Taras Byalyk

Student number: 411979

Field of study: Computer Science

“GalleryGlobe” – Next.js Database Web Application for Art Discovery

Bachelor’s thesis

Department of Computer Science

Supervisor: dr. Adam Bartoszek

Łódź 2025

Contents

Introduction

The digital age has fundamentally transformed how art is accessed and appreciated, moving gallery experiences from physical spaces to expansive online platforms. This evolution necessitates advanced digital tools capable of effectively curating, presenting, and engaging users with diverse art collections. This bachelor's thesis addresses this need through the development of "GalleryGlobe," a modern web application designed for art discovery and management.

The core objective of this work was to develop a comprehensive full-stack web application functioning as a digital art gallery. "GalleryGlobe" leverages contemporary web technologies, including Next.js for its frontend and a robust backend infrastructure for data handling and user authentication. The application aims to provide an intuitive and engaging experience for all users, from general art enthusiasts to dedicated collectors, facilitating the exploration of artworks and artists while offering personalized features for registered accounts. Through a streamlined user interface and efficient data processing, GalleryGlobe seeks to enhance digital art accessibility.

The core objective of this work was to develop a comprehensive full-stack web application functioning as a digital art gallery. "GalleryGlobe" leverages contemporary web technologies, including Next.js for its frontend and a robust backend infrastructure for data handling and user authentication. The application aims to provide an intuitive and engaging experience for all users, from general art enthusiasts to dedicated collectors, facilitating the exploration of artworks and artists while offering personalized features for registered accounts. Through a streamlined user interface and efficient data processing, GalleryGlobe seeks to enhance digital art accessibility.

Application data

Web Address:

galleryglobe.vercel.app

Credentials:

Admin account credentials:

email: admin@email.com

password: Admin123!

User: test@test2.com

password: 123qweasd

Chapter 1

Programming tools

This chapter provides an overview of the programming tools used in the development of the "GalleryGlobe" web application. As defined by Marc Hornbeek in *Continuous Testing, Quality, Security, and Feedback*, "tools are specific software or applications that perform distinct tasks or functions. These can range from compilers, debuggers, and code editors to more specialized tools for version control, project management, continuous integration, and testing" (see [4]). A balanced selection of these tools was driven by best practices in the modern world of web development.

1.1 Core Technologies and Frameworks

At its foundation, "GalleryGlobe" is a `Next.js` application. It is a full-stack `React` framework that offers additional functionality, such as file-based routing, server components, hybrid rendering capabilities, and server-side rendering. It was selected due to its wide adoption and, critically for this application, its image optimization capabilities (see [1]).

For the main programming language, `TypeScript` was chosen. It is a high-level programming language that adds static typing and optional type annotations to `JavaScript`. It improves code maintainability, reduces runtime errors, and allows working with large amounts of data across the app. Strong integration in the `Next.js` ecosystem solidified the choice further.

1.2 Database and ORM

For data persistence, "GalleryGlobe" utilizes PostgreSQL as its database mainly for its querying capabilities. The interactions with the database are handled by a type-safe ORM library - Drizzle ORM. It allows easy, fast, and safe access to the data, converting TypeScript methods to the SQL queries in one way, and SQL responses to the TypeScript objects in the other. Also, Drizzle ORM handles schemas, allowing define, modify, and migrate database tables.

1.3 Authentication and State Management

User authentication for "GalleryGlobe" is handled by NextAuth.js. This library provides a secure authentication solution. Its integration with Drizzle ORM via drizzle-adapter was a key

factor. This library integrates flawlessly into the app data flow, provides JWT-based session management, and middleware handling.

1.4 UI Components and Utilities

Styles in “GalleryGlobe”, are controlled by Tailwind CSS. It is a CSS framework, that allows building custom user interfaces with a list of ”utility” CSS classes that can be used to style each element by mixing and matching.

For reusable atomic web elements, Shadcn UI components set was used. It integrates into the code base directly, and the main advantage was its integration with already included style framework Tailwind CSS. Apart from that, key UI packages include:

- `lucide-react`: A collection of customizable open-source icons.
- `react-hook-form`: For form validation and schema definition.
- `embla-carousel-react`: For building the image carousel in the Hero section.

1.5 Development Tools

To standardize, normalize and implement best code structure practices, several several utility and development libraries were integrated in the project. They allow debugging, linting, and better development experience overall.

- `eslint` and `prettier`: For code linting and formatting, maintaining a consistent codebase.
- `drizzle-kit`: For enchance database schema management.
- `@t3-oss/env-nextjs`: For managing environment variables securely.
- `winston`: For custom logging.

Chapter 2

Source code file structure

The “GalleryGlobe” app implements recommendations, which are detailed on the official documentation page (see [7]). As a result, the project structure is rationally structured and modular. It separates client-side components, server-side logic, routing, and configurations.

2.1 Root Level & Configuration

The project root contains core configuration files and utilities that govern main functionalities.

- `auth.ts`: Centralized configuration for NextAuth.js. In this file, JWT-based session management and role-based access logic is declared.
- `drizzle.config.ts`: Configuration for Drizzle ORM, where the app sets up a connection with the hosted database using credentials.
- `logs/`: Directory for application logs, including both combined and error-specific .log files, managed by Winston.
- `routes.ts`: Application routing logic with defined public, authentication-specific and default redirect paths, used by middleware for navigation control.
- `schemas/` and `actions/`: Directories, that contain auth validation for both client components and server actions.
- Other configurations: Standard configuration files for Next.js (`next.config.js`), styling (`tailwind.config.ts`), code formatting (`prettier.config.js`), dependencies and scripts (`package.json`), environment variables (`.env`) and `package-lock.json`), and TypeScript (`tsconfig.json`)

2.2 `src/` Directory: Application Core

The primary app logic resides within the `src/` directory, as by the previously mentioned Next.js App Router conventions.

- `app/`: Core directory of the App Router, where application pages, routes, and layouts are defined.
  - `(protected)/`: Routes, that require user authentication.

  - `(public)/`: Publicly accessible routes.
  - `api/`: Next.js API routes and proxy for external WikiArt API interaction.
  - `admin/`, `artists/`, `artworks/`, `categories/`, `about/`, `auth/`: Specific route segments for displaying general pages, like artists, categories, authentication forms, etc.
  - `components/`: Custom components, each divided into it's component for reusability.
  - `page.tsx`, `layout.tsx`: Home page with core layout, that is inherited by all child routes.

- `components/`: Collection of previously mentioned reusable UI components.
- `env.js`: `.env` wrapper, that implements library `@t3-oss/env-nextjs` for safety and ease of use.
- `lib/`: Shared utility functions, helper modules, and TypeScript type definitions.
- `middleware.ts`: Next.js middleware, where the requests are handled for authentication checks.

Chapter 3

Code Documentation

This chapter provides a chronological walkthrough of the “GalleryGlobe” core functionalities and their interactions. logical pieces of code are separated into segments, each of which has a brief explanation.

3.1 Global Setup and Application Initialization

The application lifecycle starts with an npm script, declared in `package.json`. Next.js framework runs on Node.js; as such, the primary scripts are defined in `package.json` at the project’s root.

```
{
//...
  "scripts": {
    "build": "next build",
    "db:generate": "drizzle-kit generate",
    "db:migrate": "drizzle-kit migrate",
    "db:push": "drizzle-kit push",
    "db:studio": "drizzle-kit studio",
    "dev": "next dev",
    "lint": "next lint",
    "start": "next start",
    "test": "jest",
    "test:watch": "jest --watch"
  },
  "dependencies": {
  //...
  }
}
```

For running “GalleryGlobe” in development mode, one uses the CLI command `npm run dev`. As seen in the example, Next.js application is being initialized with a script that runs a Next.js CLI command `next dev`. This npm script, in this case, acts as a wrapper for the main command.

This is done for script standardization and consistency, for example, specifying server port for a configured environment with just a single line change within `package.json`.

The global application layout and metadata are configured in `layout.tsx`, establishing the basic structure for the whole project.

```tsx
import "~/styles/globals.css";

import { GeistSans } from "geist/font/sans";
import { type Metadata } from "next";
import Header from "~/components/common/header";
import Footer from "~/components/common/footer";
import { SessionProvider } from "next-auth/react";
import { auth } from "auth";
import { Toaster } from "~/components/ui/toaster";

export const metadata: Metadata = {
//...
};

export default async function RootLayout({
  children,
}: Readonly<{ children: React.ReactNode }>) {
  const session = await auth();

  return (
    <SessionProvider session={session}>
      <html lang="en" className={`${GeistSans.variable}`}>
        <body>
          <Header />
          <main>{children}</main>
          <Toaster />
          <Footer />
        </body>
</html>
    </SessionProvider>
  );
}
```

`src/app/layout.tsx` defines the root layout of the "GalleryGlobe" application. This file encapsulates shared UI elements, that are rendered across all pages, such as the `Header` and `Footer` components. It also sets metadata for SEO, and, crucially, wraps the entire application within `SessionProvider` from NextAuth.js. It recieves the asynchronously fetched user session using `await auth()`. This setup provides the user's authentication status, allowing role-based and authentication-based logic to be implemented within the project.

Apart from that, `Toaster` component indicates, that content-based notifications inside the application will be rendered globally for the user. The `children` prop represents each individual page (`page.tsx`) with possible nested layouts. Because `layout.tsx` is located at the root of App Router, every page will inherit its parent's layout. For the child pages, additional file `layout.tsx` might be added, but it will not override nor delete the parent layout by default.

Additionally, the font "GeistSans" is imported and applied in that file. As you can see, it is passed as a value inside the `ClassName` attibute. This is the way style is added to the web componens using TailwindCSS - the previously mentioned style framework.

3.2 Authentication Middleware

"GalleryGlobe" uses middleware for security and access control. Middleware is a layer that allows creating functions that execute after a user's request is made and before the request is completed - in the middle of two processes. It's initialization starts extremely early, even before static files are delivered to the client's web browser.

```
// middleware.ts
import { auth } from "auth";
import {
  DEFAULT_LOGIN_REDIRECT,
  apiAuthPrefix,
  authRoutes,
  publicRoutes,
} from "routes";

const compiledPublicRoutes = publicRoutes.map(route => new RegExp(route));
const authRoutesSet = new Set(authRoutes);

export default auth((req) => {
  const { nextUrl } = req;
  // Basic validation and redirect count logic omitted for brevity (etc.)

  const isLoggedIn = !!req.auth;
  const isApiAuthRoute = nextUrl.pathname.startsWith(apiAuthPrefix);
  const isPublicRoute = compiledPublicRoutes.some(route => route.test(nextUrl.pathname));
  const isAuthRoute = authRoutesSet.has(nextUrl.pathname);

  if (isApiAuthRoute) {
    return; // Allow API authentication routes to proceed
  }

  if (isAuthRoute) {
    if (isLoggedIn) {
      return Response.redirect(new URL(DEFAULT_LOGIN_REDIRECT, nextUrl));
    }
    return; // Allow unauthenticated users to access auth pages
  }

  // Handle protected routes: redirect unauthenticated users from non-public pages
  if (!isLoggedIn && !isPublicRoute) {
    return Response.redirect(new URL("/auth/login", nextUrl));
  }

  return; // Allow request to proceed
});

export const config = {
  matcher: [
    ↪  "/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|ttf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*
    "/(api|trpc)(.*)",
  ],
};
```

The `src/middleware.ts` file acts as the central gatekeeper for all incoming requests in the Next.js application. Inside, the `auth()` function from NextAuth.js is used to check the authen-

tication status of the user (`isLoggedIn`). This middleware divides all paths into three types of routes: `publicRoutes`, `authRoutes`, and `apiAuthPrefix`.The rules for this division are obtained from `route.ts`, where they are implemented as arrays of regular expressions (RegExes), with each expression corresponding to one or several routes within the application. The core logic ensures:

- API authentication routes are allowed to proceed without interruptions.
- Authenticated users are redirected away from authentication-specific pages, like login and register to their default login redirect path (`/profile`).
- Unauthenticated users that are trying to access protected routes are redirected to the login page.
- The omitted parts of the code include logic for the management of edge cases, comprehensive error handling, and other methods that allow the middleware to work flawlessly and to prevent common navigation bugs.

The `config.matcher` property specifies which paths the middleware should intercept. It allows for efficient adjustment the scope of middleware, keeping the application secure.

One can say that middleware in 'GalleryGlobe' possesses only functionality that is strongly recommended to be placed between sending a request and handling its response. As Sam Newman said in his book "Building Microservices, 2nd edition" - "Make sure you know what you're getting: keep your middleware dumb" (see [6]).

3.3 NextAuth.js Configuration and Session Management

The `auth.ts` file configures the NextAuth.js library, integrating it with the database and defining main rules for user sessions management.

```
import NextAuth, { type DefaultSession } from "next-auth";
import { DrizzleAdapter } from "@auth/drizzle-adapter";
import { db } from "~/server/db";
import { getUserById, getUserByEmail } from "~/server/db/queries/user-queries";
import { LoginSchema } from "schemas"; // Zod schema for validation
import bcrypt from "bcryptjs"; // For password hashing comparison

import { users } from "~/server/db/schema";
import { eq } from "drizzle-orm";

// ... (Type declarations for session and JWT)

export const { auth, handlers, signIn, signOut } = NextAuth({
  pages: {
    signIn: "/auth/login",
    error: "/auth/error",
  },
//...

```

The `/auth.ts` file handles the core configuration of application's authentication system. The sign-in and error pages are defined there, to ensure a consistent user experience during authentication process.

```
//... (email verificaion logic)
callbacks: {
  async session({ token, session }) {
    // Add user ID, role, name, email, picture to session object
    if (token.sub && session.user) {
      session.user.id = token.sub;
    }
    if (session.user && (token.role === "ADMIN" || token.role === "USER")) {
      session.user.role = token.role;
    }
    // ... (other assignments from token to session)
    return session;
  },
  async jwt({ token, user, trigger, session }) {
    // Update JWT with user data and role from DB
    if (user) { /* ... */ }
    // Fetch user role if not already in token (e.g., on first login)
    if (token.sub && !token.role) { /* ... */ }
    // Handle session updates (e.g., profile changes)
    if (trigger === "update" && session) { /* ... */ }
    return token;
  },
},
//...
```

This section customizes the logic of `session` and `jwt` tokens. Here, custom user information is getting injected into both session and token. In our case, it's user's role (ADMIN or USER). This step is crucial for role-based access logic in "GalleryGlobe". With this information, application desides, what content should and should not be accessible to the user. It also ensures that data in the token is refreshed upon profile updates.

```
//...
  adapter: DrizzleAdapter(db),
  session: { strategy: "jwt" },
//...
```

`DrizzleAdapter` is declared as the adapter for NextAuth.js. It allows authentication securely and seamlessly connect with PostgreSQL via Drizzle ORM, which allows manage user accounts directly in the database.

`session: { strategy: "jwt" }` configures JSON Web Tokens for session management. JSON Web Tokens (JWT) are compact and self-contained tokens, often used for single sign-on (SSO) scenarios. They can include user information and custom claims. JWT is stateless, making it suitable for scaling, and is widely used for securing APIs (see [9]).

```
  providers: [
    Credentials({
      async authorize(credentials) {
```

```
        const validatedFields = LoginSchema.safeParse(credentials); // Validate input with Zod
        if (validatedFields.success) {
          const { email, password } = validatedFields.data;
          const user = await getUserByEmail(email); // Fetch user from DB
          if (!user || !user.password) return null;
          const passwordsMatch = await bcrypt.compare(password, user.password); // Compare hashed
          ↪  password
          if (passwordsMatch) {
            return {
              id: user.id, name: user.name, email: user.email,
              image: user.image, role: user.role,
            };
          }
        }
        return null;
      },
    }),
  ],
});
```

Inside `providers` section, custom log in mechanism is defined. It utilizes `LoginSchema` for server-side validation of user credentials. `zod` library introduces a secure mechanism to parse data from SQL database into type-safe TypeScript objects. `bcryptjs` library securely encripts and compares passwords against hashed values, saved in the database. Upon successful validation and password match, a user object containing relevant details (ID, name, email, role) is returned, establishing the authenticated session. For security reasons, passowrd in it's original format exists close to no time inside the system - encryption is a priority. This is how "Gallery-Globe" is not prone to password leaks.

3.4 Main Page Data Flow and UI Rendering

The `Main` page serves as a prime illustration of the data fetching and UI rendering patterns, showcasing how the client-side components interact with server-side data fetching and processing logic to display dynamic context.

At its code, React has a component-based architecture. It allows code reusability, maintainability and overall code organization. However, a fully client-side React application is prone to slow load times, complex data fetching processses, where requests are tightly coupled and dependent on each other.

This is where Next.js steps in. It acts as a full-stack React framework, that implements solutions to most of React problems. Next.js introduces hybrid rendering capabilities, which guides application on how and when components should be rendered.

```
// src/app/page.tsx
"use client"; // Marks component for client-side rendering
import React from "react";
import { Artwork } from "~/lib/types/artwork"; // Type definition for Artwork
import { Loading } from "~/components/ui/loading"; // UI for loading state
import Hero from "~/components/home/hero"; // Hero section component
import ArtworkTiles from "~/components/home/arwork-tiles"; // Artwork display component
import { fetchArtworks } from "~/server/actions/data_fetching/fetch-artworks-home"; // Server Action
```

```tsx
import useSWR from "swr"; // SWR hook for data fetching

const MainPage = () => {
  const {
    data: artworks,
    isLoading,
    error
  } = useSWR<Artwork[]>('artworks', fetchArtworks, {
    revalidateOnFocus: false, // Prevents revalidation on window focus
    dedupingInterval: 60 * 1000, // Caches data for 60 seconds
    onError: (error) => {
      console.error("Error loading artworks:", error);
    }
  });

  return (
    <>
      {isLoading && <Loading />} {/* Show loading spinner */}
      <div className="min-h-screen bg-gradient-to-b from-gray-50 to-white">
        {error ? (
          <div className="text-center text-red-600 p-4">
            Failed to load artworks. Please try again later.
          </div>
        ) : (
          <>
            <Hero artworks={artworks ?? []} />
            <ArtworkTiles artworks={artworks ?? []} />
          </>
        )}
      </div>
    </>
  );
};

export default MainPage;
```

In `src/app/page.tsx`, component, that acts as a main page for my application. While the `MainPage` itself is marked with `"use client"`, which points to the fact, that application will handle it as a client-side component, that shoul be rendered in the user's browser, Next.js's underlying architecture still provides a superior structure.

Consider artwork fetching logic, at line `16`. Here, `useSWR` is a client-side hook for data fetching. Inside, core function `fetchArtworks`, which originates from the server-side part of application. As detailed in Chapter 2, this function is allowed to be executed directly on the server as a Next.js server action. This connection to server-side code execution means, that:

- `fetchArtworks`, when called, is not making an additional HTTP request to a separate API endpoint, as in the traditional client-server system designs. Instead, it invokes server-side code directly. As described in Chapter 1.2, "it can handle interactions with the database...by Drizzle ORM". This paradigm shift reduces network lagency and improves speed of data retrieval.

- `Artwork[]` type annotation inside the `useSWR` hook shows, that it is a direct benefit of using TypeScript. With strong typing, application knows that data, either it is retrieved from the database or a third party's API, will have a defined set of properties. With that, one

can have expectations, that the objects will be normalized and could build functionality with clear expectations for data, that a function will work with.

The SWR configuration includes `revalidateOnFocus: false` and `dedupingInterval`. It prevents unnecessary re-fetches and caches data, and therefore optimize the data retrieval. The component conditionally renders a loading spinner (`Loading`) or an error message based on the data fetching state. If the data is retrieved and correct, it renders the `Hero` and `ArtworkTiles` components, passing the fetched artworks data to them for display.

3.5 External API Integration and Data Fetching Logic

The `fetchArtworks` Server Action is used to get artwork data for the main page of the application. It both demonstrates interaction with an external API and error handling.

```
//
import { Artwork } from "~/lib/types/artwork"; // Type definition
// ... (imports for getRandomArtist, fetchWikiArtApi, FALLBACK_DATA, shuffleArray,
    processArtwork, MAX_ARTWORKS)

export async function fetchArtworks(): Promise<Artwork[]> {
  try {
    const artist = await getRandomArtist(); // Select a random artist

    const artworks = await fetchWikiArtApi<Artwork[]>(
      `/App/Painting/PaintingsByArtist?artistUrl=${artist.url}&json=2`,
    ); // Fetch artworks from WikiArt API

    if (!artworks || artworks.length === 0) {
      console.warn(`No artworks found for artist ${artist.artistName}, using fallback data`);
      return FALLBACK_DATA.artworks; // Return fallback data if no artworks found
    }

    // Ensure we have a diverse selection by shuffling and limit to maximum count
    const processedArtworks = shuffleArray(artworks)
      .slice(0, MAX_ARTWORKS)
      .map(processArtwork); // Process and limit artworks

    console.log(`Successfully fetched ${processedArtworks.length} artworks by ${artist.artistName}`);

    return processedArtworks;
  } catch (error) {
    console.error("Error fetching artworks collection:", error instanceof Error ? error.message :
    String(error));
    return FALLBACK_DATA.artworks; // Return fallback data on error
  }
}
```

In file `src/server/actions/data_fetching/fetch-artworks-home.ts` the function `fetchArtworks` is located. It is, as previously mentioned, a Server Action, designed to be executed on the server. Firstly, it calls a asynchronous function `getRandomArtist()`, that returns a random popular artist data. Secondly, the request is done to the WikiArt API to retrieve selected artist's artworks via `fetchWikiArtApi()` helper. After that, the response data is getting processed in several steps, to ensure it's correctness and suitability to be used further. At the end,

processed array of artworks is shuffled and and returned. if no artworks are found or an error occurs during fetching, a FALLBACK_DATA.artworks array is returned, ensuring the application remains functional even in the event of API issues. //

The interaction with the external WikiArt API is encapsulated in the fetchWikiArtApi utility.

```
// src/server/actions/data_fetching/fetch-wikiart-api.ts (conceptual path)
const cache = new Map<string, { data: any; timestamp: number }>();
const CACHE_TTL = 5 * 60 * 1000; // 5 minutes cache

export async function fetchWikiArtApi<T>(endpoint: string): Promise<T> {
  const controller = new AbortController();
  const timeoutId = setTimeout(() => controller.abort(), 10000); // 10s timeout

  try {
    const cacheKey = endpoint;
    const now = Date.now();
    const cachedItem = cache.get(cacheKey);

    if (cachedItem && now - cachedItem.timestamp < CACHE_TTL) {
      return cachedItem.data as T; // Serve from cache if valid
    }

    const url = new URL(`https://www.wikiart.org/en${endpoint}`);
    const headers: HeadersInit = { /* ... */ }; // User-Agent and Accept headers

    const response = await fetch(url.toString(), {
      signal: controller.signal,
      headers,
    });

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();
    cache.set(cacheKey, { data, timestamp: now }); // Cache successful response
    return data as T;
  } catch (error) {
    // ... (error handling for timeout or other fetch errors)
    throw error;
  } finally {
    clearTimeout(timeoutId);
  }
}
```

A pivotal feature of previously mentioned hook is its in-memory caching mechanism. Responses are stored in a `CACHE_TTL` (5 minutes), significanly lowering number of reduntant API calls and improving performance. It also implements a request timeout using `AbortController`, preventing long-running requests from blocking the application The function includes error handling for network issues or non-successful HTTP responses. It makes page more reliable.

3.6 Frontend UI Components: Hero and Artwork Display

After the data is processed and recieved in out `MainPage` component, it's being passed through in two main components - `Hero` and ArtworkTiles

3.6.1 Hero Component

```tsx
"use client"; // Marks component for client-side rendering
import { Artwork } from "~/lib/types/artwork";
import Image from "next/image";
import { useCallback, useEffect, useState } from "react";
// ... (imports for Button, ArrowRight, useRouter)

interface HeroProps {
  artworks: Artwork[];
}

export default function Hero({ artworks }: HeroProps) {
  const [activeSlide, setActiveSlide] = useState(0);
  // ... (state for heroImagesLoaded, router)

  const handleHeroImageLoad = useCallback(() => { /* ... */ }, []);
  useEffect(() => {
    if (artworks.length === 0) return;
    const timer = setInterval(() => {
      setActiveSlide((current) =>
        current === artworks.length - 1 ? 0 : current + 1,
      );
    }, 7000); // Auto-slides every 7 seconds
    return () => clearInterval(timer);
  }, [artworks.length]);

  return (
    <section className="relative h-[80vh] bg-black">
      <div className="absolute inset-0 z-10 bg-black/40" />
      <div className="relative h-full">
        {artworks.map((artwork, index) => (
          <div
            key={artwork.contentId}
            className={`absolute inset-0 transition-opacity duration-1000 ${
              index === activeSlide ? "opacity-100" : "opacity-0"
            }`}
          >
            <Image src={artwork.image} alt={artwork.title} fill className="object-cover"
              // ... (image loading and priority props)
            />
          </div>
        ))}
      </div>
      <div className="absolute inset-0 z-20 flex items-center justify-center">
      // ... (layout and styling)
            <Button onClick={() => router.push("/artworks")} variant="outline" size="lg">
              Browse Gallery <ArrowRight className="ml-2 h-4 w-4" />
            </Button>
      // ...
      </div>
    </section>
  );
}
```

All custom-made components of “GalleryGlobe” are located at `src/components/` directory. `Hero` component is not an exception, as its location is `src/components/home/hero.tsx`.

`Hero` is a client-side React component responsible for displaying the singlemost important part of “GalleryGlobe” - it’s carousel at the top of main page. Carousel in this application is a rotating banner of featured artworks. Artworks are put in sequence, to ensure, that each artwork shows before coming back to the first entry. It is handled by a `activeSlide` state that controls the artwork display logic. An useEffect hook implements an automatic slide transition every 7 seconds.

In this component, a previously mentioned crucial component is implemented - Next.js’s `Image`. It is used for optimized image loading, including `fill` for responsive sizing and `priority` for critical images. This boolean attribute tells the browser, that this is a critical resource, and that it should be loaded as soon as possible, before they are discovered in HTML. It’s used to reduce the time, needed for the page to be prepared for a display (see [8]). User can interact on this component with “Browse Gallery” button that navigates to the artworks page using `useRouter`.

3.6.2 Artwork Tiles Component

```
”use client”;
import { useState } from ”react”;
// ... other imports
import { Skeleton } from ”../ui/skeleton”;

interface ArtworkTilesProps {
  artworks: Artwork[];
}

export default function ArtworkTiles({ artworks }: ArtworkTilesProps) {
  const [imageStates, setImageStates] = useState<Record<string, {
    loading: boolean;
    error: boolean;
    retries: number;
  }>>({});

  const MAX_RETRIES = 3;

  const handleImageLoad = (artworkId: number) => { /* ... */ }; // Handles image successful loading
  const handleImageError = (artworkId: number) => { /* ... */ }; // Handles image loading errors
  const retryImage = (artworkId: number) => { /* ... */ }; // Manages image retry attempts

  return (
    <section className=”py-20”>
      <div className=”container mx-auto px-4”>
        <h2 className=”mb-12 text-center text-3xl font-bold”>
          Todays focus - {artworks[0]?.artistName}
        </h2>
        <div className=”grid grid-cols-1 gap-8 md:grid-cols-2 lg:grid-cols-3”>
          {artworks.map((artwork, index) => (
            <Link
              key={artwork.contentId}
              href={`/artworks/${artwork.contentId}`}
              className=”transition-transform hover:scale-[1.02]”
            >
              <Card className=”overflow-hidden”>
                <div className=”relative aspect-[3/4]”>
                  {imageStates[artwork.contentId]?.error ? (
                    // ... error display and retry button
```

```
                ) : (
                  <>
                    {(!imageStates[artwork.contentId] || imageStates[artwork.contentId]?.loading) && (
                      // ... skeleton loader
                    )}
                    <Image
                      src={artwork.image}
                      alt={artwork.title}
                      fill
                      loading={index < 3 ? "eager" : "lazy"} // Eager load first 3, lazy load others
                      className={`object-cover transition-opacity duration-300 ${
                        imageStates[artwork.contentId]?.loading ? 'opacity-0' : 'opacity-100'
                      }`}
                      sizes="(min-width: 1024px) 33vw, (min-width: 768px) 50vw, 100vw"
                      quality={65}
                      onLoadStart={() => { /* ... */ }} // Initializes loading state
                      onLoad={() => handleImageLoad(artwork.contentId)}
                      onError={() => handleImageError(artwork.contentId)}
                    />
                  </>
                )}
              </div>
              <CardContent className="p-4">
                <h3 className="truncate font-semibold">{artwork.title}</h3>
                <p className="text-sm text-muted-foreground">{artwork.artistName}</p>
                <p className="text-sm text-muted-foreground">{artwork.yearAsString}</p>
              </CardContent>
            </Card>
          </Link>
        ))}
      </div>
    </div>
  </section>
);
}
```

The `ArtworkTiles` component from `src/components/artwork-tiles.tsx` is a client-side component responsible for displaying grid of artworks. This component implements sophisticated error handling and image loading logic. Artwork data, after being passed in this component from `MainPage`, gets paired inside `imageStates` with state variables.

In this component, state management is handled using `Record` utility type. In TypeScript, `Record` helps define object types where property keys come from one type (K) and property values from another (T). This is particularly useful for creating configuration objects. The `Record` utility type is structurally similar to index signatures in TypeScript. Both allow you to define an object type where the keys are of a specific type, and the values can be of another type (see [2]).

In this case, Record assigns an object with `loading`, `error`, and `retries` to each artwork, ensuring, that state management will be kept secured throughout the component's life cycle.

Each artwork's data gets formatted and then displayed, implementing custom styling choices and image loading optimization. Skeletons are shown during image loading to provide a better user experience. The component uses Next.js's Image component for performance, employing `loading="eager"` for initial visible images and `lazy` loading for others. Each artwork tile is wrapped in a Link component, enabling navigation to the detailed artwork page.

Chapter 4

Data Base

This chapter dives into the foundational data layer of “GalleryGlobe” application, outlining the database architecture, the chosen Object-Relational Mapper (ORM), and the detailed schema that acts as a core for system’s data operations. As Adam Jorgensen said in his book “Microsoft SQL Server 2012 Bible” - “Good database design makes life easier for anyone who touches the database. The database schema is the foundation of the database project; and an elegant, simple database design outperforms a complex database both for the development process and the final performance of the database application” (see [5]).

4.1 Database Technology: PostgreSQL

PostgreSQL was selected as the relational database system for “GalleryGlobe” due to its reputation as one of the most suitable for web development database management systems. It is known for:

- Data Integrity: PostgreSQL is maintainable and dependable technology because it adheres to ACID (Atomicity, Consistency, Isolation, Durability) properties (see [**dulay**]).
- Scalability: It is highly scalable. PostgreSQL is designed for handling large volumes of data and working under high loads.
- Community Support: A large and active community acts as a solid reason for PostgreSQL continuously improve.

Due to these reasons, PostgreSQL use in Next.js project is considered to be the default good practice. The Next.js framework itself if database-agnostic, meaning it doesn’t dictate which database one must use. However, for developers, who work on system with lots of interactions, complex queries and strong relationships, the use of PostgreSQL via Drizzle ORM is recommended.

4.2 Object-Relational Model: Drizzle ORM

To interact with the database, “GalleryGlobe” uses Drizzle ORM. As mentioned in Chapter 1.2, Drizzle ORM is an open-source modern ORM for TypeScript that provides type-safe and secure way to interact with databases. Fluent API and strong typing enchance developer productivity by providing compile-time validation and autocompletion. Also, Drizzle ORM generates efficient

SQL queries, which optimize interactions and handles every action - from schema definition to query execution - with end-to-end type-safety

4.3 Database Schema Definition

The database schema for “GalleryGlobe” is designed around centralized `user` entity, and acts as a hub for all user-specific data and interactions. This architectural pattern facilitates efficient data retrieval and simplifies data management by creating links between all user-related information and a single user profile. The schema is composed of several interconnected entities, each serving a distinct functional purpose. Schema is defined using Drizzle ORM’s syntax, as shown below:

```
import { relations, sql } from "drizzle-orm";
import {
 //(other drizzle imports)
} from "drizzle-orm/pg-core";
import { type AdapterAccount } from "next-auth/adapters";

export const createTable = pgTableCreator((name) => `${name}`);

export const userRoleEnum = pgEnum("user_role", ["ADMIN", "USER"]);

export const users = createTable("user", {
  id: varchar("id", { length: 255 })
    .notNull()
    .primaryKey()
    .$defaultFn(() => crypto.randomUUID()),
  name: varchar("name", { length: 255 }),
  email: varchar("email", { length: 255 }).notNull(),
  emailVerified: timestamp("email_verified", {
    mode: "date",
    withTimezone: true,
  }).default(sql`CURRENT_TIMESTAMP`),
  image: varchar("image", { length: 255 }),
  password: varchar("password", { length: 255 }),
  role: userRoleEnum("role").notNull().default("USER"),
});

export const usersRelations = relations(users, ({ many }) => ({
  accounts: many(accounts),
}));
```

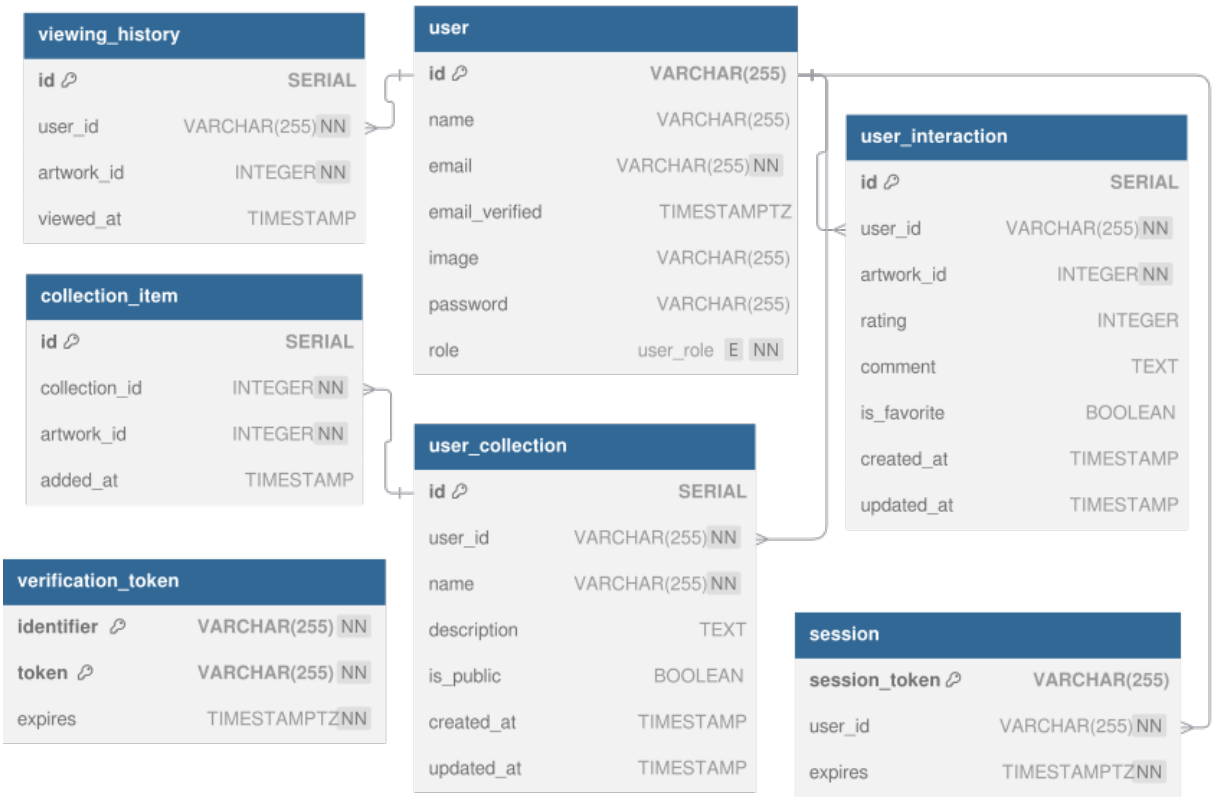
For furher comfort, explanations will go around `users` table. It is a crucial table, around which most of the functionalities of “GalleryGlobe” are focused. This table stores fundamental user attributes, such as id (primary key), name, email, and role.

- `id`: The user’s primary key, which is used to identify each user and acts as the foreign key in all related user-specific tables. It is generated using `crypto.randomUUID` - function, that returns Universally Unique Identifier.

  UUIDs are considered to be globally unique, meaning that data can be used even between different companies without thinking about possible dublicates. This confidence is caused

by the standards of initialization - it is a 128-bit label, and typically represented as a 36-character string, which includes both numberd and letters (see [10]).

- `name`: The user's display name.
- `email`: The user's email address, enforced as unique and non-nullable.
- `emailVerified`: A timestamp indicating when the user's email was verified, defaulting to the current timestamp upon creation.
- `image`: A URL to the user's profile picture.
- `password`: Stores the hashed password for users who sign up via credentials, ensuring secure authentication.
- `role`: Utilizes the custom userRoleEnum to define the user's permissions, defaulting to `"USER"` and allowing for `"ADMIN"` roles.



Relationships with the `users` table are defined as follows:

- `accounts` (One-to-Many): A single user can be associated with multiple external authentication accounts (e.g., email/password, Google, GitHub). The `accounts` table contains a `userId` foreign key that references `users.id`, with an `onDelete: "cascade"` rule ensuring that if a user is deleted, all their associated accounts are also removed.
- `userPreferences` (One-to-Many, effectively One-to-One in typical usage): Each user can have a record in the `user_preferences` table, storing personalized settings such as preferred art styles or historical periods. This link is established through a `userId` foreign key.

- `viewingHistory` (One-to-Many): Users can have numerous entries in the `viewing_history` table, which tracks the artworks they've viewed. Each entry includes a `userId` foreign key linking back to the user.
- `userCollections` (One-to-Many): Users can create multiple custom art collections, with each collection linked to its creator via a `userId` foreign key in the `user_collection` table.
- userInteractions (One-to-Many): This table records various user engagements with artworks, such as ratings or marking an artwork as a favorite. Each interaction is tied back to a specific `userId`.

In addition to preserving strong data integrity and facilitating effective querying of user-centric data across the application, these relationships are essential for providing user-specific features like customized collections, historical tracking, and tailored content.

Chapter 5

Project Initialization and Setup

The first step involves preparing the workspace. Start by opening **File Explorer** (e.g., via `Win+E` or desktop shortcut) and navigating to your preferred disk, such as Disk C. Once there, create a new folder for your project, naming it `test_1`. This folder will serve as the root directory for your new application.

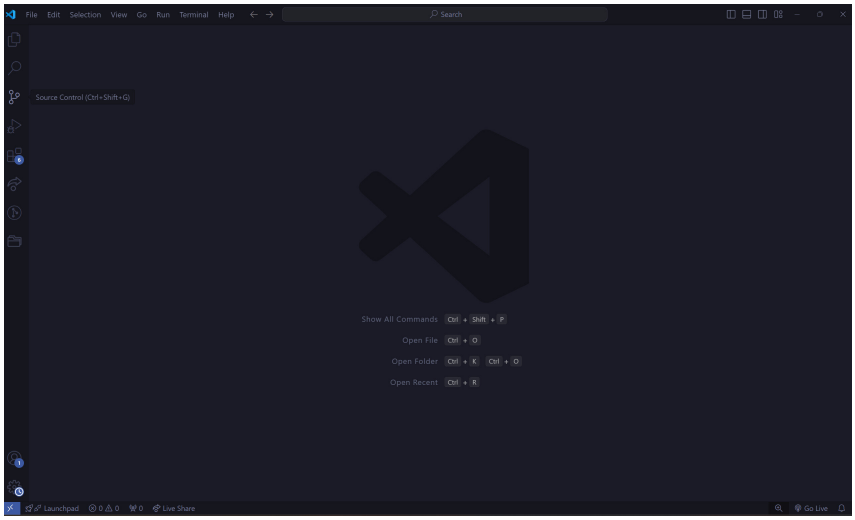


Figure 5.1: Creating the initial project folder `test_1` in File Explorer.

Next, launch **Visual Studio Code** (VS Code), your preferred code editor. After opening VS Code, navigate to **File** → **Open Folder** (or use `Ctrl+K, Ctrl+O`) and select the newly created `test_1` folder. This action loads your project into the editor, preparing it for development.

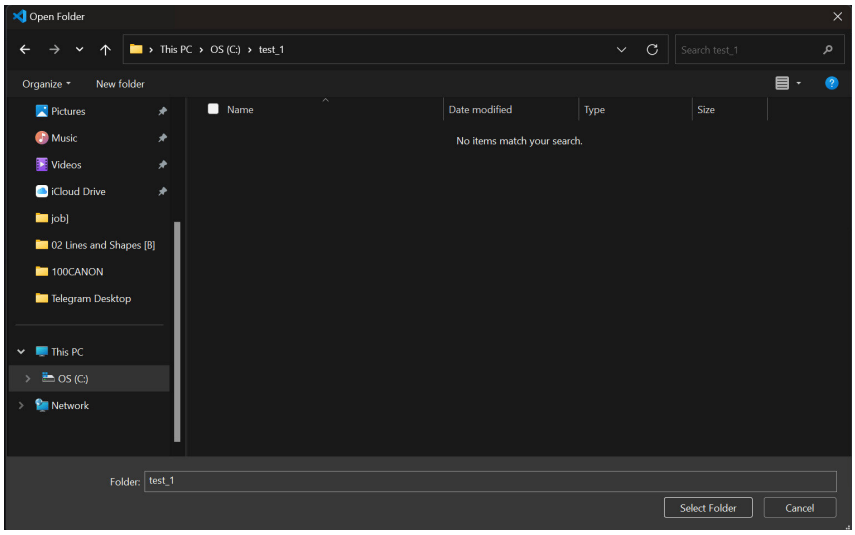


Figure 5.2: Opening the `test_1` project folder in Visual Studio Code.

A crucial dependency for Next.js is Node.js. Download the installer from the official Node.js website (`https://nodejs.org/en`), specifically the recommended LTS (Long Term Support) version. Once downloaded, run the installer, accepting the license agreement and using the default installation settings. This process will install both Node.js and npm (Node Package Manager) on your system (see [3]).

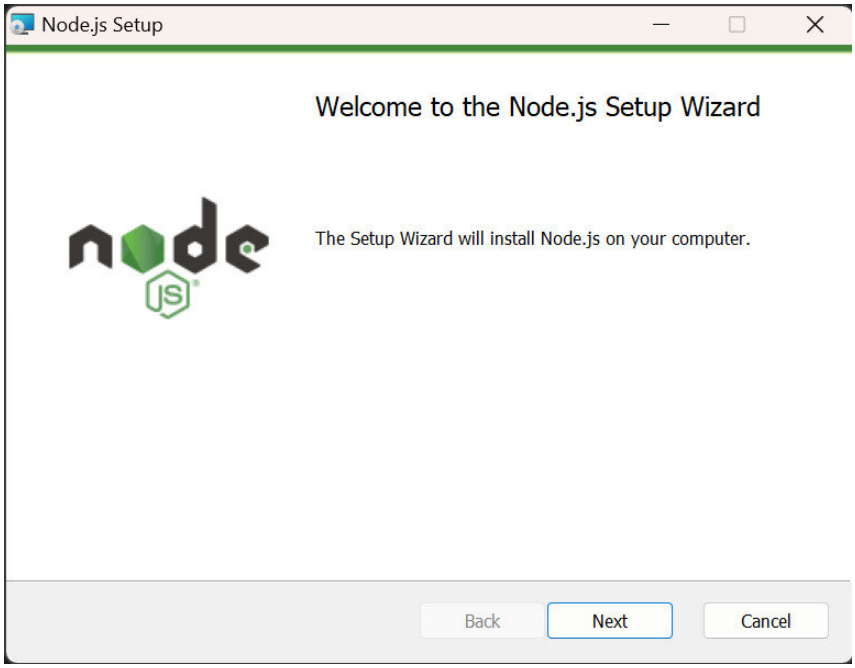


Figure 5.3: Running the Node.js installer with default settings.

Confirm that Node.js and npm are correctly installed by opening Visual Studio Code's integrated terminal (`Ctrl+Shift+`). Execute the commands `node -v` and `npm -v`. A successful output displaying version numbers indicates the environment is ready.

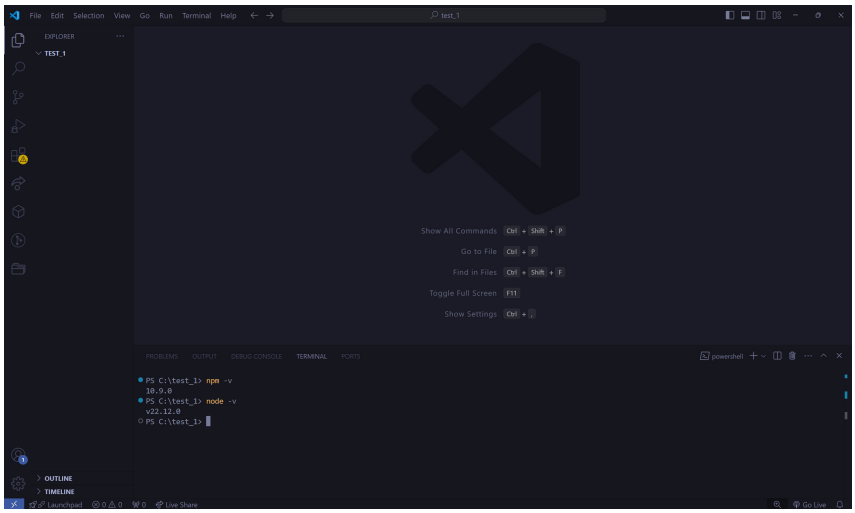


Figure 5.4: Verifying Node.js and npm versions in the VS Code integrated terminal.

Now, create the Next.js application using the command: `npx create-next-app@latest`. During the interactive setup, configure the project as follows: name it `test_1`, select `Yes` for **TypeScript** and to use an **App Router** with a `src/` directory, and `No` for ESLint, Tailwind CSS, Turbopack, and import alias customization. The setup process will then install all necessary project dependencies.

```
PS C:\test_1> npx create-next-app@latest
Need to install the following packages:
create-next-app@15.1.0
Ok to proceed? (y) y

√ What is your project named? ... test_1
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like your code inside a `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
√ Would you like to use Turbopack for `next dev`? ... No / Yes
√ Would you like to customize the import alias (`@/*` by default)? ... No / Yes
Creating a new Next.js app in C:\test_1\test_1.

Using npm.

Initializing project with template: app


Installing dependencies:
- react
- react-dom
- next

Installing devDependencies:
- typescript
- @types/node
- @types/react
- @types/react-dom
```

Figure 5.5: Creating a new Next.js application via `npx create-next-app@latest`.

With the application generated, navigate into your new project directory within the terminal by typing `cd .\test_1`. Once inside, start the development server by running the command `npm run dev`. This compiles and launches your Next.js application locally.

Figure 5.6: Navigating to the project directory and starting the Next.js development server.

Finally, access your running Next.js project. In the terminal output, locate the local HTTP address (e.g., `http://localhost:3000`). `Ctrl+Click` this link to automatically open your web browser, which will display the default Next.js starting page, confirming your project's successful setup and execution.

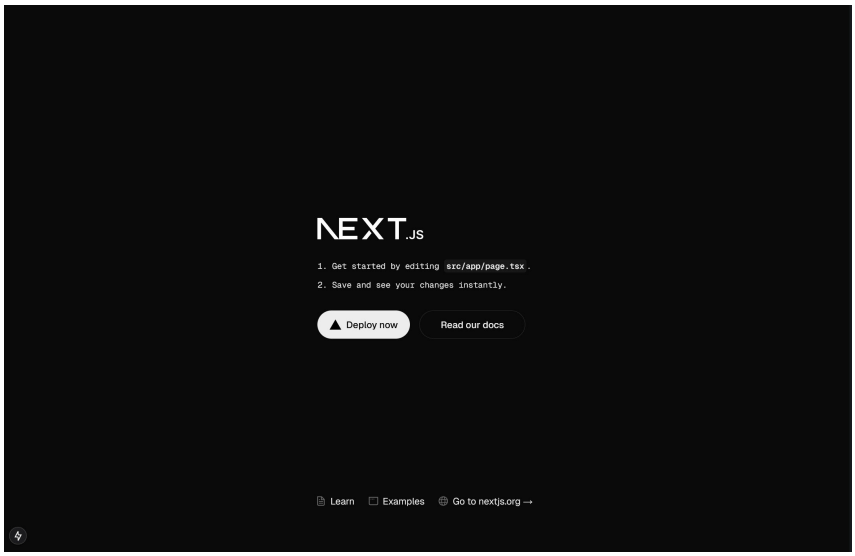


Figure 5.7: Accessing the newly created Next.js application in a web browser.

Chapter 6

User Documentation

This chapter provides a chronological walkthrough of the “GalleryGlobe” application’s operation from the user’s perspective. It serves as an extended user manual, detailing various activities users can perform and illustrating the interface with relevant screenshots.

6.1 Getting Started: Home Page and Navigation

Upon first accessing “GalleryGlobe”, users are presented with the Home Page, which serves as the central hub for content discovery and navigation.

6.1.1 Header Navigation

The application’s header, consistently displayed at the top of most pages, provides primary navigation links and access to user-specific functionalities.

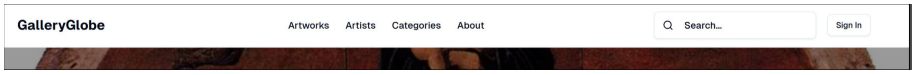


Figure 6.1: The application header for logged-out users, showing main navigation and sign-in option.

The header (Figure 6.1) features the “GalleryGlobe” logo, acting as a direct link back to the Home Page. Key content categories are accessible via “Artworks”, “Artists”, and “Categories” links. The “About” link provides information about the application. A prominent search bar allows users to quickly find specific content. For unauthenticated users, a “Sign in” button is visible, leading to the authentication flow.

6.1.2 Hero Section

The Hero Section is the most prominent visual element on the Home Page, designed to immediately capture user attention with dynamic content.

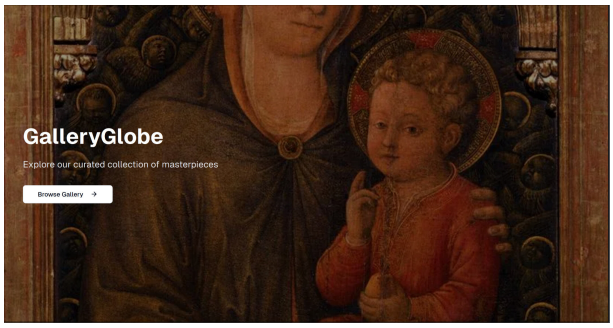


Figure 6.2: The dynamic Hero Section on the Home Page, featuring rotating artworks and a call to action.

As shown in Figure 6.2, the Hero Section features a dynamically changing background composed of featured artworks. This visual element provides an immersive introduction to the gallery's content. A "Browse Gallery" button is centrally located, serving as a primary call to action for users to explore the full collection of artworks. A search bar is also integrated within this section, offering another immediate entry point for content discovery.

6.1.3 Featured Artist Section

Below the Hero Section, the Home Page highlights a featured artist, showcasing a selection of their works.

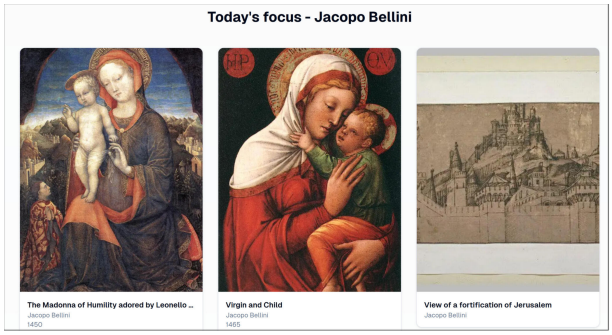


Figure 6.3: The Featured Artist Section on the Home Page, displaying an artist's name and a selection of their artworks.

Figure 6.3 illustrates the Featured Artist Section. This area prominently displays the name of a selected artist, accompanied by a tiled grid of their artworks. This section aims to introduce users to diverse artists and encourage deeper exploration of their portfolios by clicking on individual artwork tiles.

6.2 Authentication: Signing In and Registering

Users can create an account or sign in to access personalized features such as collections, favorites, and viewing history.

6.2.1 Signing In (Logged Out State)

When a user is not logged in, the header provides a clear option to initiate the authentication process.

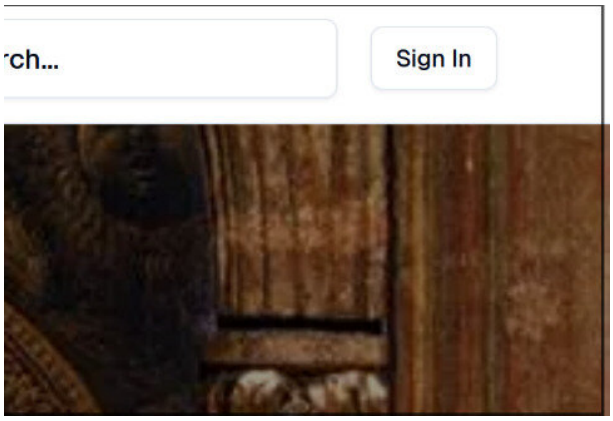


Figure 6.4: The 'Sign in' button in the header for unauthenticated users.

Clicking the "Sign in" button (Figure 6.4) in the header navigates the user to the Login Page.

Login Page

The Login Page provides the interface for existing users to authenticate their access to the application.

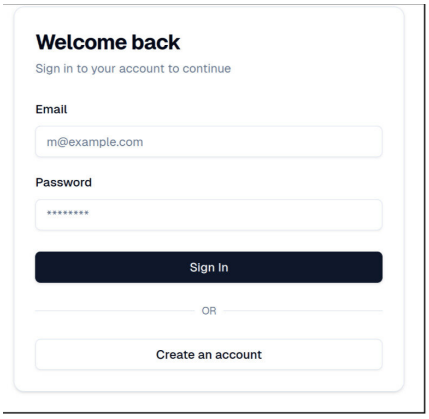


Figure 6.5: The Login Page, showing the sign-in form.

As depicted in Figure 6.5, the Login Page includes the standard application header for consistent navigation. The primary element is the Sign In Form, where users enter their registered email address and password to gain access.

Registration Page Access

From the Login Page, users who do not yet have an account can navigate to the Registration Page.

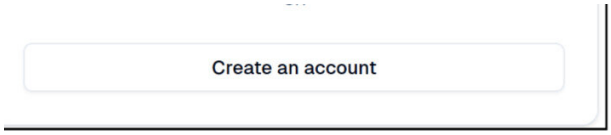


Figure 6.6: Link from the Login Page to the Registration Page.

A link, as shown in Figure 6.6, is provided on the Login Page to direct new users to the Registration Page.

Registration Page

The Registration Page allows new users to create an account within “GalleryGlobe”.

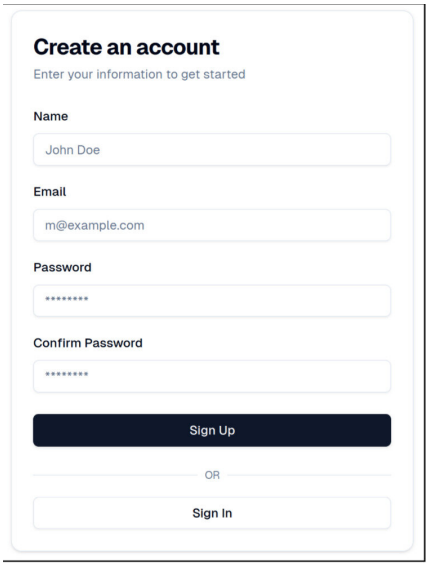


Figure 6.7: The Registration Page, featuring the sign-up form.

Figure 6.7 displays the Registration Page, which also maintains the consistent application header. The main component is the Sign Up Form, requiring users to provide a name, email, password, and confirm their password to create a new account. A link back to the Login Page is also available for users who might already have an account or decide not to register.

6.3 Authenticated User Experience: Account Management

Once logged in, the “Sign in” button in the header is replaced by an Account Dropdown, providing access to personalized features and account management options.

6.3.1 Account Dropdown

The Account Dropdown provides quick access to user information and a menu of personalized functionalities.

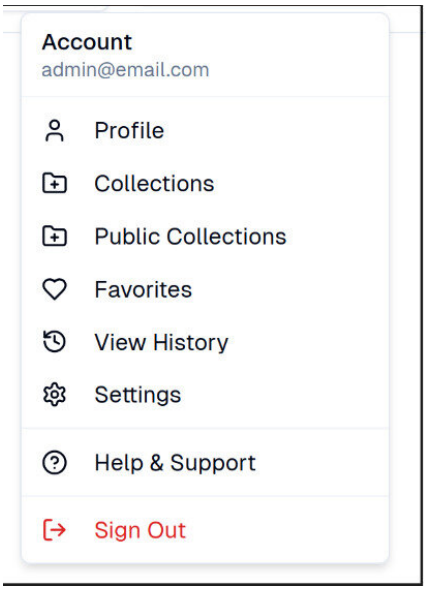


Figure 6.8: The header showing the Account Dropdown for logged-in users.

As seen in Figure 6.8, clicking on the user's avatar or name in the header reveals the Account Dropdown. This dropdown prominently displays the user's Username and Email, offering immediate confirmation of the logged-in account. Below this information, a series of controls provides navigation to various user-specific pages.

6.3.2 Profile Page

The Profile Page serves as the central hub for managing user account details and security settings.

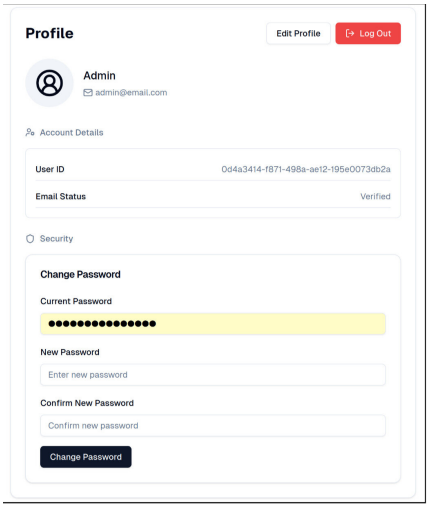


Figure 6.9: The Profile Page, showing account information and options to edit profile or manage security.

Accessed via the Account Dropdown, the Profile Page (Figure 6.9) maintains the consistent application header. It reiterates the user's Account Info (Username and Email), mirroring the information in the dropdown. This page offers two primary functionalities: Edit Profile and User Security.

Edit Profile

The Edit Profile section allows users to update their personal information.

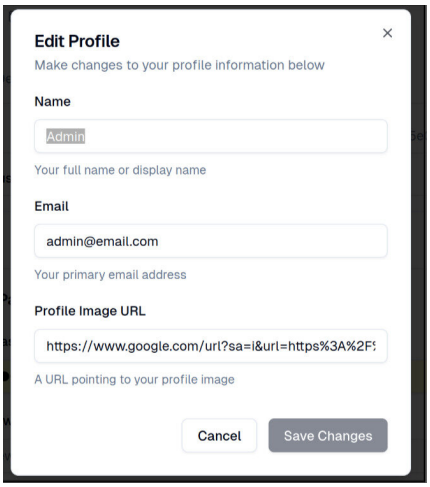


Figure 6.10: The 'Edit Profile' form, allowing users to update their name, email, and profile image.

Within the Edit Profile section (Figure 6.10), users can modify their New Name, New Email, and upload a New Profile Image. After making changes, users can save their updated profile.

User Security

The User Security section provides options for managing account security, primarily changing the user's password.

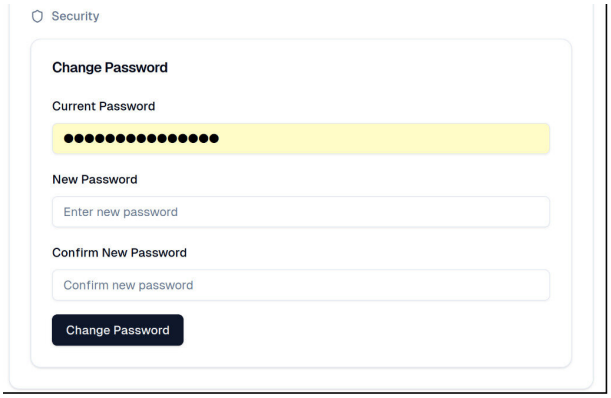


Figure 6.11: The 'Change Password' interface within the User Security section of the Profile Page.

As shown in Figure 6.11, the Change Password feature allows users to update their account password, enhancing account security. This section typically requires the old password for verification before setting a new one.

Log Out

Users can securely sign out of their account from the Profile Page.

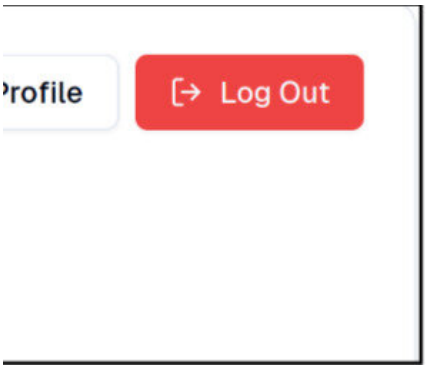


Figure 6.12: The 'Log Out' button on the Profile Page.

The Log Out button (Figure 6.12) provides a direct way for users to end their authenticated session, returning them to the logged-out state of the application.

6.3.3 Collections

The Collections feature allows users to organize and manage their favorite artworks into custom collections, which can be private or public.

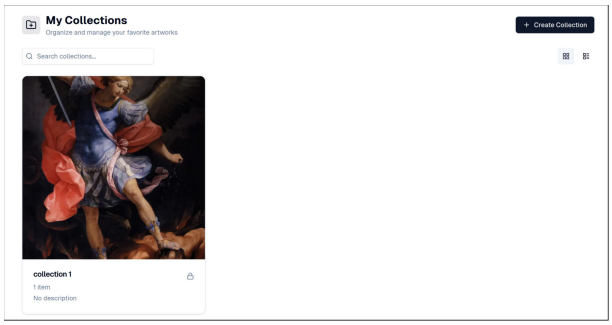


Figure 6.13: The Collections page, showing options for private and public collections, and management tools.

Accessed via the Account Dropdown, the Collections page (Figure 6.13) provides an overview of user-created collections. It maintains the consistent application header. Users can manage their Private Collections and explore Public Collections. Tools for Searching Collections and Creating a New Collection are also available.

Creating a Collection

Users can create new collections to categorize artworks.

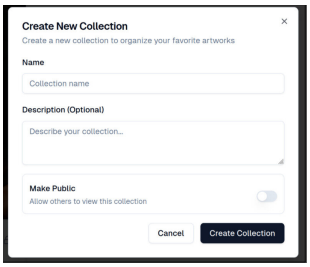


Figure 6.14: The ’Create Collection’ dialog, allowing users to define a new collection’s properties.

When creating a collection (Figure 6.14), users specify its Name and an optional Description. They also have the option to make the collection Public, allowing other users to view its contents.

Collection Display and Layout

Collections can be viewed in different layouts to suit user preference.

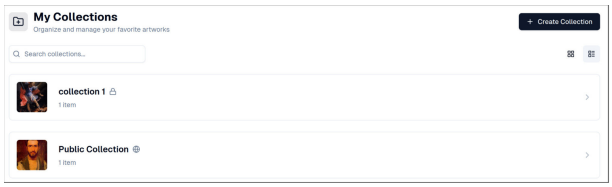


Figure 6.15: Options for changing the layout of collections (Tiling Layout and Inline Layout).

Users can change the display of their collections using the Change Layout options (Figure 6.15), typically offering a Tiling Layout for a grid view and an Inline Layout for a list view.

Private Collections Overview

The Private Collections section lists all collections created by the current user.

Figure 6.16: List of private collections, showing collection name and number of items.

Each private collection (Figure 6.16) is displayed with its Collection Name and the Number of Items it contains, providing a quick overview of its contents.

Individual Collection View

Clicking on a specific collection reveals its detailed contents.

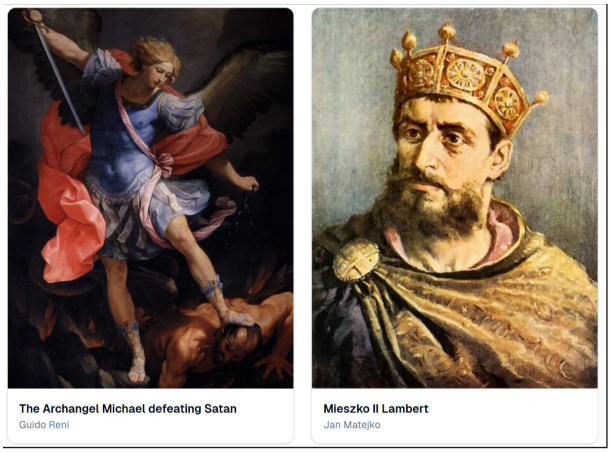


Figure 6.17: Detailed view of an individual collection, showing its name, item count, search, and contained artworks.

The individual collection view (Figure 6.17) displays the Collection Name and Number of Items, along with a Search Artworks bar to filter items within that collection. The Artworks section lists all artworks added to this specific collection.

6.3.4 Public Collections

The Public Collections section allows users to explore collections shared by others.

Figure 6.18: The 'Explore Collections' interface for public collections.

The Explore Collections interface (Figure 6.18) provides a curated view of publicly shared collections.

Figure 6.19: List of public collections, showing name, creator, item count, and contained artworks.

Each public collection (Figure 6.19) is displayed with its Collection Name, the Collection Creator, and the Number of Items it contains. The Artworks section shows a preview of the artworks within that public collection.

6.3.5 Favorites

The Favorites section allows users to manage artworks they have marked as favorites.

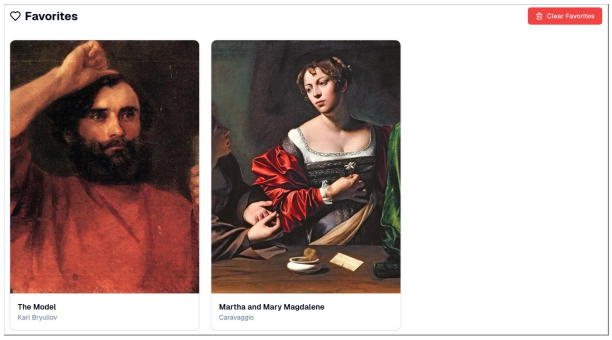


Figure 6.20: The Favorites page, showing options to clear favorites and a list of favorited artworks.

The Favorites page (Figure 6.20) provides a dedicated space for users to view all their Favorite Artworks. A Clear Favorites option is available to remove all items from this list.

6.3.6 View History

The View History feature allows users to review artworks they have previously viewed.

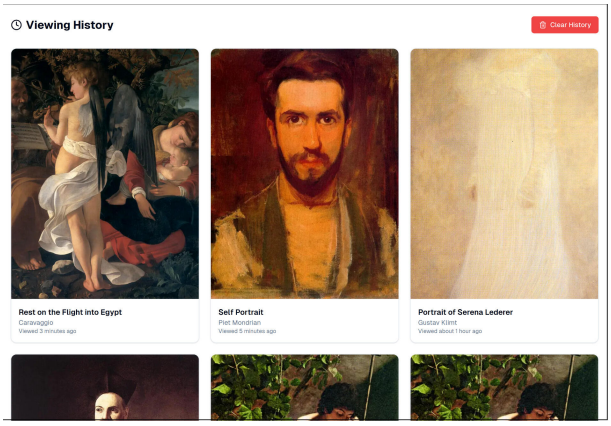


Figure 6.21: The View History page, displaying a chronological list of previously viewed artworks.

As seen in Figure 6.21, the View History page presents a chronological list of Viewed Artworks, allowing users to easily revisit content they have interacted with.

6.3.7 Signing Out

Users can securely sign out of their account from the Account Dropdown.

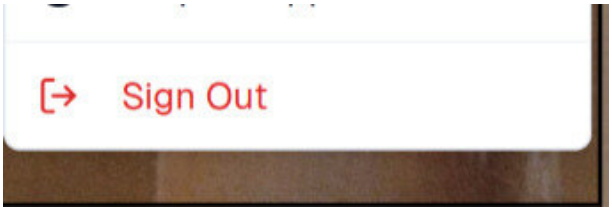


Figure 6.22: The 'Sign out' option within the Account Dropdown.

The Sign out option (Figure 6.22) allows users to end their authenticated session, returning them to the logged-out state of the application.

6.4 Browse Content

Beyond the Home Page, users can directly browse artworks, artists, and categories.

6.4.1 Browse Artworks

This section allows users to explore the full collection of artworks.

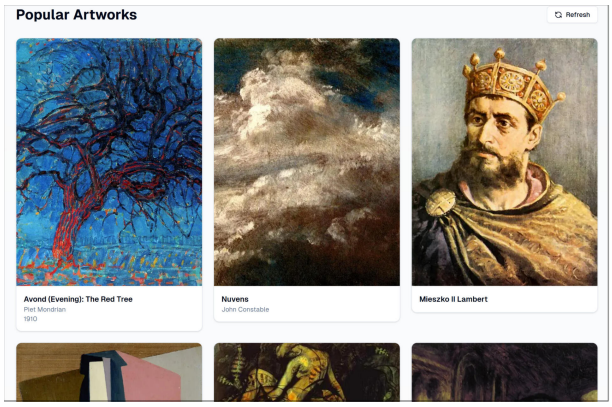


Figure 6.23: The Browse Artworks page, showing a grid view of artworks and pagination controls.

The Browse Artworks page (Figure 6.23) maintains the consistent header and displays artworks in a Grid View. Pagination controls are provided to navigate through the extensive collection.

6.4.2 Artwork Detailed View

Clicking on an individual artwork from any list (e.g., Home Page, Browse Artworks) leads to its detailed view.

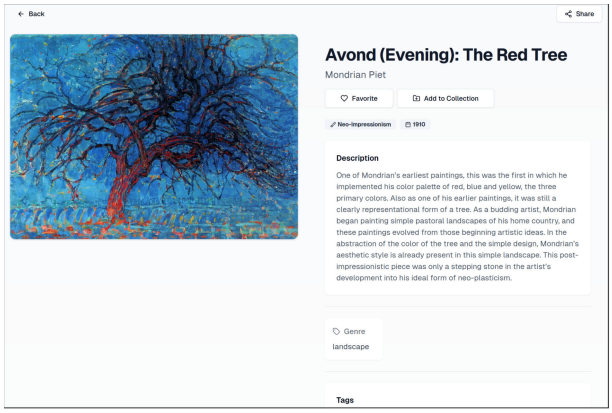


Figure 6.24: The Artwork Detailed page, displaying comprehensive information about an artwork, sharing options, and an image viewer.

The Artwork Detailed page (Figure 6.24) provides comprehensive information about a selected artwork. It includes Sharing Options (Quick Share and Social Media Share via Twitter, Facebook, LinkedIn, Email) to easily share the artwork. The Artwork Information section details the artwork's Name, Artist's Name, Style, Genre, Year of Creation, Description, and Tags. Users can Add to Collection or Add to Favorite. A dedicated Image Viewer with Zoom allows close inspection of the artwork, featuring controls for zooming in, zooming out, and resetting the view. Finally, a Recommendations section suggests other related artworks.

6.4.3 Artist Detailed View

Clicking on an artist's name from an artwork's detail page or an artist listing leads to their detailed profile.

Figure 6.25: The Artist Detailed page, presenting biographical information, sharing options, and featured works.

The Artist Detailed page (Figure 6.25) provides comprehensive information about a selected artist. Similar to artwork details, it includes Sharing Options. The Artist Information section covers their Name, Biography, Birth-Death dates, Gender, external Links, Series, and Periods of Work. A Featured Works section displays a selection of the artist's artworks.

6.4.4 Categories

The Categories page allows users to explore artworks organized by various classifications.

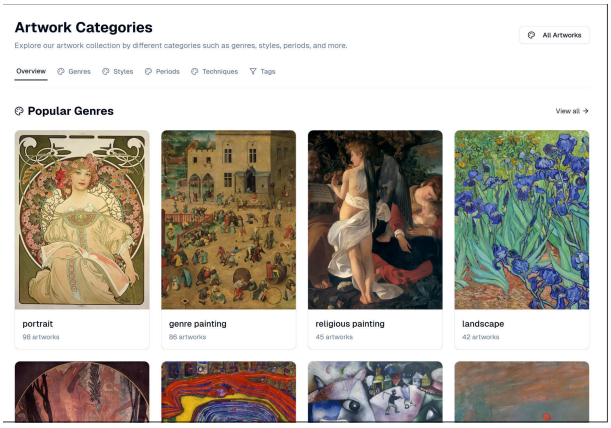


Figure 6.26: The Categories page, offering various ways to explore artworks by classification.

The Categories page (Figure 6.26) provides multiple avenues for exploration. Users can view All Artworks or delve into specific classifications. This includes Popular Genres (displayed as tiled sections), Popular Styles (also tiled), Art Periods (tiled), and Common Tags (artworks with popular tags tiled). Each tiled section provides a visual and interactive way to filter and discover content.

Chapter 7

Bibliography

Books

[1] D. Bugl. *Modern Full-Stack React Projects*. Packt Publishing Ltd, 2024.

[2] T. Despoudis. *TypeScript 5 Design Patterns and Best Practices - Second Edition*. Packt Publishing Ltd, 2025.

[3] J. Hinkula. *Full Stack Development with Spring Boot 3 and React - Fourth Edition*. Packt Publishing Ltd, 2023.

[4] M. Hornbeek. *Continuous Testing, Quality, Security, and Feedback*. Packt Publishing Ltd, 2024.

[5] A. Jorgensen et al. *Microsoft SQL Server 2012 Bible*. Wiley, 2012.

[6] S. Newman. *Building Microservices, 2nd Edition*. O'Reilly Media, Inc., 2021.

[9] S. Selvaraj. *Mastering REST APIs: Boosting Your Web Development Journey with Advanced API Techniques*. Apress, 2024.

Internet knowledge sources

[7] Next.js Documentation. *project structure and organization*. `https://nextjs.org/docs/app/getting-started/project-structure`. 2024.

[8] L. Pollard and Y. Weiss. *Preload responsive images*. `https://web.dev/articles/preload-responsive-images`. 2023.

[10] L. Wagner. *What are UUIDs, and are they better than regular IDs?* `https://blog.boot.dev/clean-code/what-are-uuids-and-should-you-use-them/`. 2023.