CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE · INDIA

Splitting Payment Module to Microservices

by

Joswin Cyril D'Souza
(2047218)

Under the guidance of
Dr. Monisha Singh,
Mr. Sudhir Kumar
&
Mr. Subhan Khan

A Project report submitted in partial fulfillment of the requirements for the award of degree of Master of Computer Applications of CHRIST (Deemed to be University)

May - 2022

CHRIST
(DEEMED TO BE UNIVERSITY)
BANGALORE · INDIA

CERTIFICATE

This is to certify that the report titled ***Splitting Payment Module to Microservices*** *is a bonafide record of work done by* ***Joswin D'Souza (2047218)*** *of CHRIST(Deemed to be University), Bengaluru, in partial fulfillment of the requirements of 4th Semester MCA during the year 2021-22.*

Head of the Department **Project Guide**

Valued-by:

	Name	: Joswin D'Souza
1.	Register Number	: 2047218
	Examination Centre	: CHRIST(Deemed to be University)
2.	Date of Exam	:

Trivium eSolutions

PRIVATE & CONFIDENTIAL

Date: 22nd Sept 2021

To,
Joswin D'Souza (Roll No. : 2047218)
Christ University

Subject: Internship

Dear **Joswin,**

In reference to your application and further interview discussions, we are happy to inform you of your selection for an internship with Trivium eSolutions based in Bangalore, India. The details of the offer extended by Trivium eSolutions are as follow:

- Duration of Internship will be from December 6th, 2021, to 30th June 2022.
- You will be designated as "INTERN" and will be entitled for a stipend of Rs. 25,000/- PM as per company's policy.
- You will be enrolled in multiple training programs to enhance your skills both technical and behavioural.
- You will be assigned a project during this period in a team setting/environment. Further details on the project and underlying technical aspects will be shared with you on commencement of your internship.
- Your performance will be evaluated periodically and over a period of internship. Based on your satisfactory performance, you will be awarded a certificate to show successful completion of your internships.
- During Internship, you are expected to abide to policies prescribed by the company for all its employees and trainees.

Post successful completion of Internship period, you will be hired on Trivium's payroll. Your joining designation would be **Associate software Engineer**. Your employment with us, as an Associate Software Engineer, starts after you obtain the course Completion Certificate/Degree from your college, which would be expected around July 1, 2022. Your Total Compensation (cost to company) as Associate Software Engineer will be Rs.5,00,000/- (Rupees Five Lakhs only) per annum. You would also be entitled to all benefits which are availed by all full-time employees of Trivium. A formal letter of appointment will be issued to you at the time of joining.

Please confirm that the above terms and conditions of this internship agreement are acceptable to you and that you accept the internship by signing the duplicate copy of this letter and return it to us immediately.

We at Trivium are looking forward to welcoming you on board our Bangalore team.

Yours sincerely,
for **Trivium eSolutions Pvt. Ltd.**

Anand

Surbhi Anand
Manager - Human Resources

Joswin Dsouza

Trivium eSolutions

PRIVATE & CONFIDENTIAL

April 11th, 2022

Name: Joshin D'souza
Intern No. : I025

Subject: Internship with Trivium eSolutions Pvt. Ltd.

This is to certify that Joshin D'souza from Christ University is undergoing his internship program with Trivium eSolutions Pvt. Ltd., as an Intern between December 6th, 2022, and June 30th, 2022. His performance and competence during this internship are satisfactory.

Yours sincerely,
for **Trivium eSolutions Pvt. Ltd.**

Surbhi Anand
Manager-HR

Trivium eSolutions Pvt. Ltd. * Bangalore *

ACKNOWLEDGEMENTS

First of all, I thank God almighty for his immense grace and blessings showered on me at every stage of this work.

I am greatly indebted to Dr Joy Paulose, Head, Department of Computer Science, CHRIST (Deemed to be University) for providing the opportunity to take up this project as part of my curriculum. I am greatly indebted to our coordinator, Dr Tulasi B, for providing the opportunity to take up this project and for helping with her valuable suggestions.

I am deeply indebted to our project guide Dr Monisha Singh, for her assistance and valuable suggestions as a guide. He made this project a reality. I am deeply indebted to my industrial supervisor Mr. Sudhir Kumar and Mr. Subhan Khan for rendering support during the project, and all my colleagues for their valuable suggestions and contributions to make this project a reality.

I express my sincere thanks to all faculty members and staff of the Department of Computer Science, CHRIST (Deemed to be University), for their valuable suggestions during the course of this project. Their critical suggestions helped us to improve the project work. Acknowledging the efforts of everyone, their chivalrous help in the course of the project preparation and their willingness to corroborate with the work, their magnanimity through lucid technical details lead to the successful completion of my project.

I would like to express my sincere thanks to all my friends, colleagues, parents and all those who have directly or indirectly assisted during this work.

ABSTRACT

The Project titled ‘Splitting Payment module to Microservices’ is aimed to disintegrate the already built monolithic application to smaller services staring from the Payment Module. The application is for UC Bank which is in Germany and focuses on majorly the European payments under the EBICS Protocol. The application is named as UC eBanking Prime and has over 4500 corporate customers across Germany.

UC Banking Prime is a monolithic application with lots of banking features like Cash management, Payments, Orders, AWV reports, European Gate, Administration, System Administration...etc. The resultant application ends up having a very large code base and poses challenges regarding high module interdependency, scalability, maintainability, deployment, and modifications. To address these challenges and become a hosted web application, analyze, and evaluate the UC Banking Prime in Microservice architecture. As a part the use case, we have taken up the Payment module and have to split the different types of payments into services.

The proposed system aims to split the already running monolithic application UC eBanking Prime into Microservices starting from the Payment Module. Applying different patterns and disintegrating the database according to the services is a major obstacle when the application has 8 lakh lines of Java code itself. And as the application was built in 2008 the technologies used then have also deprecated. So as we are applying Microservices we try to integrate it with the latest technologies and to offer specific parts of the application to client requirements. As most of the clients don’t use all the features of the application offerings separate services is advisable.

TABLE OF CONTENTS

Acknowledgments v

Abstract vi

List of Figures ix

1. Introduction 1

1.1. Organization Profile 1

1.2. Background Study 2

1.2.1 Problem Description 2

1.2.2 Existing System 4

1.2.3 Disadvantages of the existing system 4

1.2.4 Advantages of the proposed system 5

1.2.5 Project Scope 5

2. System Analysis 6

2.1. Requirement Specifications 6

2.1.1. Functional requirements 7

2.1.2. Non-Functional requirements 7

2.2. Block Diagram 8

2.3. System Requirements 10

2.3.1. Hardware requirements 10

2.3.2. Software requirements 12

2.3.3. External Tools 16

3. System Design 19

3.1. System Architecture 19

3.2. Module Design 21

3.3. Use-case Design 22

3.4. Migration Patterns 23

3.5. Decomposing the Database Patterns 30

4. Implementation 37

4.1. Coding Standard 37

4.2. Twelve-Factor Methodology 39
4.3. Screenshots 45
5. Conclusions 47
5.1. Advantages and Limitations 48
References 51

LIST OF FIGURES

Fig. No.	Figure Name	Page No.
2.1	UC eBanking Prime diagram	9
2.2	Microservices Diagram for the Payment Module	10
3.1	Architecture diagram of UC eBanking Prime	20
3.2	Flowchart of Payment Module	21
3.3	Use-case diagram of the bank user	22
3.4	Strangler Fig Example	24
3.5	Strangler Fig Implementation	25
3.6	UI Composition Diagram	25
3.7	The shared database diagram	30
3.8	Database View diagram	32
3.9	Database Wrapping diagram	33
3.10	Database service pattern diagram	35
3.11	Trace Write diagram	36
4.1	UC eBanking Prime Dashboard	45
4.2	Admin rights for the Payment Module	46

1. INTRODUCTION

Introduction deals with the primary objectives and describes the context of the project. This helps to understand the organization profile, and the problem we have identified. This section covers the detailed organization profile, which describes about the organization, their various products and services. Along with the organizational profile, there is the background study that deals with the primary study conducted to understand the problems, thus the background study deals with the problem description, existing system, disadvantages of the existing system, advantages of the proposed system and the scope of the project, all these study gives a clear idea on the need for and importance of the project.

1.1 Organization Profile

Trivium was founded in 2007 with the idea of creating a team of technology enthusiasts who are uniquely positioned to build enterprise-grade solutions and products for customers in the German-speaking Europe.

After several years of organic growth, we strengthened our value proposition with the strategic acquisition of akm software GmbH in 2016. The acquisition added competences in functional areas such as Industrial IoT, and a strong customer base of several international corporations based out of Germany and Switzerland. Since then, we have continued to grow and build our customer base in Europe.

Today, we are a team of over 150 employees with several long-standing customers across business sectors. We offer consulting and software development services, with a strategic focus on digital projects in the infrastructure and industrial domains. Trivium’s projects in the IoT space span the industrial landscape from the machine level to the edge to the cloud.

We build on a distributed delivery model across our offices in Munich and Bangalore. Our management team is based across these two locations and brings in profound experience in software development and Industrial IoT.

Some of the projects developed by Trivium are IoT Analytics Apps for Siemens Rail, Industry 4.0 Platform and HMI for Saurer, Remote Service for Caterpillar Plant Control Systems, Product configuration and order entry for Zeiss Vision Care, eBanking products for UniCredit Bank, Apps for Measurement and Machine Control for Leica, Consulting and Software Development for Deutsche Telekom and more.

1.2 Background Study

Background study helps to understand the problem and the importance of the project. The study helps in understanding the problem, the existing system and the proposed solution. The solution is intended to help the users process and analyze their data more effectively. The NGS analysis that requires a lot of computation and analysis tasks are to be properly planned and analyzed. The background study also helps to understand the importance of the project. The background study contains and covers the important concepts that include the problem description, the existing system study, disadvantages of the existing system, the advantages of the proposed system and the scope and importance of the proposed solution.

1.2.1 Problem Description

A 2018 survey found that 63 percent of enterprises were adopting microservice architectures. This widespread adoption is driven by the promise of improvements in resilience, scalability, time to market, and maintenance, among other reasons.

In a typical monolithic application, all the data objects and actions are handled by a single, tightly knit codebase. Data is typically stored in single database or filesystem. Functions and methods are developed to access the data directly from this storage mechanism, and all business logic is contained within the server codebase and the client application. It is possible to migrate several monolithic applications and/or platforms, each with its own data storage mechanisms, user interfaces, and data schema, into a unified set of

microservices that perform the same functions as the original applications under a single user interface.

Scalability is one of the primary motivations for moving to a microservice architecture. Moreover, the scalability effect on rarely used components is negligible. Components that are used by the most users should therefore be considered for migration first.

Users want their interactions with a system to return the right data at the right level of detail, usually as fast as that data can be acquired. The jobs for users each involve one or more data objects, and each data object has a set of associated actions that can be performed. The development team that designs and implements the system must consider the collection of jobs, data objects, and data actions.

UC Banking Prime is a monolithic application with lots of banking features like Cash management, Payments, Orders, AWV reports, European Gate, Administration, System Administration...etc. The resultant application ends up having a very large code base and poses challenges regarding high module interdependency, scalability, maintainability, deployment, and modifications. To address these challenges and become a hosted web application, analyze, and evaluate the UC Banking Prime in Microservice architecture. As a part the use case, take the Payment module and come up with complete analysis.

Payment modules consists of ISO 20022 standards & SWIFT messages:

1. SEPA Credit (CCC, CCT …etc)
2. SEPA Debit (CDD, CDB)
3. SEPA Instant Payment (CIP)
4. CGI Payments (XC2, XCU)
3. Foreign Payment (AZV, AXZ)
4. International Payment (MT101)
5. European Gate (XEK)

1.2.2 Existing System

UC eBanking prime is the leading multi-tier, multi-bank banking solution from UniCredit for German corporate customers. It is built on EBICS (Electronic Banking Internet Communication Standard) standard, in accordance with the existing „DFÜ-Abkommen“. EBICS enables UC eBanking prime to communicate with all banks that offer EBICS based communication channel to provide the customer with a unified overview of its corporate banking transactions. UC eBanking prime supports major functions of payments and notifications (esp. SEPA) defined by the German banking standards as well as formats of the Austrian and Swiss standards. Further major functionalities cover cash management (e.g. STA, CAMT, value balances, planning data, shadow balances, etc.) and order management (incl. distributed electronic signature) as well as customer compliance regulations like password strength, data change versioning and n-eye-principle. Currently, UC eBanking prime is used by more than 4,500 corporate customers across Germany at their premises and is available in two versions: Enterprise Version and Standard Version.

UC eBanking prime is a standard JEE architecture which supports a browser-based HTML interface. The HTML interface is complemented by a standalone application required for login and signature (primeOTC). Unlike many other electronic banking applications, UC eBanking prime is not a hosted, portal-based solution.

1.2.3 Disadvantages of the existing system

The existing system has the following disadvantages:

- Tight coupling between components, as everything is in one application
- Less reusability
- Large code base; tough for developers and QA to understand the code and business knowledge
- Less Scalable
- Does not follow SRP (Single Responsibility Principle)
- More deployment and restart times

1.2.4 Advantages of the proposed system

The proposed system has the following advantages:

- Independent components: Firstly, all the services can be deployed and updated independently, which gives more flexibility.
- Easier understanding: Split up into smaller and simpler components, a microservice application is easier to understand and manage.
- Better scalability: Another advantage of the microservices approach is that each element can be scaled independently.
- Flexibility in choosing the technology: The engineering teams are not limited by the technology chosen from the start.

1.2.5 Project Scope

The UC eBanking prime solution by UniCredit Bank AG is multi-bank compatible and is designed to accommodate the most recent standards in domestic and international payments. As opposed to many other applications in the field of electronic banking, our technology is not portal solution based. Your banking data is stored in the local systems of your own company both today and in the future as well, a feature enabled by the fact that the UC eBanking prime server is installed within your Intranet as a web application. In practice what this means is that, once the application has been installed, all authorized workstations from the different parts of the company and from different locations can access the system. This is one of the aspects that makes UC eBanking prime so innovative and unique.

The current electronic banking solution is a monolithic application with features like Banking, Administration, Audit, Dashboard, System and European Gate. The scope of our project is to Split these features starting from Payments under Banking. SEPA Credit, SEPA Debit, Foreign Payment, CGI Payments, etc. will be split as different microservices from the existing monolith ebanking Prime. We have to split the modules keeping the current security and efficiency in mind.

By spitting this monolith, we can provide only the required modules to different clients which ever is of use for them. This will save a lot of time and effort and would make the

application work faster. It will be easier to understand the huge application with more than 8 lakh lines of code. Splitting it down would even help us to migrate to a new technology as everything is independent.

2. SYSTEM ANALYSIS

This system analysis helps to design the software tool more effectively. The application is created for the use of. bioinformaticians who have less knowledge of the Linux environment. here we can identify the different requirements of the application. The requirements for the application are simple but integrating the different tools to an application needs proper planning and analysis. The feasibility of doing the same needs to be analyzed and identified for effective development. The system analysis focuses on the requirement specification, which includes the functional requirements, and the non-functional requirements. The system analysis also contains the block diagram and the system requirements. The system requirements explain the hardware and software requirements as well as the external tools used in developing the application.

2.1 Requirement Specifications

Requirement's specification is a collection of the set of all requirements that are to be imposed on the design and verification of the product. The specification also contains other related information necessary for the design, verification, and maintenance of the product. It is a description of a software system to be developed. It is modelled after business requirements specification, also known as a stakeholder requirements specification. The requirements specification of the application for next-generation sequencing analysis workflow includes the functional requirements and the non-functional requirements. The functional requirements specify the functional requirements and the non-functional requirements specify the additional feature sets that are expected to be included in the application.

2.1.1 Functional requirements

Functional requirements are product features or functions that developers must implement to enable users to accomplish their tasks. So, it's important to make them clear both for the development team and the stakeholders. Generally, functional requirements describe system behavior under specific conditions. The main functional requirements of the

application are to interface the tools, pipelines and applications. The current monolithic application has all the required features and the clients and the users using the application are satisfied. But it's been more than 14 years since the application was first built and now it has more than 6500 applications under it. So looking at the mass scale of the application breaking it down to microservices seems to be a better option.

All the current features of the application would work as separate microservices like Account Statement, Value Balance, Cash Forecast under Cash management. But firstly, we will be working on the Payment module and splitting SEPA Credit, SEPA Debit, Foreign Payment as separate microservices.

The main functional requirement of the application can be summarized in the following:

- The application must be easy to use and interactive.
- The users should get appropriate messages and notifications of the orders and transactions.
- The application should produce proper and well-formatted output in the form of reports, with an analysis summary with the expertise of the company.
- The application should be easy to install and configured remotely.
- The application should contain all the necessary tools to fulfil the user requirements.

2.1.2 Non-Functional requirements

Non-functional requirements define system attributes such as security, reliability, performance, maintainability, scalability, and usability. They serve as constraints or restrictions on the design of the system across the different backlogs. ... They ensure the usability and effectiveness of the entire system. The non-functional requirements of the application include the requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. They are contrasted with functional requirements that define specific behavior or functions. Here the non-functional requirements include the proper functioning of the application. That includes the application to express a modern user interface that helps the users with their data analysis.

The non-functional requirements further include the error-free execution of the application without crashing the system. The application should not misbehave under any circumstances in any worst case. The application should also be easy for the user to use, without a much-complicated user interface. The application should also help the users to find what they are looking for quickly and easily.

The main non-functional requirements of the application can be summarized in the following:

- As this is a banking application security plays a very vital role. The application should have the same level of security and must agree to the EBICS rules set by the German Banking Industry.
- The Microservices should be easy to use and should provide help manual and description wherever required.
- The application should be easy to install and shipped to the client.
- The system should be able to meet the expectations of all kinds of users with a modern and efficient user interface ad performance.

2.2 Block Diagram

A block diagram is a diagram of a system in which the principal parts or functions are represented by blocks connected by lines that show the relationships of the blocks. They are heavily used in engineering in hardware design, electronic design, software design, and process flow diagrams. The block diagram of the application helps the developers to understand the structures of the application. The block diagram for the application is as follows:

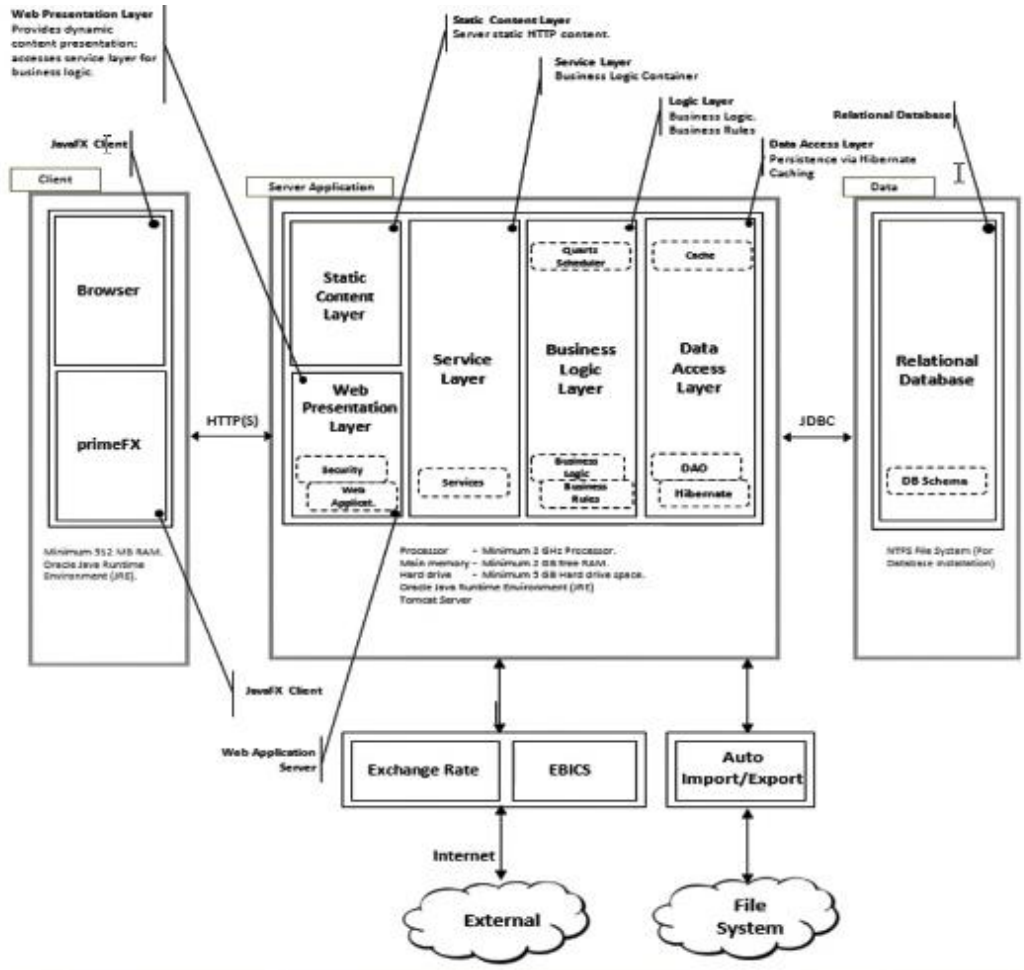


Fig. 2.1 UC eBanking Prime diagram

Figure 2.1 shows block diagram of the application in a very abstract way. The block diagram depicts and shows the simple layout of the application. The leftmost part shows the client system and how they can access the application with the help of a browser. But the application primeFX must be first installed in the system which can even be done remotely.

Then comes the main application which consists of different layers and is connected to external systems like exchange rates and the German stated EBICS protocol. All the layers are connected and perform a specific role important for the application to work. Now comes the backend connected using JDBC which is a relational database. All the necessary data about the application and client's data would be stored here.

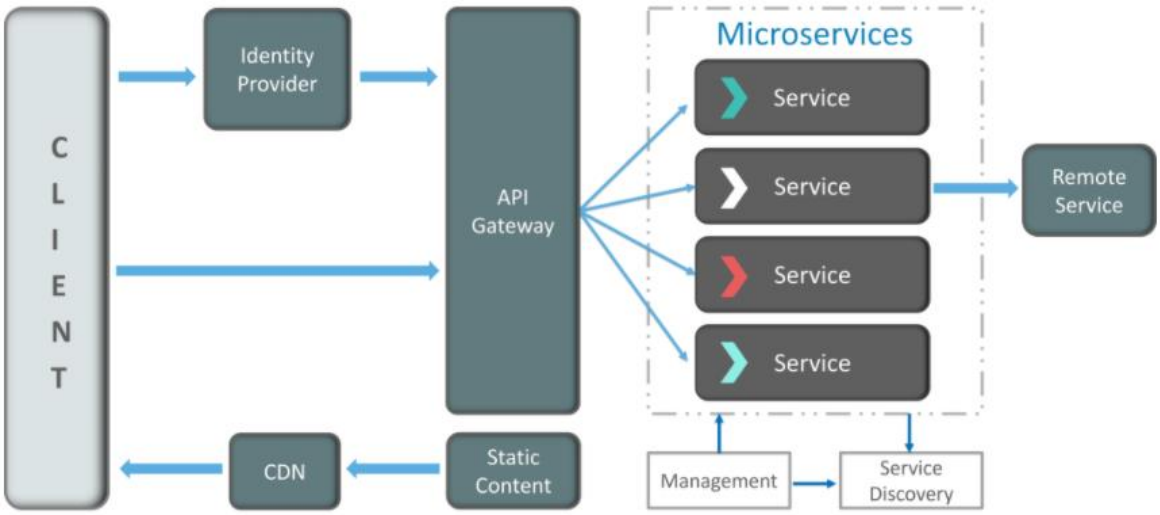


Fig. 2.2 Microservices Diagram for the Payment Module

The above figure shows the idea of how Payment module will work after being split into microservices. All the services would act as one and would be working together. All the working would be similar like the monolith and the performance is expected to improve over time.

2.3 System Requirements

The basic system requirements for the application to work and developed and tested are as follows, it is divided into two, hardware and software requirements. These are the minimum requirements for the application to work smoothly, however, the application with the tools requires a minimum of these requirements to work smoothly.

2.3.1 Hardware requirements

The expected hardware requirements for the effective performance of the application have the following considerations and specifications.

- Hardware: Dell Latitude 3410
- Memory: 16GB

- Storage: 512 GB NVMe SSD
- Processor: Intel® Core™ i5-10310U CPU @ 1.70GHz
- Graphics: Intel UHD 8GB Graphics

The Latitude 3410 features Dell Optimizer, built-in AI that learns and responds to the way you work. It's controlled through one module, at no additional cost, to reduce lag and help you run at peak performance. Based on our habits, dynamic settings keep running longer or give you a quicker ExpressCharge. And Intelligent Audio auto-adjusts for best sound quality and conferencing. It is a system built for business with a whole new experience to work, intelligently built in and modernization built on.

The DDR4 16GB RAM is High-performing memory empowers faster, powerful solutions. Samsung DDR4 delivers top speed with better bandwidth and reliability using less energy. Easily processing massive workloads with enhanced speed, the DDR4 transfers more data faster than ever before, offering 4 bank groups (total 16 banks) to reduce interleaving delays, plus 3,200 Mbps bandwidth and 1 TB/s system memory. Samsung's industry-first 1x nm process technology enables DDR4 to consume less power while boosting performance, reducing TCO. The 1.2V low operating voltage and Pseudo Open Drain (POD) interface enables lower power consumption, using 25% less energy. System reliability is ever more critical as data centers process ever more traffic. Advanced features of the Samsung DDR4 ensure superior data transmission, including Write CRC to help recognize multibit failures and parity checks for CMD/ADD to prevent system malfunctions.

Samsung 512 GB NVMe SSD is for intensive workloads on PCs and workstations, the 970 PRO delivers ultimate performance powered by Samsung's NVMe® SSD leadership. The latest V-NAND technology and new in-house controller in a compact M.2 (2280) form factor meets the needs of the most demanding tech enthusiasts and professionals. Performance that puts you in command. The 970 PRO combines the next-gen PCIe® Gen 3x4 NVMe® interface with the latest V-NAND technology and a newly enhanced in-house controller to achieve fearless read/write speeds up to 3,500/2,700 MB/s*, approximately

30 percent faster than the previous generation. Achieve a new level of drive confidence. Samsung's advanced nickel-coated controller and heat spreader on the 970 PRO enable superior heat dissipation. The Dynamic Thermal Guard automatically monitors and maintains optimal operating temperatures to minimize performance drops.

The Intel Core i5-10310U is a power efficient quad-core SoC for notebooks based on the Comet Lake (CML-U) generation and was announced in August 2019. Compared to the similar Whiskey Lake processors (e.g. Core i5-8665U), the only difference is support for higher memory speeds (DDR4-2666 vs 2400) and two additional cores in the top model (not in this i5). The processor cores are clocked between 1.7 and 4.4 GHz (unverified). Thanks to HyperThreading 8 threads can be used. Compared to the faster i5-10510U, the level 3 cache was reduced from 8 to 6 MB. More information on Comet Lake and all the models and articles on it can be found here. The CPU supports the professional management features under the vPro umbrella.

This Graphics Processing Unit is Integrated with the Intel's 8th Generation Notebooks Processors. It is based on the Kaby Lake Refresh Architecture. Compared to the previous generations, this has some new features such as 10-bit colors, H.265/HEVC Main 10 Profile, and HDCP 2.2. The Thermal Design Power (TDP) is equal to 15W which is same as the Processor. It is based on the 14nm Tri-Gate (FinFET) fabrication technique.

2.3.2 Software requirements

The expected software requirements for the effective performance of the application have the following considerations and specifications.

- **OS:** Windows 10 (64 bit).
- **Frontend Tools:** JSF, xhtml
- **Backend Tools:** Hibernate, Spring, Apache Tomcat
- **Database Tools:** PostgreSQL

2.3.2.1 Windows 10

Windows 10 makes its user experience and functionality more consistent between different classes of device and addresses most of the shortcomings in the user interface that were introduced in Windows 8. Windows 10 Mobile, the successor to Windows Phone 8.1, shared some user interface elements and apps with its PC counterpart.

Windows 10 supports universal apps, an expansion of the Metro-style first introduced in Windows 8. Universal apps can be designed to run across multiple Microsoft product families with nearly identical code—including PCs, tablets, smartphones, embedded systems, Xbox One, Surface Hub and Mixed Reality. The Windows user interface was revised to handle transitions between a mouse-oriented interface and a touchscreen-optimized interface based on available input devices—particularly on 2-in-1 PCs, both interfaces include an updated Start menu which incorporates elements of Windows 7's traditional Start menu with the tiles of Windows 8. Windows 10 also introduced the Microsoft Edge web browser, a virtual desktop system, a window and desktop management feature called Task View, support for fingerprint and face recognition login, new security features for enterprise environments, and DirectX 12.

2.3.2.2 JSP

Jakarta Server Faces (JSF; formerly JavaServer Faces) is a Java specification for building component-based user interfaces for web applications and was formalized as a standard through the Java Community Process being part of the Java Platform, Enterprise Edition. It is also an MVC web framework that simplifies construction of user interfaces (UI) for server-based applications by using reusable UI components in a page.

JSF 2.x uses Facelets as its default templating system. Users of the software may also choose to employ technologies such as XUL, or Java. JSF 1.x uses JavaServer Pages (JSP) as its default templating system.

Based on a component-driven UI design-model, JavaServer Faces uses XML files called view templates or Facelets views. The FacesServlet processes requests, loads the

appropriate view template, builds a component tree, processes events, and renders the response (typically in the HTML language) to the client. The state of UI components and other objects of scope interest is saved at the end of each request in a process called stateSaving (note: transient true) and restored upon next creation of that view. Either the client or the server side can save objects and states.

2.3.2.3 XHTML

Extensible HyperText Markup Language (XHTML) is part of the family of XML markup languages. It mirrors or extends versions of the widely used HyperText Markup Language (HTML), the language in which Web pages are formulated.

While HTML, prior to HTML5, was defined as an application of Standard Generalized Markup Language (SGML), a flexible markup language framework, XHTML is an application of XML, a more restrictive subset of SGML. XHTML documents are well-formed and may therefore be parsed using standard XML parsers, unlike HTML, which requires a lenient HTML-specific parser.

XHTML 1.0 became a World Wide Web Consortium (W3C) recommendation on 26 January 2000. XHTML 1.1 became a W3C recommendation on 31 May 2001. The standard known as XHTML5 is being developed as an XML adaptation of the HTML5 specification.

2.3.2.4 Hibernate

Hibernate ORM (or simply Hibernate) is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object–relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

Hibernate is free software that is distributed under the GNU Lesser General Public License 2.1.

Hibernate's primary feature is mapping from Java classes to database tables, and mapping from Java data types to SQL data types. Hibernate also provides data query and retrieval

facilities. It generates SQL calls and relieves the developer from the manual handling and object conversion of the result set.

Hibernate provides transparent persistence for Plain Old Java Objects (POJOs). The only strict requirement for a persistent class is a no-argument constructor, though not necessarily public. Proper behavior in some applications also requires special attention to the equals() and hashCode() methods in the object classes. Hibernate recommends providing an identifier attribute, and this is planned to be a mandatory requirement in a future release.

2.3.2.5 Spring

The Spring Framework is an application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE (Enterprise Edition) platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an addition to the Enterprise JavaBeans (EJB) model. The Spring Framework is open source.

When a user clicks a link or submits a form in their web-browser, the request goes to Spring DispatcherServlet. DispatcherServlet is a front-controller in spring MVC. It consults one or more handler mappings. DispatcherServlet has been chosen as an appropriate controller and forwards the request to it. The Controller processes the particular request and generates a result. It is known as Model. This information needs to be formatted in html or any front-end technology like JSP. This is the View of an application. All of the information is in the MODEL And VIEW object. When the controller is not coupled to a particular view, DispatcherServlet finds the actual JSP with the help of ViewResolver.

2.3.2.6 Apache Tomcat

Apache Tomcat (called "Tomcat" for short) is a free and open-source implementation of the Jakarta Servlet, Jakarta Expression Language, and WebSocket technologies. Tomcat provides a "pure Java" HTTP web server environment in which Java code can run.

Tomcat is developed and maintained by an open community of developers under the auspices of the Apache Software Foundation, released under the Apache License 2.0 license.
Tomcat 8.x implements the Servlet 3.1 and JSP 2.3 Specifications. Apache Tomcat 8.5.x is intended to replace 8.0.x and includes new features pulled forward from Tomcat 9.0.x. The minimum Java version and implemented specification versions remain unchanged.

Tomcat 9.x implements the Servlet 4.0 and JSP 2.3 Specifications.
Tomcat 10.x implements the Servlet 5.0 and JSP 3.0 Specifications.

2.3.2.7 PostgreSQL

PostgreSQL also known as Postgres, is a free and open-source relational database management system (RDBMS) emphasizing extensibility and SQL compliance. It was originally named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley. In 1996, the project was renamed to PostgreSQL to reflect its support for SQL. After a review in 2007, the development team decided to keep the name PostgreSQL and the alias Postgres.
PostgreSQL features transactions with Atomicity, Consistency, Isolation, Durability (ACID) properties, automatically updatable views, materialized views, triggers, foreign keys, and stored procedures. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users. It is the default database for macOS Server and is also available for Windows, Linux, FreeBSD, and OpenBSD.

2.3.3 External Tools

There are various external tools added to the application. That includes development and testing tools set by the industry. Some of the known and initial tools include the following

2.3.3.1 Eclipse

Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. It is the second-most-popular IDE for Java development, and, until 2016, was the most popular. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins, including Ada, ABAP, C, C++, C#, Clojure, COBOL, D, Erlang, Fortran, Groovy, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Rust, Scala, and Scheme. It can also be used to develop documents with LaTeX (via a TeXlipse plug-in) and packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala.

The Eclipse Web Tools Platform (WTP) project is an extension of the Eclipse platform with tools for developing Web and Java EE applications. It includes source and graphical editors for a variety of languages, wizards and built-in applications to simplify development, and tools and APIs to support deploying, running, and testing apps.

Eclipse uses plug-ins to provide all the functionality within and on top of the run-time system. Its run-time system is based on Equinox, an implementation of the OSGi core framework specification.

In addition to allowing the Eclipse Platform to be extended using other programming languages, such as C and Python, the plug-in framework allows the Eclipse Platform to work with typesetting languages like LaTeX and networking applications such as telnet and database management systems. The plug-in architecture supports writing any desired extension to the environment, such as for configuration management. Java and CVS support is provided in the Eclipse SDK, with support for other version control systems provided by third-party plug-ins.

2.3.3.2 Windows Server

Windows Server is a brand name for a group of server operating systems (OS) that Microsoft has been developing since 2003. The first OS to be released under this brand name was Windows Server 2003.

Microsoft's history of developing operating systems for server computers goes back to Windows NT 3.1 Advanced Server edition. Windows 2000 Server edition was the first OS to include Active Directory, DNS Server, DHCP Server, and Group Policy.

Windows Server is the platform for building an infrastructure of connected applications, networks and web services. As a Windows Server administrator, you've helped achieve your business' goals keeping the infrastructure secure, available and flexible. Windows Server has been the foundation of Microsoft's ecosystem and continues to power the hybrid cloud network today.

Windows Server File Servers host billions of files across millions of customers for storage and retrieval of files with built-in scale. Security, quotas, backup, replication and recovery are all built into the operating system.

3. SYSTEM DESIGN

The system designs the developers to design and plan the development of the application. As our monolithic application is already developed the architecture really helps us to understand how to break it into different services. The tools and technologies that were used to build the previous application. Moreover, these design diagrams are huge help for a person with less experience and ideas on the code and working face.

The application is aimed to help the bank keep records of the transactions and to facilitate the same. Our application helps the bank to sign all the transactions and see to that all the different types of payments go through.

The system design depicts the system architecture, various modules, the database design and the system configuration. The vital information on this facilitates the splitting of the application into services. We understand the current application through these diagrams and how and where exactly to split we get the answers to these questions.

3.1 System Architecture

The system architecture depicts the tool architecture how the main application communicates with the different tools. The main purpose of Prime application is to check for the transaction, get it signed and approved by the EBICS Protocol. Now our main motive is to understand the application and split the various modules into different services. The Payment module being one of the many big modules of the application has many complex architectures which has been discussed later in this report.

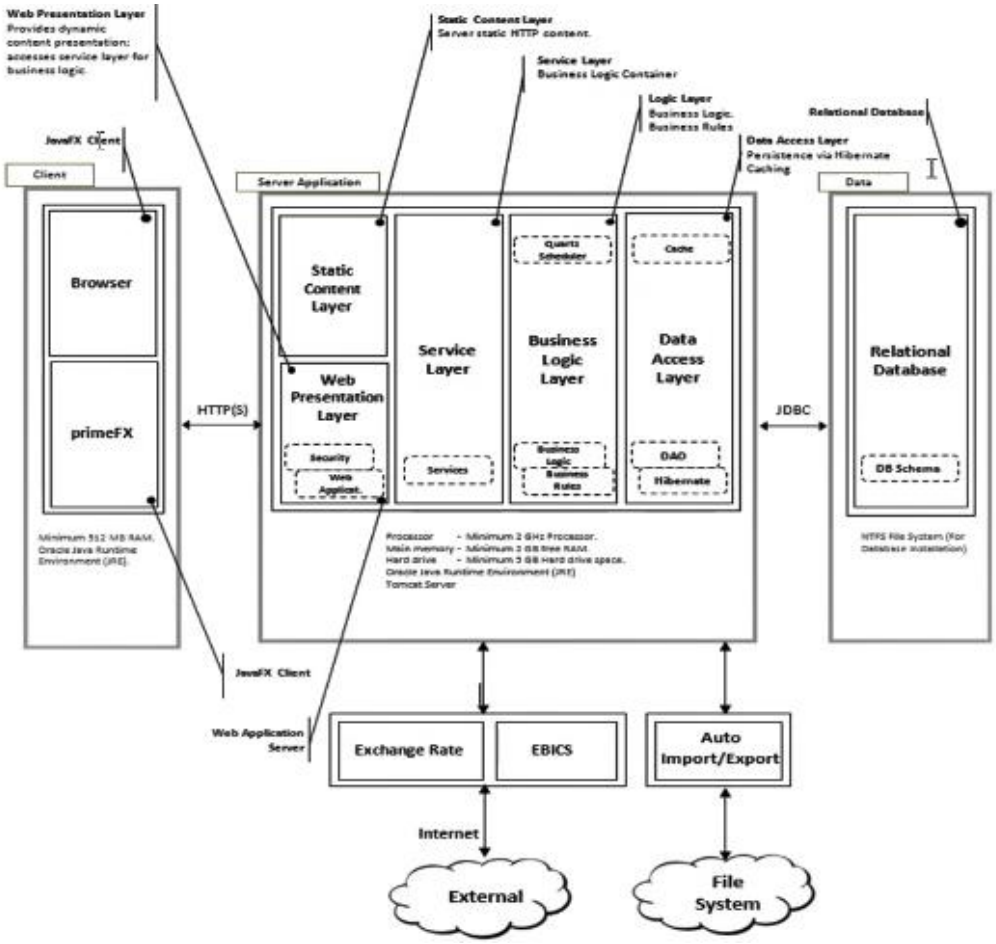


Fig. 3.1 Architecture diagram of UC eBanking Prime

The Client can use the application UC eBanking Prime in his/her browser which will communicate further with http(s) protocol. Inside the application there goes various layers like static content layer, web presentation layer, service layer, business logic layer and data access layer.

Furthermore the application will check for exchange rates and get the transaction signed by EBICS which will be done on an external platform.

Whatever the export and imports are to be done will be managed via the files on ones system.

And all these data will be stored on a relational database which will be inside ones organization only to keep the data secure. This all the clients data will be stored on clients side only.

3.2 Module Design

There are different modules for the application. The modules keep the application simple and easy to use. Each component of the application is expected to be independent, hence we can achieve the concept of microservices. It will help us better understand the application and easy to debug.

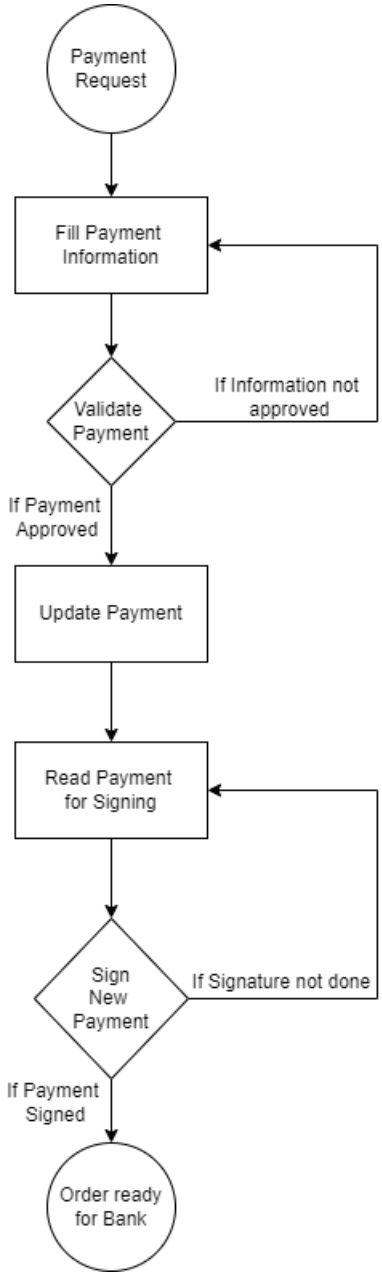


Fig. 3.2 Flowchart of Payment Module

This report focuses on the Payment module. The Payment module has SEPA Credit, SEPA Debit Foreign Payments and many other things in it. The above-mentioned diagram depicts the working of the Payment and shows the flow of how the payment is approved and signed by the system. The application receives a payment request and the payment information need to be filled then the system will validate the request, if it is not approved then it'll again ask for the proper information or else it'll validate and move to the next step.
Later it will update the record in the payment database and send it for signing to the EBICS external system. If it is not signed it will ask the system to recheck it again or else it'll sign and the order will be created to the receiving bank.

3.3 Use-case Design

The use-case diagram helps to understand an entities role in the application. In my application keeping the Payment module in mind the Bank User plays a vital role in all his/her actions.

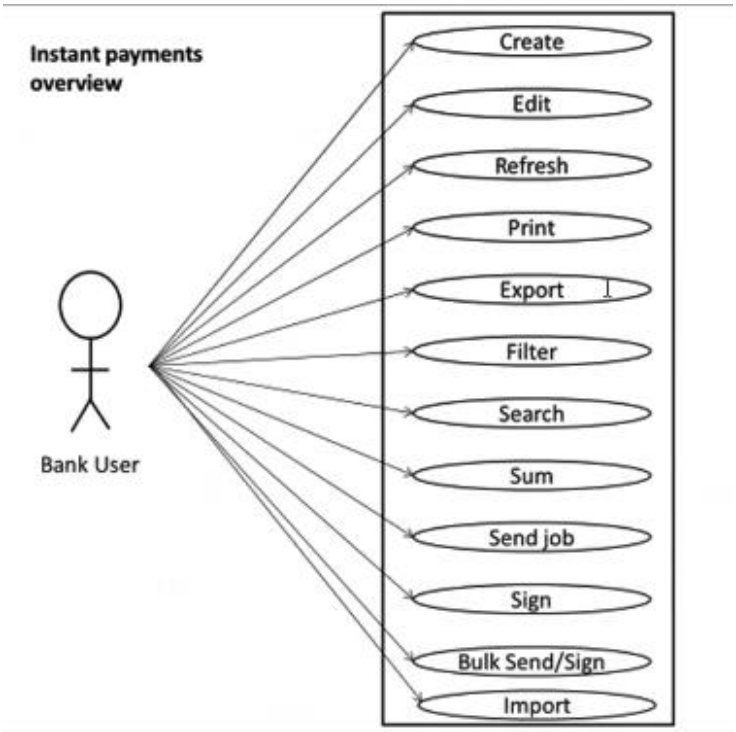


Fig. 3.3 Use-case diagram of the bank user

There are various functions that a Bank user does. These functions are mostly related to payments. He/she can create a payment or edit it being the member of the bank he has rights to do so. For report generation he can print, refresh and even filter or search the same. He can send the payment to get signed, send a job and even import.

3.4 Migration Patterns

A 2018 survey found that 63 percent of enterprises were adopting microservice architectures. This widespread adoption is driven by the promise of improvements in resilience, scalability, time to market, and maintenance, among other reasons. In this blog post, I describe a plan for how organizations that wish to migrate existing applications to microservices can do so safely and effectively.

In a typical monolithic application, all of the data objects and actions are handled by a single, tightly knit codebase. Data is typically stored in single database or filesystem. Functions and methods are developed to access the data directly from this storage mechanism, and all business logic is contained within the server codebase and the client application. It is possible to migrate several monolithic applications and/or platforms, each with its own data storage mechanisms, user interfaces, and data schema, into a unified set of microservices that perform the same functions as the original applications under a single user interface.

3.4.1 Strangler Fig Application

Implementing the Strangler Fig Pattern

a. Asset Capture: Identify the functionality to move to a new microservice.
b. Redirect Calls: Intercept calls to old functionality, and redirect to the new service.

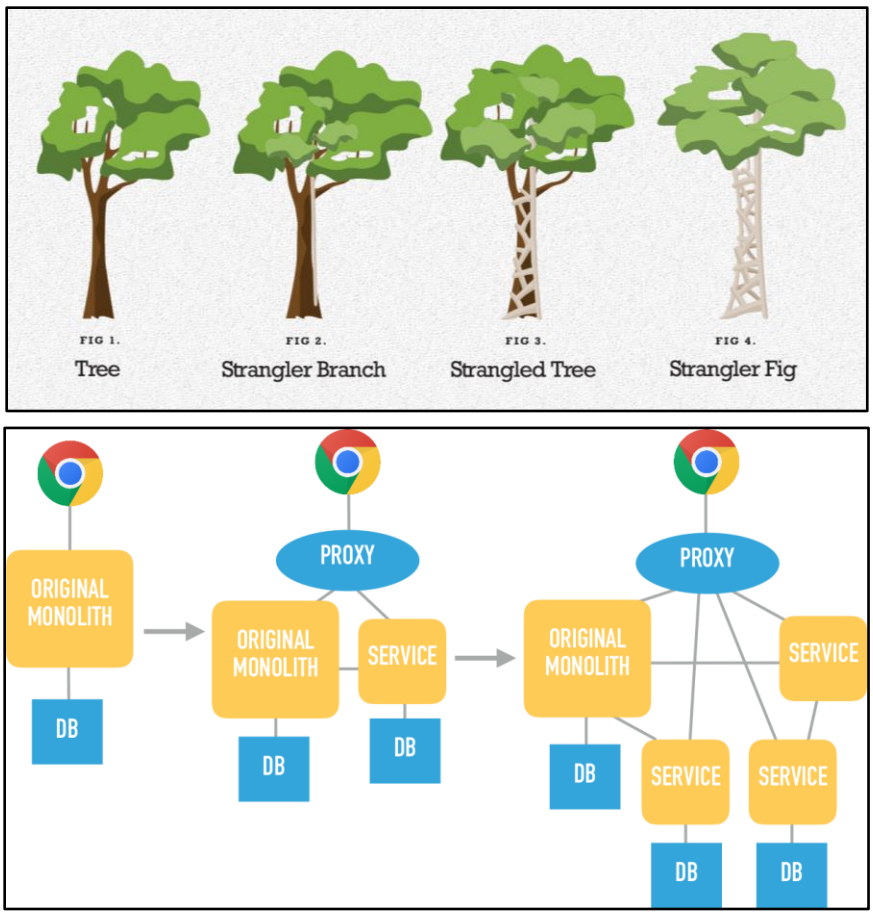


Fig. 3.4 Strangler Fig Example

Moving functionality? It might be copy and paste. More likely is a total or partial rewrite.

Advantage: You can stop rewriting at any stage of the migration. (It is not necessary to completely kill the monolith.)

Examples:

- **HTTP Reverse Proxy**
- **FTP**
- **Message Interception**

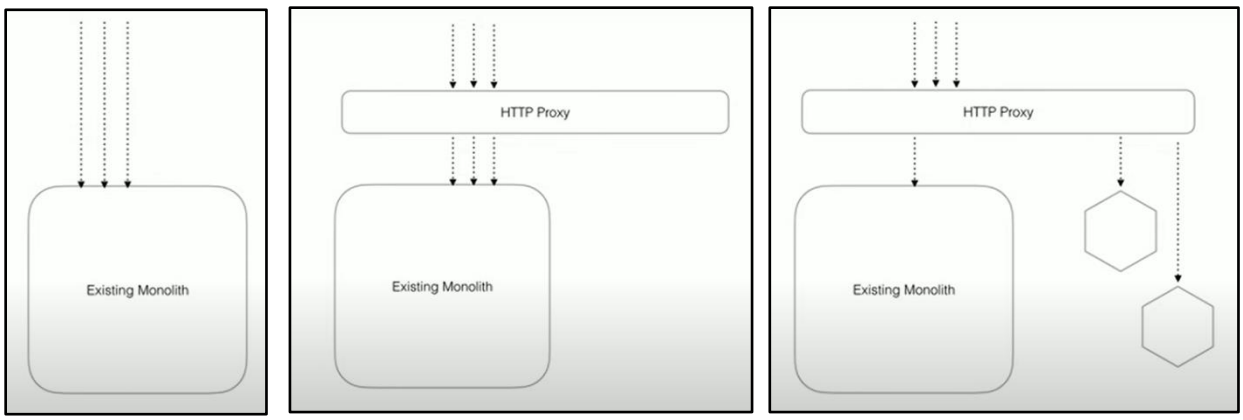


Fig. 3.5 Strangler Fig Implementation

3.4.2 UI Composition

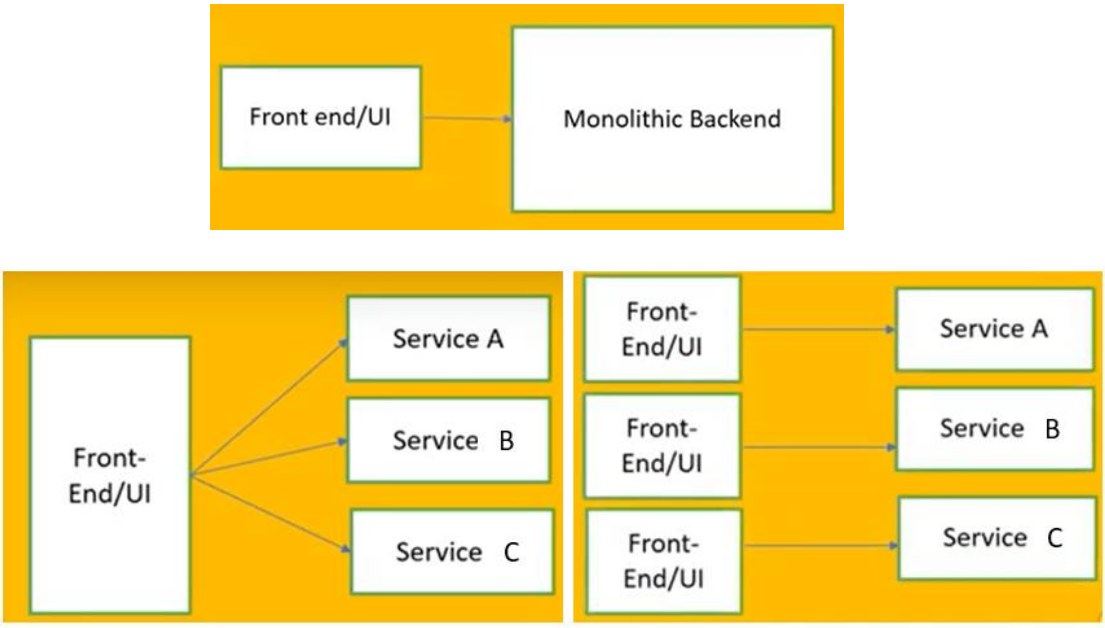


Fig. 3.6 UI Composition Diagram

UI screens/pages display data from multiple service. Consider, for example, an Amazon-style product detail page, which displays numerous data items including:

- Basic information about the book such as title, author, price, etc.
- Your purchase history for the book
- Availability
- Buying options
- Other items that are frequently bought with this book
- Other items bought by customers who bought this book

- Customer reviews
- Seller's ranking, etc.

Each data item corresponds to a separate service and how it is displayed is, therefore, the responsibility of a different team.

How to implement a UI screen or page that displays data from multiple services?

Each team develops a client-side UI component, such an AngularJS directive, that implements the region of the page/screen for their service. A UI team is responsible for implementing the page skeletons that build pages/screens by composing multiple, service-specific UI components.

Examples:

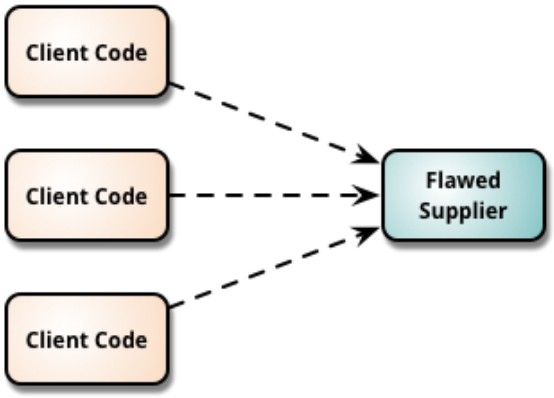
- Page Composition
- Widget Composition
- Micro Frontends

3.4.3 Branch by Abstraction

Often, when reworking parts of an existing codebase, people will do that work on a separate source code branch. This allows the changes to be made without disrupting the work of other developers. The challenge is that once the change in the branch has been completed, these changes must be merged back, which can often cause significant challenges. The longer the branch exists, the bigger these problems are.

"Branch by Abstraction" is a technique for making a large-scale change to a software system in gradual way that allows you to release the system regularly while the change is still in-progress.

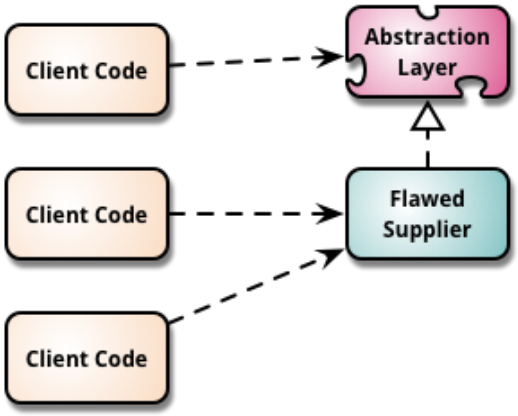
We begin with a situation where various parts of the software system are dependent on a module, library, or framework that we wish to replace



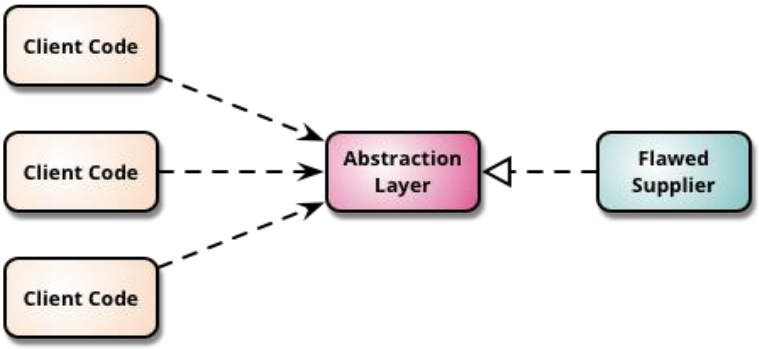
Branch by abstraction consists of five steps:

1. Create an abstraction for the functionality to be replaced.
2. Change clients of the existing functionality to use the new abstraction.
3. Create a new implementation of the abstraction with the reworked functionality.
3. This new implementation will call out to our new microservice.
4. Switch over the abstraction to use our new implementation.
5. Clean up the abstraction and remove the old implementation.

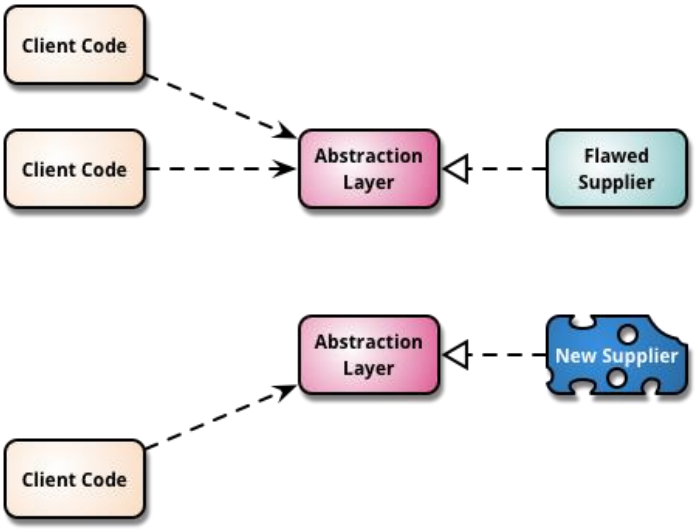
We create an abstraction layer that captures the interaction between one section of the client code and the current supplier. We change that section of the client code to call the supplier entirely through this abstraction layer.



We gradually move all client code over to use the abstraction layer until all interaction with the supplier is done by the abstraction layer. As we do this we take the opportunity to improve the unit test coverage of the supplier through this abstraction layer.



We build a new supplier that implements the features required by one part of the client code using the same abstraction layer. Once we are ready, we switch that section of the client code to use the new supplier.



We gradually swap out the flawed supplier until all the client code uses the new supplier. Once the flawed supplier isn't needed, we can delete it. We may also choose to delete the abstraction layer once we no longer need it for migration.

3.4.4 Parallel Run Pattern

When using a parallel run, rather than calling either the old or the new implementation, instead we call both, allowing us to compare the results to ensure they are equivalent.

Despite calling both implementations, only one is considered the source of truth at any given time. Typically, the old implementation is considered the source of truth until the ongoing verification reveals that we can trust our new implementation.
-- Sam Newman, Monolith to microservices

There are several ways of implementing this pattern.
Monitoring and Reporting
In order to consider an endpoint ready, it had to reach a satisfying consistency percentage.

- Matched: counter for all the requests that matched between the monolith and the Returns microservice.
- Unmatched: counter for all the requests that did not match between the two services.
- Failed: counter for all the requests where the response was terminated by temporary issues.

Rollout

The switch was done gradually, and it was done per endpoint to allow the system to be tested in a fully functional way. This was achieved by using a proxy to move the forwarding of the requests to the Returns microservice one by one once they were ready.

Advantages of this approach

1. Live data for testing
2. Gradual rollout
3. Incremental development
4. Easy rollback
5. Finding bugs
6. Load testing

The parallel run pattern is a powerful technique to overcome the complexities and stress of migration projects, but not every migration project is a match to use this pattern. Increasing traffic, complexities in comparing the results, and the amount of effort are the risks that should be considered before implementing this pattern.

3.5 Decomposing the Database Patterns

3.5.1 The Shared Database

Directly accessing the data is NOT a good idea to be used for more than a few weeks, as this is one of the easiest ways to get caught in a distributed system. ‘Creates a huge amount of (implementation) coupling.

Avoid sharing databases! – It becomes difficult to distinguish between what’s shared and what’s hidden.

What can be done is that other services instead of accessing the database directly, they request data from the respective service through well-defined service interfaces.

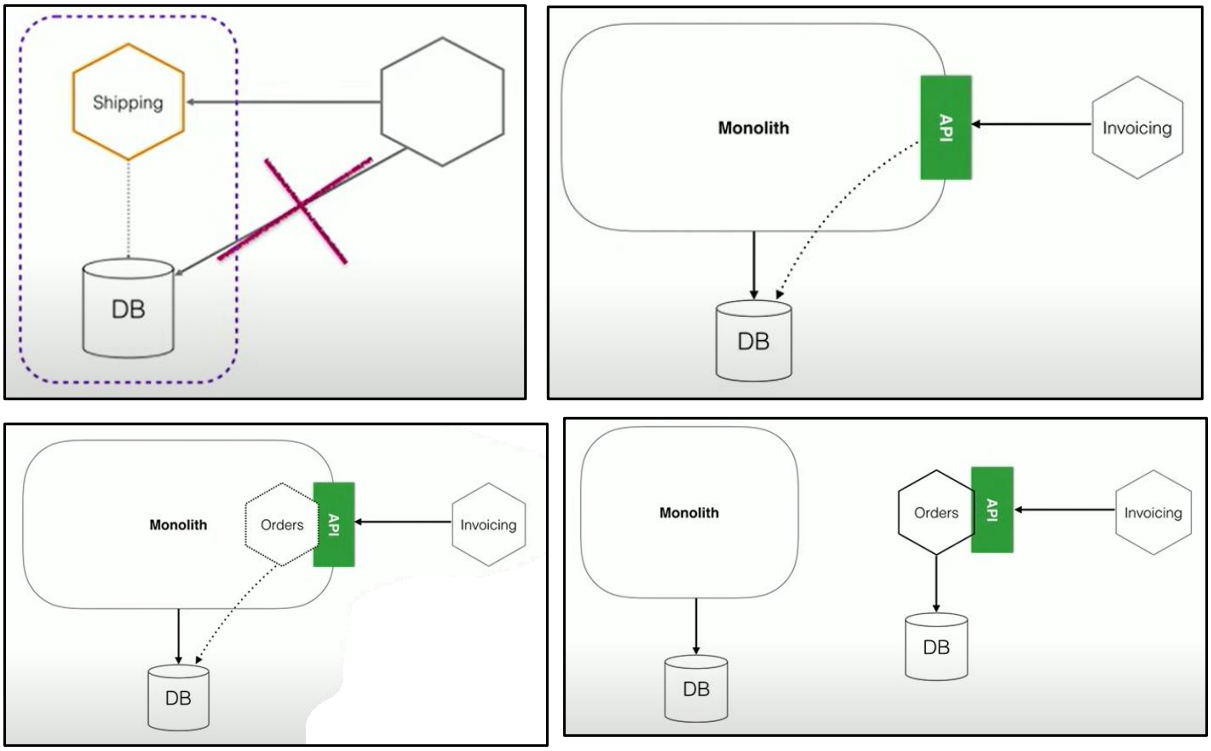
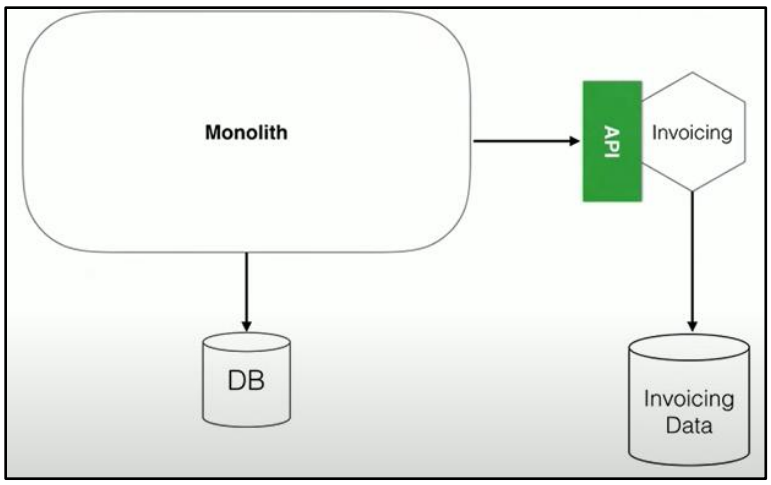


Fig. 3.7 The shared database diagram



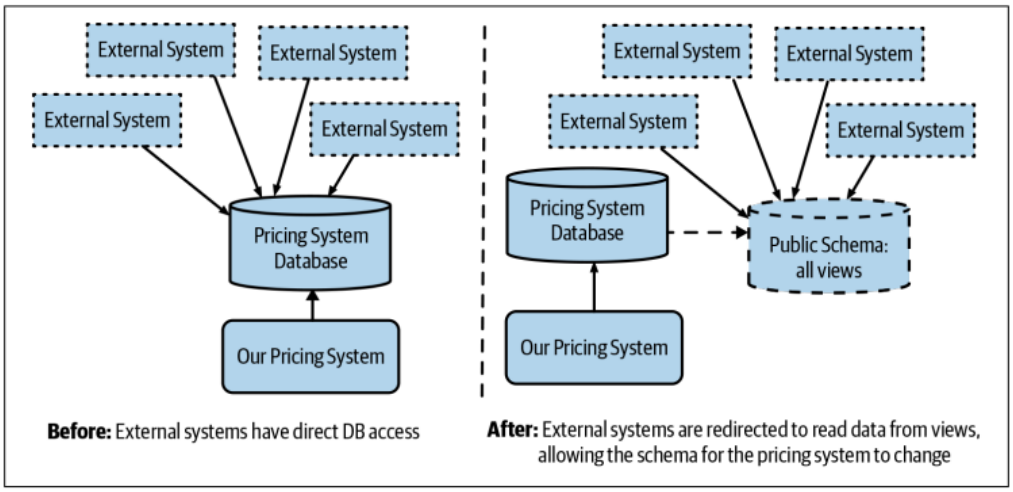
Where to Use It:

Direct sharing of a database is appropriate for a microservice architecture in only two situations.

i. When considering read-only static reference data. Example: schema holding country currency code information, postal code, or zip code lookup tables, etc.
ii. When a service is directly exposing a database as a defined endpoint that is designed and managed to handle multiple consumers (Database-as-a-Service Interface).

3.5.2 Database View

In a situation where we want a single source of data for multiple services, a view can be used to mitigate the concerns regarding coupling. With a view, a service can be presented with a schema that is a limited projection from an underlying schema. This projection can limit the data that is visible to the service, hiding information it shouldn't have access to.



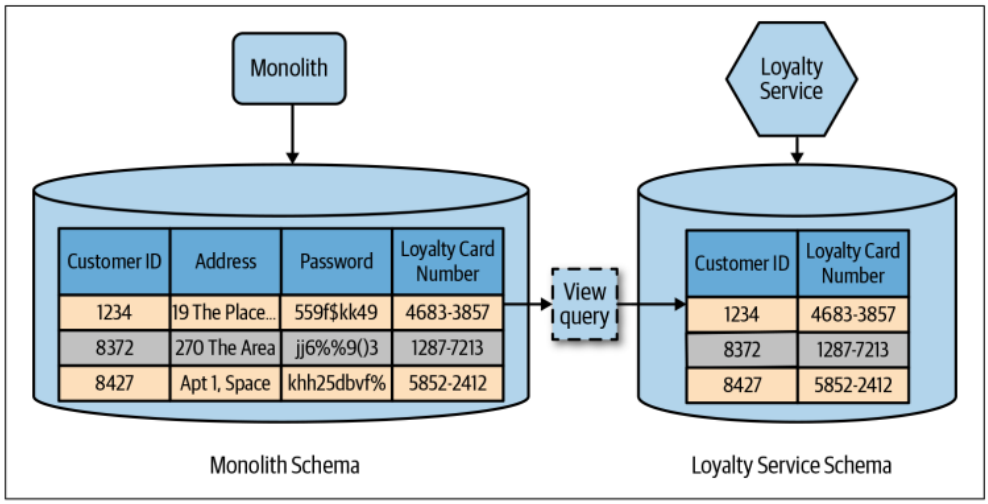


Fig 3.8 Database View diagram

Limitation:

Even if your database engine does support views, there will likely be other limitations, such as the need for both the source schema and view to be in the same database engine. This could increase your physical deployment coupling, leading to a potential single point of failure.

Where to Use It:

Use it where it is impractical to decompose the existing monolithic schema. Ideally, we should try to avoid the need for a view, if possible, if the end goal is to expose this information via a service interface. Instead, it's better to push forward with proper schema decomposition.

3.5.3 Database Wrapping Services

Here, we hide the database behind a service that acts as a thin wrapper, moving database dependencies to become service dependencies.

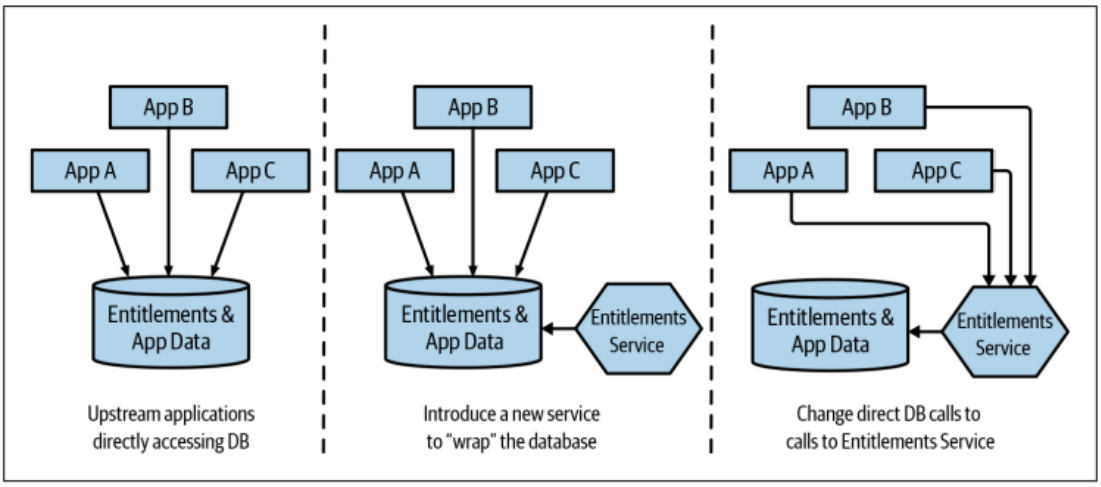
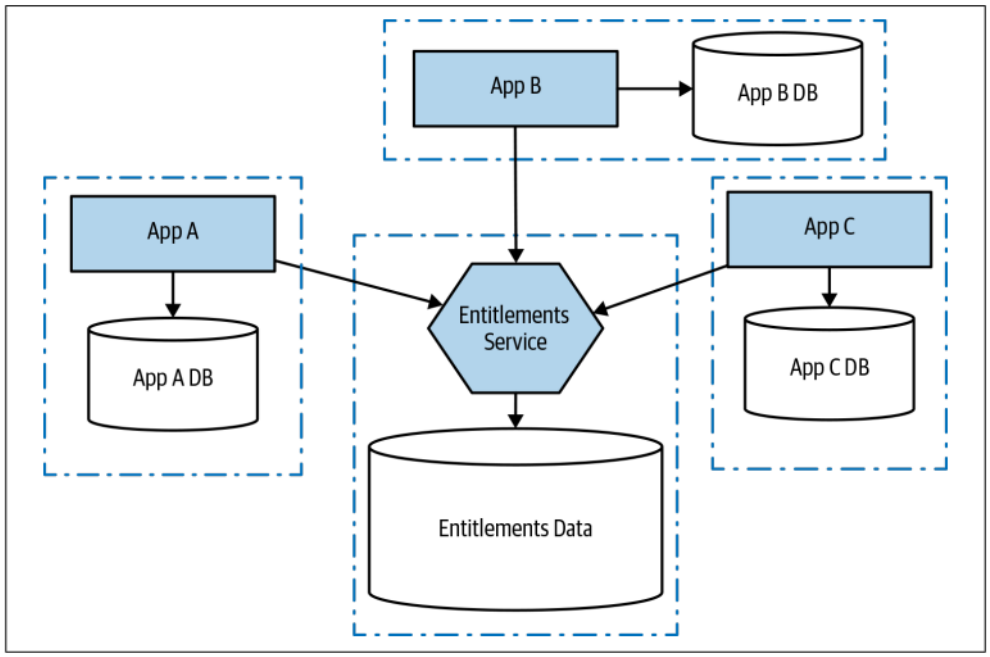


Fig 3.9 Database Wrapping diagram

The goal is to encourage the teams writing the other applications to think of the 'entitlements' schema as someone else's and encourage them to store their own data locally.



Just as with our use of database views, the use of a wrapping service allows us to control what is shared and what is hidden. It presents an interface to consumers that can be fixed, while changes are made under the hood to improve the situation.

Where to Use It:

By placing an explicit wrapper around the schema and making it clear that the data can be accessed only through that schema, you at the very least can put a brake on the database growing any further.

It works best when you align ownership of both the underlying schema and the service layer to the same team. You can write code in your wrapping service to present much more sophisticated projections on the underlying data. The wrapping service can also take writes (via API calls). Of course, adopting this pattern does require upstream consumers to make changes; they must shift from direct DB access to API calls.

3.5.4 The Database per Service Pattern

According to this pattern, we should keep all the persistent data of a microservice private to that service and accessible only via its API. It means that the database of the service is effectively part of the implementation of that service and not directly accessible by other services. There are a few different ways to keep a service's persistent data private:

- Private-tables-per-service: each service owns a set of tables that must only be accessed by that service.
- Schema-per-service: each service has a database schema that is private to that service
- Database-server-per-service: each service has its own database server. When the service must be scaled, the database can be also scaled in a database cluster, no matter the service.

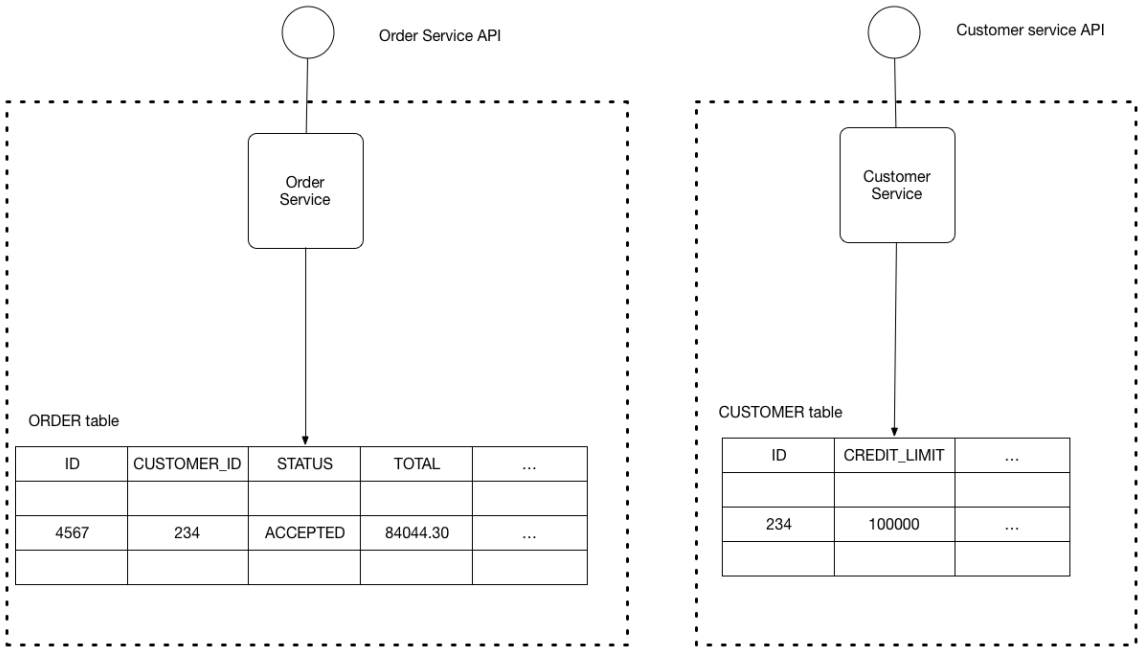


Fig. 3.10 Database service pattern diagram

3.5.5 Synchronize data in application

The decision was made that the application itself would perform the synchronization between the two data sources. The idea is that initially the existing MySQL database would remain the source of truth, but for a period of time the application would ensure that data in MySQL and Riak were kept in sync. After a period of time, Riak would move to being the source of truth for the application, prior to MySQL being retired.

- Step 1: Bulk Synchronize Data

The first step is to get to the point where you have a copy of the data in the new database.

- Step 2: Synchronize on Write, Read from Old Schema

With both databases now in sync, a new version of the application was deployed that would write all data to both databases. At this stage, the goal was to ensure that the application was correctly writing to both sources and make sure that Riak was behaving within acceptable tolerances.

- Step 3: Synchronize on Write, Read from New Schema

At this stage, it's been verified that reads to Riak are working fine. The last step is to make sure that reads work too. A simple change to the application now has Riak as being the source of truth

Once the new system has bedded in enough, the old schema could be safely removed.

3.5.6 Tracer Write

The tracer write pattern, outlined in the section Figure 4-18, is arguably a variation of the synchronize data in application pattern (see the section "Pattern: Synchronize Data in Application" on page 145. With a tracer write, we move the source of truth for data in an incremental fashion, tolerating there being two sources of truth during the migration.

The reason this pattern is called a tracer write is that you can start with a small set of data being synchronized and increase this over time, while also increasing the number of consumers of the new source of data.

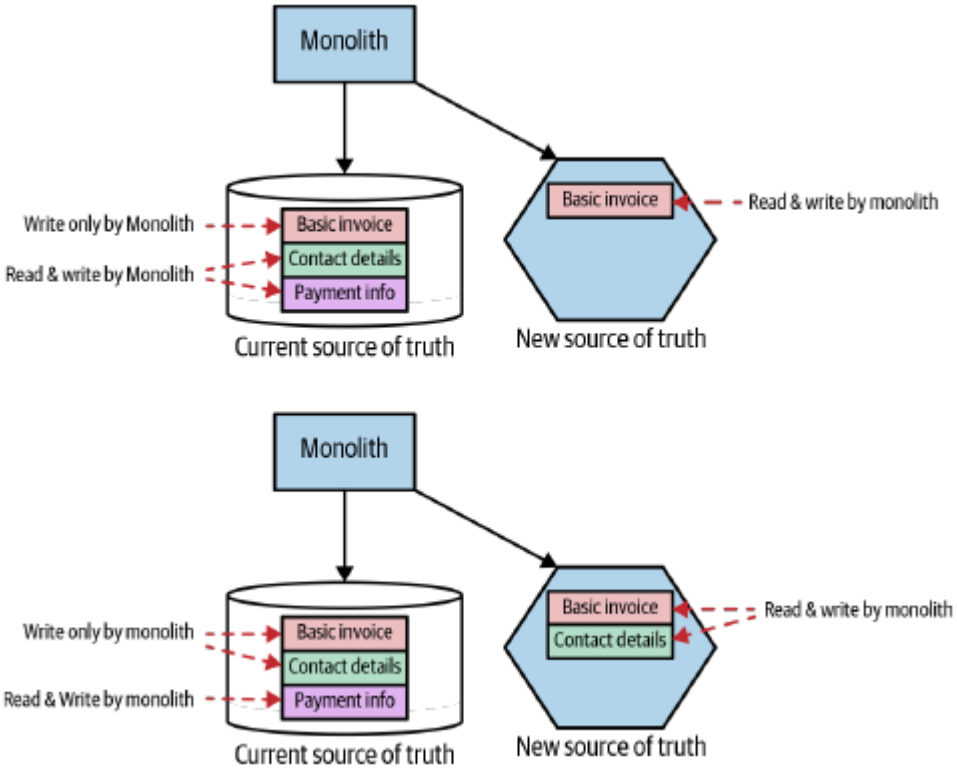


Fig 3.11 Trace Write diagram

4. IMPLEMENTATION

The application which currently is a monolithic application was built in 2008 using the Spring technology and Hibernate and Apache Tomcat being the frameworks. PostgreSQL was used for managing the backend. And for the frontend JSF and XHTML was used. This application was approved by the UC Bank which was then the 2nd largest bank of Germany and was put to use. This application was installed on their server side and the whole company could use it directly from the server.

4.1 Coding Standard

Coding conventions are style guidelines for programming. They typically cover Naming and declaration rules for variables and functions, Rules for the use of white space, indentation, and comments and Programming practices and principles. The Coding conventions secure quality: Improves code readability and Make code maintenance easier. Coding conventions can be documented rules for teams to follow, or just be your coding practice.

4.1.1 Variable Names

In the application, we use camelCase for identifier names (variables and functions). All names start with a letter. At the bottom of this page, you will find a wider discussion about naming rules.

4.1.2 Spaces Around Operators

Always put spaces around operators (= + - * /), and after commas.

4.1.3 Statement Rules

General rules for simple statements: Always end a simple statement with a semicolon. General rules for complex (compound) statements: Put the opening bracket at the end of the first line, use one space before the opening bracket, Put the closing bracket on a new line, without leading spaces, and Do not end a complex statement with a semicolon.

4.1.4 Object Rules

General rules for object definitions: Place the opening bracket on the same line as the object name. Use colon plus one space between each property and its value. Use quotes around string values, not around numeric values. Do not add a comma after the last property-value pair. Place the closing bracket on a new line, without leading spaces and always end an object definition with a semicolon.

4.1.5 Line Length

For readability, avoid lines longer than 80 characters. If a JavaScript statement does not fit on one line, the best place to break it is after an operator or a comma. The line length helps the code to be formatted in a well appealing form and the readability of the code will also be improved.

4.1.6 Naming Conventions

Always use the same naming convention for all your code. For example, Variable and function names written as camelCase. Global variables are written in UPPERCASE (We don't, but it's quite common), Constants (like PI) written in UPPERCASE. Hyphens in HTML and CSS: HTML5 attributes can start with data- (data-quantity, data-price). CSS uses hyphens in property-names (font-size). Hyphens can be mistaken as subtraction attempts. Hyphens are not allowed in JavaScript names. Underscores: Many programmers prefer to use underscores (date_of_birth), especially in SQL databases. Underscores are often used in the PHP documentation. Pascal Case: Pascal Case is often preferred by C programmers. and camelCase: camelCase is used by JavaScript itself, jQuery, and other JavaScript libraries. Do not start names with a $ sign. It will put you in conflict with many JavaScript library names.

4.1.7 Loading JavaScript in HTML

Use simple syntax for loading external scripts (the type attribute is not necessary):

4.1.8 File Extensions

HTML files should have a .html extension (.htm is allowed). CSS files should have a .css extension. JavaScript files should have a .js extension. Use Lower Case File Names Most web servers (Apache, Unix) are case sensitive about file names: london.jpg cannot be accessed as London.jpg. Other web servers (Microsoft, IIS) are not case sensitive: london.jpg can be accessed at London.jpg or london.jpg. If you use a mix of upper and lower case, you have to be extremely consistent. If you move from a case insensitive to a case sensitive server, even small errors can break your web site. To avoid these problems, always use lowercase file names.

4.1.9 Performance

Coding conventions are not used by computers. Most rules have little impact on the execution of programs. Indentation and extra spaces are not significant in small scripts. For code in development, readability should be preferred. Larger production scripts should be minified.

4.2 Twelve-Factor Methodology

The twelve-factor methodology is a set of twelve best practices to develop applications developed to run as a service. This was originally drafted by Heroku for applications deployed as services on their cloud platform, back in 2011. Over time, this has proved to be generic enough for any software-as-a-service (SaaS) development.

Microservice is an architectural style to develop software as loosely coupled services. The key requirement here is that the services should be organized around business domain boundaries. This is often the most difficult part to identify.

Moreover, a service here has the sole authority over its data and exposes operations to other services. Communication between services is typically over lightweight protocols like HTTP. This results in independently deployable and scalable services.

Now, microservice architecture and software-as-a-service are not dependent on each other. But it's not difficult to understand that, when developing software-as-a-service, leveraging

the microservice architecture is quite beneficial. It helps to achieve a lot of goals we discussed earlier, like modularity and scalability.

4.2.1 Codebase

The first best practice of twelve-factor apps is to track it in a version control system. Git is the most popular version control system in use today and is almost ubiquitous. The principle states that an app should be tracked in a single code repository and must not share that repository with any other apps.

Spring Boot offers many convenient ways to bootstrap an application, including a command-line tool and a web interface. Once we generate the bootstrap application, we can convert this into a git repository.

4.2.2 Dependencies

Next, the twelve-factor app should always explicitly declare all its dependencies. We should do this using a dependency declaration manifest. Java has multiple dependency management tools like Maven and Gradle. We can use one of them to achieve this goal.

So, our simple application depends on a few external libraries, like a library to facilitate REST APIs and to connect to a database. Let's see how we can declaratively define them using Maven.

4.2.3 Configurations

An application typically has lots of configuration, some of which may vary between deployments while others remain the same.

In our example, we've got a persistent database. We'll need the address and credentials of the database to connect to. This is most likely to change between deployments.

A twelve-factor app should externalize all such configurations that vary between deployments. The recommendation here is to use environment variables for such configurations. This leads to a clean separation of config and code.

We can use a configuration management tool like Ansible or Chef to automate this process.

4.2.4 Backing Services

Backing services are services that the application depends on for operation. For instance a database or a message broker. A twelve-factor app should treat all such backing services as attached resources. What this effectively means is that it shouldn't require any code change to swap a compatible backing service. The only change should be in configurations. Spring JPA makes the code quite agnostic to the actual database provider.

Spring detects the MySQL driver on the class path and provides a MySQL-specific implementation of this interface dynamically. Moreover, it pulls other details from configurations directly.
So, if we've to change from MySQL to Oracle, all we've to do is replace the driver in our dependencies and replace the configurations.

4.2.5 Build, Release and Run

The twelve-factor methodology strictly separates the process of converting codebase into a running application as three distinct stages:

Build Stage: This is where we take the codebase, perform static and dynamic checks, and then generate an executable bundle like a JAR. Using a tool like Maven, this is quite trivial.

Release Stage: This is the stage where we take the executable bundle and combine this with the right configurations. Here, we can use Packer with a provisioner like Ansible to create Docker images.

Run Stage: Finally, this is the stage where we run the application in a target execution environment. If we use Docker as the container to release our application, running the application can be simple enough.

Finally, we don't necessarily have to perform these stages manually. This is where Jenkins comes in as handy with their declarative pipeline.

4.2.6 Processes

A twelve-factor app is expected to run in an execution environment as stateless processes. In other words, they cannot store persistent state locally between requests. They may generate persistent data which is required to be stored in one or more stateful backing services.

A request on any of these endpoints is entirely independent of any request made before it. For instance, if we keep track of user requests in-memory and use that information to serve future requests, it violates a twelve-factor app.

Hence, a twelve-factor app imposes no such restriction like sticky sessions. This makes such an app highly portable and scalable. In a cloud execution environment offering automated scaling, it's quite a desirable behavior from applications.

4.2.7 Port Binding

A traditional web application in Java is developed as a WAR or web archive. This is typically a collection of Servlets with dependencies, and it expects a conformant container runtime like Tomcat. A twelve-factor app, on the contrary, expects no such runtime dependency. It's completely self-contained and only requires an execution runtime like Java.

Spring Boot, apart from many other benefits, provides us with a default embedded application server. Hence, the JAR we generated earlier using Maven is fully capable of executing in any environment just by having a compatible Java runtime.

Here, our simple application exposes its endpoints over an HTTP binding to a specific port like 8080. Upon starting the application as we did above, it should be possible to access the exported services like HTTP.

An application may export multiple services like FTP or WebSocket by binding to multiple ports.

4.2.8 Concurrency

Java offers Thread as a classical model to handle concurrency in an application. Threads are like lightweight processes and represent multiple paths of execution in a program. Threads are powerful but have limitations in terms of how much it can help an application scale.

The twelve-factor methodology suggests apps to rely on processes for scaling. What this effectively means is that applications should be designed to distribute workload across multiple processes. Individual processes are, however, free to leverage a concurrency model like Thread internally.

A Java application, when launched gets a single process which is bound to the underlying JVM. What we effectively need is a way to launch multiple instances of the application with intelligent load distribution between them. Since we've already packaged our application as a Docker container, Kubernetes is a natural choice for such orchestration.

4.2.9 Disposability

Application processes can be shut down on purpose or through an unexpected event. In either case, a twelve-factor app is supposed to handle it gracefully. In other words, an application process should be completely disposable without any unwanted side-effects. Moreover, processes should start quickly

For instance, in our application, one of the endpoints is to create a new database record for a movie. Now, an application handling such a request may crash unexpectedly. This should, however, not impact the state of the application. When a client sends the same request again, it shouldn't result in duplicate records.

In summary, the application should expose idempotent services. This is another very desirable attribute of a service destined for cloud deployments. This gives the flexibility to stop, move, or spin new services at any time without any other considerations.

4.2.10 Dev/Prod Parity

It's typical for applications to be developed on local machines, tested on some other environments and finally deployed to production. It's often the case where these environments are different. For instance, the development team works on Windows machines whereas production deployment happens on Linux machines.

The twelve-factor methodology suggests keeping the gap between development and production environment as minimal as possible. These gaps can result from long development cycles, different teams involved, or different technology stack in use.

Now, technology like Spring Boot and Docker automatically bridge this gap to a great extent. A containerized application is expected to behave the same, no matter where we run it. We must use the same backing services – like the database – as well.

Moreover, we should have the right processes like continuous integration and delivery to facilitate bridging this gap further.

4.2.11 Logs

Logs are essential data that an application generates during its lifetime. They provide invaluable insights into the working of the application. Typically an application can generate logs at multiple levels with varying details and output ii in multiple different formats.

A twelve-factor app, however, separates itself from log generation and its processing. For such an app, logs are nothing but a time-ordered stream of events. It merely writes these events to the standard output of the execution environment. The capture, storage, curation, and archival of such stream should be handled by the execution environment.

There are quite several tools available to us for this purpose. To begin with, we can use SLF4J to handle logging abstractly within our application. Moreover, we can use a tool like Fluentd to collect the stream of logs from applications and backing services.

This we can feed into Elasticsearch for storage and indexing. Finally, we can generate meaningful dashboards for visualization in Kibana.

4.2.12 Admin Processes

Often, we need to perform some one-off tasks or routine procedure with our application state. For instance, fixing bad records. Now, there are various ways in which we can achieve this. Since we may not often require it, we can write a small script to run it separately from another environment.

Now, the twelve-factor methodology strongly suggests keeping such admin scripts together with the application codebase. In doing so, it should follow the same principles as we apply to the main application codebase. It's also advisable to use a built-in REPL tool of the execution environment to run such scripts on production servers.

4.3 Screenshots

As UC eBanking Prime is a running application and all the rights is owned by UC Bank, Germany. The user interface and the layout of the application remains confidential as there are competitors in the same field. The image has been blurred keeping in mind the non-disclosure agreement signed and keeping the privacy in mind.

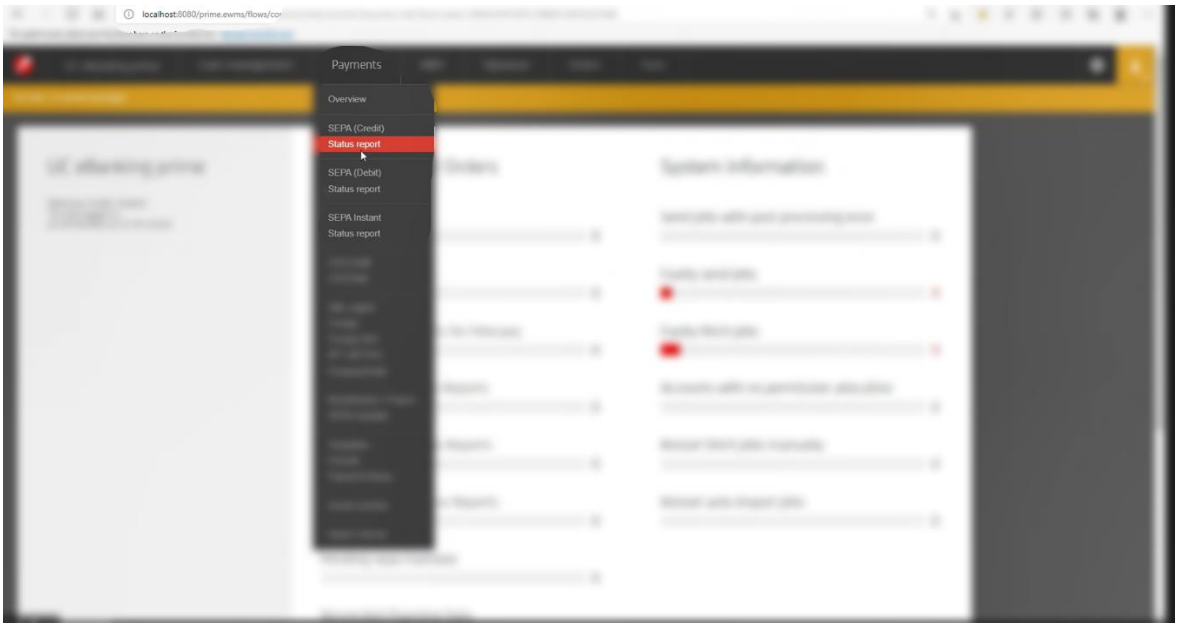


Fig. 4.1 UC eBanking Prime Dashboard

The above figure shows the dashboard of UC eBanking Prime and the visible dropdown highlights the Payment module and the different sub-modules under it namely SEPA Credit, SEPA Debit, Foreign Payment, etc. The application is run on a secure virtual machine and is timed out after every certain interval.

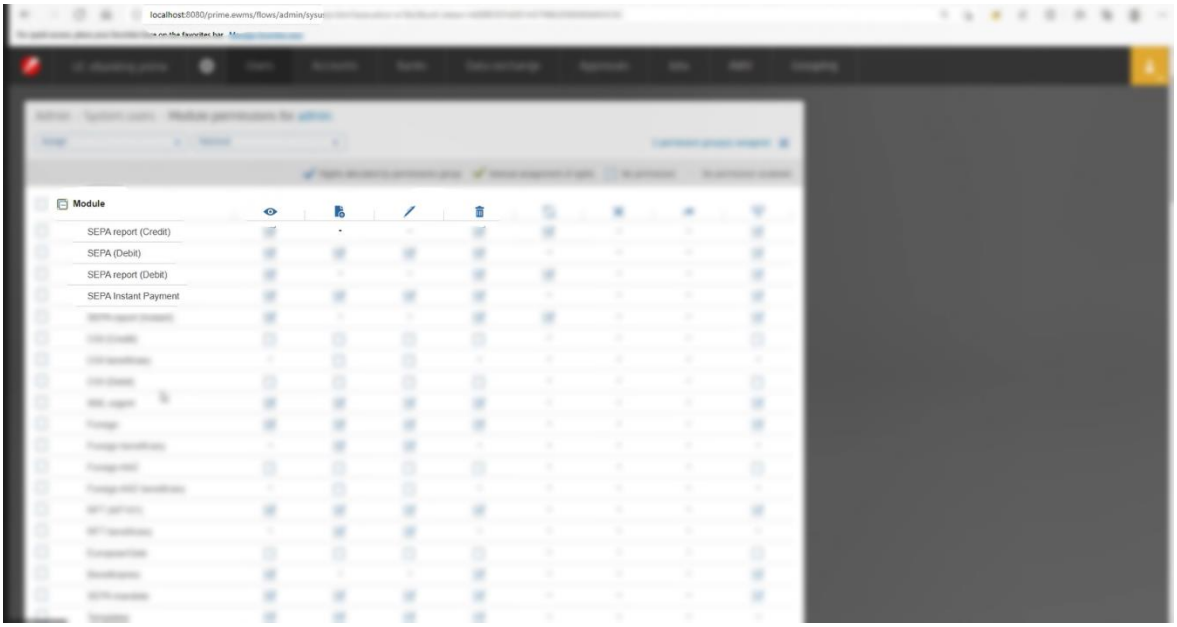


Fig 4.2 Admin rights for the Payment Module

In Figure it shows the different payment permission the admin can give to the users these permissions are different for all the sub modules. Moreover the admin can also update the permissions from time to time.

5. CONCLUSION

The application UC eBanking Prime was first launched in 2008 and currently contains 8 lac lines of Java code alone. It becomes difficult to understand a code or to solve errors in the application. Moreover, making changes in such a huge application where things are interconnected becomes a real havoc.

As the name suggests, microservices are micros, which break up a monolith app into a set of independent services. Unlike slow and heavy monolith architectures, microservices are faster to develop and deploy. Migrating from a monolithic application to microservices also enables you to optimize resources, enhance collaboration and streamline business processes. Microservices simplify app management, making it easier for you to build, deploy, update, test and scale each service independently.

Implementing the concept of Microservices and splitting the current monolithic application into microservices would be very beneficial keeping in mind the size and complexity of the application.

The microservices approach designs applications as a set of small services. Every service is a separate application with its own framework, programming language, and database. They are all developed and deployed separately from other services. They communicate through inter-process communication protocols like HTTP, AMQP, or a binary protocol like TCP.

Complex applications with many components (UC eBanking Prime) that need to scale require a microservices architecture. Amazon, Netflix, eBay, Uber, Groupon, and SoundCloud, are only some of the worldwide giants that have adopted microservices.

Microservices allow immediate releases, as CI/CD pipelines are easily integrated. Deployments can happen thousands of times a day. At the same time, the application is running as usual, as it consists of many different small components.

Hence, starting with the Payment module breaking the whole UC eBanking into different services is the future plan keeping all the features intact and not degrading the performance of the application.

5.1 Advantages and Limitations

There are multiple reasons to consider for migrating to microservices from monolithic, including:

- Decentralized
  Monolithic systems use a single database for different apps, whereas microservices have a separate database for each service. Microservices has decentralized data management and governance.

- Granular scaling
  You can auto-scale up or down individual services without scaling the entire app, using microservices. Thus, its simplicity improves the quality of a definite functionality.

- Myriad components
  You can break down microservices architecture into several component services. Then, you can independently deploy, revamp and redeploy these services.

- Simple routing
  The functioning of microservices is like that of UNIX systems. They have a much simpler routing process which starts from getting requests, processing them, and lastly, creating a response accordingly.

- Transformative
  The design of microservices architecture is evolutionary. Thus, microservices are an ideal choice for transformative systems and projects.

- Failure proof
  In a microservices architecture, the failure of a single module will not affect larger apps. Moreover, the design of microservices is such that it can help you cope with failure in diverse services.

These are the reasons that you must consider shifting to microservices.

There are some significant challenges to converting your mission critical application into smaller disparate parts, the least of which is maintaining uptime while you deploy your services.

- Increased application complexity
  One of the biggest downsides of moving from a single application package to many is a significant increase in application complexity. You now have to worry about many small applications instead of one.

Documentation will also have to cover the new services as well as integration points. Test plans and code will have to cover both the smaller services as well as a full integration test plan every time you manipulate your services.

Technical debt will also increase significantly, and intentional efforts must be made to make sure all team resources understand the smaller services that power your application.

- Inter-service communication and persistent data across services
  Communicating across services becomes more difficult when your databases are separated. While some great software packages, like gRPC, exist to make inter-service communication possible you still have the same race condition issues and

  Transactional processes become very difficult since the databases are specific to the service of groups of services. Transaction definitions must be maintained by the ORM or model service instead of the persistent data storage.

- Health monitoring and debugging become difficult at scale
  You must now monitor the health of several servers. You can no longer rely on a heartbeat monitor on your web server to make sure your application is still up and running.

REFERENCES

[1] Brent Frye, 8 Steps for Migrating Existing Applications to Microservices, SEPTEMBER 28, 2020, https://insights.sei.cmu.edu/blog/8-steps-for-migrating-existing-applications-to-microservices/

[2] Sam Newman, Monolith to Microservices, https://samnewman.io/books/monolith-to-microservices/

[3] Chris Richardson, https://microservices.io/

[4] Martin Fowler, https://martinfowler.com/

[5] Kumar Chandrakant, Twelve-Factor Methodology in a Spring Boot Microservice, January 13, 2022, https://www.baeldung.com/spring-boot-12-factor

[6] Rutva Safi, October 7th, 2021, https://www.softwebsolutions.com/resources/migrate-to-microservices-frommonolithic.html#:~:text= Migratin%20from%20a%20monolithic%20application,and%20scale%20each%20service%20independently

[7] Jax London, https://jaxlondon.com/wp-content/uploads/slides/Build_a_12-factor_microservice_in_an_hour.pdf

[8] Sara Miteva, Mar 3rd 2020, https://medium.com/microtica/why-transition-from-monolith-to-microservices-6c48449a3feb

[5] Wikimedia Foundation Contributors, “HTML”, Wikimedia Foundation, February 28, 2021, Wikipedia, March 21, 2021, https://en.wikipedia.org-/wiki/HTML

[6] Wikimedia Foundation Contributors, “Bootstrap”, Wikimedia Foundation, March 16, 2021, Wikipedia, March 21, 2021, https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

[7]. OpenJS Foundation Contributors, “jQuery”, OpenJS Foundation, November 20, 2020, OpenJS Foundation, March 21, 2021 https://jquery.com/

[8]. Wikimedia Foundation Contributors, “jQuery”, Wikimedia Foundation, March 5, 2021, Wikipedia, March 21, 2021, https://en.wikipedia.-org/wiki/JQuery

[9]. Wikimedia Foundation Contributors, “CSS”, Wikimedia Foundation, March 9, 2021, Wikipedia, March 21, 2021, https://en.wikipedia.org/-wiki/CSS

[10]. Wikimedia Foundation Contributors, “Node.js”, Wikimedia Foundation, March 18, 2021, Wikipedia, March 21, 2021, https://en.wikipedia-.org/wiki/Node.js