Name Ahsan Faraz

Lab Assignmnet 2

Fa22-bse-073

1. Tight Coupling – Netflix (Early Architecture)

- **Issue:** Netflix's early architecture was a monolithic design with tightly coupled components, meaning that different parts of the system were highly dependent on each other. This created bottlenecks, as a change in one area could require rebuilding and redeploying the entire system. It also made scaling difficult because there was no way to scale individual components independently. Additionally, the process of adding new features or updating services became slow and cumbersome.
- **Fix:** Netflix transitioned to a **microservices architecture**, which significantly improved scalability, flexibility, and maintainability. Here's a breakdown of the fix:
    - **Loose Coupling:** By breaking the monolithic architecture into smaller, independent services, Netflix could scale components individually. Each microservice was responsible for a specific business function, such as user management or recommendation systems.
    - **Dependency Injection:** To manage dependencies and decouple services, Netflix utilized **dependency injection**, allowing for easier management of service dependencies, making code easier to test and maintain.
    - **API Gateways:** Netflix used **API gateways** to provide a unified entry point for various services, allowing for better traffic management and reducing the complexity of handling direct communication between services.
    - **Improved Scalability:** With microservices, Netflix could deploy and scale different parts of the application independently, leading to improved performance as the user base grew.

2. Poor Modularity – eBay (Early Architecture)

- **Issue:** eBay's early system was monolithic and lacked modularity, making it difficult to maintain and scale. The codebase was large, and every change in the system required a significant amount of work. New features couldn't be easily added, and developers struggled to make updates or isolate bugs due to tight coupling between components.
- **Fix:** eBay moved to a **service-oriented architecture (SOA)** and later to **microservices**, which improved modularity. Here's how they addressed the issue:
    - **Separation of Concerns:** With SOA, eBay started dividing its system into smaller, more manageable services. Each service was responsible for a specific task, like managing auctions, user accounts, or payment processing.
    - **Independent Teams:** The transition allowed teams to work on different modules independently, without waiting for changes in other parts of the system. This increased development speed and reduced dependencies.
    - **Microservices Transition:** Eventually, eBay adopted microservices, which further modularized their system. Each microservice was designed to handle a specific domain or business logic, and this improved scalability by allowing teams to scale individual components.
    - **Faster Deployment:** The adoption of microservices led to faster deployments as new features or updates could be rolled out for individual services without affecting the entire platform.

3. God Object / God Class – Early Object-Oriented Systems (Various Projects)

- **Issue:** Early object-oriented systems often suffered from the **God Object** anti-pattern, where a single class or object handled too many responsibilities. This led to complex codebases that were hard to maintain, test, and scale. These God Objects violated the **Single Responsibility Principle (SRP)**, leading to code that was both difficult to understand and prone to errors.
- **Fix:** The solution to this problem was to refactor the God Object into smaller, more focused components that adhered to the **Single Responsibility Principle (SRP)** and **Domain-Driven Design (DDD)**. Here's how:
    - **SRP Refactoring:** Developers broke down the God Class into multiple smaller classes, each responsible for only one part of the logic. This made the codebase more modular and easier to maintain.
    - **Domain-Driven Design:** DDD encouraged the use of models that were more closely aligned with the business domain, making the codebase easier to understand and work with. This also facilitated better communication between developers and domain experts.
    - **Testability:** Smaller, focused classes were easier to test since each class had a single responsibility. This made unit testing and integration testing more efficient.

4. Lack of Scalability – Twitter (Early Architecture)

- **Issue:** In the early days, Twitter's architecture couldn't handle the volume of data and users. The database was monolithic and created **performance bottlenecks** as the system grew. The inability to scale horizontally (i.e., adding more servers) was a significant issue for Twitter.

- **Fix:** Twitter implemented several strategies to overcome scalability issues:
    - **Sharded Database Model:** Twitter transitioned to a **sharded database model**, where data was distributed across multiple databases (or shards). This allowed Twitter to balance the load across multiple machines, improving read/write performance and reducing bottlenecks.
    - **Horizontal Scaling:** Twitter adopted **horizontal scaling**, meaning they added more servers instead of upgrading a single server. This allowed them to handle more traffic as their user base grew.
    - **Caching:** Twitter implemented caching mechanisms to reduce the load on their databases, storing frequently accessed data in memory rather than querying the database every time. This improved response times significantly.
    - **Asynchronous Processing:** To avoid overloading the system with synchronous operations, Twitter implemented background processing for non-essential tasks, allowing the system to remain responsive even under heavy load.

5. Single Point of Failure (SPOF) – Dropbox (Early Infrastructure)

- **Issue:** Dropbox's early infrastructure had a **Single Point of Failure (SPOF)**, where a failure in one component could bring down the entire service. This was a serious risk, as it meant that if a server or database went down, users could lose access to their files.
- **Fix:** Dropbox addressed this issue by designing a **redundant and fault-tolerant system**. Here's how:
    - **Redundancy:** Dropbox created **replicated storage** systems, where copies of data were stored across multiple data centers. This ensured that if one data center failed, the data could still be accessed from another.
    - **Distributed Architecture:** Dropbox moved away from a single, centralized server architecture and adopted a **distributed architecture**, where components of the system could take over for each other in case of failure. This minimized downtime and ensured high availability.
    - **Failover Mechanisms:** Dropbox implemented **failover systems**, which allowed the service to automatically switch to backup systems when a failure occurred. This ensured that the system remained operational even in the event of hardware or software failures.

6. Security Vulnerabilities – Uber (Security Incident)

- **Issue:** Uber suffered a significant security breach, where personal data of millions of users was compromised due to **poor security practices**. The attack exposed the company's vulnerabilities in areas like data encryption and access control.
- **Fix:** Uber took several steps to address security vulnerabilities:
    - **Two-Factor Authentication (2FA):** Uber introduced **two-factor authentication (2FA)** for users to ensure that even if a hacker had access to a user's password, they couldn't access their account without the second factor (e.g., a phone).
    - **Data Encryption:** Uber strengthened its data encryption practices, ensuring that sensitive information, such as personal data and payment information, was

encrypted both **in transit** (while being transmitted) and **at rest** (while stored in databases).

- **OAuth:** Uber implemented **OAuth**, a protocol that provided more secure and flexible user authentication, especially for third-party integrations. This reduced the risk of unauthorized access to sensitive data.
- **Zero Trust Architecture:** Uber adopted a **Zero Trust Architecture**, meaning no one—inside or outside the company—was trusted by default. All access requests were thoroughly authenticated and authorized, reducing the risk of internal security breaches.

Report: Solving the Netflix Problem with Microservices Architecture

---

1. Introduction

In this report, we explore the solution to the challenges faced by Netflix in its early architecture. Initially, Netflix used a monolithic design, which resulted in several issues including tightly coupled components, scalability problems, and slow deployment cycles. The move to **Microservices Architecture** solved these problems by decoupling the services, improving scalability, and making the system more maintainable.

This report outlines the changes made in the architecture, focusing on the components such as **Controller**, **Model**, and **Service** layers, which have been restructured to form a more modular and scalable design. The new architecture leverages loose coupling, dependency injection, and API gateways to create a more flexible and scalable system.

---

2. Problem with the Monolithic Architecture

Netflix's early architecture used a **monolithic design** that led to several challenges:

- **Tight Coupling**: The components were tightly linked, meaning changes in one area required redeploying the entire system, slowing down updates.
- **Scalability Challenges**: Scaling individual components independently was difficult. If Netflix needed to scale the recommendation service, it had to scale the entire application, leading to resource inefficiencies.
- **Slower Feature Updates**: The interdependence of components meant adding new features required more time for testing and deployment across the entire system.
- **Bottlenecks**: Shared resources across the monolithic system created performance bottlenecks.

---

3. Transition to Microservices Architecture

To address these challenges, Netflix transitioned to a **Microservices Architecture** where different components of the system were broken down into smaller, independent services. Below is a breakdown of how the architecture was restructured into **Controller**, **Model**, and **Service** layers:

3.1 Loose Coupling

In the microservices model, each service operates independently. For Netflix, the system was split into distinct services like `UserService`, `RecommendationService`, and `TaskService`. The coupling between these services was made as loose as possible, meaning that each service can function and be updated independently without impacting others.

- **Example**:
  - `UserController`: Handles HTTP requests related to user-related operations.
  - `RecommendationService`: Handles the logic for providing recommendations.
  - `User`: Represents the user model with data attributes and business logic.

3.2 Dependency Injection

Dependency Injection (DI) was utilized to manage dependencies between services, which decouples components and simplifies testing and maintenance. In this architecture, DI was used to inject services such as `UserService` and `RecommendationService` into controllers.

- **Benefit**: This allowed easier management of the services and made it easier to test individual components like `UserController` and `RecommendationService`.

3.3 API Gateway Pattern

Netflix implemented the **API Gateway Pattern**, where a central gateway handles all incoming requests and directs them to the appropriate microservice. In this setup, `UserController` acts as the entry point for all user-related requests and forwards the request to the appropriate service.

- **Example**:

- A client sends a request to retrieve recommendations.
- The `UserController` receives the request and forwards it to `RecommendationService`.
- `RecommendationService` returns the response back to the `UserController`, which sends it back to the client.

3.4 Scalability and Maintenance

With microservices, Netflix could scale individual services independently. For instance, if the recommendation engine saw a sudden spike in requests, only `RecommendationService` would be scaled, rather than scaling the entire monolith.

- **Example**: If `RecommendationService` sees heavy usage, Netflix can scale it independently without needing to scale other services like `UserService`.

---

4. Key Concepts Implemented

4.1 Loose Coupling and Modularity

Breaking the system into distinct services like `UserController`, `User`, and `RecommendationService` ensured loose coupling. Each service is independent, modular, and focused on a single business functionality:

- `UserController`: Handles user-specific HTTP requests and coordinates service calls.
- `User`: A model representing user data and business logic.
- `RecommendationService`: Handles recommendation logic and provides recommendations to the user.

4.2 Dependency Injection for Flexibility

Spring Boot's Dependency Injection feature was used to inject services into the controller layer, making it easier to manage dependencies and test services independently.

- **Example**: `UserController` and `RecommendationService` were connected through Spring's DI mechanism, allowing for dynamic service injection during runtime. This enables flexible configuration of services and easier unit testing.

4.3 API Gateway Management

The API Gateway pattern, implemented in the `UserController`, acted as the central entry point to handle various requests. The controller listens for user-related requests and forwards them to the appropriate service (such as `RecommendationService`). This reduces direct communication between services, making the architecture cleaner and easier to manage.

4.4 Independent Scalability

One of the key advantages of microservices is the ability to scale services independently. For example, if Netflix experiences an increase in recommendation requests, only the `RecommendationService` can be scaled to handle the load, ensuring optimized use of resources.

---

5. Results and Benefits of Microservices Implementation

By adopting microservices, Netflix was able to:

- **Improve scalability**: The system could now scale individual services, like `RecommendationService`, independently based on demand, improving resource utilization.
- **Enhance maintainability**: With services like `UserController` and `RecommendationService` decoupled, developers could work on each component independently, simplifying maintenance and speeding up updates.
- **Faster feature updates**: New features or updates could be added to individual services without affecting the rest of the system.
- **Fault isolation**: A failure in one service (e.g., `RecommendationService`) does not affect others (e.g., `UserController`), minimizing downtime.
- **Optimized resources**: With independent scaling, only the services under heavy demand (e.g., `RecommendationService`) were scaled, preventing over-provisioning and reducing infrastructure costs.

---

6. Conclusion

The shift from a monolithic architecture to a microservices model allowed Netflix to address issues of scalability, maintainability, and feature delivery. By implementing **loose coupling**, **dependency injection**, and using an **API Gateway** approach, Netflix was able to create a more flexible, scalable, and maintainable system.

The adoption of microservices, with specific layers such as `UserController`, `User`, and `RecommendationService`, has enabled Netflix to scale services independently and deploy new features faster. This modular approach not only improved the user experience but also allowed for better resource management, ensuring that Netflix could continue growing without encountering the bottlenecks present in its earlier architecture.

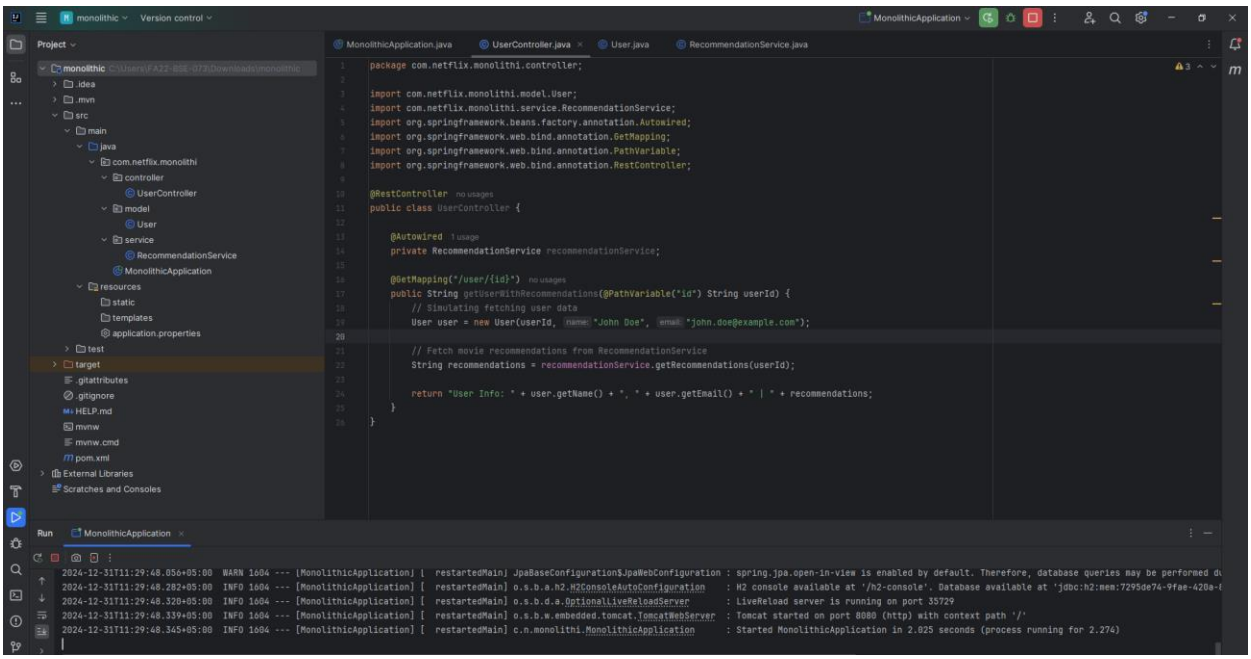
---

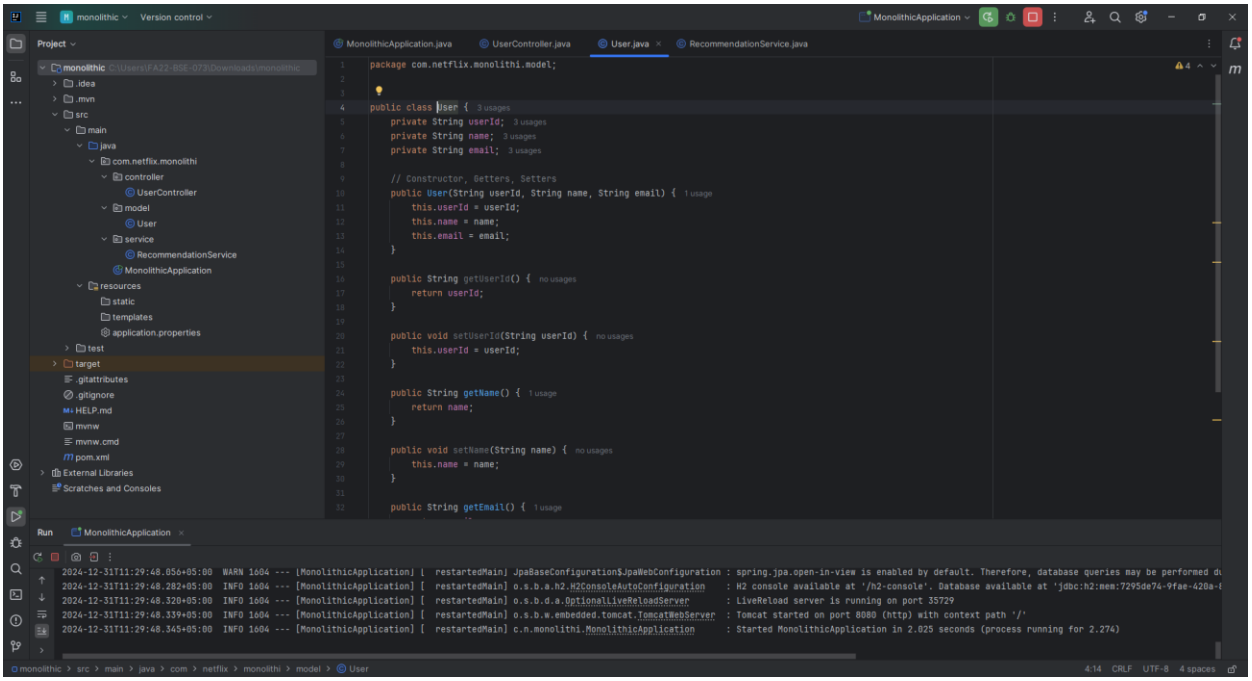
7. Further Enhancements

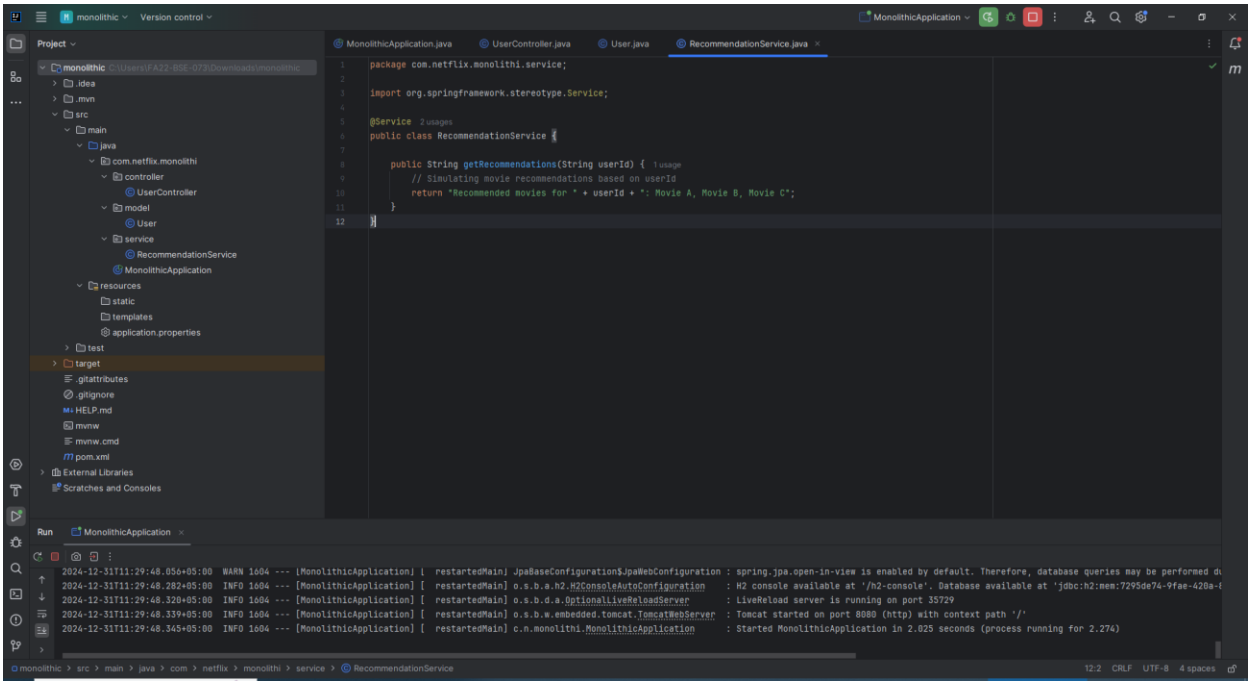
To take the system even further, Netflix could consider the following:

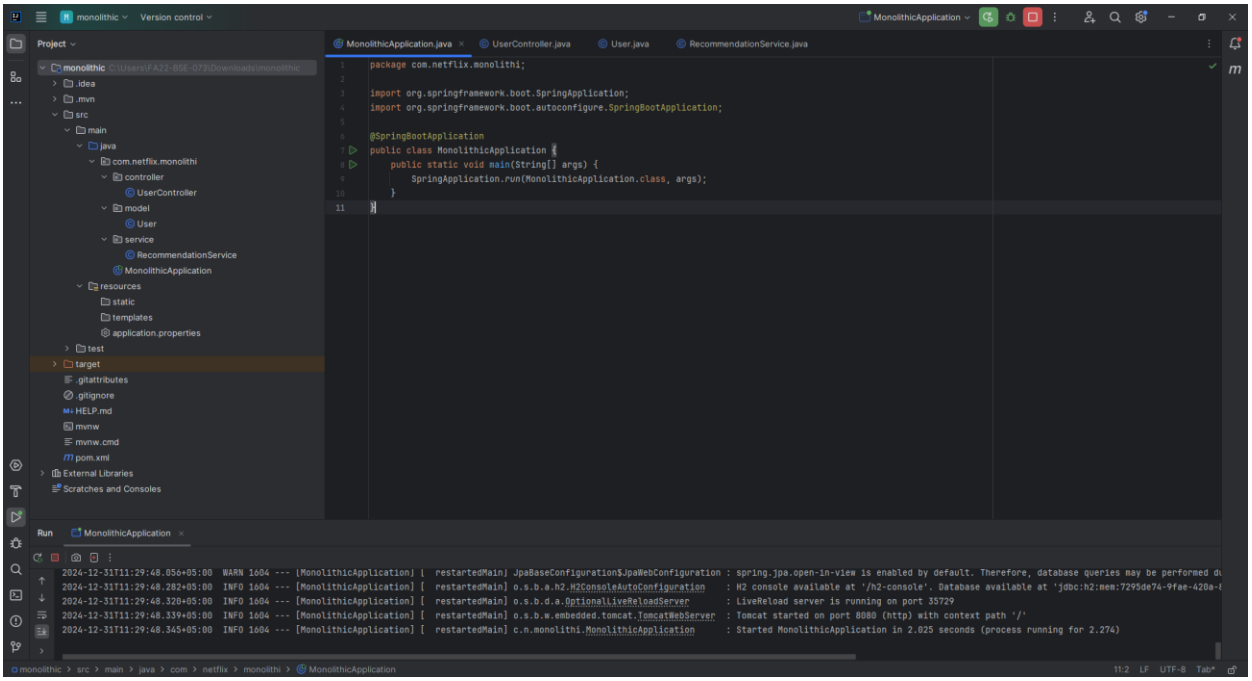
1. **Containerization and Orchestration**: Using Docker to containerize services and Kubernetes for orchestration would help in automating deployment and scaling.
2. **Advanced API Gateway**: Using specialized API Gateways like **Zuul** or **Spring Cloud Gateway** can offer enhanced traffic management, load balancing, and security features.
3. **Service Mesh**: Implementing a service mesh like **Istio** could help manage communication, security, and monitoring between microservices, improving service reliability.
4. **CI/CD Pipelines**: Automating deployments and tests using Continuous Integration and Continuous Deployment pipelines will further improve the speed of delivering new features and bug fixes.

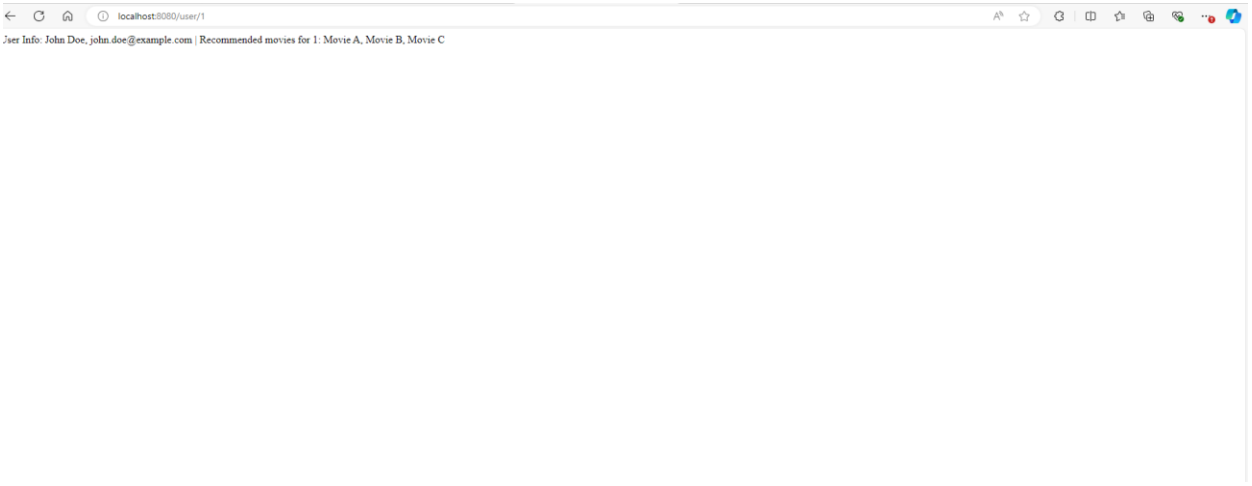
By adopting a microservices architecture with a focus on the `UserController`, `User`, and `RecommendationService` components, Netflix has successfully transformed its system into a scalable, maintainable, and resilient platform. This structure now better supports rapid growth, enabling Netflix to serve millions of users efficiently while remaining agile and responsive to changes in the market.




monolithic
Version control
MonolithicApplication
Project
MonolithicApplication.java
UserController.java
User.java
RecommendationService.java
package com.netflix.monolithi.model;
public class User {
private String userId;
private String name;
private String email;
// Constructor, Getters, Setters
public User(String userId, String name, String email) {
this.userId = userId;
this.name = name;
this.email = email;
}
public String getUserId() {
return userId;
}
public void setUserId(String userId) {
this.userId = userId;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getEmail() {
Run
MonolithicApplication
2024-12-31T11:29:48.056+05:00 WARN 1604 --- [MonolithicApplication] [restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed du
2024-12-31T11:29:48.282+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:7295de74-9fae-420a-
2024-12-31T11:29:48.320+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-12-31T11:29:48.339+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-12-31T11:29:48.345+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] c.n.monolithi.MonolithicApplication : Started MonolithicApplication in 2.025 seconds (process running for 2.274)
monolithic > src > main > java > com > netflix > monolithi > model > User
4:14 CRLF UTF-8 4 spaces


monolithic
Version control
MonolithicApplication
Project
MonolithicApplication.java
UserController.java
User.java
RecommendationService.java
package com.netflix.monolithi.service;
import org.springframework.stereotype.Service;
@Service
public class RecommendationService {
public String getRecommendations(String userId) {
// Simulating movie recommendations based on userId
return "Recommended movies for " + userId + ": Movie A, Movie B, Movie C";
}
}
Run
MonolithicApplication
2024-12-31T11:29:48.056+05:00 WARN 1604 --- [MonolithicApplication] [restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed du
2024-12-31T11:29:48.282+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:7295de74-9fae-420a-
2024-12-31T11:29:48.320+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-12-31T11:29:48.339+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-12-31T11:29:48.345+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] c.n.monolithi.MonolithicApplication : Started MonolithicApplication in 2.025 seconds (process running for 2.274)
monolithic > src > main > java > com > netflix > monolithi > service > RecommendationService
12:2 CRLF UTF-8 4 spaces


monolithic
Version control
MonolithicApplication
Project
MonolithicApplication.java
UserController.java
User.java
RecommendationService.java
monolithic C:\Users\FA22-BSE-073\Downloads\monolithic
.idea
.mvn
src
main
java
com.netflix.monolithi
controller
UserController
model
User
service
RecommendationService
MonolithicApplication
resources
static
templates
application.properties
test
target
.gitattributes
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
Scratches and Consoles
package com.netflix.monolithi;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MonolithicApplication {
public static void main(String[] args) {
SpringApplication.run(MonolithicApplication.class, args);
}
}
Run
MonolithicApplication
2024-12-31T11:29:48.056+05:00 WARN 1604 --- [MonolithicApplication] [restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed du
2024-12-31T11:29:48.282+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:7295de74-9fae-420a-
2024-12-31T11:29:48.320+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2024-12-31T11:29:48.339+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path '/'
2024-12-31T11:29:48.345+05:00 INFO 1604 --- [MonolithicApplication] [restartedMain] c.n.monolithi.MonolithicApplication : Started MonolithicApplication in 2.025 seconds (process running for 2.274)
monolithic > src > main > java > com > netflix > monolithi > MonolithicApplication
11:2 LF UTF-8 Tab*


localhost:8080/user/1
User Info: John Doe, john.doe@example.com | Recommended movies for 1: Movie A, Movie B, Movie C