MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
NATIONAL AVIATION UNIVERSITY
Faculty of cybersecurity and software engineering
Software engineering department

ADMIT TO DEFENCE
Head of the department
_______ Oleksii GORSKI
“____”_________ 2023

GRADUATE WORK

(EXPLANATORY NOTE)

GRADUATE OF EDUCATIONAL MASTER'S DEGREE

Theme: “Methodology and application of full-stack software production improvement”

Performer: Zhdanov Vladyslav Oleksandrovych

Supervisor: Ph.D. Associate Professor Tereshchenko Lidiia Yurievna

Standard controller: Mykhailo OLENIN

KYIV 2023

NATIONAL AVIATION UNIVERSITY

Faculty of cybersecurity and software engineering
Department Software Engineering
Education degree: master
Speciality: 121 Software engineering
Educational-professional program: Software engineering

APPROVED
Head of the Department
__________ Oleksii GORSKI
"___" ____________ 2023

Task
on executing the graduation work
Zhdanov Vladyslav Oleksandrovych

1. Topic of the graduation work: «Methodology and application of full-stack software production improvement».
Approved by the order of the rector from 29.09.2023 № 1994/ст.

2. Terms of work execution: from 05.09.2023 to 20.12.2023.

3. Source data of the work: android app and client-server architecture on local machine using IDE Android Studio, IntelliJ IDEA and programming language Kotlin

4. Content of the explanatory note:
- Analysis of full-stack software production using Kotlin programming language
- Methodology and Application for improving full-stack software production
- Analysis and comparison Kotlin with other programming languages

5. List of presentation mandatory slides:
- Distributed processing in client-server architecture
- Functional structure of mobile app
- Functional structure of backend server
- User interface of mobile app
- Example of prototype work.

6. Calendar schedule

№	Task	Execution term	Execution mark
1.	Acquainting oneself with the problem statement and reviewing relevant literature constitute the initial phase. The subsequent step involves drafting Section 1 and presenting it to the supervisor.	14.10.2023	
2.	Drafting the preprint for Section 1 and accompanying pages involves outlining the title, objective, schedule, abstract, list of abbreviations, content, introduction, and bibliography. This process is followed by an initial standard review.	14-27.10.2023	
3.	Writing 2 section, presentation to the supervisor	23.09.2023-10.10.2023	
4.	Writing 3 section, presentation to the supervisor	10.10.23-10.11.23	
5.	Conducting comprehensive editing and preparing the explanatory note, along with the graphical materials, for the purpose of finalization and printing.	10.11.23-30.11.23	
6.	Passing standard control	10.12.2023	
7.	Elaboration of the report's textual content and generation of visual materials intended for presentation purposes.	13.12.2023	
8.	Get feedback from the supervisor, reviews.	14.12.2023	
9.	Compilation of documentation for submission to the Departmental Examination Committee (DEC) secretary, encompassing software materials, graphic materials (GM), CD-R containing electronic software copies and presentations, supervisor's evaluation, progress certificate, and the organization into two folders and envelopes.	20.12.2023	
10.	Graduation project presentation	27.12.2023	

Date of issue of the assignment 05.09.2023
Supervisor: _____________ Lidiia TERESHCHENKO
Task accepted for execution: _________ Vladyslav ZHDANOV

РЕФЕРАТ

Пояснювальна записка до теми «Методологія та застосування вдосконалення виробництва програмного забезпечення повного стеку»: 94 с., 13 рис., 1 таблиця, 10 використаних джерел, 2 додатки.

Об'єкт дослідження – «Методологія та застосування вдосконалення виробництва програмного забезпечення повного стеку», зосереджена виключно на використанні мови програмування Kotlin для розробки як фронтенду, так і бекенду при створенні комплексних програмних рішень.

Мета роботи - аналізувати ретельно роль Kotlin як виключної мови в розробці програмного забезпечення повного стеку. Мета полягає в наголошенні переваг, адаптивності та ефективності Kotlin у впорядкуванні процесів виробництва програмного забезпечення, демонструючи його потенціал для оптимізації розробки інтегрованих програмних рішень.

Тип розробки - тип розробки, який досліджується в цьому дослідженні, включає уніфікований підхід з використанням Kotlin як єдиної мови програмування для створення компонентів програмного забезпечення як на фронтенді, так і на бекенді, відомий як розробка повного стеку.

Припущення про розвиток інструментів - Припущення передбачає, що використання Kotlin як виключної мови суттєво покращить ефективність та взаємодію розробки програмного забезпечення повного стеку. Очікується, що це покращить співпрацю, зменшить час розробки і в кінці кінців сприятиме розвитку інструментів розробки програмного забезпечення, призначених для безперешкодного досвіду розробки повного стеку.

РОЗРОБКА ПОВНОГО СТЕКУ, KOTLIN, ANDROID SDK, KTOR, JAVA, JAVASCRIPT, ЕФЕКТИВНІСТЬ, ПОРІВНЯННЯ МОВ ПРОГРАМУВАННЯ.

ABSTRACT

Explanatory note to the topic «Methodology and application of full-stack software production improvement»: 94 p., 13 fig., 1 table, 10 information sources, 2 appendixes.

The object of research is – the «Methodology and Application of Full-stack Software Production Improvement», focusing exclusively on utilizing the Kotlin programming language for both frontend and backend development in creating comprehensive software solutions.

The purpose of the thesis – aims to thoroughly analyze the role of Kotlin as the exclusive language in full-stack software development. It seeks to emphasize the advantages, adaptability, and efficiency of Kotlin in streamlining software production processes, demonstrating its potential to optimize the development of integrated software solutions.

Development type – The development type investigated in this research involves a unified approach using Kotlin as the sole programming language to create both frontend and backend software components, known as full-stack development.

The predicted assumption about the development of tools – The predicted assumption is that leveraging Kotlin as the exclusive language will significantly improve the efficiency and coherence of full-stack software development. It is expected to enhance collaboration, reduce development time, and ultimately contribute to the advancement of software development tools tailored for a seamless full-stack development experience.

FULL-STACK DEVELOPMENT, KOTLIN, ANDROID SDK, KTOR, JAVA, JAVASCRIPT, EFFICIENCY, COMPARISON OF PROGRAMMING LANGUAGES.

TABLE OF CONTENTS

LIST OF ACRONYSM AND ABBRREVIATIONS

JS	– Abbreviation for JavaScript, a commonly used programming language for web development.
API	– Application Programming Interface, used in the context of backend development to communicate between different software systems.
HTML	– HyperText Markup Language, used for creating web pages and web applications.
CSS	– Cascading Style Sheets, used for styling web pages written in HTML.
UI	– User Interface, referring to the visual elements of a software application that users interact with.
IDE	– Integrated Development Environment, a software application that provides comprehensive facilities to programmers for software development.
JVM	– Java Virtual Machine, a crucial component enabling the execution of Kotlin and Java programs.
HTTP	– Hypertext Transfer Protocol, the foundation of data communication for the World Wide Web.
REST	– Representational State Transfer, an architectural style for distributed systems such as the World Wide Web.
OOP	– Object-Oriented Programming, a programming paradigm based on the concept of "objects".
CRUD	– Create, Read, Update, Delete, representing the four basic functions of persistent storage.
CI/CD	– Continuous Integration/Continuous Deployment, a set of principles and practices to deliver code changes more frequently and reliably.

INTRODUCTION

The burgeoning demands of modern software development necessitate a meticulous examination and refinement of methodologies and tools employed in the production of software applications. In response to this imperative, this practice report delves into the comprehensive exploration of enhancing full-stack software production through the exclusive use of the Kotlin programming language. Kotlin, recognized for its conciseness, safety, and versatility, has emerged as a promising language for a unified approach encompassing both frontend and backend development.

This report embarks on a structured analysis of the subject area by delineating the fundamental characteristics of software processes in the context of employing Kotlin for full-stack software production. Through an exhaustive examination of literary sources and existing analogues, we aim to glean insights into the extant landscape and identify prominent frameworks and applications that underscore Kotlin's efficacy in the realm of full-stack development.

Furthermore, we deliberate on the relevance of adopting Kotlin as a singular language for full-stack software production, given the contemporary technological landscape and evolving industry trends. The integration of Kotlin into the full-stack development process holds promise for elevating productivity, ensuring code robustness, and enhancing the overall quality of software solutions. This report seeks to shed light on the practicality and benefits of utilizing Kotlin, elucidating its role as a vital tool for the advancement and optimization of software production methodologies.

In pursuit of a comprehensive understanding and practical application, the subsequent sections will delve into detailed analyses, case studies, and discussions, culminating in informed conclusions that contribute to a broader comprehension of the methodology and application of Kotlin in the domain of full-stack software production.

CHAPTER 1.

ANALYSIS OF FULL-STACK DEVELOPMENT IN THE CONTEXT OF MOBILE APPLICATION DEVLOPMENT AND KOTLIN PROGRAMMING LANGUAGE

1.1 The concept of full-stack development

1.1.1 What Does Full Stack Development Entail?

Full stack development refers to the comprehensive process of conceptualizing, constructing, testing, and deploying an entire web application from its inception to completion. This multifaceted approach encompasses the utilization of diverse technologies and tools, spanning front-end web development, back-end web development, and database development. The term "full stack development" characterizes a software engineer or developer proficient in both the front and back end of a website or application. A full-stack developer possesses the ability to seamlessly navigate and contribute to the technologies underpinning a website or application.

The responsibilities of full-stack developers often extend across the entirety of the web application development lifecycle, demanding a comprehensive grasp of the technologies and tools inherent to web development. Effective collaboration within a team is essential, given that web development is typically a cooperative undertaking. Most full-stack developers possess a solid foundation in fundamental web development technologies such as HTML, CSS, and JavaScript. They are also well-versed in server-side technologies like PHP, Ruby on Rails, and Node.js. Beyond technical proficiency, full-stack developers possess a profound understanding of how the various components of a website or application interact.

Due to their ability to oversee the entire process of building a website or application and promptly address any issues that may arise, full-stack developers are in high demand. When considering hiring a full-stack developer, it is advisable to inquire about their expertise in both front-end and back-end technologies.

1.1.2 Client Software (Front End)

User interface software, commonly referred to as the front end, is a category of software designed to engage with users directly. Its primary function involves creating and managing the graphical user interface (GUI) that users observe and interact with. This software facilitates users' access to and utilization of the features and functionalities embedded in the underlying software or system. Unlike residing on a remote server, client software typically operates on the user's local machine.

In numerous instances, the client software plays a pivotal role in shaping the user's overall experience with a specific system or application.

Languages for Front-End Development

Various languages serve the purpose of front-end development, such as HTML, CSS, and JavaScript. Each language possesses distinct strengths and weaknesses, necessitating a careful selection based on the specific task at hand.

- HTML, the most fundamental of the trio, is utilized to structure content within a web page.
- CSS comes into play for styling the content on a web page, enabling the creation of sophisticated layouts.
- JavaScript, known for introducing interactivity, is employed to infuse dynamic content into a web page.

Front-End Frameworks and Libraries

A plethora of front-end frameworks and libraries is available to developers in the contemporary landscape. Among the widely adopted options are React, Angular, and

Vue. Each framework boasts its own merits and demerits, making the selection crucial for project success.

- React garners popularity due to its user-friendly nature and ease of learning.
- Angular stands out as a robust choice for substantial projects owing to its feature-rich environment.
- Vue, prized for its lightweight nature and simplicity, proves beneficial for smaller-scale projects.

Regardless of the chosen front-end framework, a prerequisite is familiarity with its nuances before embarking on a project.

Beyond the frameworks, several utility libraries complement development endeavors. Examples include Lodash, Moment.js, and Axios. Once again, judicious selection of a library tailored to the project's requirements is imperative, given the unique capabilities each one brings to the table.

1.1.3 Server Software (Back End)

Backend software, also known as server-side software, assumes the responsibility of overseeing and orchestrating server activities. It ensures the continuous operation of the server and verifies the proper functioning of its diverse components. Additionally, server software furnishes an interface for user-server interaction and facilitates server management for administrators.

Prominent server software programs encompass Apache HTTP Server, Microsoft IIS, and Nginx. These applications are instrumental in executing server-side tasks such as web page hosting, handling user requests, and delivering responses.

To operate a website or application, a server equipped with server software is imperative. This software enables users to engage with your site or application. The absence of server software would render users unable to request data or information from your site.

Server software constitutes a vital component of any website or application, demanding a foundational understanding for those seeking to manage a website or application.

Backend Programming Languages

A variety of backend languages are available for developing websites or applications, with PHP, Java, Python, and Ruby being among the most popular. Each language possesses its own strengths and weaknesses, necessitating careful selection based on project requirements.

PHP proves suitable for smaller projects with straightforward functionality, while Java excels in larger projects demanding intricate functionality.

Python is a preferred choice for projects involving extensive data processing, whereas Ruby is well-suited for projects requiring substantial user interaction.

Backend Frameworks and Libraries

Diverse backend frameworks and libraries are accessible to developers in the contemporary landscape. These resources find application in developing web applications, mobile applications, and desktop applications. Notable examples include Ruby on Rails, Laravel, and Node.js. Each framework and library brings its own set of advantages and disadvantages, underscoring the importance of a meticulous evaluation of project needs before settling on a backend framework or library.

1.1.4 Popular Development Stacks

Developers commonly employ several popular stacks in their work.

The MEAN stack stands out as one of the most popular, specifically tailored for web application development. Comprising MongoDB, Express.js, AngularJS, and Node.js, this stack offers a comprehensive solution for building dynamic and scalable web applications.

Another widely used stack is the LAMP stack, designed for server-side applications, encompassing Linux, Apache, MySQL, and PHP.

1.1.5 Advantages and Disadvantages of Full Stack Development

Advantages

Full-stack development boasts numerous advantages:

One of the primary benefits is the holistic understanding it affords developers of the entire web development process. This comprehensive knowledge enhances development efficiency and effectiveness.

Full-stack development contributes to overall efficiency by enabling developers to identify potential bottlenecks and issues more easily. This proactive approach helps prevent development problems, resulting in time and cost savings.

A cohesive and user-friendly experience is another advantage, as full-stack developers, well-versed in both front-end and back-end processes, can create a unified user interface that enhances customer satisfaction and loyalty.

In summary, full-stack development offers multiple benefits, fostering efficient, effective, and user-friendly web development.

Disadvantages

Despite its advantages, full-stack development comes with potential drawbacks:

One concern is the broad skill set required, which may lead to a lack of specialization in any specific area. This could pose challenges for projects demanding intense expertise in a particular domain.

Full-stack developers may spend more time staying abreast of new developments due to the need to work with various technologies, compared to specialists focusing on a single area.

The complexity and time-consuming nature of full-stack development make it less suitable for small projects or tight deadlines.

In conclusion, while full-stack development offers versatility and comprehensive knowledge, careful consideration is necessary to navigate potential challenges and drawbacks.

Conclusion

In conclusion, the realm of development is a dynamic landscape that involves intricate processes on both the front end and the back end. The frontend, or client software, interfaces directly with users, creating a visually appealing and interactive experience. HTML, CSS, and JavaScript serve as the foundational languages, and frameworks like React, Angular, and Vue provide developers with powerful tools to enhance user interfaces.

On the other side of the spectrum lies the backend, or server software, which orchestrates the server's activities. It ensures the server's continuous operation, manages user interactions, and allows administrators to oversee server functions. The backend relies on languages such as PHP, Java, Python, and Ruby, each chosen based on project requirements. Frameworks like Ruby on Rails, Laravel, and Node.js provide developers with the infrastructure needed to build robust web, mobile, and desktop applications.

Examining popular development stacks, such as the MEAN stack and LAMP stack, reveals how developers strategically combine technologies to streamline the development process. The MEAN stack's MongoDB, Express.js, AngularJS, and Node.js synergize for dynamic web applications, while the LAMP stack's Linux, Apache, MySQL, and PHP cater to server-side applications.

Full-stack development emerges as a powerful approach, offering a comprehensive understanding of both frontend and backend processes. This holistic perspective enhances efficiency, as full-stack developers can identify and address potential issues seamlessly. However, the broad skill set required and the need to stay current with diverse technologies present challenges, making full-stack development less suitable for highly specialized or time-sensitive projects.

In the ever-evolving landscape of web development, the careful selection of languages, frameworks, and stacks plays a pivotal role in determining project success. Whether specializing in frontend or backend development or embracing the versatility of full-stack proficiency, developers must navigate the complexities of technology to deliver seamless, user-friendly, and efficient solutions to the digital world.

1.2 The concept of full-stack mobile application development

Full-stack mobile development represents a comprehensive approach that integrates both server-side and client-side components in the mobile app development process. Analogous to web and desktop app development, full-stack mobile development employs specialized programming languages such as Ionic and Meteor. Developers adept in this methodology strive to create scalable applications, necessitating meticulous research to strike the right balance across various microservice layers.

1.2.1 Conceptual Understanding

A nuanced comprehension of full-stack mobile development is pivotal for crafting mobile applications characterized by scalability and reliability. Proficient full-stack developers delve into the intricacies of databases and their interrelations, coupled with a proficiency in cloud technologies. Mastery of incorporating third-party connectors and APIs to augment applications is integral, notwithstanding the time-intensive integration process. The full-stack developer's adeptness in seamlessly merging API code with real code transforms projects from manual to automated processes.

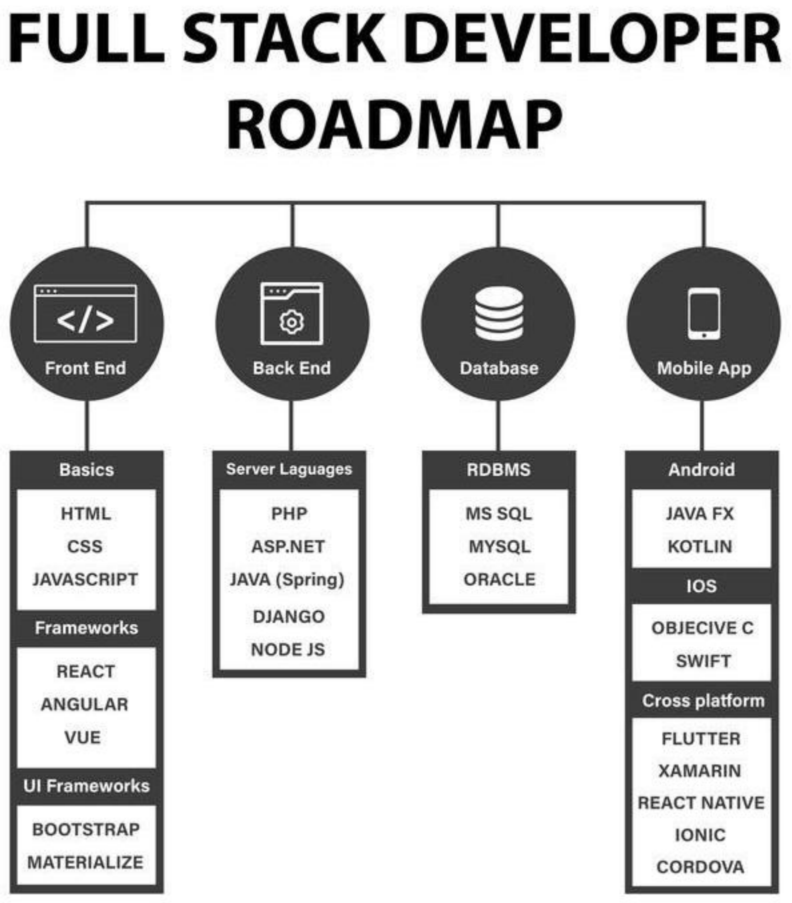


Fig. 1.1. Common paradigm of full-stack software development

Furthermore, the versatility of full-stack developers is evident as they seamlessly transition between front-end and back-end development tasks. Their adeptness extends to navigating complexities and troubleshooting issues in both developmental domains, assuring a comprehensive problem-solving capability.

1.2.3 Architectural Framework

The architecture of full-stack mobile development amalgamates server-side and client-side development principles, akin to desktop and web application development but without the confinement to a single operating system. Utilizing frameworks like Ionic or Meteor, developers create mobile applications employing version control and API calls for data interaction. Achieving scalability involves careful research into optimizing task distribution across diverse layers.

Employing a version control system is paramount for communication, management, and tracking changes in collaborative endeavors. This ensures cohesion among multiple team members located in different settings, working on distinct functionalities. A version control system delineates separate branches for contributors, ensuring a systematic approach until thorough analysis is completed.

1.2.4 Technological Landscape

Full-stack mobile development technologies amalgamate various tools, incorporating native and hybrid apps for both iOS and Android platforms. This technology stack, featuring Cordova, Ionic, and Xamarin, facilitates the creation of cross-platform apps. Ideal for developers targeting a diverse user base, these technologies offer flexibility and accessibility.

Native mobile applications, designed explicitly for specific platforms such as Android and iOS, optimize user experience through seamless integration with the operating system. Leveraging Java as a primary language, native applications capitalize on a shallow learning curve, an extensive library ecosystem, and platform independence.

1.2.5 Benefits of Full Stack Mobile Development

Proficiency in the latest technologies and the ability to navigate every facet of a project, from the minimum viable product (MVP) to quality testing, distinguish full-stack developers. Their acumen extends to staying abreast of the latest programming trends and application development strategies, engendering trust in their capability to manage projects without incurring additional fees for additional team members.

Full-stack developers showcase their prowess in synchronizing multiple data inputs, exemplified in transforming an existing website into a mobile application. This adaptive capability extends beyond mobile applications to the redesign of existing websites, showcasing the versatility and holistic skill set of full-stack developers.

1.2.6 Disadvantages of Full Stack Mobile Development

While full-stack mobile development presents numerous advantages, certain disadvantages merit consideration. Firstly, the extensive breadth of skills required may lead to a lack of depth in specialized areas, potentially posing challenges for projects demanding an intense level of expertise in a specific domain. Additionally, the need for full-stack developers to adapt to a variety of technologies may necessitate continuous efforts to stay abreast of new developments, contrasting with specialists who can focus on a singular domain.

Furthermore, the complexity and time-consuming nature of full-stack development may render it less suitable for smaller projects or those with tight deadlines. The intricacies involved in managing both frontend and backend aspects could potentially result in longer development cycles, emphasizing the importance of aligning the choice of development methodology with project requirements and constraints.

1.2.7 Comparing full-stack software development with standard software development

A simplified comparing full-stack software development with standard software development across various types of applications, shown in Table 1.1. The advantages and disadvantages can vary based on specific project requirements and team dynamics.

Table 1.1.

Comparing full-stack software development with standard software development

Aspect	Full-Stack Software Development	Standard Software Development
Scope of Expertise	Advantages: - Proficiency in both frontend and backend. - Versatility to handle end-to-end development. Disadvantages: - Potential lack of depth in specialized areas. - Continuous learning to keep up	Advantages: - Specialized expertise in either frontend or backend. Disadvantages: - Potential lack of depth in specialized areas. - Limited scope across frontend or backend.

	with technologies. - Time-consuming for in-depth expertise	

Table 1.1. (continue)

Development Efficiency	Advantages: - Ability to identify and address issues swiftly. - Streamlined project understanding and workflow. Disadvantages: - Complexity and potential longer development cycles due to dual responsibilities. - May not be ideal for smaller projects or tight deadlines.	Advantages: - Focused development in a specific area may lead to faster mastery. Disadvantages: - Potential communication challenges between frontend and backend teams.
Flexibility and Adaptability	Advantages: - Seamless transition between frontend and backend. - Adaptable to address complexities in both areas.	Advantages: - Clearly defined roles and responsibilities. Disadvantages: - May lack flexibility in

	Disadvantages: - Limited specialization may pose challenges in highly specialized projects. - Continuous adaptation to evolving technologies.	addressing issues requiring cross-domain expertise.

Table 1.1. (continue)

Cost and Resource Management	Advantages: - Reduced need for hiring specialists in both frontend and backend. - Single developer managing multiple tasks. Disadvantages: - Potential higher costs due to broader skill set. - Resource allocation challenges for complex projects.	Advantages: - Focused hiring of specialists may optimize resource allocation. Disadvantages: - May require hiring specialists for each phase. - Potential inefficiencies if roles and tasks are not clearly defined.
Project Type Suitability	Advantages: - Versatile for a variety of project types. Disadvantages:	Advantages: - Ideal for projects with clear frontend or backend requirements.

	- May face challenges in highly specialized domains that demand specific expertise.	Disadvantages: - May struggle with projects requiring in-depth expertise in either frontend or backend.

Conclusion

In conclusion, full-stack mobile development emerges as a comprehensive and versatile approach to mobile app development, seamlessly integrating server-side and client-side components. Drawing parallels with web and desktop application development, full-stack mobile development employs specialized programming languages, exemplified by Ionic and Meteor, with the aim of crafting scalable and reliable applications. The multifaceted nature of this methodology necessitates thorough research to achieve an optimal balance across diverse microservice layers.

A nuanced understanding of full-stack mobile development is essential for proficient developers seeking to create applications characterized by both scalability and reliability. Mastery of database intricacies, proficiency in cloud technologies, and the seamless integration of third-party connectors and APIs exemplify the depth of knowledge required. The adeptness of full-stack developers in seamlessly transitioning between front-end and back-end tasks further underscores their comprehensive problem-solving capabilities.

The architectural framework of full-stack mobile development mirrors principles employed in desktop and web application development, offering the flexibility to transcend the confines of a single operating system. The use of frameworks such as Ionic or Meteor, coupled with version control and API calls, contributes to the creation of scalable mobile applications. The systematic approach facilitated by a version control system ensures cohesive collaboration among team members, even when working on distinct functionalities.

Technologically, the full-stack mobile development landscape combines native and hybrid apps for both iOS and Android platforms. Technologies such as Cordova, Ionic, and Xamarin empower developers to create cross-platform applications, catering to a diverse user base. The utilization of Java as a primary language in native applications further enhances user experience through seamless integration with the operating system.

The benefits of full-stack development lie in the proficiency of developers in the latest technologies, their ability to navigate every project facet from the MVP to quality testing, and their continuous awareness of evolving programming trends. Full-stack developers exhibit prowess in synchronizing multiple data inputs, enabling them to transform existing websites into mobile applications and undertake the redesign of websites with ease.

However, as with any methodology, full-stack mobile development is not without its disadvantages. The expansive skill set required may result in a lack of depth in specialized areas, posing challenges for projects demanding intense expertise in specific domains. The need for continuous adaptation to diverse technologies may demand ongoing efforts to stay abreast of new developments, contrasting with specialists who can focus on singular domains. Moreover, the complexity and time-consuming nature of full-stack development may render it less suitable for smaller projects or those operating under tight deadlines.

In conclusion, while full-stack mobile development offers a holistic and adaptable approach, its advantages and disadvantages must be carefully weighed against the specific requirements and constraints of each project. A judicious alignment of the chosen development methodology with project goals ensures optimal outcomes in the dynamic landscape of mobile application development.

1.3 Features of Kotlin programming language

The acquisition of proficiency in the Kotlin programming language is motivated by several compelling factors. Kotlin stands as one of the rapidly expanding and extensively adopted programming languages, renowned for amalgamating superior object-oriented and functional programming features. This amalgamation renders Kotlin an attractive choice for software developers worldwide, who find its user-friendly nature and ease of comprehension conducive to various project implementations. Notably, Kotlin exhibits a notable ease of maintenance and debugging, enhancing its appeal as an entry-level language.

An important milestone in the endorsement of Kotlin was Google's official declaration in May 2017 designating it as the language of choice for Android application development. Subsequent to this proclamation, the job market witnessed a substantial surge in Kotlin-related positions, with opportunities doubling every three months, according to Dice. This surge in demand underscores Kotlin's rising prominence, with major industry players such as Google, Netflix, and Pinterest actively utilizing the language.

Kotlin, developed by Jet Brains, is characterized as a general-purpose, statically typed programming language. It seamlessly integrates features from both object-oriented and functional programming paradigms, and its interoperability with Java enables the sharing and utilization of information between the two languages. Notably, Kotlin generates bytecode akin to the Java compiler, facilitating its execution on the Java Virtual Machine (JVM).

The versatility of Kotlin extends across various domains, including the development of server-side applications, Android applications, and multiplatform mobile development. Noteworthy is its compatibility with Java libraries, allowing developers to leverage existing resources seamlessly. While Java, as a language, possesses commendable attributes, the developers of Kotlin directed their efforts toward code simplification and enhanced transparency, positioning Kotlin as a refined counterpart with additional features.

1.3.1 A Historical Overview of Kotlin

The inception of the Kotlin programming language was officially declared in 2011. Subsequently, in 2012, Kotlin marked its initial web demonstration, accompanied by the introduction of a new logo, and transitioned into an open-source language. The year 2014 witnessed the launch of Kotlinlang.org, featuring enhancements designed to augment interoperability with Java. In 2015, additional functionalities, such as companion objects and multiple constructors, were incorporated into the language.

The pivotal release of Kotlin version 1.0 transpired in 2016, emanating from the efforts of JetBrains in Russia. The impetus behind Kotlin's development stemmed from the dissatisfaction of JetBrains developers with repetitive code in Java. Confronted with the challenge of pre-existing Java codebases, the team aspired to formulate a contemporary language that seamlessly aligned with Java while encompassing desired features.

A seminal moment occurred in 2017 when Google proclaimed official support for Kotlin on Android, setting the stage for subsequent releases. The evolution continued with Kotlin 1.2, introducing capabilities such as code sharing between JVM and

JavaScript. Subsequent iterations followed, with Kotlin 1.3 in 2018, Kotlin 1.4 in 2020, and Kotlin 1.5 in 2021.

1.3.2 Features of Kotlin: An In-depth Analysis

Kotlin encompasses a myriad of features that contribute to its versatility and efficacy:

- Platform-agnostic Usage: Kotlin facilitates cross-platform utilization, mitigating the need for redundant code creation and maintenance across diverse platforms. It operates independently of the virtual machine on the target platform.
- Modern Language Features: Noteworthy features, such as null safety, Lambda functions, and Smart casts, enhance the language's expressiveness and address issues associated with null references.
- Extension Functions: Kotlin supports extension functions, enabling the addition of functionality without resorting to class inheritance. This characteristic promotes code readability and ease of maintenance.
- Higher-Order Functions: The language allows functions to be passed as parameters, signifying a higher-order function paradigm. This capability extends the versatility of functions by treating them akin to variables.
- Interoperability with Java: Kotlin seamlessly integrates with Java, enabling the conversion of Java files to Kotlin through a simple script. Furthermore, Kotlin operates on the JVM, ensuring compatibility with existing Java libraries.
- Data Class: Kotlin introduces the concept of data classes—classes devoid of operational methods, solely focused on encapsulating state. The inherent advantage lies in the automatic generation of code, alleviating the need for manual method implementation.

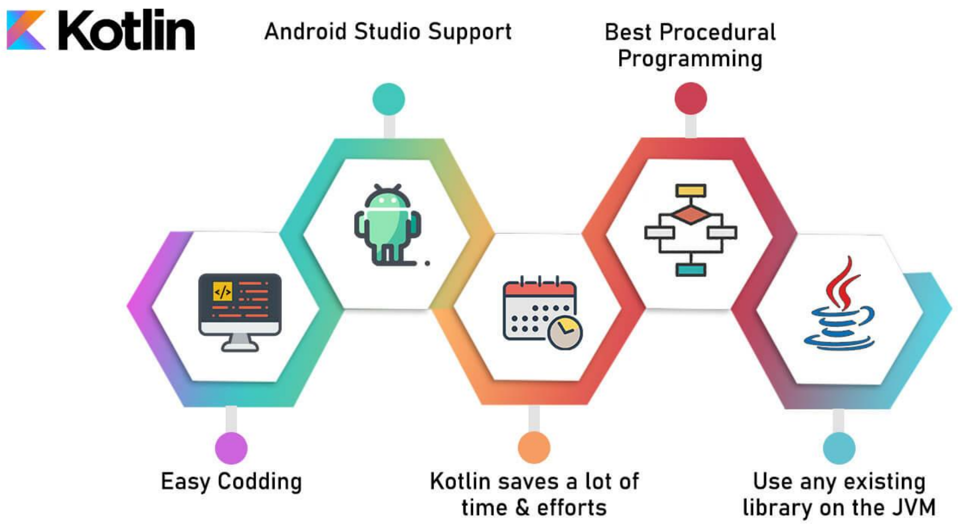

Kotlin
Android Studio Support
Best Procedural Programming
Easy Codding
Kotlin saves a lot of time & efforts
Use any existing library on the JVM

Fig. 1.2. Kotlin benefits

1.3.3 Applications of Kotlin

Kotlin finds application across diverse domains:

- Web Development: Kotlin emerges as a viable alternative for web development, demonstrating compatibility with Java and offering enhanced simplicity compared to Java. Its seamless integration with frameworks like Spring further reinforces its suitability.
- Data Science: The language's attributes, including null safety, static typing, maintainability, and JVM compatibility, position Kotlin as a proficient choice for data science applications.
- Android Development: Kotlin has established prominence in Android development since Google's endorsement. Its appeal lies in its expressiveness, conciseness, and seamless interoperability with Java, motivating a substantial community of developers to adopt Kotlin for Android app development.

In summary, the history of Kotlin reflects a trajectory marked by continuous development and strategic enhancements, aligning with the evolving needs of the software development landscape. The language's features and applications underscore its adaptability and utility across diverse domains, solidifying its position as a prominent player in the contemporary programming paradigm.

1.3.4 Utilization of the Kotlin Language by Corporations

Several prominent companies have embraced the Kotlin programming language, leveraging its capabilities to enhance various aspects of their software development endeavors. Notable instances of such adoption include:

- Airbnb, an American enterprise specializing in an online marketplace for lodging and homestays, integrates the Kotlin language into its Android framework, specifically within the MvRx framework. This strategic incorporation is intended to optimize the mobile user experience, ensuring seamless interactions.
- Google, a global technology giant, not only acknowledges Kotlin as a first-class programming language but has also extensively integrated it into their production code. With over 60 applications, including prominent ones like Google Homes, Google Maps, Google Drive, Google Pay, and Google Sheets, Google has solidified its commitment to Kotlin across diverse domains of application development.
- Zomato, recognized as India's foremost food delivery company, has adopted Kotlin for the development of its Android application. This choice of programming language has proven instrumental in streamlining code, resulting in a reduction of code lines and enhancing code conciseness for improved application performance.
- Netflix, the world's leading streaming service, opted for the Kotlin language to reconstruct the user interface (UI) player within the Android application. This strategic implementation signifies a deliberate choice to harness Kotlin's

capabilities for enhancing the user experience and ensuring the robustness of their Android app.

- Uber, an American mobility company offering a diverse array of services, ranging from ride-hailing and food delivery to package delivery and couriers, aligns with the popularity of Kotlin in the realm of Android development. The language's interoperable nature with Java aligns seamlessly with Uber's diverse software requirements, contributing to its adoption.

In summation, the aforementioned companies represent a subset of enterprises that have recognized and embraced the advantages offered by the Kotlin language, incorporating it strategically into their development processes for improved efficiency and enhanced user experiences.

Conclusion

In conclusion, the exploration of Kotlin, both in terms of its historical trajectory and its contemporary applications in corporate settings, reveals a dynamic and versatile programming language that has earned its place as a prominent player in the software development landscape. The historical evolution of Kotlin, from its announcement in 2011 to its designation as the official language for Android development by Google in 2017, showcases a deliberate and strategic response to the challenges faced by developers at JetBrains. The commitment to creating a language that combines the strengths of object-oriented and functional programming while ensuring compatibility with Java has resulted in a language that not only addresses the practical concerns of developers but also resonates with major industry players.

Examining the features of Kotlin in an academic light further reinforces its appeal. The language's platform-agnostic nature, modern features like null safety and extension functions, support for higher-order functions, and seamless interoperability with Java all contribute to its efficacy and versatility. These features, coupled with the adoption by corporations like Airbnb, Google, Zomato, Netflix, and Uber, underscore Kotlin's applicability across diverse domains, from web and Android development to data science.

The discussion on why one should learn Kotlin provides valuable insights into the burgeoning demand for Kotlin proficiency in the job market. The endorsement by Google has significantly propelled Kotlin into mainstream usage, leading to a substantial increase in job opportunities. The language's user-friendly attributes, ease of maintenance, and debugging further contribute to its status as an ideal starting point for developers entering the programming sphere.

In an academic exploration of Kotlin's definition, its origin as a general-purpose, statically typed language, its compatibility with Java, and its bytecode generation for the JVM emphasize its robust and adaptable nature. The acknowledgment of Kotlin as more than just a language, but rather a simplified and transparent alternative to Java, reveals a deliberate focus on enhancing code simplicity and clarity.

In essence, the journey through Kotlin's history, features, corporate applications, and educational aspects converges to depict a programming language that not only addresses technical requirements but also aligns with the broader industry trends, making it a compelling choice for developers and businesses alike in the dynamic landscape of modern software development.

1.4 Comprehensive examination of Kotlin's role in full-stack software production.

The extensive exploration of Kotlin's fundamental role in the landscape of full-stack software production constitutes a pivotal phase in advancing the domain. This analytical endeavor entails an all-encompassing investigation into the multifaceted aspects that govern the integration of Kotlin as the exclusive programming language in both frontend and backend development. The primary objectives encompass unraveling the contextual underpinnings, elucidating the essential definition of full-stack software development, comprehending Kotlin's inherent contribution to this paradigm, and meticulously scrutinizing the key characteristics inherent in the software processes that define the domain.

1.4.1 Understanding the Context

Understanding the contextual landscape in which Kotlin operates as the primary language for comprehensive software development is an essential prerequisite for an insightful analysis. Context here encapsulates the broader industry dynamics, technological trends, and evolving paradigms within software development. A nuanced understanding of these contextual factors is imperative to navigate and comprehend the intricate interplay of Kotlin in the realm of full-stack software production. This includes a deep dive into industry trends, emerging technologies, and shifts in user expectations that influence the choice and application of Kotlin as a full-stack language.

1.4.2 Role of Kotlin in Full-stack Development

Kotlin's role as the principal programming language in the ambit of full-stack development warrants an in-depth exploration (img. 2). This involves a thorough elucidation of its unique attributes, capabilities, and adaptability that render it a suitable choice for both frontend and backend development. The sub-paragraph delves into

Kotlin's conciseness, versatility, safety features, and rich ecosystem, underscoring how these aspects equip it to effectively serve the needs of both frontend and backend development. Furthermore, it explores the role of Kotlin in streamlining communication and collaboration between frontend and backend teams in the development process.

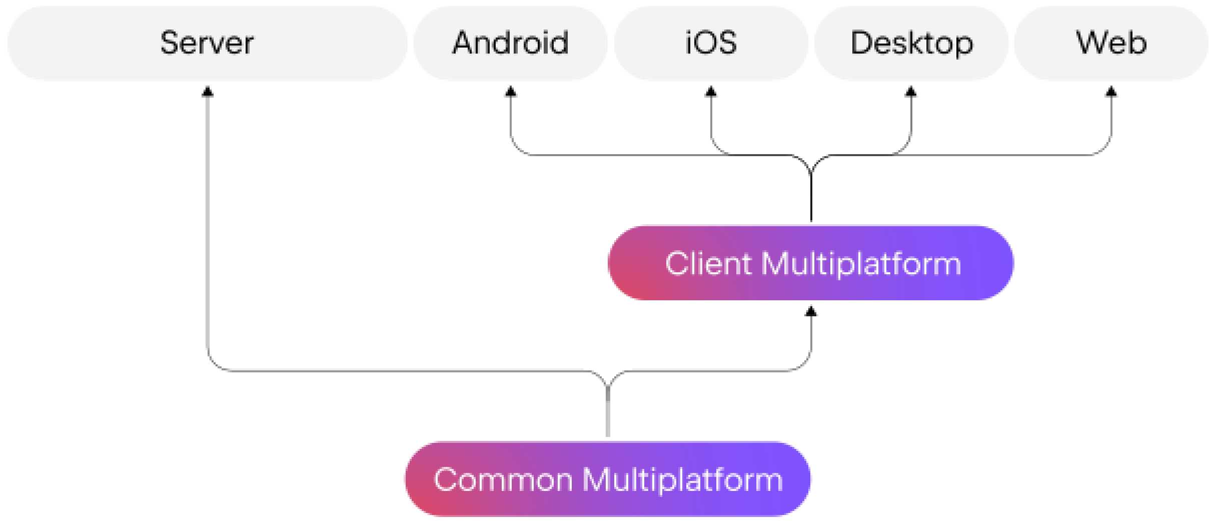


Fig. 1.3. Kotlin multiplatform

1.4.3 Main Characteristics of Software Processes

An incisive examination into the core characteristics governing the software processes in Kotlin-driven full-stack development forms the bedrock of this academic inquiry. These characteristics span the entire lifecycle of software development, encompassing methodologies, and essential processes intrinsic to frontend and backend domains. The sub-paragraph scrutinizes various development methodologies (e.g., Agile, Scrum) and their adaptation to Kotlin-based full-stack development. It also explores the significance of modularization, code reuse, and testing in the context of Kotlin, emphasizing how these aspects contribute to improved software processes and overall productivity in full-stack development.

Conclusion

The comprehensive analysis of Kotlin's role in full-stack software production underscores its foundational significance and transformative potential within the domain. This thorough exploration delved into the intricacies that position Kotlin as the exclusive programming language employed for both frontend and backend development. The primary objectives encompassed unraveling the contextual underpinnings of Kotlin's applicability, defining the fundamental nature of full-stack software development, explicating Kotlin's intrinsic contribution to this integrated paradigm, and meticulously scrutinizing the key characteristics inherent in the software processes defining this domain.

Understanding the contextual landscape within which Kotlin operates is paramount for insightful analysis. By encapsulating the broader industry dynamics, technological trends, and evolving paradigms within software development, a nuanced understanding was obtained. This includes a deep dive into industry trends, emerging technologies, and shifts in user expectations that influence the choice and application of Kotlin as a full-stack language.

Defining the paradigm of full-stack software development formed a foundational cornerstone of this analytical journey. This involved a meticulous elucidation of the integrated approach, seamlessly developing both frontend and backend components within a unified framework. The delineation provided a detailed account of the encompassed elements, architectural layers, and the symbiotic relationship between frontend and backend domains. Additionally, it fostered a comprehensive discussion on the challenges and advantages of full-stack development, shedding light on how Kotlin uniquely addresses these concerns.

The role of Kotlin as the principal programming language in the ambit of full-stack development was explored in-depth. This included a thorough elucidation of its unique attributes, capabilities, and adaptability, rendering it a suitable choice for both frontend and backend development. The sub-paragraph delved into Kotlin's conciseness, versatility, safety features, and rich ecosystem, underscoring how these aspects equip it to effectively serve the needs of both frontend and backend development. Furthermore, it explored Kotlin's role in streamlining communication and collaboration between frontend and backend teams in the development process.

An incisive examination into the core characteristics governing the software processes in Kotlin-driven full-stack development formed the bedrock of this academic inquiry. These characteristics spanned the entire lifecycle of software development, encompassing methodologies and essential processes intrinsic to both frontend and backend domains. The sub-paragraph scrutinized various development methodologies (e.g., Agile, Scrum) and their adaptation to Kotlin-based full-stack development. It also explored the significance of modularization, code reuse, and testing in the context of Kotlin, emphasizing how these aspects contribute to improved software processes and overall productivity in full-stack development. In conclusion, this analysis accentuates the pivotal role Kotlin plays in the advancement of full-stack software production methodologies and practices, illustrating its potential to revolutionize the landscape of software development.

CHAPTER 2.

METHODOLOGY AND APPLICATION FOR IMPROVING FULL-STACK SOFTWARE PRODUCTION USING KOTLIN

2.1 Developing a mobile application using the Android SDK and Kotlin

2.1.1 Critical Phases in the Mobile Application Development Process

The mobile application development journey typically encompasses seven pivotal stages, which we shall delve into in the ensuing discourse. While the specifics of the process may exhibit variations based on the unique requisites and objectives of individual projects, the fundamental sequence endures as outlined below:

Stage 1: Strategic Inception

Commencing with the strategy phase, commonly referred to as the discovery process, the initiation involves meticulous ideation. This stage necessitates the delineation of the product's objectives and a comprehensive examination of the prerequisites for app development. The project team systematically assimilates intelligence concerning market dynamics, competitors, and the intended user demographic. Concurrently, developers meticulously select the technological stack, platform, and application features aligned with the conceptualization. Simultaneously, designers conceive mobile app screens and articulate user experience (UX) concepts. An essential facet of this phase involves the estimation of the app's projected cost and the adoption of a monetization model conducive to sustainable profitability.

In this phase, it is imperative to emphasize the importance of conducting a thorough feasibility analysis to ascertain the viability and potential challenges associated with the proposed mobile app. An exhaustive exploration of emerging industry trends and technological advancements is integral to informed decision-making.

Stage 2: Analysis and Strategic Blueprinting

Advancing to the analysis and planning stage, the project team assimilates detailed insights into the mobile application's target market, potential user base, and functional requirements. In response to this wealth of information, a meticulous development plan is formulated, culminating in the creation of a comprehensive product roadmap.

The strategic blueprinting process should encompass a risk assessment and mitigation strategy, identifying potential hurdles and devising preemptive measures to ensure the project's smooth progression.

Stage 3: Artistry in Mobile App Design

The evolution of the mobile app's user interface (UI) and user experience (UX) design unfolds in this phase.

- User Experience Design:

Designers meticulously engineer the information architecture, delineating the arrangement and display of data and content within the application. Attention is devoted to optimizing user interaction for a seamless and intuitive experience.

- User Interface Design:

The creation of wireframes, serving as visual blueprints of the app's design structure, is an initial step. Concurrently, designers develop a style guide or design system encompassing aspects such as fonts, color schemes, and button configurations. Subsequently, designers craft mockups, including app screen designs, ensuring visual coherence and consistency.

- Prototyping:

The final phase within the design ambit involves the development of application design prototypes, offering a tangible preview of the envisioned user experience.

A human-centric design approach, grounded in usability studies and feedback loops, is paramount in ensuring the resonance of the mobile app with end-users.

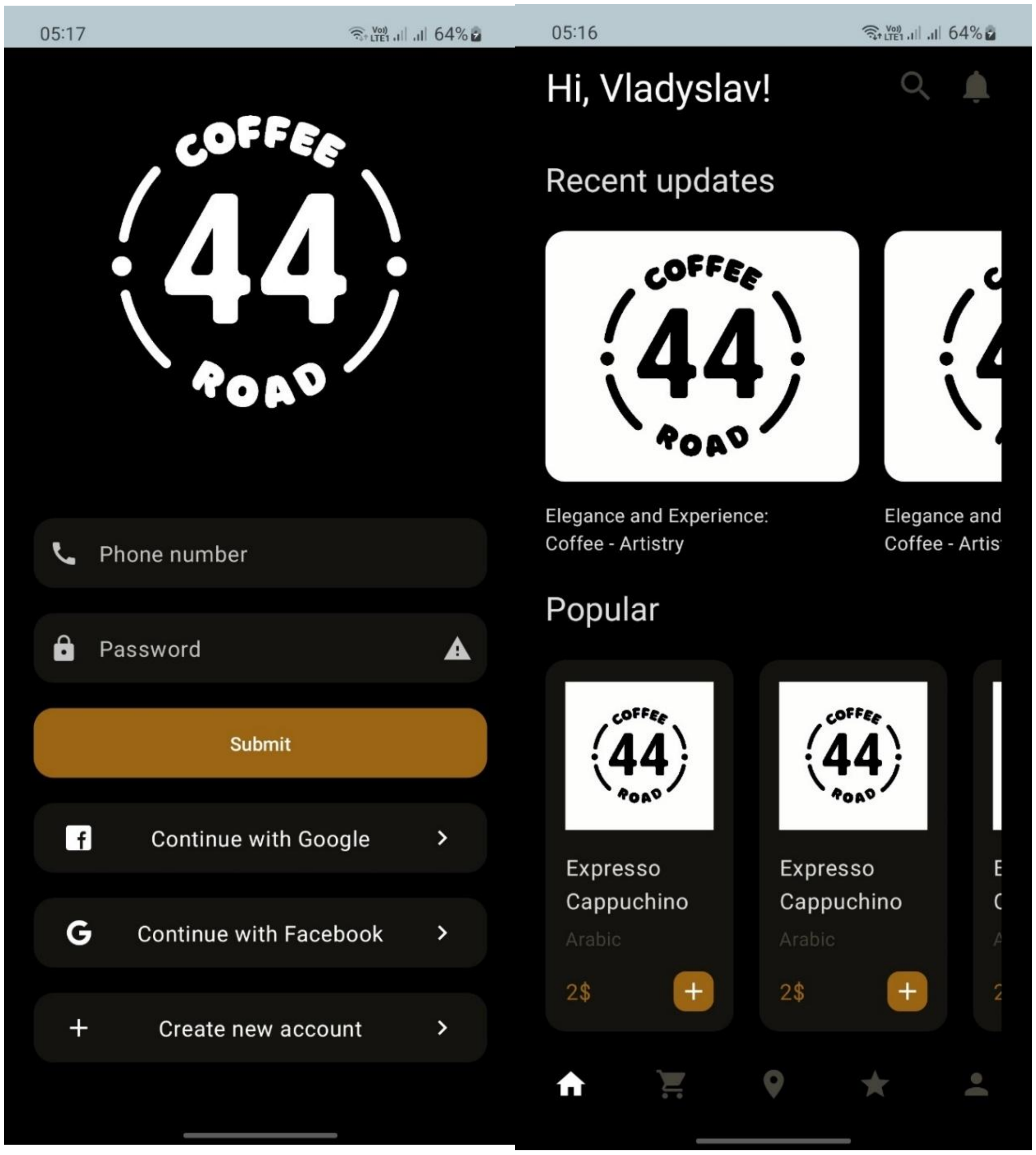


Fig. 2.1. Mobile app design

Stage 4: Development Process

The development stage witnesses the active engagement of developers who translate the envisioned features into reality employing the designated technology stack. This phase yields a fully realized front-end and back-end for the mobile application.

A collaborative approach between developers and designers is pivotal, facilitating seamless integration of design elements with the functional aspects of the application. Continuous integration and testing practices are imperative to identify and rectify potential issues during the development lifecycle.

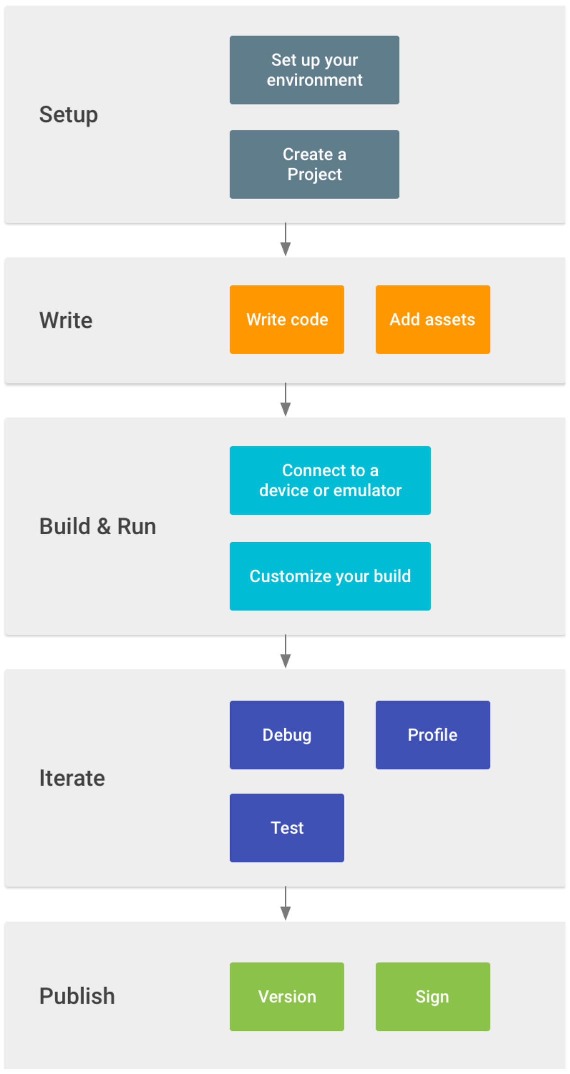
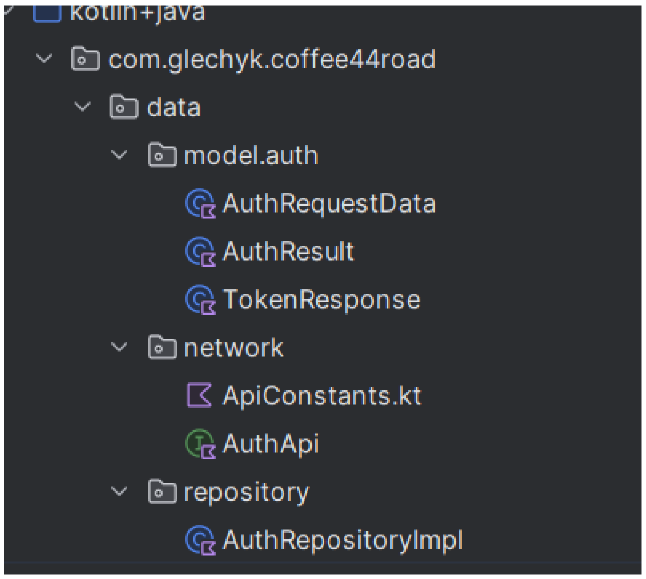
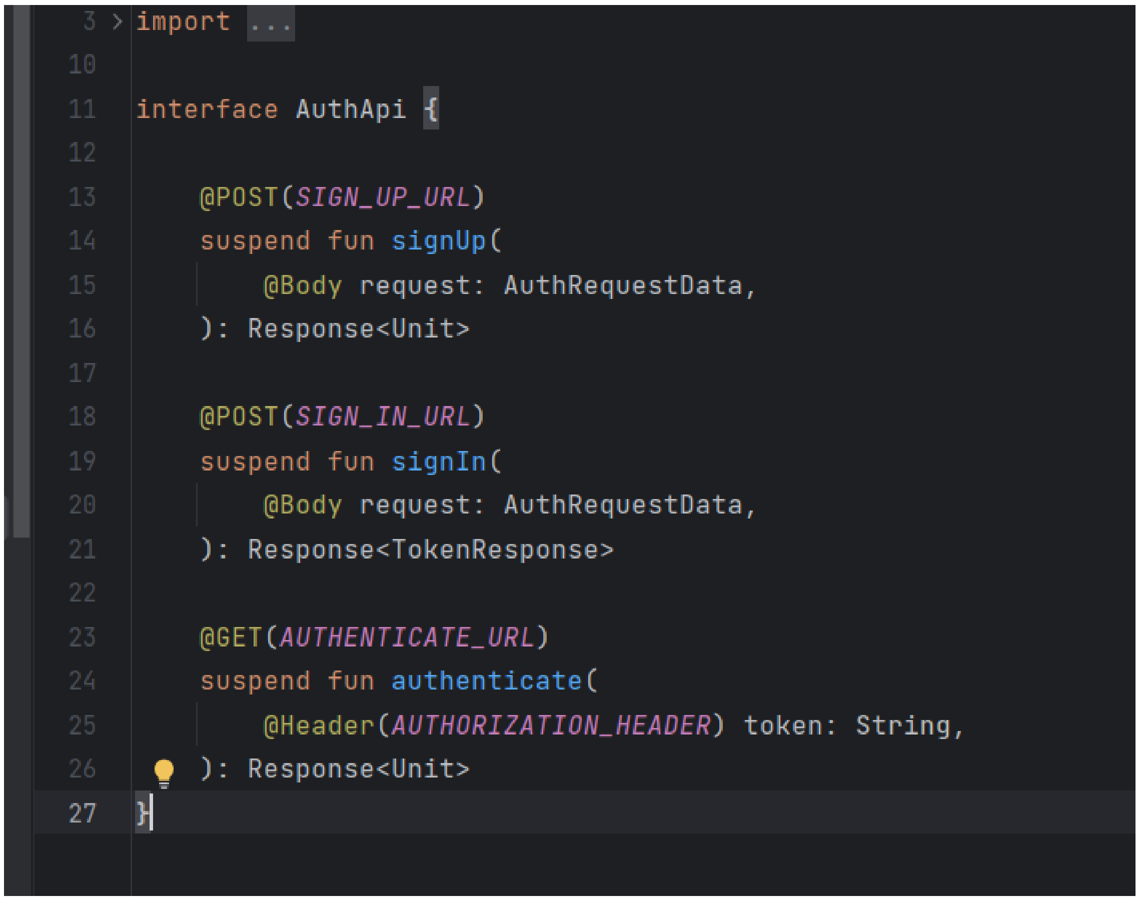


Fig. 2.2. Mobile app development process



```
kotlin+java
  com.glechyk.coffee44road
    data
      model.auth
        AuthRequestData
        AuthResult
        TokenResponse
      network
        ApiConstants.kt
        AuthApi
      repository
        AuthRepositoryImpl
```

Fig. 2.3. Mobile app package structure



```kotlin
import ...

interface AuthApi {

    @POST(SIGN_UP_URL)
    suspend fun signUp(
        @Body request: AuthRequestData,
    ): Response<Unit>

    @POST(SIGN_IN_URL)
    suspend fun signIn(
        @Body request: AuthRequestData,
    ): Response<TokenResponse>

    @GET(AUTHENTICATE_URL)
    suspend fun authenticate(
        @Header(AUTHORIZATION_HEADER) token: String,
    ): Response<Unit>
}
```

Fig. 2.4. API calls app implementation

Phase 5: Rigorous Evaluation through Mobile App Testing

Upon the completion of Android or iOS app design, coupled with the successful implementation of both front-end and back-end components by your development team, the imperative task of refining your product ensues. This necessitates the involvement of Quality Assurance (QA) specialists, whose expertise becomes instrumental in subjecting your application to comprehensive evaluations encompassing functionality, performance, security, and usability. Furthermore, QA specialists rigorously assess the compatibility of your mobile application across diverse devices and operating system versions, including conducting tests with real users. This meticulous testing regime enables the identification and rectification of bugs and limitations, thereby ensuring the optimal performance of your product prior to its market debut.

It is essential to underscore the significance of employing automated testing tools and methodologies during this phase to enhance efficiency and expedite the identification of potential issues across various testing parameters.

Phase 6: Strategic Rollout through Mobile App Deployment

In this pivotal stage, the culmination of efforts is manifested through the official release of your mobile app to the market. The selection of distribution models aligns with the operating system of the application, with platforms such as the Apple App Store and Google Play Store serving as exemplary channels for product dissemination. Concurrently, the strategic implementation of diverse marketing strategies assumes prominence as a means to enhance the visibility and popularity of your application among the target user base.

Consideration of App Store Optimization (ASO) techniques, encompassing keyword optimization, compelling visuals, and captivating descriptions, can significantly augment the discoverability of your mobile app within the competitive app marketplace.

Phase 7: Sustained Excellence through Post-Launch Support and Maintenance

Concluding the developmental trajectory is an ongoing commitment to excellence in the form of post-launch support and maintenance. This perpetual phase entails a continuum of enhancements, encompassing the introduction of new features, refinement of existing functionalities, implementation of design modifications, and rectification of any persisting bugs. This iterative process ensures the sustained relevance, user satisfaction, and overall performance of the mobile application in response to evolving market dynamics and user feedback.

Adherence to agile development methodologies, with regular sprint cycles and feedback loops, facilitates the seamless incorporation of user-driven enhancements and the swift resolution of emerging issues. The establishment of robust customer support mechanisms further fortifies user engagement and satisfaction.

2.1.2 The Comprehensive Framework of Mobile Application Development: A Technical Exploration

The intricate process of mobile application development entails the direct engagement of engineers, who play a pivotal role in shaping the final product. This developmental journey comprises two integral facets: front-end and back-end, each wielding significance in the creation of a seamless mobile application interface.

Front-End Development: Crafting User-Centric Experiences

The front-end development phase is dedicated to the creation of the visible components of the mobile application, encapsulating the mobile User Interface (UI).

Depending on the chosen app platform, front-end specialists employ various approaches:

1) Platform-Specific (Native) Development:

In this paradigm, the front-end expert dedicates efforts to the individualized design and development of the iOS or Android app. This approach ensures tailored optimization for each platform.

2) Cross-Platform Development:

Alternatively, front-end developers may adopt a cross-platform strategy, unifying the mobile interface for both iOS and Android. This is achieved through the utilization of universal code and tools, streamlining the development process.

3) Hybrid Development:

Hybrid development strikes a balance, leveraging tools for both platform-specific and cross-platform app creation. Here, the specialist employs standard code, encapsulated within technology for native applications.

Front-end developers, in addition to crafting the visible facets of the app, synchronize these elements with the back-end to ensure seamless functionality.

2.1.3 Post-Launch Support and Maintenance: A Continuous Imperative

The post-launch support and maintenance stage constitutes an ongoing commitment demanding substantial effort from the development team to sustain the operational integrity of the mobile application.

1) Continuous Testing:

The necessity of Quality Assurance (QA) specialists persists even after the application's release. Systematic testing remains imperative to identify and rectify bugs that may have eluded detection during the development phase.

2) User Feedback and Iterative Enhancements:

Gathering user feedback post-launch is integral to the iterative refinement of the application. Analyzing user input allows for the implementation of enhancements, addressing identified inconveniences and integrating suggestions for additional features.

3) Agile Adaptation to User Needs:

Prioritizing user needs is paramount during the post-launch phase. The user base serves as a valuable source of insights into usability issues and feature preferences. This iterative process of adaptation fosters continuous updates, cultivating user loyalty and sustained user engagement.

Conclusion

In conclusion, the development of a mobile application encompasses a dynamic interplay between front-end and back-end development, culminating in a perpetual commitment to post-launch support and maintenance that ensures the enduring relevance and excellence of the mobile application in a competitive digital landscape.

In view of the challenging yet promising trajectory of mobile app development, contextualizing it within the context of the burgeoning mobile solutions market and pervasive smartphone utilization underscores its indispensability for contemporary businesses.

The orchestration of a mobile application demands the collaborative efforts of a diverse team, encompassing front-end and back-end developers, UI/UX designers, business analysts, project managers, and Quality Assurance (QA) specialists. This multifaceted ensemble operates synergistically to navigate the intricate stages of development, ensuring the creation of a robust and user-centric mobile application.

2.2 Developing the server side using Ktor and Kotlin

Backend server development constitutes a critical facet of the overarching mobile application development process, playing a pivotal role in ensuring the robustness, scalability, and seamless functionality of the application. The intricate journey of backend server development unfolds through several key stages, each delineated below:

1) Formulate Backend Responsibilities

In the realm of application development, the backend often serves as more than a mere repository for simple data. In complex applications, intricate tasks are performed on the backend rather than the frontend. Two scenarios necessitating such complexity include:

- Data Validation: Verification of user permissions to perform actions, such as playing an audio file, involves server-side validation. This process requires querying a database to ascertain the user's entitlement.
- System Interaction: Applications often interact with various components of the system, such as databases. For instance, registering a user triggers updates to the user database, initiates a verification email to the user, and notifies the system administrator via email.

Various other considerations may warrant backend processing over frontend execution. It is imperative to delineate these considerations before commencing development.

2) Establish Initial Endpoints and Prototype Implementation

Following the determination of backend responsibilities, the next step involves the implementation of initial server endpoints to create a functional backend. Employing the backward backend technique entails developing stub functions for both the backend and the frontend. While not a fully-fledged API, these stubs serve to establish a minimal working backend, forming the foundation for subsequent development.

3) Develop and Document the Backend API

The third phase entails the comprehensive design and documentation of the backend API. While adherence to RESTful principles is often emphasized, a pragmatic approach advocates clear documentation over strict conformity to principles.

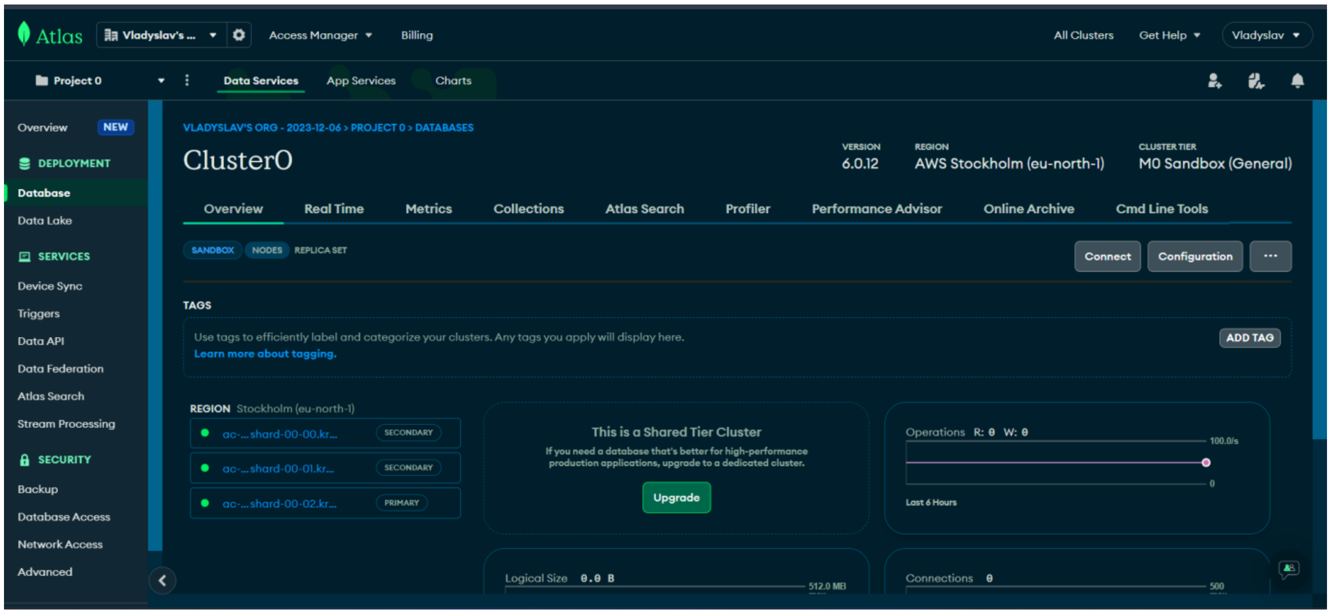
- Endpoint Functionality Documentation: Rather than conforming strictly to RESTful standards, prioritize explicit documentation for each backend API endpoint. Define the specific responsibilities of each endpoint, detailing the client-supplied values and the backend's corresponding responses.
- Mandatory and Optional Parameters: Thoroughly document which parameters are mandatory and which are optional for each endpoint. For instance, consider a 'get_menu' endpoint, wherein a date range parameter influences the returned menu list. If the client provides a category, the endpoint returns menus from that category; otherwise, it defaults to the whole list. Clearly articulating such intricacies is essential.
- Continuous Documentation Maintenance: The simplicity of the documentation process belies its significance. Consistently update and maintain the documentation to reflect any changes in the API. Keeping this documentation current ensures a comprehensive understanding of the backend's functionality and aids seamless collaboration between development teams.

In summary, the strategic progression from backend responsibility determination to API design and documentation involves thoughtful decision-making and meticulous planning, laying the groundwork for robust and scalable application development.

4) Formulate and Execute Database Design

Deriving insights from the API documentation, the structure and storage methodology for the data can now be meticulously planned and implemented.

- Database Selection: For a majority of scenarios, MySQL is recommended as the database management system, given its robust capabilities and widespread use.
- Data Normalization Principles: Adherence to sound data normalization principles is imperative. Strive for optimal normalization that aligns with the application's requirements, avoiding unnecessary complexity beyond what is justified.



Use tags to efficiently label and categorize your clusters. Any tags you apply will display here.
Learn more about tagging.

This is a Shared Tier Cluster
If you need a database that's better for high-performance production applications, upgrade to a dedicated cluster.

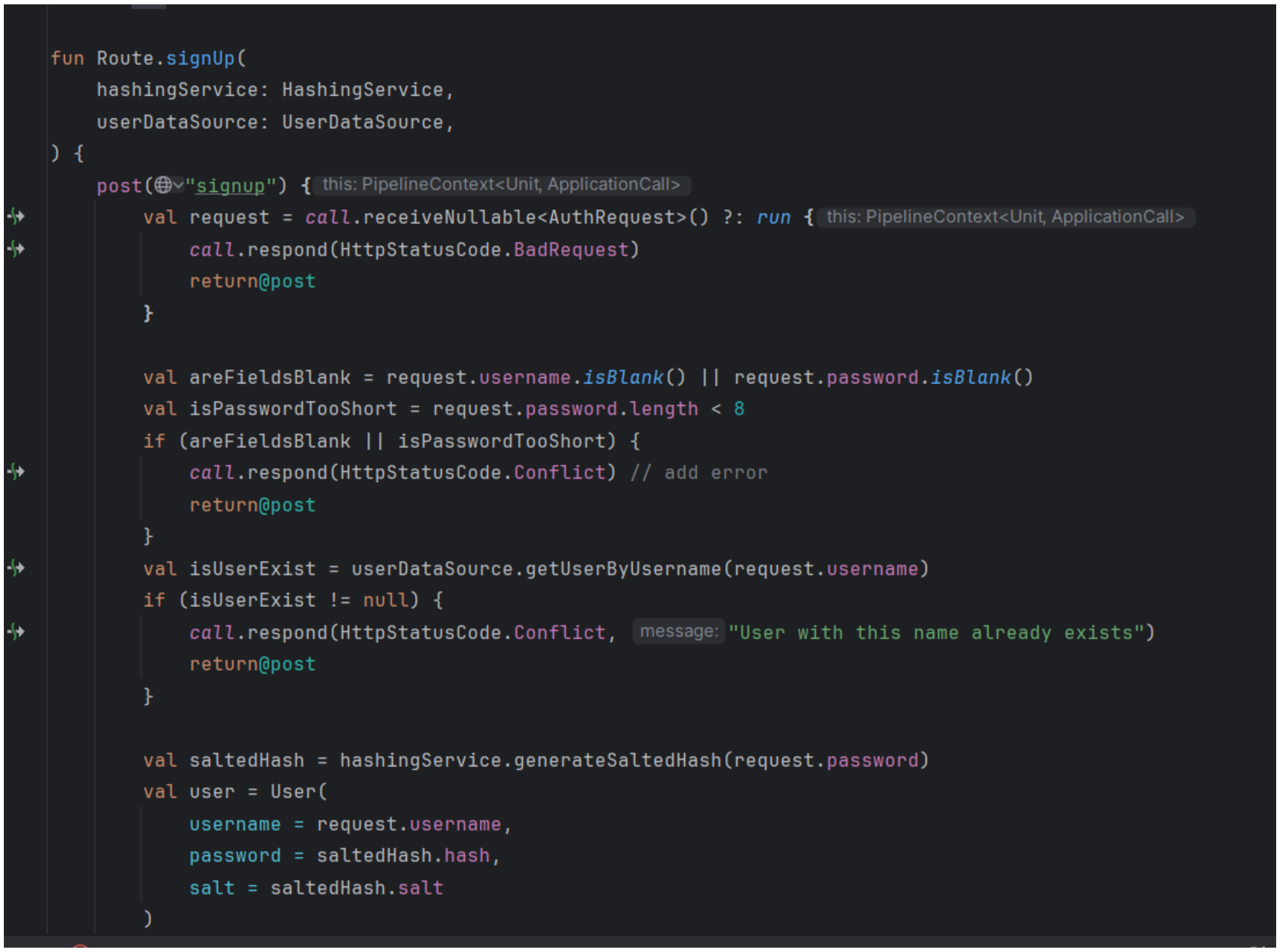
Fig. 2.5. MongoDB control panel

5) Develop a Backend Testing Script

A crucial step in the backend development process involves the creation of a comprehensive testing script, authored in bash/curl or Kotlin. This script systematically verifies the functionality of all backend endpoints, ensuring the reliability and integrity of the implemented features.

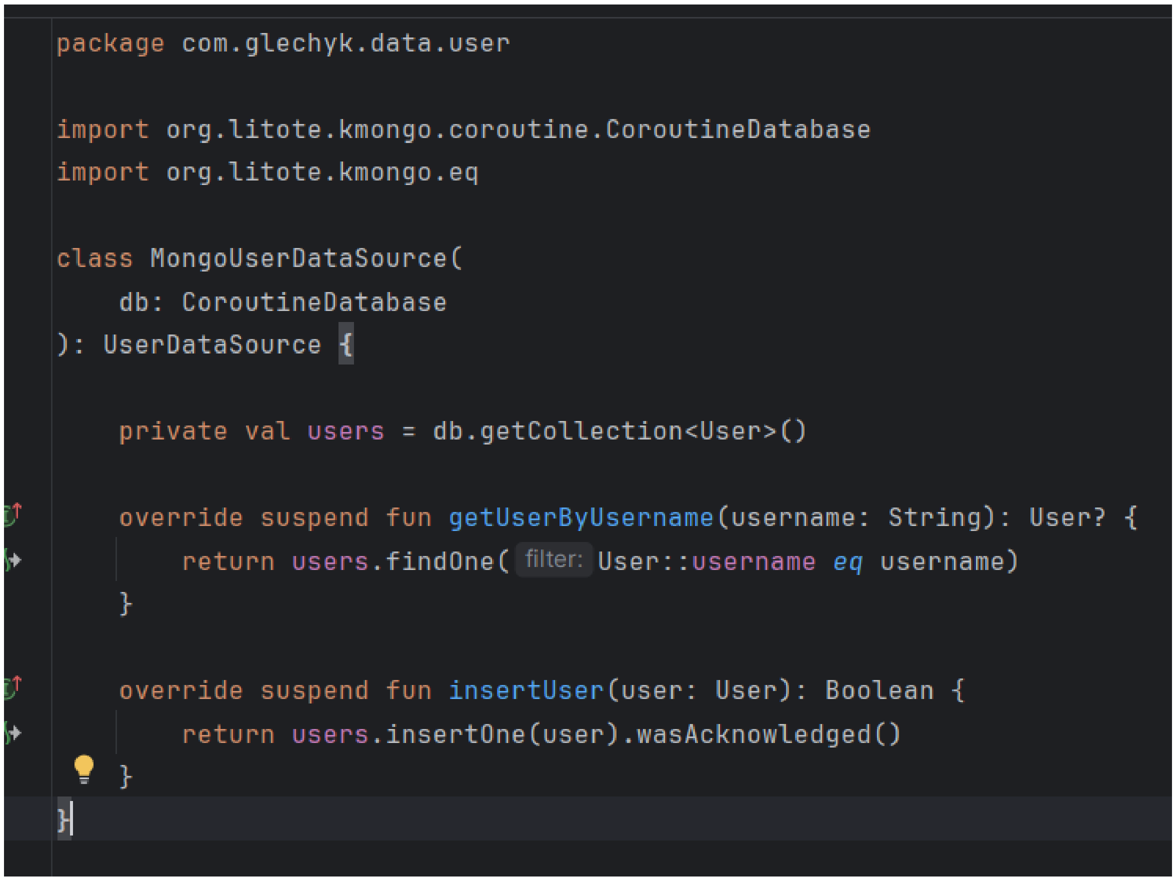
6) API Implementation Utilizing a Programming Language

The subsequent phase entails the actualization of the API through the utilization of a chosen programming language. The preferred language for this task is Kotlin, chosen for its efficiency and versatility. Further elucidation on this preference is provided in detail within supplementary screencasts.



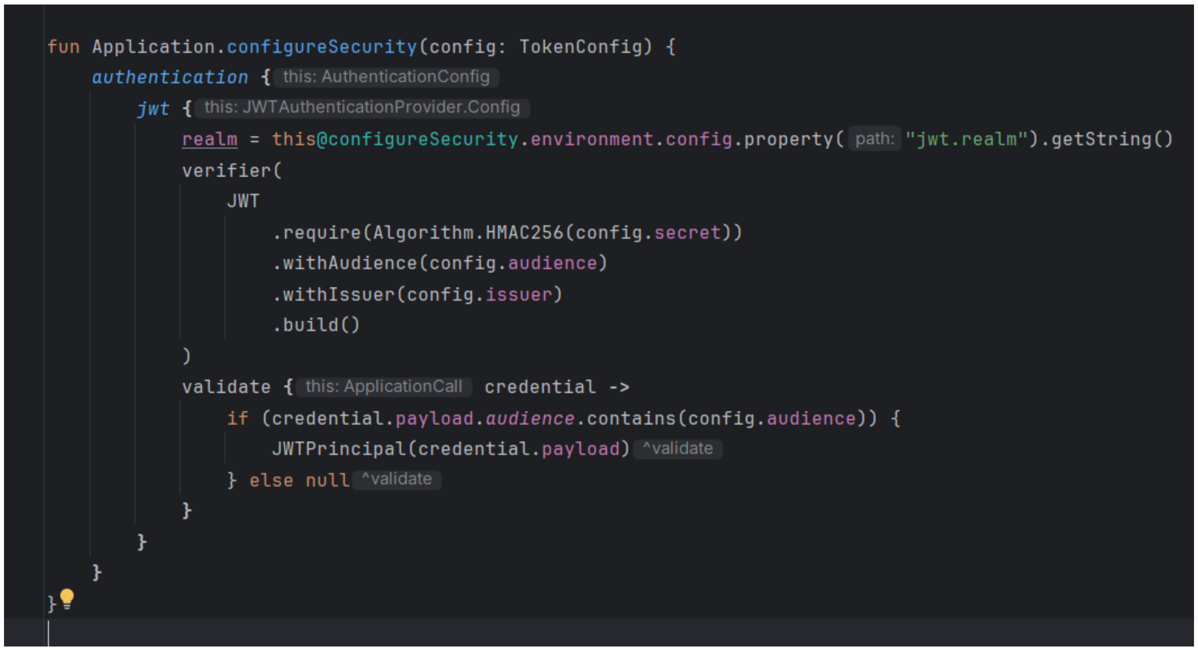
```
fun Route.signUp(
    hashingService: HashingService,
    userDataSource: UserDataSource,
) {
    post("signup") { this: PipelineContext<Unit, ApplicationCall>
        val request = call.receiveNullable<AuthRequest>() ?: run { this: PipelineContext<Unit, ApplicationCall>
            call.respond(HttpStatusCode.BadRequest)
            return@post
        }

        val areFieldsBlank = request.username.isBlank() || request.password.isBlank()
        val isPasswordTooShort = request.password.length < 8
        if (areFieldsBlank || isPasswordTooShort) {
            call.respond(HttpStatusCode.Conflict) // add error
            return@post
        }
        val isUserExist = userDataSource.getUserByUsername(request.username)
        if (isUserExist != null) {
            call.respond(HttpStatusCode.Conflict, message: "User with this name already exists")
            return@post
        }

        val saltedHash = hashingService.generateSaltedHash(request.password)
        val user = User(
            username = request.username,
            password = saltedHash.hash,
            salt = saltedHash.salt
        )
```

Fig. 2.6. API calls on the server side



```kotlin
package com.glechyk.data.user

import org.litote.kmongo.coroutine.CoroutineDatabase
import org.litote.kmongo.eq

class MongoUserDataSource(
    db: CoroutineDatabase
): UserDataSource {

    private val users = db.getCollection<User>()

    override suspend fun getUserByUsername(username: String): User? {
        return users.findOne( filter: User::username eq username)
    }

    override suspend fun insertUser(user: User): Boolean {
        return users.insertOne(user).wasAcknowledged()
    }
}
```

Fig. 2.7. Database requests implementation



```kotlin
fun Application.configureSecurity(config: TokenConfig) {
    authentication { this: AuthenticationConfig
        jwt { this: JWTAuthenticationProvider.Config
            realm = this@configureSecurity.environment.config.property( path: "jwt.realm").getString()
            verifier(
                JWT
                    .require(Algorithm.HMAC256(config.secret))
                    .withAudience(config.audience)
                    .withIssuer(config.issuer)
                    .build()
            )
            validate { this: ApplicationCall  credential ->
                if (credential.payload.audience.contains(config.audience)) {
                    JWTPrincipal(credential.payload) ^validate
                } else null ^validate
            }
        }
    }
}
```

Fig. 2.8 Security implementation

7) Deployment of the Backend Infrastructure

The final step involves deploying the backend infrastructure, a multifaceted process that warrants meticulous consideration.

- Cloud Service Deployment: Optimal deployment options include leveraging cloud services such as AWS for hosting the backend. This offers scalability, flexibility, and efficient resource management.
- Dedicated Hosting Considerations: Alternatively, a dedicated hosting solution can be employed based on specific project requirements. A dedicated host provides more control over the infrastructure but demands a comprehensive understanding of server management.
- Future Elaboration on Deployment: The intricacies of backend deployment represent a substantial topic deserving in-depth exploration, which will be addressed comprehensively in subsequent articles.

In summary, the holistic approach to backend development encompasses strategic database design, rigorous testing, adept API implementation, and judicious deployment choices, all of which collectively contribute to the creation of a robust and scalable backend infrastructure.

Conclusion

In conclusion, the comprehensive framework outlined herein provides a systematic and strategic approach to backend development, encompassing pivotal stages from backend responsibility determination to database design, testing, API implementation, and deployment. The discerning delineation of backend responsibilities at the outset ensures a nuanced understanding of the application's intricacies, facilitating informed decision-making throughout the developmental process.

The subsequent phases, including database design, emphasize the significance of aligning data structures with API documentation, advocating MySQL as the database management system of choice. The integration of sound data normalization principles underscores the commitment to maintaining optimal data integrity without undue complexity.

Testing procedures, manifested in a meticulously crafted backend test script, serve as a linchpin in the development lifecycle, ensuring the functionality and reliability of backend endpoints. The subsequent implementation of the API, utilizing the Kotlin programming language, affords efficiency and versatility, thereby enhancing the overall developmental efficiency.

The conclusive step of backend deployment encompasses nuanced considerations, advocating for cloud services like AWS for scalability and flexibility, or dedicated hosting solutions for enhanced control over infrastructure. Recognition of the intricacies of backend deployment underscores its pivotal role in the overarching developmental process.

Collectively, this comprehensive approach not only addresses the immediate imperatives of backend development but also establishes a foundation for scalability, maintainability, and adaptability. The integration of these strategic elements serves to fortify the backend infrastructure, laying the groundwork for robust and resilient applications within the dynamic landscape of contemporary software development.

2.3 Integration of the mobile application and the server side

The ubiquity of mobile devices and the escalating demand for sophisticated mobile applications accentuate the imperative nature of seamless integration between client-side and server-side components. This scholarly exploration delves into the intricate facets of this integration, underscoring its critical role in augmenting application performance, fortifying data security, and refining user experiences within the digital realm.

2.3.1 Components of Integration between Mobile Applications and Server-Side Infrastructure:

1) Communication Protocols:

The meticulous consideration and deployment of communication protocols form a pivotal aspect of the amalgamation between mobile applications and server-side infrastructure. This scholarly inquiry involves an in-depth analysis of prevalent protocols, encompassing HTTP/HTTPS, WebSocket, and RESTful APIs. It investigates the nuanced intricacies of each protocol, evaluating their applicability and impact on the efficiency and security of data transmission.

2) Data Synchronization:

An integral component of the integration process involves devising strategies for optimal data synchronization between mobile devices and server entities. This academic exploration scrutinizes methodologies for achieving efficient synchronization, emphasizing the examination of conflict resolution techniques and mechanisms ensuring steadfast data consistency. Comprehensive insights are provided into the challenges and innovations in data synchronization for a harmonized mobile-server integration.

3) Authentication and Authorization:

Security considerations loom large in the integration paradigm, particularly concerning the implementation of robust authentication and authorization mechanisms. This academic discourse delves into the intricacies of security protocols, investigating token-based authentication, OAuth, and other prevailing industry-standard approaches.

A critical evaluation is conducted to discern the strengths and vulnerabilities of each method, thereby informing best practices for ensuring the integrity of user authentication and authorization processes.

2.3.2 Challenges and Considerations in Integration:

1) Network Latency:

A substantial challenge in the integration landscape is posed by network latency and its potential ramifications on real-time interactions. This academic investigation dissects the complexities associated with network latency, offering in-depth analyses of mitigation strategies aimed at delivering responsive mobile applications. Consideration is given to advanced techniques and emerging technologies that alleviate latency concerns, ensuring a seamless user experience.

2) Scalability:

Scalability emerges as a critical consideration in the integration of mobile applications and server-side infrastructure. This scholarly exploration delves into the intricacies of scalability, examining the role of load balancing, caching mechanisms, and other strategies to sustain optimal performance under varying workloads. The discourse encompasses a detailed evaluation of scalability challenges and the deployment of cutting-edge technologies to address evolving demands effectively.

3) Security Concerns:

Security concerns in the integration process demand meticulous attention. This scholarly inquiry scrutinizes potential vulnerabilities and proactively identifies strategies to mitigate security risks. A comprehensive examination of encryption methodologies, secure communication protocols, and best practices for safeguarding data during transfer is undertaken. The discourse endeavors to fortify the integration process against security threats, ensuring the confidentiality and integrity of sensitive information.

2.3.3 Emerging Trends and Future Directions:

1) Edge Computing:

An emerging trend in the integration landscape is the integration of edge computing to optimize mobile application and server-side interactions. This scholarly discourse elucidates the pivotal role of edge computing in enhancing integration efficiency and explores decentralized processing models. Insights into the potential benefits and challenges associated with this paradigm shift are expounded upon, offering a forward-looking perspective on the evolution of mobile-server integration.

2) Microservices Architecture:

The adoption of microservices architecture stands out as a transformative trend in mobile application development and server-side integration. This scholarly exploration dissects the intricacies of microservices architecture, elucidating its potential benefits and challenges. The discourse extends to the decomposition of monolithic server-side applications, providing nuanced insights into the adoption and optimization of microservices for seamless mobile-server integration.

Conclusion

This comprehensive discourse serves to elucidate the multifaceted nature of the integration between mobile applications and server-side infrastructure, emphasizing its critical role in contemporary digital ecosystems. By unraveling the intricacies of communication protocols, data synchronization, authentication, and addressing challenges and emerging trends, this academic exploration contributes valuable insights for researchers, developers, and industry practitioners. It paves the way for the continued evolution of robust, scalable, and secure mobile solutions, enriching the digital experience for end-users.

2.4 Evaluating the effectiveness of using the Kotlin programming language for full-stack development

The selection of a programming language in the context of full-stack development is a pivotal determinant of project efficiency and success. Kotlin, a language developed by JetBrains, has garnered attention owing to its conciseness, expressiveness, and adaptability. This scholarly article presents a rigorous case study that employs Kotlin for both client-side and server-side development, utilizing the Android SDK and the Ktor framework, respectively.

2.4.1 Client-Side Development Using Kotlin and Android SDK

The client-side of the comprehensive full-stack application is instantiated through the utilization of Kotlin in tandem with the Android SDK. The interoperability of Kotlin with Java, coupled with its refined syntax and incorporation of null-safety features, contributes to the establishment of a more resilient and comprehensible codebase. This study meticulously assesses the influence of these distinctive features on the developmental process, code maintenance, and the overall efficiency of the project.

- Strengths and Advantages

The amalgamation of the Android SDK and Kotlin results in a streamlined developmental experience, empowering developers to craft feature-rich and performance-optimized mobile applications. The expressive nature of Kotlin facilitates the creation of succinct code, thereby diminishing the likelihood of errors and augmenting the clarity of the codebase. Additionally, Kotlin's null-safety features play a crucial role in mitigating common runtime errors, enhancing the robustness of the application.

- Limitations and Challenges

Notwithstanding its strengths, challenges may emerge, particularly in relation to the learning curve encountered by developers transitioning from Java to Kotlin. Furthermore, certain constraints, such as limitations in third-party library support and disparities in community resources, have the potential to influence the broader developmental ecosystem. A meticulous exploration of these challenges is essential for a comprehensive understanding of the overall implications associated with the adoption of Kotlin in full-stack development scenarios.

2.4.2 Server-Side Development Utilizing Kotlin and Ktor Framework: A Comprehensive Examination of Performance, Scalability, and Development Productivity

The development of the server-side component in the overarching full-stack application is executed through the adept utilization of Kotlin in conjunction with the Ktor framework - a lightweight and versatile framework specifically designed for the construction of asynchronous servers and clients. This scholarly investigation rigorously assesses the efficacy of this amalgamation with respect to performance metrics, scalability considerations, and the overall productivity of the development process.

- Strengths and Advantages

The seamless integration of Kotlin with the Ktor framework fosters the creation of server-side applications characterized by scalability and high performance. The asynchronous nature of Ktor enhances its capacity for handling concurrent requests, rendering it well-suited for the demands of contemporary web applications. Furthermore, the concise syntax inherent to Kotlin remains consistently applied, thereby promoting the maintainability of server-side codebases.

- Limitations and Challenges

In consonance with the inherent complexities of technology stacks, challenges may manifest, particularly in relation to the requirement for developers to acquire proficiency in the idiosyncrasies of the Ktor framework. Additionally, the extent of third-party library availability and the level of community support for Ktor could exert an influence on the broader developmental ecosystem.

Conclusion

This exhaustive case study illuminates both the strengths and weaknesses intrinsic to the integration of the Kotlin programming language in full-stack development scenarios, encompassing the use of the Android SDK on the client-side and the Ktor framework on the server-side. The discerned findings furnish invaluable insights into the holistic effectiveness of Kotlin within a multifaceted development environment. Despite the identified challenges, the discernible advantages stemming from Kotlin's characteristics - conciseness, expressiveness, and interoperability - underscore its compelling suitability for full-stack development initiatives.

Future research endeavors may delve into additional dimensions within the Kotlin ecosystem, encompassing advancements in tooling, the dynamics of community support, and the evolutionary trajectory of the Ktor framework. A continual assessment of the performance attributes and adaptability of this technology stack vis-a-vis evolving industry standards promises to contribute substantially to a nuanced comprehension of Kotlin's pivotal role in shaping the expansive domain of full-stack development.

CHAPTER 3.

ANALYSIS AND COMPARISON WITH OTHER PROGRAMMING LANGUAGES

3.1 Comparison with Java in the context of full-stack development

Kotlin, a statically-typed programming language introduced by JetBrains in 2011, aims to surpass Java in terms of conciseness and safety. In contrast, Java, with a history spanning over two decades, is renowned for its portability, scalability, and security.

The ongoing discourse between Kotlin and Java has persisted for several years, with developers deliberating on the merits and drawbacks of each language. Some favor Kotlin's succinct syntax and robust features, while others assert that Java's stability and resilience position it as the superior choice.

This scholarly exposition delves into the syntax and attributes of Kotlin and Java, scrutinizing their performance, productivity, community dynamics, adoption rates, and use cases. By the conclusion of this discourse, readers will attain a comprehensive understanding of the strengths and weaknesses inherent in each language, empowering them to make informed decisions when selecting the most suitable language for their forthcoming projects.

3.1.1 Origins of Java and Kotlin

Kotlin and Java trace their origins to distinct sources, shaping their respective design and evolution.

JetBrains, a software development company conceived Kotlin. Initially formulated to rectify deficiencies in Java and enhance developer efficiency, JetBrains subsequently released Kotlin as an open-source language in 2012, garnering recognition for its simplicity and potent features.

In contrast, Java emerged from the efforts of James Gosling and his team at Sun Microsystems during the early 1990s. Crafted to be portable, secure, and scalable, Java swiftly gained prominence, particularly in the realm of web development. In 2009, Oracle Corporation acquired Sun Microsystems, assuming responsibility for the ongoing development and maintenance of the language.

Despite their shared object-oriented paradigm and some semblance in syntax and functionalities, the distinctive origins of Kotlin and Java have wielded influence over their developmental trajectories. Kotlin was purposefully crafted to address specific drawbacks in Java, such as null safety and verbosity. Meanwhile, Java's extensive history and widespread adoption contribute to its expansive and firmly established ecosystem.

3.1.2 Java vs Kotlin: A Comparative Analysis of Key Features

Java, a longstanding programming language widely utilized in diverse application development contexts, possesses distinctive features that contribute to its prominence. Noteworthy characteristics of Java include:

1) Checked Exceptions:

Java incorporates a "checked exceptions" mechanism, compelling the handling or declaration of exceptions by the calling method. While enhancing code reliability, this feature introduces complexity.

2) Primitive Types as Non-Classes:

Java employs primitive data types (e.g., int, float, boolean) that are not represented as objects. This design choice, utilized in bytecode to minimize memory footprint and program overhead, contributes to improved performance.

3) Static Members Replacement:

Static members in Java, accessible without class instance creation, find their counterparts in Kotlin through companion objects, top-level functions, extension functions, or @JvmStatic annotations. This substitution minimizes boilerplate code and enhances code readability.

4) Wildcard Types Replacement:

The wildcard types in Java, expressing unknown types, are supplanted in Kotlin by declaration-site variance and type projections. This transition facilitates more precise type inference and fosters safer code practices.

5) Ternary Operator Replacement:

Java's ternary operator (a ? b : c) is substituted in Kotlin with if expressions, contributing to code conciseness and reducing redundancy.

6) Maturity:

Java's extensive deployment in large-scale projects establishes its reputation as a reliable and stable language, positioning it as a secure choice for enterprise-level endeavors.

In contrast, Kotlin, a language characterized by distinct features, offers a unique set of advantages:

1) No Raw Types:

Kotlin eliminates raw types, enhancing type safety and mitigating the risk of runtime errors.

2) Invariant Arrays:

Kotlin treats arrays as invariant in its type system, preventing potential issues associated with covariant or contravariant arrays.

3) Null-Safety:

Kotlin's null-safety features identify null references at compile-time, reducing the likelihood of runtime errors.

4) String Templates:

Kotlin simplifies string interpolation through string templates, reducing the need for boilerplate code.

5) Properties:

Kotlin introduces properties, streamlining code by replacing traditional Java getters and setters.

6) Declaration-Site Variance & Type Projections:

Kotlin offers variance and type projection features, facilitating flexible type handling and diminishing the necessity for explicit casting.

7) Companion Objects:

The introduction of companion objects in Kotlin provides a means to associate functionality with a class without relying on static members.

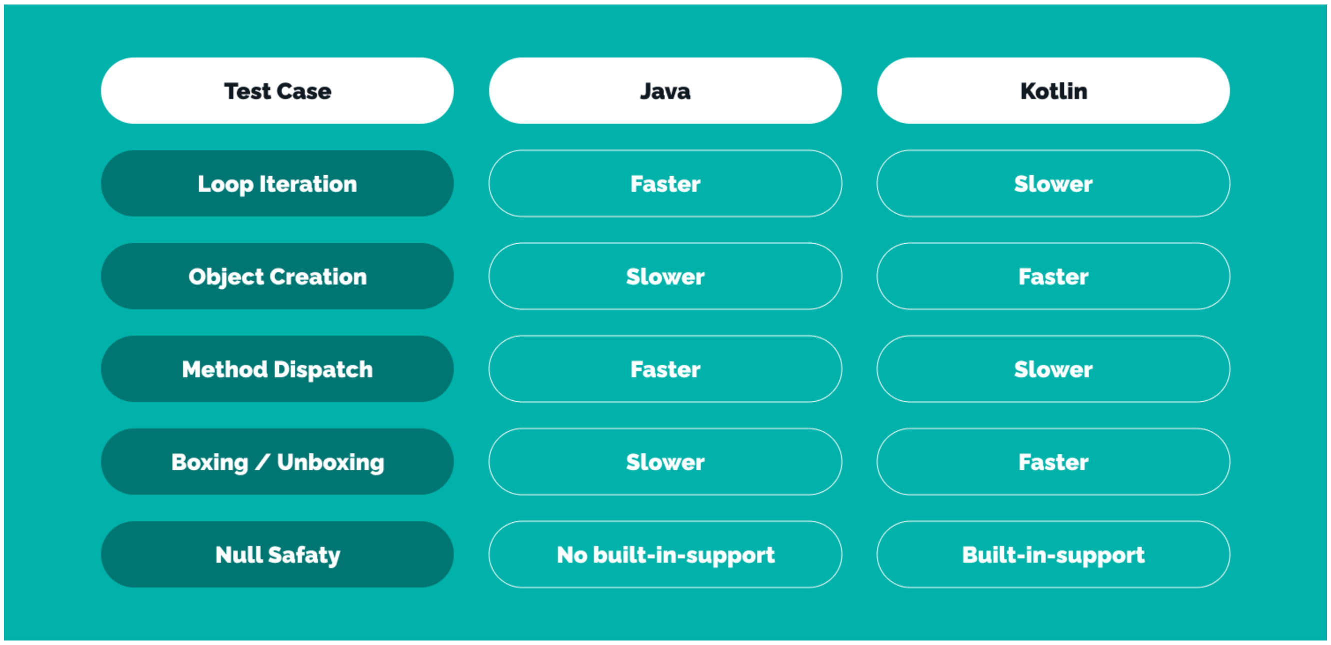
8) Separate Interfaces for Collections:

Kotlin's standard library incorporates distinct interfaces for read-only and mutable collections, enhancing code conciseness and reducing error potential.

In summary, Java and Kotlin exhibit distinct sets of features, each tailored to address specific programming challenges. The evaluation of these features aids developers in making informed decisions based on the requirements of their projects.

3.1.3 Performance Disparities between Kotlin and Java

Over the years, Kotlin has undergone substantial performance enhancements, occasionally surpassing Java in certain scenarios. Notably, Kotlin's incorporation of inline functions and reified type parameters has proven effective in mitigating overhead associated with generic types. Furthermore, Kotlin's adept management of null values and utilization of immutable data structures contribute to its generally lower memory usage compared to Java.



Test Case	Java	Kotlin
Loop Iteration	Faster	Slower
Object Creation	Slower	Faster
Method Dispatch	Faster	Slower
Boxing / Unboxing	Slower	Faster
Null Safaty	No built-in-support	Built-in-support

Fig. 3.1. Java with Kotlin performance comparison

Contrarily, Java has traditionally exhibited superior raw performance, owing to well-established optimization techniques and a mature ecosystem. Nonetheless, the practical variance in performance between Kotlin and Java tends to be marginal in real-world applications.

The ultimate determination of whether Kotlin or Java offers superior performance hinges on the specific demands of the application and the inherent trade-offs among performance, maintainability, and developer productivity. Regular benchmarking and profiling are imperative to identify performance bottlenecks and optimize applications for optimal performance.

3.1.4 Productivity Dynamics in Kotlin versus Java

When selecting a programming language, productivity emerges as a pivotal consideration, exerting a substantial influence on development timelines and the resultant product quality.

Kotlin, conceived with a focus on enhancing developer productivity, incorporates various features conducive to efficiency. Notably, Kotlin's succinct syntax and type inference contribute to codebase conciseness, while its null safety mechanisms serve as a preventive measure against NullPointerExceptions - a prevalent source of errors in Java.

Moreover, Kotlin's robust support for functional programming constructs enhances code expressiveness and conciseness. The language's seamless interoperability with Java fosters a smooth transition, allowing developers to integrate existing Java code and libraries effortlessly.

Conversely, Java, with its more mature ecosystem and expansive community, boasts a broader array of tools, frameworks, and libraries. This abundance streamlines development processes, enabling developers to capitalize on these resources for swift application construction.

3.1.5 Use Cases of Java

1) Enterprise Applications and Web Development:

Java serves as a cornerstone for developing enterprise applications, encompassing financial systems, e-commerce platforms, and inventory management systems. Additionally, it is a prevalent choice in web development, with frameworks like Spring and Struts facilitating the creation of robust web applications.

2) Android Application Development using Android SDK:

Java stands as the principal language for crafting Android applications, leveraging the Android SDK to harness a rich set of libraries and tools for mobile app development.

3) Server-Side Applications with Frameworks like Spring and Hibernate:

Java is frequently employed in constructing server-side applications, with frameworks such as Spring and Hibernate providing support for the development of web services, RESTful APIs, and other server-side functionalities.

4) Large-Scale Applications with High-Performance Requirements:

Java's performance and scalability make it an apt choice for constructing large-scale applications demanding high-performance and reliability. This spans diverse sectors, including trading systems, airline reservation systems, and telecom billing systems.

5) Desktop Applications using JavaFX:

JavaFX equips developers with tools and libraries for constructing desktop applications, rendering Java suitable for applications like trading platforms, CAD software, and data visualization tools.

6) Scientific and Engineering Applications:

Java finds utility in the development of scientific and engineering applications that necessitate high-performance computing and intricate algorithms. Applications encompass simulations, data analysis, and image processing.

3.1.6 Use Cases of Kotlin

Kotlin, in turn, is applied in various scenarios:

1) Android App Development with Android-Specific Features:

Kotlin emerges as a favored choice for Android application development, boasting a concise syntax, null safety features, and compatibility with Java. Specific extensions like Anko and KTX further enhance its utility in the Android ecosystem.

2) Server-Side Development with Ktor and Spring:

Kotlin proves adept in server-side application development, collaborating with frameworks like Ktor and Spring to facilitate the creation of web services and RESTful APIs.

3) Cross-Platform Development via Kotlin Native:

Kotlin Native empowers developers to write code that can be compiled into native binaries, enabling the construction of cross-platform applications for desktop, mobile, and embedded devices.

4) Reactive and Event-Driven Applications with Functional Programming Concepts:

Kotlin's embrace of functional programming concepts, including higher-order functions, positions it as an apt choice for the development of reactive and event-driven applications. This is particularly advantageous in scenarios dealing with real-time data streams and asynchronous events.

5) Building Tools and Plugins for Development Environments:

Kotlin's interoperability with Java and its support for Domain-Specific Languages (DSLs) render it suitable for building tools and plugins within development environments such as IntelliJ IDEA.

3.1.7 The Future Trajectory of Java and Kotlin

Both Java and Kotlin are anticipated to undergo continual evolution and enhancement. The forthcoming release of Java 17 introduces novel features like pattern-matching instances and records. In parallel, Kotlin is expected to augment its capabilities and ecosystem, emphasizing improved support for multiplatform development and enhanced tooling.

The evolving landscape of Java and Kotlin is likely to influence their utilization differentially. As Java progresses, incorporating new features and enhancements, it stands poised to remain an appealing option, particularly for enterprise-scale and large applications. Kotlin's strategic focus on multiplatform development and functional programming positions it favorably for cross-platform development and the unfolding landscape of emerging application architectures.

An evident trajectory toward multiplatform development and the integration of functional programming concepts is anticipated. This trend may contribute to heightened adoption of Kotlin and other languages aligned with these developments. Simultaneously, the ascendancy of artificial intelligence and machine learning may propel the utilization of languages like Python and R, particularly in the realm of data analysis and modeling. Nonetheless, Java's sustained popularity and robust community support suggest its enduring prevalence in the foreseeable future.

3.1.8 Key Distinctions Between Java and Kotlin

In summary, the disparities between Kotlin and Java encompass various facets:

1) Syntax:

Kotlin exhibits a more concise and expressive syntax compared to Java, resulting in the achievement of similar functionality with less code.

2) Null Safety:

Kotlin integrates built-in null safety features, mitigating the susceptibility to NullPointerException errors in contrast to Java.

3) Functional Programming:

Kotlin demonstrates superior support for functional programming concepts, such as lambdas and higher-order functions, which were only recently incorporated into Java.

4) Interoperability:

Kotlin seamlessly interacts with Java code, exhibiting full interoperability. However, the reverse integration is not universally true.

5) Learning Curve:

Kotlin is recognized for its ease of learning and use, rendering it particularly suitable for beginners and smaller projects compared to Java.

6) Community:

Java boasts a larger community and a more extensive array of libraries and frameworks than Kotlin, thereby offering a broader spectrum of resources and solutions for Java developers.

Conclusion

Java, as a mature language, caters to diverse domains, including enterprise applications, web development, Android app development, and large-scale applications. In parallel, Kotlin, characterized by its modern attributes, excels in domains such as Android development, server-side operations, cross-platform development, reactive and event-driven applications, as well as tools and plugins for development environments.

For Android app development, Kotlin stands out due to its compatibility with Java and specialized features tailored for the Android platform. In server-side development, both Java and Kotlin remain viable, with Java being the more prudent choice for larger and intricate applications. In the domain of cross-platform development, Kotlin Native shows promise, albeit with a comparative novelty compared to other established frameworks. For desktop applications and the development of tools/plugins, Java maintains its robust standing, supported by a vibrant community and ecosystem.

Ultimately, the selection between Kotlin and Java hinges on the specific project requisites and the developer's experience and inclinations.

3.2 Comparison with JavaScript: advantages and disadvantages

3.2.1 What is Node.js

Node.js serves as a runtime environment for executing JavaScript code, diverging from traditional JavaScript execution in browsers. It leverages the Chrome V8 engine, translating JavaScript calls into machine code. In essence, Node.js distinguishes itself as an operational environment for the JavaScript language.

The genesis of Node.js traces back to 2009, initiated by the efforts of American developer Ryan Dahl. Departing from the prevailing approach of allocating one thread per connection, Dahl's novel platform underscored event-driven principles, with a primary emphasis on constructing scalable network servers.

3.2.2 Advantages of Adopting Node.js

Node.js has evolved into a standalone command-line utility renowned for constructing expeditious and scalable applications. Its numerous advantages render it a compelling choice for businesses and developers alike. The ensuing section delineates some key advantages inherent in Node.js development services.

1) Fast Performance:

Node.js is underpinned by the Google V8 JavaScript engine, recognized for its remarkable speed and efficiency. This attribute positions Node.js as an exemplary solution for applications necessitating real-time data processing or managing substantial data volumes, particularly when juxtaposed with Kotlin.

2) Lightweight Application Development:

Node.js distinguishes itself through its lightweight architecture, lending itself well to the creation of nimble applications. This characteristic proves particularly advantageous for mobile or web applications requiring optimization for speed and performance.

3) Node Package Manager:

An integral facet of Node.js is its built-in package manager, npm, acknowledged as the world's largest package registry. Developers benefit from the ease with which they can install and explore packages, facilitating the augmentation of application functionality without resorting to coding from scratch.

4) Rich Repository of Libraries:

The Node.js ecosystem boasts approximately 836,000 libraries for project development, with nearly 10,000 new libraries introduced weekly. This extensive repository empowers developers with a diverse array of tools and resources, expediting the construction of resilient and scalable applications.

5) Elimination of Binary Model Conversion Overheads:

Node.js employs a single-threaded event loop, obviating the need for binary model conversions. This design choice engenders high efficiency, surpassing other technologies that necessitate such conversions.

6) Scalable Backend Technology:

Positioned as a scalable backend technology, Node.js incorporates a lightweight framework ideal for applications handling substantial traffic volumes. The event-driven, non-blocking I/O model enhances its efficacy in managing extensive data sets and real-time applications.

7) Continual Technological Advancement:

Node.js stands as a dynamically evolving technology, marked by consistent updates and feature enhancements. Noteworthy endeavors, such as the ongoing development of TypeScript, exemplify efforts aimed at simplifying the creation of scalable and maintainable code for developers.

3.2.3 Optimal Utilization of Node.js: A Comparative Examination

Node.js, renowned for its versatility, manifests an extensive spectrum of applications. The employment of Node.js proves integral in scenarios necessitating rapid, multi-user real-time data processing. While conventional paradigms involve the utilization of Java for both frontend and backend development, an alternative approach is to leverage Node.js across diverse domains. The ensuing enumeration delineates prominent areas where the services of a Node.js app development company may be indispensable:

1) Collectors and Translators:

Node.js demonstrates efficacy in scenarios requiring data collection and translation, leveraging its event-driven architecture to facilitate seamless processing.

2) Batch Processing and Deferred Processing Scenarios:

The application of Node.js extends to batch processing and situations demanding deferred processing, showcasing its prowess in managing asynchronous tasks.

3) Scripts, CLI (Command-Line Interfaces):

Node.js serves as a proficient tool for crafting scripts and command-line interfaces, offering developers a streamlined means of executing tasks.

4) Document Generation and Delayed Report Generation:

The capabilities of Node.js find relevance in the generation of documents and the delayed production of reports, underscoring its utility in handling time-sensitive data operations.

5) Web Applications and SPA Servers:

Node.js emerges as a compelling choice for the development of web applications and servers dedicated to Single Page Applications (SPAs), harnessing its efficiency in real-time, interactive scenarios.

6) Servers and APIs for Mobile Applications:

The deployment of Node.js extends to the creation of servers and APIs tailored for mobile applications, capitalizing on its ability to manage concurrent connections effectively.

7) Message Servers and Event Broadcasting (Chats, Games, Interactive):

Node.js excels in the domain of message servers and event broadcasting, exemplified in applications involving chats, games, and interactive functionalities.

8) Patches on Pre-existing Systems, Written in Other Source Languages:

Node.js proves instrumental in implementing patches on pre-existing systems, originally scripted in alternative source languages, attesting to its versatility in interoperability.

9) Window Applications (nw.js, Node-Webkit):

Node.js facilitates the development of window applications through frameworks like nw.js and Node-Webkit, demonstrating its adaptability in diverse application architectures.

10) Crawlers, Parsers, and Data Collection:

The event-driven nature of Node.js is particularly advantageous in the context of crawlers, parsers, and data collection, enabling efficient management of diverse data sources.

11) Microcontroller Programming:

The application of Node.js extends to microcontroller programming, underscoring its utility in diverse hardware integration scenarios.

12) Industrial Automation:

Node.js finds relevance in the realm of industrial automation, contributing to the creation of streamlined, event-driven systems.

13) CMS, Content Publishing:

Node.js is instrumental in Content Management Systems (CMS) and content publishing, offering an agile platform for managing and disseminating content.

14) Mass E-commerce and Trade:

The deployment of Node.js proves advantageous in the expansive domain of mass e-commerce and trade, where real-time processing and scalability are pivotal.

In conclusion, the judicious selection of Node.js across these diverse applications underscores its adaptability and effectiveness in addressing complex, real-time data processing requirements.

3.2.4 Core Features of Kotlin

One of Kotlin's foundational characteristics lies in its unwavering commitment to object-oriented programming (OOP), wherein code is structured into objects for enhanced reusability and extensibility. Notable OOP features, including encapsulation, inheritance, and polymorphism, underscore Kotlin's capacity to facilitate the development of intricate and modular applications.

3.2.5 Advantages of Utilizing Kotlin

Kotlin, being a high-level programming language, excels in addressing a broad spectrum of general problems. Amidst the proliferation of emerging technologies, Kotlin development services retain several merits that contribute to its competitive edge:

1) Object-Oriented Programming:

Embracing an object-oriented paradigm, Kotlin organizes code into reusable and extendable objects, facilitating the development of intricate, modular applications with long-term efficiency gains.

2) Platform Independence:

Kotlin's code, compiled into bytecode, exhibits platform independence, enabling execution on any platform equipped with a Kotlin Virtual Machine (KVM). This intrinsic feature streamlines the development and deployment of applications, irrespective of underlying hardware or operating system disparities.

3) Security Features:

Kotlin integrates built-in security mechanisms, including the Security Manager, empowering developers to delineate access rules and restrict entry to sensitive resources. Notably, Kotlin's absence of support for pointers contributes to mitigating memory errors and enhancing overall security.

4) Automatic Memory Management:

Kotlin incorporates a sophisticated garbage collector system, automating memory management by identifying and deallocating objects no longer in use. This eliminates the need for manual memory management, mitigating potential sources of bugs and associated issues.

5) Multithreading:

Kotlin's support for multithreading enables the concurrent execution of multiple threads without mutual interference. In comparison to the Node vs Kotlin comparison, this feature optimizes the utilization of system resources, potentially enhancing application performance.

In conclusion, Kotlin's robust object-oriented foundation, platform independence, security features, automatic memory management, and support for multithreading collectively position it as a compelling programming language, effectively addressing the evolving landscape of contemporary software development.

3.2.6 Optimal Utilization of Kotlin: A Comprehensive Evaluation

Given its array of advantageous features, Kotlin has garnered extensive utilization, albeit often inconspicuous in its prevalence. In juxtaposing Node.js vs Kotlin, discerning considerations reveal distinct application domains for Kotlin:

1) Android Applications:

Kotlin distinguishes itself as a prominent choice for the development of Android applications, leveraging its robust features to enhance the mobile app development landscape.

2) Server Applications in Financial Services:

Kotlin proves instrumental in crafting server applications within the realm of financial services, underscoring its efficacy in handling complex and data-intensive scenarios.

3) Web Applications:

The versatility of Kotlin extends seamlessly to the domain of web applications, offering developers a reliable platform for constructing interactive and scalable web-based solutions.

4) Software Tools:

Kotlin's applicability transcends conventional application development, finding utility in the creation of software tools. Its features contribute to the development of efficient and versatile tools for diverse purposes.

5) Trading Applications:

In the context of trading applications, Kotlin emerges as a competent choice, providing a robust foundation for the development of applications within dynamic and high-stakes financial environments.

6) J2ME Applications:

Kotlin finds resonance in the development of applications utilizing the Java 2 Platform, Micro Edition (J2ME), showcasing its adaptability to diverse platforms and environments.

7) Embedded Systems:

The utilization of Kotlin extends to embedded systems, highlighting its versatility in scenarios requiring efficient and resource-conscious programming.

8) Big Data:

Kotlin stands as a viable option in the domain of Big Data, offering developers a language capable of addressing the complexities inherent in large-scale data processing and analytics.

9) High-Frequency Trading Spaces:

In the high-frequency trading arena, Kotlin proves to be a strategic choice, affording developers the tools necessary to navigate the intricacies of rapid and data-intensive trading environments.

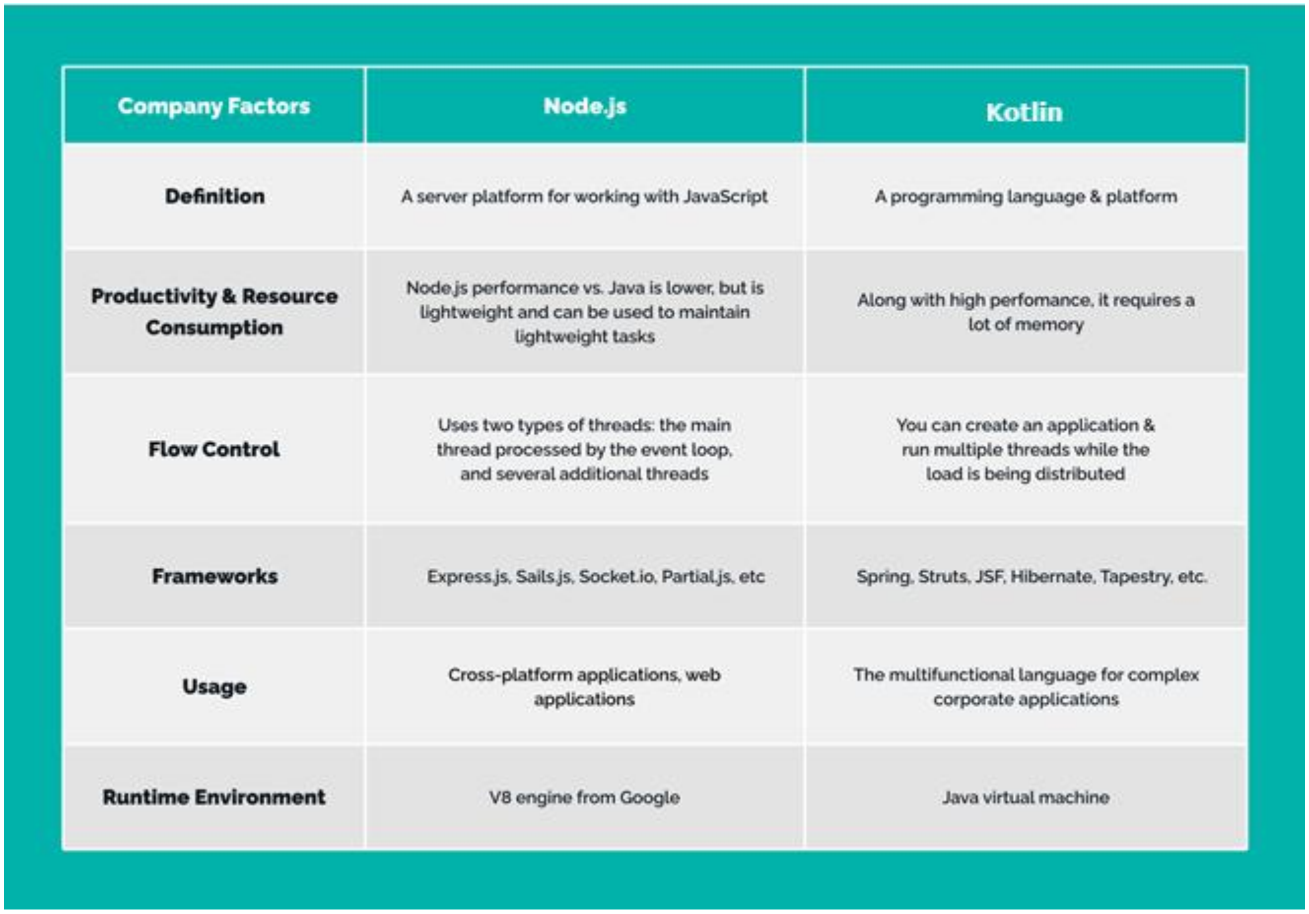
10) Scientific Applications:

Kotlin lends itself to the development of scientific applications, catering to scenarios necessitating precise computation, data analysis, and complex algorithmic implementations.

In summary, the nuanced attributes of Kotlin render it a versatile programming language, adeptly applied across diverse domains, from mobile and web development to financial services, software tools, and specialized environments such as high-frequency trading and scientific applications.

3.2.7 Key differences between Node.js and Kotlin

Below is a comparative table of Node.js vs. Kotlin to demonstrate their distinctions more clearly.



Company Factors	Node.js	Kotlin
Definition	A server platform for working with JavaScript	A programming language & platform
Productivity & Resource Consumption	Node.js performance vs. Java is lower, but is lightweight and can be used to maintain lightweight tasks	Along with high perfomance, it requires a lot of memory
Flow Control	Uses two types of threads: the main thread processed by the event loop, and several additional threads	You can create an application & run multiple threads while the load is being distributed
Frameworks	Express.js, Sails.js, Socket.io, Partial.js, etc	Spring, Struts, JSF, Hibernate, Tapestry, etc.
Usage	Cross-platform applications, web applications	The multifunctional language for complex corporate applications
Runtime Environment	V8 engine from Google	Java virtual machine

Fig. 3.2. Key differences between Node.js and Kotlin

3.2.8 Performance Evaluation: Kotlin vs Node.js

In a comparative analysis of Kotlin and Node.js performance, data sourced from a Hackernoon report unveils substantial improvements achieved by integrating Node.js, as exemplified by PayPal and Netflix reducing startup times from over 40 minutes to under a minute. This remarkable boost in speed, cutting loading times by 50%-60%, underscores Node.js's prowess in enhancing application performance.

The intrinsic event-driven architecture of Node.js ensures effective multitasking, contributing to the acceleration of application speed. Augmenting its efficiency is the utilization of the V8 JavaScript engine, renowned for its rapid and effective execution.

In the Node.js vs Kotlin performance context, Node.js emerges as the superior performer. The asynchronous and non-blocking inheritance within Node.js fosters an environment conducive to executing minor operations without impeding the primary application thread. The multitasking capabilities further enhance performance, and the adoption of the fastest JavaScript engine contributes to the preservation of server speed, effectively managing substantial data flows.

Conversely, Kotlin, as per the Kotlin vs Node.js performance comparison, demonstrates swifter performance based on the test results. The statically typed nature of Java facilitates quicker and more precise construction, minimizing errors, and the Just-In-Time (JIT) compiler, collaborating with the Java Virtual Machine (JVM), enhances overall performance. Java's adherence to strict writing and error-handling practices establishes a predictable and robust workflow, particularly advantageous in complex applications.

Despite Kotlin's commendable performance, Node.js stands out in terms of speed, runtime efficiency, and chunked data output. Kotlin, reliant on compilers, experiences relative slowness due to bytecode compilation, and its garbage collection function, while beneficial in certain contexts, poses memory challenges. In contrast, Node.js, with its non-buffering approach, ensures swifter runtime and streamlined data output.

3.2.9 Future Trajectories of Kotlin vs Node.js

The anticipated trajectory of Node.js points towards continued expansion in web and mobile application development. Renowned for its adept handling of real-time data, scalability, and cross-platform compatibility, Node.js remains a valuable asset for businesses seeking a competitive edge.

In the enterprise space, Kotlin, characterized by an established and robust ecosystem, is poised to sustain its formidable presence. With a history of reliable performance, robust community support, and an extensive array of libraries and tools, Kotlin remains an attractive choice for constructing large-scale, intricate applications.

Conclusion

In conclusion, the comparative analysis of Node.js and Kotlin reveals nuanced advantages and disadvantages. Kotlin, universal language, boasts complexity in contrast to the speed and lightness of Node.js. While Node.js excels in speed, its single-threaded nature is compensated by agility and efficiency. Both technologies exhibit strengths in diverse application scenarios, emphasizing their respective roles in real-time applications, streaming, Android apps, large data projects, online shopping, scientific software, trading software, and e-commerce.

4.3 Justification for choosing Kotlin for full-stack development

In recent years, the landscape of full-stack development has witnessed a paradigm shift, with developers exploring alternatives to traditional programming languages. Kotlin, a statically-typed programming language developed by JetBrains, has gained considerable attention due to its pragmatic design and seamless integration across various platforms. This article delves into the reasons behind the increasing adoption of Kotlin in full-stack development and provides a nuanced perspective on its advantages.

4.3.1 Conciseness and Readability

Kotlin's concise syntax and enhanced readability are critical factors contributing to its popularity in full-stack development. The language reduces boilerplate code, thereby improving developer productivity and minimizing the likelihood of errors. By embracing modern programming constructs, such as lambda expressions and extension functions, Kotlin facilitates the creation of clean, maintainable code, which is essential for the success of full-stack projects.

4.3.2 Interoperability

One of Kotlin's distinctive features is its interoperability with existing Java codebases. This characteristic is particularly advantageous in full-stack development scenarios where legacy systems or libraries are prevalent. Kotlin seamlessly integrates with Java, allowing developers to leverage existing investments in Java-based solutions while gradually transitioning to a more modern and expressive language.

4.3.3 Safety and Nullability

Kotlin addresses common programming pitfalls related to nullability through its type system, providing enhanced safety features compared to other languages. The introduction of nullable and non-nullable types reduces the likelihood of null pointer exceptions, a prevalent source of runtime errors. In a full-stack development environment, where reliability is paramount, Kotlin's focus on null safety contributes significantly to the robustness of applications.

4.3.4 Coroutines for Asynchronous Programming

Asynchronous programming is a crucial aspect of modern full-stack development, especially when dealing with network requests and concurrent operations. Kotlin introduces coroutines, a lightweight and efficient concurrency mechanism, simplifying asynchronous code implementation. This feature enhances code readability and maintainability, making Kotlin an attractive choice for full-stack developers grappling with complex asynchronous workflows.

4.3.5 Tooling and Community Support

An integral aspect of selecting a programming language for full-stack development is the availability of robust tooling and a supportive community. Kotlin benefits from excellent tooling support, including a feature-rich Integrated Development

Environment (IDE) and a comprehensive set of libraries. Additionally, the language boasts an active and growing community, fostering knowledge exchange and continuous improvement.

Conclusion

In conclusion, the adoption of Kotlin in full-stack development is justified by its concise syntax, readability, interoperability with Java, safety features, support for asynchronous programming, and a thriving community. This article has presented an academic exploration of these factors, establishing Kotlin as a compelling choice for developers aiming to build robust and scalable full-stack applications. As the software development landscape continues to evolve, Kotlin's versatility positions it as a language of choice for addressing the dynamic requirements of contemporary full-stack development projects.

CONCLUSIONS

In the pursuit of advancing full-stack development methodologies, this research has undertaken a comprehensive exploration of the Kotlin programming language's efficacy as a preferred choice for full-stack development. The examination began with an elucidation of the fundamental concepts of full-stack development and extended to the specific realm of full-stack mobile application development. The distinct features of Kotlin, such as conciseness, interoperability, null safety, and coroutines, were scrutinized in detail, laying the foundation for a nuanced understanding of its suitability for full-stack projects.

The methodology employed in this research involved the practical application of Kotlin in the development of a mobile application using Android SDK and Ktor for the server side. The integration of these components allowed for a firsthand evaluation of Kotlin's effectiveness in the context of full-stack development. Through a systematic analysis, this research sought to contribute valuable insights into the practical implications of adopting Kotlin for building robust and scalable full-stack software solutions.

A critical juncture in this exploration involved a comparative analysis with other prominent programming languages in the full-stack development arena. The juxtaposition of Kotlin with Java and JavaScript elucidated the distinctive advantages offered by Kotlin, solidifying its position as a language uniquely tailored to meet the dynamic challenges of full-stack development. The research highlighted Kotlin's conciseness and compatibility with Java as differentiating factors, addressing concerns that have traditionally hindered the adoption of other languages.

In light of the research findings, several recommendations emerge for developers and organizations venturing into full-stack development. Firstly, the adoption of Kotlin is recommended for its ability to enhance productivity through concise syntax and improved readability. Its interoperability with Java offers a strategic advantage, allowing a seamless transition from existing codebases. The emphasis on null safety and the introduction of coroutines contribute to the creation of more reliable and efficient asynchronous workflows.

Furthermore, the community support and robust tooling surrounding Kotlin are crucial factors that contribute to a conducive development environment. Developers are encouraged to leverage the resources available within the Kotlin community and explore the extensive tooling options to streamline the development process.

In conclusion, the results of this research affirm that Kotlin is a well-founded choice for full-stack development, providing a balanced amalgamation of practicality, expressiveness, and efficiency. As the software development landscape continues to evolve, Kotlin's adaptability positions it as a language capable of meeting the diverse and evolving demands of modern full-stack projects. This research aims to serve as a guiding resource for developers and organizations, offering empirical insights into the advantages of incorporating Kotlin into the fabric of their full-stack development endeavors.

REFERENCES

1. Kotlin Programming Language Documentation. 2023. [Electronic resource] – Mode of access: https://kotlinlang.org/docs/

2. Android Developer Documentation. 2023. [Electronic resource] – Mode of access: https://developer.android.com/docs

3. Ktor Documentation. 2023. [Electronic resource] – Mode of access: https://ktor.io/docs/

4. Full-stack conception. [Electronic resource] – Mode of access: https://www.simplilearn.com/what-is-full-stack-development-article

5. Full stack mobile conception. [Electronic resource] – Mode of access: https://medium.com/@usermediamora111/full-stack-mobile-development-123b9eae29fc

6. Kotlin features. [Electronic resource] – Mode of access: https://www.simplilearn.com/tutorials/kotlin-tutorial/what-is-kotlin-programming-language

7. Android app development processes. [Electronic resource] – Mode of access: https://arounda.agency/blog/a-step-by-step-guide-on-mobile-app-development-process

8. Backend development processes. [Electronic resource] – Mode of access: https://medium.com/@moeburney/the-seven-steps-of-backend-development-for-mobile-applications-6fdcf31d6079

9. Kotlin with Java comparison. [Electronic resource] – Mode of access: https://www.techmagic.co/blog/kotlin-vs-java/

10. Kotlin with JavaScript comparison. [Electronic resource] – Mode of access: https://www.techmagic.co/blog/node-js-vs-java-what-to-choose/

APPENDIX A

List of the mobile app source code

AuthRepositoryImpl.kt

```
package com.glechyk.coffee44road.data.repository

import android.content.SharedPreferences
import com.glechyk.coffee44road.data.model.auth.AuthRequestData
import com.glechyk.coffee44road.data.model.auth.AuthResult
```

```kotlin
import com.glechyk.coffee44road.data.network.AuthApi
import com.glechyk.coffee44road.domain.repository.AuthRepository
import javax.inject.Inject

class AuthRepositoryImpl @Inject constructor(
    private val api: AuthApi,
    private val prefs: SharedPreferences,
) : AuthRepository {

    override suspend fun signUp(username: String, password: String):
AuthResult<Unit> {
        val response =
            api.signUp(
                request = AuthRequestData(
                    username = username,
                    password = password
                )
            )
        return if (response.isSuccessful) {
            signIn(username = username, password = password)
            AuthResult.Authorized()
        } else {
            AuthResult.UnknownError()
        }
    }

    override suspend fun signIn(username: String, password: String):
AuthResult<Unit> {
        val response =
            api.signIn(
                request = AuthRequestData(
                    username = username,
                    password = password
                )
            )
        val token = response.body()?.token
        return if (response.isSuccessful && token != null) {
            prefs.edit().putString("jwt", token).apply()
            AuthResult.Authorized()
        } else {
            AuthResult.UnknownError()
        }
    }

    override suspend fun authenticate(): AuthResult<Unit> {
        return try {
            val token = prefs.getString("jwt", null) ?: return
AuthResult.Unauthorized()
            val response = api.authenticate("Bearer $token")
            if (response.isSuccessful) {
                return AuthResult.Authorized()
            } else {
                return AuthResult.Unauthorized()
            }
        } catch (e: Exception) {
            AuthResult.UnknownError()
        }
    }
}
```

AuthApi.kt

```
package com.glechyk.coffee44road.data.network

import com.glechyk.coffee44road.data.model.auth.AuthRequestData
import com.glechyk.coffee44road.data.model.auth.TokenResponse
import retrofit2.Response
import retrofit2.http.Body
import retrofit2.http.GET
import retrofit2.http.Header
import retrofit2.http.POST

interface AuthApi {

    @POST(SIGN_UP_URL)
    suspend fun signUp(
        @Body request: AuthRequestData,
    ): Response<Unit>

    @POST(SIGN_IN_URL)
    suspend fun signIn(
        @Body request: AuthRequestData,
    ): Response<TokenResponse>

    @GET(AUTHENTICATE_URL)
    suspend fun authenticate(
        @Header(AUTHORIZATION_HEADER) token: String,
    ): Response<Unit>
}
```

AuthRequestData.kt

```kotlin
package com.glechyk.coffee44road.data.model.auth

data class AuthRequestData(
    val username: String,
    val password: String,
)
```

TokenResponse.kt

```kotlin
package com.glechyk.coffee44road.data.model.auth

data class TokenResponse(
    val token: String,
)
```

AuthResult.kt

```kotlin
package com.glechyk.coffee44road.data.model.auth

sealed class AuthResult<T>(val data: T? = null) {

    class Authorized<T>(data: T? = null) : AuthResult<T>(data)

    class Unauthorized<T>() : AuthResult<T>()

    class UnknownError<T>() : AuthResult<T>()
}
```

AuthViewModel.kt

```kotlin
package com.glechyk.coffee44road

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.glechyk.coffee44road.data.model.auth.AuthResult
import com.glechyk.coffee44road.domain.repository.AuthRepository
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class AuthViewModel @Inject constructor(
    private val authRepository: AuthRepository,
) : ViewModel() {

    private val _authState = MutableStateFlow<AuthState?>(null)
    val authState = _authState.asStateFlow()

    init {
        viewModelScope.launch {
            when (authRepository.authenticate()) {
                is AuthResult.Authorized -> _authState.update {
AuthState.AUTHORIZED }
                is AuthResult.Unauthorized -> _authState.update {
AuthState.UNAUTHORIZED }
                is AuthResult.UnknownError -> {}
            }
        }
    }
}

enum class AuthState { AUTHORIZED, UNAUTHORIZED }
```

APPENDIX B

List of backend server source code

Application.kt

```
package com.glechyk

import com.glechyk.data.user.MongoUserDataSource
import com.glechyk.plugins.configureMonitoring
```

```kotlin
import com.glechyk.plugins.configureRouting
import com.glechyk.plugins.configureSecurity
import com.glechyk.plugins.configureSerialization
import com.glechyk.security.hashing.SHA256HashingService
import com.glechyk.security.token.JwtTokenService
import com.glechyk.security.token.TokenConfig
import io.ktor.server.application.*
import org.litote.kmongo.coroutine.coroutine
import org.litote.kmongo.reactivestreams.KMongo

fun main(args: Array<String>) {
    io.ktor.server.netty.EngineMain.main(args)
}

fun Application.module() {
    val mongoPassword = System.getenv("MONGO_PASSWORD")
    val dbName = "coffee44road-auth"
    val db = KMongo.createClient(
        connectionString =
"mongodb+srv://glechyk:$mongoPassword@cluster0.krdxjzq.mongodb.net/$dbName?retryWr
ites=true&w=majority"
    ).coroutine.getDatabase(dbName)
    val userDataSource = MongoUserDataSource(db)
    val tokenService = JwtTokenService()
    val tokenConfig = TokenConfig(
        issuer = environment.config.property("jwt.issuer").getString(),
        audience = environment.config.property("jwt.audience").getString(),
        expiresIn = 365L * 1000L * 60L * 60L * 24L,
        secret = System.getenv("JWT_SECRET")
    )
    val hashingService = SHA256HashingService()

    configureSerialization()
    configureMonitoring()
    configureSecurity(tokenConfig)
    configureRouting(userDataSource, hashingService, tokenService, tokenConfig)
}
```

MongoUserDataSource.kt

```kotlin
package com.glechyk.data.user

import org.litote.kmongo.coroutine.CoroutineDatabase
import org.litote.kmongo.eq

class MongoUserDataSource(
    db: CoroutineDatabase
): UserDataSource {

    private val users = db.getCollection<User>()

    override suspend fun getUserByUsername(username: String): User? {
        return users.findOne(User::username eq username)
    }
    override suspend fun insertUser(user: User): Boolean {
        return users.insertOne(user).wasAcknowledged()
    }
}
```

TokenService.kt

```kotlin
package com.glechyk.security.token

interface TokenService {

    fun generate(
        config: TokenConfig,
        vararg claims: TokenClaim,
    ): String
}
```

Routing.kt

```kotlin
package com.glechyk.plugins

import com.glechyk.authenticate
import com.glechyk.data.user.UserDataSource
import com.glechyk.getSecretInfo
import com.glechyk.security.hashing.HashingService
import com.glechyk.security.token.TokenConfig
import com.glechyk.security.token.TokenService
import com.glechyk.signIn
import com.glechyk.signUp
import io.ktor.server.application.*
import io.ktor.server.routing.*

fun Application.configureRouting(
    userDataSource: UserDataSource,
    hashingService: HashingService,
    tokenService: TokenService,
    tokenConfig: TokenConfig,
) {
    routing {
        signIn(userDataSource, hashingService, tokenService, tokenConfig)
        signUp(hashingService, userDataSource)
        authenticate()
        getSecretInfo()
    }
}
```

Security.kt

```kotlin
package com.glechyk.plugins

import com.auth0.jwt.JWT
import com.auth0.jwt.algorithms.Algorithm
import com.glechyk.security.token.TokenConfig
import io.ktor.server.application.*
import io.ktor.server.auth.*
import io.ktor.server.auth.jwt.*

fun Application.configureSecurity(config: TokenConfig) {
    authentication {
        jwt {
            realm =
this@configureSecurity.environment.config.property("jwt.realm").getString()
            verifier(
                JWT
                    .require(Algorithm.HMAC256(config.secret))
                    .withAudience(config.audience)
                    .withIssuer(config.issuer)
                    .build()
            )
            validate { credential ->
                if (credential.payload.audience.contains(config.audience)) {
                    JWTPrincipal(credential.payload)
                } else null
            }
        }
    }
}
```

JwtTokenService.kt

```
package com.glechyk.security.token

import com.auth0.jwt.JWT
import com.auth0.jwt.algorithms.Algorithm
import java.util.*

class JwtTokenService: TokenService {

    override fun generate(config: TokenConfig, vararg claims: TokenClaim): String
{
        var token = JWT.create()
            .withAudience(config.audience)
            .withIssuer(config.issuer)
            .withExpiresAt(Date(System.currentTimeMillis() + config.expiresIn))
        claims.forEach { claim ->
            token = token.withClaim(claim.name, claim.value)
        }
        return token.sign(Algorithm.HMAC256(config.secret))
    }
}
```

SHA256HashingService.kt

```
package com.glechyk.security.hashing

import org.apache.commons.codec.binary.Hex
import org.apache.commons.codec.digest.DigestUtils
import java.security.SecureRandom

class SHA256HashingService: HashingService {

    override fun generateSaltedHash(value: String, saltLength: Int): SaltedHash {
        val salt = SecureRandom.getInstance("SHA1PRNG").generateSeed(saltLength)
        val saltAsHex = Hex.encodeHexString(salt)
        val hash = DigestUtils.sha256Hex("$saltAsHex$value")
        return SaltedHash(
            hash = hash,
            salt = saltAsHex
        )
    }

    override fun verify(value: String, saltedHash: SaltedHash): Boolean {
        return DigestUtils.sha256Hex("${saltedHash.salt}$value") ==
saltedHash.hash
    }
}
```

AuthRouting.kt

```
package com.glechyk

import com.glechyk.data.requests.AuthRequest
import com.glechyk.data.responses.AuthResponse
import com.glechyk.data.user.User
import com.glechyk.data.user.UserDataSource
import com.glechyk.security.hashing.HashingService
import com.glechyk.security.hashing.SaltedHash
import com.glechyk.security.token.TokenClaim
import com.glechyk.security.token.TokenConfig
import com.glechyk.security.token.TokenService
import io.ktor.http.*
import io.ktor.server.application.*
import io.ktor.server.auth.*
import io.ktor.server.auth.jwt.*
import io.ktor.server.request.*
import io.ktor.server.response.*
import io.ktor.server.routing.*

fun Route.signUp(
    hashingService: HashingService,
    userDataSource: UserDataSource,
) {
    post("signup") {
        val request = call.receiveNullable<AuthRequest>() ?: run {
            call.respond(HttpStatusCode.BadRequest)
            return@post
        }

        val areFieldsBlank = request.username.isBlank() ||
request.password.isBlank()
        val isPasswordTooShort = request.password.length < 8
        if (areFieldsBlank || isPasswordTooShort) {
```

```kotlin
            call.respond(HttpStatusCode.Conflict)
            return@post
        }
        val isUserExist = userDataSource.getUserByUsername(request.username)
        if (isUserExist != null) {
            call.respond(HttpStatusCode.Conflict, "User with this name already
exists")
            return@post
        }

        val saltedHash = hashingService.generateSaltedHash(request.password)
        val user = User(
            username = request.username,
            password = saltedHash.hash,
            salt = saltedHash.salt
        )

        val wasAcknowledged = userDataSource.insertUser(user)
        if (!wasAcknowledged) {
            call.respond(HttpStatusCode.Conflict) //server side error
            return@post
        }

        call.respond(HttpStatusCode.OK)
    }
}

fun Route.signIn(
    userDataSource: UserDataSource,
    hashingService: HashingService,
    tokenService: TokenService,
    tokenConfig: TokenConfig,
) {
    post("signin") {
        val request = call.receiveNullable<AuthRequest>() ?: run {
            call.respond(HttpStatusCode.BadRequest)
            return@post
        }

        val user = userDataSource.getUserByUsername(request.username)
        if (user == null) {
            call.respond(HttpStatusCode.Conflict, "Incorrect username")
            return@post
        }

        val isValidPassword = hashingService.verify(
            value = request.password,
            saltedHash = SaltedHash(
                hash = user.password,
                salt = user.salt,
            )
        )
        if (!isValidPassword) {
            call.respond(HttpStatusCode.Conflict, "Incorrect password")
            return@post
        }
        val token = tokenService.generate(
            config = tokenConfig,
            TokenClaim(
                name = "userId",
                value = user.id.toString()
            ),
        )

        call.respond(
            status = HttpStatusCode.OK,
            message = AuthResponse(
                token = token
            )
        )
    }
}

fun Route.authenticate() {
    authenticate {
        get("authenticate") {
            call.respond(HttpStatusCode.OK)
        }
    }
}

fun Route.getSecretInfo() {
    authenticate {
        get("secret") {
            val principal = call.principal<JWTPrincipal>()
            val userId = principal?.getClaim("userId", String::class)
            call.respond(HttpStatusCode.OK, "Your userId is $userId")
        }
    }
}
```