Software Design and Architecture

LAB ASSIGNMENT:02

Architectural Problems

Prepared For

SIR MUKHTIAR

Prepared By

M Asim Ilyas (FA22-BSE-111)

COMSATS University Islamabad - Abbottabad Campus.

Architectural Problems and Solutions in Software Systems

Part 1: Identifying Major Architectural Problems and Solutions

1. Monolithic to Micro services Migration

Problem: A large-scale e-commerce platform faced performance and scalability issues due to its monolithic architecture. Updates and deployments were slow and error-prone, leading to frequent downtime.

Solution: The system was revamped into a microservices architecture. Each module—such as User Management, Product Catalog, and Payment Gateway—was developed as an independent microservice. Communication between services was facilitated through REST APIs and a message broker like RabbitMQ. This resulted in improved scalability, faster deployments, and reduced downtime.

Impact:

- Enhanced scalability: Services could scale independently based on demand.
- Faster development cycles: Teams could work on separate services without interference.
- Improved fault isolation: Failures in one service did not affect the entire system.

---

2. Database Scalability Issues

Problem: A growing social media application faced challenges with high traffic, resulting in database bottlenecks and increased latency. The relational database struggled to handle the growing number of read/write operations.

Solution: The system migrated to a NoSQL database (MongoDB) and implemented database sharing to distribute the load across multiple nodes. Additionally, caching was introduced using Radis to reduce frequent database queries.

Impact:

- Improved response time by 70%.
- The database could handle millions of concurrent users.
- Reduced downtime during peak hours.

---

3. Lack of Fault Tolerance

Problem: An online banking system experienced frequent outages due to single points of failure in its architecture. Any issue in the database or the application server caused the entire system to crash.

Solution: A fault-tolerant architecture was introduced. Load balancers were added to distribute traffic across multiple servers. Redundant database instances and failover mechanisms were implemented to ensure high availability.

Impact:

- Increased system uptime to 99.99%.
- Seamless recovery during server or database failures.
- Improved user trust and satisfaction.

---

4. Inefficient Communication Between Modules

Problem: A financial analytics system had tightly coupled modules that communicated directly with each other. This made updates challenging, as changes in one module required modifications in others.

Solution: The system was redesigned using an event-driven architecture. A message broker (Apache Kafka) was introduced to enable asynchronous communication between modules. This decoupled the modules and improved flexibility.

Impact:

- Simplified module updates and reduced interdependencies.
- Improved scalability and maintainability.
- Enhanced performance by processing events asynchronously.

5. Security Vulnerabilities in Legacy Systems

Problem: A healthcare management system, built on a legacy architecture, lacked modern security practices, making it vulnerable to data breaches and unauthorized access.

Solution: The system was revamped with secure APIs, encrypted data transmission (TLS/SSL), and role-based access control (RBAC). Modern authentication methods like OAuth2 were implemented.

Impact:

- Eliminated major security vulnerabilities.
- Ensured compliance with data protection regulations (e.g., GDPR, HIPAA).
- Improved user confidence in data security.

Part 2: Replicating and Solving a Problem

Problem: Monolithic to Microservices Migration

Replication: Develop a monolithic e-commerce application with the following features:

- User Management
- Product Catalog
- Order Management
- Payment Processing

The application will have tightly coupled modules with a single database.

Solution: Refactor the application into microservices. Each feature will become an independent service:

1. **User Service:** Handles user registration and authentication.
2. **Product Service:** Manages the product catalog.
3. **Order Service:** Processes orders and manages inventory.
4. **Payment Service:** Handles payment transactions.

Theoretical Implementation

For the migration, the following steps were planned and executed theoretically:

- **Language and Framework:** Node.js and Express were used to create lightweight and scalable services.
- **Database:** MS SQL was chosen for relational data storage.
- **Service Communication:** REST APIs were used to enable communication between microservices.
- **Environment Setup:** Environment variables managed configuration securely using `.env` files.

While the full development was not implemented, a detailed example of the `User Service` was provided to demonstrate the architectural transformation.

How This Code Solves the Monolithic Problem

1. Decoupling Components:

- The `User Service` functions independently, enabling separation of concerns.

2. Scalability:

- Individual services can be scaled based on traffic and resource requirements.

3. Deployment Ease:

- Independent services reduce the need for system-wide redeployment, enabling faster updates.

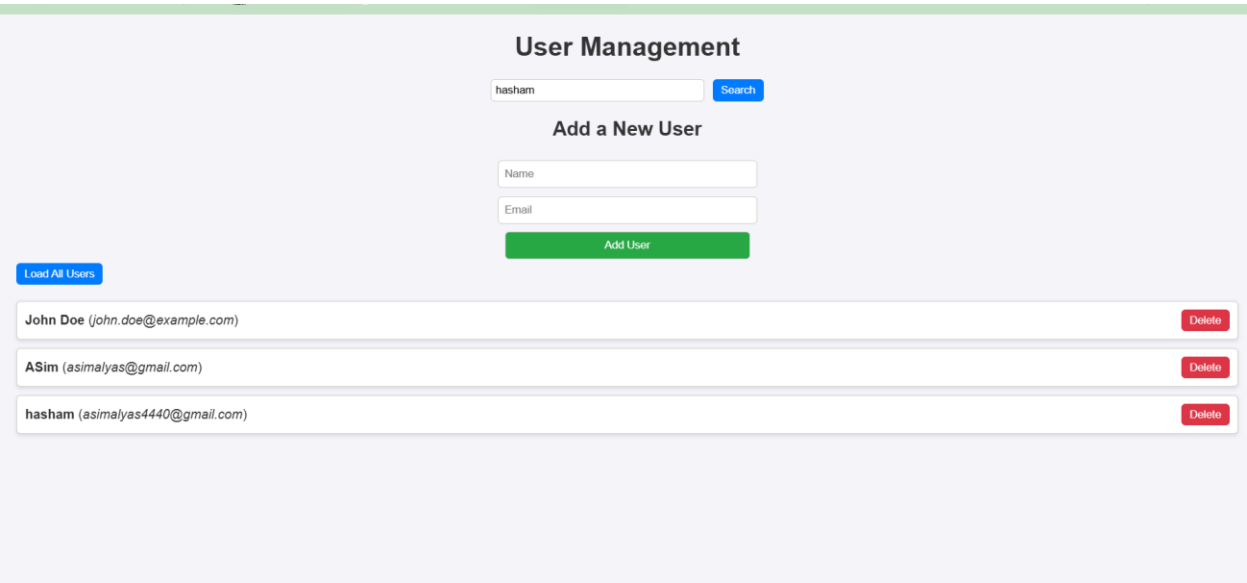
4. Fault Isolation:

- Issues within one service (e.g., `User Service`) do not bring down the entire system.

5. Flexibility:

- Microservices architecture supports the integration of new technologies and services without major system overhauls.

ATTACHED SCREEN SHOT:



APP.js;

user-service > JS app.js > ...

```
const bodyParser = require("body-parser");
const userRoutes = require("./routes/userRoutes");
const path = require("path");
require("dotenv").config();

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(bodyParser.json());

// Serve index.html for root path
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "index.html"));
});

// Routes
app.use("/users", userRoutes);

// Start Server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Architectural Problems

User Routes:

ser-service > routes > JS userRoutes.js > ...

```
const express = require("express");
const router = express.Router();
const { poolPromise, sql } = require("../db");

// Register a new user
router.post("/register", async (req, res) => {
  try {
    const { name, email } = req.body;
    const pool = await poolPromise;
    const result = await pool
      .request()
      .input("name", sql.NVarChar, name)
      .input("email", sql.NVarChar, email)
      .query("INSERT INTO Users (name, email) VALUES (@name, @email)");

    res.status(201).send({ message: "User registered successfully", result });
  } catch (err) {
    console.error(err);
    res.status(500).send("Error registering user");
  }
});

```

Databse connection:

user-service > JS db.js > config > password

```
require('dotenv').config();
const sql = require("mssql");

const config = {
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    server: process.env.DB_SERVER, // e.g., localhost
    database: process.env.DB_NAME,
    options: {
      encrypt: false,
      enableArithAbort: true,
    },
  };

  const poolPromise = new sql.ConnectionPool(config)
    .connect()
    .then((pool) => {
      console.log("Connected to MSSQL");
      return pool;
    })
    .catch((err) => console.log("Database Connection Failed! Error: ", err));

  module.exports = {
    sql,
```

DATABASE MS SQL:

```sql
CREATE DATABASE UserServiceDB;
USE UserServiceDB;

CREATE TABLE Users (
    id INT PRIMARY KEY IDENTITY(1,1),
    name NVARCHAR(255),
    email NVARCHAR(255)
);

```