deployment: AWS Amplify

For the Smart Inventory system's development, deployment, and hosting, we've chosen AWS Amplify. This decision complements our use of Angular on the front-end and integrates seamlessly with our AWS-based back-end services.

AWS Amplify is a set of tools and services that enables rapid development and deployment of scalable full-stack applications. It offers several key features beneficial to our project:

- CLI and Libraries: Provides a command-line interface and libraries that integrate well with Angular, simplifying AWS service integration.
- Continuous Deployment: Offers Git-based workflows for continuous deployment, automating the build and deploy process.
- Hosting: Includes a fast, secure hosting service with globally distributed content delivery.

AWS Amplify aligns with our project requirements in several ways:

- Scalability: Automatically scales resources based on application demand.
- Maintainability: Provides a consistent structure for both front-end and back-end code, improving maintainability.
- Usability: Offers pre-built UI components and form builders, enhancing rapid development of user-friendly interfaces.
- Security: Integrates security best practices by default, supporting our security requirements.

By leveraging AWS Amplify, we streamline the development and deployment process for the Smart Inventory system. It provides a cohesive development experience that bridges our Angular front-end with our AWS-based back-end services. This choice supports our quality requirements, particularly in terms of scalability, maintainability, and usability. Amplify's integration capabilities ensure a smooth workflow from development to production, enhancing overall system reliability and reducing time-to-market.

11.3 Amplify Cloud Sandbox

Rapid Prototyping and Development:

- Amplify Cloud Sandbox allows developers to quickly set up and test back-end resources in a sandbox environment, accelerating development and reducing time spent on infrastructure management.

Maintainability:

- The sandbox facilitates iterative development and testing cycles within a controlled environment, promoting maintainability.
- It leverages open-source tools under the hood.

12 Architectural Diagram