Report: Personalized Health Tracker Application

## Introduction

The Personalized Health Tracker is a microservices-based application designed to provide users with a convenient way to monitor their health. Users can calculate their BMI by entering their height and weight and receive personalized exercise recommendations. Additionally, the application allows users to create a profile with personal details, enabling the automatic calculation of BMI and tailored exercise recommendations. The system emphasizes scalability, fault tolerance, and independent deployment to provide a robust and efficient solution.

## Problem Statement

#### Objective:

* Design and implement a microservices-based application with four distinct services.
* Deploy the application on a Kubernetes cluster, highlighting independent deployment, scalability, and fault tolerance.
* Demonstrate the use of Docker to simplify and automate the deployment process.

#### Service formulation:

* Microservices architecture with four distinct services:
  + User Profile Service
  + Health Prediction Service
  + Exercise Recommendation Service
  + Frontend Streamlit

Each service was formulated to perform a specific function, ensuring that the system remains modular and maintainable.

* Key characteristics
  + **Scalability** using Horizontal Pod Autoscaling (HPA).
  + **Fault tolerance** through Kubernetes’ liveness and readiness probes.
* Docker was used to simplify and automate the deployment process. Each service was containerized using Docker, ensuring that the application can be deployed consistently across different environments.
* Deployment on Kubernetes Cluster
  + Independent Deployment: Each service was deployed independently, allowing for flexible scaling and management.
  + Autoscaling: Horizontal pod autoscaling was implemented to ensure that the application can handle changes in workload.
  + Fault Tolerance: Liveness and readiness probes were implemented in each service to detect and restart faulty containers.

## Microservice Architecture over Monolithic

The microservices architecture was chosen over a monolithic architecture for the following reasons:

* Independent Deployment: Each service operates independently, allowing updates or bug fixes without impacting other components. This is especially important in an application where changes to one feature (e.g., BMI calculation logic) should not disrupt the user profile or exercise recommendation functionalities.
* Scalability: Services can scale individually based on demand, reducing resource wastage. For instance, the *exercise\_recommendation\_service* might require more resources during peak usage compared to the *user\_profile\_service.*
* Technology Diversity: Each service can use different programming languages, frameworks, or tools based on specific requirements. This flexibility allows the team to use Flask for backend APIs while leveraging Streamlit for the frontend.
* Ease of Maintenance: Smaller, focused codebases for each service simplify development and troubleshooting. Debugging an isolated issue in the *user\_profile\_service* is faster and less error-prone than dealing with an interconnected monolith.

## Microservices Overview

##### User Profile Service

* **Purpose:** Manages user profiles, including name, age, gender, height, and weight.
* **Functionality:** Stores user details for easy retrieval and communication with other services.
* **Technology:** Flask-based API.

##### Health Prediction Service (BMI Service)

* **Purpose:** Calculates BMI based on user-provided height and weight.
* **Functionality:**
  + BMI = Weight (kg) / [Height (m)]^2.
  + Provides BMI category (e.g., underweight, normal, overweight).
  + Offers this category to other services for further processing, like personalized exercise recommendations.
* **Technology:** Flask-based API.

##### Exercise Recommendation Service

* **Purpose:** Recommends exercise regimes based on BMI.
* **Functionality:** Provides tailored suggestions to help users achieve or maintain a healthy BMI. These personalized recommendations are generated using the service's own dedicated database, which evaluates user-specific details, including the closeness of their BMI to target categories.
* **Technology:** Flask-based API.

##### Frontend Streamlit Service:

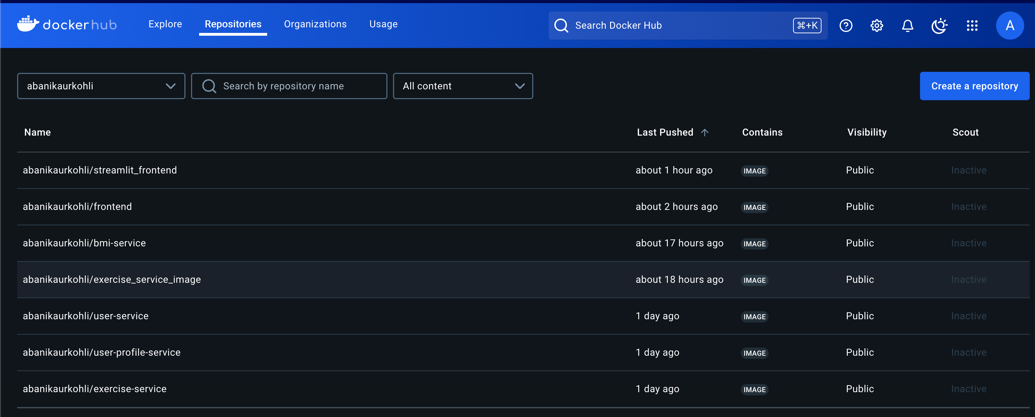
* + **Purpose:** Provides a user-friendly interface for interacting with the application.
  + **Functionality:**
    - Input forms for BMI calculation.
    - Profile creation and exercise regime visualization.
  + **Technology:** Streamlit framework.

## Scalability and Fault Tolerance

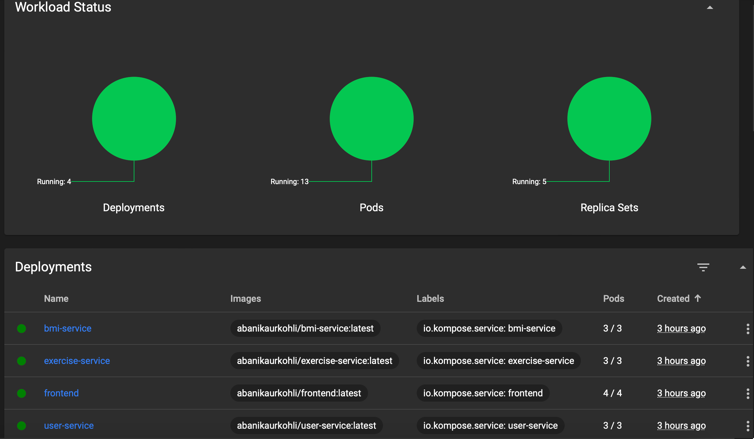
1. **Horizontal Pod Autoscaling (HPA):**
   * Each microservice is configured to scale automatically based on CPU and memory usage.
   * Kubernetes monitors resource utilization and adjusts the number of pods as needed to handle varying loads efficiently.
2. **Liveness and Readiness Probes:**
   * Liveness probes ensure that the application is running and responsive. If a service becomes unresponsive, Kubernetes restarts the pod.
   * Readiness probes verify that the service is ready to handle requests. Pods failing this check are removed from the load balancer until they recover.

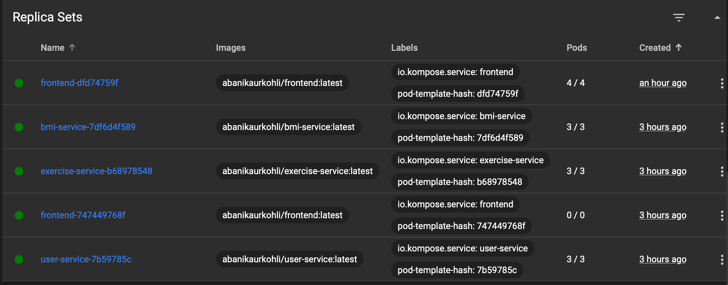


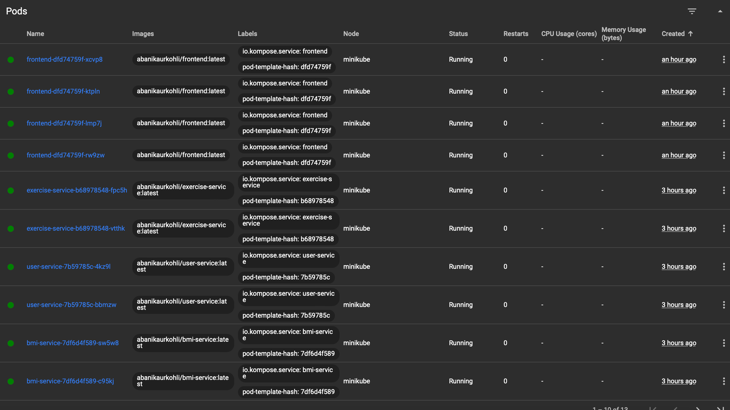
1. **Deployment Strategy**
2. **Docker:**
   * Each service is containerized using Docker, ensuring consistent environments across development, testing, and production.



1. **Kubernetes:**
   * **Deployment Files:** Each service has a separate Kubernetes deployment YAML file, specifying replicas, container images, and resource limits.
   * **Network:** All services communicate through an internal Kubernetes network.





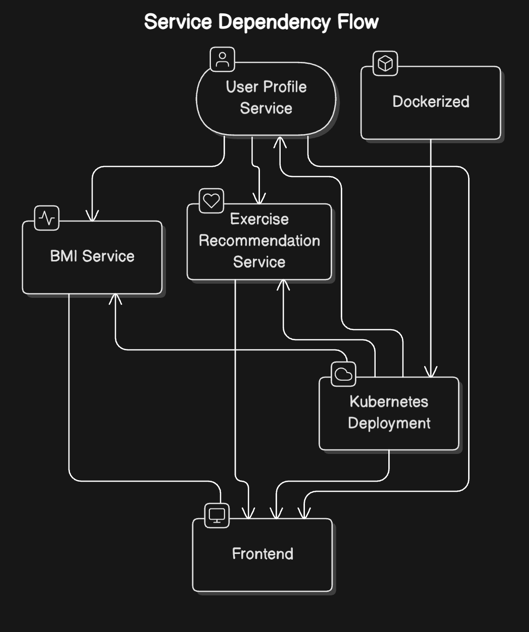


1. **Load Balancing:**
   * Kubernetes ensures even distribution of traffic among service replicas using a built-in load balancer.

**System Dependency Flow**

The diagram outlines the interaction between the four services:

* User Profile Service interacts with the BMI and Exercise Recommendation Services.
* Frontend\_Streamlit integrates all services for a seamless user experience.
* Kubernetes manages deployment, scaling, and fault tolerance.



**Challenges and Solutions**

* **Challenge:** Managing inter-service communication. **Solution:** Kubernetes’ internal DNS and service discovery simplified communication.
* **Challenge:** Ensuring fault tolerance. **Solution:** Implemented liveness and readiness probes.

**Conclusion**

The Personalized Health Tracker application demonstrates the power of microservices architecture in building scalable, fault-tolerant, and independently deployable systems. The use of Docker and Kubernetes streamlines deployment and management, ensuring robust performance under varying workloads. By leveraging modern technologies, this project highlights the advantages of modular design and efficient resource utilization.