UNIVERSITY OF PIRAEUS – DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Computer Science»

ΠΜΣ «Πληροφορική»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title Τίτλος Διατριβής	**Architecting Scalable Solutions: A Diplomatic Exploration into Micro-Services and Modern Technologies** **Σχεδίαση Κλιμακούμενων Λύσεων: Μια Διπλωματική Έρευνα στις Μικρό-Υπηρεσίες και τις Σύγχρονες Τεχνολογίες**
Student's name-surname Ονοματεπώνυμο φοιτητή	**Konstantinos Kolios** **Κωνσταντίνος Κολιός**
Father's name Πατρώνυμο	**Lampros** **Λάμπρος**
Student's ID No Αριθμός Μητρώου	**ΜΠΠΛ21032**
Supervisor Επιβλέπων	**Efthimios Alepis, Associate Professor** **Ευθύμιος Αλέπης, Αναπληρωτής Καθηγητής**

Delivery Date/ Ημερομηνία Παράδοσης **April 2024/ Απρίλιος 2024**

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Efthimios Alepis	**Maria Virvou**	**Konstantinos Patsakis**
Associate Professor	*Professor*	*Associate Professor*
Ευθύμιος Αλέπης	*Μαρία Βίρβου*	*Κωνσταντίνος Πατσάκης*
Αναπληρωτής Καθηγητής	*Καθηγήτρια*	*Αναπληρωτής Καθηγητής*

Table of Contents

Abstract 8

Περίληψη 8

Chapter 1: Modern Challenges: Monolithic vs. Microservices 9

1.1 Introduction 9

1.2 Monolithic Architecture: Pros and Cons 9

1.3 Micro-Services Architecture: Pros and Cons 10

1.4 Architectural Decision-Making: Simplifying Complexity 10

Chapter 2: Overview of Key Technologies Utilized in the Application Development 11

2.1 Spring Boot: Streamlining Java-Based Microservices 11

2.2 Gradle: Modern Build Automation 11

2.3 Database and Schema Management 12

2.3.1 PostgreSQL: A Powerful Relational Database 12

2.3.2 Liquibase: Database Migration Made Easy 12

2.4 Monitoring and Metrics 13

2.4.1 Prometheus: Flexible Monitoring Solution 13

2.4.2 Grafana: Rich Visualization for Monitoring Data 14

2.5 Logging Solutions 14

2.5.1 Elasticsearch: Scalable Search and Analytics Engine 14

2.5.2 Logstash: Centralized Logging and Log Parsing 15

2.5.3 Kibana: Visualizing Elasticsearch Data 15

2.5.4 Filebeat: Lightweight Shipper for Log Data 16

2.6 Docker: Containerization for Seamless Deployment 16

2.7 Keycloak: Secure Authentication with JWT Tokens 17

2.8 Thymeleaf: Evaluating the Technology 17

Chapter 3: Application Architecture Breakdown 18

3.1 Advantages of Micro-services Adoption 18

3.2 Utilizing Eureka for Service Discovery 19

3.3 Gateway Service: Efficient Routing and Load Balancing 19

3.4 Configuration Service Management 20

3.4.1 Leveraging RabbitMQ for Centralized Configurations 20
3.5 Ensuring Business Logic Separation 21
3.5.1 Doctor Portal Service 21
3.5.2 Appointment Service 22
3.5.3 Payment Service 22
3.6 Choosing Gradle Over Maven: Enhancing Build Automation and Flexibility 22
3.7 Embracing Liquibase for Database Versioning: Robustness and Cross-Database Compatibility 23
3.8 Architectural Choice: Why Thymeleaf over React or Angular 24
Chapter 4: Application Guide 25
4.1 Role-Based Scopes 25
4.1.1 Doctor: Prescription Management, Patient Interactions, Appointment Handling, Payment Tracking 25
4.1.2 Patient: Appointment Scheduling, Doctor Search, Prescription Access, Payment Records 25
4.1.3 Admin: Role Permissions Management, Logging and Metrics Monitoring 25
4.2 Operational Scenario: Doctor Registration, Patient Simulation, Prescription Management 26
4.3 A Use Case for Adding Prescriptions in Our Healthcare Application 26
4.3.1 Setup local environment 26
4.3.2 Project Setup and Configuration 27
4.3.3 Microservices Initialization 27
4.3.4 Database Initialization (Initial Setup Only): 28
4.3.5 User Registration and Authentication: 28
4.3.6 Appointment Booking Process: 28
4.3.7 Doctor Confirmation and Payment Processing: 28
4.3.8 Prescription and Medical Procedures: 29
4.3.9 Visual Representation Use Case 29
4.3.9.1 Register User 29
4.3.9.2 Validate register user using mock taxis net 31
4.3.9.3 Schedule an Appointment 32
4.3.9.4 Ensure Appointment Status, Payment, and Prescription Validation for Patient Review: 35

4.3.9.4.1 Appointments 35
4.3.9.4.2 Payments: 36
4.3.9.4.3 Prescriptions: 36
4.3.9.5 Admin Panel: 36
4.3.9.5.1 Inspect the logging using ELK(Elastic Logstash Kibana): 37
4.3.9.5.2 Monitor the application using Grafana: 37
4.3.9.5.3 Monitor the database using Grafana: 38
4.3.9.5.4 Monitor the application instances: 38
Chapter 5: Concluding Thoughts and Future Enhancements 39
5.1 Delving into the Potential of Micro-Services: Balancing Enthusiasm with Pragmatism 39
5.2 Effortless Deployment with Cloud-Native Solutions 40
Bibliography: 42
Online Resources: 43

To my beloved family.

Abstract

In this thesis, a fully productive application will be presented, showcasing contemporary solutions and designs aimed at the development and expansion in the future within the healthcare sector. Its goal is to facilitate communication between doctors and patients. Challenges in selecting the appropriate software architecture for development are discussed, focusing on the differences and the advantages - disadvantages between Monolithic and Micro-services architectures. Finally, instructions for installing the application are provided, allowing the reader of this document to follow the user scenario described for a complete understanding of the implementation.

Περίληψη

Στην παρούσα διπλωματική θα παρουσιαστεί μια πλήρως παραγωγική εφαρμογή που παρουσιάζει σύγχρονες λύσεις και σχεδιασμό, με σκοπό την ανάπτυξη ,συντήρηση καθώς και την επέκταση της στο μέλλον σχετικά με τον τομέα υγείας με στόχο την διευκόλυνση της επικοινωνία μεταξύ γιατρών και ασθενών. Αναφέρονται οι προκλήσεις στην κατάλληλη επιλογή αρχιτεκτονικής για την ανάπτυξη λογισμικού, εστιάζοντας στην διαφορά και στα πλεονεκτήματα – μειονεκτήματα μεταξύ Monolithic και Micro-services. Τέλος, παρέχονται οδηγίες για την εγκατάσταση της εφαρμογής, ώστε να μπορεί ο αναγνώστης του παρόντος κειμένου να ακολουθήσει το σενάριο χρήστη που περιγράφεται με σκοπό την πλήρη κατανόηση της υλοποίησης.

Chapter 1: Modern Challenges: Monolithic vs. Microservices

1.1 Introduction

In today's rapidly evolving software landscape, developers face a host of challenges stemming from the growing complexity of applications and the need for scalable, resilient solutions. This chapter explores these contemporary programming challenges and the pivotal role that architectural design plays in addressing them.

1.2 Monolithic Architecture: Pros and Cons

Monolithic architecture, characterized by its cohesive structure where all components are tightly integrated into a single codebase, has long been a dominant approach in software development. Below, we dissect the pros and cons of this traditional architectural style:

Pros of Monolithic Architecture*:*

- *Simplicity*: Unified codebase simplifies development, deployment, and maintenance.
- *Ease of Testing*: Testing the entire system as a unified entity ensures comprehensive test coverage.
- *Seamless Integration*: Tightly coupled components facilitate smooth communication between modules.

Cons of Monolithic Architecture*:*

- *Scalability Challenges*: Scaling the entire system can lead to inefficiencies and performance bottlenecks.
- *Technology Lock-In*: Difficulty in adopting new technologies or updating existing ones without significant refactoring.
- *Deployment Complexity*: Updates may require redeploying the entire system, resulting in downtime and operational overhead

1.3 Micro-Services Architecture: Pros and Cons

In contrast to monolithic architectures, microservices architecture advocates for a decentralized approach, where applications are composed of loosely coupled, independently deployable services. Let's explore the advantages and disadvantages of this modern architectural paradigm:

Pros of Micro-Services Architecture:

Scalability: Granular scalability enables individual services to be scaled independently, enhancing flexibility and resource utilization.

Technology Diversity: Allows for the use of diverse technology stacks for each service, fostering innovation and adaptability.

Continuous Deployment: Facilitates rapid delivery of updates and features, enhancing agility and time-to-market.

Cons of Micro-Services Architecture:

Complexity: Managing service discovery, communication, and management introduces complexity, requiring robust infrastructure and tooling.

Distributed Systems Challenges: Network latency, data consistency, and fault tolerance pose challenges that require careful design and implementation.

Operational Overhead: Monitoring, logging, and coordination between services increase operational complexity and resource consumption.

1.4 Architectural Decision-Making: Simplifying Complexity

In many cases, significant time is devoted to deliberating over architectural choices, detracting from valuable implementation efforts. In our rapidly evolving environment, embracing a micro-services approach may offer a prudent solution to scalability concerns. However, considering the need for swift results, a monolithic architecture serves as a pragmatic starting point, allowing for rapid deployment and subsequent expansion as requirements evolve.

It's crucial to emphasize the importance of problem-solving over the creation of unnecessary complexities with overly intricate solutions. By adopting a pragmatic approach that prioritizes practicality and efficiency, we can navigate the complexities of architectural decision-making with clarity and purpose.

Chapter 2: Overview of Key Technologies Utilized in the Application Development

In the realm of modern software development, the choice of core technologies lays the foundation for the success of a project. Here, we delve into essential tools and frameworks that facilitate efficient development, deployment, and management of the existing application.

2.1 Spring Boot: Streamlining Java-Based Microservices

Spring Boot represents a paradigm shift in the Java ecosystem, offering developers a streamlined approach to building Micro-Services. At its core, Spring Boot embraces convention over configuration, reducing the need for verbose boilerplate code and simplifying setup tasks. Leveraging embedded servers, Spring Boot eliminates the complexity of external server configuration, enabling developers to focus on application logic rather than infrastructure concerns.

Moreover, Spring Boot's auto-configuration feature intelligently configures application components based on classpath scanning, minimizing manual configuration efforts. This approach significantly accelerates development cycles, allowing teams to rapidly prototype and iterate on microservices architectures. Additionally, Spring Boot provides a comprehensive ecosystem of starter dependencies, facilitating seamless integration with popular libraries and frameworks for tasks such as data access, security, and messaging.

2.2 Gradle: Modern Build Automation

Gradle emerges as a modern and flexible build automation tool, empowering developers to automate project tasks with ease. Unlike traditional build tools, Gradle adopts a Groovy-based DSL (Domain Specific Language) that offers expressive and readable build scripts. This declarative syntax simplifies the definition of project tasks, dependencies, and configurations, enhancing the maintainability of build scripts.

One of Gradle's standout features is its support for incremental builds, which ensures that only modified or dependent tasks are executed during each build cycle. This optimization minimizes build times, particularly in large-scale projects with complex dependency graphs. Additionally,

Gradle's dependency management capabilities enable seamless integration with external libraries and modules, promoting code reuse and modularity.

2.3 Database and Schema Management

Effective management of databases and schemas is fundamental to ensuring data integrity, consistency, and performance in modern applications. In this section, we explore two key components that facilitate database management and schema evolution.

2.3.1 PostgreSQL: A Powerful Relational Database

PostgreSQL stands as a robust and feature-rich relational database management system (RDBMS), trusted by developers and organizations worldwide. Renowned for its adherence to SQL standards and ACID (Atomicity, Consistency, Isolation, Durability) properties, PostgreSQL offers a reliable foundation for storing and querying structured data.

One of PostgreSQL's notable strengths lies in its extensibility, with support for a wide range of data types, indexing mechanisms, and advanced SQL features. Whether handling complex transactions, performing full-text search, or managing geospatial data, PostgreSQL delivers optimal performance and scalability. Furthermore, its active open-source community ensures ongoing development and support, contributing to its continued relevance in modern application development.

2.3.2 Liquibase: Database Migration Made Easy

Liquibase simplifies the process of managing database schema changes and versioning, providing developers with a robust and flexible solution for database migration. As an open-source tool, Liquibase allows developers to define database changes using version-controlled, XML-based change log files.

One of Liquibase's key advantages is its platform-agnostic nature, supporting various database management systems, including PostgreSQL, MySQL, Oracle, and SQL Server. This flexibility enables developers to seamlessly migrate database schemas across different environments without vendor lock-in. Additionally, Liquibase's support for rollback operations and change set

management ensures data consistency and integrity throughout the migration process, minimizing the risk of errors and downtime.

2.4 Monitoring and Metrics

Monitoring and metrics play a crucial role in ensuring the reliability, performance, and availability of modern applications. In this section, we explore two essential tools that facilitate effective monitoring and visualization of application metrics.

2.4.1 Prometheus: Flexible Monitoring Solution

Prometheus stands out as a versatile and powerful monitoring solution designed for cloud-native environments. Built with scalability and reliability in mind, Prometheus adopts a pull-based model for collecting time-series data, allowing for flexible instrumentation of applications and infrastructure components.

One of Prometheus's key features is its robust query language, PromQL, which enables developers to perform ad-hoc analysis and visualization of metrics. With PromQL, developers can define custom queries to extract insights from the collected data, facilitating informed decision-making and proactive incident management. Additionally, Prometheus's support for alerting and alert management enables developers to define custom alerting rules and receive notifications when certain conditions are met, empowering them to respond swiftly to potential issues.

Furthermore, Prometheus integrates seamlessly with container orchestration platforms such as Kubernetes, making it well-suited for monitoring dynamic and ephemeral environments. Its ecosystem of exporters allows developers to collect metrics from various sources, including applications, databases, and system components, providing comprehensive visibility into the entire infrastructure stack.

2.4.2 Grafana: Rich Visualization for Monitoring Data

Grafana complements Prometheus by providing rich visualization capabilities for monitoring data collected from various sources. With its intuitive and customizable dashboards, Grafana empowers developers and operators to gain actionable insights into system performance, resource utilization, and application behavior.

One of Grafana's strengths lies in its extensive library of built-in and community-contributed plugins, which enhance its versatility and extend its capabilities. Developers can choose from a wide range of visualization options, including graphs, charts, tables, and heatmaps, to create tailored dashboards that meet their specific requirements. Moreover, Grafana's support for alerting and notification channels enables teams to set up proactive monitoring and incident response workflows, ensuring timely detection and resolution of issues.

Overall, Prometheus and Grafana form a powerful combination for monitoring and observability, providing developers with the tools they need to monitor, analyze, and visualize application metrics effectively. By leveraging these tools, organizations can ensure the reliability and performance of their applications in today's dynamic and complex computing environments.

2.5 Logging Solutions

Logging solutions are indispensable tools for modern application development, offering essential capabilities for centralizing, parsing, visualizing, and analyzing log data. In this section, we explore key components of the logging ecosystem, each serving unique functions to enhance observability and troubleshooting efforts.

2.5.1 Elasticsearch: Scalable Search and Analytics Engine

Elasticsearch emerges as a cornerstone of the logging ecosystem, providing a dynamic platform for indexing, searching, and analyzing vast amounts of log data in real-time. Renowned for its scalability and performance, Elasticsearch excels in handling diverse data types and supporting complex querying operations.

At its core, Elasticsearch boasts a distributed architecture designed for horizontal scalability and fault tolerance. By distributing data across multiple nodes in a cluster, Elasticsearch ensures resilience and high availability, even under heavy workloads. Its powerful query language enables developers to perform advanced searches and aggregations, extracting valuable insights from log data with ease.

Furthermore, Elasticsearch's integration with Kibana and other Elastic Stack components enriches its capabilities, enabling seamless visualization and exploration of log data through intuitive dashboards and visualizations.

2.5.2 Logstash: Centralized Logging and Log Parsing

Logstash complements Elasticsearch by providing a versatile data processing pipeline for ingesting, parsing, and transforming log data from various sources. As a centralized logging solution, Logstash simplifies the collection and enrichment of log events before they are indexed in Elasticsearch.

A notable feature of Logstash is its extensive plugin ecosystem, which includes input, filter, and output plugins for handling diverse data formats and sources. Whether parsing structured logs, enriching log events with metadata, or routing data to different destinations, Logstash offers flexibility and extensibility to accommodate a wide range of use cases.

Moreover, Logstash's configuration-driven approach empowers developers to define custom data processing pipelines tailored to their specific requirements. This flexibility, combined with robust error handling and retry mechanisms, ensures reliable and efficient log processing in complex environments.

2.5.3 Kibana: Visualizing Elasticsearch Data

Kibana serves as the visualization and exploration layer of the Elastic Stack, providing developers and operators with intuitive tools for analyzing and visualizing log data stored in Elasticsearch. With its user-friendly interface and rich visualization options, Kibana enables users to uncover insights and trends within their log data effortlessly.

Key features of Kibana include customizable dashboards, interactive visualizations, and powerful search capabilities. Developers can create dynamic dashboards comprising multiple

visualizations, such as line charts, histograms, and heatmaps, to monitor application performance and detect anomalies in real-time.

Additionally, Kibana's integration with Elasticsearch's query language allows for ad-hoc searching and filtering of log data, facilitating rapid troubleshooting and root cause analysis. Whether exploring historical trends or investigating live incidents, Kibana empowers users to derive actionable insights from their log data with precision and efficiency.

2.5.4 Filebeat: Lightweight Shipper for Log Data

Filebeat serves as a lightweight log shipper designed to simplify the collection and forwarding of log data to Elasticsearch or Logstash for further processing. As an agent-based solution, Filebeat offers low resource consumption and minimal setup overhead, making it ideal for deployment in resource-constrained environments.

One of Filebeat's primary strengths lies in its broad compatibility with various data sources and formats, including log files, system logs, and container logs. By leveraging lightweight modules and efficient log harvesting techniques, Filebeat ensures reliable and timely delivery of log events to downstream logging pipelines.

Furthermore, Filebeat's support for dynamic configuration and stateful processing capabilities enables seamless integration with dynamic environments and log rotation scenarios. Whether monitoring standalone servers, containerized applications, or cloud-based infrastructure, Filebeat provides a flexible and scalable solution for collecting log data and enhancing observability across distributed systems.

2.6 Docker: Containerization for Seamless Deployment

Docker revolutionizes application deployment through containerization, offering a standardized and efficient approach to packaging, distributing, and running applications across diverse environments. At its core, Docker containers encapsulate application code, dependencies, and runtime environment, enabling consistent and reproducible deployments from development to production.

A key advantage of Docker is its lightweight nature, which allows developers to create and deploy containers quickly and efficiently. By leveraging container images and Dockerfiles, developers can define application environments and dependencies in a declarative manner, ensuring consistency and reproducibility across different environments.

Furthermore, Docker's ecosystem of tools and services, including Docker Compose for multi-container orchestration and Docker Swarm for cluster management, simplifies the deployment and scaling of containerized applications. Whether deploying microservices architectures or monolithic applications, Docker provides a flexible and scalable solution for modernizing and streamlining the deployment pipeline.

2.7 Keycloak: Secure Authentication with JWT Tokens

Keycloak emerges as a leading identity and access management (IAM) solution, offering robust authentication and authorization capabilities for modern applications. Built on open standards such as OAuth 2.0 and OpenID Connect, Keycloak provides a secure and flexible foundation for implementing authentication and authorization workflows.

A key feature of Keycloak is its support for JSON Web Tokens (JWT), which enable stateless authentication and token-based access control. By issuing JWT tokens to authenticated users, Keycloak simplifies the implementation of single sign-on (SSO) and federated identity solutions, allowing users to access multiple applications with a single set of credentials.

Furthermore, Keycloak's extensive feature set includes support for user management, role-based access control (RBAC), and fine-grained authorization policies. Administrators can define and enforce access policies based on user attributes, roles, and group memberships, ensuring compliance with security requirements and regulatory standards.

2.8 Thymeleaf: Evaluating the Technology

Thymeleaf is a modern server-side Java template engine for web and standalone environments. It stands out for its natural templating capabilities that enable the creation of dynamic and interactive web pages. Thymeleaf templates can be processed both on the server-side and on the client-side, offering flexibility in rendering approaches. Its seamless integration with Spring Boot and other Java frameworks makes it a popular choice for Java developers. Thymeleaf's

syntax closely resembles HTML, making it easy to learn and use, particularly for developers with HTML experience. Its templating features include attribute modification, iteration, conditionals, and more, enabling the creation of complex UIs with minimal effort. Thymeleaf emphasizes simplicity, making it suitable for projects where backend logic takes precedence over extensive frontend interactivity.

Chapter 3: Application Architecture Breakdown

In this chapter, we explore our application's micro-services architecture and our approach to build automation.

3.1 Advantages of Micro-services Adoption

Micro-Services adoption offers several compelling advantages that align closely with the requirements and objectives of the application. By decomposing the monolithic architecture into smaller, loosely coupled services, the application gains agility, scalability, and resilience.

One significant advantage is the ability to develop, deploy, and scale each service independently, allowing for rapid iteration and continuous delivery. This agility is particularly beneficial in a dynamic healthcare environment where evolving patient needs and regulatory requirements demand rapid adaptation.

Moreover, microservices facilitate technology diversity, enabling teams to select the most suitable tools and frameworks for each service's specific requirements. This flexibility fosters innovation and empowers developers to leverage the latest advancements in technology to address complex healthcare challenges effectively.

Additionally, microservices promote fault isolation, ensuring that failures in one service do not propagate to others, thereby enhancing system reliability and availability. This fault tolerance is crucial in healthcare applications where downtime or data inconsistencies can have significant consequences for patient care and safety.

3.2 Utilizing Eureka for Service Discovery

Eureka, a service discovery tool provided by Netflix OSS, plays a pivotal role in the application's architecture by facilitating dynamic service registration and discovery. As the application comprises numerous microservices serving various functions, Eureka enables seamless communication and interaction between these services.

Eureka operates on a client-server architecture, where microservices register themselves with the Eureka server upon startup. Through periodic heartbeats and health checks, Eureka ensures that the registry remains up-to-date, allowing clients to discover and consume available services dynamically.

By leveraging Eureka for service discovery, the application achieves improved resilience and scalability. Services can be added or removed dynamically without manual intervention, enabling elastic scaling and efficient resource utilization based on demand fluctuations.

3.3 Gateway Service: Efficient Routing and Load Balancing

The gateway service serves as the entry point to the application, responsible for routing incoming requests to the appropriate microservices and performing essential cross-cutting concerns such as authentication, authorization, and rate limiting.

By centralizing these concerns in the gateway service, the application simplifies the management of cross-cutting functionalities and ensures consistent enforcement across all microservices. This centralized approach enhances security, maintainability, and governance, mitigating the risk of unauthorized access or abuse.

Furthermore, the gateway service enables efficient load balancing and traffic management, distributing requests evenly across multiple instances of each microservice. This load balancing strategy optimizes resource utilization, improves response times, and enhances overall system performance and scalability.

3.4 Configuration Service Management

Configuration management is critical for maintaining consistency and coherence across the application's distributed architecture. The configuration service centralizes the management of configuration properties and settings, ensuring that all microservices adhere to the same configuration standards.

By externalizing configuration from the codebase and storing it in a centralized repository, the application enhances flexibility, scalability, and maintainability. Changes to configuration settings can be applied dynamically without redeploying the affected services, enabling rapid adaptation to evolving requirements.

Moreover, the configuration service supports versioning and auditing of configuration changes, providing transparency and accountability in the configuration management process. This audit trail is invaluable for troubleshooting issues, tracking changes, and ensuring compliance with organizational policies and regulatory requirements.

3.4.1 Leveraging RabbitMQ for Centralized Configurations

In this section, we delve into the strategic utilization of RabbitMQ for centralized configuration management within our application architecture. By leveraging RabbitMQ, we aim to streamline the distribution and management of configuration settings across the microservices ecosystem, thereby enhancing scalability, flexibility, and maintainability.

RabbitMQ serves as a robust message broker that facilitates the exchange of messages between distributed components within our application. Leveraging its asynchronous messaging capabilities, RabbitMQ enables seamless communication and coordination between microservices, allowing for efficient distribution of configuration updates in real-time.

One of the primary advantages of utilizing RabbitMQ for centralized configuration management is its support for pub/sub (publish/subscribe) messaging patterns. This enables us to decouple the configuration management process from individual microservices, ensuring that updates are propagated to all relevant components without introducing tight coupling or dependencies.

Additionally, RabbitMQ provides features such as message durability, acknowledgments, and message routing, which contribute to the reliability and resilience of our configuration

management system. Messages containing configuration updates can be persisted to durable queues, ensuring that they are not lost in the event of system failures or network issues.

Moreover, RabbitMQ's support for message routing enables us to implement sophisticated routing logic based on message attributes or content, allowing us to target specific microservices or groups of microservices with tailored configuration updates.

By centralizing configuration management with RabbitMQ, we can adapt to changing requirements and dynamic environments more effectively. Configuration updates can be propagated to all relevant microservices in a timely and consistent manner, enabling us to maintain configuration coherence and integrity across the entire application ecosystem.

3.5 Ensuring Business Logic Separation

In this section, we delve into the imperative of ensuring a robust separation of business logic through the strategic decomposition of our application into distinct microservices. This architectural paradigm facilitates enhanced modularity, scalability, and maintainability by segregating specific functionalities into self-contained components.

3.5.1 Doctor Portal Service

Positioned as the central hub of user interaction, the Doctor Portal Service serves as the primary UI interface for the application. This pivotal service orchestrates seamless communication with other integral components, including the Appointments and Payments services. Herein lies the crux of user authentication, meticulously facilitated by Keycloak. This pivotal authentication mechanism ensures stringent access control and user validation, leveraging JWT token authentication for role-based access and authorization. Moreover, beyond its authentication prowess, the Doctor Portal Service encapsulates a significant portion of the application's core business logic. This consolidation of logic within the Doctor Portal Service streamlines operational efficiency and promotes code coherence, thereby elevating the overall robustness and maintainability of the application.

3.5.2 Appointment Service

The Appointment Service emerges as a dedicated microservice singularly focused on managing appointment-related functionalities. By encapsulating appointment-specific logic within its purview, this service embodies a paradigm of functional specialization, thus fostering clarity and agility in development endeavors. The discrete nature of the Appointment Service empowers independent scalability and evolution, facilitating seamless adaptation to evolving business requirements and operational exigencies.

3.5.3 Payment Service

Exclusively designated for the handling of payment-related operations, the Payment Service stands as a bastion of financial transaction processing within the application architecture. By extricating payment logic into its autonomous microservice, we establish a delineation of responsibilities that fosters operational cohesion and resilience. This discrete segregation of payment functionalities not only enhances clarity and maintainability but also fortifies the application's security posture, mitigating potential risks associated with financial transactions.

3.6 Choosing Gradle Over Maven: Enhancing Build Automation and Flexibility

In this section, we explore the strategic decision to favor Gradle as our primary build automation tool over Maven. While Maven has long been a stalwart in Java development, Gradle offers compelling advantages that make it a superior choice for our project's dynamic requirements.

Gradle's build script DSL (Domain-Specific Language) stands in stark contrast to Maven's XML-based configuration, providing developers with an intuitive and expressive syntax. This allows for finer control and customization, empowering developers to craft tailored build workflows that precisely meet our project's needs. The flexibility and expressiveness of Gradle's DSL enhance productivity and adaptability, enabling smoother development processes.

Moreover, Gradle's support for incremental builds and task caching mechanisms significantly accelerates build times, offering a more responsive and agile development experience. This

efficiency is particularly valuable in environments where rapid iteration and deployment are essential, ensuring faster feedback cycles and quicker time-to-market.

In addition to its superior flexibility and efficiency, Gradle excels in managing complex project structures and dependencies. Its robust dependency resolution algorithm ensures reliable and consistent build outcomes, even in projects with intricate interdependencies or large-scale requirements.

Compared to Maven, Gradle offers a more modern and adaptable approach to build automation, making it better suited for our project's multifaceted needs. Its advanced features, intuitive syntax, and efficient build execution make Gradle the optimal choice for enhancing productivity, flexibility, and agility in our development workflow

3.7 Embracing Liquibase for Database Versioning: Robustness and Cross-Database Compatibility

In this segment, we delve into our strategic adoption of Liquibase as the primary solution for managing database schema versioning within our application, comparing it to the alternative tool, Flyway. Liquibase presents a comprehensive and adaptable approach to database evolution, empowering us to confidently navigate database changes with precision and efficiency.

One notable advantage of Liquibase is its database-agnostic nature, enabling seamless migration across various relational database management systems (RDBMS). This cross-platform compatibility liberates us from vendor lock-in, facilitating smooth transitions between different database platforms without sacrificing data integrity or consistency.

Additionally, Liquibase's declarative approach to database change management simplifies the process of defining and executing schema changes. By encapsulating changes within version-controlled change log files, Liquibase ensures transparency, traceability, and auditability throughout the database evolution journey.

Furthermore, Liquibase offers robust support for advanced features such as rollback scripts, change set dependencies, and preconditions, facilitating the orchestration of complex database

migrations with ease. Its seamless integration with version control systems like Git enhances collaboration and accountability, enabling effective teamwork and streamlined workflows.

While both Liquibase and Flyway serve as viable options for database versioning, Liquibase's broader feature set and cross-platform compatibility make it the preferred choice for our application's needs. By embracing Liquibase, we reinforce our application's resilience and scalability, establishing a robust foundation for seamless database evolution and management across diverse technological environments.

3.8 Architectural Choice: Why Thymeleaf over React or Angular

In making the architectural decision to utilize Thymeleaf for frontend development over more intricate frameworks such as React or Angular, personal preferences and project-specific needs significantly influenced the choice. Thymeleaf was selected primarily for its seamless integration with Spring Boot, a technology stack I am intimately familiar with. This integration offered a smooth development experience, aligning perfectly with the project's backend-driven focus. The simplicity of Thymeleaf's templating approach also resonated with me, allowing for rapid iteration on frontend designs without sacrificing attention to backend feature refinement.

Furthermore, Thymeleaf's server-side rendering capability emerged as a crucial factor in the decision-making process. Given the project's emphasis on fast initial page loads and SEO performance, Thymeleaf's approach proved advantageous over client-side rendering frameworks like React and Angular. While React and Angular offer superior frontend interactivity, the learning curve and complexity associated with these frameworks were potential obstacles that could have impacted development timelines and resource allocation.

Ultimately, the decision to choose Thymeleaf was rooted in the project's specific requirements and my familiarity with the technology stack. While React and Angular may offer advanced frontend capabilities, Thymeleaf's simplicity, compatibility, and efficiency aligned more closely with the project's goals, ensuring a smooth and efficient development process.

Chapter 4: Application Guide

4.1 Role-Based Scopes

To ensure security and enhance reusability within the application's user interface, a deliberate decision was made to segregate user roles based on specific contexts. These roles were delineated into doctors, patients, and administrators, each assigned distinct scopes and associated business logic. Below, I will elucidate the responsibilities and privileges inherent to each role:

4.1.1 Doctor: Prescription Management, Patient Interactions, Appointment Handling, Payment Tracking

The doctor role encompasses a myriad of responsibilities crucial to the efficient functioning of the application. Doctors possess the authority to accept or reject appointment requests, track payment history associated with appointments, and crucially, issue prescriptions to patients. This role serves as the linchpin in facilitating seamless interactions between healthcare providers and patients, with a primary focus on delivering quality care and ensuring effective treatment outcomes.

4.1.2 Patient: Appointment Scheduling, Doctor Search, Prescription Access, Payment Records

Patients wield the ability to navigate the healthcare ecosystem through a comprehensive suite of functionalities tailored to their needs. Key among these capabilities is the capacity to schedule appointments with healthcare providers, search for suitable doctors based on specific criteria, access prescribed medications, and view comprehensive records of payment transactions. This role empowers patients to take an active role in managing their healthcare journey, fostering a collaborative and patient-centric approach to treatment.

4.1.3 Admin: Role Permissions Management, Logging and Metrics Monitoring

Administrators serve as the gatekeepers of the application's operational integrity, entrusted with overseeing role permissions management and monitoring critical system metrics. In addition to managing user accounts and defining role-specific permissions, administrators play a

pivotal role in maintaining the application's robustness through diligent logging and metrics monitoring. This role ensures adherence to regulatory standards, identifies potential issues proactively, and safeguards the application against unauthorized access or malicious activity.

4.2 Operational Scenario: Doctor Registration, Patient Simulation, Prescription Management

In this section, I will present a practical use case scenario illustrating the application's functionality. Through a series of screenshots and step-by-step instructions, I will demonstrate the process of registering a new doctor within the system, mock patient profiles for appointment scheduling purposes, and simulate the prescription issuance process. By providing a visual walkthrough of these essential operations, users will gain a comprehensive understanding of the application's capabilities and user workflows, facilitating seamless adoption and utilization.

4.3 A Use Case for Adding Prescriptions in Our Healthcare Application

In this scenario, we illustrate the process of adding prescriptions for patients within our healthcare application, demonstrating how healthcare providers can efficiently manage patient treatments.

4.3.1 Setup local environment

1. **Git**: Version control system for tracking changes in source code during development.
    - Download: Git
2. **Gradle**: Build automation tool for managing dependencies and building projects.
    - Download: Gradle
3. **JDK 17**: Java Development Kit, the environment required to develop and run Java applications.
    - Download: JDK 17
4. **Docker Desktop**: Platform for building, sharing, and running containerized applications.
    - Download: Docker Desktop

5. **Liquibase**: Database schema change management tool for tracking, managing, and applying database changes.
    - Download: Liquibase
6. **Python**: Optional, but useful for various scripting and development tasks.
    - Download: Python

4.3.2 Project Setup and Configuration

Clone the project repository using the command: git clone https://github.com/SylvanasGr/my-doctor-app.

Open the project directory located at '../my-doctor-app'.

Follow the instructions provided in 'my-doctor-app/api-gateway/keycloak.md' for setting up Keycloak.

4.3.3 Microservices Initialization

Start the microservices in the following sequence:

- discovery-server
- configuration-service
- api-gateway
- my-doctor-app-service
- payment-service
- appointment-service

4.3.4 Database Initialization (Initial Setup Only):

If setting up for the first time:

- Open a terminal and navigate to the path '~/my-doctor-app-service/liquibase/changelog'.
- Execute the command 'liquibase update' to create the schemas.
- Run the script located at './global/bash_scripts/citizen_mock_data.sh' to populate mock citizen data.

4.3.5 User Registration and Authentication:

Select a random record from the 'citizen' table to simulate user identity (*SELECT * FROM citizen ORDER BY random() LIMIT 1*).

Visit http://localhost:8500/login and select '**Login**' -> '**Keycloak**', then choose '**Register**'.

Provide your real information, excluding the data from the user selected in the previous step.

Upon successful login, navigate to the **'User Page**' to validate your existence by providing your tax number and social number to the system.

4.3.6 Appointment Booking Process:

Navigate to the '**Doctors**' tab to request an appointment.

Choose **'Schedule Appointment**' and select a date and time.
Monitor appointment status in the '**Appointments**' tab.

4.3.7 Doctor Confirmation and Payment Processing:

Open another browser, log in as the chosen doctor from the previous step, and navigate to '**Appointments**' to accept the appointment.

Refresh the page to validate that the appointment has been accepted (Status: Accepted), and check the '**Payments**' tab to view the payment history.

4.3.8 Prescription and Medical Procedures:

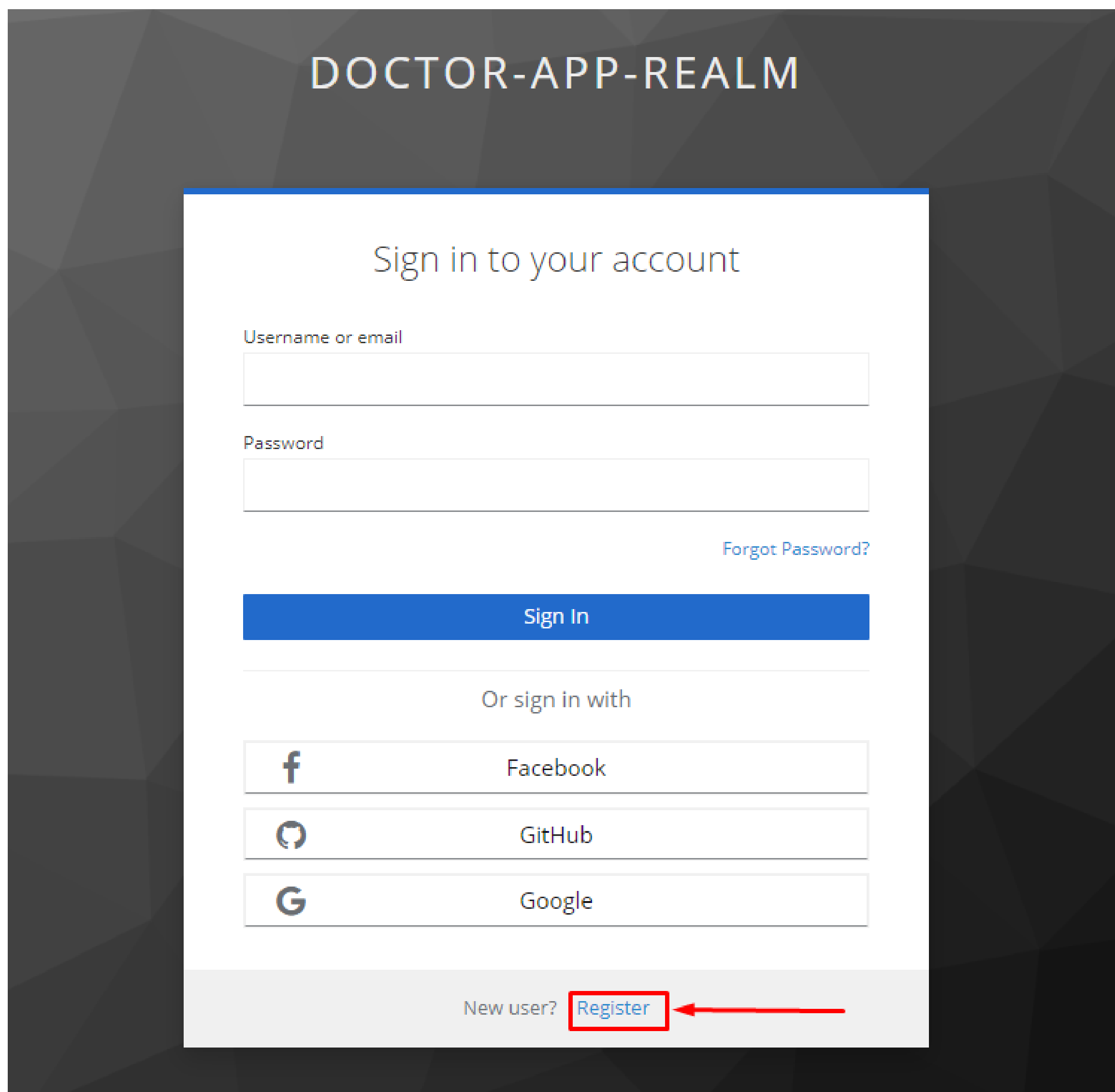
As a doctor, add the patient for the appointment using the '**Add Patient**' button and prescribe necessary medications (**Add Prescriptions**).

Check the associated prescriptions from the '**Prescription**' tab as a user.

4.3.9 Visual Representation Use Case

4.3.9.1 Register User

In order to register a user select the 'Register' option:



Fill in the relevant data and submit the registration by pressing the 'Register' button:

DOCTOR-APP-REALM

Register

First name
Konstantinos

Last name
Kolios

Email
konstantinoskolios@unipi.com

Username
konstantinoskolios1994

Password
•

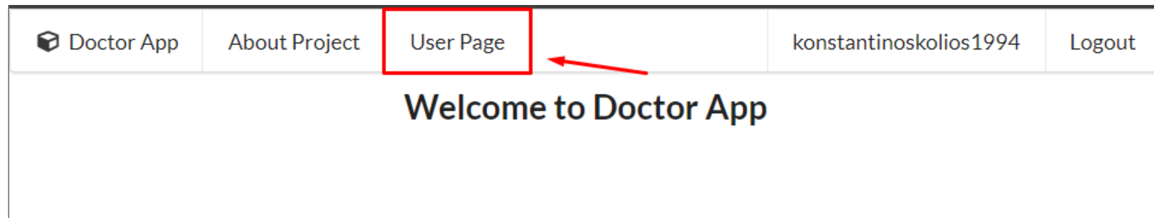
Confirm password
•

« Back to Login

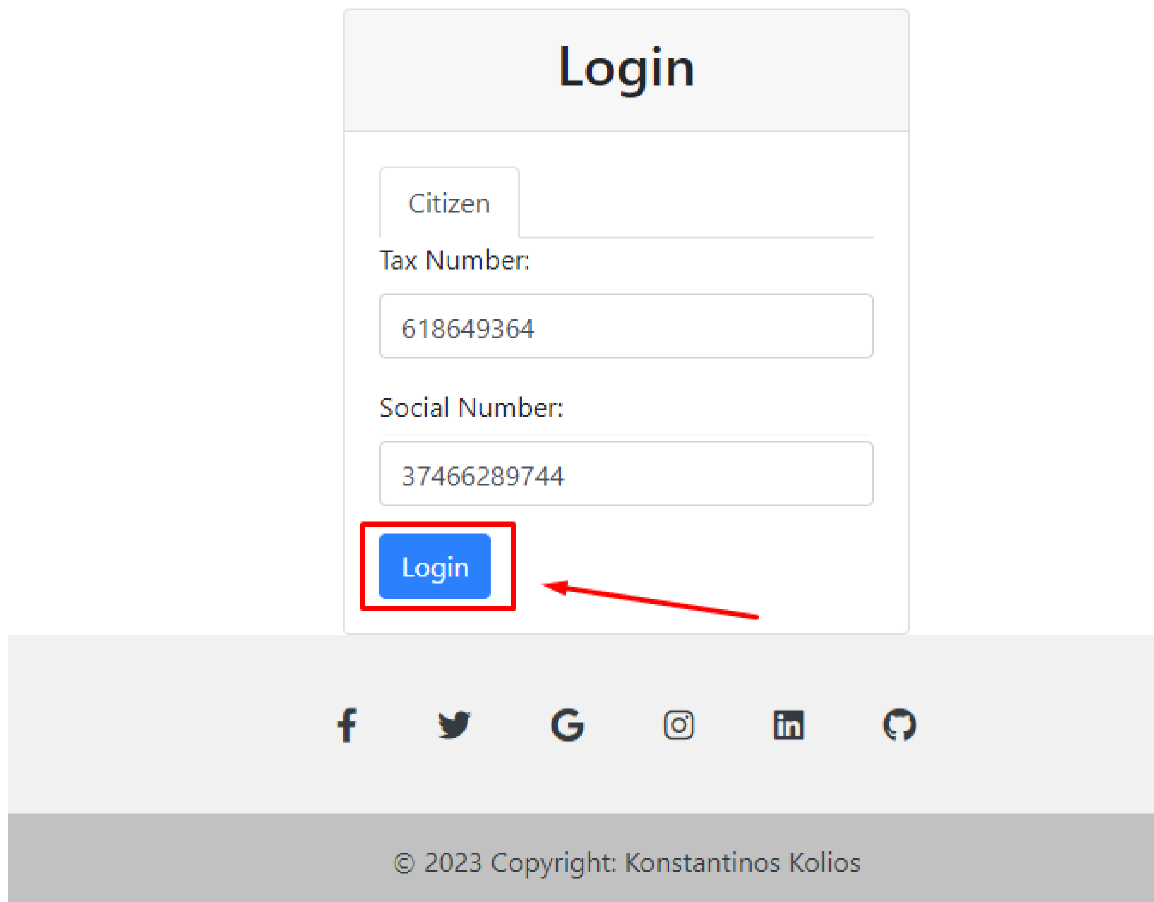
Register

4.3.9.2 Validate register user using mock taxis net

Navigate to 'User Page' panel:

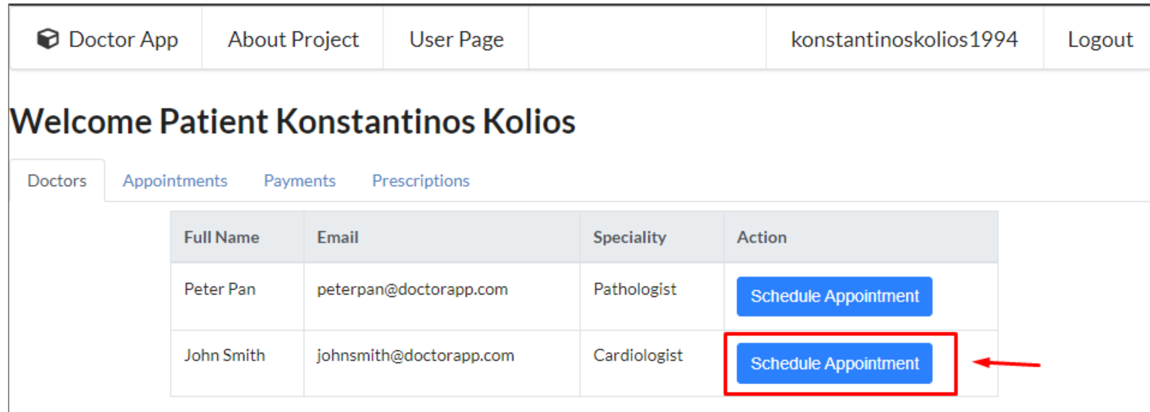


Fill the data and press the 'Login' button:

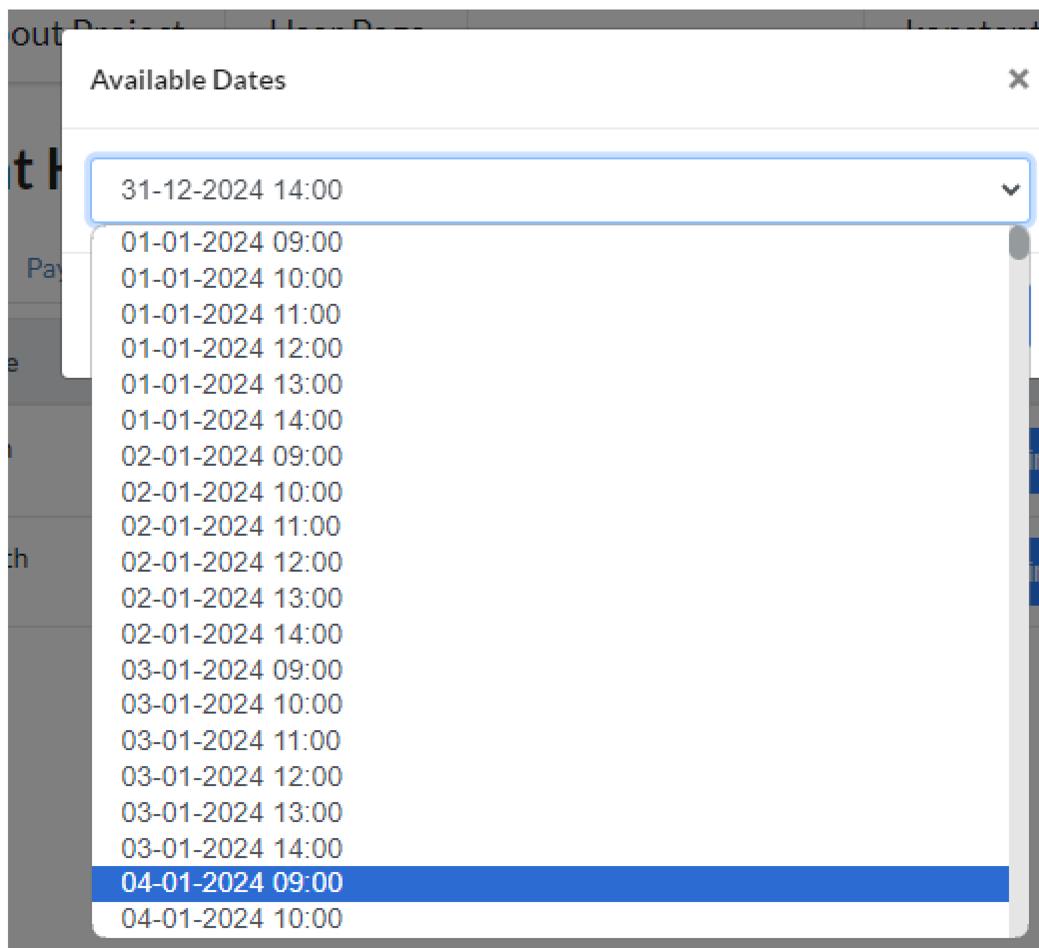


4.3.9.3 Schedule an Appointment

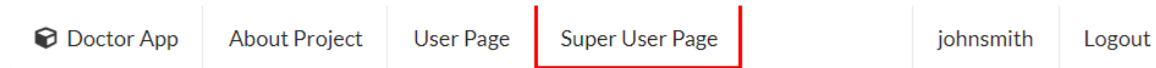
Select 'Scheduled Appointment' button:



Pick a date and press 'Schedule' button:



Open a new tab and log in to the application with John Smith doctor as describe at the section 4.3.2, and choose 'Super User Page' and then 'Appointments' and Accept the appointment request:



Welcome Dr. John Smith, Cardiologist

Patients Appointments Payments

Full Name	Social Security Number	Phone Number	Appointment Date	Status	Action
Konstantinos Kolios	37466289744	6942899650	04-01-2024 09:00	Pending	Accept Reject

Then add the patient to the doctor's list and write a prescription:

Welcome Dr. John Smith, Cardiologist

Patients Appointments Payments

First Name	Last Name	Tax Number	Social Security Number	Phone Number	Birthdate	Comments (Press enter to save)	Action

Add Patient

Choose associate patient:

Citizen Search

First Name | Ascending | Apply Sort

Search by Tax, Mobile, or Social Number | Search

First Name	Last Name	Father First Name	Tax Number	Social Security Number	Phone Number	Birthdate	
George	Smith	Adam	462035278	35012437791	6950909920	1985-08-04	Add Patient
Konstantinos	Kolios	Willie	618649364	37466289744	6942899650	1957-09-24	Add Patient

Page 1 of 1
Displaying items 1 - 2 of 2

© 2023 Copyright: Konstantinos Kolios

Choose Prescription:

Doctor App | About Project | User Page | Super User Page | johnsmith | Logout

Welcome Dr. John Smith, Cardiologist

Patients | Appointments | Payments

First Name	Last Name	Tax Number	Social Security Number	Phone Number	Birthdate	Comments (Press enter to save)	Action
Konstantinos	Kolios	618649364	37466289744	6942899650	1957-09-24	As soon as possible!	Add Prescription Remove Patient

Add Patient

Write a prescription with the following order:

Patient Name: Konstantinos Kolios

Category 2 | Prescription Name 3 | Date 4 | + 1

Metabolic | COVID-19 PCR Test | 01/04/2024 | -

Submit 5

© 2023 Copyright: Konstantinos Kolios

4.3.9.4 Ensure Appointment Status, Payment, and Prescription Validation for Patient Review:

4.3.9.4.1 Appointments:

Doctor App | About Project | User Page | konstantinoskolios1994 | Logout

Welcome Patient Konstantinos Kolios

Doctors | Appointments | Payments | Prescriptions

Full Name	Email	Speciality	Appointment Date	Status
John Smith	johnsmith@doctorapp.com	Cardiologist	04-01-2024 09:00	Accept

4.3.9.4.2 Payments:

Doctor App | About Project | User Page | konstantinoskolios1994 | Logout

Welcome Patient Konstantinos Kolios

Doctors | Appointments | Payments | Prescriptions

Full Name	Email	Speciality	Created Date	Amount
John Smith	johnsmith@doctorapp.com	Cardiologist	2024-03-20	75
				75

4.3.9.4.3 Prescriptions:

Doctor App | About Project | User Page | konstantinoskolios1994 | Logout

Velcome Patient Konstantinos Kolios

Doctors | Appointments | Payments | Prescriptions

Full Name	Email	Prescription	Category	Date
John Smith	johnsmith@doctorapp.com	COVID-19 PCR Test	Metabolic	01/04/2024

4.3.9.5 Admin Panel:

Lastly you can log in with Admin User and see how many users register to the app for specific periods:

Doctor App | About Project | Admin Page | User Page | admin | Logout

Welcome Admin Kolios

Business | ELK | Grafana | Pg Exporter | Eureka

Admin Dashboard

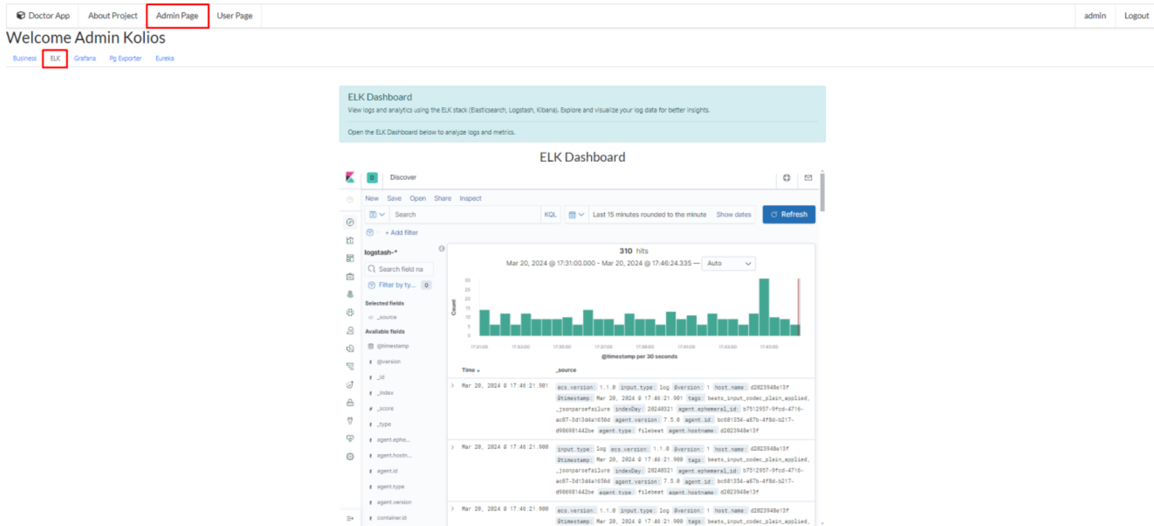
Start Date:
01-01-2024

End Date:
01-04-2026

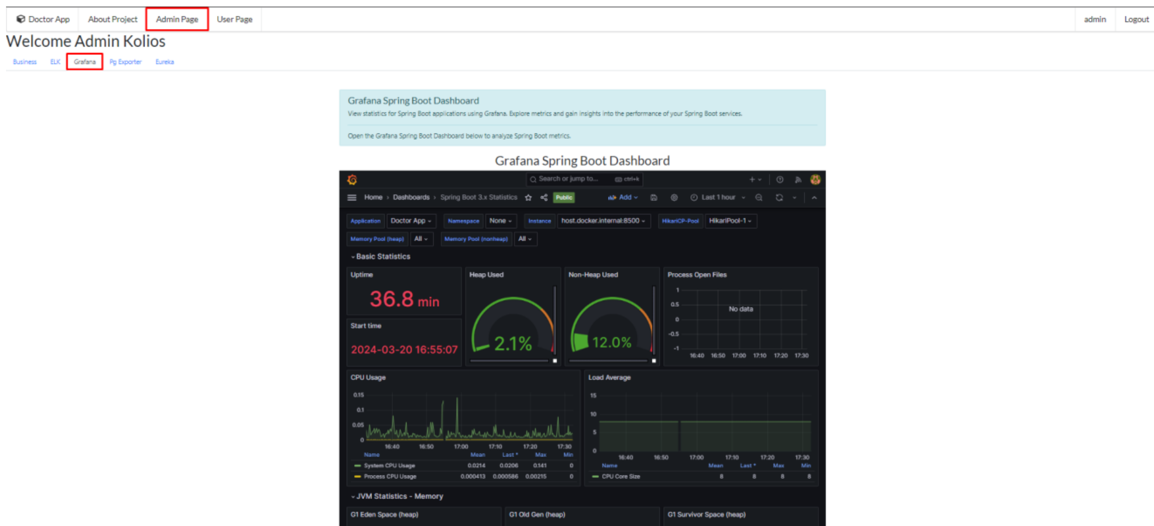
Check Counts

Total Patient Accounts: 2 and Total Doctor Accounts: 2 for period: [01-01-2024 - 01-04-2026]

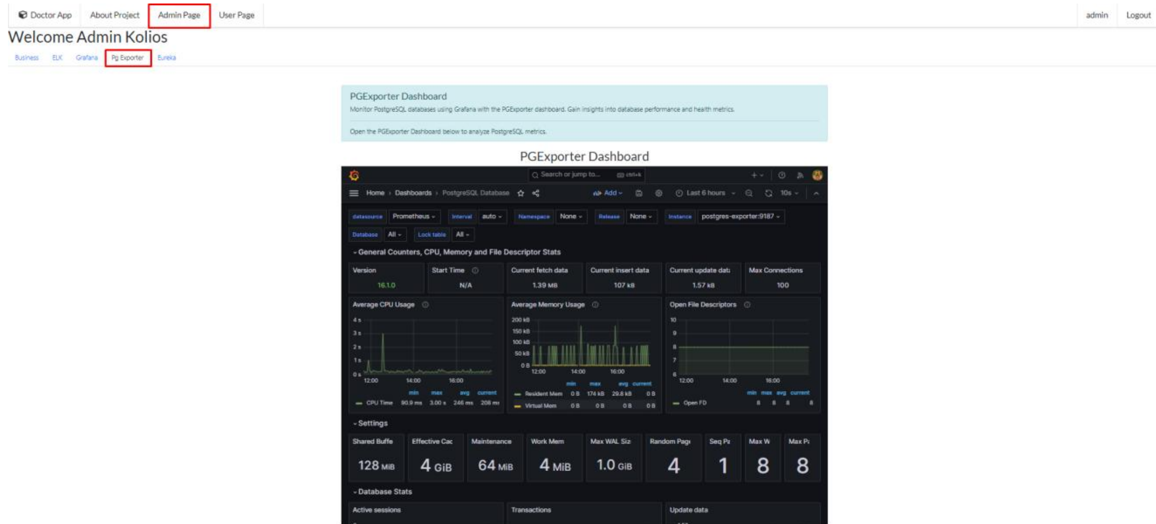
4.3.9.5.1 Inspect the logging using ELK(Elastic Logstash Kibana):



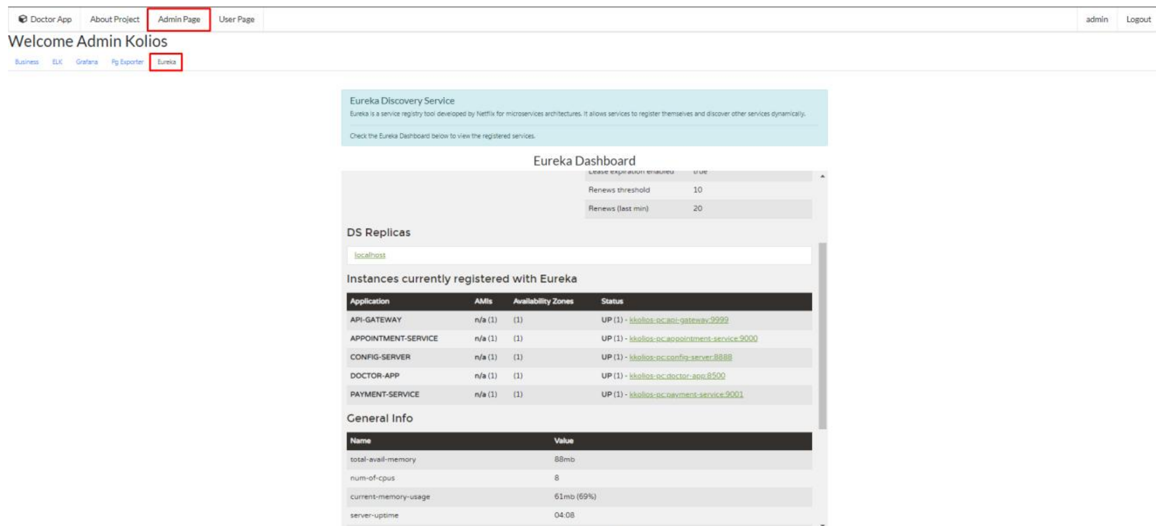
4.3.9.5.2 Monitor the application using Grafana:



4.3.9.5.3 Monitor the database using Grafana:



4.3.9.5.4 Monitor the application instances:



Chapter 5: Concluding Thoughts and Future Enhancements

5.1 Delving into the Potential of Micro-Services: Balancing Enthusiasm with Pragmatism

When first introduced to micro-services architecture, I found myself exhilarated by its promises of unparalleled flexibility and scalability. The prospect of breaking down monolithic applications into smaller, independently deployable services seemed like a revolutionary leap forward, especially in the context of scaling solutions and facilitating collaborative development efforts. The ability to utilize diverse languages, conduct autonomous testing, and streamline development processes held immense appeal, particularly in projects involving multiple teams working in parallel.

However, as with any architectural approach, it's essential to temper enthusiasm with pragmatism. While micro-services offer compelling advantages, they are not a one-size-fits-all solution. In scenarios where projects are nascent and business logic is relatively simple, the overhead of implementing micro-services may outweigh the benefits. Monolithic architectures, despite their drawbacks such as longer build times and centralized solutions, remain a viable option for rapidly prototyping and iterating on ideas. The transition from a monolithic to micro-services architecture can be a gradual process, initiated as the project matures and the need for scalability and autonomy becomes apparent.

Reflecting on my journey with micro-services, I discovered that simplicity often lies in elegance. Initially, I encountered challenges stemming from excessive HTTP calls between microservices, leading to unnecessary complexity and performance overhead. It dawned on me that a monolithic approach, consolidating all business logic within a single application, could potentially streamline operations and reduce network dependencies.

However, through proper redesign and refactoring of the micro-services architecture, I was able to overcome these challenges. By optimizing communication pathways and minimizing unnecessary interactions between services, I achieved a more efficient and resilient system. Moreover, I came to realize the fundamental difference between micro-services, which are network-independent, and monolithic architectures, which are CPU-dependent. This insight prompted a shift in perspective, highlighting the importance of understanding the underlying principles and trade-offs associated with different architectural paradigms.

In conclusion, my journey with micro-services architecture has been a transformative experience, marked by both excitement and introspection. While the allure of scalability and flexibility initially captivated me, it was through practical challenges and iterative refinement that I truly grasped the nuances of architectural decision-making. By embracing pragmatism, continuous learning, and a willingness to adapt, I have gained valuable insights into the complexities of software design and the importance of striking a balance between innovation and practicality.

5.2 Effortless Deployment with Cloud-Native Solutions

One of the key hurdles encountered in implementing the use case outlined in this thesis is the intricacy of setting up a local development environment. The process involves tasks such as installing dependencies, ensuring compatibility across various operating systems, and managing infrastructure configurations, which can pose significant barriers to the application's adoption and exploration.

To overcome these challenges and lay the groundwork for future enhancements, I propose deploying the application to the cloud using Kubernetes in conjunction with a suitable cloud provider. By harnessing Kubernetes' robust container orchestration capabilities, the aim is to abstract away the complexities associated with managing infrastructure, thus facilitating seamless scaling and deployment processes.

Furthermore, by embracing cloud-native solutions such as Amazon Web Services (AWS) or Google Cloud Platform (GCP), I intend to capitalize on managed services and principles of infrastructure-as-code (IaC). This approach will enable the streamlining of deployment workflows, enhancing operational efficiency and scalability.

In addition to cloud deployment, the plan involves implementing pipelines and automation for essential development tasks such as testing and linting. By incorporating continuous integration and continuous deployment (CI/CD) pipelines, the goal is to enable rapid iteration, ensuring that changes are thoroughly tested and seamlessly integrated into the application. Automation of repetitive tasks and enforcement of coding standards will further enhance the reliability, maintainability, and scalability of the application over time.

In conclusion, by carefully evaluating the suitability of micro-services for new projects, and by embracing cloud-native deployment options such as Kubernetes and AWS, organizations can effectively address deployment challenges and unlock the full potential of modern application architectures. Through strategic planning, disciplined execution, and a commitment to continuous improvement, we can pave the way for a future where software development is characterized by agility, scalability, and resilience.

Bibliography:

1. Fowler, M. (2012). **Monolithic vs. Microservices Architecture**: *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Addison-Wesley Professional.
2. Malin, M., & Berglund, E. (2016). **Spring Boot in Action**: *Spring Boot in Action*. Manning Publications.
3. Muschko, B. (2020). **Gradle Beyond the Basics**: *Gradle Beyond the Basics*. O'Reilly Media.
4. Momjian, B., & Rigsbee, J. (2017). **PostgreSQL Up and Running**: *PostgreSQL: Up and Running*. O'Reilly Media.
5. White, N. (2012). **Liquibase: Continuous Database Evolution**: *Liquibase: Continuous Database Evolution for Databases in an Agile World*. Manning Publications.
6. Soundararajan, A., & Sarkar, S. (2018). **Monitoring with Prometheus**: *Prometheus: Up & Running*. O'Reilly Media.
7. Barciś, S., & Kowalewski, T. (2020). **Grafana Observability**: *Grafana Observability*. Packt Publishing.
8. Gormley, C., & Tong, Z. (2015). **Elasticsearch: The Definitive Guide**: *Elasticsearch: The Definitive Guide*. O'Reilly Media.
9. Gialluca, S., & Lancini, M. (2016). **The Logstash Book**: *The Logstash Book*. Logstash Book.
10. Penton, A. (2019). **Kibana Essentials**: *Kibana Essentials*. Packt Publishing.
11. Pothulapati, A. (2018). **Docker on Windows**: *Docker on Windows*. Packt Publishing.
12. Luksa, M. (2018). **Keycloak: Securing Your Apps with OAuth2 and OpenID Connect**: *Keycloak: Securing Your Apps with OAuth2 and OpenID Connect*. O'Reilly Media.
13. Grails, G., & Rocher, G. (2010). **The Definitive Guide to Thymeleaf**: *The Definitive Guide to Thymeleaf*. Apress.

Online Resources:

1. *Spring Boot official documentation:* https://spring.io/projects/spring-boot
2. *Gradle official documentation:* https://gradle.org/
3. *PostgreSQL official documentation:* https://www.postgresql.org/docs/
4. *Liquibase official documentation:* https://www.liquibase.org/
5. *Prometheus official documentation:* https://prometheus.io/docs/
6. Grafana official documentation: https://grafana.com/docs/
7. Elasticsearch official documentation: https://www.elastic.co/guide/index.html
8. Logstash official documentation: https://www.elastic.co/guide/en/logstash/current/index.html
9. Kibana official documentation: https://www.elastic.co/guide/en/kibana/current/index.html
10. Docker official documentation: https://docs.docker.com/
11. Keycloak official documentation: https://www.keycloak.org/documentation.html
12. Thymeleaf official documentation: https://www.thymeleaf.org/documentation.html