CT70A9700 Cloud Services and Infrastructure

Technical Project Documentation

FITNICS:

Cloud Based Fitness Management Platform

Group Z

Beenish Nazam, beenish.nazam@student.lut.fi

Sumaiya Antara, sumaiya.antara@student.lut.fi

Sindhuja Chandrasekaran, sindhuja.chandrasekaran@student.lut.fi

Zarak Iftikhar, zarak.iftikhar@student.lut.fi

Sheraz Anwar, sheraz.anwar@student.lut.fi

Table of Contents

1. Introduction

It was planned at first that to create something that could genuinely help people. So after multiple brainstorming sessions and discussion, the idea of Fitnics was shared. The goal was simple but powerful. A simple but cloud native fitness companion app solely focused on physical fitness. This app target fitness with workouts, meals management and others such as water intake that could be tracked with analytics. These solutions were all brought up to one place. Fitnics was the answer to that gap. A platform was envisioned where users could manager their workouts, nutrition, hydration, fitness tracking and even detailed health analytics all shown in a unified experience. This project was not just a useful product but a technical challenge. It was an opportunity for us to understand the cloud native implication principles, containerization, exploring microservices, setup CI/CD pipelines with cloud environment deployment. Monitoring and logging integration with robust security measures was also applied. This project was handled by our team like a real world professional pushing it beyond the academic goals. This documentation captures the entire journey of building Fitnics from conception to planning architecture design, development and deployment.

Fully functional cloud application website: https://app.fitnics.space

Project GitHub Repository: https://github.com/sindhuja2002/fitnics

2. Project Objectives and Scope

The official journey of moving forward with building Fitnics was the first exercise that was completed as a team was defining the objectives and scopes clearly. It was known that setting of strong foundation with well defined goals would help keep the development focused. This helped the project delivered completely with a polished solution by the timeline. The primary objective was building a fully cloud native fitness companion. It offered an integrated experience across multiple dimensions of health management. It was envisioned as true health partner with handling all the fitness objectives with visualized and real time analytics of their journey.

This vision helped us to define several main technical objectives:

- Building a modular and scalable architecture with each component of the system. This should be independent but integrated by allowing for future scaling or modification without affecting the entire system.

- Using containerization and orchestration to build real world cloud native skills. Docker containers and Docker Swarm for service deployment and orchestration was used.
- Protecting user data was a top priority so a secure and token based authentication system was implemented. JWT (JSON Web Token) for stateless authentication was brought up.
- CI/CD workflows were practiced for automated build, test and deployment process using GitHub Actions with a professional development pipeline.
- Integrating system monitoring and centralized logging for a real cloud system was needed for observability. So Prometheus for metrics, Grafana for dashboards and Loki for centralized log management was integrated.
- For a better user experience, the application should be intuitive, responsive and visually engaging to encourage users to interact with it daily.

The functional scope of Fitnics were also defined carefully:

- Users should be able to register and log in to a secure personal account.
- Users should be able to explore a rich workout database, search exercises and plan their own workouts.
- The platform should offer nutrition checker where users can look food items with their nutritional value.
- Users should be able to calculate their Basal Metabolic Rate (BMR) to understand their caloric needs.
- A Meal Plan feature should allow users to plan their meals for each day.
- Water intake management should allow users to log hydration habits easily and follow their daily progress.
- A fitness tracker should have manual entry of fitness metrics like steps, calories burned, workout time, heart rate, weight and sleep.
- Users should have access to a powerful analytics dashboard to visualize trends over time.
- Notifications should support reminders for hydration, meals and workouts based on user preferences.

Each of these functional requirements was tied to a technical implementation goal. That's why a balanced platform was built where features and architecture supported each other. The final scope also included full deployment to a cloud hosting provider. It would make Fitnics live on the internet reachable from any device with service over a secure HTTPS connection. A clear objective and boundaries were set at the start. A clear map was followed

throughout development helping us stay focused while measuring the progress. Eventually able to deliver a complete and professional cloud native application.

3. Technology Stack

Choosing the right technology stack was a first decision which shaped the rest of the project. The tools needed were reliable, widely adopted and cloud friendly to the type of modular architecture planned. A significant time was spent researching different options for each layer of the stack like frontend, backend, microservices, database, orchestration, CI/CD and monitoring. The following technologies were finalized after careful deliberation:

Component	**Technology Used**
Frontend	ReactJS
Backend API	ExpressJS (Node.js)
Database	MongoDB
Microservice (Analytics)	FastAPI (Python)
Authentication	JWT (JSON Web Tokens)
Proxy Server	Traefik
Containerization	Docker
Orchestration	Docker Swarm
CI/CD Pipeline	GitHub Actions
Monitoring and Logging	Prometheus, Grafana, Loki
Cloud Hosting Provider	Digital Ocean

Each technology was selected not only for its features but also for larger cloud native philosophy. ReactJS was chosen for its component based structure, handling stateful frontends and support for fast user interactions. ExpressJS offered the lightweight backend server needed for RESTful API creation without unnecessary complications. MongoDB gave a flexible NoSQL document model which is used for storing different types of fitness related user data without rigid schemas. FastAPI native support for asynchronous operations was a good choice for our microservice needs. This is specially for handling analytics without slowing down the core backend. JWTs allowed us to implement scalable user authentication for tokens being validated at each service layer.

Traefik was used in the dynamic environment of Docker Swarm for traffic management and SSL/TLS certificate. Services can appear and disappear as needed. Docker is to containerize

every service cleanly while Swarm is to orchestrate and scale services without complex setup. Whenever changes were pushed to the main branch, GitHub Actions streamlined our CI/CD workflows, testing and code deployment. Prometheus collected system metrics for observability, Grafana visualized them and Loki added logs into the system. Digital Ocean was chosen for its developer friendly cloud environment with setting up droplets easily and built in monitoring features.

4. System Overview and Design Approach

Fitnics needed to be seamless to user while being modular, secure and scalable in its internal structure. Every technical choice made and layered architect was aimed to achieve this goal. The system consists of three major service layers at highest level that are frontend, backend API and analytics microservice. Each layer operates independently in close communication with others through secured APIs and internal networking. The frontend is a dynamic ReactJS application which is designed as a single page app (SPA). It communicates with the backend through RESTful APIs secured by JWT authentication. The frontend handles all user interactions like login, registration, profile updates, workouts, fitness management and viewing analytics. The backend API is built using ExpressJS. It acts as the core server which handles all business logic, user management, security enforcement and CRUD operations against MongoDB database. Also forwards heavy computational requests to the analytics microservice. It shows a set of clean RESTful endpoints and on every secured request responsible for token validation. The backend also hands in routing and internal networking within Docker Swarm by managing communication with Traefik.

The FastAPI microservice is a layer which is responsible for all advanced analytics and heavy data aggregation tasks. We kept the main backend lightweight and responsive by offloading intensive computations in this microservice. The microservice is called only when necessary for processing requests like generating user health summaries. The MongoDB database is the persistent storage layer for the platform. It has database for user profiles, workout logs, diet profiles, meal plans, water intake logs, fitness tracker metrics and notifications. Traefik is at the edge which routes all incoming HTTP/HTTPS traffic. It provides SSL/TLS certificates automatically while routing requests to the correct service based on path rules. It also monitors the health of backend services when necessary for smooth failover. The observability layer of system includes Prometheus for scraping service metrics, Grafana for displaying performance dashboards and Loki for logs collection from all containers. This gives us deep insight into the system's health in real time so that active responses to issues. Using Docker all services are containerized for environment consistency and easy deployment. Docker Swarm orchestrates these containers for load

balancing management, service discovery and scaling through Digital Ocean droplets. Continuous deployment is automated through GitHub Actions.

A new Docker images are created every time code is pushed to the main branch. This is tested, pushed to registry and deployed with zero downtime while updating in Docker Swarm.

5. UML Diagrams

5.1 Component Diagram

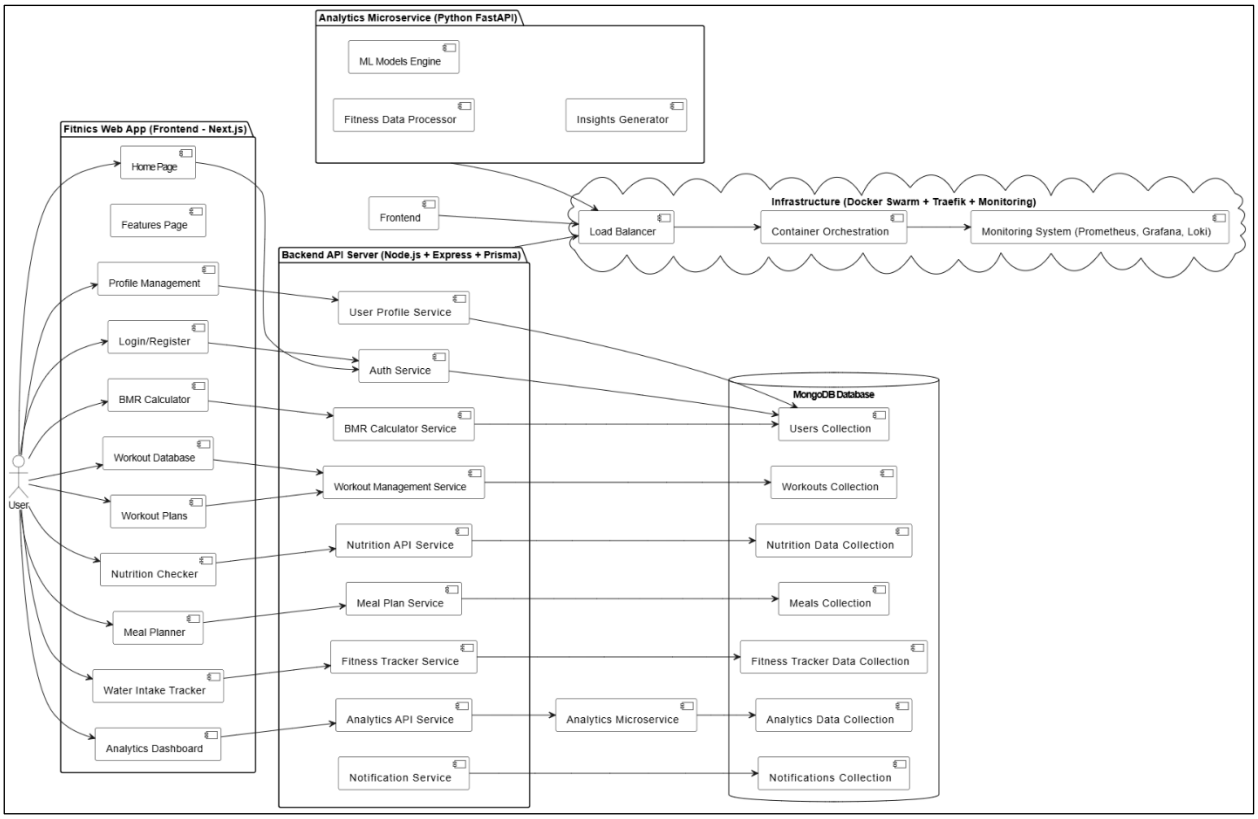

Figure 1: Component Diagram

This component diagram shows way Fitnics system is structured by using different services. It also shows how the services interact with each other. It includes the frontend (Next.js), backend API (Node.js + Express), a Python-based analytics microservice and a MongoDB database. All components are connected through a cloud infrastructure using Docker Swarm, Traefik and monitoring tools like Prometheus, Grafana and Loki.

5.2 Deployment Diagram

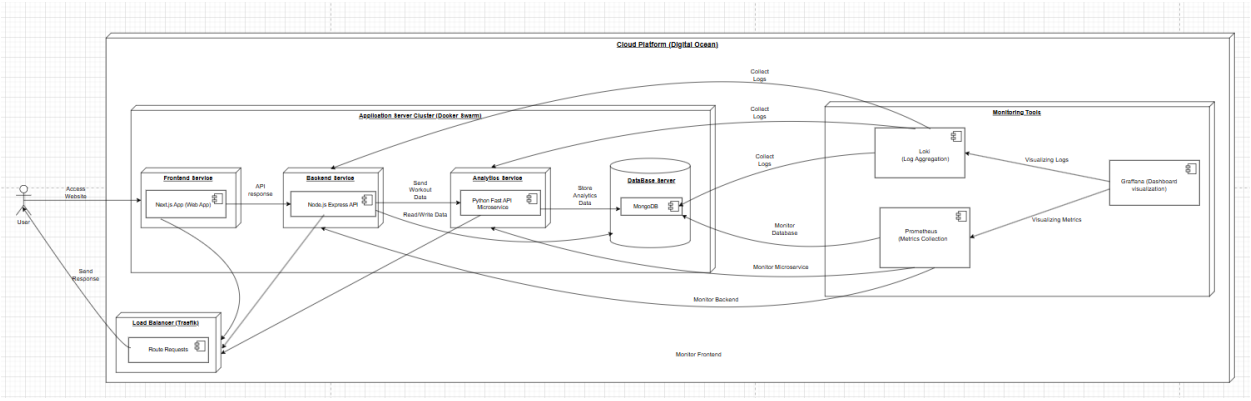

Figure 2: Deployment Diagram

This deployment diagram shows how the Fitnics system is hosted on cloud using DigitalOcean. It includes frontend, backend, analytics microservice and a MongoDB database managed through Docker containers. Monitoring tools like Prometheus, Loki and Grafana are used to track system health, logs and performance.

5.3 Data Flow Diagram

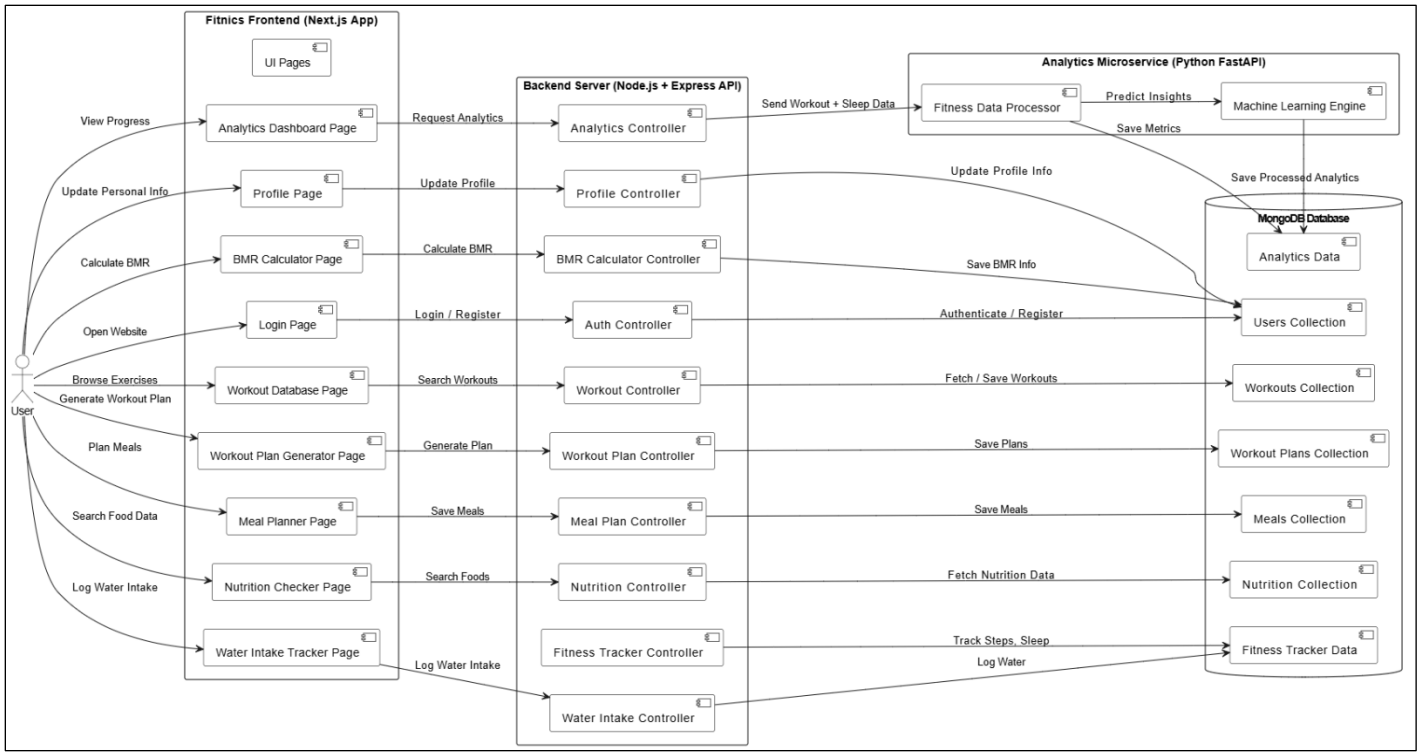

Figure 3: Data Flow Diagram

This data flow diagram shows how user actions move through the Fitnics system from the frontend pages to backend controllers and finally to the database. It illustrates how requests like logging workouts, planning meals and calculating BMR are processed and stored. The diagram also highlights how analytics data is sent to a microservice for processing and insights.

5.4 Use Case Diagram

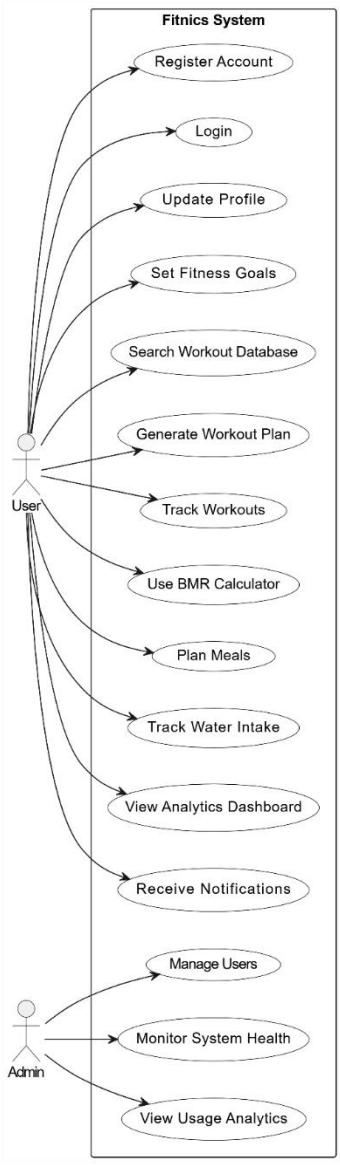

Figure 4: Use Case Diagram

This use case diagram shows how two types of users that are regular users and admins interact with Fitnics system. It shows user actions like registering, planning meals, tracking workouts and viewing analytics while admins have access to system monitoring and user management. The diagram provides a clear overview of all major features and who can use them.

6. Frontend Application

The user experience for designing the frontend for Fitnics defines if people experience the platform engaging or not. Users were given the opportunity to be in control of their fitness journey from the very first click. Several major design decisions were made on how the frontend should be structured and how users would interact with it. ReactJS was chosen as frontend framework due to its flexibility and better support for building dynamic single page applications (SPA). It helps to break down the platform into smaller reusable components. Each component is responsible for a specific piece of functionality. This modular approach made the maintenance of codebase easier with consistent user interface delivery on different sections of the app. The core principle is the simplicity for users to find quickly what the are looking for without confusing navigation. The navigation bar at the top of the screen is always present with access to Home, Features, Workout Database, Workout Plans, Nutrition Checker, BMR Calculator, About, Login and Register. Once user logs in, the navigation bar dynamically show their name with quick access to profile settings and the logout option.

The Home page introduces an overview of its capabilities for first time visitors. It is designed to be informative offering with enough details to encourage exploration. The Features page highlights the core strengths of Fitnics. It describes tools like Workout Database, Nutrition Checker, BMR Calculator, Meal Planner and Water Intake Log to visually introduce features in sections. Inside the Workout Database, users can search for exercises based on muscle groups. This allows users to quickly find exercises for Back, Chest, Cardio, Lower Arms, Lower Legs, Upper Arms, Upper Legs, Shoulders, Waist, and Neck. Each search result is presented cleanly with exercise names and available details in a friendly layout. The Workout Plans section generates structured training routines for selected muscle groups. The Nutrition Checker offers a simple search field where users can type in food items to get nutritional data. This removes the need for users to juggle multiple websites to check food macros. The BMR Calculator asks users for only three fields that are Age, Weight and Height and display their calculated Basal Metabolic Rate.

Form validation and feedback are treated with care throughout the platform. If a user enters incorrect data or misses a required field, they are gently alerted with error messages. When forms are submitted successfully, users receive friendly confirmations. The most important aspects of our frontend strategy was responsiveness. We built Fitnics to work well on desktops, tablets and smartphones. Flexbox and responsive CSS made the layouts to adjust to different screen sizes without losing functionality. To keep fast load times, we engaged code splitting with lazy loading of non critical components and optimizing image assets. This

kept the initial bundle size small where users start interacting with the platform quickly even on slower internet connections.

Accessibility was a focus area such that text contrasts met accessibility standards. Navigation was also keyboard friendly and input fields had clear labels. While there is always room to accessibility support, these foundations were important to make Fitnics more inclusive from the starting day. During the designing of frontend experience for Fitnics, we believed that users should never feel lost or frustrated instead every interaction should feel supportive. This led to an application which is not just functional but enjoyable.

7. Backend Architecture

Building the backend for Fitnics was both a challenge and an opportunity. Every frontend click would ultimately be processed through our backend API. ExpressJS was selected as backend framework as it is lightweight, flexible and ideal for building RESTful APIs. Its architecture allows to insert security, validation, logging and error handling layers cleanly. The API structure followed RESTful conventions closely. Each resource like Users, Workouts, Meals, WaterLogs and FitnessMetrics had its own set of endpoints supporting standard CRUD operations. These endpoints were grouped logically by router files which improved modularity making the codebase easier to find.

For security, user passwords were never stored in plain text. They were hashed using the bcrypt library with a strong salt factor. Authentication was managed using JWT (JSON Web Tokens). The server issued a signed token after a successful login containing the users ID and basic metadata. This token required access of all protected routes where that user data could not be accessed without proper authentication. Each request to the backend passes through a JWT verification middleware. If a valid token is present, the middleware attaches the user's identity to request context which allows secure user specific operations. If the token is missing, expired or invalid, the request is rejected with 401 Unauthorized statuses. This keeps unauthorized access attempts firmly blocked. Every API endpoint that processed user input whether updating a profile or submitting fitness metrics used validation middleware to sanitize and verify the incoming data. This stopped common security vulnerabilities like injection attacks, malformed requests or accidental data corruption. Other than security, performance was another focus area. The backend is designed to be stateless possible to make it easier to scale horizontally in case traffic increases. Heavy computational tasks specially the one involving historical data aggregation for analytics were delegated to FastAPI microservice.

We used Mongoose ODM (Object Document Mapper) for database operations that simplified interaction with MongoDB. Mongoose models were created for each major entity by enforcing schema consistency while still allowing the flexibility MongoDB offers. This structured approach reduced the chance of accidental data inconsistencies. Also it makes it easier for developers to understand how each collection was organized. Logging and error handling were built into the backend as well. Every request either successful or not was logged with timestamps, request metadata and status codes. Secure error logs were captured in the event of error for easy debugging without exposing sensitive information to users. We also adhered to good API hygiene practices throughout development such as consistent naming conventions, standardized error messages, proper use of HTTP status codes and documentation of endpoints. The backend was crafted with great care not just to fulfill functional requirements but to serve as a solid for the entire Fitnics ecosystem. It acts as the reliable bridge between users, data, services and analytics powering every interaction in a safe manner.

8. Analytics Microservice

It became clear during developing Fitnics that simply logging user data was not enough. We needed to provide meaningful insights based on the users collected data to empower them to understand their health and fitness journey. This realization led to the creation of the Analytics Microservice of architecture letting Fitnics from a simple tracker into a powerful personal fitness dashboard. The purpose of the analytics microservice was to handle all heavy tasks that would slow down the main backend server. Rather than having the ExpressJS backend perform deep data aggregation, these operations offloaded to a dedicated microservice built using FastAPI. This separation of concerns let each system to specialize. The backend could stay lightweight and fast for everyday operations while the microservice could focus entirely on data crunching and analytics. FastAPI supports for asynchronous request handling, automatic generation of OpenAPI documentation and excellent performance benchmarks. It meant that we could process large volumes of data efficiently. The microservice was designed to be stateless receiving JSON payloads from the backend, performing calculations and returning clean analytics results.

The microservice followed a simple but powerful workflow in term of design. When users requested analytics like when visiting their fitness dashboard, the frontend triggered an API call to the backend. The backend after authenticating the request would package the relevant user data. After that it would forward it as a secure HTTP request to the microservice. The microservice would then:

- Parse the incoming data.
- Perform required computations like aggregating daily steps or analyzing workout trends.
- Generate JSON responses containing summarized actionable insights.
- Send the results back to the backend which would then relay them to the frontend for visualization.

This structure provided important performance benefits. By keeping heavy processes off the main backend, the system was able to serve standard requests like login, workout logging or meal updates without noticeable delays. Meanwhile, analytics generation could run independently such that even large datasets could be processed without creating bottlenecks. Although the microservice was not exposed to external traffic but it still required a valid JWT token for each request from the backend. It means only authorized backend services could access the microservice protecting user data integrity. One of the advantages of this microservice architecture is scalability. If user numbers and data volumes increase in the future, we can easily scale the microservice separately. This is by deploying multiple instances behind a load balancer or migrating it to orchestration environment like Kubernetes. Its stateless nature makes it ready for horizontal scaling. Another benefit for future is flexibility. As analytics are processed separately, we can extend the microservice with time to present more sophisticated analysis techniques. This could be machine learning based predictions without disturbing the main application flow. The microservice embodies the spirit of Fitnics to deliver a smarter fitness platform.

9. Database Strategy

While designing the database for Fitnics, a strategy that could adapt and grow alongside user needs was needed. The data users would generate workout logs, nutrition information, hydration management, fitness metrics and notification were varied and dynamic. Heavily relational database structure would too restrictive for this kind of platform. So we chose MongoDB a NoSQL document oriented database for flexible and scalable data management. MongoDB's document model assembled data that naturally mirrored the real world actions. Each user had multiple related types of data which is also able to nest documents. They can create references between collections that gave us the flexibility and organization balance.

Our database was designed around several major collections. The Users collection stored core account information which includes name, email, hashed password and metadata like account creation date. It also referenced other user specific collections by User ID for

efficient queries. The Workouts collection stored detailed logs of user workout activities. Each workout entry took the type of exercise, targeted muscle group, any additional notes and the timestamp. We purposely separated workouts in their respective collection instead of embedding them. The Diet Profiles collection has users physical and dietary settings which includes height, weight, goal weight, age, gender, activity level and fitness goals. This let user update their diet information without affecting other data. It can also be supported for future extensions like automated meal plan generation. The Meal Plans collection has users to plan daily meals through meal and snack slots. Each meal plan entry has a date which is linked back to the user by ID. This design has ease to retrieve user's meal history or update plans for a specific date. The Water Logs collection tracked daily water intake of user. Each record included the quantity log with the date of consumption and related user ID. We could combine hydration metrics for analytics by keeping water intake records separate.

The Fitness Metrics collection was designed for different types of health management like steps taken, calories burned, workout duration, heart rate, weight and sleep duration. Each metric entry was timestamped and tagged by type. This gives users and system flexibility in analyzing their fitness habits over time. For reminding alerts, the Notification collection stored user preferences containing fields for entering user cell number along with notification types and last updated time. The Notification types are Water, Meal and Workout. This modular provide ease for managing notification subscriptions without disturbing core user data. By keeping clear separation between collections and linking them through User IDs, we got a scalable, flexible, efficient and secure database design.

We also implemented soft deletion principles if appropriate. Instead of permanently deleting important records like workouts or fitness logs, we used flags to mark them as deleted. This could help us for potential future recovery or historical analysis. This database strategy gave Fitnics a solid foundation so that user data could be stored and retrieved. We learned a critical lesson in our database design. This data management strategy is not just a technical decision but it deeply influence the user experience and system performance.

10. Authentication and Authorization

To meet the security needs, authentication flow based on JWT (JSON Web Tokens) was designed by our team. JWTs provided the stateless authentication solution fitting perfectly with our goals. It lets the system remain flexible and horizontally scalable without server side session storage dependency. The authentication road starts at the registration stage. When a user creates an account, the backend server first validates all incoming fields. This checks that the name, email, password and confirmation password are properly formatted.

Passwords are never stored directly but use the bcrypt hashing algorithm. It hashes the password securely with a strong salt factor before saving it in the MongoDB Users collection. Even if the database ever compromise, raw passwords would never be exposed.

When a user tries to log in, they provide their email and password. The backend gets the hashed password for email from the database using bcrypt for verification. This matches the provided password with stored hash. The backend generates a signed JWT token if verification succeeds. This token has the user's ID and few metadata fields which are signed securely with a server side secret key. The JWT token is sent back to the user's browser after that and stored in memory or secure cookie. This token must be attached to the header of all succeeding API requests in the form of Authorization: Bearer <token> header. A JWT verification middleware intercepts all protected route requests on the backend side. This middleware checks if Authorization header exists, parses the token, verifies its signature and also it is not expired. If all checks pass, the middleware attaches the user's identity information to the request object. This securely process the request in context of authenticated user. If validation fails, the middleware blocks the request immediately with a 401 Unauthorized response. Following steps should be made sure with this setup.

- Only authenticated users can do actions like logging workouts, updating profiles, planning meals or viewing personal analytics.
- Users cannot change or access data belonging to other users.
- Sensitive operations are protected from unauthorized manipulation.
- The backend remains stateless with horizontal scaling.

We also designed the tokens with appropriate expiration times. Short living tokens bound the vulnerability window if a token is ever leaked. We plan to implement refresh tokens in the future for balancing security and user convenience. This can let session continuity while keeping short lived access tokens. We also implemented authorization controls in addition to core authentication throughout the backend routes. Before performing sensitive operations, the backend double checks the authenticated user ID in match with the data owner's ID. This prevents malicious user attempt to interact with another user's records by modifying request payloads. Security focused design patterns extended past the authentication as well. Inputs through the system are validated for injection attacks prevention.

Building this authentication and authorization flow taught us how to implement secure login functionality. It showed us the broader principle about trust with protection at every layer of a platform. The authentication system is not just a technical feature but a trust model for

Fitnics. Our users deserve to know their journey toward better health is protected with high standards of security.

11. CI/CD Pipeline

We chose GitHub Actions as our CI/CD platform after researching several options. Such that our code repositories were already hosted on GitHub. Using GitHub Actions provided us great experience with excellent community support and powerful integration with Docker and DigitalOcean. Our CI/CD pipeline was designed for automatic trigger whenever code was pushed to the main branch. The workflow followed a logical sequence. Automated tests were run first. Even though our project was more focused on deployment and full stack integration. We still wrote key sanity tests for checking core backend routes which responded correctly with no broken basic authentication flow. These tests acted as our first line of defense where critical errors were caught immediately after code was committed. Once the tests passed, the pipeline moved to the Docker build phase. A new Docker images were built for frontend, backend and analytics microservice. Multistage Docker build was used to keep the final images as lightweight and secure as possible. After the build, images were pushed to our private Docker registry. This registry served as a secure repository where production server could pull latest versions of each service during deployment. By pushing images into a centralized managed registry, it was made sure that deployment artifacts were consistent and traceable.

The pipeline triggered a remote deployment script on our production server after the successful push to the registry. This script instructed Docker Swarm to pull the latest images and update the services with rolling updates without downtime. With stateless service, rolling updates could proceed safely without session loss to active users. To further defend deployments, health checks were built into Docker Swarm configuration. If a new container failed to start correctly, Swarm would roll back automatically to previous stable version. The following fully automated CI/CD setup brought several benefits to Fitnics:

- **Speed**: Deployments could happen within minutes of merging code.
- **Consistency**: Every build went through the same test and deployment steps which reduces human error.
- **Security**: Images were built from scratch every time reducing hidden vulnerabilities.
- **Scalability**: As more features were added, CI/CD pipeline could tackle parallel deployments with different environments.

CI/CD pipeline gave us confidence knowing that every change was automatically built, tested and deployed. This removed our anxiety which often is accompanied by manual

release processes. It helped our team to focus on writing code while trusting the infrastructure. We overcame challenges for speed, managing multi service deployments and monitoring pipeline failures. Each challenge became an opportunity to understand modern DevOps practices. GitHub Actions based CI/CD pipeline became silent but major part of Fitnics. It turned our project from a simple app to a cloud platform with continuous improvement.

12. Setup and Maintenance Guide

To set up the application locally or in production, configuring environment variables, deploying services using Docker Swarm, and managing ongoing operations through monitoring tools like Prometheus, Loki, and Grafana. Whether running Fitnics on a local machine for development or deploying it on a cloud platform like Digital Ocean, this guide outlines each step required to ensure the application is correctly installed, securely configured, and fully operational.

Refer Readme in the following Fitnics github repo:

https://github.com/sindhuja2002/fitnics/blob/dev/README.md

13. API Documentation

The Fitnics platform provides a complete list of APIs using Swagger making it easier to view and test API directly from a web browser. The document can be accessed for documentation by visiting https://backend.fitnics.space/api-docs. All the available API endpoints used in the system can be found here. The APIs could be logging in, registering, planning meals and tracking fitness activities. Each API shows the information needed to send like name, email or workout details. This also expect a kind of response from the back of the server whether its a success message or an error. Swagger interface is interactive meaning it can try out each API by entering data and seeing its response. This makes it useful for developers and testers to understand the backend mechanism.

14. Containerization and Orchestration

One of the main features of cloud applications is the use of containerization and orchestration. We chose Docker as our containerization platform due to its maturity and ease of use. Every component of Fitnics was containerized into its own Docker image. Each image was built from a minimal base, optimized for performance, security and small size. The Docker build process for each service was created carefully. For example, the React frontend was built using a two stage Dockerfile. Node.js built the production assets in the first stage and in the second stage, a tiny Nginx image served those assets. This method reduced the size of the final image and eliminated unnecessary build tools. Also like this, the backend and microservice images were based on slimmed down Node.js and Python images.

Containerization gave us major advantages during development as well. By running each service locally in containers during the coding phase, we understood that what we built in development would behave the same way in production. There were no issues from differences in environment configurations. We moved to orchestrating the services in production after they were containerized. We selected Docker Swarm for orchestration instead of Kubernetes because Swarm provided an easier learning curve. Swarm helped us define our full stack using simple YAML files called docker-compose.yml and stack.yml. These files declared the services, networks, volumes, environment variables and deployment policies for the complete application. We could deploy or update the entire Fitnics platform with a single command. The Swarm cluster was configured carefully during production. We set up a DigitalOcean droplet as the Swarm manager node with Traefik running as an ingress controller on the same node. Internal services were deployed as Swarm services with restart policies, health checks and resource limits. Swarm controlled internal DNS resolution where services could reference each other by name instead of IP addresses. Traffic to replicated services was balanced automatically through available containers. Deployments of new code versions could occur gradually without downtime.

An important part of orchestration setup was network segmentation. An internal overlay networks was created for service to service communication and public network for incoming HTTP/HTTPS traffic. Also we used persistent volumes for services like MongoDB and Loki. This means container restarts or node failures would not result in data loss. By combining Docker and Swarm, we got a system architecture that was elegant in its simplicity. Containerization and orchestration taught us where you can rebuild, redeploy and recover entire services from versioned images automatically.

15. Monitoring and Observability

Creating a production grade platform is not just about setting up deployments but also monitoring and observation. To build layer of system visibility, we integrated monitoring and observability stack based on three powerful tools that are Prometheus, Grafana and Loki. Prometheus is an open source system monitoring toolkit designed for reliability and scalability. It was responsible for metrics collection. Prometheus continuously scraped metrics from our Docker Swarm services at regular intervals. These included data points like CPU usage, memory consumption, container uptime, network traffic, request response times and custom application specific metrics. Service endpoints were configured to expose metrics using standard formats to make Prometheus work well with Docker Swarm. For example, the backend and microservice were set up to expose lightweight HTTP endpoints.

Grafana brought those collected metrics to life visually. It is an open source analytics platform exceling to create rich dashboards from diverse data sources. We integrated Grafana with Prometheus and built several key dashboards. A System Health Dashboard displaying CPU and memory usage around all services. An API Performance Dashboard tracking response times, error rates and request counts for backend endpoints. A Database Health Dashboard which monitors MongoDB operations, connection pool usage and query latencies. A Network Traffic Dashboard which visualize inbound and outbound bandwidth for each container. These dashboards gave us real time windows into our system behavior. They also gave us historical perspectives helping us correlate spikes in resource usage.

We used a lightweight logging solution named Loki to get log aggregation. Loki collects logs from all Docker containers in our Swarm cluster. It stores them in a centralized repository indexed by labels like service name, container ID and timestamp. We could simply open Loki browser that is Grafana integrated instead of manually SSH-ing into servers for checking log files. This improved our ability to debug issues quickly. The observability stack did more than just catching problems. It taught us how the system behaves under real conditions like how memory usage grows with increasing traffic. We also learned to set alert rules based on Prometheus metrics. For example, Prometheus would trigger an alert if the backend CPU usage exceeded a certain threshold. These alerts could be configured to send notifications via email or any other channels. Observability turned our deployment into a smart system that could be monitored, understood and continuously improved. Thanks to Prometheus, Grafana and Loki, we can see everything so that we can act and improve on time.

16. Proxy Server Setup

The need for intelligent traffic management was missed as Fitnics evolved. We needed a reliable way to route user requests to the correct service. This is where Traefik comes in. We chose Traefik as our proxy server and ingress controller. Traefik is a modern HTTP reverse proxy and load balancer designed specially for dynamic cloud native environments like Docker Swarm. Traefik discovers new services automatically as they are deployed. This routes traffic based on rules and easily cares TLS certificate management. Setting up Traefik involved several important steps. At first, we deployed Traefik as a Docker Swarm service with the necessary permissions to monitor Docker events. This allowed Traefik to detect automatically when a new service started. Services wanted to be exposed externally had to declare specific labels. This had to be in their Docker Compose files describing routing rules like hostname, path prefixes and entry points (HTTP or HTTPS).

For Fitnics, we configured Traefik to perform path based routing. Incoming requests to https://app.fitnics.space/ with no path prefix were routed to the frontend React application to serve the main user interface. API requests with path prefixes like /api were directed to the backend ExpressJS server. This made sure that API endpoints remained organized and separated from frontend routes. Requests for the analytics microservice with prefixes like /analytics were routed to the FastAPI service. One of the prevailing features of Traefik is its native support for automatic HTTPS. Using the Let's Encrypt service, Traefik can request, renew and manage SSL/TLS certificates without intervention manually. Traefik was set up during deployment to request certificates for the fitnics.space domain and its subdomains. This means that all user communications were encrypted by default. This removed the risks associated with manual certificate management which improved the overall security posture of the platform. It also meant that users could trust the green lock icon in their browsers as visual signal of safety.

Other than that, Traefik also figured load balancing transparently. If we deploy multiple replicas of the backend or frontend services in the future, Traefik can distribute automatically the incoming requests from all healthy instances. This gives better performance, fault tolerance and scalability without any additional load balancers or manual configurations. Traefik’s metrics integration with Prometheus further improved our monitoring abilities. We could visualize traffic volumes, request latencies and routing errors in Grafana dashboards. Thus giving us full visibility of the behavior of proxy layer.

One important advantage of Traefik was how it simplified our DevOps workflows. We reduced the need for complex and manual configuration files by relying on container labels and dynamic service discovery. Updates to routing logic could automatically happen as service deployments. It means that changes to Fitnics' architecture could be deployed safely without major downtime. While working with Traefik, we learnt a deeper understanding of modern traffic management in distributed cloud systems. We saw how a smart and dynamic proxy layer could simplify complexity, improve security and give more resilient user experience. In Fitnics, Traefik is not just a traffic router but a guardian of system health and security. It makes sure every user request reaches its correct destination quickly, safely and efficiently.

17. Cloud Hosting

Our project required a dependable, economical and developer-friendly platform which could support the containerized and orchestrated deployment architecture as per our plan. So, we needed to give a serious consideration to choose the best cloud hosting provider for Fitnics. We considered multiple providers like AWS, Google Cloud and Azure. Finally, we could come up with the decision to use Digital Ocean as our cloud hosting provider. We selected Digital Ocean because its benefits were perfectly aligned with our project requirements. Besides, this platform is simpler and affordable. We could create and manage the droplets very easily. So, we were able to put more focus and time on the development and implementation of Fitnics. Keeping a limited budget was a crucial factor in this project. We needed a reliable platform where we would remain fully aware of the infrastructure cost and we would not be charged for any hidden cost. And digital ocean provides more transparent and predictable pricing plans compared to other cloud service providers. Thus, this factor was taken as an important consideration while provider selection. We configured a droplet for the deployment so that we could balance CPU power, memory and SSD storage. The React Frontend, Express Backend. FastAPI microservice, MongoDB, Traefik, Prometheus, Grafana and Loki were all hosted on the droplet. The droplet acted as our Docker Swarm Manager node. After deploying the server, we updated the packages, set up a firewall using UFW (Uncomplicated Firewall) and set up SSH using public key authentication. These steps of security hardening made it sure that the production server could only be accessed by authorized team members.

After that, the Swarm Cluster was initialized, and Docker and Docker Compose were installed. Persistent storage volumes for MongoDB and Loki were also set up. SSL certificates were handled with Let’s Encrypt by configuring Traefik. Traefik also managed the inbound traffic and forwarded requests to the relevant services based on URL paths. The

deployment process of Fitnics became simple because of Github Actions and Docker Swarm. Our Docker registry received the most recent images from the automated CI/CD pipeline. Because of simple pull and stack deploy on droplet, all services were updated with little downtime. We set up health checks for every service to increase the resilience of the system. Docker Swarm automatically restarted the service in case of failed responses. The usage of Traefik helped to keep checking on service availability and made sure that user requests were never being sent to any unhealthy container. For tracking CPU usage, memory usage, disk I/O and bandwidth consumption, the built-in monitoring features of Digital Ocean played a vital role. We combined this with our own observability stack to get a full picture of system health all the time. Besides, we only made the necessary ports available to public for reducing the attacking surfaces. Every other service was closed off from outside access. Those services internally communicated over the overlay networks of Docker. For scaling up with large user base in future, Digital Ocean would be helpful by allowing us to add more droplets to the Swarm cluster. We would be able to distribute services among nodes and utilize the managed databases and load balancers of Digital Ocean.

18. Website Features

While developing Fitnics, the goal was to build a platform to inspire, empower and engage the users. For establishing system reliability, user experience was equally considered as an essential aspect along with infrastructure, microservices and backend. Users needed to be assured that Fitnics is beneficial for achieving their fitness goals and it is a user-friendly system.

Home page is the first place where the visitor interacts with the site for first time. The home page provides a concise overview of Fitnics. The visitors instantly get to know that it is a comprehensive fitness platform combined with management of nutrition, hydration and workout along with personal analytics. The navigation bar is featured with large buttons to encourage new users to Register or Login. On the Features page, an outline of Fitnics' features can be found. The users can easily understand the main features like Workout Database, Nutrition Checker, BMR Calculator, Meal Planner and Water Intake Log. This page gives the users a clear idea about what they can expect from signing up. It instantly gives them a sense of purpose and value. The Workout Database is one of the most comprehensive features of Fitnics. The muscle group categorized exercises are available for those users who want to create or enhance their workout plans. Back, Chest, Cardio, Lower Arms, Lower Legs, Upper Arms, Upper Legs, Shoulders, Waist and Neck are included in the categories. By choosing a category, the database is filtered to offer those exercises that are targeted for that group. The users of this database can easily select their ideal workout plans

according to their needs regardless of their workout experience. Users can create a Bro-Split Workout Plan in the Workout Plan section. This is a common technique for involving different muscle groups on different days of the week. The muscles that users want to target are chosen. Then Fitnics automatically creates a structured workout plan which offers diversity and targeted training. This option eliminates the nervousness of many beginners when they attempt to plan their workouts. Any food item can be searched in the Nutrition Checker to obtain detailed information. The measure of calories, protein, fats and carbohydrates are included in the results. Because of this, users can choose their diets intelligently without having to search in other platforms and relying on unreliable sources.

Another personalized tool for users is the BMR Calculator. Users can calculate their basal metabolic rate (number of calories they burn while at rest) by providing their height, weight and age. Users can plan their calorie intake according to their fitness goals by knowing their BMR. The About page gives a more human touch to the platform by reminding users that actual people built it with passion and care. After the users register and log in, the platform becomes stronger. Usernames are displayed in the top-right corner of the navigation bar to represent their individual relationship to the platform. When they click on their names, two key options Profile and Log Out appears. After entering the Profile area, a dashboard style interface with a sidebar menu of customizable choices appears. At this point, Fitnics becomes a comprehensive health management system for every user. The Update Profile section allows the users to manage their basic account details like name, email and password. The form is easier to use and offers friendly feedback and real time validation.

In the Update Diet Profile section, users can input and update critical fitness-related metrics like height, weight, goal weight, age, gender, activity level and fitness goal. The system guides users towards their nutritional goals based on this data by generating target values for daily calories, protein intake, fat intake and carbohydrate intake. Through the Meal Plan section, users can organize up to five meals and a snack for the day. Users can easily maintain their nutritional goals by saving meal plans. It also helps them to avoid spontaneous and less healthy choices throughout the day. The Water Intake tracker provides a user-friendly interface for logging water usage. Users can track how many liters they are consuming. This encourages them for building up better hydration habits. Users can manually log key health metrics like steps taken, calories burned, workout duration, heart rate, weight and sleep duration in the Fitness Tracker section. For every metric, platform shows progress bars and provides instant visual feedback on daily achievements. The Analytics dashboard pulls all logged fitness data together into powerful visualizations. Graphs and tables allow users to see trends over time. For example, improvements in daily steps or consistency in sleep habits. This real-world feedback can motivate users to stay

committed to their goals. In the Notifications section, users can configure mobile reminders for important tasks like drinking water, eating meals and completing workouts. These reminders help to keep up with healthy routines. Lastly, users can quickly mimic the activity data by clicking the Generate Test Data button. This button is especially useful for exploring the analytics dashboard before sufficient real-world data has been gathered. We were able to verify functionality without having to enter any input manually during the development and testing phases. Along with tracking statistics, Fitnics helps users to develop better habits, make wise decisions and maintain self-accountability through a smooth, robust and user-friendly experience.

19. Features of Profile Area

The Profile area inside Fitnics is the most personalized and powerful feature. It changes the application from a general fitness tool into a personalized health and lifestyle assistant for each individual user. We knew it from the beginning of the project that real engagement required personalizing the experience. Users can view their Profile Dashboard after logging in and clicking their username on the top-right corner. The dashboard is neat and well-organized. The sidebar menu provides access to various personalized sections. Users will not be overloaded with too many options because of this design. They can easily find what they need. The Update Profile section is the initial point of contact. Users can update their name, email, and password. The form provides prompt feedback when any of the fields are left blank, there is password mismatch or incorrect email format is provided. This is because it has real-time validation feature. Quick and easy profile updates encourage users to maintain their accounts up-to-date and secure. Further, the Update Diet Profile section allows users to adjust their fitness planning according to their own body metrics and goals. Users enter information like their current height, weight, goal weight, age and gender. Users can choose between two dropdown menus to select their activity level (Sedentary, Lightly Active, Active or Very Active) and their fitness goal (Maintenance, Cutting or Bulking).

After entering this information, the system determines the daily nutritional goals of every user. A demo of nutritional goals is presented in below table:

Metric	**Calculated Value**
Calories	2,300 kcal
Protein	150 g
Fat	70 g
Carbohydrates	280 g

The above table provides users a specific and achievable goals that are directly relate to their objectives. It removes guesswork and uncertainty from their nutrition planning.

Through the Meal Plan feature, users can plan their meals for each day. They can fill out details for Meal 1 through Meal 5 and a Snack slot. They can use this plan as a guide to stay disciplined and avoid impulsive and unhealthy eating choices. Hydration is also important as nutrition and exercise. This is why we built the Water Intake section. The users can log the amount of water they consume throughout the day. The interface shows the total amount consumed and offers a quick form to record additional intake. This quick and easy interaction encourages users to maintain good hydration habits constantly. In the Fitness Tracker section, users can manually log a variety of health metrics like steps taken, calories burned, workout duration, heart rate, weight and sleep duration. Each metric is visualized through daily progress bar. It offers immediate feedback and a sense of accomplishment. The Analytics integration gives Fitnics a unique value. When users visit the Analytics page, they can see their logged metrics through graphs and charts. Trends appear on a daily, weekly and monthly basis. The trends show how regularly users are meeting their step goals, how their weight is progressing toward targets or how well they are maintaining good sleep hygiene. The Analytics dashboard summarizes critical insights like Total Records (showing how much data the user has collected), Tracked Metrics (listing all the active metrics being monitored) and Latest Update (timestamp of the last activity entry). After the users view these metrics, they get to know about historical and predictive information. It enables them to celebrate little successes, modify habits early and maintain motivation during times when progress gets slower.

The Notifications settings are also customizable. Users can turn on mobile notifications for getting reminded about water intake, mealtimes and workout sessions. Users can create a supportive system with a simple mobile number and toggle switches. This can be fitted with their daily routines. The Generate Test Data is a unique feature which allows users to instantly populate their profile with simulated fitness data. This is especially helpful for new users who want to explore analytics features without having to wait weeks to build up real-world data. It also enabled us to perform actual testing throughout the development. Every section of the Profile area was designed by keeping ease of use, visual feedback and real impact in mind. The Profile area represents Fitnics as a personalized fitness ally platform. It made users visit to a personalized fitness ally that develops with them, adapts to them and supports them every step of the way.

20. Security Measures

From the very beginning, Security was a foundational pillar for Fitnics. We realized that users were trusting us with sensitive personal data like body metrics, workout logs, nutritional habits and sleep patterns. It was our responsibility to keep this information confidential, protected, and managed according to the highest standards of modern application security. Strong authentication was at the core of our security strategy. As mentioned previously, we developed a JWT-based login system that allowed for stateless and secure sessions. Passwords were never stored in plain text and every password provided by users during registration was salted and hashed using bcrypt before being stored in MongoDB. Hashing makes it impossible for hackers to obtain real user passwords even if the database is hacked. Also, our backend was designed to validate every request carefully. Every protected API route required a valid JWT token presented in the Authorization header. And every request used a secure server-side secret to validate the tokens. The request was instantly denied with a 401 Unauthorized error if the token was absent, expired or tampered with. We significantly considered input validation and sanitization across the entire application. Validation middleware was used to process all user inputs. So, this method prevented database corruption or server instability due to injection attacks, XSS vulnerabilities and faulty data. Sensitive backend data were never revealed and validation errors were always handled gently with user-friendly error messages. We totally kept sensitive environment variables like database URIs, JWT secret keys and API tokens out of the codebase. We stored them in secure .env files and injected into the containers at runtime through Docker Swarm secrets. Even if our repository was accidentally exposed, critical credentials would remain safe because of this.

Our major focus was also on securing data in transit. We deployed Traefik as a proxy server and configured it to automatically obtain SSL/TLS certificates from Let's Encrypt. This guaranteed that all traffic to and from the Fitnics platform was encrypted using HTTPS. Users could confidently interact with the site. They would know that their login credentials, personal data and analytics information were protected from interception or eavesdropping. The internal traffic between containers was secured using overlay networks inside our Docker Swarm setup. Critical services like MongoDB and the analytics microservice were never exposed directly to the internet. They communicated only via internal Docker networks. So, outside attackers could not even try to reach them. We also implemented basic rate limiting and request throttling techniques to prevent brute-force attacks and service abuse. Even though application-layer protections like Web Application Firewalls (WAFs) and DDoS protection could be added in the future as the platform grows, our current configuration provided a robust foundational level of security. Logging and

monitoring were also essential elements of security. Every API request was logged in a structured format via Loki whether successful or not. It helped us to detect suspicious patterns like repeated failed login attempts, unusual traffic surges or malformed payloads. We created an early-warning system that can detect potential incidents by comparing application logs with system data in Grafana dashboards.

Moreover, we applied the least privilege approach to all services. MongoDB collections, Traefik, and internal scripts were configured with only the necessary rights. This reduced the potential blast radius in case if any service was ever compromised. Also, teamwork and education have an impact. Our development team followed secure coding standards throughout the whole development process. It included frequent code reviews, discussions on potential vulnerabilities and use of actively maintained libraries with strong security record. The security measures we took across Fitnics are summarised below:

- Hashing and salting passwords using bcrypt.
- Token validation to secure JWT authentication
- Full input validation and sanitization on every route.
- SSL/TLS encryption using Let's Encrypt for automated certificate management.
- Isolation of secrets and environment variables using Docker Swarm secrets.
- Internal container networking to protect backend services.
- Organized logging and monitoring to detect anomalies.
- Rate limiting and request throttling to prevent misuse.
- Applying the principle of least privilege for all services and roles.

We protected both the integrity of the system and the trust of our users throughout this multi-layered defense approach. Fitnics was built not only for serving fitness needs. We prioritized the safeguard of the digital wellbeing of every user who chooses to make it part of their personal journey.

21. Challenges

Even though we aimed for a smooth and straightforward project journey, building a real-world cloud-native application like Fitnics inevitably brought lots of challenges and obstacles. These challenges tested our technical skills, our problem-solving abilities and our resilience as a team. When we look back on those situations now, we realize that these were not failures. All those hurdles were invaluable learning opportunities that sharpened our abilities far beyond that textbooks could have taught.

The service communication in Docker Swarm was one of the earliest and most persistent challenges we faced. When we initially deployed several services, it was more difficult to get them to communicate with each other within the overlay network. We spent hours troubleshooting these issues where services were unable to resolve each other's container names. This process taught us the importance of careful network configuration, Docker service labels and proper health checks. We eventually developed a deep understanding of Docker Swarm internals. It made our entire architecture more robust. Using Traefik and Let's Encrypt to handle SSL/TLS certificates was also a major challenge. Setting up Traefik to dynamically obtain certificates without manual intervention was more complex than expected. We encountered problems where misconfigured domain names, DNS settings or firewall rules prevented certificates from being issued. These experiences taught us the importance of domain validation processes, proper port forwarding (especially ports 80 and 443) and how to handle the finer points of ACME challenges. Authentication security also taught some lessons. Building a JWT authentication system sounded simple in theory. But it demanded great attention to detail in practice. In the early stage of development, we realized that if tokens were not carefully managed, users could end up being locked out of their accounts or get worse authenticated forever without proper session expiry. These insights made us implement strict token lifespans, careful validation and clear logout processes to maintain security without compromising user convenience. Additionally, we ran across performance challenges when handling bigger user data sets. When we simulated more active users through the Generate Test Data feature, we noticed that some API endpoints slowed down significantly. It happened especially when aggregating historical fitness metrics for analytics. These performance constraints required optimizing MongoDB queries, creating efficient indexes and delegating aggregation workloads properly to the FastAPI microservice. This experience taught us the importance of designing for scale even before the user base grows. This is a precious lesson we learnt which we will apply to all our future projects.

Monitoring and alerting setup were also a big challenge for us. Creating Grafana dashboards were easy. But setting up meaningful alert rules that did not generate noise took much more fine-tuning. We were overwhelmed with too many alerts. We could learn that DevOps is also an art besides being a science. It helped us to learn how to balance between sensitivity and stability. Team coordination became crucial challenge at some times. We faced multiple risks of merge conflicts and integration bugs. This happened because one person was focusing on backend routes, another focused on microservices, another on frontend components and another on CI/CD pipeline. We had to improve our communication patterns through clear task division, version control best practices and a shared understanding of deployment schedules. This sharpened our practical teamwork skills that

will serve us in any future collaboration. Maintaining consistent user experience across the entire platform was quite challenging. Different team members contributed to different pages. This made it easy for the frontend to drift into inconsistent button styles, form layouts or error handling approaches. We implemented a style guide and reusable UI components in React. Through this way, we maintained visual and functional consistency and made Fitnics feel like a single and unified application. There were some inevitable moments of frustration. Workloads became heavier than expected, got failed deployments and bugs emerged. Our shared sense of ownership and pride kept us going. We could conquer every challenge and strengthen our team bonding. In the end, we could add significance to our final product and this documentation.

22. Lessons Learned

While starting our work for Fitnics, we were initially about to build a cloud-native fitness application. But after finishing the project, we realized that we also built a deep body of experience, skill and professional growth. This whole project journey taught us far more than any textbook, tutorial or lecture could have. This project was not only about writing code. It was about building system architecture, solving real problems, collaborating effectively and adopting the mindset of modern cloud-native engineers. One of the most significant lessons we learned was the importance of modular architecture. User authentication, workout logging, analytics and notification settings could have easily been bundled into one giant backend server. But when we separated the backend API, analytics microservice, frontend SPA, we could create more scalable and resilient system. We learned that when a software is divided into isolated and manageable components, a good software is developed. We could learn another precious lesson was about automation and consistency through CI/CD pipelines. We only had idea about the theoretical concepts of automated testing, Docker builds and remote deployments. We could learn continuous integration and deployment through GitHub Actions. We learned to create a fully automated pipeline for faster development cycles, fewer human errors and smoother experience. We realized that real DevOps is not just about writing scripts. It is also about establishing trust, consistency and discipline in the growth process. Our skills on containerization and orchestration developed through this project. When we first started using Docker, we saw that the applications were running inside lightweight containers and separated from messy host dependencies. It really felt magical. However, the deployment of full-stack apps in a multi-service Docker Swarm cluster made us aware several complexities. Load balancing, persistent storage, networking and service discovery are among them. It taught us the way services communicate, recover and grow in distributed environments.

We could learn the advance knowledges of cloud and infrastructure management. We could know about the practical aspects of hosting production systems. These include configuring firewalls, securing SSH access, setting up domain names, managing SSL certificates, optimizing droplet resources etc. We gained confidence in deploying secure and efficient cloud services which will help us while working in real-world applications in future. Another significant learning was once about security practices. Putting plain text passwords and exposing container ports were much common. But now, we instinctively validate inputs, hash passwords, verify authentication tokens, encrypt traffic, isolate services and follow the principles of least privilege. This project made security practices more than just being abstract concepts. It became our default way of developing and operating every system. Additionally, we got the chance to grow as frontend developers. Our works did not remain limited only to building static pages. We created rich and dynamic SPAs that interact gracefully with APIs. By creating the Profile area, we could gain deeper understanding about integrating real-time form validations, organizing responsive layouts and considering user flows. We got to deal with working under pressure. Especially, teamworking under pressure is a big challenge. We had to learn to synchronize work across multiple modules and resolve the merge conflicts. Even if we had disagreements, we still had to respect each other's programming style, hold each other accountable and make necessary joint decisions. This experience taught us about team dynamics in real-world professional environments which is a significant technical skill. We became more confident through this project. As developed a functioning system from the ground up, it boosted our confidence. Previously, we used to have nervousness while troubleshooting, delivering functional codes, monitoring, implementing any fixes and improving the code. After this project, we could figure out the way to fix and improve the system even if it breaks.

In short, Fitnics was not only a course project for our team. It simulated the dynamic, chaotic and demanding nature of real-world employment. Every time a service was connected and new feature was launched, we could feel the sense of victory in a working environment.

23. Future Roadmap

The exciting areas for future development is the introduction of a mobile application. Fitnics is fully responsive smartphone's web browsers but a dedicated application would provide smoother experience. Mobile app features like push notifications, health data integration, offline logging and biometric authentication could improve engagement. Building the mobile app in React Native would let us use frontend components which we have already. Another future implementation could be wearable device integration. Users mostly track their fitness through smartwatches or any other similar wearable device. This offers users to sync

steps, heart rate data, sleep hours and calories burned automatically from devices. It would position Fitnics a competitive app in the app market. APIs for many of these devices already exist and could be incrementally integrated into the platform.

Some social features like friends, group challenges, leaderboards and shared meal plans could improve Fitnics through social support. Fitness journeys work better in social experience. We would also like to improve machine learning capabilities in analytics microservice. We could offer predictive insights with sufficient data like recommendation of workout plan to improve results. Simple models like nutrition suggestions based on past choices would upgrade Fitnics's intelligence. Another future consideration is the scaling infrastructure for larger audience. Fitnics runs good on single Docker Swarm node right now but user growth, shifting to Kubernetes could give better scaling. For data security and privacy, we plan to implement two factor authentication (2FA) with another layer of security. There could also be possibility of premium features which user can subscribe. Core features would remain free to promote health for everyone. Premium subscription would contain advanced analytics dashboard, personalized coaching recommendation or device integrations. Some additional ideas could be adding macro calculators for user planning custom diets and pre built workouts for different goals. Also there could be habit trackers added and guided programs which is adjusted weekly based on progress.

We hope to implement global CDN distribution at the infrastructure level for users worldwide with faster experience. Also moving database operations to managed MongoDB Atlas cluster would help scalability, backups and availability. The ideas are ambitious and could require much development efforts. The future of Fitnics is not only about adding features but strengthening the user focus on their health with richer and rewarding engagement. We have done this work so far to lay the ground work for future implementations. The real journey is when Fitnics started with the aim on focusing human health.

24. Conclusion

This project was never about final deliverable to the course but it was about a real world journey about making something meaningful from the start. This journey is about transforming ourselves from students into builders and architects. We embraced every unfamiliar technologies and faced problems with leaning to work under pressure. The process was unpredictable like real world but we never lost our project's aim. Fitnics is a platform where users can manage their fitness journey in personalized and insightful manner. It's a system running on scalable infrastructure with modern authentication and

deployed through automated pipelines defining the future of software engineering. Fitnics is more than a technical achievement while making fitness accessible to anyone and everywhere. All along the project, we learned skills that will stay helpful with us every time even after the project. We learned to design systems and automation for speed. We just didn't secure the data but also targeted user trust and solve any problem that feels roadblock. The most important the collaboration of our group who worked like real engineers as a team focusing on a mission. We got each others support throughout the project with mental and technical challenges. This skill will help us in every future project in every company we become part of.

In last, our project stands as a cloud native platform and it also shows the handwork our group skill growth. A group of students with a shared purpose reminds us that even complex projects can be achieved step by step with courage. The lessons learned in this projects will stay with us in our future careers. Though the codebase and newer tools will evolve but the core experiences of creativity and collaboration will stay forever.

25. Acknowledgment

Our team is incredibly thankful to every person who supported and guided us throughout the whole journey of bringing Fitnics to life. Our special thanks to the Cloud Services and Infrastructure course instructor. His focus on real-world applications, system thinking and hands-on experience gave us the foundation to build something truly meaningful. The high standards set for this project pushed us to explore architecture, security, deployment and scaling. We could learn something more beyond theoretical knowledge and get hands on experience on cloud services. We also owe immense gratitude to many open-source communities whose tools and documentation made it possible to build Fitnics. Their tools like Docker, ReactJS, FastAPI, Prometheus, MongoDB and Traefik allowed us to smoothly continue our work without rebuilding everything from scratch. This project would not have been possible without teamwork and mutual support among all project members. We could turn every challenge into solution through collaboration, patience and persistence. This made Fitnics a fully functional system rather than remaining just a concept.

Most importantly, we would like to express our gratitude towards the users who are committed to healthier lives and fitness goals. The idea of Fitnics was because of them and their strive made us think about developing such an application so that these users can easily fulfil their fitness goals. We hope that Fitnics supports and celebrates their journey even in a smaller way.

This project was just not an academic requirement for us. It also marked a milestone in our personal and professional growth. It is a proof of what can be achieved when people have a shared vision, resilient spirit and commitment to excellence.

Video Presentation:

1. The system in action (live demo of functionality).

https://drive.google.com/file/d/1XeDBLtZHkL540lwvqBkqX-yVOqviMOGj/view?usp=sharing

2. Architectural overview explaining service interaction, The development and deployment process and Challenges encountered and how they were solved are in the below video link:

https://drive.google.com/file/d/1XmYCpozhs9W0cR3xbWg2UG3ykGpXWAbI/view?usp=sharing

References

Lecture Videos and provided github materials in course.

Declaration of AI Usage:

1. *We have taken the help of ChatGPT to improve the clarity and flow of the technical documentation. ChatGPT was only used to support the writing process and polish the document. But the entire documentation content was solely based on our applied concepts, project files and outputs.*
2. *Concepts and Errors were solved by taking help form ChatGPT*