Applied Software Project Report

By

ANAND MOHAN TIWARI

A Master's Project Report submitted to Scaler Neovarsity - Woolf in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

October, 2024

Scaler Mentee Email ID: naman@scaler.com

Thesis Supervisor: Naman Bhalla

Date of Submission: 19/10/2024

Certification

I confirm that I have overseen / reviewed this applied project and, in my judgment, it adheres to the appropriate standards of academic presentation. I believe it satisfactorily meets the criteria, in terms of both quality and breadth, to serve as an applied project report for the attainment of Master of Science in Computer Science degree. This applied project report has been submitted to Woolf and is deemed sufficient to fulfill the prerequisites for the Master of Science in Computer Science degree.

Naman Bhalla

……

Project Guide / Supervisor

DECLARATION

I confirm that this project report, submitted to fulfill the requirements for the Master of Science in Computer Science degree, completed by me from 10 May 2022 to 23 Sep 2024, is the result of my own individual endeavor. The Project has been made on my own under the guidance of my supervisor with proper acknowledgement and without plagiarism. Any contributions from external sources or individuals, including the use of AI tools, are appropriately acknowledged through citation. By making this declaration, I acknowledge that any violation of this statement constitutes academic misconduct. I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving the degree.

ANAND MOHAN TIWARI

Signature of the Candidate **Date: 19 October 2024**

ACKNOWLEDGMENT

I would like to express my deepest gratitude to everyone who played a pivotal role in helping me achieve this milestone. To my beloved family, especially my wife, who stood by me through thick and thin, providing unwavering encouragement and patience, I am forever thankful. Your love, understanding, and constant support have been the backbone of my journey, pushing me to keep striving even during the toughest moments. To my parents and family members, thank you for believing in my dreams and always reminding me of the value of perseverance and hard work. Your faith in me fueled my drive to excel and stay committed to my goals.

I would also like to extend my heartfelt appreciation to my mentors and instructors at Scaler. Your dedication to imparting knowledge and your commitment to our success were truly inspiring. The countless hours you invested in teaching, clarifying concepts, and offering guidance were invaluable in shaping my understanding and strengthening my technical expertise. Your insights and advice went beyond the curriculum and will continue to influence my professional journey in profound ways.

Additionally, I am grateful to my peers and friends who have been a source of motivation and healthy competition. Working alongside such talented individuals pushed me to constantly improve and gave me the confidence to face any challenge. I am also deeply indebted to those who, even from a distance, inspired and motivated me, whether through their achievements, words of encouragement, or simply by believing that I could reach this goal. This Master's degree is a testament to all the support, love, and encouragement I received, and I am truly blessed to have such a remarkable network of people in my life.

Thank you all for being part of this journey. Without your unwavering support, this achievement would not have been possible.

Table of Contents

List of Tables

List of Figures

Abstract

The BookMyShow app revolutionizes the entertainment ticketing experience by providing a seamless platform for users to discover, book, and manage their entertainment choices, including movies, events, and activities. This project focuses on developing a user-friendly mobile application that caters to the evolving needs of today's consumers, emphasizing efficiency, convenience, and personalization.

The development process incorporates robust features such as real-time ticket availability, user reviews, personalized recommendations, and secure payment options. Through a comprehensive market analysis and user feedback, the app's design prioritizes an intuitive interface, ensuring that users can easily navigate through various categories, explore new content, and complete transactions with minimal effort.

Additionally, the BookMyShow app leverages advanced technologies, such as machine learning and data analytics, to enhance user engagement and retention. By analyzing user behavior and preferences, the app tailors’ recommendations to provide a personalized experience, thereby increasing user satisfaction and loyalty.

Ultimately, the BookMyShow app serves as a vital tool in the entertainment industry, bridging the gap between consumers and event organizers while contributing to the growth of digital ticketing solutions. The project aims to set a benchmark in app development by delivering a feature-rich, efficient, and secure platform that meets the dynamic demands of the entertainment landscape

Project Description

The BookMyShow project involves the development of a comprehensive mobile application designed to enhance the ticket booking experience for users seeking entertainment options, including movies, concerts, theater performances, and sporting events. The app aims to streamline the process of discovering and purchasing tickets while providing users with essential information about upcoming events, including showtimes, venue details, and seat availability.
The application will feature an intuitive user interface that allows for easy navigation, personalized recommendations based on user preferences, and secure payment options. Additionally, the app will include social features, such as user reviews and ratings, enabling users to make informed decisions while promoting community engagement.

The BookMyShow app aims to deliver a user-friendly interface that is both intuitive and aesthetically pleasing, enhancing the overall user experience while simplifying navigation through various entertainment options. The app focuses on providing a seamless booking experience by streamlining the ticket purchasing process, including real-time seat selection and secure payment gateways, to ensure convenience and safety for users. Additionally, personalized recommendations will be offered using data analytics and machine learning algorithms, enabling users to receive content and event suggestions based on their preferences and past behaviors. The app will feature comprehensive event listings, offering a diverse range of entertainment options such as movies, concerts, theater shows, and sports events, with detailed information on showtimes, venues, and ticket prices. To promote user engagement and community building, features will be integrated that allow users to leave reviews, rate events, and share their experiences, fostering a sense of community. Furthermore, real-time notifications will keep users informed of upcoming events, ticket availability, and special offers, ensuring continued engagement with the app**.**

The BookMyShow app is highly relevant in today’s digital landscape, where consumers increasingly prefer online solutions for their entertainment needs. As the entertainment industry continues to evolve, the demand for efficient and accessible ticketing solutions grows. The app offers convenience by providing a one-stop platform for discovering and booking tickets, meeting the modern consumer's preference for efficiency. With the market growth of the global online ticketing sector driven by the rise of digital platforms, BookMyShow is positioned to capture a significant share of this expanding market by offering a robust and feature-rich application. By focusing on a user-centric approach, the app ensures that users feel valued through personalized experiences. Enhanced engagement is fostered through community-building features such as user reviews and ratings, encouraging user loyalty and repeat bookings. Additionally, the app supports event organizers by providing a platform for them to reach a wider audience, promoting their events and boosting ticket sales.

Requirement Gathering

Functional Requirements

The functional requirements for the BookMyShow app define the specific behaviors or functions it must provide to users. The app must allow user registration and login using email, phone number, or social media accounts, with password recovery options readily available. For event browsing, users should be able to search events by category (such as movies, concerts, or sports) and filter or sort them by date, popularity, and user ratings. Users should also have access to event details, including a synopsis, cast information, reviews, location, and showtimes. The ticket booking feature must support real-time seat selection and purchasing, with multiple payment options such as credit/debit cards, digital wallets, and UPI. Additionally, users must be able to leave reviews and ratings for the events they attended. A favorite and Wishlist feature will enable users to save events for future reference. The app should also send push notifications to alert users about upcoming events, ticket availability, and special offers. On the organizer side, event management functionality will allow event organizers to create, update, and manage their events. Finally, the app will provide an analytics dashboard for administrators to view key performance metrics, sales data, and user engagement statistics.

Non-Functional Requirements

The non-functional requirements for the BookMyShow app emphasize its performance, usability, and security. The app must support high concurrent user access without performance degradation, ensuring that event listings and details load in less than three seconds. In terms of usability, the app should feature an intuitive user interface, accessible to users of all ages, and a responsive design across various devices and screen sizes. To ensure security, the app must implement secure payment gateways to protect user financial information, and user data privacy must be safeguarded through encryption and secure authentication methods. Scalability is another important aspect; the system architecture should be able to handle a growing user base and an increasing number of event listings. Lastly, the app must offer high availability, operating 24/7 with minimal downtime for maintenance.

User Identification

The primary users of the BookMyShow app are divided into three main groups. End Users are individuals looking to book tickets for various entertainment options, including movies, concerts, and sports events. Event Organizers consist of companies or individuals who promote events, such as movie studios, concert promoters, and sports teams. Lastly, Administrators include app managers, customer support teams, and marketing personnel responsible for managing the app's operations.

Use Cases

The following table outlines the use cases describing the interactions between different users and the BookMyShow app:

Use Case Name	Actor	Description
User Registration	End User	Allows users to create an account using email, phone number, or social media.
User Login	End User	Allows users to log into their accounts with registered credentials.
Browse Events	End User	Allows users to search, filter, and sort events based on categories, date, and ratings.
View Event Details	End User	Allows users to view detailed information about a selected event.
Book Tickets	End User	Allows users to select seats and purchase tickets for events in real-time.
Leave Reviews	End User	Allows users to rate and review events they have attended.
Manage Events	Event Organizer	Allows event organizers to create, update, and manage their events.
View Analytics	Administrator	Allows admins to view metrics related to app performance and user engagement.

Table 1: Use Case

Feature Set of BookMyShow App

Feature ID	Feature Name	Description	User Type
F-1	User Registration	Allows users to register for an account.	End User
F-2	User Login	Provides access to registered users.	End User
F-3	Event Browsing	Enables users to search and filter events by categories.	End User
F-4	Event Details	Displays detailed information for each event.	End User
F-5	Ticket Booking	Facilitates the purchase of tickets in real-time.	End User
F-6	User Reviews	Allows users to submit ratings and reviews for events.	End User
F-7	Favorites and Wishlist's	Enables users to save events for future reference.	End User
F-8	Push Notifications	Sends alerts for upcoming events and special offers.	End User
F-9	Event Management	Allows organizers to create and manage their events.	Event Organizer
F-10	Analytics Dashboard	Provides admins with insights into app performance and user behavior.	Administrator

Table 2: Feature Set of BookMyShow App

Payment Integration

Defining Payment Gateway

A payment gateway is a customer's payment information guardian. It may be a POC terminal in a physical store or a checkout portal on a website. The function of the payment gateway is to accept financial credentials and transfer them to a merchant's bank account.

Benefits of Custom Payment Gateways

Custom development has advantages when discussing fintech services. But if you decide to engage in payment gateway system design, you'll get certain benefits, including:

- Saving on development and subscription fees.
- Ability to modify user's infrastructure according to your business needs.
- Ability to process multi-currency transactions.
- Increasing the profits (since you don't have to rely on third-party apps, which require commission).
- Challenges of Custom Payment Gateway Development
- For as long as you're designing a payment gateway architecture, you may face some of these issues:

High pricing: fintech software development is quite expensive because you create the whole system from a scratch.
Time-consuming: building a solution, compatible with other systems and personal software processes (PSPs) cannot be done in a few days. Not even in a month.
Cybersecurity: you'll need good experts in the area to ensure the modules follow security protocols.

Payment Gateway Architecture: The Main Points

A payment gateway is a transit station between a customer placing an order and the issuing bank receiving money. The key factors to consider for payment system architecture include below main points as in figure: -

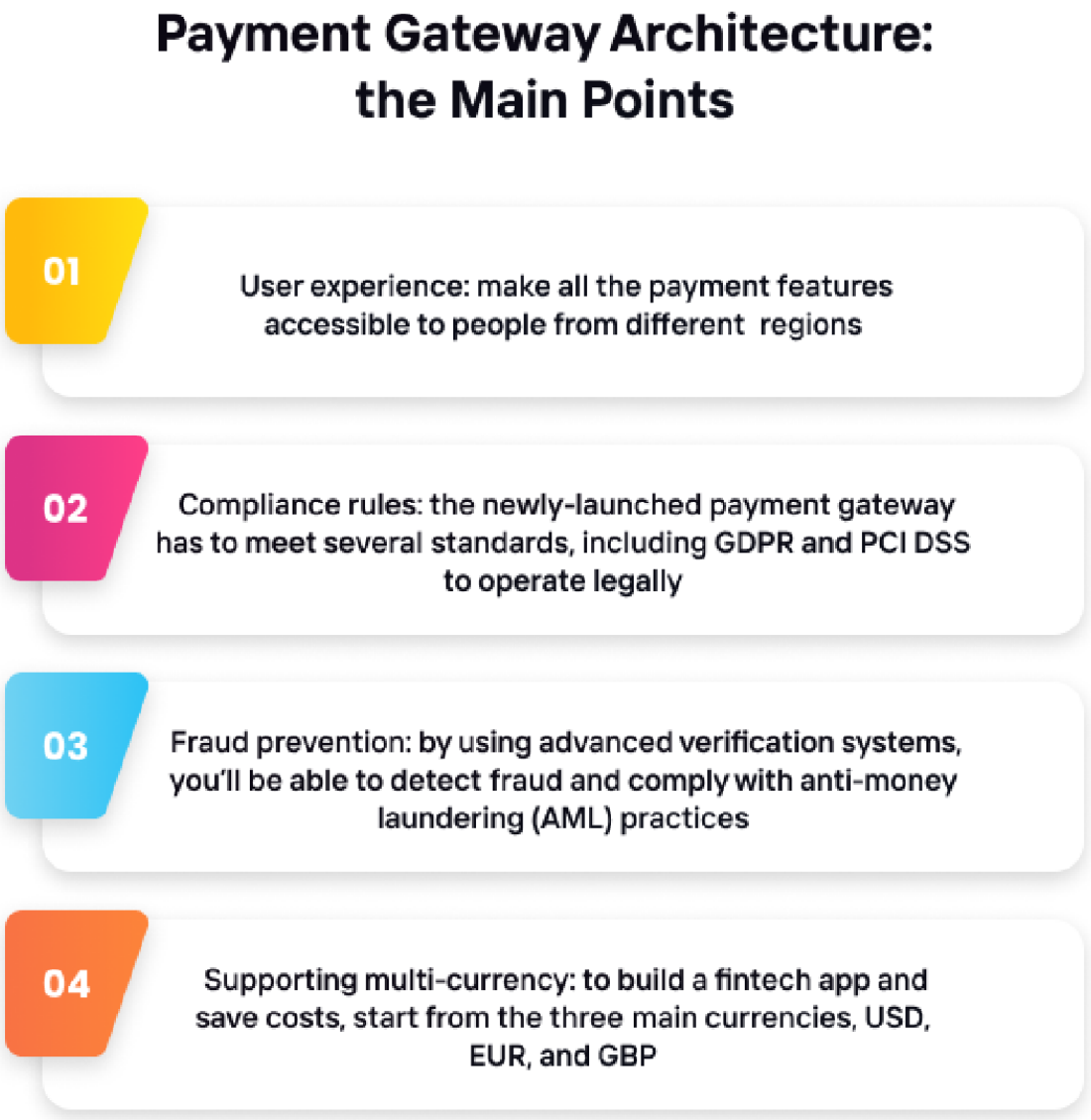

Fig 1 Payment Gateway Architecture

The main perk payment gateway has is a seamless transaction. People don't even notice how money goes from a credit card, because it happens in a matter of seconds. The truth is, your costs pass a complicated itinerary before they reach a merchant's bank.

Payment System Design Architecture: How It Works

Let's take a routine scenario when a customer wants to buy something online. Here's a look at payment architecture at work:

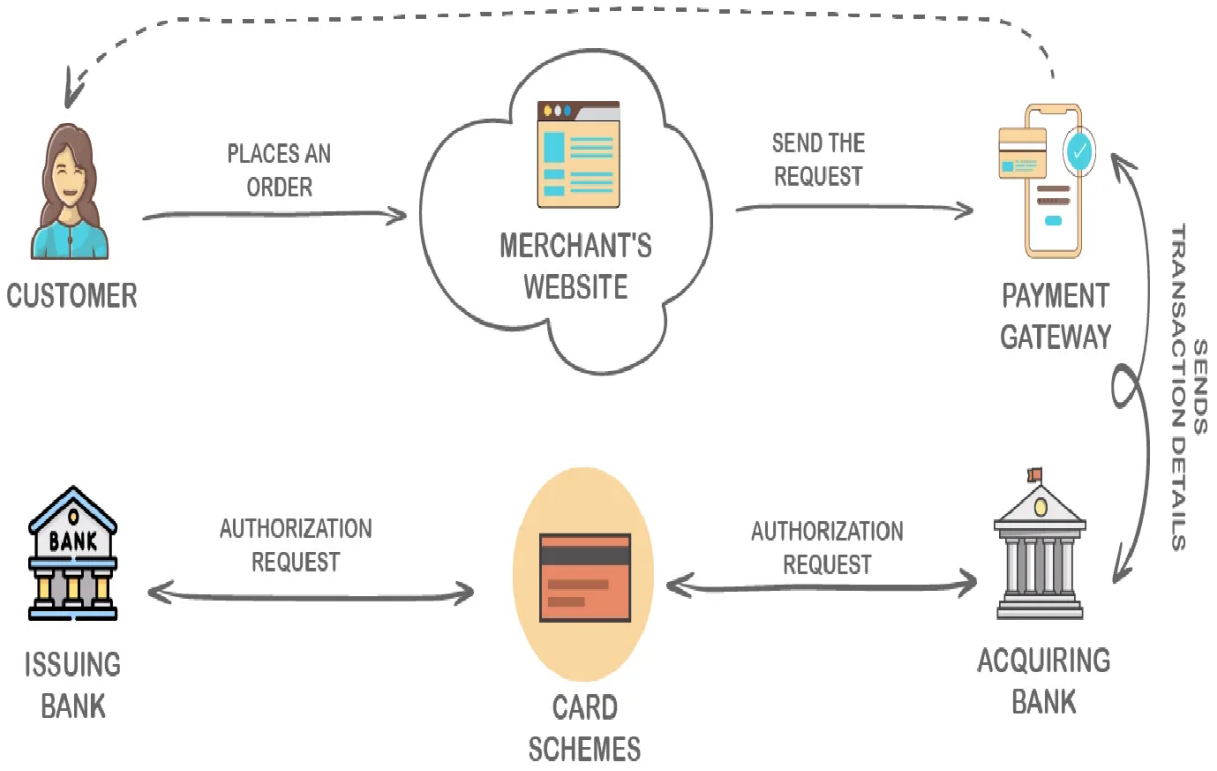

Fig 2 Payment System Design Architecture

If there's no problem with the payment procedure (the customer has enough costs and no limit on a card), the issuing bank informs PG about a successful transaction. That's when the user gets an online receipt and receives an order confirmation.

The Reasons Online Businesses Need Payment Systems

Payment architecture is necessary for any business, making transactions online (financial institutions, eCommerce industry, and subscription services), due to several reasons:

- Security protocols, encrypting financial data: so, the customers will ensure their credit card details are safely stored for future payments
- Payment system sends funds directly to the receiver without interruptions
- Flexibility in terms of payment methods: you can accept Mastercard, Visa, American Express, or any other provider, increasing the number of international customers
- Streamlining payment processes: you'll be able to assist more customers, receive more feedback, improve your company's reputation, and increase revenue.

The Types of Payment Gateways

A payment gateway is a relatively new approach to online banking and processing online transactions. With technological advancements, business owners have quickly moved from direct bank transfers to more secure online payments.

You have to adjust a payment architecture in accordance with what the business needs. That's why there are different types of payment gateways, each one with its specific features. Besides, you have to integrate a payment gateway into your website to start gaining customers.

Here's a short overview of payment gateways, depending on their development strategy.

Payment Systems Based on Distributed Architecture

The distributed architecture includes the software components, which are fully independent, and can function across various geographic locations. These components communicate via short messages and have to complete the tasks assigned to them.

Using distributed system architecture, your business will meet the highest requirements of reliability and speed, making software scalable at the same time. One of the classic examples of such a gateway is PayPal.

Payment Systems Following Security Protocols

Safety is the most vulnerable feature every payment system has. By choosing to check out online, customers entrust you with their payment information, and, therefore, all the money they can get from a credit card.

The platforms with solid security protocols are Stripe, Square, and PayPal. They are considered one of the best systems for secure transactions.

Before creating a payment platform, make sure to comply with the following standards:

- PCI DSS Level 1: to avoid identity theft
- PSD2 (stands for "Payment Services Directive"): to regulate the third-party financial providers
- KYC (stands for "Know Your Customer"): to run verification account checks from time to time
- AML (stands for "Anti-Money Laundering"): to protect the accounts from criminal monetary activity.

Payment Systems Supporting Subscriptions

These systems are the ones providing recurrent payments; they repeat certain transactions in an agreed period of time (usually, a month).

The best way to set yourself up a recurring payment system is to have an all-in-one merchant account, that handles transactions, plus reliable security features (as customers keep their information online).

A subscription is something a modern person can't live without. Therefore, no need to mention the giants like Netflix or Apple Music, featuring this type of payment gateway.

Payment Systems with SOA-Oriented Architecture

A service-oriented architecture (SOA) is a type of software development, that uses web services for business applications. Business owners implement SOA together with simple object access protocol (SOAP) and web services description language (WSDL).

There are three main roles in SOA building blocks: service provider, service repository, and service consumer.

The main benefit of SOA is that each service has its database and performs a specific function. The services communicate through application programming interfaces (APIs) only. The developers use the SOA principle for developing mobile banking apps.

You may find different payment gateways and common examples in the tab below.

Hosted PG	Self-hosted PG	API hosted PG	Local bank integration gateways
During checkout, clients go to the Payment Service Provider (PSP) site.	The payment form is inside the merchant's site, not on PSP	The merchant's website sends the client's information to PG through the API interface	The customers put their payment info on the bank's website
Paypal, Amazon Pay, Unicorn Payment	QuickBooks Commerce's B2B Payments, Shopify Payments	Stripe, Razorpay, Instamojo, PayU	Authorize.Net, Payza, SecurionPay

Table 3 Payment Gateway examples

What to Consider When Choosing a Payment System Architecture

Finding the best payment option for your platform is a long process. Do your research, and find opportunities to sell and attract more customers. Don't forget, that even though payment is the final stage of purchase, it defines the customer's feedback and the Repeat Customer Rate (RCR).

The main factors to consider, when looking for a platform gateway are

- Security: the security standards should comply with the PCI DSS rules, and meet the main requirements of fighting fraudulent or scam activities.
- Seamless website integration: payment gateway needs to have reliable customer support, integrate with other applications, and quickly resolve any issues.
- Time of payout: the reliable PG services are stable in payout terms. The time varies for every company, but the average rate of payouts should be every two weeks.
- Multi-currency support: as mentioned earlier, you should let international customers come to you. And having payment options with different currencies is the best way to do so!

The main features of the payment system are integration and security. Customers entrust you with their payment information, wanting to receive exceptional service. That's why you have to consider these four main factors to avoid unpleasant situations in the future.

5 Stages of Integrating a Payment Gateway into a Website

Before starting a payment architecture process, make sure you complete the below mentioned steps:

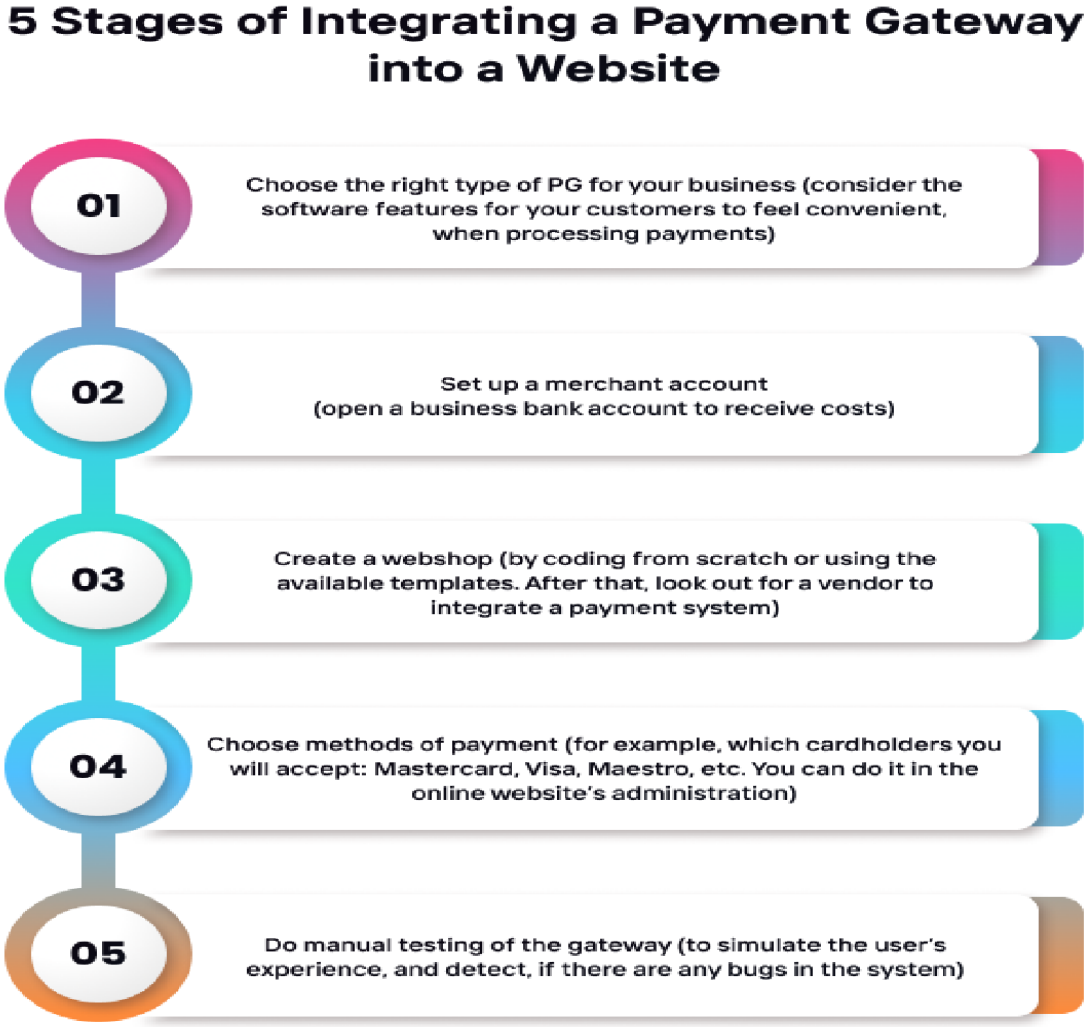

Fig 3 Stages of Integrating Payment Gateway

Razorpay Payment Integration

Selected Payment Gateway: Razorpay

Razorpay is a widely-used payment gateway in India, offering a comprehensive range of payment solutions, including credit and debit card processing, UPI, wallets, and more. It is highly regarded for its developer-friendly API and quick integration process, making it an excellent choice for online payment integration.

Integration Code and Documentation

The integration of Razorpay into a Node.js application can be carried out by following the detailed documentation provided on the Razorpay website. Below is a brief outline of the code structure:

- Node.js Integration: To integrate Razorpay into a Node.js application, refer to the documentation available at the Razorpay Node.js Integration Guide.

- React-Specific Integration: For React applications, Razorpay offers a well-documented guide on integrating their checkout system, which can be accessed here: Razorpay Checkout with React.

Code Example

Here is a simplified example of how to initiate a payment using Razorpay in a Node.js backend:

```
const Razorpay = require("razorpay");

const razorpay = new Razorpay({

  key_id: 'YOUR_KEY_ID', // Replace with your Razorpay Key ID

  key_secret: 'YOUR_KEY_SECRET' // Replace with your Razorpay Key Secret

});



// Create an order

const options = {
```

```
  amount: 50000, // Amount in paise (50000 paise = 500 INR)
  currency: "INR",
  receipt: "receipt#1",
};
razorpay.orders.create(options).then((order) => {
  console.log(order);
}).catch((error) => {
  console.error(error);
});
```

Security

PCI DSS Compliance

PCI DSS provides a list of requirements designed to enhance cardholder security. These requirements are divided into twelve major numbered parts and many subparts. This document references these part numbers to add context, but the section references are not an exhaustive list of applicable requirements.

Your PCI DSS compliance requirements vary depending on how your company handles payment card transactions (type) and how many transactions it performs each year (level).

As your number of transactions increases, your PCI DSS merchant level increases, and the PCI DSS compliance guidelines become stricter. At the highest merchant level, Level 1, PCI DSS requires an audit. Levels vary by the card brand. Level 1 is defined by American Express as 2.5 million annual transactions, and by Visa, Mastercard, and Discover as 6 million annual transactions. Each card brand has additional level requirements that are beyond the scope of this document. Ensure that your payment-processing environment is audited to support your merchant level.

Because Google Cloud is a Level 1 PCI DSS 4.0–compliant service provider, it can support your PCI DSS compliance needs no matter what your company's merchant level is. The Committed to compliance section lays out which areas are covered for you by Google.

The other fundamental variable is your SAQ type. The SAQ outlines criteria that you must address to comply with PCI DSS if you are eligible for self-assessment. Your SAQ type is determined by your app architecture and the precise way you handle payment card data. Most merchants in the cloud are one of the following:

SAQ type	Description
A	Merchants that have fully outsourced payment card processing to a third-party site. Customers leave your domain (including through an <iframe> web form), complete payment, and then return to your app. In other words, your company can't touch customer card data in any way.
A-EP	Merchants that outsource payment processing to a third-party provider, but who can access customer card data at any point in the process. Merchants that can access card data include merchant-controlled page elements such as JavaScript or CSS that are embedded in the third-party payment page. In other words, your payment processing app forwards card data to a processor on the client side, or the processor renders any content hosted by you.
D	Merchants that accept payments online and don't qualify for SAQ A or SAQ A-EP. This type includes all merchants that call a payment processor API from their own servers, regardless of tokenization. In other words, if you are not SAQ A or SAQ A-EP, you are SAQ D. SAQ D differentiates between merchants and service providers. Service providers are not discussed in this document, and all SAQ D references address merchants as defined in PCI DSS.

Fig 4 SAQ Type and Description

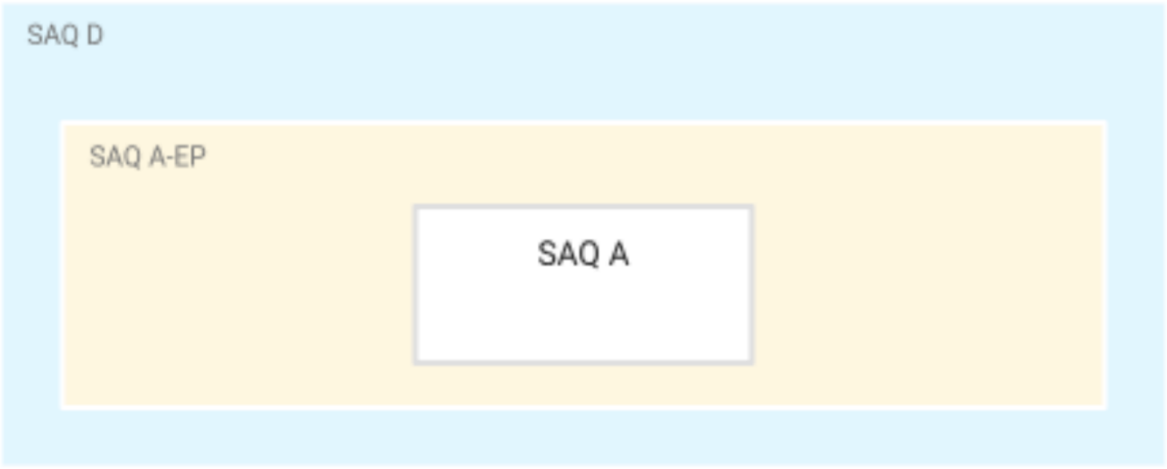

Fig 4 Venn diagram of SAQ requirements.

Architecture overview

SAQ A

SAQ A is the most basic payment-processing architecture. Payments are processed by a third party, and no card data is accessed by merchant apps or pages.

At a high level, the payment-processing flow is as follows:

1. The customer makes their selections and proceeds to check out.
2. The checkout app redirects the customer to a third-party payment processor.
3. The customer enters their payment card information into a payment form that the third-party processor owns and maintains.
4. The third-party payment processor checks the payment card information and then charges or declines the card.
5. After processing the transaction, the third-party payment processor sends the customer back to the merchant app along with transaction details.
6. The merchant app sends a verification request to the payment processor to confirm the transaction.
7. The payment processor responds to verify the transaction details.

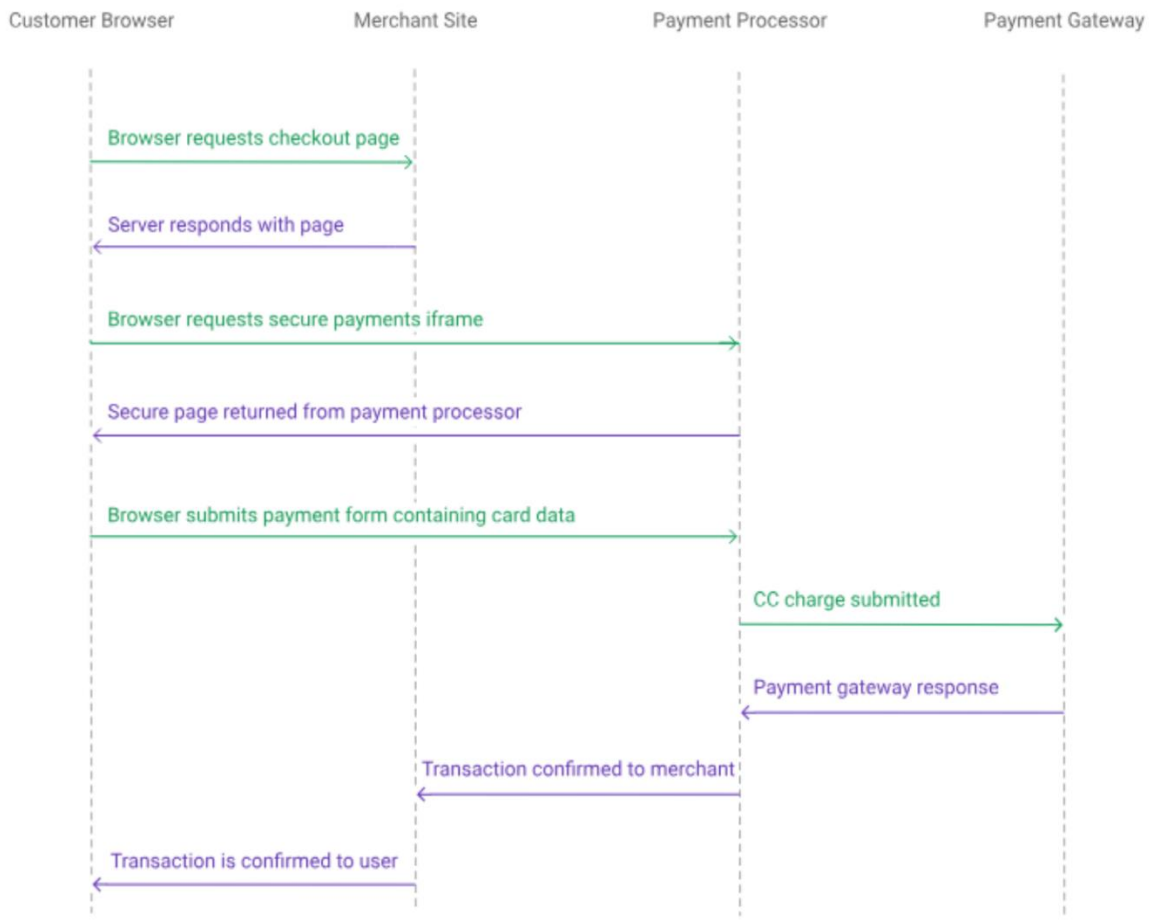

Fig 5 Architecture of an SAQ A third-party payment-processing environment.

SAQ A-EP

The SAQ A-EP payment-processing architecture is centered around a payment-processing app that runs on Compute Engine virtual machine instances. These instances are in a secure private network, and they use secure channels to communicate with services that are outside the network.

At a high level, the payment-processing flow is as follows:

1. The customer enters their payment card information into a payment form that your company owns and maintains.
2. When the customer submits their information, the form information is securely passed on to a third-party payment processor.
3. The third-party payment processor checks the payment card information and then charges or declines the card.

4. The payment processor sends a response back to your payment app, which then passes a message to your core app.

All of these interactions are logged and monitored with Cloud Logging and Cloud Monitoring.

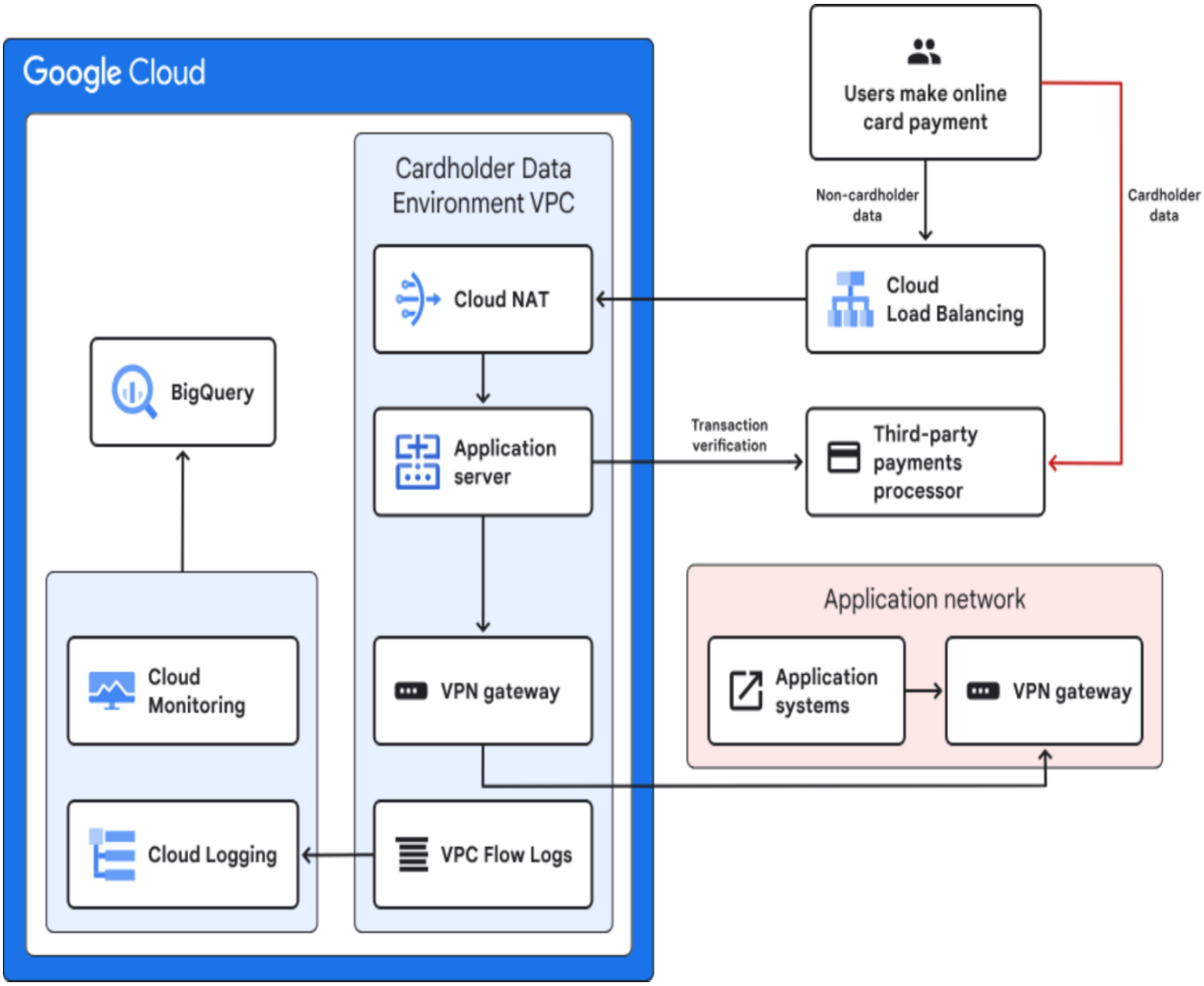

Fig 6 Architecture of an SAQ A-EP payment-processing environment.

SAQ D

The SAQ D payment–processing architecture centers around a payment-processing app that runs on Compute Engine virtual machine instances. These instances are in a secure private network and use secure channels to communicate with services that are outside the network.

At a high level, the payment-processing flow is as follows:

1. The customer enters their payment card information into a payment form that your company owns and maintains.
2. When the customer submits their information, your payment app receives the form information.

3. Your payment app validates the payment information and securely passes it on to a third-party payment processor through a backend API.
4. The third-party payment processor checks the payment card information and then charges or declines the card.
5. The payment processor sends a response back to your payment app, which then passes a message to your core app.

All of these interactions are logged and monitored with Logging and Monitoring.

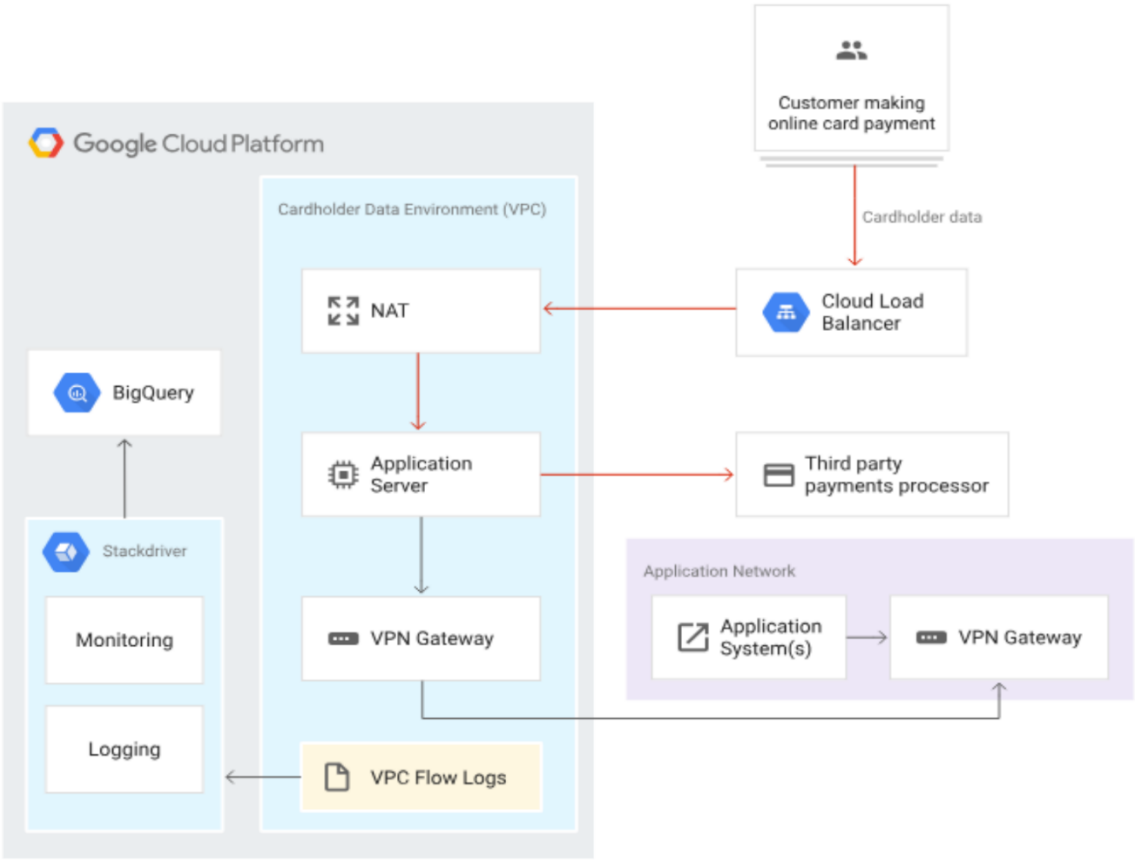

Fig 7 Architecture of a SAQ D payment-processing environment.

WEBHOOK

Role of Webhooks

From system failure events to trade and stock notifications, modern software simply cannot wait for a poling process or a nightly refresh. Webhooks are one way to integrate systems by a series of near-instant message updates, each tied to an independent event.

Webhook is a term used to describe a callback method in which one software system uses APIs to instantly notify another of an event. That means one application can send a web-based message request every time a qualifying event happens. To use webhooks effectively, the notifying system must possess the ability to register events and link them to a URL. When a particular event happens, like a buy order on a trading desk, the URL is called over the web with a payload of the event.

An API (in the form of the new URL) will then notify a second system. This flips the idea of the API around: The platform isn't asking for anything, but instead relays its own request message. Exactly what message this contains, as well as where it goes, is completely customizable, requiring only a user-supplied recipient web address.

The term webhook generally means posting an event to another system over the web through an API. Webhooks use HTTP POST messages to trigger actions in another application when an event fires. The current standard data format for this is JSON, but it's possible to build a webhook system using SMS or SMTP events. Developers can create command lines, set up configuration files or pipe it directly through a user interface to achieve this.

1. Application A registers a user operation as an event.
2. The event triggers and sends a HTTP POST request to Application A.
3. The Public API receives the POST request.
4. The Public API completes request and messages Application A to indicate the task is complete.

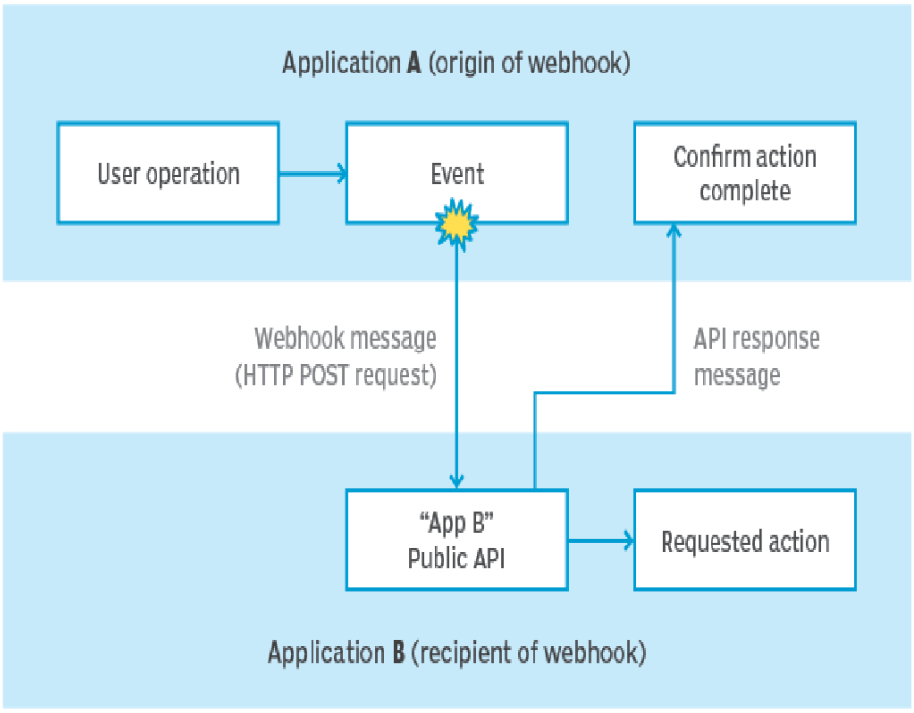

Fig 8 Webhook Design

HTTP Tunneling

HTTP tunneling is a technique that allows an HTTP request to pass through a firewall or proxy server. When the backend server is running locally during development, HTTP tunneling can be useful for exposing the local server to external services like Razorpay. Tools like Ngrok can be used to create a secure tunnel to the local server.

Security

Security is a crucial element of application development, especially when managing sensitive user information such as payment details and personal data. This section discusses the use of bcrypt for password hashing, the importance of salt in cryptography, and various Node.js security packages that can enhance the overall security measures of the BookMyShow app.

Bcrypt and Password Hashing

Bcrypt is a widely used password-hashing function designed to securely store passwords by making brute-force attacks significantly more difficult. It uses a computationally intensive key derivation function to generate secure password hashes. The generated hash is stored in the database rather than the actual password, ensuring that plaintext passwords remain protected even if the database is compromised.

A password-hashing function that ensures secure storage by generating a hash of the password, rather than storing it in plaintext form. Due to its computationally intensive nature, bcrypt makes brute-force attacks more difficult, increasing overall security. Bcrypt stores the hash of the password in the database, meaning even if the database is breached, attackers cannot easily recover the original passwords.

Salt and Encryption

Salt and encryption are important concepts in cryptography that further protect sensitive information such as passwords. A random value added to the password before hashing. It ensures that even if two users have the same password, their hashed values will differ because each user has a unique salt. Salt prevents rainbow table attacks, which rely on precomputed hashes to crack passwords.

Encryption is the process of converting plaintext data into a coded format that can only be decoded by someone with the proper key. While bcrypt hashing is a one-way function (irreversible), encryption is a two-way function that allows data to be encrypted and later decrypted.

Using Bcrypt for Password Hashing

To use bcrypt for password hashing in a Node.js application, follow these steps:

To install bcrypt, run the following command:

npm install bcrypt

Hashing a Password:

The following code shows how to hash a password using bcrypt in a Node.js application:

```
const bcrypt = require('bcrypt');

const saltRounds = 10; // The cost factor controls the time required to calculate a single bcrypt hash.

// Hashing a password
const password = 'user_password';
bcrypt.hash(password, saltRounds, function(err, hash) {
    if (err) throw err;
    // Store the hash in your password DB
    console.log(hash);
});
```

Deployment Flow

Deployment is a crucial phase in the software development lifecycle, ensuring that the application is accessible to users. This report outlines the deployment flow for the BookMyShow app, including code management, deployment options, and considerations for handling cross-origin resource sharing (CORS).

The deployment process for the BookMyShow app is a crucial aspect of ensuring that the application is properly set up and accessible to users. This section discusses the deployment workflow, including code management with GitHub, options for deploying both frontend and backend, and addressing CORS (Cross-Origin Resource Sharing) issues when connecting frontend and backend systems.

Code Management with GitHub

GitHub is a popular platform for version control and collaborative development. It allows developers to manage their codebase efficiently, track changes, and collaborate with team members. The code for the BookMyShow app should be stored in a GitHub repository, ensuring easy access for the team and streamlining the deployment process. Using GitHub will also facilitate continuous integration and deployment, making it easier to handle version control and updates.

Deployment Options

Render for Simple Deployment:

Render is a simpler alternative for deployment, allowing easy and quick deployment of web applications with minimal configuration. Render supports automatic deployments from GitHub, making it a convenient choice for teams that want a straightforward deployment process.

Frontend Deployment with Netlify

Netlify is a popular platform for deploying static websites and frontend applications. It supports continuous deployment from GitHub, enabling easy updates to the frontend as changes are made. The BookMyShow frontend can be deployed to Netlify, which can then be connected to backend APIs hosted on Render by configuring the appropriate API URLs.

Connecting Frontend and Backend

To connect the frontend deployed on Netlify with the backend hosted on Render, the frontend will need to interact with backend services for functionalities like user authentication, data retrieval, and payment processing. Ensure that the frontend is correctly pointing to the backend endpoints by setting up the API URLs in the frontend configuration.

Handling CORS Issues

CORS (Cross-Origin Resource Sharing) is a security feature implemented by web browsers to restrict web pages from making requests to a different domain than the one that served the page. When connecting a frontend hosted on Netlify to a backend hosted on Render, CORS issues may arise. These issues can be addressed by:

Backend Configuration: Ensure that the backend server allows requests from the Netlify domain by configuring CORS settings in the backend code.

Using Middleware: In a Node.js application, you can use the cors middleware to enable CORS easily. Here is an example:

```
const cors = require('cors');
const express = require('express');

const app = express();
app.use(cors({
  origin: 'https://your-netlify-app.netlify.app', // Replace with your Netlify domain
}));
```

Technologies Used

The BookMyShow app is developed using the MERN stack, which consists of MongoDB, Express.js, React.js, and Node.js. This report provides a detailed overview of each technology used in the project, along with examples of how they can be applied in real-life applications.

MERN Stack Overview

Node JS

Node.js has emerged to be one of the most popular web development frameworks. This is because of its fast, scalable, and robust architecture. Many big companies, like Netflix, Uber, Trello, PayPal, etc., have reaped the benefits of Node.js. The scaling of the application also became easy. This is all because of the architectural pattern followed by Node js.

Introduction

Node.js is completely free and open source. It is also used and supported by a large number of developers all around the world. Node.js can be called the combination of Chrome's V8 Js engine, event loop, and low-level I/O API. The below diagram shows the architecture of Node.js:

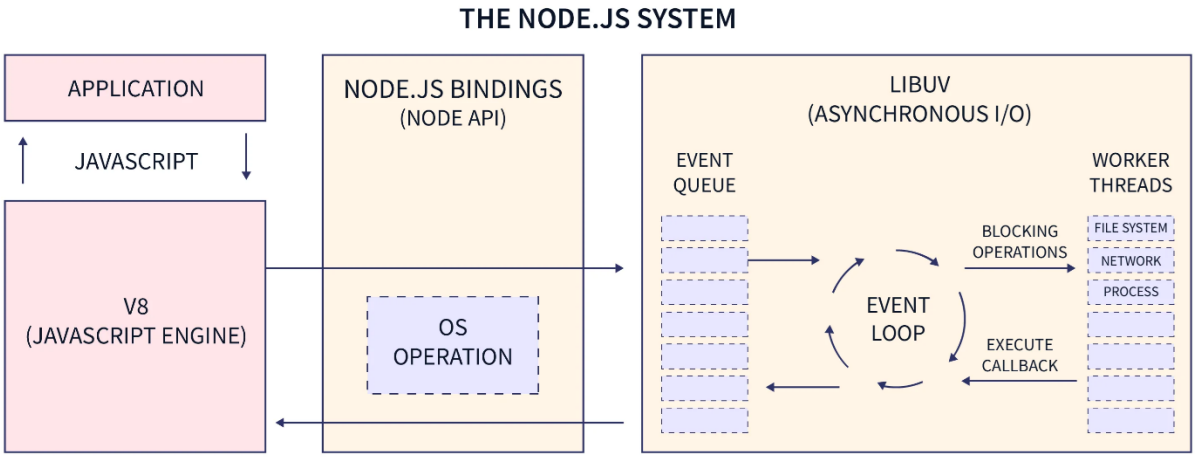

Fig 9 Node JS System

Node.js has its core part written in C and C++. Node js is based on a single-threaded event loop architecture which allows Node to handle multiple client requests. Node js uses the concept of an asynchronous model and non-blocking I/O. We will look at these terms in detail.

Node.js is one of the best options for other server-side platforms because of the architectural pattern it follows.

Single-Threaded Event Loop Architecture in Node.js

Node.js is based on the single threaded event driven architecture. It has a non-blocking I/O model. Let us look at these concepts in detail:

Node.js is Single Threaded Event Loop

The Event loop in Node.js is single-threaded. All requests may have two parts - synchronous and asynchronous. The event loop or the main thread takes up the synchronous part and assigns the asynchronous part to a background thread to execute. The main thread then takes up the synchronous part of other requests.

Example:

```
public class EventLoop {
   while(true) {
      if(Event Queue receives a JavaScript Function Call) {
                  ClientRequest request = EventQueue.getClientRequest();
            if(request requires BlokingIO or takes more computation time)
               Assign request to Thread T1
            Else
               Process and prepare a response
      }
   }
}
```

Node has non-blocking I/O Model

The Main thread in Node.js allocates the time-consuming asynchronous operation to a background thread. It does not wait for the background thread to complete that operation; it takes up the next operation in the event queue.

The main thread only handles the synchronous part of every request. After the background thread completes the async task, it notifies the main thread to take up the operation for further execution

of the callback code. Hence the main thread is not blocked while I/O is being performed by the async operation.

Example of the filesystem module. Following is an example of a blocking operation:

```
const fs = require('fs');

const data = fs.readFileSync('/file.txt'); // blocks here until file is read


console.log ('Node.js Architecture'); // Will execute after the file is read.
```

The asynchronous version of the above code that shows non-blocking:

```
const fs = require('fs');


fs.readFile('/file.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});


console.log('Node.js Architecture'); // Will execute before the file is read.
```

Event-Based, Instead of Waiting for I/O Operation

The thread running in the background performing some asynchronous operation notifies the main thread using Events. After the async task is completed, the associated callback function is needed to be called. If the main thread is busy executing some other request, then whenever it becomes idle, the main thread reacts to the event notification of the background thread. It runs the callback code and sends back the response to the client.

```
const fs = require('fs');


fs.readFile('/file.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

In the above code, after the file is read the associated callback function is executed.

Components of the Node.js Architecture

Let us see all the components one by one:

- **Requests:** Requests are sent by clients to fetch some server resources or by the server itself to some other server to fetch resources from it. Hence the resource can be of two types: incoming and outgoing.
- **Node js Server:** It is the server-side or the backend platform that receives requests from users at various endpoints defined. All the requests are processed by the server, and the response generated is sent back to the client. The server constantly listens on a specified port in the system. The clients send requests by specifying the target IP and port of the server.
- **Event loop:** It is an infinite loop that has six phases. All these six phases are repeated until there is no code left to execute. In these six phases, the event loop receives requests, processes them, and returns the response of these requests to the clients. The six phases of the event loop are:
    1. Timers
    2. I/O Callbacks
    3. Waiting / Preparation
    4. I/O Polling
    5. setImmediate() callbacks
    6. Close events

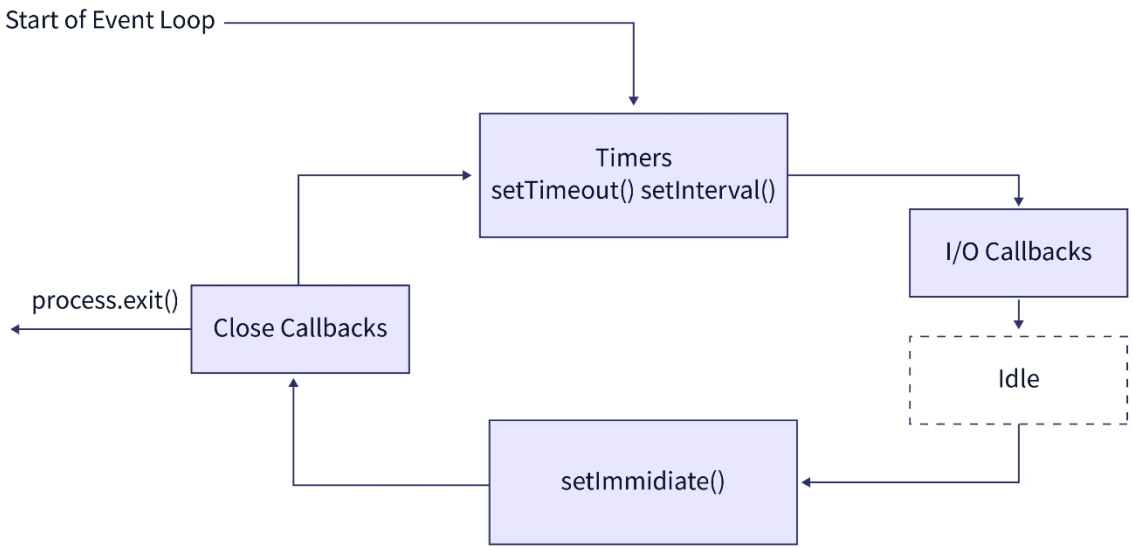

Fig 10 Components of node js Architecture

- **Event Queue:** All the requests sent by the clients are stored in the event queue. They are then passed one by one to the event loop. The callbacks of the operations running on other threads are also added to it so that the main thread takes them up.
- **Thread pool:** Other than the main thread, there are other threads managed in a thread pool. All the asynchronous processes and non-blocking I/O are attached to one of these threads as they continue executing in the background.
- **External resources:** These are the resources that the server has to fetch to fulfill client requests. They are needed to deal with the blocking requests. E.g., computation, data storage, etc.

Workflow of Nodejs Server

The workflow of a web server created with Node.js involves all the components discussed in the above section. The entire architectural work has been illustrated in the below diagram.

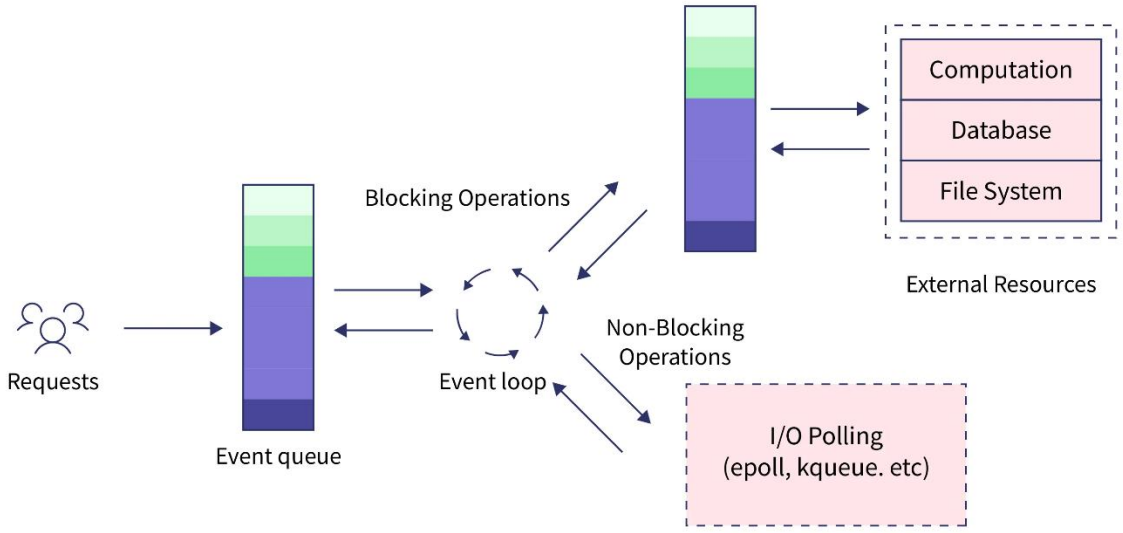

Fig 11 Workflow of node js Server

- Clients send requests to the web server. These requests can be either blocking (complex) or non-blocking (simple). The purpose of the requests may be to query for data, delete data or update data.
- The requests are retrieved and added to the event queue.
- The requests are then passed from the event queue to the event loop one by one.
- The simple requests are handled by the main thread, and the response is sent back to the client.
- A complex request is assigned to a thread from the thread pool.

Publisher-Subscriber Model

The pub/sub model is a messaging pattern in which the components publish messages and others subscribe to them. This is done to notify others and send data to them. Hence there are two components involved in this communication system - the publishers and the subscribers.

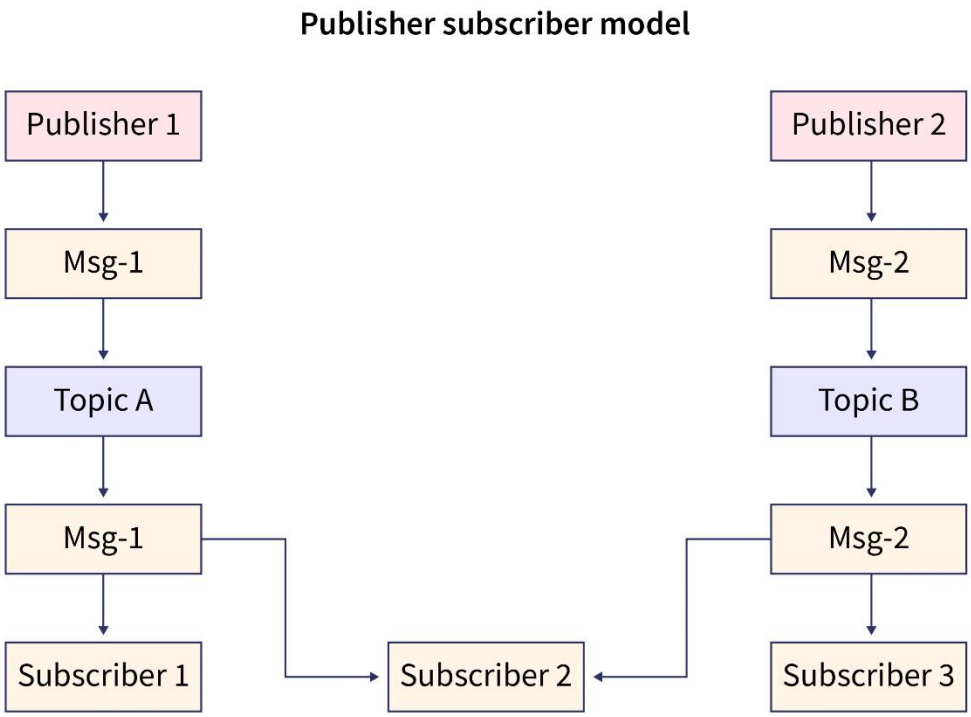

Fig 12 Publisher Subscriber Model

Messages are sent through specified channels by the publishers without any knowledge of the subscribers or the receiving end. The subscribers on the other end of these channels listen to these messages without the knowledge of the publishers. Hence multiple nodes can be connected using this data-sharing model. All the subscribers can listen to one action and respond to it.

Let us now understand the concept with the help of some examples. If we have to send notifications about some event to a large number of users, then it may require sending messages to the clients in a rapid way. This is a big challenge. It becomes bigger when the number of users gets scaled up. When the users are not active, there is a chance that users will miss these messages and not respond to them. Therefore, it falls on the system to make sure the client receives the message when it is active.

The pub/sub model allows the decoupling of the components, and each component is bound to rely on a message broker. The publishers send messages, and the clients subscribe to them to receive messages from the broker. The brokers are implemented to save the message and deliver the

message when the client comes online. Developers are hence not required to worry about the interface and structure of other components. This also allows the scalability of the system.

Adopt a Layered Approach

The use of express.js allows the easy distribution of the logic of the application into various categories. By dividing the concerns, the code quality improves and debugging becomes a lot easier. The setup separates the logic of the application from the API routes so that the background processes don't become complex. The code is divided into three categories - business logic, database, and API routes.

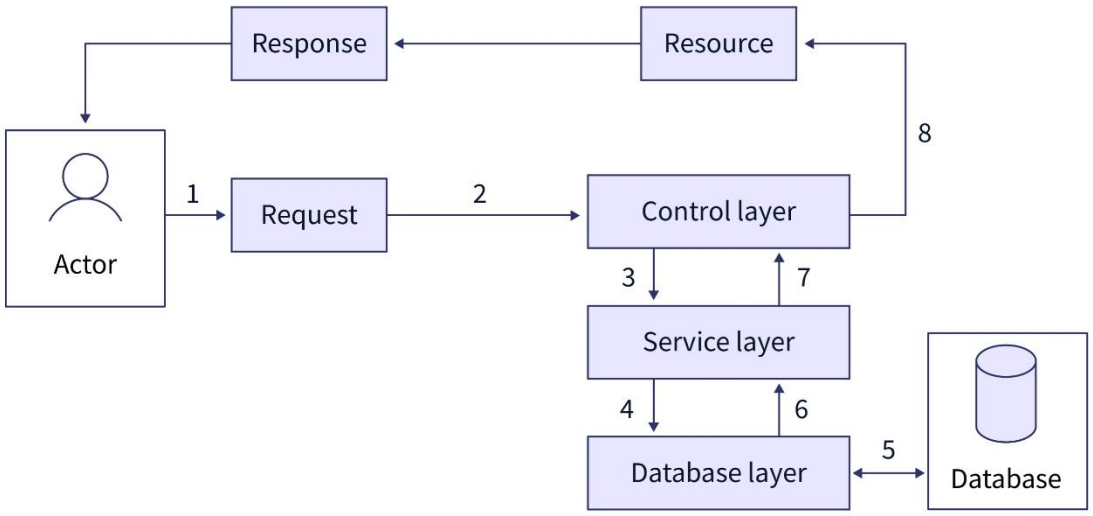

Fig 13 Layered Approach

Controller layer: This is where the API routes are defined. The request is destructured here, and based on the information in the request, some processing is done. The processing done is collected and passed on to the service layer.

Service layer: The business logic related to the application is defined here. The classes and functions used are included in this layer. The SOLID principles of Object-oriented programming are followed. The processing logic for various routes is also in this layer.

Data access layer: This layer takes care of the database handling. All the reading, writing, or any other manipulation of the database is handled by this layer. The SQL queries, connections to the database, document models, and other related code is defined here.

Use Dependency Injection

Dependency injection is a design pattern for software in which the dependencies of the software are passed in as parameters rather than including them or creating the dependencies inside the

software. This technique improves the flexibility, independence, scalability, and reusability of the modules. The testing also becomes easy.

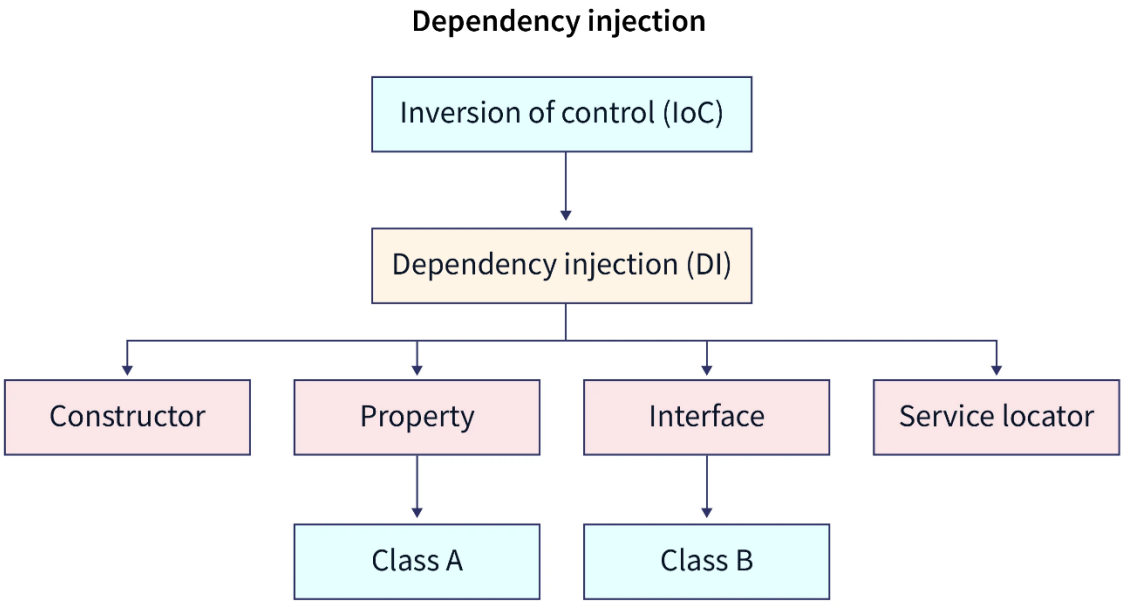

Fig 14 Dependency Injection

Let's take an example to understand this concept:

```
class PostManager {
  constructor(postStore) {
    this.posts = [];
    this.postStore = postStore;
  }

  getPosts = () => {
    return this.posts;
  };

  loadPosts = async () => {
    let res = await this.postStore.getList();
    this.posts = res;
```

```
  };
}
```

In the above code example, we are focused only on managing the posts rather than how the posts are stored. The dependency for fetching posts is passed in the PostManager, and it is used without any concern about the dependency's internal working. The PostManager class only needs to know how the postStore is to be used.

The dependency injection improves the scalability and understandability of code. The code is also easy to test.

Utilize Third-party Solutions

Node.js is blessed with a large community of developers supporting it. NPM, the Node package manager, is a package manager that can be used to install all the third-party modules in Node.js easily. NPM has a lot of well-documented and maintained frameworks. Making use of all these makes development very easy, and developers can focus on the logic part more.

Some other Node.js libraries that can also help are:

- Moment (date and time)
- Nodemon (automatically restarts the app when there's a code update
- Agenda (job scheduling)
- Winston (logging)
- Grunt (automatic task runner)

There are several third-party modules available. But they should be used smartly. Overuse or relying on these modules is not good as it may affect the safety of the application. Also, they may introduce a large number of dependencies in the project.

Apply a Uniform Folder Structure

We have already discussed the layered structure. The division of code into various modules helps in making debugging and testing easy. It also promotes reusability.

Here is a basic folder structure you should follow while setting up a new application in Node.js:

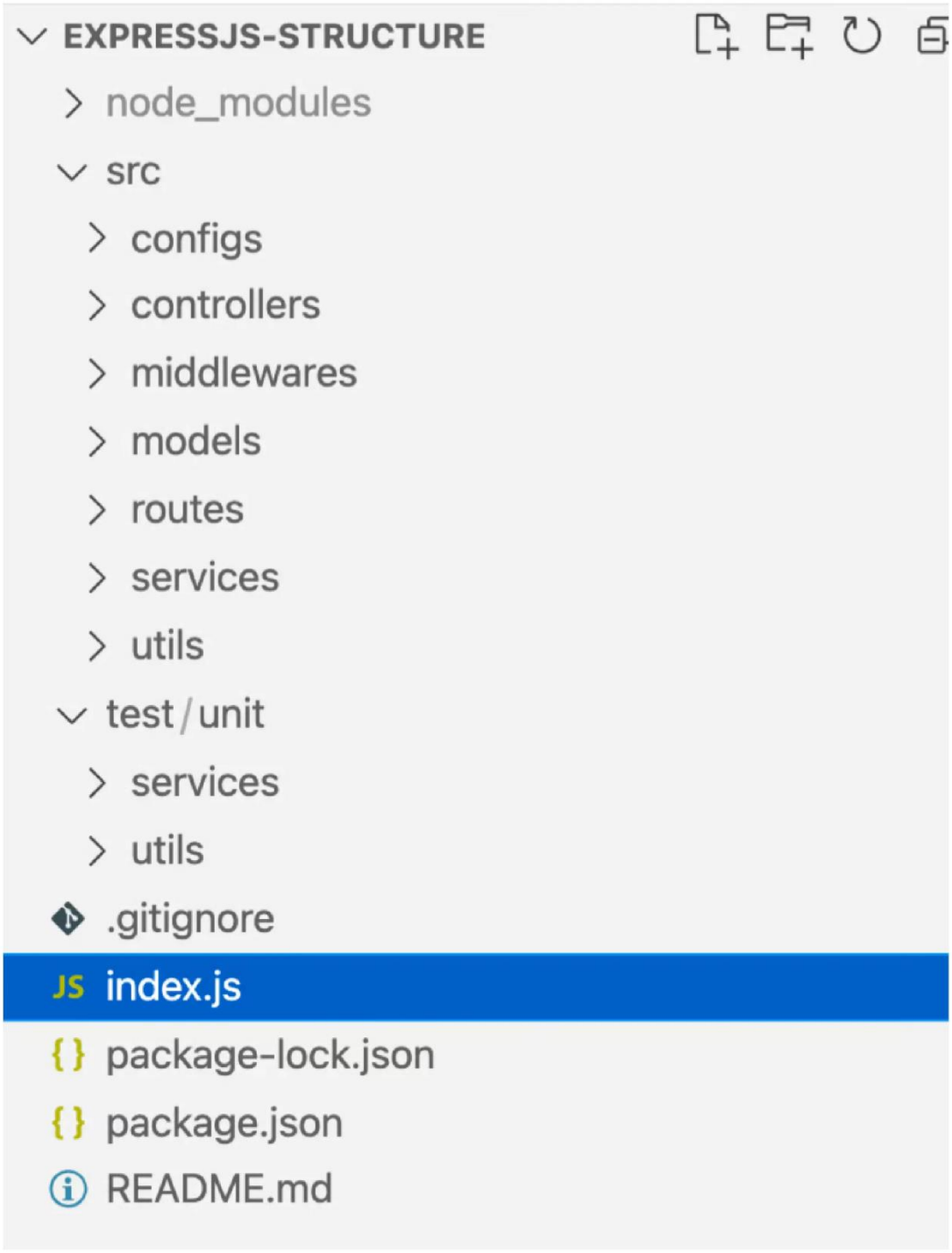

Fig 15 Express Folder Structure

The controller layer is the API directory, the service layer is the Services directory, and the data access layer is the Models directory. /Config stores the environment variables,

while /scripts store's workflow automation scripts. The /test directory contains the test cases, and /or subscribers store the event handlers in the pub/sub pattern.

Use Linters, Formatters, Style Guides, and Comments for Clean Coding

Linting and formatting: Static code analyzers that check the code for bugs, errors, and other wrong constructs are called linters. They help in the identification of bugs and other harmful patterns in our code. Some examples are Jslint and Eslint. Formatters on the other hand ensure that a consistent style is followed in the project. A prettier code formatter is an example of a formatted one. Linters and formatters are now available as plugins in many IDEs.

Style guides: Style guides help in following the naming conventions and other coding standards used by top developers. Some of the examples are style guides by Google and Airbnb.

Adding comments: To make code more readable and understandable to others we must use comments at every part of the project. They tell others what logic you have used in the current part of the code. Comments are also a good way to document the details like author, functionality, and other purposes.

Rectify Errors with Unit Testing, Logging, and Error-Handling

Unit testing: It is done to check the accuracy and correctness of one unit or part of the code. This helps reduce the debugging time and cost. It helps in verifying that the individual units are performing their tasks correctly or not. Jest, Mocha, and Jasmine are some frameworks for this purpose.

Logging and error-handling: Error is the problem or fault occurring in the program. It is shown with the error message and possible fixation. Many programming languages have built-in log generators. These logging systems are essential at every stage of the development process.

In Node.js, the most common way of logging is the use of the console.log() function that prints the information in the console.

There are three important streams checked in the logging process:

- **Stdin –** deals with input to the process (for example – keyboard or mouse)
- **Stdout –** deals with the output to the console or a separate file
- **Stderr –** deals specifically with error messages and warnings.

Handling errors is also an important part of the development of Node.js applications. Issues like callback hell must be addressed properly. A centralized error-handling component should be made to avoid duplication errors. The component should send messages to the admin, handle monitoring events and log everything. The use of try-catch should be done wherever necessary.

Using Config File and Environment Variables

When the application gets scaled, the need for global variables that can be accessed by every module arises. We can separate all the global variables in one file in the config folder. All the environment variables can be saved in a .env file.

The API keys, database passwords, and other such information can be stored this way. It gets saved as a .env file that has all the environment variables.

DB_HOST=localhost

DB_USER=root

DB_PASS=abc@123

The dotenv package can be used to import all these environment variables. If any changes are to be made, they can be done in one place and it will be reflected in the application.

Employ Gzip Compression

Whenever we have to transfer files, large files may create a problem. Hence using a file compression technique can help. Gzip is a lossless compression mechanism that can compress files so that they can be transferred quickly. You can compress the multimedia files that are served on the web pages to reduce the load and make processing faster.

Express.js helps in the easy implementation of compression techniques. In the documentation of Express.js, it is recommended to use Gzip compression.

Conclusion

- Node.js is based on a single-threaded event loop architecture.
- The main thread only handles the synchronous part of every request & assigns the asynchronous part to a background thread.
- The pub/sub model is a messaging pattern in which the components publish messages and others subscribe to them.
- The layered architecture provides reliability to the application.
- Dependency injection is a design pattern for software in which the dependencies of the software are passed in as parameters
- The division of code into various modules helps in making debugging and testing easy.
- All the environment variables can be saved in a .env file.
- Gzip is a lossless compression mechanism to compress files.

MongoDB Database Architecture

When designing a modern application, chances are that you will need a database to store data. There are many ways to architect software solutions that use a database, depending on how your application will use this data. In this article, we will cover the different types of database architecture and describe in greater detail a three-tier application architecture, which is extensively used in modern web applications.

The term "database architecture" refers to the structural design and methodology of a database system, which forms the core of a Database Management System (DBMS). This architecture dictates how data is stored, organized, and retrieved, playing a crucial role in the efficiency and effectiveness of data management.

Database architecture describes how a database management system (DBMS) will be integrated with your application. When designing a database architecture, you will make decisions that will change how your applications are created.

First, you'll decide on the type of database you would like to use. The database could be centralized or decentralized. Centralized databases store all data in a single location, often managed by one entity; decentralized databases distribute data across multiple locations or nodes. Centralized databases are typically used for regular web applications and will be the focus of this article. Decentralized databases, such as blockchain databases, might require a different architecture.

Once you've decided the type of database you want to use, you can determine the type of architecture you want to use. Typically, these are categorized into single-tier or multi-tier applications, which we'll explore in more detail below.

The architecture of a database is not a one-size-fits-all solution. It varies significantly based on the needs of the organization, the type of data being managed, and the specific applications that interact with the database. From simple structures that manage daily transactions in a small business to complex architectures that handle massive amounts of data in large enterprises, the spectrum of database architecture is broad and diverse.

At its core, a database system is an intricate set of software tools and mechanisms that store, manage, and retrieve data. A DBMS acts as an intermediary between the user and the database, ensuring that the data is easily accessible, consistently organized, and securely maintained. The effectiveness of a DBMS hinges on its underlying architecture, which must be robust, scalable, and capable of handling various data-related tasks with efficiency.

The primary functions of a DBMS include data storage, retrieval, updating, and management of database security. It should provide a convenient and efficient way to define, create, manage, and control access to the database. A well-designed DBMS architecture ensures that these functions are performed seamlessly, providing a stable and reliable platform for data management.

Types of database models

One of the fundamental aspects of database architecture is its tier architecture. This concept refers to the physical and logical separation of functionalities into different layers or tiers, such as data storage and data processing.

For instance, in a three-tier architecture, the first tier might be dedicated to raw data storage, the second tier to processing and managing data, and the third to presenting the data to users through a GUI or API.

The most common models include:

One-tier architecture

In one-tier architecture, the database, user interface, and application logic all reside on the same machine or server. It's typically used for small-scale applications where simplicity and cost-effectiveness are priorities. Because there are no network delays involved, this type of tier architecture is generally a fast way to access data.

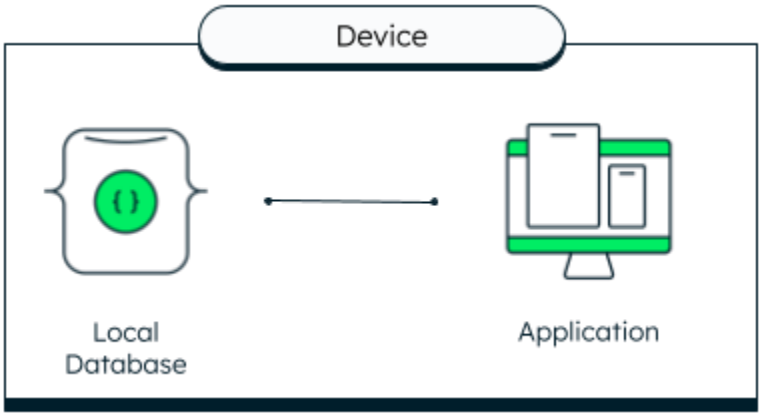

Fig 16 MongoDB One-tier Architecture

On a single-tier application, the application and database reside on the same device.

An example of a one-tier architecture would be a mobile application that uses Realm, the open-source mobile database by MongoDB, as a local database. In that case, both the application and the database are running on the user’s mobile device.

Two-tier architecture

Two-tier architecture consists of multiple clients connecting directly to the database. This tier architecture is also known as client-server architecture.

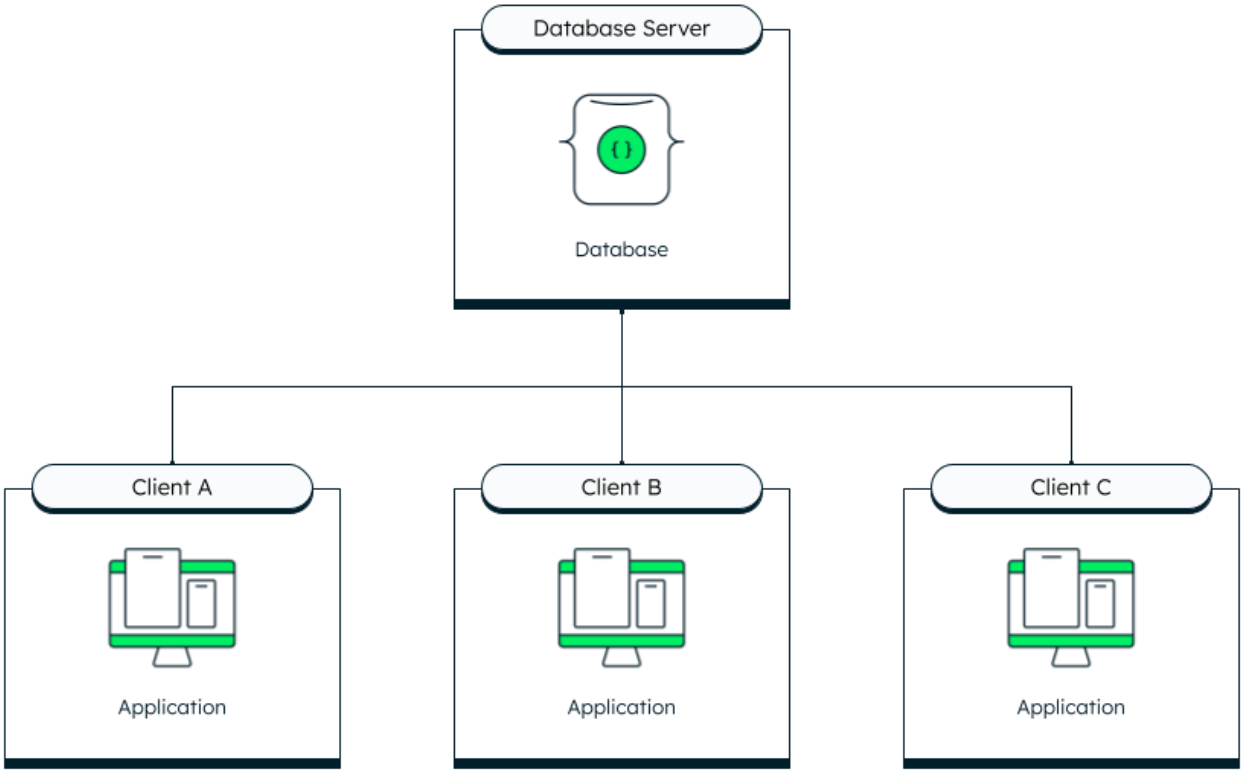

Fig 17 MongoDB Two-tier Architecture

In a two-tier architecture, clients are connecting directly to a database.

This tier architecture used to be more common when a desktop application would connect to a single database hosted on an on-premise database server — for example, an in-house customer relationship management (CRM) that connects to an Access database.

Three-tier architecture

Most modern web applications use a three-tier architecture. In this architecture, the clients connect to a back end, which in turn connects to the database. Using this approach has many benefits:

- Security: Keeping the database connection open to a single back end reduces the risks of being hacked.
- Scalability: Because each layer operates independently, it is easier to scale parts of the application.
- Faster deployment: Having multiple tiers makes it easier to have a separation of concerns and to follow cloud-native best practices, including better continuous delivery processes.

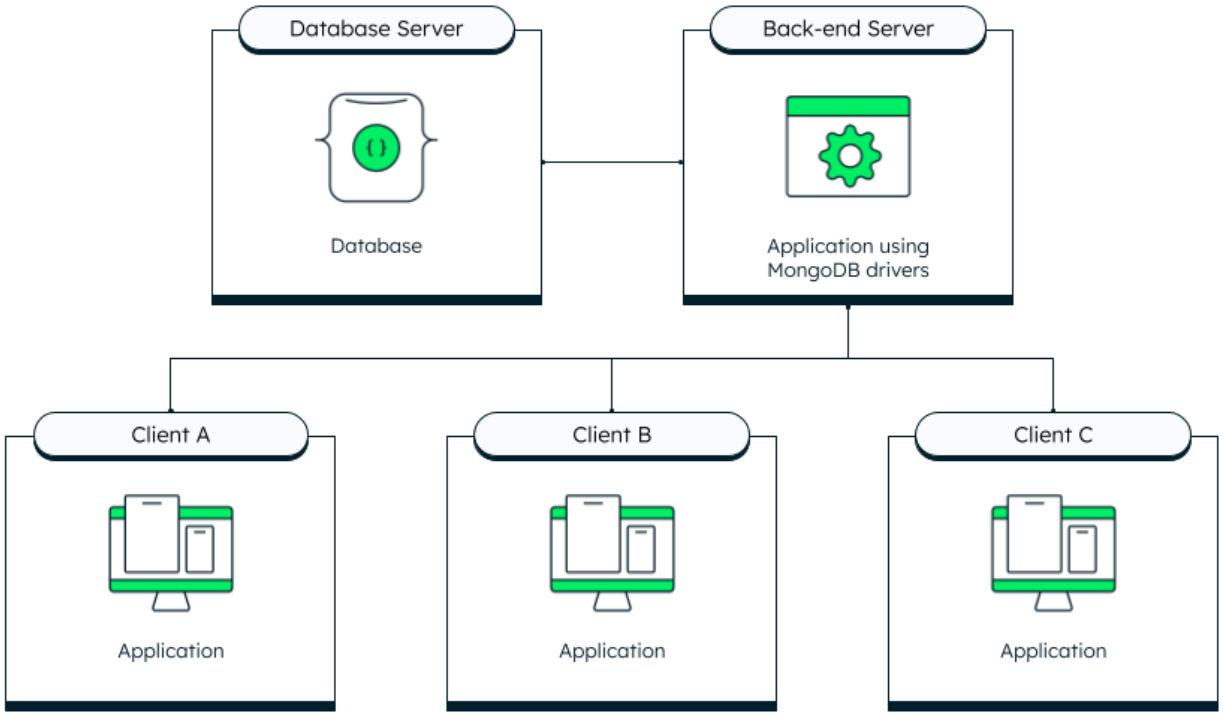

Fig 18 MongoDB Three-tier Architecture

In a three-tier architecture, the information between the database and the clients is relayed by a back-end server.

An example of this type of architecture would be a React application that connects to a Node.js back end. The Node.js back end processes the requests and fetches the necessary information from a database such as MongoDB Atlas, using the native driver. This architecture is described in greater detail in the next section.

Importance of keys in database architecture

In database architecture, keys uniquely identify records within a table (primary keys) and create essential links between other tables (foreign keys); keys form the foundation for relational data organization. Keys are an integral part of database architecture, serving as the cornerstone for establishing relationships between tables and ensuring data uniqueness. The primary key of a table is a unique identifier for each record in that table. It is essential for establishing links between tables, especially in a relational database model.

Composite keys, which are formed by combining two or more columns in a single table, can also serve as primary keys, particularly when no single column uniquely identifies a record. A foreign key, on the other hand, is used to create a link between two related tables, ensuring referential integrity and enabling the establishment of relational connections.

An example of a composite key is a “MonthlySales” table. Neither the Month or the ProductID columns individually could uniquely identify a record. But combining these two columns into a

composite key ensures each record, representing sales for a particular product in a specific month, is unique.

Client-server architecture in databases

A pivotal aspect of modern database architecture is the client-server model, particularly relevant in two-tier and three-tier architectures. In this model, the database server houses the DBMS and the actual database, handling data storage, query processing, and transaction management.

The client, usually a front-end application, interacts with the server, requesting data and presenting it to the end user. This separation enhances data integrity, security, and management efficiency, allowing for a more robust and scalable system.

Managing relationships and normal forms

Effective database design involves not just structuring data but also managing how different pieces of data relate to each other. This is where concepts like one-to-one, one-to-many, and many-to-many relationships come into play.

These relationships dictate how data in one table links to data in another, influencing how queries are constructed and how data integrity is maintained.

In database design, normal forms are sets of guidelines that reduce redundancy and improve data integrity by ensuring each table is structured properly. They include rules like each field should contain only atomic values in the first normal form, the requirement for non-key attributes to be functionally dependent on the primary keys in the second normal form, and the need for every non-key attribute to be directly dependent on the primary key in the third normal form.

These forms progressively refine table structure. Moreover, adherence to normal forms — such as the first normal form, second normal form, and third normal form — is crucial in reducing redundancy and dependency in database tables. Each normal form addresses specific types of anomalies and dependencies, contributing to a more streamlined and efficient database.

Database design process: from concept to implementation

The process of designing a database is meticulous and requires careful consideration of various factors. It begins with understanding the data needs and ends with a fully functional database system. Key steps in the database design process include:

- Requirement analysis: Understanding what data needs to be stored and how it will be used.
- Conceptual design: Creating a high-level visual representation of the database, often using entity-relationship diagrams.
- Logical design: Defining the database tables, keys, and relationships in more detail.
- Physical design: Deciding on the actual storage of data on disk, considering factors like disk space, performance, and scalability.

Database tables are fundamental components of a database structure, serving as the primary means of storing and organizing data. Each table in a database is uniquely identified by its table's primary key, which ensures that each record within the table is distinct. The primary key's role becomes even more crucial when dealing with multiple tables, as it facilitates the linking of related tables through foreign key relationships.

The database design process often begins with ensuring that tables adhere to the first normal form (1NF). This normal form stipulates that each field within a table should contain only one value, and each record must be unique. This initial step in normalization helps eliminate duplicate data, thereby improving data integrity and reducing disk space usage.

Normalization and data integrity

Normalization plays a significant role in the database design process. It is a systematic approach to organizing data that involves dividing databases into tables and defining relationships between them to minimize redundancy and dependency.

By applying normalization rules, designers can ensure that the database structure is efficient and that data is stored without unnecessary redundancy. This not only improves performance but also enhances data integrity, making the database easier to maintain and update.

Evolution of database models

The evolution of database models has been significant, from flat-file systems to hierarchical, network, and the widely used relational database models. Each model has its unique architecture, advantages, and limitations. The choice of a database model is often influenced by the specific requirements of the application, the nature of the data, and the desired performance characteristics.

Efficient data management and scalability

A core objective of any database architecture is efficient data management. This encompasses not just the speed and efficiency of data retrieval and storage but also aspects like scalability and adaptability to changing data needs. As databases grow and evolve, the architecture must be capable of scaling up or down to meet the demands of the users and the application.

Addressing redundant data and ensuring consistency

One of the challenges in database design is managing redundant data. Redundancy can lead to inconsistencies and increased storage requirements. Through techniques like normalization and proper database design, redundancy can be minimized, ensuring that data remains consistent across the database.

Database architecture in the context of emerging technologies

As technology advances, database architecture continues to evolve. The integration of cloud computing, big data analytics, and artificial intelligence in database systems is transforming how data is managed and accessed. These technologies bring new dimensions to database architecture, such as distributed data storage, real-time analytics, and predictive modeling, which require innovative architectural approaches.

The three levels of database architecture in MongoDB Atlas

The most common DBMS architecture used in modern application development is the three-tier model. Since it's so popular, let's look at what this architecture looks like with MongoDB Atlas.

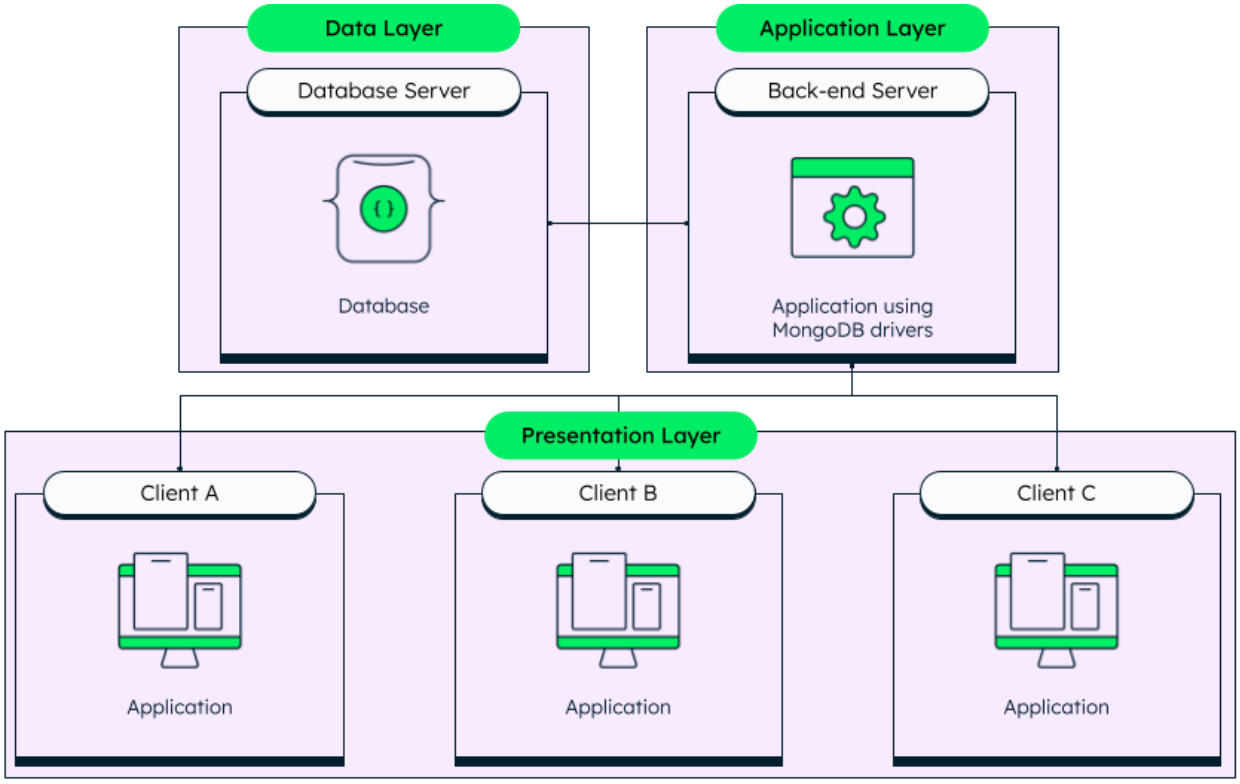

Fig 19 MongoDB three levels database Architecture

A three-tier application is composed of three layers: the data, the application, and the presentation.

As you can see in this diagram, the three-tier architecture comprises the data, application, and presentation levels.

Data (database) layer

As the name suggests, the data layer is where the data resides. In the scenario above, the data is stored in a MongoDB Atlas database hosted on any public cloud — or across multiple clouds, if needed. The only responsibility of this layer is to keep the data accessible for the application layer and run the queries efficiently.

Application (middle) layer

The application tier is in charge of communicating with the database. To ensure secure access to the data, requests are initiated from this tier. In a modern web application, this would be your API. A back-end application built with Node.js (or any other programming language with a native driver) makes requests to the database and relays the information back to the clients.

Presentation (user) layer

The final layer is the presentation layer. This is usually the UI of the application with which the users will interact. In the case of a MERN or MEAN stack application, this would be the JavaScript front end built with React or Angular.

Database architecture: a summary

In this article, you've learned about the different types of database architecture. A three-tier architecture is your go-to solution for most modern web applications. However, there are other topologies that you might want to explore. For example, the type of database you use could be a dedicated or a serverless instance, depending on your predicted usage model. You could also supplement your database with data lakes or even online archiving to make the best use of your hardware resources. If you are ready to concretize your database architecture, why not try MongoDB Atlas, the database-as-a-service solution from MongoDB? Using the realm-web SDK, you can even host all three tiers of your web application on MongoDB Atlas

Express.js

Express.js Architecture Diagram

To illustrate the architecture of Express.js, we can break it down into several key components that work together to handle HTTP requests and responses. Below is a detailed description of each component along with their interactions:

1. Client

- The client initiates an HTTP request to the server. This could be a web browser, mobile app, or any other client capable of making HTTP requests.

2. Server

- The server receives the incoming request from the client. In this case, it is running an instance of an Express.js application.

3. Middleware

- Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.
- Middleware can perform various tasks such as logging requests, parsing request bodies, handling authentication, and more.
- They can be defined globally for all routes or specifically for certain routes.

4. Routes

- Routes define how an application responds to specific HTTP requests made to a particular endpoint (URL path) using specific HTTP methods (GET, POST, PUT, DELETE).
- Each route is associated with a handler function that processes the request and sends back a response.

5. Controllers

- Controllers contain the logic for handling requests related to specific resources.
- They interact with models to retrieve or manipulate data and return responses back to the client.

6. Models

- Models represent data structures in your application and are often used in conjunction with databases.

- They define how data is stored and retrieved from databases like MongoDB or SQL databases.

7. Database

- The database stores persistent data used by your application.
- It can be any type of database system such as relational databases (MySQL, PostgreSQL) or NoSQL databases (MongoDB).

8. Response

- After processing the request through middleware and routing logic, a response is sent back to the client.
- The response may include HTML content, JSON data, status codes indicating success or failure, etc.

The interaction between these components can be visualized in a flow diagram where:

1. The **Client** sends an HTTP request to the **Server**.
2. The **Server** processes this request through various **Middleware**, which may modify the request/response objects or terminate the cycle.
3. If not terminated by middleware, it reaches defined **Routes**, which call appropriate **Controllers** based on URL patterns and methods.
4. Controllers interact with **Models**, which communicate with the **Database** to fetch or store data.
5. Finally, a **Response** is generated and sent back through middleware (if applicable) before reaching back to the **Client**.

This architecture allows for modular development where different parts of an application can be developed independently while still working together seamlessly.

Top 3 Authoritative Sources Used

1. Express Documentation

- The official documentation provides comprehensive information about Express.js features including routing, middleware usage, and best practices for building applications.

2. MDN Web Docs - Express Tutorial

- Mozilla Developer Network offers tutorials on web technologies including Express.js which cover fundamental concepts like routing and middleware in detail.

3. Node.js Design Patterns Book

- This book discusses design patterns in Node.js applications including those used in Express.js applications for structuring code effectively and managing asynchronous operations efficiently.

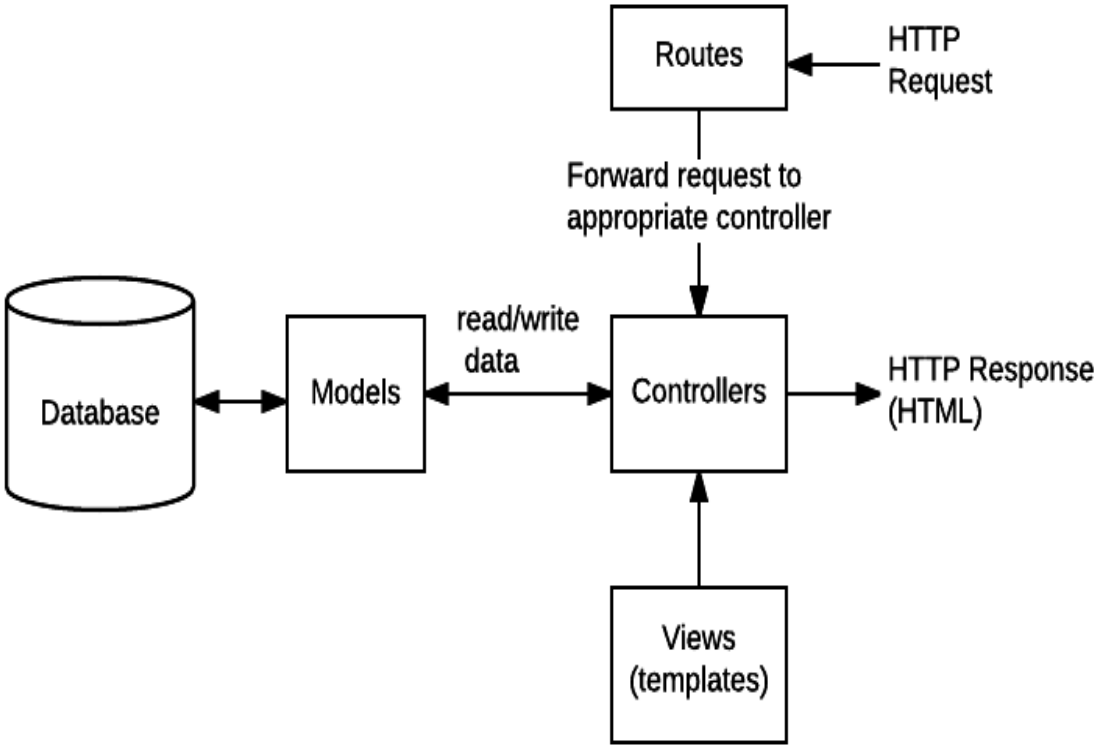

Fig 20 Express js Managing Asynchronous Operation

The overall architecture

I want to introduce you to the architecture and layout how each connection is made when we talk about each part.

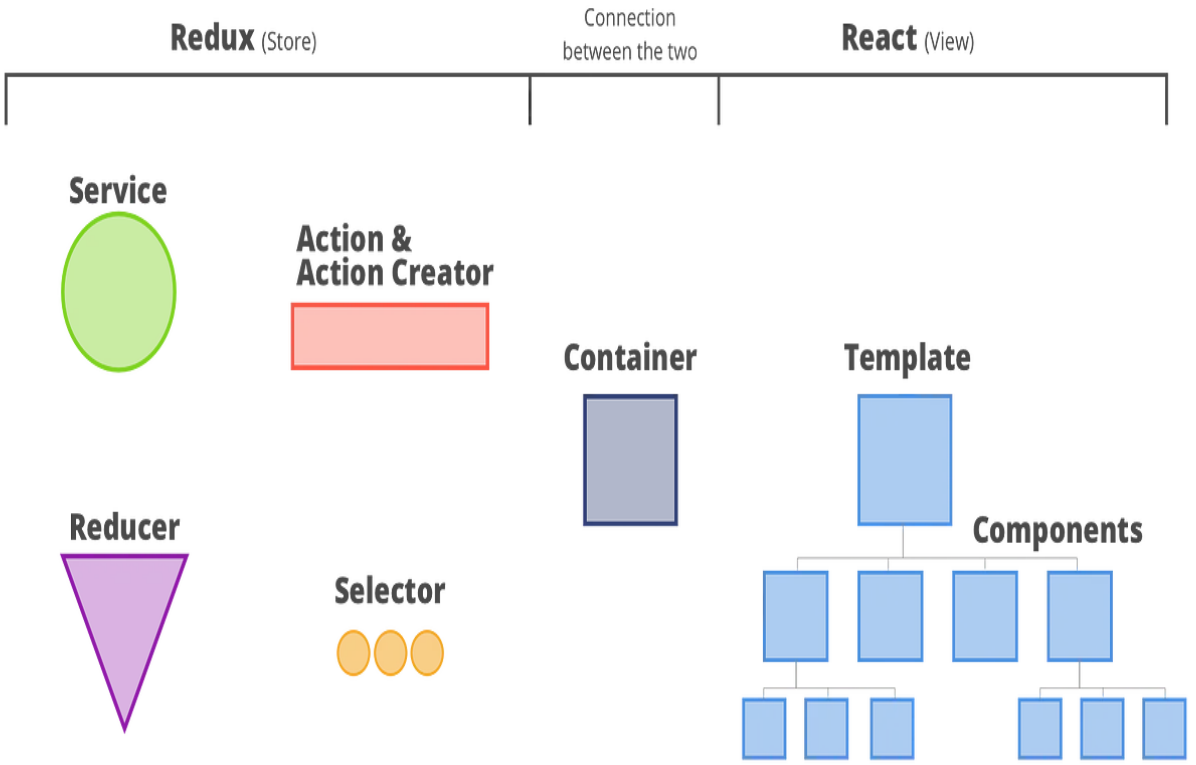

Fig 21 Architecture for React and Redux

There are a lot of parts there, but each plays a very specific purpose. These purposes will be explained as we get into each part. At this time, it is important to know that Services, Actions & Action Creators, Reducers, and Selectors are redux. Templates and Components are react. The Container is piece that allows them to be loosely coupled together. That is really important because it allows the store and the view to be able to change and grow independently of each other.

I am going to keep building on that diagram.

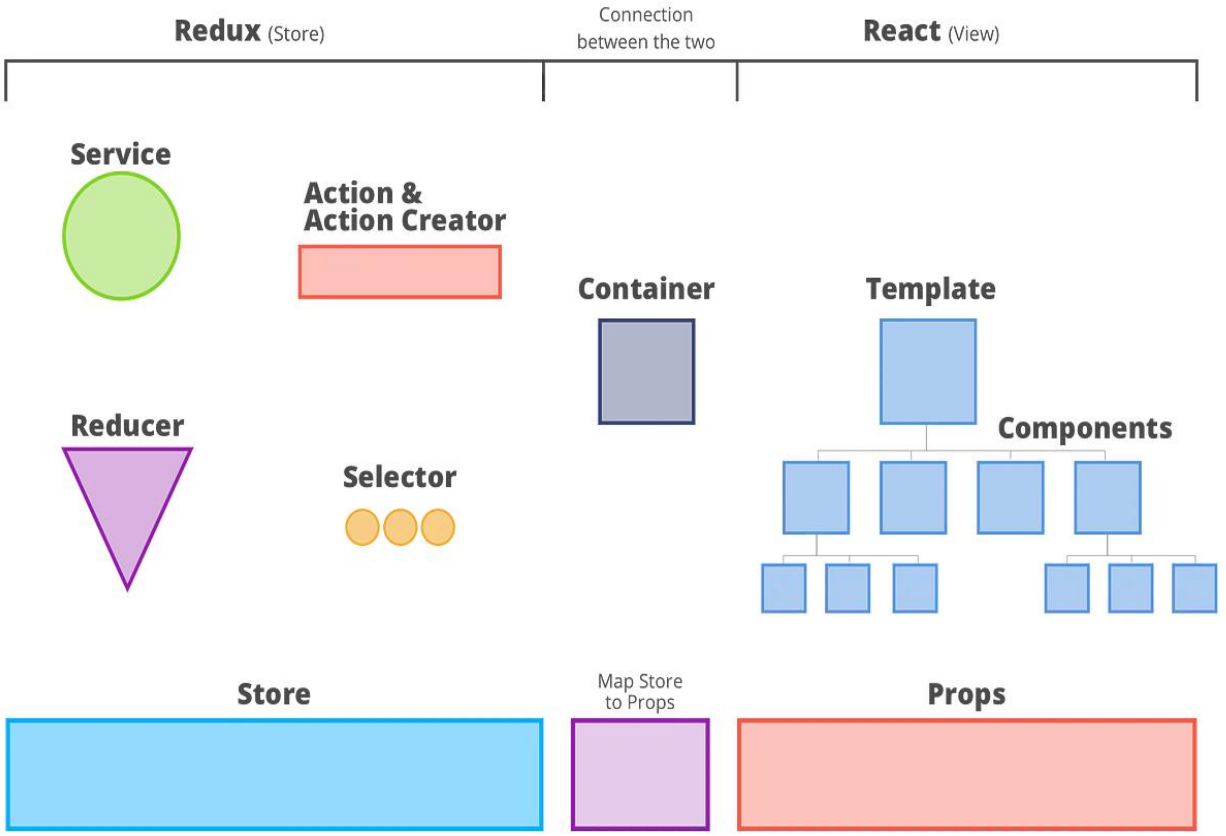

Fig 22 Architecture for React and Redux adding state

When I talk about state what I am talking about is all the data that is needed for the app to be in a particular…well state. State being the condition of a person or thing, as with respect to circumstances or attributes. This state is in the store for redux, and in the props for react. When the store is updated it re-maps through the container and re-renders the DOM. When there is an event in the DOM that triggers events through the container and to the Actions. The biggest thing to know here is that the store is where redux stores the data, the data is mapped to props, where React displays through components. And all that is state.

So, let's map the relationships.

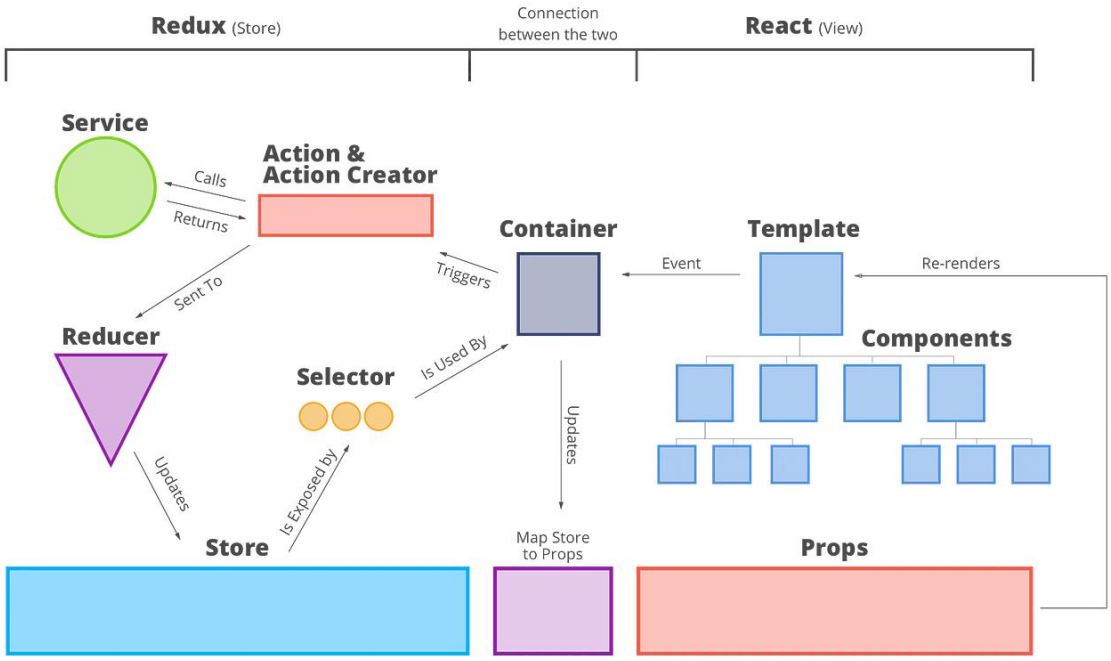

Fig 23 React and Redux Mapping of connections between pieces

Parts and Connections

Component

The component is rendered with a set of props that get passed down from its parent component. In this case it is a template, but let's not fool ourselves, a template, a component, and even a container is just a component. It is a set of code that uses properties and can call other components and gives them properties.

A component really starts with an API. What data do I need in order to render something. For an example let's think about a tile, a very root level component used in a lot of places and is almost on every website.

So, let's define a tile. It needs an image, a title, a teaser description, a link, and sometimes it has some metadata.

```
import {Component as ReactComponent} from 'react';class Component extends ReactComponent {
  render(){
    const{image,title,teaser,link,meta}=this.props;return(
      <div>
```

```
            <imgsrc={image}alt=""/>
            <span>{title}</span>
            <p>{teaser}</p>
            <p>{meta}</p>
          </div>
      )
   }
}
export default Component;
```

The thing to notice most here is the props. Those props get mapped to HTML markup using JSX. This isn't about creating components so I am not going to get to detailed in this article.

So, if I was to use this it would look like this:

```
<Component title={title} meta={meta} image={image} link={link} teaser={teaser} />
    //or you could use the object spread operator
    <Component {...props} />
```

So, we can see that the component gets used by passing properties down to it. Then the components react to the properties coming down. We edit data that then edits the DOM. The other part of that is that data never goes up, instead an event is triggered and sent through the Container to an action to start the cycle.

Template

I am not going into much detail here as it is simply a component. But I do want to mention it is a special kind of component that does things more specifically than the lower components do. I refer to this as adding opinions to the components. A component may allow any title, but the template may only pass down a single title for anything that is there. That would be an opinion.

Container

The container is the glue that connects react to redux. It connects it in a lot of ways. This is where the react-redux module is used and I usually call that connect as that is what I use it for. It takes three arguments, an object that maps state to props — mapStateToProps, an object that maps actions to dispatch (these are used to wire events to actions) — mapActionsToDispatch, and merge Properties merges all the properties together and passes them to react for rendering.

Actions & Action Creators

I put these together because they are kind of referred to as each other. Often when people are referring to actions, they really mean action creators. What is the difference?

Action: This is an object that contains the type of action and the state that was changed because of the action.

Action Creator: This is the code that is called to create an action and send it along to the reducer.

I kind of think of the Action as an event for redux. When an event is fired there is an event type (like on Click, on Hover, etc.) and has an event object that contains the data from the event. Well, an action is kind of the same, it has a type and the data.

Let's look at how Containers and Actions play together. This is the container file:

```
//Step 1: Pull in the function for the action.
import { getArticles } from '../store/articleList/actions.js';//Step 2: This connects the action,
getArticles, to dispatch the action.
const mapActionsToDispatch = (dispatch) => ({
   loadArticles: () => dispatch( getArticles() )
});//Step 3: This takes the connected action and merges it to a prop for React.
constmergeProps=(state, actions)=>({
   ...state,
   ...actions,
   loadArticles:(=>actions.loadArticles()
});//Step 4: Merging it to a prop makes it available in the component,
// in this case on the componentDidMount life cycle event.
// This could just as easily be a click, submit, type, or other type of event.
class LandingContainer extends ReactComponent {
   componentDidMount() {
      const {loadArticles} = this.props;
      loadArticles();
   }
   render() {
      return <Landing {...this.props} />
   }
}
```

If you follow the structure, you can see how an action gets mapped to a property in a component. The file above was rearranged a little so you could see the flow more easily.

Now let's look at the action:

```
// This is the connection to the Reducer. This is where the action type is connected.
import {ADD_ARTICLE, CLEAR_ARTICLES} from './reducers.js';
// This connects redux to the API.
import getArticleList from './services.js'; /* Sync Action Creators */
//By default all actions are synchronous. But we need to call an outside API.
```

```
// Here is where the action is created, a type, with a set of data.
export const addArticle = (article = []) => ({
    type: ADD_ARTICLE,
    article: article
});
// This is also a creator, a simple one. All it will do is remove whatever state is in the store for articles.
export const clearArticles = () => ({type: CLEAR_ARTICLES});
/* Async Action Creators */
// Here is the async creator. This is using Thunk that allows us to pass a function as the data in an action.
// it gets run on the other side. Here we calling our service, getArticleList, that returns a promise.
// then we dispatch the action when the promise is resolved.
export const getArticles = () => (dispatch) => {
    getArticleList ()
  . then((articles) => {
        articles.forEach((article) => {
            dispatch(addArticle(article))
        })
    })
};
```

That is how the action is connected to a service, connected to the container, and gets data. But how it sends the data on… for that we need to move to the reducer.

Reducer

The key thing to know about a reducer is that for every dispatch every reducer is called and given the dispatched action. Then it is up to the reducer to handle it or pass it on.

```
// This is declaring the types of actions.
export const ADD_ARTICLE = `ADD_ARTICLES`;
export const CLEAR_ARTICLES = `CLEAR_ARTICLES`;// This will be called for every dispatch. It is passed the type of action.
export default function(state = [], {type, article } ) {
    // Here I ignore or handle each action type.
    switch(type) {
        case CLEAR_ARTICLES:
            return [];

        case ADD_ARTICLE:
            return [
                ...state,
                article
            ];
        // if I don't care about the action I just pass along the state that was given.
        default:
            return state;
```

```
    }
}
```

The other part of the reducer that is really important to know is that you don't manipulate state. You create new state. So rather than doing something like this state. Property = "some new value" instead you would do return [...state, property]. State then goes into your store.

Selector

The selector is how you would get data out of your store in the container. As I show the code it may not seem very valuable.

```
//Selector.js
export default (state) => (state. property)//Container.js
import selectArticles from '../store/articleList/selectors.js'; const mapStateToProps = (state) => {
    return {
        // React is looking for the tiles property in the template, here we map articles to tiles.
        tiles: selectArticles(state)
    }
}
```

ReactJS

React Architecture

React architecture is a collection of components responsible for building a software's User Interface (UI). You can also see it as an organization of your codebase that helps you build your unique project. For example, a React architecture will include several UI components such as buttons, forms, API services, Central State Management services, etc.

Because of this ReactJs architectural freedom, it is the most preferred JavaScript library (framework) for building front-end applications.

The following are some of the ways React architecture helps in web development:

1. React assists you in developing a component-based software architecture that simplifies maintenance and code reuse.
2. It enables you to maintain a global state variable by utilizing state management libraries such as Redux.
3. Because React architecture allows you to build out components, code expansion becomes much easier as your project grows.
4. Since it is component-based, the React architecture makes unit testing much easier.

Every React project has an architectural representation that is tailored to the project's purpose and dependencies. A React Web3 project, for example, will not have the same architecture as a React PWA project. However, when getting started with React, we recommend following the structure below which includes:

- Project Directory
- src
- Assets
- Components
- Services
- Store
- Utils
- Views

1. Navigating a Directory Structure

Using the example below, a React project has a source (src) folder where all the essential files and folders are listed:

```
└── /src
  ├── /assets
  ├── /components
  ├── /views
  ├── /services
  ├── /utils
  ├── /hooks
  ├── /store
  └── App.js
  ├── index.js
  ├── index.css
```

- The assets folder contains all of the project's static files, such as your logo, fonts, images, and favicons.
- Components folder contains a bit collection of UI codes such as buttons, forms, avatars, and so on.
- The views folder contains all your React application's web pages.
- Services folder contains code that allows you to interact with external API resources.
- The utils folder contains reusable function snippets for performing quick tasks like text truncation or down casing.
- Hooks folder contains codes and logic that can be reused across multiple components.
- The store folder houses your state management files, such as Redux, which are used to make certain functions and variables available throughout your application.
- The main component of your React application is the App.js file. This file connects all components and views.
- Index.js file is the React application's entry point. It is responsible for bootstrapping the React library and mounting it on the root element.
- index.css is the main and global CSS file for our application. Any writing style to this file will apply throughout the project.

2. Common Modules

Because React is a non-opinionated framework, it doesn't care how you divide your modules when building your application. But what exactly are modules? They are bundled reusable sets of functions, classes, and components that aid in the software development process.

React assists you in reducing complexity and creating open, reusable, and shared structures across your application. Common modules include reusable custom components, custom hooks, business logic, constants, and utility functions.

These modules, which are shared throughout your software, can be used in a variety of components, views, and even projects.

3. Add Custom Components in Folders

Below is an example of a React custom input component. In your components directory, create a new folder called Input. Now, within this input folder create the files represented below:

```
└── /src
  ├── /components
| ├── /Input
| | ├── Input.js
| | ├── Input.css
| | ├── Input.test.js
```

- Input.js is the JavaScript file for writing all the logic for this component.
- The Input.css contains all the styles for this Input component.
- Input.test.js contains all the test cases for this component.

Next, you will then have to create an index.js file in the components directory to bundle up the components. See the code below for example:

```
// /src/components/index.js

import { Button } from './Button/Button';

import { Input } from './Input/Input';

import { Card } from './Card/Card';

export { Button, Card, Input };
```

With this file set up, you can then use it on any view or pages in your project.

4. Create Custom Hooks

A custom Hook is a function that begins with "use" and may invoke other Hooks. Understanding this definition will help you use custom hooks effectively. Creating a custom hook is essential for reducing code complexity.

For example, let's say that you have two distinct pages in your React app called a Login and Registration page. Each of these pages has input fields where visitors may enter their information and submit at the click of a button. If you want a password visibility toggling feature as part of the password field in both the pages, you may have to duplicate your code.

To reduce duplications and complexities from your codebase, writing a custom hook that toggles the password's visibility is the right approach. Here's an example of a custom hook:

Create a hook as described in the code snippet below:

```
└── /src
  ├── /hooks
    ├──usePasswordToggler.js
```

The below custom hooks are used to create a state of a hook with three objects it returns:

- type: This returns a form type, either text or password.
- passwordVisibility: This returns a boolean value indicating whether the password is hidden or visible.
- handlePasswordVisibility: This is a function that toggles the above state variables of type and passwordVisibility either to show or hide

```
// ./src/hooks/usePasswordToggler.js

import {useState} from 'react';

export const usePasswordToggler = () => {

 const [passwordVisibility, setPasswordVisibility] = useState(true);

 const [type, setType] = useState('password');

 const handlePasswordVisibility = () => {

  if (type === 'password') {

   setType('text');

   setPasswordVisibility(!passwordVisibility);

  } else if (type === 'text') {
```

```
    setType('password');
    setPasswordVisibility(!passwordVisibility);
   }
  };
  return {
   type,
   passwordVisibility,
   handlePasswordVisibility
  };
};
```

Next, we can use this hook in your regular React component, see the example below:

```
import React from 'react';
import { usePasswordToggler } from './hooks/usePasswordToggler';
import './App.css';
function App() {
  const { type, passwordVisibility, handlePasswordVisibility } = usePasswordToggler()
  return (
   <main>
    <div>
     <input type={type} placeholder='Enter password...' />
          <button onClick={handlePasswordVisibility}>{passwordVisibility ?  'Show'  : 'Hide'} Password</button>
    </div>
   </main>
  );
}
export default App;
```

The above codes will produce the following outputs:

Redux

Redux is a small standalone JS library. However, it is commonly used with several other packages:

Redux Toolkit

Redux Toolkit is our recommended approach for writing Redux logic. It contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

React-Redux

Redux can integrate with any UI framework, and is most frequently used with React. React-Redux is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.

Redux DevTools Extension

The Redux DevTools Extension shows a history of the changes to the state in your Redux store over time. This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".

Redux Basics

Now that you know what Redux is, let's briefly look at the pieces that make up a Redux app and how it works.

The Redux Store

The center of every Redux application is the store. A "store" is a container that holds your application's global state.

A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object:

- You must never directly modify or change the state that is kept inside the Redux store
- Instead, the only way to cause an update to the state is to create a plain action object that describes "something that happened in the application", and then dispatch the action to the store to tell it what happened.
- When an action is dispatched, the store runs the root reducer function, and lets it calculate the new state based on the old state and the action
- Finally, the store notifies subscribers that the state has been updated so the UI can be updated with the new data.

State, Actions, and Reducers

We start by defining an initial state value to describe the application:

```
// Define an initial state value for the app
const initialState = {
  value: 0
}
```

For this app, we're going to track a single number with the current value of our counter.

Redux apps normally have a JS object as the root piece of the state, with other values inside that object.

Then, we define a reducer function. The reducer receives two arguments, the current state and an action object describing what happened. When the Redux app starts up, we don't have any state yet, so we provide the initialState as the default value for this reducer:

```
// Create a "reducer" function that determines what the new state
// should be when something happens in the app
function counterReducer(state = initialState, action) {
  // Reducers usually look at the type of action that happened
  // to decide how to update the state
  switch (action.type) {
    case 'counter/incremented':
      return { ...state, value: state.value + 1 }
    case 'counter/decremented':
      return { ...state, value: state.value - 1 }
    default:
      // If the reducer doesn't care about this action type,
      // return the existing state unchanged
      return state
  }
}
```

Action objects always have a type field, which is a string you provide that acts as a unique name for the action. The type should be a readable name so that anyone who looks at this code understands what it means. In this case, we use the word 'counter' as the first half of our action type, and the second half is a description of "what happened". In this case, our 'counter' was 'incremented', so we write the action type as 'counter/incremented'.

Based on the type of the action, we either need to return a brand-new object to be the new state result, or return the existing state object if nothing should change. Note that we update the state *immutably* by copying the existing state and updating the copy, instead of modifying the original object directly.

Store

Now that we have a reducer function, we can create a **store** instance by calling the Redux library createStore API.

```
// Create a new Redux store with the `createStore` function,
// and use the `counterReducer` for the update logic
const store = Redux.createStore(counterReducer)
```

We pass the reducer function to createStore, which uses the reducer function to generate the initial state, and to calculate any future updates.

UI

In any application, the user interface will show existing state on screen. When a user does something, the app will update its data and then redraw the UI with those values.

```
// Our "user interface" is some text in a single HTML element
ConstvalueEl=document.getElementById('value')
// Whenever the store state changes, update the UI by
// reading the latest store state and showing new data
functionrender(){
  conststate=store.getState().valueEl.innerHTML=state.value.toString()
}

// Update the UI with the initial data
render()
// And subscribe to redraw whenever the data changes in the future
store.subscribe(render)
```

In this small example, we're only using some basic HTML elements as our UI, with a single showing the current value.

So, we write a function that knows how to get the latest state from the Redux store using the store.getState() method, then takes that value and updates the UI to show it.

The Redux store lets us call store.subscribe() and pass a subscriber callback function that will be called every time the store is updated. So, we can pass our render function as the subscriber, and know that each time the store updates, we can update the UI with the latest value.

Redux itself is a standalone library that can be used anywhere. This also means that it can be used with any UI layer.

Dispatching Actions

Finally, we need to respond to user input by creating action objects that describe what happened, and dispatching them to the store. When we call store. dispatch(action), the store runs the reducer, calculates the updated state, and runs the subscribers to update the UI.

```
// Handle user inputs by "dispatching" action objects,
// which should describe "what happened" in the app


document.getElementById('increment').addEventListener('click',function(){
  store.dispatch({type:'counter/incremented'})
})

document.getElementById('decrement').addEventListener('click',function(){
  store.dispatch({type:'counter/decremented'})
})

document.getElementById('incrementIfOdd').addEventListener('click',function(){
    // We can write logic to decide what to do based on the state
    if(store.getState().value%2!==0){
      store.dispatch({type:'counter/incremented'})
    }
  })

document.getElementById('incrementAsync')
  .addEventListener('click',function(){
    // We can also write async logic that interacts with the store
    setTimeout(function(){
      store.dispatch({type:'counter/incremented'})},1000)})
```

Here, we'll dispatch the actions that will make the reducer add 1 or subtract 1 from the current counter value.

We can also write code that only dispatches an action if a certain condition is true, or write some async code that dispatches an action after a delay.

Data Flow

We can summarize the flow of data through a Redux app with this diagram. It represents how:

- actions are dispatched in response to a user interaction like a click
- the store runs the reducer function to calculate a new state
- the UI reads the new state to display the new values

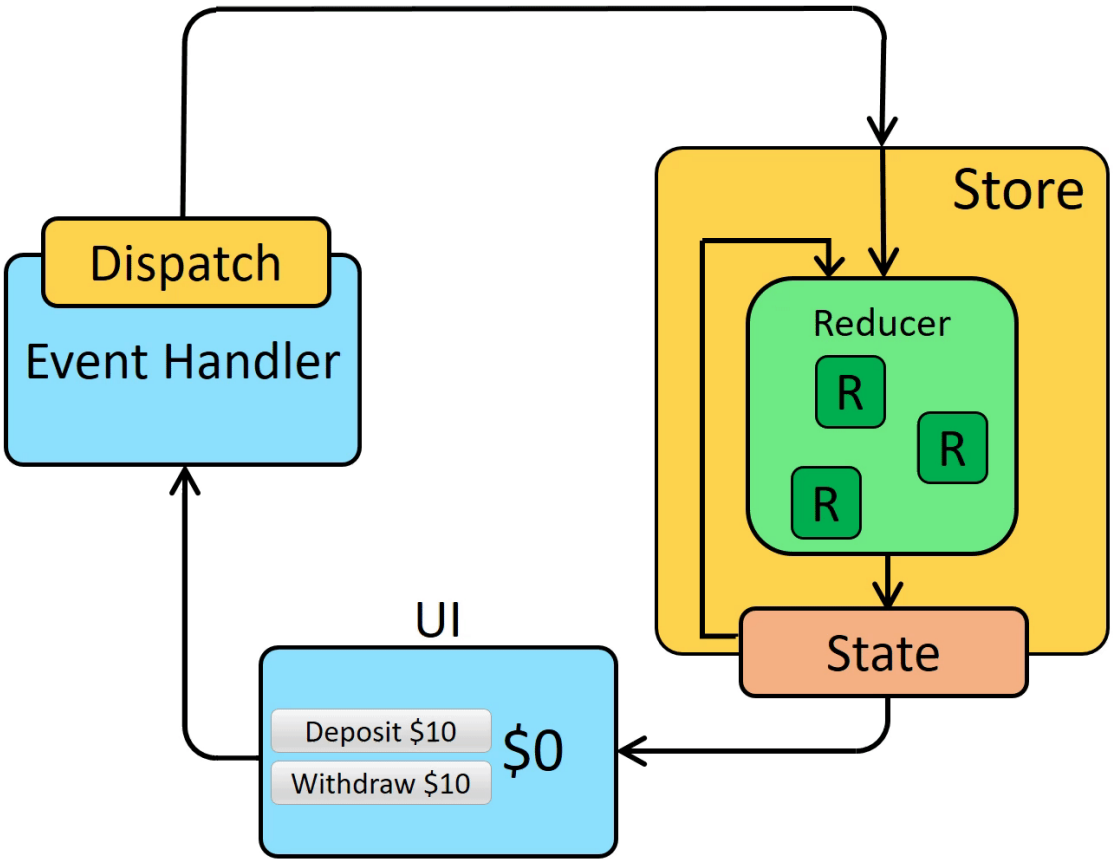

Fig 24 Flow of data through Redux App

Conclusion

The BookMyShow project has provided valuable insights into the MERN stack, which encompasses MongoDB, Express.js, React.js, and Node.js, along with essential technologies like HTML, CSS, JavaScript, Bootstrap, and Redux. Key takeaways include the effective management of server-client interactions, the implementation of state management with Redux, and the use of MongoDB for efficient data storage.

These technologies have practical applications in developing dynamic web applications, particularly in the entertainment industry, where user engagement and real-time updates are crucial. The project highlights the significance of creating intuitive user interfaces and robust backend services to enhance user experiences.

However, there are limitations associated with these technologies. For instance, the learning curve for MERN can be steep for beginners, and performance may degrade with large-scale applications if not optimized properly. Cost implications can arise from hosting and database services, especially when scaling the application.

To improve, focusing on optimizing performance through code splitting and lazy loading can enhance user experience. Additionally, incorporating testing frameworks can ensure application reliability and maintainability. Overall, the project illustrates the powerful capabilities of the MERN stack in building modern web applications while acknowledging areas for growth and improvement.

References

MongoDB: https://www.mongodb.com/resources/basics/databases/database-architecture , Sep 1 2024, official website
Express.js: https://expressjs.com/en/5x/api.html Aug 26, official website
React.js: https://legacy.reactjs.org/docs/getting-started.html , Sep 22, 2024**,** official web site
Redux js: https://redux.js.org/tutorials/index , Sep 20, 2024**,** official website
Node.js: https://nodejs.org/en , Aug 25, official website