**Logo

Description automatically generated**

Course: **Guided Research I**

Title: **Scalability experiment of microservice architecture on an online bookstore application**

**Weekly Progress Report 4**

Student: Tural Mehtiyev

Next Steps:  
  
1. Fixing some bugs in the functionality of Order Management Service.

2. To Continue Performance Testing in Apache Jmeter and record the results.

3. Conduct Statistical Analysis based on the result data and prepare visualizations.

4. Work more on the advancement of the architecture diagram for its presentability in the final report.

This repository contains various resources used for our project. Here's a quick rundown of the primary directories and their contents:

* **data**: This directory contains all the data sets used for the experiments and analysis.
* **papers**: This directory holds all the academic papers that have been reviewed and analyzed for the project.
* **presentations**: Any presentations related to the project, including both planning and results, are stored here.
* **reports**: This directory is for all the detailed reports generated throughout the project. Each report is named according to the date it was created, following the format report\_DATE.pdf.
* **reviews**: This directory contains the reviews of research proposals submitted by peers. Each review is named according to the date it was created and the submitter's name, following the format Review\_Date\_Submitter\_Name.pdf.

**Order Management Microservice**

This microservice manages the processing of customer orders.

**API Documentation**

**GET /orders:**

Get all orders.

**Response**

200 OK: Successful operation. Returns a list of orders.

@app.get("/orders")

def get\_all\_orders():

cur = conn.cursor()

cur.execute("SELECT \* FROM orders")

orders = cur.fetchall()

if not orders:

return {"message": "No orders were found.", "orders": []}

return {"orders": orders}

**GET /orders/{customer\_id}:**

Fetch all orders for the specified customer.

**Parameters**

customer\_id: ID of the customer for whom to retrieve the orders.

**Response**

200 OK: Successful operation. Returns a list of orders.

404 Not Found: No orders found for the specified customer.

@app.get("/orders/{customer\_id}")

def get\_orders(customer\_id: int):

cur.execute("SELECT \* FROM orders WHERE customer\_id = %s", (customer\_id,))

orders = cur.fetchall()

if not orders:

raise HTTPException(status\_code=status.HTTP\_404\_NOT\_FOUND,

detail=f"Orders for customer with id: {customer\_id} were not found")

return {"orders": orders}

**GET /orders/{order\_id}:**

Returns the order with the specified ID.

**Parameters**

order\_id: ID of the order to retrieve.

**Response**

200 OK: Successful operation. Returns the order with the specified ID.

404 Not Found: Order with the specified ID not found.

Configuring the Database Connection of the Services

Last week, I undertook the critical task of configuring the database connections for the various microservices. This was essential to ensure each service can seamlessly interact with the database and perform its designated tasks. It involved setting up the required parameters, such as the database URL, credentials, and the connection pool properties. By properly setting up the connections, I ensured optimal communication between our microservices and the databases, which is pivotal for the performance and scalability of our online bookstore application.

Deploying the Databases in Amazon RDS

I successfully deployed our databases in Amazon Relational Database Service (RDS), a highly scalable and secure cloud database service. This was done to leverage the benefits of managed service, such as automatic software patching, backups, and disaster recovery capabilities, among others. It also allows us to scale the database instances in response to our application's demand easily. I used the AWS Management Console and AWS CLI to automate the deployment process, thereby reducing the margin for error.

Deploying the Services in Deta

Once the databases were set, I proceeded to deploy our microservices on Deta, a cloud platform ideal for deploying and scaling microservices. Given Deta's benefits in terms of automatic scaling, zero server management, and straightforward deployment process, it was a logical choice. Each microservice was deployed separately, ensuring that they function as isolated units, thus abiding by the principles of microservice architecture. I utilized Deta's CLI for the deployment and monitored the process until each service was up and running.

Testing the Functionality of the Services via Postman

Finally, after successfully deploying the databases and services, I tested each microservice using Postman, a popular API client. Postman was used to send requests to the services and assess their responses. This allowed me to confirm that each service was functioning as expected, and the database connections were correctly configured. It also helped me identify any possible performance issues or bottlenecks that we need to address in the subsequent stages of our performance testing process.

This concluded the activities for last week. I look forward to the next steps in the project, which will include comprehensive performance testing, stress testing, and optimization of our online bookstore's microservice architecture.  
  
My main progress in the last week was about deployment of the services and testing the functionality.

Performance testing is a type of software testing that focuses on how a system performs under a particular workload. It's not just about finding bugs or verifying that the application works correctly, like functional testing. Instead, it's about understanding the scalability of the application and identifying any bottlenecks that could impact its performance.

Here are some of the key characteristics of performance testing:

It tests the speed and effectiveness of the system under load: Performance testing measures how quickly the system responds, its robustness, reliability, and scalability under a particular workload. It can also help you identify the maximum operating capacity of an application.

It focuses on non-functional requirements: Unlike functional testing, which verifies whether the system works according to the specified functional requirements, performance testing validates the non-functional requirements such as system's speed, stability, and scalability.

It simulates concurrent users: Performance testing often involves simulating multiple users accessing the system at the same time, which isn't typically a concern in functional testing.

There are different types of performance testing, including:

Load Testing: It checks how the system behaves when multiple users access it concurrently. It helps to identify the maximum load the system can handle.

Stress Testing: It involves testing the system under extreme loads to identify the breaking point or the limit at which the system fails.

Endurance Testing: It is done to make sure the software can handle the expected load over a long period of time.

Spike Testing: This type tests the software’s reaction to sudden large spikes in the load generated by users.

Volume Testing: This is done to analyze the system performance by increasing the volume of data in the database.

The goal of performance testing is to identify any performance problems, establish a benchmark for future testing, and ensure that the system meets performance expectations.

Functional testing and performance testing are both critical parts of the software development lifecycle. While functional testing ensures that the system behaves as expected, performance testing ensures that the system can handle the expected load effectively and efficiently.