PEEK A BOOK

ESTD 2022

Ghostwriter

ANONYMOUS BOOK CLUB

GhostWriter Portfolio

GhostWriter: Development Tool and Language

Front-end and Back-end

- JavaScript
- Next.js
- CSS3

Data Base

- My SQL
- MongoDB

Server and Repository

- GitHub

GhostWriter: Concept

Logo

Operating under the concept of shy ghosts, GhostWriter ensures anonymity, allowing members to freely recommend books and write book diaries. The club employs a cute ghost image logo to maintain its friendly atmosphere.

Color

#FF6100

#FFFFFF

#000000

When writing book diaries, users can choose colors that match their own diaries. However, to maintain simplicity, the website primarily utilizes neutral colors, with the exception of the accent color #FF6100.

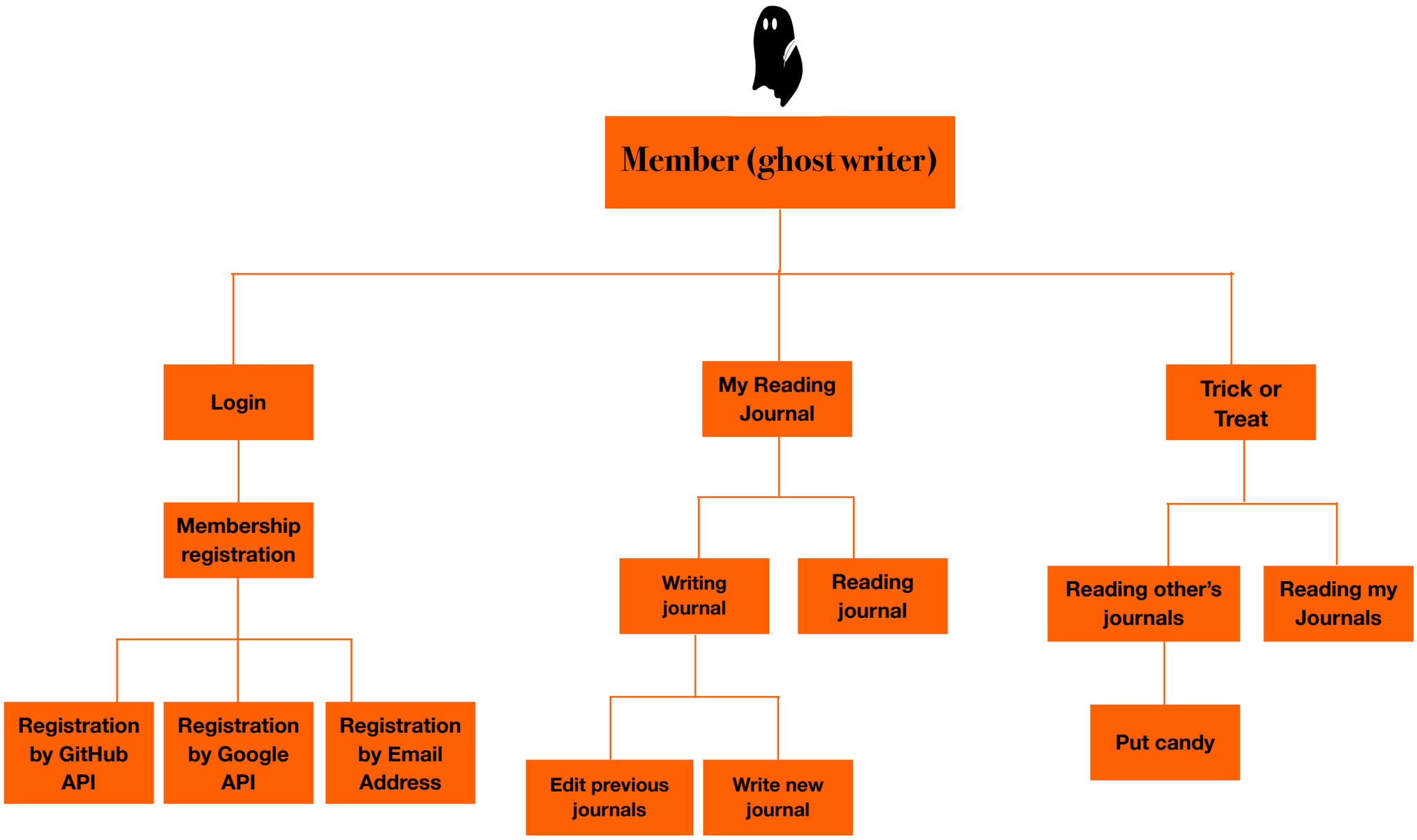
Font

ABC123

ABC123

To give a simple and classic feel, the default fonts used are Bodoni 72 and Distillery Display.

GhostWriter: Use Case



GhostWriter: Mock-up

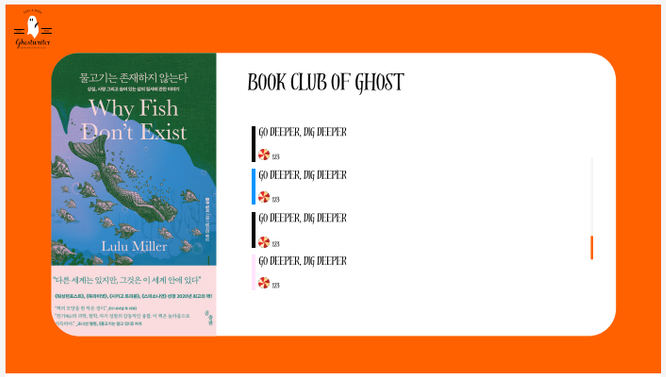
Main page

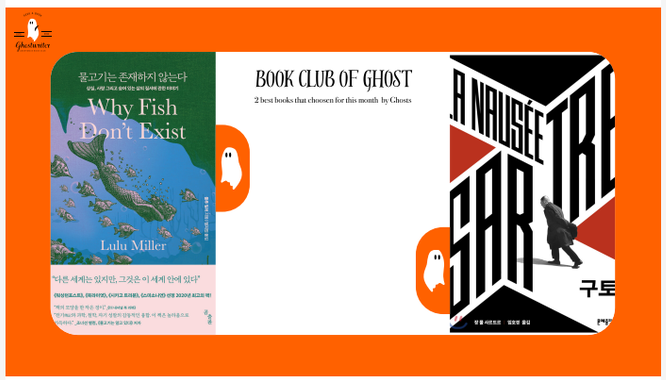
Main page and navigator

Sign In and log In

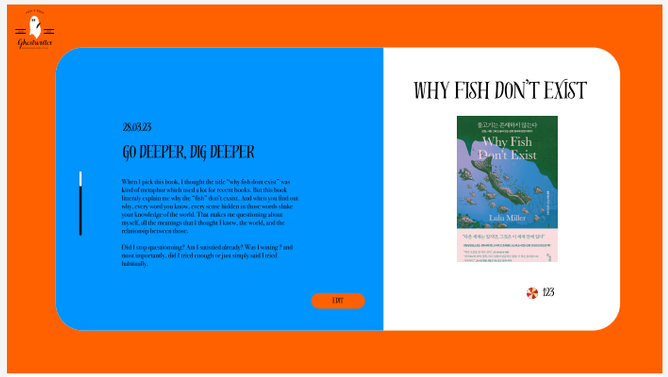
GhostWriter: Mock-up

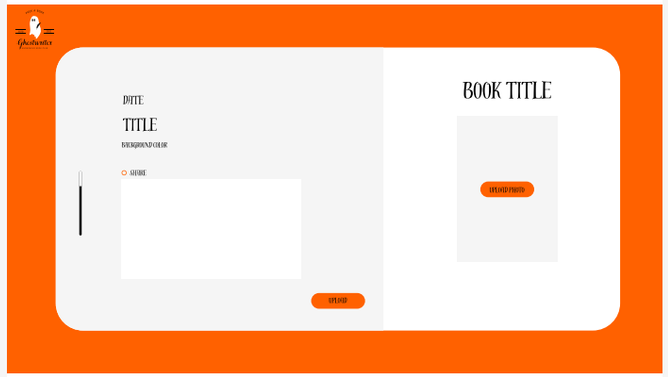
This month's book page

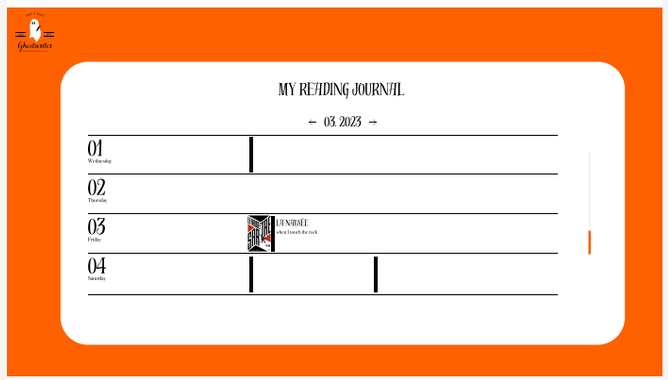




Journal page





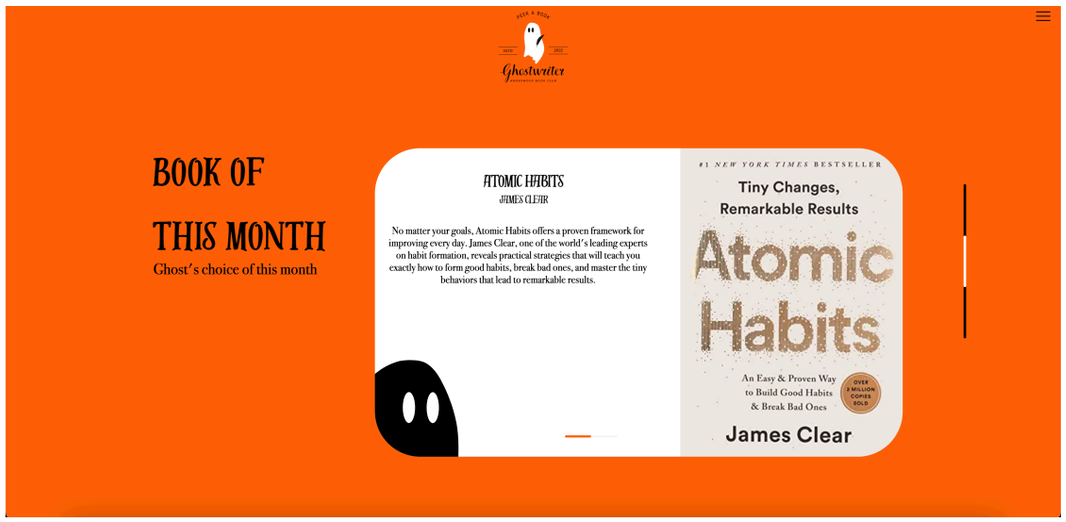


GhostWriter: Main Page(video)

The main page focuses on showcasing the site's concept using various JavaScript effects. It includes a greeting, recommended books for the month, and instructions on how to use the site.

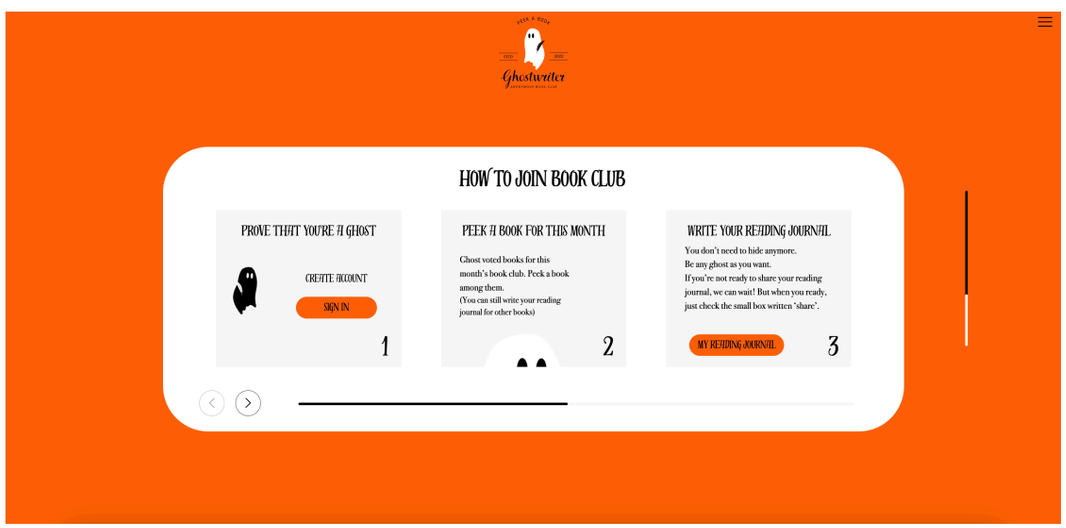
GhostWriter: Main Page(image)

This is a page that shows the site's images and overall concept.



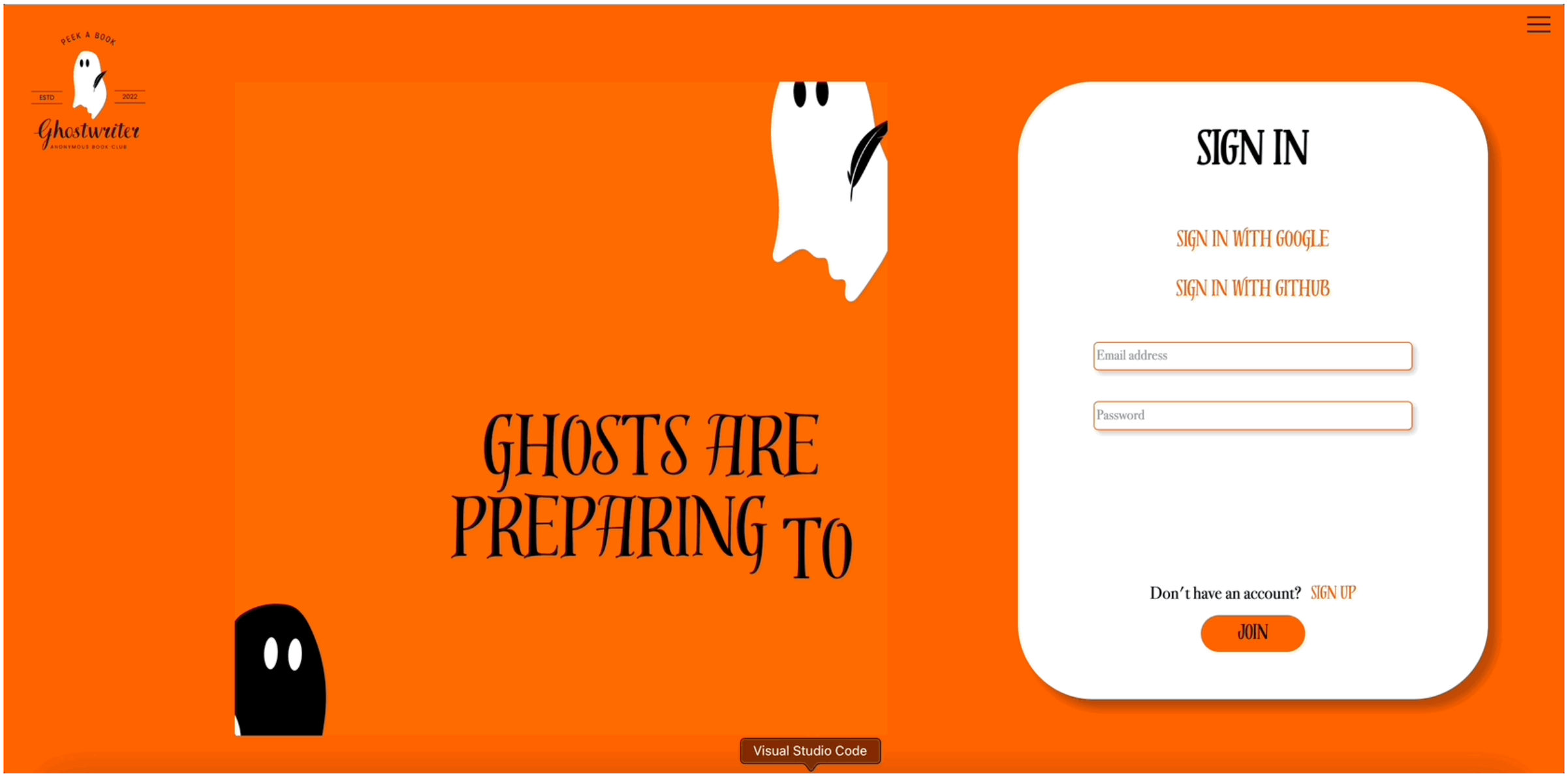
Book recommendations are retrieved using the OpenBook API. (https://openlibrary.org/)

Each month, the system retrieves the two most trending books of the current week—based on search volume—including their title, author, description, and cover image.



This slide shows explanations on how to use the site, what actions can be performed on each page, and the navigation paths between them.

GhostWriter: Sign in and Log in

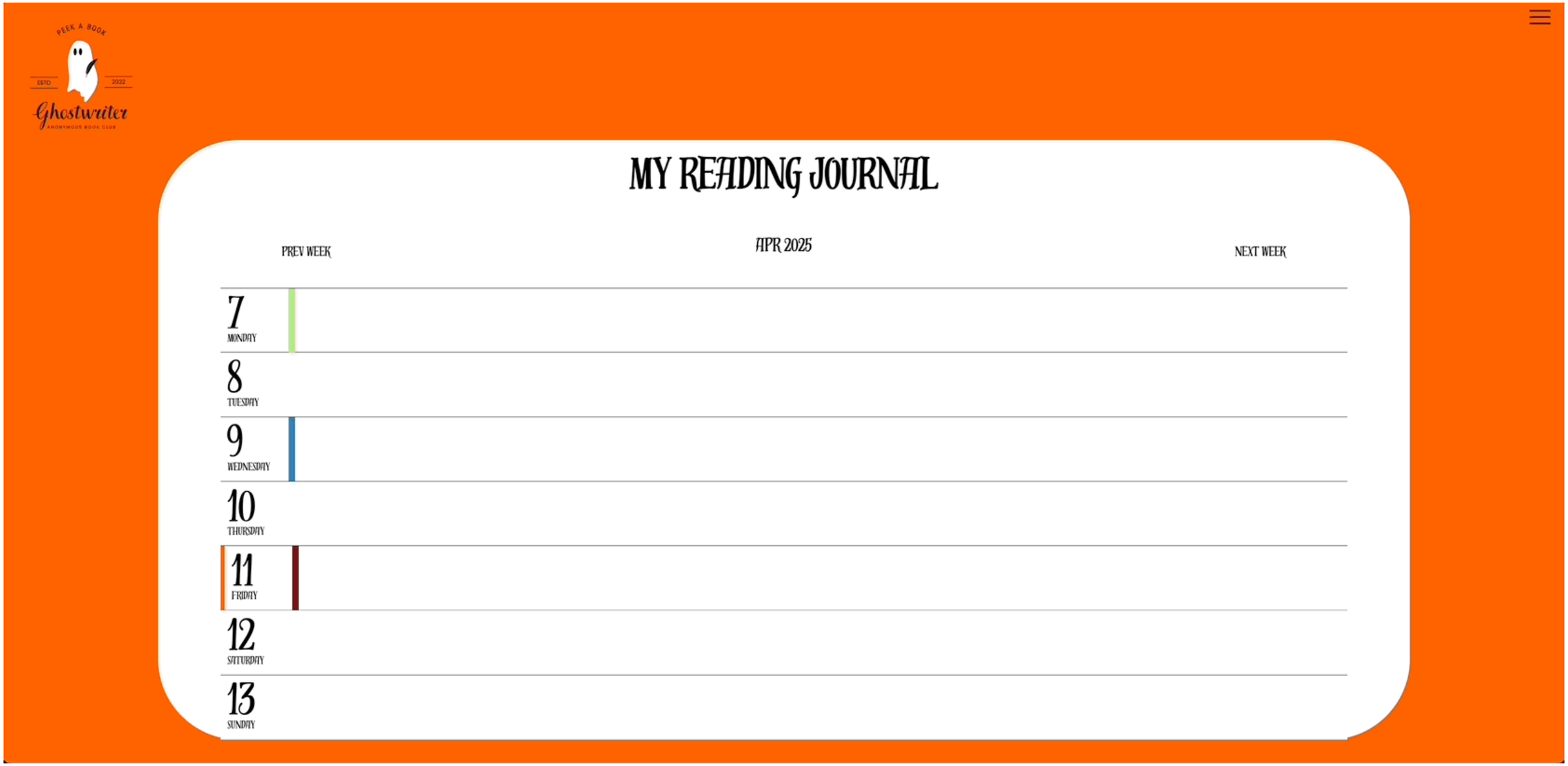


The site allows for both internal registration as well as integration with Google and GitHub APIs for sign-up. Keeping with the concept of anonymity, the registration process is designed to require minimal information.

GhostWriter: My Journal

This page is where you can either manage your previously written journals or create new ones. When you write a journal, you choose a color to represent it, and that color is used to highlight the corresponding date on the calendar. When you hover over a date with a journal entry, the cover image of the book associated with that entry appears, providing a visual reminder of your reading experience.

GhostWriter: Writing Journal(video)



Users can begin composing a journal entry by selecting a specific date on the calendar. Upon entering a book title, the system will automatically retrieve and display the corresponding book cover image. Additionally, users may customize their journal by selecting a background color. If they wish to receive 'candy'—a form of appreciation—from other Ghostwriters, they have the option to make their journal entry publicly shareable.

GhostWriter: Writing Journal(image)

While users can write a new journal entry by selecting a date on the calendar, clicking on a date with an existing entry will retrieve the previously written journal, allowing the user to review and edit it.

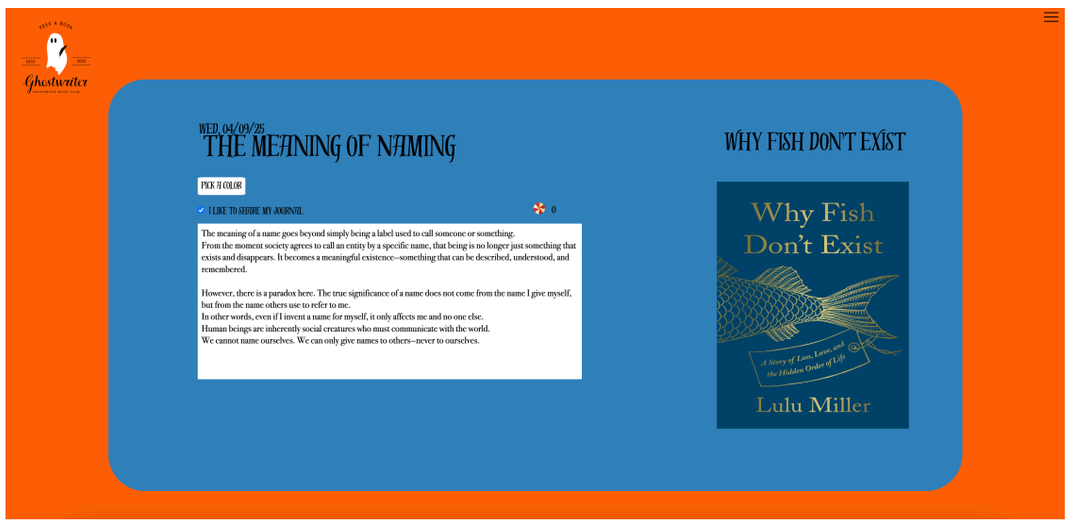
The book cover images are retrieved using the OpenLibrary API.

GhostWriter: Trick or Treat(video)

This page allows you to read journals shared by other ghost writers. After reading someone else's journal, you can choose to "Put Candy" to send a sweet gift to the ghost writer. And you also can see how much candy you got.

GhostWriter: Trick or Treat(image)

If you selected “I like to share my journal” when writing your journal, it will be visible on this page. Here, you can read journals written by other ghost writer, and if you′d like, you can click the “Put Candy” button to send them a sweet gift.



You can also view your own journals on this page, but you won’t be able to send candy to your own journals using the “Put Candy” button.

If you have already given candy to a journal, you cannot give it again. (Each journal can only receive candy from the same user once.)

GhostWriter: Code

Main Page

```
"use client";
import React from "react";
import Image from "next/image";
import { Swiper, SwiperSlide } from "swiper/react";
import "swiper/css";
import "swiper/css/scrollbar";
import "./swiper.css";
import "./globals.css";
import { Scrollbar, Pagination, Keyboard } from "swiper/modules";
import Slide01 from "./index/Slide01";
import Slide02 from "./index/Slide02";
import Slide03 from "./index/Slide03";

export default function App() {
  return (
    <>
      <Image
        src="/images/Ghostwriter.svg"
        alt="logo_image"
        width={250}
        height={250}
        className="mainLogo"
      />
      <Swiper
        scrollbar={{ draggable: true, forceToAxis: true }}
        modules={[Scrollbar, Pagination, Keyboard]}
        pagination={true}
        keyboard={true}
        direction={"vertical"}
        className="mySwiper"
      >
        <SwiperSlide>
          <Slide01 />
        </SwiperSlide>
        <SwiperSlide>
          <Slide02 />
        </SwiperSlide>
        <SwiperSlide>
          <Slide03 />
        </SwiperSlide>
      </Swiper>
    </>
  );
}
```

To implement importing for dividing the screen into three parts with swiper module, receiving child components, and enabling drag and scroll functionality using modules and scrollbar properties.

GhostWriter: Code

Main Page: Slide01

```
"use client";

import React, { useState, useEffect } from "react";
import Image from "next/image";
import "swiper/css";
import "swiper/css/scrollbar";
import "../swiper.css";

export default function Slide01() {
  const [subtitle1Visible, setSubtitle1Visible] = useState(false);
  const [subtitle2Visible, setSubtitle2Visible] = useState(false);
  const [isMouseOver, setIsMouseOver] = useState(false);

  const handleMouseEnter = () => {
    setIsMouseOver(true);
  };
  const handleMouseLeave = () => {
    setIsMouseOver(false);
  };

  useEffect(() => {
  const timer1 = setTimeout(() => {
    setSubtitle1Visible(true);
  }, 500);

  const timer2 = setTimeout(() => {
    setSubtitle2Visible(true);
  }, 1000);

  return () => {
    clearTimeout(timer1);
    clearTimeout(timer2);
  };
}, []);
return (
  <div className="first_slide">
    <div className="title_box">
      <p className={`sub_title_1 ${subtitle1Visible ? "visible" : ""}`}>
        Welcome to book club of ghost
      </p>
      <p className={`sub_title_2 ${subtitle2Visible ? "visible" : ""}`}>
        Are you ready to peek a book?
      </p>
    </div>
    <Image
      src="/images/mainGhost.svg"
      alt="main_ghost"
      width={650}
      height={650}
      className={`first_slide_img ${isMouseOver ? "fade-out" : ""}`}
      onMouseEnter={handleMouseEnter}
      onMouseLeave={handleMouseLeave}
    />
  </div>
);
```

Using the ‘useState’ hook to manage the states of ‘subtitle1Visible’, ‘subtitle2Visible’, and ‘isMouseOver’, which are used to track whether specific parts of the page are visible and whether the mouse is hovering over the image.

The ‘useEffect’ hook is used to sequentially change the ‘subtitle1Visible’ and ‘subtitle2Visible’ states when the component mounts to achieve a partial appearing effect.

The ‘handleMouseEnter’ function is called when the mouse moves over the image, and the ‘handleMouseLeave’ function is called when the mouse moves out of the image to implement image behavior.

Timers are used to change the ‘subtitle1Visible’ and ‘subtitle2Visible’ states at regular intervals to create a gradual disappearing and appearing effect for the image.

GhostWriter: Code

Main Page: Slide02

```
"use client";

import React, { useState, useEffect } from "react";
import Image from "next/image";
import { Swiper, SwiperSlide } from "swiper/react";
import { Scrollbar, Keyboard, Pagination } from "swiper/modules";
import "swiper/css";
import "swiper/css/scrollbar";
import "../swiper.css";

const Slide02 = () => {
  const [works, setWorks] = useState([]);
  const [authors, setAuthors] = useState([]);
  const [coverIds, setCoverIds] = useState([]);
  const [descriptions, setDescriptions] = useState([]);

  useEffect(() => {
    const fetchBookData = async () => {
      try {
        const response = await fetch(
          "https://openlibrary.org/trending/weekly.json"
        );
        const data = await response.json();

        if (data && data.works) {
          const firstTwoWorks = data.works.slice(0, 2);
          setWorks(firstTwoWorks);

          const bookPromises = firstTwoWorks.map(async (work, index) => {
            let description = "No description available";
            let author = "Unknown Author";
            let coverId = null;

            try {
              const workResponse = await fetch(
                `https://openlibrary.org${work.key}.json`
              );
              const workData = await workResponse.json();

              if (workData.description) {
                description =
                  typeof workData.description === "string"
                    ? workData.description
                    : workData.description.value;
              }
            } catch (error) {
              console.error(
                `Error fetching description for ${work.title}:`,
                error
              );
            }
```

This component utilizes the useState hook to manage multiple pieces of data state:
works stores the top two trending book objects retrieved from the API, while authors, coverIds, and descriptions each store derived data extracted from additional API requests (author name, book cover image ID, and book description respectively).

When the component mounts, the useEffect hook is triggered to begin fetching book data from the OpenLibrary API. Specifically, it sends a request to the trending/weekly.json endpoint to retrieve the currently trending books of the week. From the returned results, only the first two books are selected for display, which are then stored using setWorks().

GhostWriter: Code

Main Page: Slide02

```
          try {
            if (work.authors && work.authors.length > 0) {
              const authorKey = work.authors[0]?.key;
              if (authorKey) {
                const authorResponse = await fetch(
                  `https://openlibrary.org${authorKey}.json`
                );
                const authorData = await authorResponse.json();

                if (authorData && authorData.name) {
                  author = authorData.name;
                } else {
                  console.warn(`No name found for author: ${authorKey}`);
                }
              }
            } else if (work.author_name) {
              author = work.author_name[0] || "Unknown Author";
            }
          } catch (error) {
            console.error(`Error fetching author for ${work.title}:`, error);
          }
```

```
          try {
            const searchResponse = await fetch(
              `https://openlibrary.org/search.json?title=${encodeURIComponent(
                work.title
              )}`
            );
            const searchData = await searchResponse.json();

            if (searchData.docs && searchData.docs.length > 0) {
              coverId = searchData.docs[0].cover_i;
            }
          } catch (error) {
            console.error(
              `Error fetching cover image for ${work.title}:`,
              error
            );
          }

          return { description, author, coverId };
        });

        const results = await Promise.all(bookPromises);

        setDescriptions(results.map((result) => result.description));
        setAuthors(results.map((result) => result.author));
        setCoverIds(results.map((result) => result.coverId));
      }
    } catch (error) {
      console.error("Error fetching book data:", error);
    }
  };
  fetchBookData();
}, []);
```

For each of the selected trending books, a series of asynchronous requests are made to fetch additional metadata:

- **Book Description**: Using the work.key, the component makes a request to the individual work's JSON endpoint to retrieve a detailed description. If the description exists as an object, the .value field is used.
- **Author Name**: The component checks if the authors array exists and retrieves the first author's name using the author key. If not found, it attempts to fallback to author_name from the original response.
- **Book Cover ID**: A search request is made using the book title to find an appropriate cover_i identifier, which is later used to build the book cover image URL.

All of these requests are executed in parallel using Promise.all, and the resulting values are mapped and saved into the component's state using setDescriptions(), setAuthors(), and setCoverIds(). This architecture allows the Swiper slides to render the book's title, author, description, and cover image efficiently once data is fully fetched.

GhostWriter: Code

SignIn

```
"use client";
import "./signin.css";
import "../globals.css";
import Image from "next/image";
import Link from "next/link";
import AuthBtn from "./AuthBtn";
import Video from "next-video";
import React, { useRef } from "react";
import { signIn } from "next-auth/react";

export default function Signin() {
  const emailRef = useRef(null);
  const passwordRef = useRef(null);

  const handleSubmit = async () => {
  const result = await signIn("credentials", {
      email: emailRef.current,
      password: passwordRef.current,
      redirect: true,
      callbackUrl: "/",
    });
};
return (
  <div>
    <Link href="/">
      <Image
        src="/images/Ghostwriter.svg"
        alt="logo_image"
        width={250}
        height={250}
        className="logo"
      />
    </Link>
    <div className="login_box">
      <h4 className="title">SIGN IN</h4>
      <AuthBtn />
      <form method="POST" action="/api/auth/login" className="form_box_s">
        <input
          id="email"
          name="email"
          type="text"
          required
          placeholder="Email address"
          className="input_email_s"
          ref={emailRef}
          onChange={(e) => {
            emailRef.current = e.target.value;
          }}
        />
        <input
          id="password"
          name="password"
          type="password"
          required
          placeholder="Password"
          className="input_password_s"
          ref={passwordRef}
          onChange={(e) => {
            passwordRef.current = e.target.value;
          }}
        />
        <button className="btn_submit" onClick={handleSubmit}>
          JOIN
        </button>
      </form>
      <div className="signin_box">
        <span>Don't have an account?   </span>
        <Link href="/register" className="signin_link">
          SIGN UP
        </Link>
      </div>
    </div>
  );
}
```

The ‘handleSubmit’ function is called when the login form is submitted, and it authenticates the user and logs them in using the signIn function from [...nextAuth].js. The ‘signIn’ function authenticates the user using the credentials provider, attempting authentication with the provided email and password, and setting the redirect option to true to automatically redirect the user after logging in, while also specifying the callbackUrl to navigate to after redirection.

Each input field tracks its value using ref, allowing the captured input values to be used in the ‘handleSubmit’ function through onChange event handlers.

GhostWriter: Code

SignIn

```
import { connectDB } from "@/util/database";
import { MongoDBAdapter } from "@next-auth/mongodb-adapter";
import NextAuth from "next-auth";
import GithubProvider from "next-auth/providers/github";
import GoogleProvider from "next-auth/providers/google";
import CredentialsProvider from "next-auth/providers/credentials";
import bcrypt from "bcrypt";

export const authOptions = {
  providers: [
    GithubProvider({
      clientId: process.env.GOOGLE_CLIENT_ID,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    }),

    GoogleProvider({
      clientId: process.env.GITHUB_CLIENT_SECRET,
      clientSecret: process.env.GITHUB_CLIENT_SECRET,
    }),

  CredentialsProvider({
    // credentials provider 설정
    name: "Credentials",
    credentials: {
      email: { label: "email", type: "text" },
      password: { label: "password", type: "password" },
    },
    async authorize(credentials) {
      let db = (await connectDB).db("ghostwriter");
      let user = await db
        .collection("member")
        .findOne({ email: credentials.email });
      if (!user) {
        console.log("해당 이메일은 없음");
        return null;
      }
      const pwcheck = await bcrypt.compare(
        credentials.password,
        user.password
      );
      if (!pwcheck) {
        console.log("비번틀림");
        return null;
      }
      return user;
    },
  }),
],

//3. jwt 써놔야 잘됩니다 + jwt 만료일설정
session: {
  strategy: "jwt",
  maxAge: 30 * 24 * 60 * 60, //30일
},
```

```
session: {
    strategy: "jwt",
    maxAge: 30 * 24 * 60 * 60, //30일
  },

  callbacks: {
    //4. jwt 만들 때 실행되는 코드
    //user변수는 DB의 유저정보담겨있고 token.user에 뭐 저장하면 jwt에 들어갑니다.
    jwt: async ({ token, user }) => {
      if (user) {
        token.user = {};
        token.user.name = user.name;
        token.user.email = user.email;
        token.user.role = user.role;
      }
      return token;
    },
    //5. 유저 세션이 조회될 때 마다 실행되는 코드
    session: async ({ session, token }) => {
      session.user = token.user;
      return session;
    },
  },
  pages: {
    signIn: "/signin",
    logIn: "/login",
  },
  secret: process.env.NEXTAUTH_SECRET,
  adapter: MongoDBAdapter(connectDB),
};
export default NextAuth(authOptions);
```

This code sets up options for configuring authentication using NextAuth. It configures the 'providers' to specify which authentication providers to use, including GitHub, Google, and the Credentials provider. The 'CredentialsProvider' is configured to authenticate users using their email and password. In the 'authorize' function of this provider, it searches for the user in the database using the provided email and password, performs authentication, and returns the user object once authenticated.

GhostWriter: Code

SignIn

```
import { connectDB } from "@/util/database";
import bcrypt from "bcrypt";

export default async function handler(req, res) {
  const db = (await connectDB).db("ghostwriter");
  let email = await db.collection("member").findOne({ email:
req.body.email });
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

  if (req.method === "POST") {
    if (
      req.body.name === "" ||
      req.body.email === "" ||
      req.body.password === ""
    ) {
      res.write(
        "<script>alert('Hostname, Email address, and password are
required')</script>"
      );
      res.write("<script>window.location='../../../register'</script>");
      return res.end(); // 응답 종료
    } else {
      if (!emailRegex.test(req.body.email)) {
        res.write("<script>alert('Invalid email address format')</script>");
        res.write("<script>window.location='../../../register'</script>");
        return res.end(); // 응답 종료
      } else if (email != null) {
        res.write(
          "<script>alert('Your email address is already taken')</script>"
        );
        res.write("<script>window.location='../../../register'</script>");
        return res.end(); // 응답 종료
      } else if (req.body.password !== req.body.confirm_password) {
        res.write(
          "<script>alert('Please check your confirm password')</script>"
        );
        res.write("<script>window.location='../../../register'</script>");
        return res.end(); // 응답 종료
      } else {
        delete req.body.confirm_password;
        let hash = await bcrypt.hash(req.body.password, 10);
        req.body.password = hash;
        let db = (await connectDB).db("ghostwriter");
        await db.collection("member").insertOne(req.body);
        return res.redirect(302, "/"); // 바로 리다이렉트
      }
    }
  }
}
```

The code serves as a Next.js API route handler, performing the functionality of securely processing and validating user-submitted registration form data to facilitate safe registration. It encompasses the steps of email verification, email format validation, password confirmation, password hashing, and subsequent redirection upon successful registration.

It verifies if the email submitted through the registration form already exists in the database. If it does, indicating that the email is already in use by another user, registration is denied. It checks if the submitted email adheres to the correct format, validating it using regular expressions. It verifies whether the submitted password matches the password confirmation and enhances security by hashing the password. Using the bcrypt library, it hashes the password and stores the hashed password in the database. Upon successfully passing all validation steps, it stores the submitted information in the database to complete the registration process and redirects the user to the homepage.

GhostWriter: Code

Register

```
import { connectDB } from "@/util/database";
import bcrypt from "bcrypt";
import { signIn } from "next-auth/react";

export default async function handler(req, res) {
  const db = (await connectDB).db("ghostwriter");

  if (req.method === "POST") {
    const { email, password } = req.body;

    if (email === "" || password === "") {
      res.write(
        "<script>alert('Please enter your email address and password')</script>"
      );
      res.write("<script>window.location='../../../signin'</script>");
      return res.end();
    }

    // 사용자 이메일로 검색
    const user = await db.collection("member").findOne({ email });
    if (!user) {
      res.write("<script>alert('Invalid email address')</script>");
      res.write("<script>window.location='../../../signin'</script>");
      return res.end();
    }

    // 비밀번호 확인
    const isPasswordValid = await bcrypt.compare(password, user.password);
    if (!isPasswordValid) {
      res.write("<script>alert('Invalid password')</script>");
      res.write("<script>window.location='../../../signin'</script>");
      return res.end();
    }

    // 인증이 성공하면 메인 페이지로 이동
    console.log(user);
    res.redirect(302, "/");
    return res.end();
  } else {
    // POST 요청이 아닌 경우 405 Method Not Allowed 응답
    res.writeHead(405);
    return res.end();
  }
}
```

This code verifies and securely handles login form data submitted by users.

It connects to the database and checks if the request method is POST. If it's a POST request, it extracts the email and password from the request. If either the email or password is empty, it displays a warning prompting the user to enter their email address and password and redirects to the login page.

It searches for the user in the database using the email. If the user is not found, it notifies that the email address is invalid and redirects to the login page. If the user is found, it compares the entered password with the hashed password in the database to verify if they match. If the passwords don't match, it notifies that the password is invalid and redirects to the login page.

If authentication is successful, it redirects the user to the main page; otherwise, it returns a 405 Method Not Allowed response.

GhostWriter: Code

MyJournal(Calendar)

```
"use client";

import "../globals.css";
import Image from "next/image";
import Link from "next/link";
import React, { useState, useEffect } from "react";
import "react-calendar/dist/Calendar.css";
import "./myJournal.css";
import { getSession } from "next-auth/react";

import {
  format, subMonths, addMonths, startOfWeek, addDays, isSameDay, lastDayOfWeek, getWeek, addWeeks,
subWeeks,
} from "date-fns";
import { useRouter } from "next/navigation";

const Calendar = () => {
  const [currentMonth, setCurrentMonth] = useState(new Date());
  const [currentWeek, setCurrentWeek] = useState(getWeek(currentMonth));
  const [selectedDate, setSelectedDate] = useState(new Date());
  const router = useRouter();
  const [journalData, setJournalData] = useState([]);

  useEffect(() => {
    const fetchJournalData = async () => {
      try {
        const response = await fetch("/api/journalList");
        if (!response.ok) {
          throw new Error("Failed to fetch journal data");
        }
        const data = await response.json();
        setJournalData(data);
      } catch (error) {
        console.error("Error fetching journal data:", error);
      }
    };

    fetchJournalData();
  }, []);

  const onDateClickHandle = (day, dayStr) => {
    setSelectedDate(day);
    showDetailsHandle(dayStr);
  };

  const changeMonthHandle = (btnType) => {
    if (btnType === "prev") {
      setCurrentMonth(subMonths(currentMonth, 1));
    }
    if (btnType === "next") {
      setCurrentMonth(addMonths(currentMonth, 1));
    }
  };

  const changeWeekHandle = (btnType) => {
    if (btnType === "prev") {
      setCurrentMonth(subWeeks(currentMonth, 1));
      setCurrentWeek(getWeek(subWeeks(currentMonth, 1)));
    }
    if (btnType === "next") {
      setCurrentMonth(addWeeks(currentMonth, 1));
      setCurrentWeek(getWeek(addWeeks(currentMonth, 1)));
    }
  };
```

```
  const renderHeader = () => {
    const dateFormat = "MMM yyyy";
    return (
      <div className="header row flex-middle">
        <div className="col col-start"></div>
        <div className="col col-center">
          <span>{format(currentMonth, dateFormat)}</span>
        </div>
        <div className="col col-end"></div>
      </div>
    );
  };
```

This portion of the code defines the foundational logic for managing the state of a calendar-based journaling interface. The component utilizes React's useState hook to maintain key variables such as the currently displayed month (currentMonth), the selected date (selectedDate), and the current week (currentWeek). It also tracks a list of journal entries fetched from the backend API through the journalData state.

Upon initial rendering, the useEffect hook triggers an asynchronous function, fetchJournalData, which retrieves journal entries via a request to /api/journalList. The retrieved entries are stored in the journalData array, enabling the calendar to visually represent which dates contain existing journals.

In addition to data fetching, the component provides several utility functions to support user interactions. The changeMonthHandle function enables month navigation by incrementing or decrementing the currentMonth state. Similarly, the changeWeekHandle allows users to scroll through the calendar on a weekly basis by adjusting both currentMonth and currentWeek. When a user selects a specific date, the onDateClickHandle function updates the selectedDate and triggers showDetailsHandle, which programmatically navigates the user to the journal-writing page for that day.

The renderHeader function returns the top section of the calendar view, displaying the current month in a formatted string (e.g., "Apr 2025"). This enhances user orientation and provides context for the calendar's currently displayed period.

GhostWriter: Code

MyJournal(Calendar)

```
const renderCells = () => {
  const startDate = startOfWeek(currentMonth, { weekStartsOn: 1 });
  const endDate = lastDayOfWeek(currentMonth, { weekStartsOn: 1 });
  const dateFormat = "d";
  const dateFormat2 = "EEEE";
  const rows = [];
  let days = [];
  let day = startDate;
  let formattedDate = "";

  while (day <= endDate) {
    for (let i = 0; i < 7; i++) {
      formattedDate = format(day, dateFormat);
      const cloneDay = day;
      const hasJournalEntry = journalData.some((entry) => {
        const entryDate = new Date(entry.date);
        return isSameDay(entryDate, cloneDay);
      });
      const journalEntry = journalData.find((entry) => {
        const entryDate = new Date(entry.date);
        return isSameDay(entryDate, cloneDay);
      });

      days.push(
        <div
          className={`col cell ${
            isSameDay(day, new Date())
              ? "today"
              : isSameDay(day, selectedDate)
              ? "selected"
              : ""
          }`}
          key={day}
          onClick={() => {
            const dayStr = format(cloneDay, "ccc dd MMM yy");
            onDateClickHandle(cloneDay, dayStr);
          }}
        >
          <div className="date">
            <span className="number">{formattedDate}</span>
            <span className="days">
              {format(addDays(startDate, i), dateFormat2)}
            </span>
            {hasJournalEntry && (
              <div
                className="journal-entry-indicator"
                style={{ backgroundColor: journalEntry.colorPicker }}
                onClick={() => {
                  const dayStr = format(cloneDay, "ccc dd MMM yy");
                  onJournalLiskHandle(cloneDay, dayStr);
                }}
              >
                <div className="image-preview">
                  <img
                    className="bookCoverLst"
                    id="bookCover"
                    name="bookCover"
                    src={`https://covers.openlibrary.org/b/id/${journalEntry.coverId}-M.jpg`}
                    alt="Book Cover"
                    onClick={() => {
                      const dayStr = format(cloneDay, "ccc dd MMM yy");
                      onJournalLiskHandle(cloneDay, dayStr);
                    }}
                  />
                </div>
              </div>
            )}
          </div>
        </div>
      );
      day = addDays(day, 1);
    }
```

The second portion of the component is responsible for rendering the individual cells of the calendar and managing interactions related to journal entries. The renderCells function determines the start and end dates of the week using startOfWeek and lastDayOfWeek, respectively. It then iterates over each day, formatting the date and checking for any corresponding journal entries in the journalData array.

If a journal entry exists for a given date, the calendar visually indicates this by rendering a colored background (using the colorPicker value) and displaying the book cover image associated with the journal, which is retrieved using the coverId field from the OpenLibrary API. Each calendar cell is interactive: clicking on a date invokes onDateClickHandle to initiate writing a new journal entry, while clicking on a cover image or indicator calls onJournalLiskHandle, which navigates the user to a detailed view of the selected journal.

The renderWeek function introduces user-friendly navigation controls, specifically buttons labeled “prev week” and “next week,” which facilitate weekly browsing by calling changeWeekHandle with the appropriate direction.

GhostWriter: Code

MyJournal(Calendar)

```
      rows.push(
        <div className="row" key={day}>
          {days}
        </div>
      );
      days = [];
    }
    return <div className="body">{rows}</div>;
};
const renderWeek = () => {
  return (
    <div className="header rowBottom flex-middle">
      <div className="col col-start">
        <div className="icon" onClick={() => changeWeekHandle("prev")}>
          prev week
        </div>
      </div>
      <div className="col col-end" onClick={() => changeWeekHandle("next")}>
        <div className="icon">next week</div>
      </div>
    </div>
  );
};

//writingJournal
const showDetailsHandle = (day) => {
  router.push("/myJournal/writingJournal/" + day);
};

//JounralList
const onJournalLiskHandle = (day) => {
  router.push("/myJournal/journalDetail/" + day);
};
```

Additionally, two routing functions are defined at the end of the component. The showDetailsHandle function constructs a dynamic route to the journal-writing page (/myJournal/writingJournal/:date), while onJournalLiskHandle directs the user to the journal detail page (/myJournal/journalDetail/:date) for any existing entry. Together, these functions ensure a seamless navigation experience within the application, allowing users to create, edit, and view journal entries directly through the interactive calendar interface.

```
import { connectDB } from "@/util/database";
import bcrypt from "bcrypt";
import { getToken } from "next-auth/jwt";

export default async function handler(req, res) {
  const db = (await connectDB).db("ghostwriter").collection("journal");
  const token = await getToken({ req: req });
  const email = token.email;
  const journalList = await db.find({ email: email }).toArray();
  if (journalList != null) {
    res.status(200).json(journalList);
  } else {
    res.status(405).end();
  }
}
```

Using this email address, the handler queries the journal collection to find all documents that match the user's email. The result is an array of journal entries authored by the current user.

If the query returns a result (i.e., the journal entries exist), the server responds with an HTTP status code of 200 OK, along with the journal data in JSON format. If no journal entries are found or an unexpected condition arises, it responds with a 405 Method Not Allowed status to indicate that the operation could not be completed as expected.

In summary, this endpoint enables authenticated users to retrieve their personal journal entries from the database. It is typically used in scenarios where the client application needs to load and display a user's previously written journal entries.

GhostWriter: Code

Writing Journal

```
"use client";
import React from "react";
import "./writingJournal.css";
import "../../../globals.css";
import Image from "next/image";
import Link from "next/link";
import Video from "next-video";
import JournalBox from "./journalBox.js";
import { useParams, useSearchParams } from "next/navigation";

const WritingJournal = () => {
  const params = useParams();
  const searchParams = useSearchParams();
  const type = searchParams.get("type");
  const id = searchParams.get("id");

  return (
    <div>
      <Link href="/">
        <Image
          src="../../../../images/Ghostwriter.svg"
          alt="logo_image"
          width={250}
          height={250}
          className="logo"
        />
      </Link>
      <JournalBox date={params} type={type} id={id} />
    </div>
  );
};

export default WritingJournal;
```

The renderHeader function renders the header section of the calendar, displaying the current month and providing navigation options to switch between months.

The renderCells function generates the grid cells for displaying dates in the calendar. It creates rows and columns of date cells, with each cell representing a day in the current month. It also highlights the current day and the selected day.

The renderHeader function renders the header section of the calendar, displaying the current month and providing navigation options to switch between months.

The renderCells function generates the grid cells for displaying dates in the calendar. It creates rows and columns of date cells, with each cell representing a day in the current month. It also highlights the current day and the selected day.

The renderWeek function renders a control for navigating between weeks, allowing users to view the calendar by week.

Finally, the Calendar component is exported as the default export, making it available for use in other components. It also includes JSX for rendering the logo, title, and the calendar itself within a parent container.

GhostWriter: Code

Writing Journal

```
"use client";

import "./writingJournal.css";
import "../../../globals.css";
import Image from "next/image";
import Link from "next/link";
import Video from "next-video";
import ColorPicker from "./colorPicker";
import React, { useState, useEffect } from "react";
import { useRouter } from "next/navigation";
import { useSession } from "next-auth/react";

const WritingJournal = (props) => {
  const router = useRouter();
  const { data: session } = useSession(); // 로그인 정보 가져오기
  const myEmail = session?.user?.email;
  const decodedDate = decodeURIComponent(props.date.date);
  const type = props.type;
  const id = props.id;

  const parsedDate = new Date(decodedDate);
  const formattedDate = parsedDate.toLocaleDateString("en-US", {
    weekday: "short",
    year: "2-digit",
    month: "2-digit",
    day: "2-digit",
});
const [typeReady, setTypeReady] = useState(false);

useEffect(() => {
  if (type !== null && type !== undefined) {
    setTypeReady(true);
  }
}, [type]);

//신규
const [inputText, setInputText] = useState("");
const [coverId, setCoverId] = useState("");
const [author, setAuthor] = useState("");
const [email, setEmail] = useState("");
const [backgroundColor, setBackgroundColor] = useState(null); // 초기 배경색 상태
const [journalData, setJournalData] = useState(null);
const [titleValue, setTitleValue] = useState("title");
const [contextValue, setContextValue] = useState("write your journal!");

//수정
const [formData, setFormData] = useState({
  title: journalData?.title || "", // undefined일 경우 빈 문자열로 초기화
  context: journalData?.context || "",
  bookTitle: journalData?.bookTitle || "",
  coverId: journalData?.coverId || "",
  author: journalData?.author || "",
  email: journalData?.email || "",
  colorPicker: journalData?.colorPicker || "",
  bookTitle: journalData?.bookTitle || "book title",
  candy: journalData?.candy || "0",
});
```

The cmponent imports essential modules, including styling, image handling, routing, and session management. It also retrieves dynamic route parameters (date, type, and id) passed via props, decoding and formatting the date for consistent internal use and display.

A central aspect of the component′s behavior is governed by React′s useState and useEffect hooks. These hooks manage various states such as the journal's title, content, associated book information (cover ID and author), color selection, and user session data. The component initializes a blank journal for new entries or populates fields with existing data when editing. This is achieved by asynchronously fetching the journal's details from the backend using the useEffect hook once the component is mounted and the necessary identifiers are available.

In terms of session management, the useSession hook from NextAuth is used to access the logged-in user's information, which plays a role in rendering user-specific views and determining whether certain actions (e.g., uploading or giving candy) are permitted. Conditional rendering based on this session ensures that users only see options appropriate to their ownership of the journal.

Throughout the component, user inputs are dynamically managed using controlled components bound to useState. These include the journal′s title, context, selected color, and book title. When a user enters a book title, an external API call to OpenLibrary retrieves the corresponding cover image and author information, which are then integrated into the form state. The background color, selected using a ColorPicker component, is similarly stored and applied to the visual style of the writing area.

GhostWriter: Code

Writing Journal

```
useEffect(() => {
  // journalData가 바뀔 때 상태를 업데이트
  if (journalData) {
    setFormData({
      title: journalData.title,
      context: journalData.context,
      bookTitle: journalData.bookTitle,
      coverId: journalData.coverId,
      author: journalData.author,
      colorPicker: journalData.colorPicker,
      bookTitle: journalData.bookTitle,
      email: journalData.email,
      candy: journalData.candy,
    });
  }
}, [journalData]); // journalData가 변경될 때만 실행

const handleChange = (e) => {
  const { name, value } = e.target;
  setFormData((prevState) => ({
    ...prevState, // 기존 상태 유지
    [name]: value, // 해당 필드만 업데이트
  }));
};

useEffect(() => {
  const fetchJournalData = async () => {
    try {
      const response = await fetch(
        `/api/journalDetail?date=${formattedDate}&type=${type}&id=${id}`
      );
      if (!response.ok) {
        throw new Error("Failed to fetch journal data");
      }
      const data = await response.json();
      setJournalData(data);
      setBackgroundColor(data.colorPicker || "#F5F5F5");
    } catch (error) {
      console.error("Error fetching journal data:", error);
    }
  };

  fetchJournalData();
}, [decodedDate]);
```

Within the WritingJournal component, the useEffect and handleChange functions collaboratively manage the synchronization between external data and internal component state. One useEffect hook is specifically responsible for monitoring changes to the journalData object. When journalData is updated—typically after an asynchronous fetch—this hook ensures that the corresponding state object, formData, is accurately populated with the latest journal values such as title, context, bookTitle, coverId, author, colorPicker, email, and candy. This design ensures that any modifications or pre-filled content from existing entries are immediately reflected in the form.

The handleChange function is used as a generic input handler for controlled form elements. It listens for changes in form inputs, extracts the name and value of the modified field, and updates the appropriate key in the formData state. This approach promotes reusability and scalability, allowing multiple fields to be managed uniformly without requiring individual handlers for each input.

Another useEffect hook is responsible for fetching the journal's data when the component first mounts or when the decodedDate value changes. It sends a request to the /api/journalDetail endpoint using the formattedDate, type, and id as query parameters. Upon receiving a successful response, the data is stored in journalData, which in turn triggers the first useEffect to populate the form. Simultaneously, the journal's associated color is applied to the component's background using the setBackgroundColor function. This sequence ensures that the component is visually and functionally aligned with the corresponding journal entry from the database.

GhostWriter: Code

Writing Journal

```
const handleBlur = async () => {
  const searchTerm = formData.bookTitle; // bookTitle을 사용
  const encodedTerm = encodeURIComponent(searchTerm).replace(/%20/g, "+");
  try {
    const response = await fetch(
      "https://openlibrary.org/search.json?title=" + encodedTerm
    );
    const data = await response.json();
    if (data.docs && data.docs.length > 0) {
      const newCoverId = data.docs[0].cover_i;
      const newAuthor = data.docs[0].author_name
        ? data.docs[0].author_name[0]
        : "Unknown"; // 저자 이름 가져오기
      setFormData((prevState) => ({
        ...prevState,
        coverId: newCoverId, // coverId 업데이트
        author: newAuthor, // author 업데이트
      }));
    } else {
      alert("No book cover found.");
    }
  } catch (error) {
    console.error("Error fetching book data:", error);
  }
};

// 배경색 변경을 처리하는 콜백 함수
const handleColorChange = (color) => {
  setBackgroundColor(color.hex);
  //document.getElementById(journalBox).style.backgroundColor(color.hex);
};

const handleFocus = (value) => {
  if (value == "title") {
    setTitleValue(""); // 포커스가 잡히면 값이 'title'일 때만 지우기
  }
  if (value == "context") {
    setContextValue("");
  }
  if (value == "bookTitle") {
    setBookValue("");
  }
};
```

The handleBlur function is responsible for retrieving book metadata from the OpenLibrary API based on the title input by the user. When the input field for the book title loses focus, this function is triggered. It encodes the book title into a URL-safe format and sends a GET request to the OpenLibrary search endpoint. Upon receiving a valid response, it checks whether there are available search results. If so, it extracts the coverId and author_name from the first result and updates the formData state accordingly, allowing the corresponding cover image and author to be displayed automatically. If no matching results are found, the function alerts the user that a cover image could not be retrieved. Error handling is also included to log any issues encountered during the fetch operation.

The handleColorChange function serves as a callback that responds to changes in the selected background color. When a user interacts with the color picker component and selects a color, this function is called with the chosen color object. It updates the backgroundColor state using the color's hexadecimal value, which in turn dynamically alters the background styling of the journal component. This allows users to personalize the appearance of their journal entries with visual context.

The handleFocus function enhances the user experience by clearing placeholder-like default values from input fields when they receive focus. When invoked, it determines which field is being interacted with—such as title, context, or bookTitle—and resets the corresponding state variable to an empty string. This prevents users from having to manually delete default prompt values and ensures a smoother and cleaner input process.

GhostWriter: Code

Writing Journal

```
//사탕
const handlePutCandy = async () => {
  try {
    const response = await fetch("/api/addCandy", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ id }), // ObjectId로 넘기기
    });

    if (response.status === 409) {
      const data = await response.json();
      alert(data.message);
      return;
    }

    if (!response.ok) throw new Error("Candy update failed");

    const updated = await response.json();
    setFormData((prev) => ({
      ...prev,
      candy: updated.candy,
    }));
  } catch (err) {
    console.error("Failed to add candy:", err);
  }
};
```

The handlePutCandy function is responsible for handling the process of gifting a candy to a journal entry. This asynchronous function is triggered when a user opts to give a candy, typically through a button click. It begins by making a POST request to the /api/addCandy endpoint, passing the journal's unique identifier (id) in the request body as JSON.

If the server responds with a 409 Conflict status code, it indicates that the current user has already given a candy to the specified journal. In such a case, the function extracts the server's message and alerts the user accordingly, then exits early without proceeding further.

If the response status is not OK and not a conflict, the function throws an error indicating the candy update has failed. Otherwise, upon a successful response, it parses the returned data, retrieves the updated candy count, and updates the formData state to reflect the new number of candies. This ensures that the UI remains consistent with the backend data following a successful operation. Error handling is included to log any unexpected failures during the network request or response handling.

GhostWriter: Code

Trick or Treat

```jsx
import { useEffect, useState } from "react";
import Image from "next/image";
import Link from "next/link";

const JournalList = () => {
  const [journals, setJournals] = useState([]);

  useEffect(() => {
    // API Route에서 데이터 가져오기
    fetch("/api/getJournal")
      .then((res) => res.json())
      .then((data) => {
        setJournals(data);
      });
  }, []);

  const formatDate = (dateString) => {
    const date = new Date(dateString); // journal.date 문자열을 Date 객체로 변환
    return date.toLocaleDateString("en-US", {
      weekday: "short", // 요일
      day: "2-digit", // 날짜
      month: "short", // 월 (Jun, Jul 형식)
      year: "2-digit", // 연도 (24 형식)
    });
  };

  return (
    <div className="listBox">
      <ul className="listUl">
        {journals.map((journal) => {
          const formattedDate = formatDate(journal.date); // 날짜 변환
          const encodedDate = encodeURIComponent(formattedDate); // URL 인코딩
          const id = journal._id; // URL 인코딩

          return (
            <li key={journal._id} className="listLi">
              <Link
                href={{
                  pathname: `/myJournal/writingJournal/${encodedDate}`,
                  query: { type: "read", id: id },
                }}
              >
                <div>
                  <p
                    className="journalColor"
                    style={{ backgroundColor: journal.colorPicker }}
                  ></p>
                  <p className="journalTitle">{journal.title}</p>
                  <div className="candy_box">
                    <Image
                      src="/images/mint.png"
                      alt="candy"
                      width={20}
                      height={20}
                      className="candy_image"
                    />
                  </div>
                  <p className="candy_num">{journal.candy}</p>
                </div>
              </Link>
            </li>
          );
        })}
      </ul>
    </div>
  );
};

export default JournalList;
```

the component uses the useEffect hook to make an asynchronous request to the /api/getJournal API endpoint. This endpoint is expected to return an array of journal objects. Once the data is received and parsed, it updates the local component state (journals) using the useState hook.

Each journal object includes details such as the date it was written, the journal title, the color associated with the journal (used for display), and the number of ″candies″ received—an interaction feature of the platform.

For each journal, the formatDate helper function converts the journal's date string into a human-readable format (e.g., Fri, 12 Jul 24) and applies URI encoding for safe usage in dynamic route paths.

In the return statement, the component maps over the journals array and renders each item inside a styled <li> element. Each journal entry is wrapped in a Link component from Next.js, allowing users to navigate to the journal reading view via dynamic routes. The route includes the formatted date as a path parameter and the journal ID and type (read) as query parameters.

Visually, each journal list item displays:

- A color-coded indicator (colorPicker value),
- The journal′s title,
- A mint candy icon (mint.png),
- And the number of candies the journal has received.

This component enables users to browse and revisit their past journals in a structured and visually engaging format.

GhostWriter: Code

Trick or Treat

```
// pages/api/addCandy.js
import { connectDB } from "@/util/database";
import { ObjectId } from "mongodb";
import { getToken } from "next-auth/jwt";

export default async function handler(req, res) {
  if (req.method === "POST") {
    const { id } = req.body;
    console.log("id" + id);
    if (!id) return res.status(400).json({ message: "ID required" });

    const token = await getToken({ req });
    const db = (await connectDB).db("ghostwriter").collection("journal");
    const db2 = (await connectDB).db("ghostwriter").collection("candyStore");
    const objectId = new ObjectId(id);

    const alreadyGave = await db2.findOne({
      journal: id,
      giver: token.email,
    });

    if (alreadyGave) {
      return res
        .status(409)
        .json({ message: "You have already given candy to this Journal!" });
    }

    const result = await db.findOneAndUpdate(
      { _id: objectId },
      { $inc: { candy: 1 } }
    );

    if (result.value) {
      await db2.insertOne({
        journal: id,
        email: token.email,
        candy: result.value.candy + 1,
        givenAt: new Date(),
      });
      res.redirect(302, "/trickOrTreat");
    } else {
      res.status(404).json({ message: "Journal not found" });
    }
  } else {
    res.status(405).json({ message: "Method not allowed" });
  }
}
```

This API route is responsible for handling the logic behind the *"put candy"* feature in the Ghostwriter application, allowing users to reward others' journal entries with a candy.

When a POST request is made, the handler first retrieves the journal entry's ID from the request body and checks for its validity. It then uses the getToken function to authenticate the user and retrieve the email address associated with the current session.

The system connects to two MongoDB collections: one for journal entries and another (candyStore) for logging each candy given. Before proceeding, the handler checks whether the user has already given a candy to the selected journal by searching for a record in the candyStore collection that matches both the journal ID and the user's email. If a record exists, a 409 Conflict response is returned to prevent duplicate gifting.

If the user has not given candy to this journal before, the handler increases the candy count of the corresponding journal document using the $inc operator. It then records this action in the candyStore collection, saving the journal ID, the user's email, the updated candy count, and the timestamp of when the candy was given.

If the update is successful, the user is redirected to the /trickOrTreat page. If the journal entry cannot be found, a 404 Not Found response is returned. If a method other than POST is used, the API responds with 405 Method Not Allowed.