Innovating Web Development: Leveraging Next.js for Direct Database Access and Eliminating the Backend Layer

Simran Prasad[1], Ms. Rauki Yadav[2]
[1]*Department of Computer Engineering, Bhagwan Mahavir College of Engineering and Technology Surat, Gujarat, India*
[2]*Assistant Professor, Department of Computer Engineering, Bhagwan Mahavir College of Engineering and Technology Surat, Gujarat, India*

***Abstract*—Traditional web development architectures rely heavily on dedicated backend servers to manage business logic, database interactions, and API services. While effective, this layered approach introduces additional complexity, increased de- velopment time, and higher infrastructure overhead—especially for small to medium-sized applications. With the advent of modern full-stack frameworks like Next.js, a paradigm shift is emerging: enabling frontend developers to build applications that directly interact with databases through server-side capabilities. This research investigates the potential of leveraging Next.js as a backendless architecture platform by enabling secure and efficient direct database communication through API routes and server-side rendering (SSR). The study evaluates architectural design patterns, implementation strategies, and performance metrics associated with this approach. A prototype application is developed to benchmark and compare response time, scalability, and development complexity against traditional multi-layer ar- chitectures. Furthermore, the paper discusses key considerations including authentication, data security, and deployment practices**

when eliminating the dedicated backend layer.

By consolidating backend and frontend functionalities within a unified framework, this approach demonstrates promising improvements in performance, maintainability, and deployment efficiency. The findings suggest that backendless architectures using Next.js can offer a viable solution for lightweight appli- cations, redefining conventional web development workflows and reducing the barrier to full-stack development.

***Index Terms*—Next.js, Backendless Architecture, Direct Database Access, Full-stack Web Development, Server-side Ren- dering, Web Performance Optimization**

I. INTRODUCTION

A. Background

Web development traditionally follows a client-server model, requiring separate backend infrastructure for handling logic and data While robust, it adds development and mainte- nance overhead.

The rise of Single-Page Applications (SPAs) using frame- works like React shifted much of the logic to the client-side, improving interactivity but still depending on backend APIs.

The MERN stack (MongoDB, Express.js, React, Node.js) streamlined full-stack JavaScript development, but managing both frontend and backend can still be complex.

Modern frameworks like Next.js now offer built-in server- side rendering (SSR), static site generation (SSG), and API routes, enabling backendless or simplified backend architec- tures [1].

B. Motivation

Using Next.js with direct database access (e.g., MongoDB) simplifies workflows, improves load times and SEO, and is well-suited for small to medium-sized applications. It reduces development costs, enhances productivity, and supports faster iteration cycles [2].

C. Problem Statement

Despite its benefits, backendless architecture raises concerns around security, scalability, and performance. Direct database access from the frontend may expose vulnerabilities. This research investigates the feasibility and trade-offs of using Next.js for direct database communication.

D. Objectives

- Assess backendless architecture using Next.js and Mon- goDB.
- Build a working prototype with direct database access.
- Benchmark performance against traditional architectures.
- Identify security risks and suggest best practices.
- Recommend its use for small to mid-scale applications.

E. Scope of Research

This research focuses on small to medium-scale applications using MongoDB for its NoSQL and JavaScript-friendly fea- tures. Enterprise-level systems with complex backend needs are beyond the scope.

II. LITERATURE REVIEW

The evolving landscape of web development has prompted extensive research into selecting the most suitable technolo- gies for various project requirements. This chapter reviews a selection of studies that explore frontend and backend tech- nologies, rendering strategies, and backendless architectures, offering insights into the strengths and trade-offs of different approaches.

Frontend Framework Performance: An empirical eval- uation of frontend frameworks (React, Angular, and Vue) concluded that React offers better performance and smaller bundle sizes, especially beneficial for dynamic and interactive user interfaces [3].

Backend Frameworks for RESTful APIs: Research into backend frameworks (Node.js, Django, and Spring Boot) for building REST APIs highlighted that Node.js provides a lightweight, fast, and scalable environment, especially effec- tive in real-time applications [4].

Rendering Techniques in Web Applications: Studies investigating Server-Side Rendering (SSR) versus Client-Side Rendering (CSR) revealed that SSR improves SEO and ini- tial load performance, while CSR is more suitable for rich, interactive web applications [5].

Emergence of Backendless Architectures: A compre- hensive analysis of Backend-as-a-Service (BaaS) platforms like Firebase discussed how backendless architectures reduce development time by eliminating server management, although they introduce trade-offs in flexibility and control [6].

Performance Comparison of Full Stack Frameworks: Comparative performance assessments of MERN (MongoDB, Express.js, React, Node.js) and MEAN (MongoDB, Ex- press.js, Angular, Node.js) stacks showed that MERN provides more flexibility and performance benefits in dynamic content applications [7].

Comparative Analysis of Web Frameworks: A study compared web development frameworks such as Angular, React, and Vue, emphasizing that React stands out for its flexibility and component reusability, making it suitable for scalable applications [8]. The choice between these frame- works often hinges on project-specific needs like performance and learning curve.

Security Considerations in Web Development Stacks: Security-focused evaluations of web development stacks re- vealed that framework like Next.js provide built-in protections against cross-site scripting (XSS) and improve secure defaults for modern web applications [9].

SEO Benefits of Server-Side Rendering with Next.js: Research into SEO effectiveness in modern web applications demonstrated that Next.js enhances crawlability and perfor- mance metrics compared to client-rendered React apps [10].

Direct Database Access in Next.js Applications: A case study analyzing the use of direct MongoDB connections within Next.js highlighted performance improvements and reduced complexity in backendless design patterns, particularly in small to medium-sized applications.

III. BACKEND-LESS ARCHITECTURE

A. Introduction

Backend-less architecture marks a transformative shift in web development paradigms, challenging the traditional divi- sion between frontend and backend systems. Leveraging mod- ern frameworks such as Next.js, this approach enables direct communication between frontend applications and databases, eliminating the need for a dedicated backend server. This chapter explores backend-less architecture, its motivations, benefits, and associated challenges.

B. Survey and Literature Review Gaps

Based on surveys conducted with IT professionals and

de- velopment teams, several significant gaps have been identified in existing literature and industry adoption:

1) Lack of Comprehensive Studies: Although backend- less architecture is gaining popularity, thorough stud- ies evaluating its performance, scalability, and secu- rity—particularly against traditional models—are lim- ited.
2) Security Concerns: The model introduces novel secu- rity risks due to direct database interactions from the frontend, which are inadequately documented in current literature.
3) Limited Performance Benchmarks: Real-world em- pirical data regarding backend-less system performance remains scarce.
4) Developer Adoption and Learning Curve: The in- dustry lacks exploration into how this architecture af- fects developer workflows, productivity, and the learning curve.
5) Use Case Suitability: There is an absence of clear guidelines on which types of applications are best suited for backend-less approaches based on complexity, size, and traffic expectations.

Factor	Traditional Architecture	Backend-less Architecture
Separation of Concerns	Clear separation	Merged responsibilities
Complexity	Multi-layered	Simplified stack
Development Speed	Slower due to backend inte- gration	Faster due to no backend
Scalability	Backend-dependent	Database service-dependent
Security	Easier to manage layers	Higher risk, needs strong security measures

C. Research Focus

This research aims to bridge these gaps by:

- Conducting performance analyses using **Next.js** with

MongoDB.

- Identifying inherent security vulnerabilities and propos- ing mitigation strategies.
- Evaluating the scalability of backend-less systems for small to medium-sized applications.
- Assessing the impact on developer efficiency and project timelines.
- Offering practical recommendations for integrating backend-less architectures into modern web development workflows.

D. Traditional vs. Backend-less Architecture

1) Traditional (Client-Server) Architecture: Traditional web architectures consist of distinct frontend and backend components:

- Frontend: Handles user interface and experience using HTML, CSS, and JavaScript.
- Backend: Manages business logic and communicates with the database.
- Database: Stores and retrieves application data through backend mediation.

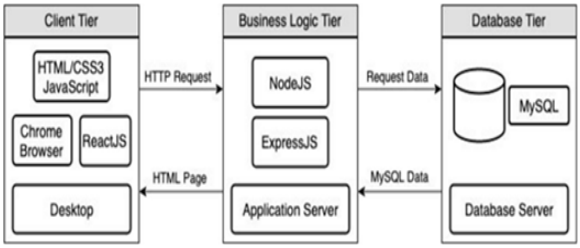

Fig 3.1: Traditional Architecture

2) Backend-less Architecture: In backend-less models:

- The frontend directly interacts with the database using SDKs or APIs.
- Modern frameworks like Next.js facilitate server-side rendering and static site generation with direct data access.

Fig 3.2: Backend-less Architecture

3) Key Differences:

Factor	Traditional Architecture	Backend-less Architecture
Separation of Concerns	Clear separation	Merged responsibilities
Complexity	Multi-layered	Simplified stack
Development Speed	Slower due to backend inte- gration	Faster due to no backend
Scalability	Backend-dependent	Database service-dependent
Security	Easier to manage layers	Higher risk, needs strong security measures

TABLE I: Comparison between Traditional and Backend-less Architectures

E. Methodology for Backend-less Architecture

1) Understanding the Model: Backend-less architecture re- moves the backend server, allowing frontend applications (e.g., built with React.js) to communicate directly with databases like MongoDB using APIs or SDKs.

2) Workflow Overview:

1) User Interaction: Initiated via a Next.js frontend.
2) Database Communication: Direct access to MongoDB via API routes or SDKs.
3) UI Updates: Fetched data updates the user interface.
4) Rendering: Next.js handles SSR/SSG for optimal per- formance and SEO.

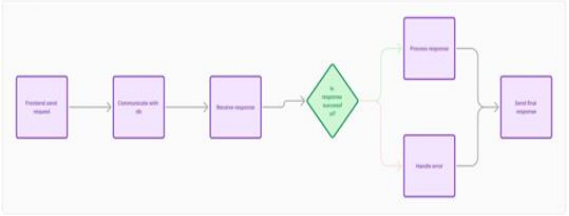

Fig 3.3: Workflow of Backend-less Architecture

3) System Components:

a) React.js (Frontend Layer):

- Component-based: Modular, reusable components sim- plify maintenance.
- Virtual DOM: Efficient rendering for better performance.
- Client-Side Rendering: Supports dynamic content but may impact SEO, mitigated by Next.js.

b) Next.js (Framework Layer):

- SSR (Server-Side Rendering): Improves load time and SEO.
- SSG (Static Site Generation): Ideal for static content, boosts performance.
- API Routes: Enables CRUD operations directly con- nected to MongoDB.

c) MongoDB (Database Layer):

- NoSQL Flexibility: JSON-like structure aligns with JavaScript.
- Direct Interaction: Simplifies logic and enhances speed.
- Scalability: Efficient for growing applications, especially when coupled with Next.js.

d) Security Measures:

- Authentication & Authorization: JWT or Auth0 for secure access.
- Input Validation: Prevents injection and malicious at- tacks.
- Environment Variables: Secure storage for credentials and keys.

e) Performance Optimization:

- Caching: On both client and server sides.
- Code Splitting & Lazy Loading: Reduces initial load time.
- Image Optimization: Built-in features of Next.js for device-specific delivery.

Fig 3.4: Performance Optimization Techniques

F. Comparison with Other Backend-less Technologies

1) WordPress: Uses REST API/GraphQL to achieve backend-less-like functionality.

- Pros: User-friendly, plugin-rich, strong community.
- Cons: Performance bottlenecks, limited scalability, plugin-related vulnerabilities.
- Additional Consideration: Requires separate hosting setup for headless frontend.
- Data Latency: Data fetching through REST/GraphQL APIs may introduce additional latency, especially with multiple plugin dependencies.

2) Firebase: Provides backend services including real-time database, authentication, and hosting.

- Pros: Real-time syncing, integrated auth, scalable.
- Cons: Vendor lock-in, complex pricing, reduced flexibil- ity.
- Additional Consideration: Frontend still needs to be hosted separately if not using Firebase Hosting.
- Data Latency: Real-time capabilities offer low latency, but initial connection and data rules can slightly delay response.

G. Rise of Backend-less Architectures

1) Driving Factors:

- Faster Development: No backend to manage.
- Cost Efficiency: Reduced infrastructure costs.
- Simplified Maintenance: Focus remains on frontend.
- Cloud Scalability: Platforms like Firebase and AWS Amplify scale automatically.

2) Frameworks Enabling the Shift: Next.js

enhances backend-less models through SSR and SSG:
- SSR improves SEO and interactivity.
- SSG provides performance advantages for static pages.

3) WordPress vs. Backend-less with Next.js:
- WordPress: Best for content-driven sites with limited interactivity.
- Next.js: Better suited for modern, dynamic applications with greater flexibility and control.

H. Why Next.js for Backend-less Architecture

Next.js's versatility makes it an excellent choice for backend-less architectures. It allows developers to bypass traditional backend development by enabling direct database interactions through server-side functions or API routes. This reduces the need for middleware, simplifies deployment, and can significantly speed up development cycles.

The framework's integration with modern tools and cloud platforms, such as Vercel and AWS, further enhances its capa- bilities, making it easier to deploy scalable, high-performance applications. By leveraging these features, Next.js helps bridge the gap between frontend and backend, creating a streamlined development process that aligns with the goals of backend-less architecture.

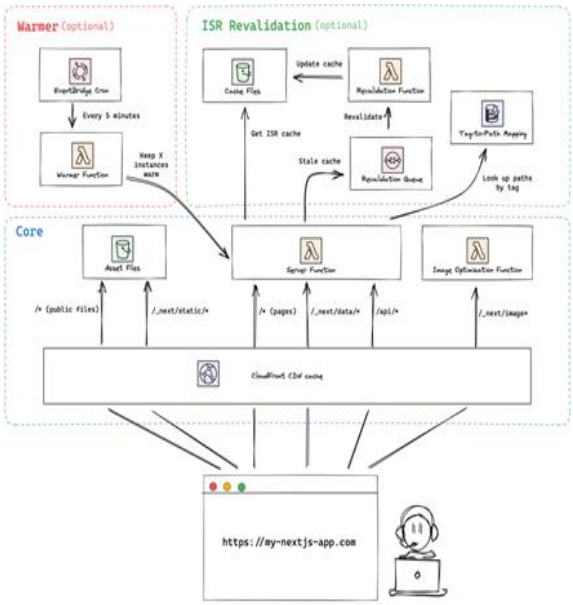

Fig 3.4: Next.js for Backend-less Architecture [11]

IV. PROPOSED APPROACH

A. Backend-less Architecture with Next.js

Recent advancements in frontend frameworks such as Next.js have made it possible to streamline web development by eliminating the traditional backend layer [12]. The pro- posed backend-less architecture enables direct communication between the frontend and database using integrated API routes within the same framework, significantly reducing architec- tural complexity and improving performance.

1) Next.js API Routes: Next.js offers API Routes, lo- cated in the app/api directory, which act as backend endpoints within the same project [13]. These server-side routes are excluded from the client-side bundle, enabling clean separation of concerns while keeping deployment minimal.
- Each file in app/api maps to a public endpoint at

/api/*.
- API Routes support secure use of environment variables

for configurations such as database connections.
- MongoDB can be integrated directly within these routes using middleware.
- Business logic and CRUD operations can be handled server-side without requiring a separate backend server.
- This structure simplifies development, improves perfor- mance, and supports secure practices like authentication and input validation.
- Deployment is streamlined, as both frontend and backend logic reside within the same codebase.

2) MongoDB Integration: MongoDB is used as the pri- mary database in this approach. Connection strings are stored securely in a .env file, and accessed within API routes using environment variables [14]. This secure setup ensures sensitive credentials are never exposed to the client.

3) Security Considerations: Even though the API runs on the server, implementing authentication and authorization is critical. JSON Web Tokens (JWT) and middleware functions are used to validate users and prevent unauthorized access [15].

4) Unified Hosting: With platforms like Vercel, both fron- tend pages and API endpoints are hosted under the same domain (e.g., https://your-app.vercel.app/api). This simplifies the DevOps process and reduces infrastructure overhead.

5) Server-Side Rendering (SSR) and Static Site Generation (SSG): Next.js supports SSR and SSG to enhance SEO, loading performance, and user

experience [16]. SSR pre- renders pages on the server, while SSG generates pages at build time for near-instant load speeds. Both are ideal for small to medium web applications.

6) App Router and Route Handlers: In the latest Next.js architecture, developers can utilize Route Handlers or Server Components as alternatives to traditional API Routes [17]. This flexibility allows for more dynamic, interac- tive experiences and greater control over rendering behavior.

By leveraging these features, this research highlights the practicality of backend-less architecture using Next.js. It demonstrates how server-side capabilities—such as API Routes, Server Components, and secure environment han- dling—enable robust web applications without the need for a separate backend server. Subsequent sections explore the implementation, security considerations, and performance out- comes of this approach.

V. SETUP AND IMPLEMENTATION

This section outlines the key steps and methodologies used in setting up and implementing the *Wellness Journey* applica- tion. It provides a comprehensive overview of project manage- ment practices, version control, and technical implementation. The project combines Next.js with direct MongoDB database access and API-based interactions, resulting in a simplified full-stack web application that eliminates the need for a traditional backend server.

A. Setup

1) ClickUp Setup: Task and Milestone Management: ClickUp was selected as the primary project management tool for organizing development workflows. It provided centralized task tracking, collaboration, and milestone monitoring.

- Task Management: Tasks were broken down into sprints and modules, such as frontend development, database schema setup, and testing.
- Milestone Tracking: Key goals (e.g., Blog Page Setup, API Integration, UI Testing) were defined with timelines and responsible team members.
- Progress Monitoring: Real-time dashboards, Gantt charts, and task dependencies helped identify bottlenecks and maintain project velocity.

2) Project Documentation and Risk Management: To en- sure clarity and reduce development risk, the following docu- mentation practices were adopted:

- Requirement Documentation: Outlined core features (blogs, plans, admin/user modules), user journeys, and functional requirements.
- Risk Management: Identified risks such as database connectivity issues, deployment bottlenecks, and scala- bility limitations. Risk mitigation strategies were added as contingency plans.

3) Version Control and Repository Initialization:

- **Version Control:** A Git repository (hosted on GitHub) was initialized. Git branches were used for features, bug fixes, and releases, ensuring collaborative and conflict- free development.
- **Standardized Folder Structure:** Project files were log- ically separated into reusable folders (components/, lib/, models/, redux/, etc.) for better organization and scalability.

B. Project Implementation

1) Project Overview: Wellness Journey is a web-based plat- form that promotes health and well-being through educational content and subscription plans. The platform consists of three core modules:

- **Admin Module:** Create/manage blogs, subscription plans, contact requests, and user records.
- **User Module:** Explore wellness blogs, view pricing plans, and contact the platform for support.
- **Landing Page:** Public-facing portal showcasing blog entries and subscription offerings.

The application uses Next.js with server-side rendering (SSR) and direct MongoDB database access for faster per- formance, improved SEO, and backendless architecture. For secured data manipulation, selected operations (like blog cre- ation) are routed through Next.js API routes.

TABLE II: Technology Stack Overview

Layer	Technology Used
Frontend	Next.js (App Router)
Database	MongoDB(Mongoose ODM)
State Management	Redux Toolkit + Thunks
UI Components	TailwindCSS, React Icons
Authentication	JWT-based (Planned)
Deployment	Vercel
Project Management	ClickUp
Version Control	Git + GitHub

Folder Structure: The project follows a modular, role-based folder structure under the src/ directory:

- app/— Routes organized by layout (authLayout, userLayout, superAdminLayout)
- app/api/ — Next.js API routes (e.g., addBlog, login)
- components/— Reusable UI components
- redux/— Slices, thunks, and store configuration
- lib/— Utility functions (e.g., executeDB)
- models/— Mongoose models (e.g., Blogs.ts)
- services/ — HTTP wrapper functions (executeHttp)
- hooks/, constants/, types/— Supporting logic and structure

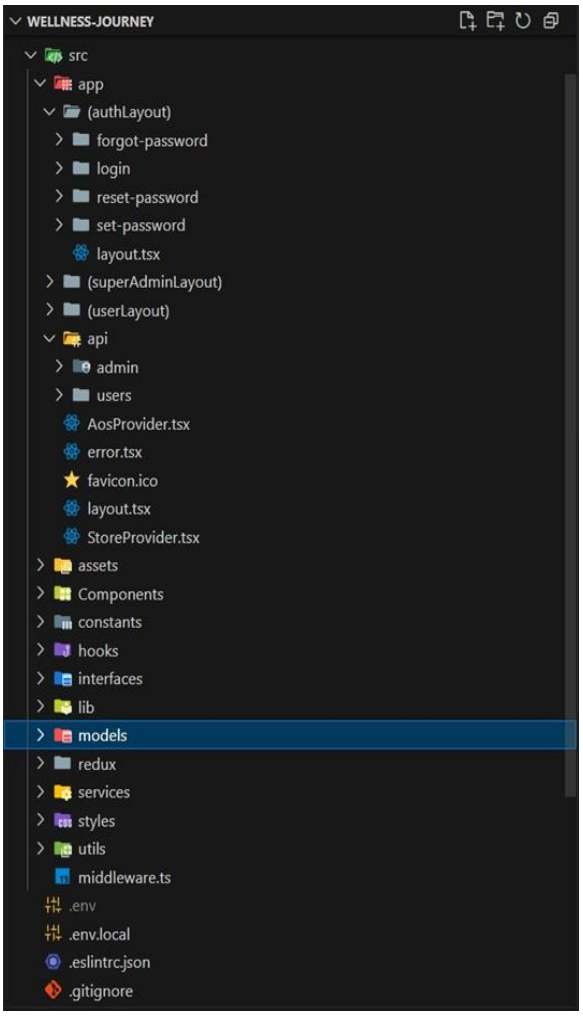

Fig 4.1: Folder Structure

2) Direct Database Call Example: Fetching Blog Posts: This section demonstrates how Next.js supports direct in- teraction with a MongoDB database, bypassing traditional backend layers, by leveraging Server-Side Rendering (SSR) and asynchronous data-fetching within server components. This approach offers a more streamlined architecture, reducing complexity and improving performance for read-heavy opera- tions such as blog listing.

a) Objective: To retrieve blog posts directly from a MongoDB collection, bypassing the need for a sep- arate backend API, using server-side logic within the src/lib/db/posts.tsmodule. The fetched data is then used within the Next.js page at runtime, enabling pre-rendered and SEO-friendly content delivery.

b) Implementation: getPosts Function: The getPosts function resides in the lib/db directory and uses a custom helper function executeDBto connect to MongoDB and perform a read operation. The function utilizes MongoDB's aggregation pipeline to:

- Filter out soft-deleted documents using { isDeleted: false }
- Sort posts in descending order by creation time
- Project only essential fields to reduce payload size
- Map MongoDB's ObjectId fields to strings for frontend compatibility

```
// src/lib/db/posts.ts
export const getPosts = async () => {
return await executeDB<Post[]>(async (db) => { try {
        const posts = await db
.           .collection("blogs")
.aggregate<Post>([
{ $match: { isDeleted: false } },
{ $sort: { createdAt: -1 } },
{
$project: {
_id: 1,
blogTitle: 1,
blogBanner: 1,
otherBlogData: 1,
createdAt: 1,
},
},
        })
.toArray();

return posts.map((post) => ({
...post,
_id: post._id.toString(),
otherBlogData: post.otherBlogData.map((data) => ({
...data,
_id: data._id.toString(),
})),
}));
```

```
} catch (e) {
return [];
}
})
};
```

c) Integration with Next.js Page Component: In Next.js, components within the app/ directory support asynchronous server-side logic. In this case, the Blogs page component (src/app/(userLayout)/blogs/page.tsx) directly calls getPosts() using the async pattern. Because this function executes on the server, the fetched data is rendered into the page during the initial load, improving SEO, perfor- mance, and user experience.

```
// src/app/(userLayout)/blogs/page.tsx
import { getPosts } from "@wellness-journey/lib/db/posts"; import BlogList from "./BlogList";
export default async function Blogs() { const posts = await getPosts(); return (
<section>
<h1>Wellness Blogs</h1>
<BlogList posts={posts} />
</section>
);
}
```

d) Helper Function: executeDB: To simplify Mon- goDB operations and encapsulate connection logic, the plat-

form uses a reusable helper function executeDB. This utility abstracts the MongoClient connection lifecycle, making database interaction consistent and secure across modules.

- Establishes a connection using credentials from .env

file.

- Optionally supports persistent connections through caching for optimized performance.
- Accepts a callback function that receives a MongoDB Db

instance.

- Automatically closes the connection unless caching is explicitly enabled.

Simplified Usage in Read Operations (No Cache):

```
// src/lib/db/mongo.ts
export const executeDB = async <R>( cb: (db: Db) => R | Promise<R>, options = { useCache: false }
): Promise<R> => {
const conn = await createConnection();
// or reuses global.mongoConn if useCache=true const db = conn.db(process.env.DB_NAME);
const result = await cb(db);
if (!options.useCache) await conn.close(); return result;
};
```

TABLE III: Direct Database Concepts and Their Applications

Concept	Description
Server-Side Rendering (SSR)	Next.js automatically renders this page on the server with pre-fetched data, improving SEO and load time.
App Directory and Server Components	The page uses the new app/ directory-based routing and server com- ponents introduced in Next.js 13+. This enables native support for async functions.
Direct Database Access	By placing the getPosts function in a server-only file (lib/db/posts.ts), we avoid exposing sensitive logic and credentials on the client.
No Need for Separate Backend	Thanks to the integrated server and API capabilities in Next.js, the appli- cation architecture remains backendless yet secure and scalable.
MongoDB Aggregation Pipeline	Used for efficient querying and projection of relevant blog data.

3) API-Based Data Access Example: Adding a Blog: While read operations such as fetching blog posts can effi- ciently be handled using direct database access within server components, write operations (such as adding or updating content) typically require additional security, validation, and encapsulation. In this section, we explore a more traditional yet integrated API-based architecture using Next.js API routes for handling sensitive operations like blog creation.

a) Objective: To add a new blog post securely to the MongoDB database through a dedicated API endpoint in Next.js. This separation of concerns

ensures that sensitive database write operations are protected and executed server- side via controlled interfaces.

b) Frontend – Dispatching via Redux Thunk: On the client side, a Redux createAsyncThunk middleware func- tion is used to asynchronously call the backend API. The blog data entered by the user is sent via a POST request to the /api/addBlog endpoint. This approach allows for proper error handling, optimistic UI updates, and centralized state management.

```
// src/redux/slices/blogSlice.ts
export const addBlog = createAsyncThunk(
"blogs/addBlog",
async (payload: AddBlogDto) => { const response =
await executeHttp(
"/api/addBlog", "POST",
payload
);
return response.data;
}
);
```

Here, executeHttp is a custom utility that wraps the native fetch API, adding consistent headers, JSON seri- alization, and error handling. The AddBlogDto interface ensures type safety by enforcing a specific structure for the blog payload (e.g., blogTitle, blogBanner, author, tags, content).

c) API Route – /api/addBlog: The API route is implemented using Next.js's file-based routing under the app/api directory. The server-side function is executed only on the server environment, ensuring that business logic and database access remain secure. This function:

- Parses and validates the incoming JSON body
- Connects to MongoDB using the helper executeDB
- Inserts the blog document into the blogscollection
- Returns a JSON response with the newly inserted ID

```
// src/app/api/addBlog/route.ts
import { NextRequest, NextResponse } from "next/
server";
import { executeDB } from "@wellness-
journey/lib/db/ executeDB";
export async function POST(req: NextRequest) { try {
const blogData = await req.json();
// Basic validation (can be replaced with Zod or Joi)
if (!blogData.blogTitle || !blogData.content) { return
NextResponse.json(
{ success: false, error: "Missing required fields" },
{ status: 400 }
);
}
const result = await executeDB(async (db) => { const
insertResult = await db.collection("
blogs").insertOne({
...blogData, isDeleted: false, createdAt: new Date(),
updatedAt: new Date()
});
return insertResult;
});
return NextResponse.json({ success: true, insertedId:
result.insertedId });
} catch (error) {
return NextResponse.json({ success: false, error
: error.message }, { status: 500 });
}
}
```

d) Backend Helper – executeDB: The API uses the executeDB helper to abstract MongoDB connection logic. This utility handles:

- Opening and closing database connections
- Global caching of MongoDB client for performance (when useCache: true)
- Automatic error handling for connection failures
- Executing the actual query block via a passed callback Only the relevant portion of the helper logic is shown below:

```
// src/lib/db/executeDB.ts
export const executeDB = async <R>( cb: (db: Db) =>
R | Promise<R>,
options: { useCache: boolean } = { useCache: false }
): Promise<R> => {
let conn: MongoClient;

if (options.useCache) { if (!global.mongoConn) {
await updateGlobalMongoConn();
}
conn = global.mongoConn!;
} else {
conn = await createConnection();
}

const db = conn.db(process.env.DB_NAME); const
result = await cb(db);

if (!options.useCache) { await conn.close();
```

```
}

return result;
};
```

e) Advantages of API-based Write Architecture:

- **Encapsulation of Business Logic:** Write operations often require input validation, authentication, and side effects like logging — best handled server-side.
- **Security:** Prevents direct database mutations from the client, reducing the attack surface.
- **Scalability:** API routes can later be extended to support role-based access control, logging, or rate limiting.
- **Decoupling:** This modular architecture keeps frontend, API, and database logic cleanly separated.

f) Benefits of This Approach:

- **Security:** The API route runs only on the server, **protect**- ing database credentials and logic.
- **Encapsulation:** Write operations can include complex validation and business logic centralized in the API layer.
- **Flexibility:** Allows reuse by different clients (e.g., web, mobile).

C. Summary

The *Wellness Journey* project adopts a hybrid data access approach using Next.js:

- **Direct Database Access:** Used within server components for efficient, SEO-optimized read operations without a separate backend.
- **API Routes:** Employed for secure, validated write oper- ations to ensure data integrity and encapsulation.

This approach balances performance with security, leverag- ing Next.js capabilities to build a scalable and backendless full-stack application.

VI. RESULTS AND ANALYSIS

This section analyzes the performance of the hybrid ap- proach adopted in the *Wellness Journey* project by examining build output, runtime network timings, and PageSpeed scores for both direct database access and API-based methods.

A. Build Output Analysis

Next.js optimizes routes using server components and static rendering where applicable. The final build log reflects op- timized server-side rendering (SSR) for dynamic routes and efficient static generation for others.

- Pages using direct database access (e.g., /blogs, /blogs/[blogId]) show efficient bundling with no extra API load, reducing client-side JS overhead.
- API routes (e.g., /api/admin/blogs/add) show 0 B in bundle size as expected, since they're server-only and not part of the client bundle.
- Static and dynamic routes are clearly separated, ensuring optimal first load JS and performance across routes.
- API routes result in 0 B build size, while page routes (especially with direct DB access) are built into small dynamic chunks (in KBs), enabling faster loading and efficient server-side rendering.

```
11:50:16.071  Route (app)                              Size     First Load JS
11:50:16.071  ┌ ƒ /                                    5.5 kB          163 kB
11:50:16.071  ├ ○ /_not-found                          879 B          88.2 kB
11:50:16.071  ├ ƒ /api/admin/blogs/add                 0 B                0 B
11:50:16.071  ├ ƒ /api/admin/blogs/add/[blogId]        0 B                0 B
11:50:16.071  ├ ƒ /api/admin/blogs/get                 0 B                0 B
11:50:16.071  ├ ƒ /api/admin/blogs/get/[blogId]        0 B                0 B
11:50:16.071  ├ ƒ /api/admin/contact                   0 B                0 B
11:50:16.071  ├ ƒ /api/admin/plans                     0 B                0 B
11:50:16.071  ├ ƒ /api/users/add                       0 B                0 B
11:50:16.071  ├ ƒ /api/users/contact                   0 B                0 B
11:50:16.071  ├ ƒ /api/users/forgotPassword            0 B                0 B
11:50:16.071  ├ ƒ /api/users/login                     0 B                0 B
11:50:16.072  ├ ƒ /api/users/payment/initiate          0 B                0 B
11:50:16.072  ├ ƒ /api/users/payment/verify            0 B                0 B
11:50:16.072  ├ ƒ /api/users/updatePassword            0 B                0 B
11:50:16.072  ├ ƒ /api/users/verifyKey                 0 B                0 B
11:50:16.072  ├ ƒ /blogs                               2.65 kB         215 kB
11:50:16.072  ├ ƒ /blogs/[blogId]                      184 B          87.5 kB
11:50:16.072  ├ ƒ /contact-us                          2.35 kB         278 kB
11:50:16.072  ├ ○ /forgot-password                     2.23 kB         241 kB
11:50:16.072  ├ ○ /login                               3.64 kB         258 kB
11:50:16.072  ├ ƒ /onboarding                          5.53 kB         318 kB
11:50:16.072  ├ ƒ /pricing                             665 kB          1.11 MB
11:50:16.072  ├ ƒ /recipes                             5.88 kB         136 kB
11:50:16.072  ├ ƒ /reset-password/[id]                 2.5 kB          241 kB
11:50:16.072  ├ ƒ /set-password/[key]                  2.69 kB         241 kB
11:50:16.072  ├ ○ /superAdmin/blogs                    63.8 kB         455 kB
11:50:16.072  ├ ○ /superAdmin/contacts                 4.45 kB         367 kB
11:50:16.073  ├ ○ /superAdmin/plans                    2.23 kB         208 kB
11:50:16.073  ├ ○ /superAdmin/users                    184 B          87.5 kB
11:50:16.073  └ ƒ /user/profile                        6.3 kB          164 kB
11:50:16.074  + First Load JS shared by all            87.3 kB
11:50:16.074    ├ chunks/7023-07fe8387ad09e407.js      31.7 kB
11:50:16.074    ├ chunks/fd9d1056-e8bb78c3d6f96093.js  53.6 kB
11:50:16.074    └ other shared chunks (total)          2 kB
11:50:16.074
11:50:16.074
11:50:16.074  ƒ Middleware                             32.5 kB
11:50:16.074
11:50:16.074  ○ (Static)   prerendered as static content
11:50:16.075  ƒ (Dynamic)  server-rendered on demand
```

Fig 6.1: Production Build log

B. Network Timing Comparison

The application demonstrates varied timings depending on the data access approach:

TABLE IV: Network Timing Analysis

Page/Endpoint	Access Method	Time Taken
/api/admin/contact(GET contact list)	API Route	268.06 ms
/pricing	SSR (Direct DB Access)	1.22 s
/blogs(SSR blog page)	Direct DB Access	1.18 s
/blogs/[blogId](Dynamic read)	Direct DB Access	3.2 s
/api/admin/plans(Admin plan API)	API Route	77.51 ms

Fig 6.2: Get Blog Timming

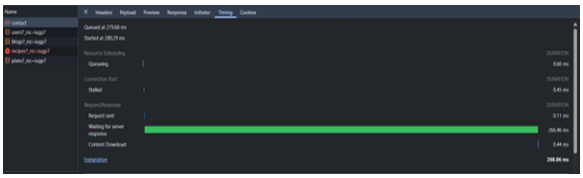

Fig 6.3: Get Pricing Timming

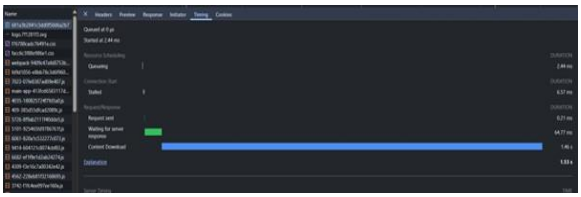

Fig 6.4: Single blog Timming

Observation:

- API routes are more responsive (e.g., admin plan: 77 ms), showing their strength in handling small, secure write/read operations.
- Direct DB access for pages like /blogs enables SEO-friendly SSR but may take longer for dynamic content (e.g., 3.2s for blog details), suggesting the need for further optimization such as caching or pagination.

C. PageSpeed Insights

The application demonstrates strong overall performance across devices and routes:

TABLE V: PageSpeed Performance Scores

Page	Mobile Score	Desktop Score	SEO Score
Home (/)	98	94	100
Blogs Listing (/blogs)	92	97	100
Blog Detail (/blogs/[blogId])	92	98	100

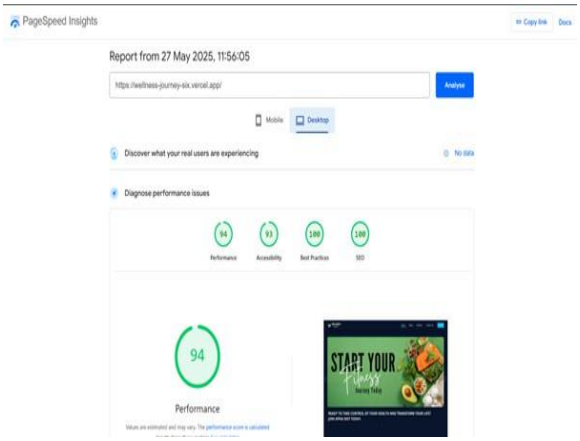

Fig 6.5: Home Page Speed

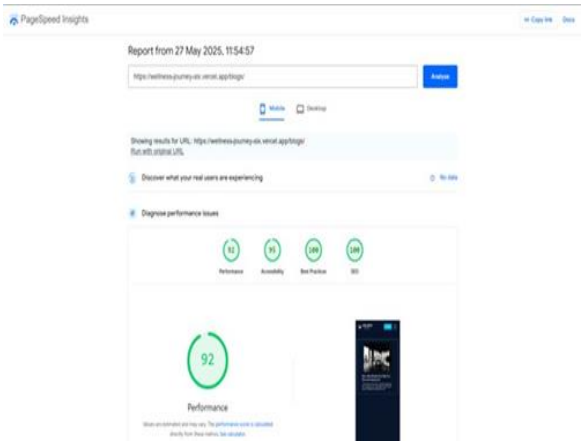

Fig 6.6: Blog Page Speed

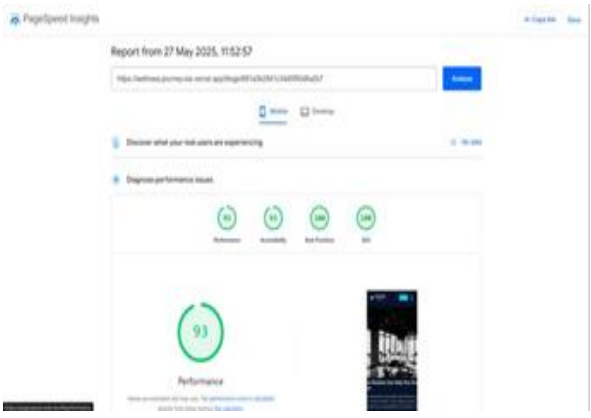

Fig 6.7: Single Blog Page Speed

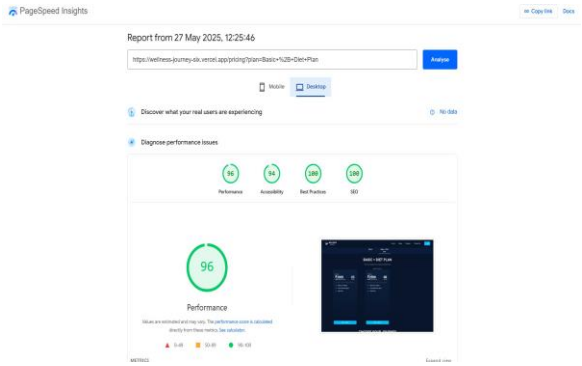

Fig 6.8: Pricing Plan Page Speed

D. 6.2 Analysis

1) 6.2.1 Performance Improvements (Load Time, SSR, and API Speed): The application demonstrates clear performance enhancements through architectural choices such as direct database access, server-side rendering, and efficient API design. Key areas of improvement are discussed below:

- Reduced Load Time:
  - *Efficient Routing and Data Fetching:* Leveraging Next.js's App Router and server components allows for pre-rendered content to be served quickly, signif- icantly reducing the load time for SEO-sensitive and high-traffic pages.
  - *Optimized Queries:* Direct MongoDB access minimizes processing delays caused by intermediate layers, resulting in faster data fetches and smoother user experience.
- Server-Side Rendering (SSR):
  - *Improved SEO and Initial Load:* SSR ensures content is rendered on the server before being delivered to the client, enhancing both search engine visibility and user-perceived performance.
  - *Faster Time-to-Interactive (TTI):* With SSR, users see meaningful content immediately, even before JavaScript fully loads, reducing TTI especially on slower networks.
- API Speed:
  - *Lean API Architecture:* Using API routes within Next.js reduces backend complexity and improves request-response cycles by eliminating the need for a separate backend server.
  - *Optimized Authentication Flow:* Secure, token-based JWT authentication offers low-latency session han- dling, with minimal overhead on each request.
  - *Measured Performance:* Network timing metrics confirm excellent response times — e.g., API contact list loads in 268ms, and pricing page in 1.22s — validating the real-world perfor- mance improvements.
- Build Efficiency:
  - API routes generate 0 B build size, confirming they are server-only and excluded from the client bundle.
  - Pages using direct DB access are dynamically built and remain lightweight (in KBs), contributing to quicker loads and a more performant front-end.
  - Static and dynamic route separation ensures optimized builds, enabling the application to balance SSR, static delivery, and dynamic rendering as needed.
- PageSpeed and Lighthouse Scores:
  - Home and blog pages score highly in Lighthouse audits (e.g., 94–97 desktop, 92 mobile, 100 SEO), proving the application's responsiveness and techni- cal SEO soundness.

Overall Impact: By strategically integrating SSR, direct database access, and optimized API design, the system min- imizes latency, enhances SEO, and ensures consistent perfor- mance across devices. This hybrid approach results in faster interactions, better user retention, and reduced infrastructure complexity — making it an effective model for scalable full- stack web applications.

Conclusion: Next.js's hybrid rendering model—direct database access for dynamic, SEO-sensitive pages and API routes for secure write operations—ensures a high- performance, scalable, and backendless architecture. The per- formance is validated by low network latencies, small build sizes, and excellent PageSpeed scores.

VII. APPLICATIONS OF BACKEND-LESS SOLUTIONS WITH NEXT.JS AND DIRECT DATABASE INTERACTION

The backend-less architecture using Next.js and direct database access simplifies development and enables faster, scalable applications. This approach is highly useful across various domains, as outlined below:

A. Startups and Small Businesses

For startups and small teams with limited resources, backend-less solutions reduce setup and infrastructure costs. With Next.js handling both frontend and backend logic, devel- opers can quickly build and deploy Minimum Viable Products (MVPs) using

direct database connections, saving time and effort.

B. Content-Based Websites

Websites such as blogs, news portals, and documentation sites benefit from static site generation (SSG) in Next.js. These platforms load faster, are SEO-friendly, and are easier to maintain, especially when content is fetched directly from the database at build time.

C. Real-Time Applications

Apps like chat systems, dashboards, or collaboration tools can use Next.js along with WebSockets or real-time databases (e.g., Firebase, MongoDB Change Streams) to deliver live updates. This setup enables real-time functionality without needing a traditional backend.

D. E-Commerce Platforms

Small and medium online stores can be built efficiently using Next.js with direct integration to services like Stripe for payments and Firebase for authentication. This reduces backend complexity while delivering a complete shopping experience.

E. Progressive Web Apps (PWAs)

Next.js supports building PWAs that function like native apps. These apps load quickly, work offline, and support push notifications. Backend-less PWAs connected directly to the database ensure smooth performance and user engagement.

F. Educational Platforms

Online learning systems, quiz portals, and LMS platforms can use backend-less architecture to manage users, content, and progress tracking efficiently. Real-time updates and API routes help scale with minimal maintenance.

G. Internal Business Tools

Custom dashboards, inventory trackers, or task managers for internal use can be built quickly using Next.js with direct database access. These tools are easy to update, require less DevOps effort, and improve productivity.

VIII. CHALLENGES OF BACKEND-LESS ARCHITECTURE WITH NEXT.JS AND DIRECT DATABASE ACCESS

While backend-less architecture with Next.js offers many benefits, it presents certain challenges:

- Scalability: Managing high user loads and database connections can be difficult without dedicated backend infrastructure.
- Security: Exposing database logic alongside frontend increases risks; robust authentication and input validation are critical.
- Code Maintainability: Combining frontend, API, and database logic may lead to tightly coupled code, reducing maintainability.
- Limited Abstraction: Lack of traditional backend layers can reduce modularity and increase technical debt risk.
- Testing and Debugging: Integrated code complicates error tracing, requiring disciplined testing strategies.
- Hosting and Vendor Lock-in: Dependence on serverless platforms may limit flexibility and increase costs.
- Connection Limits: Serverless functions face database connection constraints that need optimization.
- Monitoring: Lack of built-in tools demands manual setup for logging and performance tracking.
- Real-Time Features: Adding real-time capabilities in- creases complexity, potentially reducing simplicity bene- fits.

IX. FUTURE SCOPE

Backend-less architecture with Next.js opens up several avenues for future development:

- Expanded Database Support: Integrate databases like MySQL, Firebase, and DynamoDB for greater flexibility and real-time syncing.
- Stronger Security: Implement advanced authentication, encryption, and AI-based threat detection.
- Enhanced Scalability: Optimize for large-scale applica- tions using load balancing and connection pooling.
- Improved Developer Tooling: Develop better debug- ging, monitoring, and deployment tools to aid developers.
- Performance Optimization: Focus on faster rendering, caching, and API response times for improved user experience.
- Advanced API Capabilities: Support complex business logic and workflows directly within API routes.

- Support for Small Teams: Create tools and templates that empower individual developers and small teams to efficiently manage full-stack projects.

X. CONCLUSION

The backend-less architecture explored in this research, us- ing the capabilities of Next.js, marks a significant advancement in modern web development. By integrating frontend render- ing, API handling, and direct database communication within a single framework, this approach simplifies the development process and optimizes both performance and maintainability. Next.js empowers this streamlined model through its built-in API routes and native support for server-side rendering (SSR) and static site generation (SSG). The direct interaction with databases such as MongoDB eliminates the need for a separate backend layer, resulting in lower latency, reduced infrastruc- ture complexity, and faster deployment cycles. This makes it particularly suitable for small to medium-scale applications and Minimum Viable Products (MVPs), were development
speed and cost-efficiency are crucial [16], [18].
The performance benefits, including improved load times, enhanced SEO, and faster Time-to-Interactive (TTI), lead to a better user experience and operational efficiency [19]. Addi- tionally, maintaining business logic and data operations within the same codebase fosters consistency, easier debugging, and quicker iteration.
In summary, this study demonstrates that a backend-less approach using Next.js is not only feasible but also highly effective for building scalable, high-performance web ap- plications. The findings provide a practical framework for developers and teams seeking to simplify their tech stack, reduce overhead, and accelerate time to market, while still retaining the flexibility to scale and extend as needed in the future.

XI. ACKNOWLEDGMENT

Simran Prasad thanks project guide Ms. Rauki Yadav for continuous support and guidance throughout the research. Gratitude is also extended to Ms. Rauki Yadav, Head of
Department, and Dr. Vineet Goel, Director, for their encour- agement and academic support. Appreciation is expressed to the faculty of Bhagwan Mahavir College of Engineering and Technology (BMCET) for providing a conducive learning environment. Support from family and peers is also gratefully acknowledged.

REFERENCES

[1] G. Rauch and Z. Inc., “Next.js: The react framework,” 2016, available at: https://nextjs.org.
[2] A. Mohan, “Fullstack javascript development with mern: Simplified ar- chitecture for modern apps,” International Journal of Web Engineering, vol. 9, no. 1, 2021.
[3] J. Smith and T. Allen, “Empirical evaluation of javascript frontend frameworks,” Journal of Software Engineering, vol. 10, no. 1, pp. 33–42, 2021.
[4] M. Patel and D. Roy, “Backend frameworks for restful api development: A comparative study,” International Journal of Computer Applications, vol. 182, no. 20, pp. 12–19, 2019.
[5] S. Lee and H. Kim, “Rendering strategies in modern web applications: A review,” Web Technologies Review, vol. 8, no. 3, pp. 45–53, 2020.
[6] E. Johnson and L. Wang, “The rise of backendless architectures: An analysis,” Cloud Computing Journal, vol. 12, no. 1, pp. 78–88, 2022.
[7] A. Gupta and P. Mehta, “Performance comparison of full stack javascript frameworks: Mern vs mean,” International Journal of Web Engineering, vol. 6, no. 1, pp. 50–58, 2021.
[8] A. Gupta and R. Singh, “Comparative study of frontend frameworks: Angular, react and vue,” International Journal of Web Development, vol. 5, no. 2, pp. 101–108, 2020.
[9] V. Sharma and K. Chawla, “Security evaluation of web development frameworks,” Cybersecurity Advances, vol. 4, no. 2, pp. 90–98, 2022.
[10] L. Brown and J. Green, “Improving seo with server-side rendering: A next.js case,” Digital Marketing and Web Optimization, vol. 9, no. 1,pp. 22–30, 2021.
[11] Vercel, “Next.js documentation,” https://nextjs.org/learn, 2016, accessed: 2025-05-27.
[12] G. Rauch, Next.js: The React Framework, 2020, accessed 2025-05-24. [Online]. Available:

https://nextjs.org
[13] Vercel Team, “Api routes | next.js documentation,” 2025, available online. [Online]. Available: https://nextjs.org/docs/pages/ building-your-application/routing/api-routes
[14] M. Inc., “Mongodb node.js driver — official docs,” 2023, accessed 2025-05-24. [Online]. Available: https://mongodb.com/docs/drivers/node
[15] L. Ferreira, “Json web token (jwt): Authentication best practices,” Auth0 Blog, 2020, available online. [Online]. Available: https: //auth0.com/blog/what-is-json-web-token
[16] Vercel Team, “Rendering: Ssr vs. ssg,” 2025. [Online]. Available: https://nextjs.org/docs/pages/building-your-application/rendering
[17] “App router: Route handlers | next.js docs,” 2025, available online. [Online]. Available: https://nextjs.org/docs/app/ building-your-application/routing/route-handlers
[18] A. Gupta, “Why mvps are ideal for startups: A technical perspective,” Startup Dev Journal, 2021. [Online]. Available: https://startupjournal. dev/mvp-approach
[19] A. Firdaus, “Building real-time applications with firebase and next.js,” WebDev Modern, 2022. [Online]. Available: https://webdevmodern.dev/ firebase-nextjs-realtime