ChatGPT

Plan for Building a Singapore Events Map Website

Project Overview

You want to create a web platform that displays **all events happening in Singapore on a given day**, visualized on an interactive map. Key features include: a map view of events (filterable by date), user accounts with authentication, the ability for users/organizers to submit events (with paid listing fees for monetization), a daily email newsletter of events, and a sleek **Apple-styled UI** that works well on desktop and mobile. In essence, this will be an event aggregation and discovery website for Singapore, with built-in monetization streams.

Example of a map interface with event markers (illustrative concept).

This project will involve both front-end and back-end development. As a solo junior developer, it's important to choose a tech stack that is **easy to develop and maintain**, with plenty of documentation and community support. Below we outline the recommended tech stack and a step-by-step approach to building the site, along with details on each component.

Tech Stack Summary

Front-End: Next.js (React framework) – for building the website UI and enabling server-side rendering for SEO.
UI Design: Tailwind CSS (utility-first CSS) – to quickly style pages with an **Apple-like design** (clean, minimal, with iOS-inspired elements).
Interactive Map: Google Maps API (via the `@react-google-maps/api` library) or Mapbox GL JS – to display an interactive map with event markers.
Authentication: Clerk – third-party auth service to handle user sign-up, login, and account management with minimal code.
Back-End: Next.js API Routes (Node.js serverless functions) – to handle backend logic (event queries, submissions, payments) within the Next.js app. No separate server is needed for a small project, which keeps things simpler.
Database: PostgreSQL – for storing users and events data. Use a hosted solution like Supabase (managed Postgres) or Neon, so you don't worry about server setup.
Payments: Stripe – for processing payments (e.g. charging users to list an event). Stripe is developer-friendly and supports one-time payments and subscriptions.
Email Service: SendGrid (or similar) – to send out daily email newsletters of events. Coupled with a cron job (scheduled function) to automate daily sends.
Deployment: Vercel – ideal for hosting Next.js apps, with support for serverless functions and scheduled cron jobs. Vercel's free tier will likely suffice during initial development and it makes deployments straightforward.

Below we dive into each of these aspects in detail, explaining how to implement them and why they're chosen.

Front-End Framework: Next.js (React)

Using **Next.js** for the front end gives you a robust React framework with built-in support for server-side rendering (SSR) and static site generation. SSR is particularly beneficial for an event listing site because it improves SEO – pages are pre-rendered on the server so search engines can index the content easily [1]. This means if someone searches for "events in Singapore on [date]", your site's pages (with event listings for that date) have a better chance of appearing in search results. Next.js will also handle routing for different pages (e.g. an events page, a submission form page, etc.) and can generate dynamic routes if you have individual event detail pages.

Other reasons to use Next.js include:
- **Familiar React Syntax:** You can build your UI with React components as usual.
- **File-Based Routing:** Create pages by simply adding files in the `pages/` (or `app/`) directory. For example, you might have `pages/index.js` for the homepage (showing today's events), `pages/[date].js` for listing events on a specific date, etc. Next will handle the routing.
- **API Routes:** Next.js lets you define backend endpoints under `pages/api/*.js`. You can write server-side code here (for example, an API route to fetch events from the database, or an API endpoint to handle form submissions or Stripe webhooks). This means you **don't need a separate backend server**, reducing complexity.

Next.js also makes it easy to create a responsive web app that works on mobile browsers. You'll build the UI in a responsive way (more on UI design below), and Next's pages will be accessible via mobile easily. You can even turn it into a Progressive Web App later (so users can "Add to Home Screen" on mobile) if desired, but initially focusing on a responsive website is sufficient.

TypeScript is optional but recommended for a project like this to catch errors early; Next.js supports TypeScript out-of-the-box if you want to use it [2]. For a junior dev, starting with JavaScript is fine, but as the project grows you might gradually introduce TypeScript.

UI Design: Apple-Styled, Responsive Interface

You specified an **"Apple styled UI"**, which we interpret as a clean, modern design inspired by Apple's Human Interface Guidelines. This often means: a focus on simplicity and clarity, ample white space, crisp typography (using Apple's San Francisco font if possible), and **glassmorphism** effects (blurred translucent backgrounds) to mimic iOS aesthetics. For example, Apple's design might include frosted-glass panels, subtle shadows, and rounded corners. An excerpt from an Apple-inspired design brief: *"The design system has to be the same as iOS… White theme with a colored blur in shades of white and blue. Everything must be in glassmorphism with transparency and contrast."* [3].

To achieve this on the web, **Tailwind CSS** is a great choice. Tailwind will let you easily apply styles like `backdrop-blur` (for translucent backgrounds), `bg-white/50` (white background with 50% opacity), rounded corners, etc., without writing a lot of custom CSS. You can also utilize pre-built components or examples from the Tailwind community that emulate iOS style. Some tips for Apple-like design:

- **Typography:** Use a clean sans-serif font. Apple uses "San Francisco" for its platforms. On the web, you might use `-apple-system` font family (which uses San Francisco on Apple devices and fallback fonts on others) in your CSS for a similar feel. Tailwind's default font family already includes `-apple-system`.
- **Color Palette:** Stick to neutral tones, whites, grays, with **accent colors** similar to macOS or iOS (blue accents for interactive elements, etc.). Apple's palette tends to be minimal and consistent.
- **Glassmorphism:** Implement translucent panels. For example, a navbar that is slightly transparent with blur so content behind shows through, similar to iOS Control Center. Tailwind can do this with classes like `backdrop-blur-md` and background opacity classes.
- **Icons and Buttons:** Use simple, thin icons (you can use an icon set like Feather or Apple's SF Symbols (via a web-friendly set) for a similar look). Buttons and toggles should be clean – possibly use Tailwind to style buttons with rounded-full, subtle hover effects, etc., rather than using something like Material UI which has a different aesthetic.
- **Layout:** Keep it uncluttered. Use plenty of spacing (Tailwind's spacing scale) to avoid crowding elements. Perhaps use cards or panels with subtle shadows to separate sections.

Ensure the site is **fully responsive** so that it works on mobile screens. Next.js + Tailwind makes this straightforward: design mobile-first and use Tailwind's responsive utilities (like `md:` prefixes for larger screens). The map component can be made responsive (e.g., taking the full width of the screen on mobile, maybe a smaller height, etc.). You might have a layout where on desktop the map is shown alongside a list of events, but on mobile the map might be on top and list below, or the map could even be full-screen with points that can be tapped.

By prioritizing a responsive web design, the "web website which can be used for mobile too" requirement (point 4) will be met. Users on smartphones should have a native-app-like experience, with touch-friendly UI elements (make buttons large enough, use mobile viewport meta tags which Next includes by default, etc.).

Interactive Map Integration

Displaying events on an interactive map is a core feature. There are two primary options: **Google Maps** or **Mapbox** (or any other map library, but these two are the most popular and well-supported).

- **Google Maps API:** Google Maps is ubiquitous and familiar to users. For web integration, you can use the Google Maps JavaScript API. A convenient way in a React app is to use the `@react-google-maps/api` library, which provides ready-to-use React components for Google Maps. For example, you can create a Map component like:

```
import { GoogleMap, LoadScript, Marker } from '@react-google-maps/api';
// ...
<LoadScript googleMapsApiKey={process.env.NEXT_PUBLIC_GOOGLE_MAPS_API_KEY}>
  <GoogleMap center={centerLatLng} zoom={12}
mapContainerStyle={{height:'400px', width:'100%'}}>
    <Marker position={centerLatLng} />
    {/* Add Marker components for each event's coordinates */}
  </GoogleMap>
</LoadScript>
```

This shows how you wrap the `GoogleMap` in a `LoadScript` with your API key, set the center and zoom, and add markers [4] [5]. You would dynamically generate a `<Marker>` for each event (each marker could include a tooltip or info window with the event name/short info on click). The Google Maps API also allows clustering markers if there are many, and customizing marker icons (you could use custom icons to fit your design).

To use Google Maps, you'll need to obtain an API key from Google Cloud Platform and enable the Maps JavaScript API [6]. Google offers a free tier with a certain number of map loads per month which is usually sufficient for development and initial launch (check current Google Maps API pricing for the free credit limits). Be sure to secure the API key (restrict it to your domain). The `@react-google-maps/api` documentation and examples [7] [8] will be helpful as you implement this.

- **Mapbox GL JS:** Mapbox is an alternative that provides more custom-styled maps and doesn't require Google's ecosystem. Mapbox has a free tier for up to a certain number of map loads. You can use Mapbox's **GL JS** library directly in React or use a React wrapper (like `react-map-gl`). Mapbox might be beneficial if you want a map style that matches your Apple UI theme – for example, Mapbox Studio allows you to design a custom map style (you could create a light theme or even a style that looks similar to Apple Maps). However, using Mapbox will require including their script and CSS, and an access token.

If an **Apple Maps** aesthetic is desired and you have an Apple Developer account, there is also **MapKit JS** (Apple's map JS API), but it requires an Apple developer key and isn't as straightforward for beginners as Google or Mapbox. For simplicity, Google Maps is recommended – it's well-documented and the React integration is straightforward. You can later adjust the Google Maps styling (Google offers map style JSONs – e.g., you can find a style that is grayscale or minimal to fit your design). The example image provided was a grayscale map; you can achieve a similar look by applying a custom style to Google Maps (using the Styled Maps feature with a JSON style object).

Implementing the Map: Start with a simple map centered on Singapore. Singapore's lat/long roughly is (1.3521 N, 103.8198 E). Set that as the center. Then ensure you can plot a single sample marker. After that, integrate it with your event data: fetch the list of events (for the selected date) and for each event render a `<Marker position={{lat, lng}} />`. You might also implement clicking on a marker to see details (Google Maps API InfoWindow, or you can handle clicks in React and perhaps show a sidebar or modal with event info).

Handling Geolocation: Each event in your database will need coordinates (latitude & longitude) to plot on the map. If your event entry is an address (like "Marina Bay Sands, Singapore"), you'll need to convert that to coordinates (geocoding). Google has a Geocoding API, and Mapbox has a Geocoding API as well. An approach could be: when an event is added, use a geocoding service to get its lat/lng and store those in the DB. This way map loading is faster (no need to geocode on each page load). Google's Geocoding API is straightforward (though has a rate limit). Alternatively, use an open API like OpenStreetMap's Nominatim for geocoding to avoid charges. But given a low volume of event additions, geocoding costs should be low.

User Authentication with Clerk

For user accounts and authentication, **Clerk** is an excellent choice because it offloads the complexity of auth. Clerk provides pre-built components and handles everything from password storage to social logins. As a junior dev going solo, this saves you from having to implement secure auth on your own (which can be error-prone).

Setting up Clerk involves: creating a Clerk application on their website, obtaining your Frontend API key and other IDs, and then integrating their React components into your Next.js app. Clerk has a Next.js SDK that works seamlessly with the Next App Router or Pages Router. According to a developer who tried it: *"Clerk has a great suite of components for Next.js apps... The first step was to bootstrap a Next site and get authentication set up so I can have users sign into it."* [9]. Clerk supports social login providers too; for example, you can enable Google or Facebook login with a few config tweaks on Clerk's dashboard, and the UI will offer those options.

Integration Steps:
1. **Install Clerk:** You'd add Clerk's package (`@clerk/nextjs`) to your Next.js project.
2. **Environment Variables:** Add your Clerk frontend API key and Clerk JWT template (if using) to your `.env.local`. These keys are provided when you set up your Clerk app [10].
3. **Next.js Middleware:** Clerk recommends using their middleware to protect routes. You create a `middleware.ts` in your project root with `import { authMiddleware } from "@clerk/nextjs"; export default authMiddleware({ /* options */ });`. This will enable Clerk's session handling. You can specify which routes to protect or public. Likely, you want most of the site public (browsing events doesn't require login), but features like **submitting an event or managing an event** should require login. So you'd configure the middleware to make routes like `/submit-event` or `/my-account` require auth, while the main pages remain public. Clerk's docs show how to configure public routes explicitly [11].
4. **Wrap the App:** In Next.js App router, you wrap your `<App>` with `<ClerkProvider>`. In Pages router (if using `_app.js`), similarly you wrap the app with Clerk's provider. This ensures Clerk is initialized and can use context in your components [12].
5. **Add Components:** Clerk provides React components like `<SignIn />`, `<SignUp />`, or even pre-styled buttons like `<SignInButton/>`. You can have a sign-in page (just render `<SignIn />` and Clerk handles the form, validation, etc.), or use Clerk's <UserButton/> to show the logged-in user's avatar and menu. For example, one can simply add `<SignUpButton />` to the page to get a "Sign Up" button; clicking it triggers Clerk's hosted sign-up UI, and after completion the user is signed in [13]. Clerk then redirects back and you have the user session. It's that easy – *"Clicking that will bring me to Clerk's UI to create or sign in... After signing in, I'm redirected back"* [14].

Clerk manages the user data, so you do not store passwords or handle authentication logic on your server. However, you can still integrate Clerk users with your database: each Clerk user has an ID, which you can use as a foreign key in your events table to know which user posted which event. Clerk provides APIs to fetch the user's info (client-side or server-side). In Next.js API routes or server components, you can verify the user via Clerk's JWT or use Clerk's server SDK to get the user if needed. But often, simply storing the Clerk `userId` string in your event records is enough to associate events with users.

Authorization: If you plan roles (like normal user vs admin), Clerk doesn't enforce that, you'd handle it in your app logic. For example, you might have a field in your database or a Clerk metadata that flags an account as an admin who can add events without payment or approve events, etc. As a starting point, focus on basic authentication (account creation and login), and you can add roles/permissions as needed later.

Database and Data Management

For storing data, a **relational database (PostgreSQL)** is ideal for structured data like events and users. You'll have at least two main tables: one for **Events** and one for **Users** (plus potentially others for subscriptions to the newsletter, payments, etc.).

Since you're using Clerk for authentication, you might not need to store user credentials in the database (Clerk does that). But you can still have a Users table for profile info or preferences if needed (or you could rely on Clerk's user object for basics like name, email, etc.). At minimum, you'll store the Clerk user ID alongside events to know who submitted them.

Using a cloud-hosted DB service will simplify your work – you don't want to maintain a DB server yourself. **Supabase** is a great choice: it provides a hosted Postgres database and a convenient UI to manage it, plus it can directly integrate with Next.js. Supabase is essentially "an open-source alternative to Firebase" and can handle your backend needs including the Postgres DB, storage, etc [15]. With Supabase, you get a RESTful API for your tables out-of-the-box and can also use their JavaScript client library to query the database. However, since you're already using Next.js API routes, you can also connect to the Postgres DB using a traditional approach (like an ORM or query builder) within your API routes.

Possible approaches for the database:

- *Option A:* Use **Supabase** and its JS client in your Next.js app. You'd create your tables via the Supabase web UI or SQL, then from your Next.js frontend or API routes, call Supabase's client to fetch or insert data. Supabase also has Row-Level Security rules which you can configure to secure data. Supabase can handle auth too, but since we use Clerk, we'd mainly use Supabase for database and perhaps storage (if you want to store images, though events likely just have text info and maybe an image URL). Supabase's free tier is generous and should cover development and initial usage.

- *Option B:* Use a **direct Postgres** service (like Neon.tech or Railway). For example, Neon offers a free Postgres and is very developer-friendly. You'd then use an ORM like **Prisma** or a query builder (like Knex.js) in your Next.js API routes to interact with the DB. Prisma in particular can simplify database interactions and migrations, but it does add a bit of setup. If you prefer minimal setup, you could even use Next.js's built-in support with the `pg` library to make SQL queries in API routes.

Given your experience level, Supabase might be slightly easier since it has a GUI and docs for integrating with Next.js. It also automatically provides APIs if you ever want to use them, and it can send realtime updates (not that you need realtime now, but nice to have). If you go with Supabase, you'd initialize it with your Supabase URL and anon key (from the Supabase dashboard) in your Next app. Supabase's JS library can be used on the client side (for maybe reading public data) or server side with the service role key (for protected operations).

Data Schema Design: Let's outline what the **Event** data might include. An **Events** table could have columns: - `id` (primary key)

- `title` – name of the event
- `description` – details about it
- `event_date` – the date (and maybe time) of the event
- `location_name` – text of venue name or address
- `latitude` and `longitude` – for map plotting
- `category` – e.g., concert, exhibition, etc., if you want classification
- `created_by` – a reference to the user who added it (could store Clerk user ID or a foreign key to a Users table if you mirror users in DB)
- `created_at` – timestamp
- `is_approved` – boolean (if you want to manually approve events before they show up, could be useful if the platform is moderated)
- `is_paid` – boolean or enum (to mark if the listing fee was paid, or what type of listing it is: free, paid-standard, paid-featured)

The **Users** table (if used) might have: - `id` (primary, could map to Clerk user ID or have your own and store Clerk ID as another field)
- `name`, `email` (though you get these from Clerk as well)
- `clerk_user_id` (if your primary key is separate)
- `is_admin` (bool for admin rights)
- `subscribe_newsletter` (bool if they want the daily email, though you might also manage this via a separate table or even via an external email service list)

Because Clerk manages auth, you might not need a full Users table. Instead, you might only store additional info about users as needed. Clerk provides user profile data via their API, which you can query whenever you need (or you can use Clerk webhooks to sync data to your DB if needed). To keep it simple, you might start without a Users table at all, and just store the `user_id` string on events. Later if you need to store something like “user’s preferences” or “plan type”, you can introduce a Users table or use Clerk’s user metadata.

Database queries: With Next API routes, you’ll create endpoints like `/api/events` that returns events (possibly filtered by date). This API route will connect to the database (via Supabase client or via an ORM) and run a query for events on the requested date, then return JSON. Your front-end page can call this API (using `fetch` or SWR or any data fetching method) to get the events and then render them (including rendering the map markers).

Example: To get events for a given date, you might implement in an API route:

```
// pages/api/events.js (if using Pages router)
import { supabase } from '../../lib/supabaseClient';

export default async function handler(req, res) {
  const { date } = req.query;
  const { data: events, error } = await supabase
    .from('events')
    .select('*')
    .eq('event_date', date);
  if (error) return res.status(500).json({ error: error.message });
  res.status(200).json({ events });
}
```

If using the App router and React Server Components, you could also directly query the DB in a server component using Supabase or an ORM (since Next 13 allows that), but for clarity, using API routes or route handlers might be easier to start with.

Event Data: Automatic Aggregation vs. User Submissions

The project scope says “shows all the events in Singapore today”. A challenge here is acquiring all these event listings. There are a few approaches:

1. **Automatic Data Aggregation (point 1: “automatically”):** You can integrate with external data sources to populate events without requiring every event to be user-submitted. For Singapore, the **Singapore Tourism Board (STB)** provides an API called the Tourism Information Hub (TIH) that includes an Events dataset. Developers can request an API key to access various tourism-related data including events [16]. Using this API, you could fetch official events (e.g., concerts, exhibitions, etc. that are in their database) and display them. The TIH API has endpoints like `Get Events` which returns data for events given an ID or allows listing by datasets [17]. This could serve as a starting database of events. You would need to consult their docs on how to query events by date or get all upcoming events – likely, they have an endpoint or you may need to fetch all and filter by date. Keep in mind the API usage limits and ensure you cache or store the results to avoid excessive calls.

Using the API: After obtaining an API key from STB, you’d call their events endpoint (which likely returns a JSON of events). You might set up a script or cron job to regularly pull new events from this source (say, daily or weekly) and insert them into your database. This way your site stays populated **automatically** with a baseline of events. This addresses the “automatically show all events” requirement to an extent. However, STB’s data might not include every small event (like community meetups, private events, etc.). For comprehensiveness, you might still allow users or organizers to submit events that aren’t captured elsewhere.

1. **User Submissions (with Payment):** You definitely want a feature where users (event organizers or just users who know of events) can submit events to list on the site. This is also tied to monetization – you plan to **charge a fee for listing an event** (point 2). The workflow would be: if a user has an event to advertise, they would create an account (or log in), fill out a form with event details, and then pay a fee to have it listed. Upon payment, the event becomes visible on the site (perhaps immediately or after admin approval). We will discuss payments in the next section.

For implementing submissions:
- Create an **Event Submission Form** page (protected by login, so only logged-in users can access). This form would collect all necessary details (title, description, date & time, venue, possibly a category tag, maybe an image or link, etc.).
- On form submit, you likely **create a record in the database in a “pending” state**, and then redirect the user to pay (or initiate a payment process). Alternatively, you integrate the payment into the form submission flow (more on that under Payments).
- If payment is successful, mark the event as `is_paid = true` and active in the DB. If you want to allow

free listings for some events (maybe non-profits or small events), you could have a pathway for free submission that goes to an approval queue instead (up to you). But the requirements sounded like listing is primarily paid.

You might also implement an **admin interface** (maybe just via a database UI or a simple admin page) to manage events – e.g., remove inappropriate ones, or approve events if you hold them for review. As a solo dev, initially you can monitor via the DB directly.

1. **Combining Both:** A good strategy could be to seed your database with automatically fetched events from an API (so the site has content from day one), and then allow additional user-contributed events via the submission flow. This way, "all events" can cover both official sources and community-added events. You could even integrate other sources: for example, many event organizers use Eventbrite or Meetup. While Eventbrite's public API for searching events was deprecated [18] , one could still periodically scrape or use RSS feeds from various sources. However, that can be complex and beyond a junior dev's scope initially. Focus on one source (like the STB API) for automatic data, and rely on user submissions for the rest.

In summary, **to cover point (1) "automatically"**: set up a pipeline to fetch events from a reliable source like a government or open data API. And for user-driven content, implement the submission form with payment.

Monetization and Payments (Charging for Listings)

Monetization is planned through **paid event listings** (point 2 and 3). This means when a user wants to add an event, they must pay a fee. Additionally, you plan to offer a **free daily email** of events – this is a user benefit, and potentially a future monetization angle (like sponsorships in the email or a premium tier for ads-free email, etc.). Let's break down the monetization:

Paid Listings (Charging Users to Add Events)

To implement payments on the site, **Stripe** is the go-to solution. Stripe allows you to securely handle payments with minimal effort from your side, and it supports one-time payments (and subscriptions if ever needed). For your use case, a one-time fee per event listing is probably what you want. For example, you might charge a fixed amount (say S$X per event listing) or maybe have tiers (standard listing vs "featured" listing that might highlight the event).

Stripe Integration: In Next.js, you can use Stripe's SDK in your API routes to create checkout sessions or payment intents. One of the simplest methods is to use **Stripe Checkout**, which is a pre-built checkout page hosted by Stripe. Stripe Checkout can handle the payment process (including card details, Apple Pay, etc.) so you don't have to build a custom payment form. According to Vercel's guide, *"Stripe Checkout is the fastest way to get started with Stripe and provides a Stripe-hosted checkout page with various payment methods supported out of the box"* [19] . The way this would work: when a user submits the event form, your code calls Stripe to create a Checkout Session (with the amount to charge). Stripe responds with a URL for the checkout. You then redirect the user to that URL. The user completes payment on Stripe's site, and Stripe can then redirect back to your site (success or cancel URL). On success, you finalize the event listing.

Concretely, steps:
1. Install Stripe's Node SDK for your API routes (`npm install stripe`). Also include Stripe's client JS for any client-side actions (though if using Checkout redirect, you might not even need the client library except to possibly confirm things).
2. Set up Stripe account and get API keys (publishable and secret key). Add those to your `.env.local` (never expose secret key to client). [20]
3. Create an API route for checkout. For example, `POST /api/checkout_session` which takes something like `{eventId}` or `{listingFee}` in the request. In this handler, use `stripe.checkout.sessions.create()` with parameters such as `line_items` (with price or amount), `success_url` and `cancel_url` (pointing to your site). This returns a session object with an `id` or `url`. You then send the `url` back to the frontend.
4. On the frontend, after the user hits "Submit and Pay" on the event form, you call this API (e.g., using `fetch`). When you get the session URL, you can either redirect via JavaScript: `window.location = sessionUrl;` which sends them to Stripe's checkout. Alternatively, Stripe has a slightly different approach if using Stripe Elements, but Checkout is simpler for now.
5. On Stripe's side, after payment, it will redirect to the `success_url` you specified. You can include query params like the session id or your event id in that URL. For example, `success_url: yoursite.com/submit-success?session_id={CHECKOUT_SESSION_ID}`. After redirect, you'd have a page that checks the session id (you can call Stripe's API to verify the payment succeeded, or listen for a webhook as a more robust solution).
6. Upon successful payment confirmation, mark the event as paid/active in your database. If you created the event entry as pending before, now you update it. If you waited to create it until after payment, you can create it now with an active status. But better to create before payment and just toggle a field after payment (so you don't lose the submitted info if payment fails or is abandoned; you can then remind the user or allow them to retry).
7. Optionally, send a confirmation email to the user or a notification to admin about the new event listing.

Because handling post-payment logic can be complex, you might also consider using **Stripe Webhooks**. A webhook is Stripe's server-to-server notification to your app when a payment is completed. You can set up a webhook endpoint (`/api/stripe-webhook`) that Stripe calls with event data (like `checkout.session.completed`). In that handler, you verify the event (Stripe signs it) and then retrieve the session, find the associated event (you might store the event ID in the `metadata` of the Stripe session when you create it), and then mark it paid. This ensures even if the user doesn't return to your site (say they close the tab at the thank-you page), your system still knows the payment succeeded. However, webhooks add complexity (you need to handle secret verification, etc.). Given this is MVP and a small scale, you might opt to simply rely on the user returning to the site for now. Just keep it in mind as a future improvement for reliability.

To implement Stripe safely in Next.js, recall that any sensitive operations (creating a checkout session, confirming payment intent, etc.) should happen in **API routes (server-side)**, not in client-side code, to keep your secret key safe. The client will only handle the redirect and perhaps the display of a payment form if you go that route (e.g., using Stripe Elements for a custom form – but that's more work and not necessary if using Checkout).

Also note: If you expect to handle local payments or other methods, Stripe can handle many methods by default (credit cards, Apple Pay/Google Pay, etc., via Checkout).

Pricing Strategy: Determine how you want to charge. For instance, a flat fee per event listing (e.g., S$5 per event). Or you could offer packages (e.g., 10 listings for a subscription fee, etc.). To keep it simple: start with a flat fee. You can set up a Stripe Price for this (in Stripe's dashboard, you can create a Product like "Event Listing Fee" and a Price of $X). Then in your checkout session creation, you reference that Price ID so that Stripe knows how much to charge. Alternatively, you can specify a one-time amount on the fly (useful if you want to dynamically set the fee). Since it's likely fixed, using a predefined Price ID is straightforward.

The Dev.to event project reference emphasizes payment integration with Stripe or PayPal and implementing the payment flow on frontend [21], underlining that Stripe is a common choice for this kind of functionality. Stripe also has good documentation and example Next.js integrations.

Daily Events Newsletter (Free Email Subscription)

You plan to send out a **daily email** listing the events. This can serve to engage users and can later be monetized (for example, you could allow sponsored events to be highlighted in the email, or eventually charge a subscription for premium features – but initially it's free as stated).

To implement this, you need two things: (a) a way for users to sign up for the newsletter, and (b) a system to send the email every day automatically.

Collecting Subscribers: Since users are creating accounts via Clerk, you have their emails by default. You could consider all registered users as email subscribers, but that might not be ideal (some users might register only to post an event and might not want newsletters). It's better to allow users to opt-in. Perhaps during sign-up or in their profile settings (you can create a simple toggle "Subscribe me to daily events email"). If you want to allow even non-logged-in people to subscribe, you could have a separate form for just entering an email to get the newsletter. That would be an unauthenticated subscription (you'd need to store those emails separately, perhaps in a `subscribers` table or even use a mailing list service).

To keep it straightforward: possibly make it so that any user account can opt in (store a flag in the DB or in Clerk user metadata), *and* provide a form for visitors to subscribe by email without full registration (store those in a `NewsletterSubscriptions` table with email and maybe date subscribed).

Sending Emails: Use an email service like **SendGrid** (or Mailgun, Amazon SES, etc.) to actually send the emails. SendGrid offers a free tier for a certain number of emails per day which might be enough initially. You'll need to obtain API keys and possibly verify your sender domain to avoid spam issues. Once set up, you can send emails via an HTTP API call or SMTP. The API approach (using SendGrid's Node library or a simple `fetch` to their endpoint) is typically easier in a serverless environment.

Email Content: The email would list all events for "today" (or for the next day, depending on what you decide). You'll query the events table for the date in question and format them into an email (could be simple text or HTML listing event names, times, and a link to the site for details). Keep the design simple for now (plain text or basic HTML, since responsive email design can be tricky – you can improve it later or use a template if needed).

Automation (Cron Job): To ensure the email goes out daily without manual intervention, you can set up a **cron job**. If deploying on Vercel, they now support cron jobs for serverless functions on all plans [22] [23]. This means you can configure in `vercel.json` a schedule for an API route. For example, you might create an API route `/api/sendDailyEmail` that when called, gathers today's events and sends the email out. In `vercel.json`, you schedule this route to run every day at a specific time (perhaps early morning). Vercel will invoke it automatically each day, triggering the emails. This is a great serverless way to do cron. Alternatively, if you were not on Vercel or needed more, you could use a separate cron service or even GitHub Actions scheduled triggers to hit your endpoint – but Vercel's built-in cron is easiest.

So the flow:
- Write the handler at `/api/sendDailyEmail`. In it, do something like: fetch all events where `event_date = today` (or whatever logic, maybe you send "today's events" or "tomorrow's events"). If you want the newsletter to contain events for the current day each morning, query for events of that date. Format the list into an email body.
- Use SendGrid (or another email API) inside this function: e.g., using SendGrid's Node library, compose a mail with subject "Events in SG on [Date]" and the body list. Set the recipients – which would be all subscribed emails. If the list is small, you could loop and send individually, but that's inefficient and may hit API limits. Better is to use SendGrid's feature to send to multiple recipients at once or use their marketing campaign features. However, for MVP, you can simply BCC all subscribers or send one email per subscriber in a loop (just be mindful of free tier limits, maybe start with one email to yourself for testing).
- The cron job triggers daily.

Using a service like SendGrid aligns with advice to *"use a service like SendGrid for email notifications"* [24]. Another approach could be using an email marketing platform like Mailchimp and integrate their API – but that might be overkill and their free tier is limited in number of contacts. Since you can code, using an email API gives more control.

Be sure to include an unsubscribe mechanism in the emails (especially if adding people without double opt-in). Perhaps an unsubscribe link that hits an API on your site to remove them from the list (or instruct them to uncheck the setting in their profile). This keeps your emails compliant with anti-spam laws.

Deployment and Hosting

Hosting Platform: Vercel is highly recommended for deploying this Next.js app. It's literally built for Next.js and offers easy CI/CD (continuous deployment) – you connect your GitHub repo and every push can auto-deploy. Vercel also supports environment variables (for your API keys), serverless functions (for API routes), and cron jobs as mentioned. The frontend will be served globally on their CDN, which means good performance. And Vercel has a generous hobby/free tier.

Deployment steps would be: push your code to Git (GitHub), create a Vercel project, add your environment vars in Vercel's dashboard (API keys for Clerk, Stripe, DB URL, SendGrid, etc.), and deploy. As per a guide, *"Deploy the Next.js application on Vercel… Connect your GitHub repository to Vercel for automatic deployments."* [25]. Once deployed, test that all features (map, auth, payment webhooks if any, email cron) work in the production environment.

If you use Supabase for DB, that's hosted elsewhere (Supabase will give you a connection string). Supabase's servers are hosted in the cloud (you can choose region, e.g., you might choose one in Asia for latency). There's no server to manage – just ensure your Vercel functions can reach the Supabase REST API or DB. Typically, it's fine. If using direct Postgres with an ORM, ensure the DB connection string is set and maybe use Prisma connection pooling (or use Vercel's Postgres connection if you opted for Neon through Vercel integration).

Domain: Vercel will give you a *.vercel.app domain by default. You can add a custom domain (e.g., yoursite.com) later. Not critical in development, but for a production launch, having a nice domain is good.

SSL: Vercel handles HTTPS automatically for custom domains. So no worry there.

Monitoring & Error Tracking: As a solo dev, it's good to set up something like Sentry (easy to add to Next.js) for error tracking, so you catch any runtime errors users encounter. Also monitor your Stripe dashboard and other services for any issues.

Development Roadmap (Step-by-Step for a Solo Dev)

Finally, let's outline a suggested step-by-step plan to build this project. Breaking it into manageable tasks will help you stay organized:

1. **Set Up the Project:** Initialize a new Next.js app. For example, run `npx create-next-app@latest events-map-app`. Test that it runs. Install Tailwind CSS and configure it (Tailwind's setup guide will have you add config files and import Tailwind in your globals.css) [26] [27]. Also, install any essential libraries you know you'll use, like `@react-google-maps/api`, `@clerk/nextjs`, etc. but you can also add them as you go. Initialize a git repository and commit the base setup.

2. **Implement Authentication (Clerk):** Follow Clerk's Next.js quickstart. Create a Clerk account and application. Install Clerk SDK, add the ClerkProvider in `_app.js` or layout, set up the Clerk middleware for protected routes. Create a basic sign-in page to verify it works. Test logging in/out with Clerk's default UI. At this stage, you'll have user auth working in the dev environment.

3. **Design the Database Schema:** Determine your data models (Events, Users or not, etc. as discussed). If using Supabase, create a Supabase project and define the tables (you can use their GUI Table Editor or SQL). If using Prisma with a direct Postgres, define the schema in `schema.prisma` and migrate. In either case, get a connection string and test connecting from a simple script or in Next API. Ensure you can insert and read a sample record. Don't worry about every field initially; maybe create a simple event with just title and date to test connectivity.

4. **Events Listing Page:** Build the page that shows events on a given date. This could be your homepage for "today's events". Start with static content if needed, then make it dynamic. For now, you might manually insert a couple of sample events in the DB (or a JSON) to test. Implement an API route `/api/events?date=YYYY-MM-DD` to fetch events. On the Next page, use `getServerSideProps` (if using Pages router) or `useEffect`/SWR in a client component (if App router with client component) to call that API and retrieve events. Display them in a list (title, time, etc.) to ensure data flow is working. Once that works, integrate the **Map** on this page: use the Google Map component to show markers. Ensure that if you click a marker, maybe it highlights the event in the list or shows a tooltip (you can refine this later).

5. **Map Integration:** Focus on getting the map UI right. Set the map's center to Singapore and a reasonable zoom. Use the sample events' coordinates to plot markers. If you don't have real coordinates, use placeholders (e.g., coordinates of some known places in Singapore). Test panning, zooming, etc. and how it looks on mobile (Google Maps should be responsive if container width is 100%, just ensure a good height on small screens).

6. **Event Submission Flow:** Create a page for adding a new event (e.g., `/add-event` or `/submit`). This page should only be accessible if logged in – you can enforce that by using Clerk's `SignedIn` component or by checking in `getServerSideProps` for a session (Clerk provides helpers). The form should include all necessary fields. You can use a controlled React form or a form library. On submit, call a handler (maybe an API route `/api/submit-event`). That handler will: if you are doing pay-to-list, possibly create the event in DB with a status "pending" and return some identifier or a Stripe session as response.

7. **Payment Integration (Stripe):** Set up Stripe keys in env and the checkout session creation as described. When the user submits the event form, you can have two stages: first, call an API to create the event record (get an ID), then create a Stripe Checkout Session for that event, then return the session URL, then redirect the user to it. Alternatively, your `/api/submit-event` could do both: insert event and create checkout session, then respond with the URL. In any case, get the Stripe Checkout working in test mode. Use Stripe's test card numbers to simulate a payment. Make sure after paying, it goes to your success page. On the success page, retrieve the session (either via query param or webhook) and update the event's status to active. This is one of the harder parts, so take it step by step with Stripe's docs. Test a few times the whole flow (from form to Stripe to back). Once working, this covers the monetization path (point 2 and 3 regarding "pay to list").

8. **Daily Email Cron:** Implement the daily email sender. First, code a function that can fetch today's events and compose an email. Perhaps make it an API route `/api/sendDailyEmail` that you can trigger manually for testing (by opening the URL or using curl). Integrate SendGrid API – set up a free account, get an API key, and use their Node SDK (`@sendgrid/mail`) or REST calls. Craft a simple email template (even just a newline-separated list of event names and times). Test sending an email to yourself (you might need to verify your sender identity on SendGrid, which typically means confirming an email or domain). Once you can successfully send an email via the API route, configure the scheduled job. In Vercel, you'd add to `vercel.json` something like:

```
{
  "cron": {
    "jobs": [
      {
        "path": "/api/sendDailyEmail",
        "schedule": "0 9 * * *"
      }
    ]
  }
}
```

This example would schedule it daily at 9:00 UTC (you can adjust cron time for Singapore morning). Vercel will call that endpoint on schedule, and your code sends out the emails. Test the cron by perhaps temporarily scheduling it every 5 minutes and see if it triggers (or manually invoke the function as mentioned).

Also create a way for users to subscribe/unsubscribe. For MVP, you might assume all registered users who opt in will get it. You can add a checkbox in the registration (Clerk might allow custom fields, or you handle it after sign up in a profile page) – but even simpler, you could start by having a public form just to collect emails for newsletter (store them in your DB table). Focus on the sending mechanism first, then add the nice-to-have of user managing preferences.

1. **Polish the UI:** Now that core functionality is in place (auth, map, events, payment, email), spend time making the site look good according to the Apple-style goal. Use Tailwind to refine typography, spacing, colors. Perhaps implement a nice navbar (with maybe the logo, and a user profile menu if logged in). Ensure the map and list layout is clean. Possibly add filters for category or a search bar to find events (could be a future enhancement if not immediate). This is also the time to test on mobile devices and make adjustments (Tailwind's responsive classes). Add any Apple-like touches: e.g., add a blur effect to the navbar when scrolling (using CSS `backdrop-filter`), make buttons with subtle hover animations, etc. These details will elevate the look.

Additionally, you might incorporate an **Apple Maps-like marker icon** (for example, a round dot or a custom pin icon that matches your theme) instead of the default Google Maps pin. Google Maps API allows custom marker images.

1. **Testing and Launch:** Go through every feature as a user: account creation, browsing events, filtering by date (you might implement a date picker to see events on other dates, which would call your API for that date), submitting an event and paying, receiving the confirmation, and receiving the next daily email. Ensure edge cases are handled (e.g., what if no events on a certain date – show a friendly "No events" message). Fix any bugs encountered.

   Once confident, deploy the latest to production (Vercel) and share with a small group of users or friends to beta test. Monitor your logs (Vercel function logs, any errors). Also monitor Stripe dashboard to ensure payments register, and your email sending to check if any bounces or issues.

2. **Monetization Enhancements (Future):** After initial launch, you can consider more monetization features: for example, **featured events** (an organizer pays extra to have their event highlighted or appear at top of list or on the map with a star icon). Or advertising slots on the site or in the email (but keep the user experience in mind). Also, as the user base grows, you might implement analytics to track which events are popular (views or clicks) – useful data you could potentially use to justify charging more for featured placement, etc. All these can be built over time once the core is running.

Throughout development, take advantage of the documentation and community for each tool: Next.js docs for any framework questions, Clerk docs for auth, Stripe docs for payment examples, and so on. There are many tutorials and examples (we cited some) that can guide implementation details. By choosing this stack, you have picked technologies that have a lot of support and integrations, which should help you overcome obstacles.

Finally, remember to maintain **comprehensive citations** and references for any content if this were a report (since the instructions above mention preserving citations in the output). In an actual development project, you'd keep documentation links handy, but for your planning purposes, we included references where appropriate to back up the recommendations.

Good luck with building your events map site! With this detailed plan and modern tools, you're set up for success to create a functional and monetizable application.

Sources:

- Next.js SSR improves SEO by pre-rendering dynamic pages [1]
- Apple-style design cues (glassmorphism, iOS-inspired layout) [3]
- React Google Maps API usage for interactive maps [4] [5]
- Clerk integration simplifies auth with prebuilt components [9] [14]
- Singapore Tourism Board's Tourism Info Hub provides event data via API (open datasets) [16] [17]
- Stripe is a common choice to integrate payments (e.g., via Stripe Checkout) [19] and you can implement secure payment flows in the frontend [21].
- SendGrid or similar services can be used for automated email notifications [24], and Vercel supports cron jobs for scheduling tasks like daily emails [23].
- Supabase offers a hosted Postgres database and a full backend-as-a-service to simplify development [15].

---

[1] How to Do Server-Side Rendering (SSR) in Next.js - DEV Community
https://dev.to/hamzakhan/how-to-do-server-side-rendering-ssr-in-nextjs-444e

[2] [19] [20] Getting started with Next.js, TypeScript, and Stripe Checkout
https://vercel.com/guides/getting-started-with-nextjs-typescript-stripe

[3] Create a Stunning Apple-Style Dashboard with Tailwind and HTML
https://purecode.ai/community/appledashboardcomponent-tailwind-visiondashboard

[4] [5] Next.js Google Maps: Step-by-Step Integration Guide
https://www.mindbowser.com/nextjs-google-maps/

[6] [7] [8] Creating an Interactive Map with the Google Maps API in Next.js - DEV Community
https://dev.to/adrianbailador/creating-an-interactive-map-with-the-google-maps-api-in-nextjs-54a4

[9] [10] [11] [12] [13] [14] Adding Clerk Authentication to a NextJS App - DEV Community
https://dev.to/brianmmdev/adding-clerk-auth-to-a-nextjs-app-2n9c

[15] Setting Up Supabase in Next.js: A Comprehensive Guide | by Yagyaraj | Medium
https://yagyaraj234.medium.com/setting-up-supabase-in-next-js-a-comprehensive-guide-78fc6d0d738c

[16] [17] Tourism Info Hub | Open Data as a Service API
https://tih-dev.stb.gov.sg/open-data-service-api/apis

[18] Eventbrite v3 Search API is deprecated. Turning off Feb 20, 2020 #83
https://github.com/Automattic/eventbrite-api/issues/83

[21] [24] [25] [26] [27] Develop Full Stack Event Management System - DEV Community
https://dev.to/nadim_ch0wdhury/develop-full-stack-event-management-system-45f9

[22] [23] Cron Jobs
https://vercel.com/docs/cron-jobs