# Design Doc Lab 2

## Introduction

In 50 B.C., in an innovative turn of events, the Gauls led by Asterix and Obelix have launched the first online store in all of Gaul. This groundbreaking venture, utilizing carrier pigeons for order delivery, has not only delighted the Gaulish citizens but also sparked envy among the Romans. To cater to the booming demand and ensure the store's future scalability and efficiency, Asterix has decided to revamp the store's architecture. This document outlines the proposed transition from a monolithic server to a two-tier microservices architecture, enhancing the Gaulish Toy Store's ability to serve its expanding customer base more effectively.

## Objective

The primary objective of this architectural overhaul is to accommodate growing demand by adopting a modern, scalable architecture. This will be achieved by transitioning to a two-tier design, comprising a front-end tier and a back-end tier, with microservices powering each tier.

## Proposed Architecture

### Overview

The new architecture divides the application into two main tiers:

- **Front-End Tier**: Acts as the interface between the users and the back-end services, handling user requests.
- **Back-End Tier**: Split into two distinct microservices - the Catalog Service and the Order Service, each responsible for a specific subset of business logic.

### Components

1. **Front-End Service**:
    - Serves as the single entry point for the application.
    - Communicates with users through API calls.

- Forwards user requests to the appropriate back-end service based on the action (e.g., product inquiry or order placement).

2. **Catalog Service** (Back-End):
   - Manages the product catalog, including information such as product names, stock levels, and prices.
   - Handles requests for product information and stock checks.
   - Ensures real-time accuracy of product data for both the front-end and order processing needs.
3. **Order Service** (Back-End):
   - Processes orders, including stock validation, order logging, and confirmation.
   - Interacts with the Catalog Service to update stock levels upon successful order placement.
   - Generates unique order identifiers and ensures transaction integrity.

### Communication

- **Synchronous REST API Calls**: The primary mode of communication between the front-end service and the back-end microservices, facilitating request forwarding and data retrieval.

## CatalogService

### Functionality

- Manages the product catalog.
- Supports querying and updating stock levels for products.

### Concurrency Management

- Uses a `ReentrantReadWriteLock` for concurrent read/write access to the product catalog.
- Reads (query operations) can occur concurrently, while write operations (stock updates) are exclusive to ensure data integrity.

### Dynamic Thread Pool

- Utilizes an `ExecutorService` with a cached thread pool (`Executors.newCachedThreadPool()`), allowing for dynamic allocation of threads based on demand.
- This choice ensures that the service can efficiently handle varying loads, with thread creation as needed and reuse of idle threads.

### Interface

- **Port**: 12345
- **Endpoints**:
  - **GET** `/`: Query item stock.
    - **Parameters**: `queryItemStock=<productName>`
    - **Response**: JSON object with product `name`, `price`, and `quantity`.
  - **POST** `/`: Update item stock.
    - **Parameters**: `updateItemStock=<productName>&quantity=<quantity>`
    - **Response**: A simple success message if the stock is successfully updated.

### Implementation Details

- Maintains a catalog of toys, represented as a map of product names to `Toy` objects (name, price, stock).
- Utilizes a `ReadWriteLock` to manage concurrent access to the toy catalog.
- Initializes the catalog from a predefined database file and allows updates to be persisted to the file.

## OrderService

### Functionality

- Processes orders, including validation against stock levels and order logging.

### Interaction with CatalogService

- Communicates with `CatalogService` for stock updates, ensuring that orders are fulfilled only if sufficient stock is available.

### Order ID Management

- Uses an `AtomicInteger` for generating unique order IDs in a thread-safe manner, avoiding synchronization issues.

### Dynamic Thread Pool

- Employs a cached thread pool similar to `CatalogService`, optimizing resource use under changing request volumes.

### Interface

- **Port**: 12346
- **Endpoints**:
  - **POST** `/orders`: Process a new order.
    - **Request Body**: JSON object with `productName` and `quantity`.
    - **Response**: JSON object with either an `order_number` on success or an error message if the product is out of stock.

### Implementation Details

- Communicates with `CatalogService` to update stock levels when processing orders.
- Generates order IDs using an `AtomicInteger` for thread-safe incrementing.
- Logs successful orders to a file.

## FrontendService

### Functionality

- Provides the user interface for product queries and order submissions.
- Acts as a proxy to the `CatalogService` and `OrderService`.

### Dynamic Thread Pool

- Also uses a cached thread pool for managing request handling, ensuring scalability and efficient resource utilization.

### Interface

- **Port**: 8080

- **Endpoints**:
  - **GET** `/products/<productName>`: Query product details.
    - **Response**: Forwards the response from `CatalogService`.
  - **POST** `/orders`: Place an order.
    - **Request Body**: JSON object with `productName` and `quantity`.
    - **Response**: Forwards the response from `OrderService`.

## Implementation Details

- Acts as a gateway to the `CatalogService` and `OrderService`, hiding the details of backend services from clients.
- Uses simple HTTP client functionality to forward requests and relay responses between the client and backend services.

## Scalability and Performance

- **Microservices Architecture**: Allows for independent scaling of each service based on demand, improving resource utilization and response times.

## Reliability and Fault Tolerance

- **Service Replication**: Each microservice will have multiple instances to ensure high availability and fault tolerance.