Journal of Electrical Engineering, Electronics, Control and Computer Science – JEEECCS, Volume 9, Issue 33, pages 31-38, 2023

Scalable Software Development with Microservices

Agharese Rosemary Usiobaifo
Department of Computer Science, Faculty of Physical Sciences, University of Benin, P.M.B 1154, Benin City Nigeria
rosemary.usiobaifo@uniben.edu

Roseline Oghogho Osaseri
Department of Computer Science, Faculty of Physical Sciences, University of Benin, P.M.B 1154, Benin City Nigeria
roseline.osaseri@uniben.edu

***Abstract* – Microservices is a software architecture that allows for the development and deployment of independently deployable, modular services. This approach to software development is designed to be scalable, making it particularly well-suited for large, complex systems that require the ability to handle a high volume of traffic or data. In this study, we demonstrated the benefits of using microservices for scalable software development, including the ability to deploy and update individual services without disrupting the entire system, and the ability to easily scale specific components of the system as needed. We also discuss some of the challenges and considerations involved in implementing a microservices architecture, including the need for robust communication and integration between services and the potential for increased complexity in the development process. Overall, microservices offer a promising approach to building scalable, maintainable software systems in today's fast-paced, data-driven world.**

Keywords- Microservices; Scalability; Modular services; Deployment; Complexity

I. Introduction

In recent years, microservices architecture has gained significant attention as a software design pattern for building scalable, maintainable, and fault-tolerant applications. This approach involves decomposing a large, monolithic application into a set of small, independent services that communicate with each other through well-defined interfaces. The microservices architecture enables teams to work on individual services independently, allowing for faster development and deployment of new features, as well as easier scaling of individual services.

The University of Benin, like many other educational institutions, faces the challenge of managing a large and complex student information system. While the West Africa e-University Project (Kofa) portal currently used by the University provides extensive features such as academic record management, result processing, and payment processing, it has limitations in terms of meeting the needs of the large student and staff populations. To address these limitations, this paper proposes the use of microservices architecture to extend the current students' portal of the University of Benin.

The main objective of this research is to reduce the probability of an outage on the portal, remove single points of failure and other vulnerabilities, enhance the experience of students and staff of the university, remove the constraints around elections, evaluations, and fee payments on campus, and make the source code of the portal more accessible to students for contributions. The paper also highlights the technical flaws of the current Kofa application and how the use of microservices architecture can overcome these flaws.

Overall, this research aims to provide a comprehensive understanding of microservices architecture and its application in the context of extending the University of Benin students' portal. The findings of this study can be valuable for software engineers, architects, and educational institutions interested in designing scalable and maintainable software systems.

II. Related Work

Monolithic architecture is a software design where all components of an app are built and deployed together as a single unit, making them large, complex, and difficult to maintain and scale. All application components are tightly coupled, making it challenging to change one part without affecting other parts. Moreover, testing and deploying monolithic architectures can be difficult, as changes to the codebase require rebuilding and redeploying the entire application. Monolithic architectures, while simple to develop and understand, have become less popular due to their difficulty in maintaining and scaling as applications become more complex. In contrast, the microservices architecture has gained popularity as it allows for individual services to be developed and deployed independently, promoting the principles of separation of concerns and single responsibility. Microservices allow for decentralized governance and data management, enabling cross-functional teams to take ownership of specific services and choose the best technology stacks for their needs. Microservices were proposed by [8] as a solution to the frustrations of monolithic systems, and have since become a popular architectural style for modern software applications.

A. *Motivations for Adopting Microservices Architecture*

Reference [11] conducted a survey of practitioners who adopted the microservices architecture and found that motivations included easier scaling, the delegation of team responsibilities, and support for CI/CD, fault tolerance, and experimentation with new technologies.

However, respondents also reported challenges in decoupling monolithic applications, managing intercommunication between services, and estimating programming effort. Additionally, the benefits of remote service deployment in microservices require additional DevOps efforts and can increase software deployment costs.

Reference [10] conducted a systematic review of written and video materials from technology companies that have adopted the microservices architecture, highlighting the pains and gains of the architecture and providing recommendations for future research directions. Their research provides a foundation for further academic exploration of the topic.

In reference [12] developed an application based on microservices gaining attraction over monolithic applications. An approach for the development and deployment of applications based on resources: microservices technology software architecture, a continuous integration framework and environment for the deployment of microservices with high scalability and availability

Reference [13] conducted a survey on the decomposition of multi-level scalability assessment. The results of an-in-depth evaluation so that the approach can effectively support engineers in decomposing monolithic or coarse grained microservices into scalable microservices.

B. *Migrating a Legacy Application from Monolithic to Microservices Architecture*

The template is used to format your paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings, and not as an independent document. Please do not revise any of the current designations.

The transition from a monolithic design to a microservices architecture can be a difficult and time-consuming process. Each legacy monolithic program is unique, and the specific conversion issues will differ depending on the application's features. There are various methodologies and techniques for migrating a monolithic program to microservices, each having advantages and disadvantages. The purpose of migration, in general, is to break the monolithic application into a series of separate, modular services that can be created, deployed, and scaled independently. This might need major application re-architecting, which can be a time-consuming and difficult procedure. The final result, however, is frequently a more adaptable, scalable, and robust application that can better meet the changing demands of the company. Reference [5] reviewed different migration techniques. They highlighted their benefits and drawbacks as captured and represented in table 1 below

TABLE I. REVIEW OF DIFFERENT MIGRATION TECHNIQUES

S/N	Method	Benefits	Drawbacks
1	Mazlami *et al* (2019) proposed a microservice extraction model with a tool for structured service decomposition using graph cutting. It recommends microservice candidates based on 16 coupling criteria from literature and industry know-how, using input from software engineering artifacts.	Approach scales with revision history size. The proposed approach reduces microservice team size.	The extraction model is limited to using classes as the atomic unit, but using methods or functions could improve granularity and precision.
2	Knoche et al (2019) proposed a five-step migration process to decompose an application into microservices, including defining external and internal service facades and replacing service implementations with microservices.	Improve interfaces, reduce entry points, and remove redundancy.	Proprietary UI tech may not work with a modernization approach.
3	Dehghani (2018) proposed a formal migration process from monolith to microservice architecture consisting of principles such as minimizing dependency back to the monolith, splitting sticky capabilities early, decoupling vertically, and releasing data early. The process involves going macro first, then micro, and migrating in atomic evolutionary steps.	The approach allows safe, incremental migration.	The migration process is very long and formal without measurements.
4	Fan et al proposed a migration process based on SDLC, including all of the methods and tools required during design, development, and implementation	Advantages of microservices architecture: specialized, fault-tolerant, and automated.	Microservices require complex configurations and use more resources due to the need for multiple tools for flexibility.

Reference [1] proposes a Situational Method Engineering (SME) approach for migrating to a cloud-native architecture, such as microservices. The approach involves creating a repository of reusable process patterns or method chunks, which are then used to construct a specific method for migration based on the requirements and needs of the project. This allows for greater flexibility and adaptability to the unique needs of each migration, as opposed to a one-size-fits-all methodology.

Furthermore, the microservices architecture offers better fault isolation and resiliency than monolithic architecture. If a service fails, it only affects that particular service and not the entire application. This way, the system can continue to function, and the impact of the failure is limited. The independent deployment of services also allows for continuous delivery and deployment, making it easier to roll out updates and new features without disrupting the entire application.

III. System Analysis And Design

This section discusses the importance of system analysis and design in identifying inefficiencies and proposing solutions to improve performance. System analysis involves breaking down the system into smaller parts to understand its behavior and identify problems, while system design defines the architecture and specifications of the system to meet requirements. These processes are crucial in complex system development and inform decision-making and project planning.

A. Architecture of the Current Monolith System

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

As illustrated in figure 1, the current system as a monolith has all the features on the platform as modules in a single application. HTTP requests into the application are handled by a router that processes the request using the URL parameter(s) in the request. The appropriate controller in the concerned module is then used to handle database queries and other required operations.

In this monolithic application, only one database server is usually utilized. Modules can directly access the database to insert, update, read, and delete data. For example, the payment and voting services can directly fetch and modify records in the user table which is relevant to all features of the application.

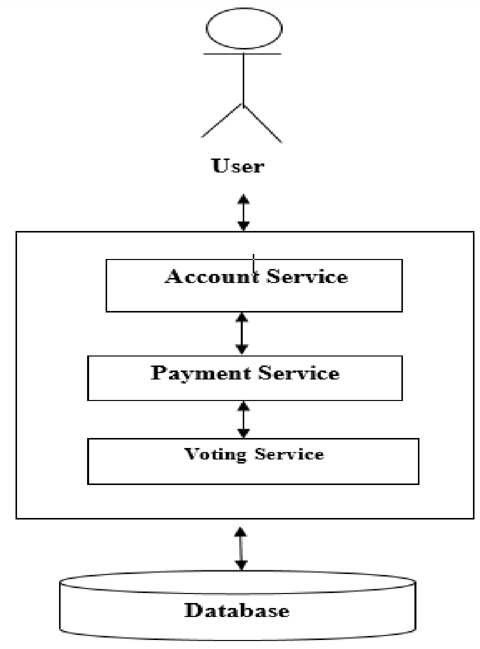


Figure 1. Architecture of a monolithic application

B. Units Architecture of the Microservices System

The proposed microservices architecture is represented in figure 2, requests are handled by the Application Programming Interface (API) gateway. This gateway serves as a bridge among all the services by intercepting incoming requests and routing them to routers in the destination services. The gateway can also be enhanced by introducing a load balancer to efficiently distribute traffic between copies of the vertically scaled services.

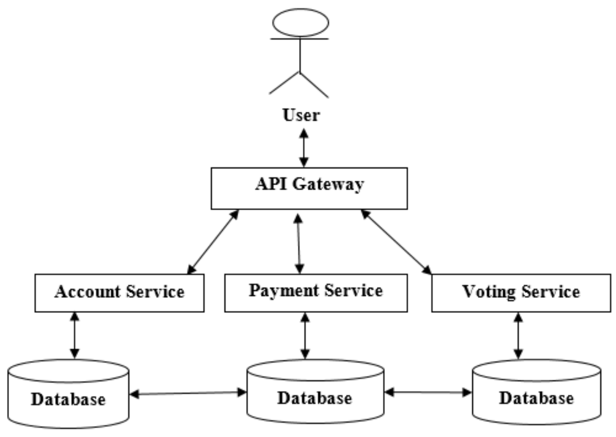


Figure 2. Architecture of a monolithic application

C. Use Case Diagram

Use case diagrams are drawn to visualize high-level system functional requirements. The diagrams control primary actors and use cases. Actors are entities that will interact with the application while use cases are system functions that actors can perform as depicted in Figure 3 below a student as an actor using the system and the functions that the user can perform.

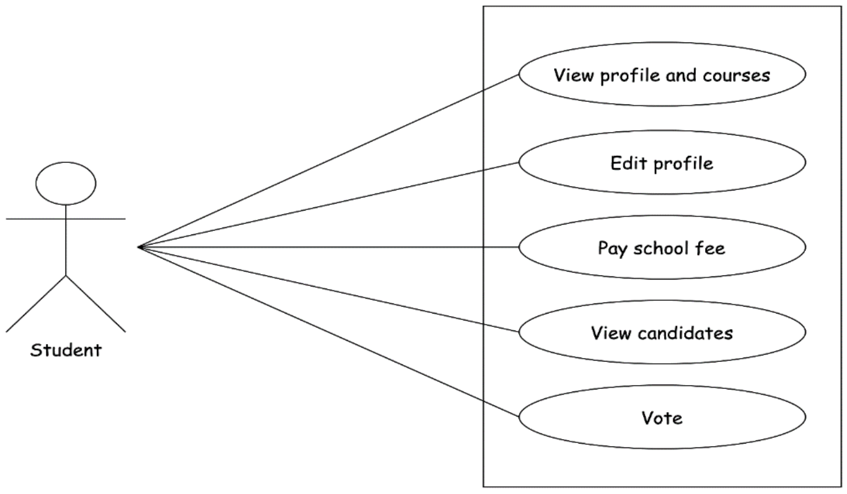


Figure 3. Use case diagram for a student in the proposed system

D. Entity-Relationship Diagrams

Figure 4 represent the database schema for the account services, An Entity Relationship (ER) diagram is a visual tool that depicts the relationships between entities in a database. It represents entities as boxes and their relationships as lines connecting the boxes. An entity refers to an object or concept about which data is stored, and a relationship is a connection between two or more entities. Each entity has attributes that represent its properties. The article also provides an example of an ER diagram illustrating the relationship between tables in an account service: the user table, containing authentication credentials and other required data, and the profile table, containing additional user account data. The figure also shows a one-to-many relationship between a user and fee payments records. This is because each student is expected to pay different fees in the course of their studentship in the university. At the beginning of each session, the school_fee_paid column in the user table is set to false for all students. After successful payment of the school fee, the value is changed to true and the new status is broadcast to all other services for an update. The database normalization by removing redundancy in duplicating user data in the candidates and votes tables. Rather, a one-to-many relationship is established between the user and candidates table, between the user and votes table, and between the candidates and votes tables.

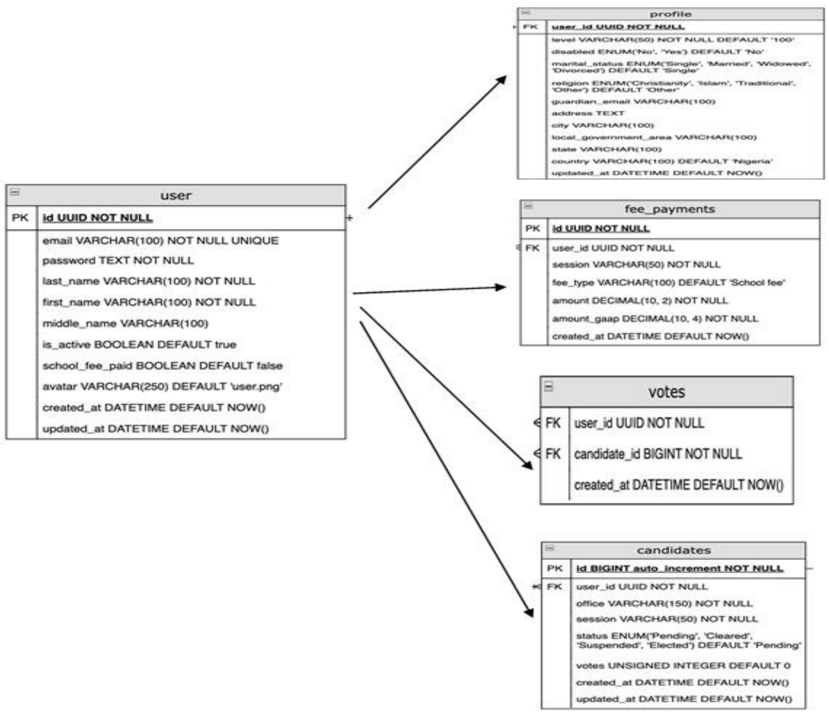

user
PK
id UUID NOT NULL
email VARCHAR(100) NOT NULL UNIQUE
password TEXT NOT NULL
last_name VARCHAR(100) NOT NULL
first_name VARCHAR(100) NOT NULL
middle_name VARCHAR(100)
is_active BOOLEAN DEFAULT true
school_fee_paid BOOLEAN DEFAULT false
avatar VARCHAR(250) DEFAULT 'user.png'
created_at DATETIME DEFAULT NOW()
updated_at DATETIME DEFAULT NOW()
profile
FK
user_id UUID NOT NULL
level VARCHAR(50) NOT NULL DEFAULT '100'
disabled ENUM('No', 'Yes') DEFAULT 'No'
marital_status ENUM('Single', 'Married', 'Widowed', 'Divorced') DEFAULT 'Single'
religion ENUM('Christianity', 'Islam', 'Traditional', 'Other') DEFAULT 'Other'
guardian_email VARCHAR(100)
address TEXT
city VARCHAR(100)
local_government_area VARCHAR(100)
state VARCHAR(100)
country VARCHAR(100) DEFAULT 'Nigeria'
updated_at DATETIME DEFAULT NOW()
fee_payments
PK
id UUID NOT NULL
FK
user_id UUID NOT NULL
session VARCHAR(50) NOT NULL
fee_type VARCHAR(100) DEFAULT 'School fee'
amount DECIMAL(10, 2) NOT NULL
amount_gaap DECIMAL(10, 4) NOT NULL
created_at DATETIME DEFAULT NOW()
votes
FK
user_id UUID NOT NULL
FK
candidate_id BIGINT NOT NULL
created_at DATETIME DEFAULT NOW()
candidates
PK
id BIGINT auto_increment NOT NULL
FK
user_id UUID NOT NULL
office VARCHAR(150) NOT NULL
session VARCHAR(50) NOT NULL
status ENUM('Pending', 'Cleared', 'Suspended', 'Elected') DEFAULT 'Pending'
votes UNSIGNED INTEGER DEFAULT 0
created_at DATETIME DEFAULT NOW()
updated_at DATETIME DEFAULT NOW()

Figure 4. Database schema for the account service

E. Entity-Relationship Diagrams Systems Requirements

System requirements in software development refer to the specific hardware and software requirements that must be met for a software application to function properly. These requirements can vary depending on the complexity and functionality of the application, as well as the operating system and platform it will be used on.

The interface between the user and the system permits data to be submitted by users of the system. Most of the data input is done through the keyboard and mouse or touchpad; this works for the text input, dropdown menu, and checkbox fields.

The input variables in this paper are from the students, they are generally grouped as input data for authentication, profile update, fee payment, and the e-voting form. Output requirements specify the results the application will generate. These include; The student's profile data, courses registered, electoral aspirants, aspirants' candidacy status, voting results, etc

The functional requirements include accepting valid login input, showing authenticated student profile data, allowing profile updates, processing fee payments, displaying a list of electoral aspirants and their candidacy status, and allowing authenticated students to vote after paying fees. Non-functional requirements include extensibility, maintainability, security, resilience in the event of an outage or security breach, and graceful handling of large traffic. The backend features are implemented in TypeScript, is a superset of JavaScript that compiles to plain JavaScript, and it includes features such as type annotations, interfaces, classes, modules, and optional static typing. TypeScript was chosen for its improved code reliability, better code organization, ecosystem support, and improved development experience. The best way for the microservice to work, is to use the backend for frontend design pattern. A popular architectural pattern, as an alternative to the API gateway pattern. Its defining features are that, each separate front-end app has its own dedicated API which communicates with the extended APIs. That way, different frontend apps of varying purpose and structure including those for mobile applications/mobile interfaces or desktop web URL. The frontend of microservices gateway is the whole system APIs of every microservices application in the system. The microservices application which will provide the backend capabilities through exposing API. The database used for this paper implementation is PostgreSQL.

IV. Implementation

This section discusses the system implementation, including the design, building, and deploying of the computer application. The focus is on testing and debugging the functions of the system, including unit testing and integration testing. Unit tests are conducted to verify individual units or components of a system are working correctly, while integration tests are designed to ensure that the different units or components of a system work together as intended. Both types of testing are crucial to ensuring that software meets specified requirements and functions as expected. The TypeScript framework used to build the home page is the NestJS framework. Model-view controller (MVC) pattern was used to render the view and the handlebar template engine. It is a simple templeting language that uses template and an input object to generate html or other text format.

As illustrated in figure 5 the visitors to the application are shown a landing page with navigation links, event updates, and more relevant information.

A. Features of the System

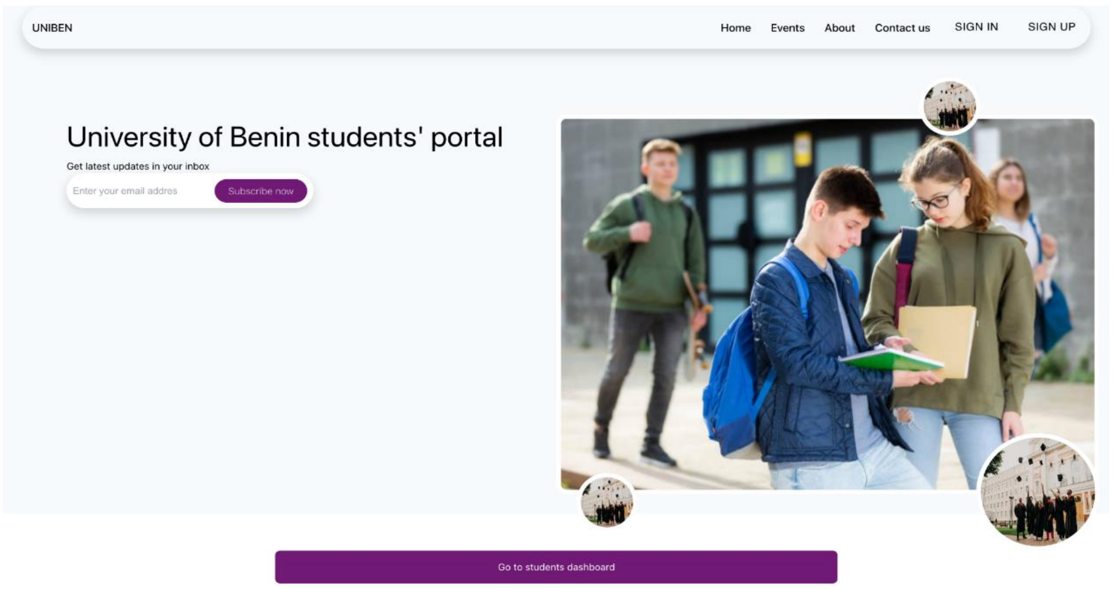


Figure 5. Homepage with navigation links

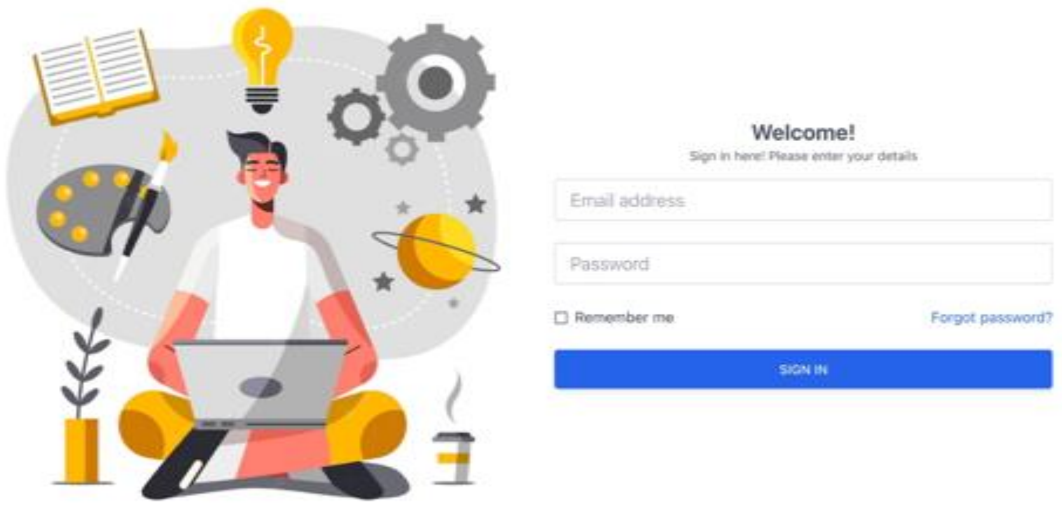
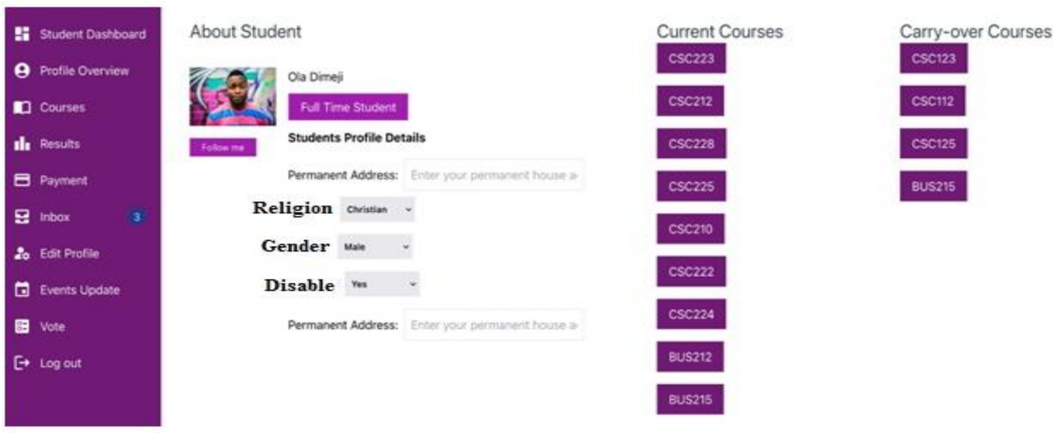

Welcome!
Sign in here! Please enter your details
Email address
Password
☐ Remember me
Forgot password?
SIGN IN

Figure 6. Students' login page

Figure 6 displays an interface through which the user's email address and password are submitted. The credentials entered are validated and the user is granted access to the dashboard if they are correct.



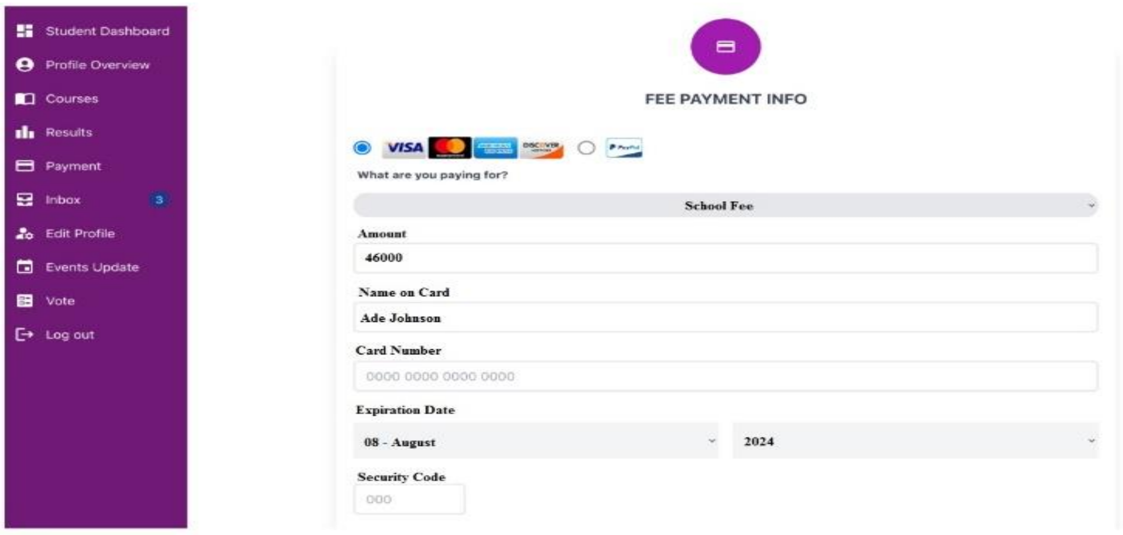
About Student

Ola Dimeji
Full Time Student
Students Profile Details
Permanent Address: Enter your permanent house a
Religion: Christian
Gender: Male
Disable: Yes
Permanent Address: Enter your permanent house a

Current Courses	Carry-over Courses
CSC223	CSC123
CSC212	CSC112
CSC228	CSC125
CSC225	BUS215
CSC210	
CSC222	
CSC224	
BUS212	
BUS215	

Figure 7. Profile info page

The student's profile data are displayed here in figure 7 as well as the registered courses for the session.

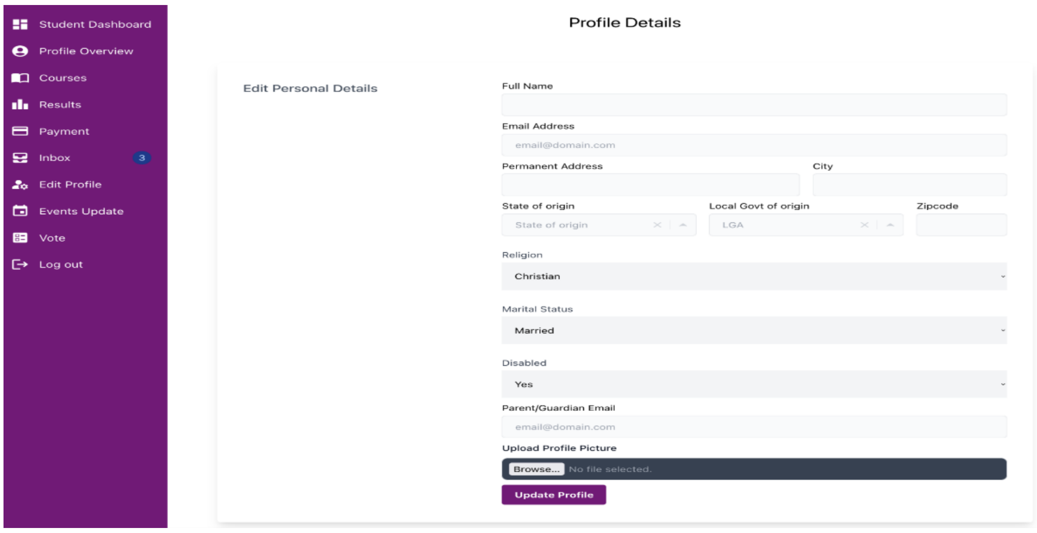


FEE PAYMENT INFO

What are you paying for?: School Fee
Amount: 46000
Name on Card: Ade Johnson
Card Number: 0000 0000 0000 0000
Expiration Date: 08 - August 2024
Security Code: 000

Figure 8. Fee payment page

Figure 8 displays an interface for making fee payments. The student selects the fee type in the dropdown menu and inputs the details of his or her ATM card.



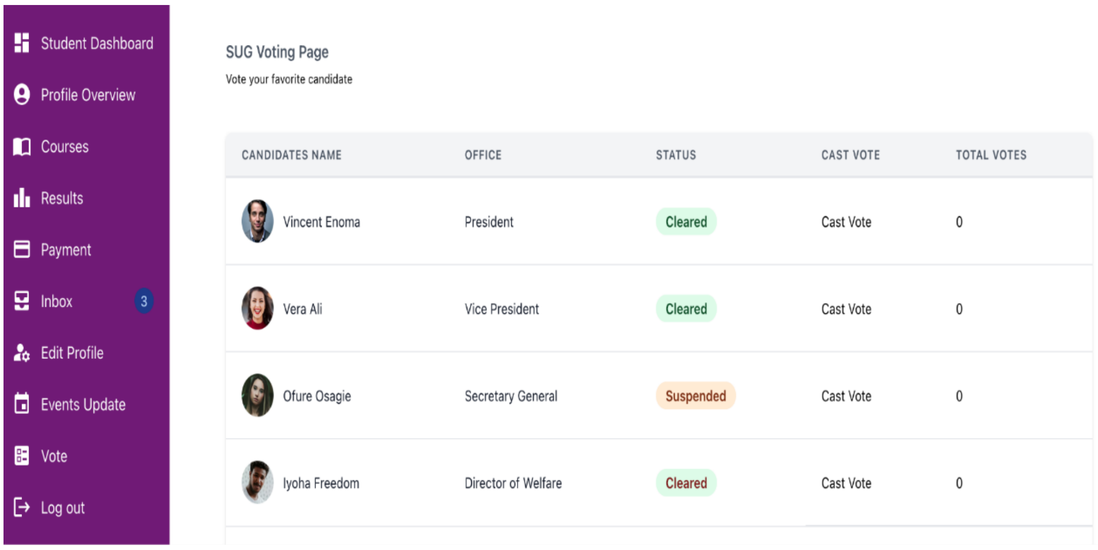
Profile Details

Edit Personal Details

Full Name	
Email Address	email@domain.com
Permanent Address	
City	
State of origin	State of origin
Local Govt of origin	LGA
Zipcode	
Religion	Christian
Marital Status	Married
Disabled	Yes
Parent/Guardian Email	email@domain.com
Upload Profile Picture	No file selected.

Figure 9. Profile update page

Figure 9 presents an interface for the user to update his or her profile information.



SUG Voting Page

Vote your favorite candidate

CANDIDATES NAME	OFFICE	STATUS	CAST VOTE	TOTAL VOTES
Vincent Enoma	President	Cleared	Cast Vote	0
Vera Ali	Vice President	Cleared	Cast Vote	0
Ofure Osagie	Secretary General	Suspended	Cast Vote	0
Iyoha Freedom	Director of Welfare	Cleared	Cast Vote	0

Figure 10. SUG voting page

Students get to vote for their preferred Student Union Government election candidate by clicking on **Cast Vote** in front of the candidate's name as captured in figure 10.

V. CONCLUSIONS

This study aimed to demonstrate the advantages of implementing microservices architecture over a monolithic one in a web-based application, particularly in the student portal of the University of Benin. The paper broke down a monolithic application into smaller, individual services that can be combined to work like a single application. The resulting application allows students to log in, view their profile information, make fee payments, and vote for student government candidates. The benefits of implementing a microservices architecture include flexibility, scalability, and improved control over school management.

It is recommended that the necessity of implementing a microservices architecture should be carefully considered, and the choice of programming language, database system, and other components should be deliberately chosen. Data migration should be handled by experienced database administrators, and at least one competent DevOps engineer should oversee the deployment and maintenance of the application. Finally, the monolithic application should be gradually phased out and replaced with the microservices application, ensuring that all necessary features and data have been migrated.

In summary, the microservices architecture is a more flexible and scalable approach to software development compared to monolithic architecture. It allows for greater agility and faster development times by breaking down the application into smaller, independent services that can be developed and deployed independently. While it comes with its challenges, such as the complexity of managing a distributed system, the benefits of microservices architecture are driving its adoption across.

REFERENCES

[1] A. Balalaie, A. Heydarnoori, A. and P. Jamshidi, “Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture”, IEEE Software, vol. 33, P 3, 2016.

[2] R .Chen, S. Li,. and Z .Li,. "From Monolith to Microservices: A Dataflow-Driven Approach," 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, pp. 466–475, 2017.

[3] Z. Dehghani, “How to break a Monolith into Microservices” [Online].Available:https://martinfowler.com/articles/break-monolith-nto-microservices.html (accesed on 2018).

[4] C Fan,. and S. Ma, "Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report," IEEE International Conference on AI & Mobile Services (AIMS), Honolulu, HI, pp. 109–112, 2017.

[5] J .Kazanavičius,. and D Mažeika, “Migrating Legacy Software to Microservices Architecture”, IEEE Open Conference of Electrical, Electronic and Information Sciences, pp. 1-5, 2019.

[6] H. Knoche, and W .Hasselbring, "Using Microservices for Legacy Software Modernization," IEEE Software, vol. 35, no. 3, pp. 44–49, 2018.

[7] A .Levcovitz., R .Terra. and M. T Valente, “Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems.” 3rd Brazilian Workshop on Software Visualization, Evolution, and Maintenance (VEM), pp. 97–104, 2015.

[8] J. Lewis, and M. Fowler, “Microservices” [Online]. Available: https://martinfowler.com/articles/microservices.html (2018).

[9] G .Mazlami, J. Cito, J. and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," IEEE International Conference on Web Services (ICWS), Honolulu, HI, pp. 524–531, 2017.

[10] J .Soldani, D. A Tamburri, and W.Van Den Heuvel,. “The Pains and Gains of Microservices: A Systematic Grey Literature Review”, Journal of Systems and Software, Volume 146, pp. 215-232. 2018.

[11] D.Taibi, , V. Lenarduzzi and C. Pahl., “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation”, IEEE Cloud Computing, pp. 1-12. 2017.

[12] V.Saquicela, G.Campoverde, J.Avila andM.A. Fajardo, “Building Microservices for Scalability and Availability: Step by Step, fromBegining to the End” In book: New Perspective in Software Engineering. Pp169-184. January 2021.

[13] M.Camelle and C.Calarusso, “Actor-Driven Decomposition of Microservices through Multi-level Scalability Assessment” ACM Transaction on Software Engineering and Methodology. Vol 32, issue 5, 117, pp1-46. July 2023.