From Prototype to Production: A Comprehensive Guide to Building a Google Calendar Booking App with Angular

Introduction: From Static Mockup to Dynamic Application

The provided HTML file, nails.txt, represents an exemplary starting point for a modern web application.[1] It showcases a well-conceived user flow for an appointment booking system, complete with a clean design, responsive layout using Tailwind CSS, and engaging micro-animations powered by Anime.js. This document serves as a comprehensive technical guide to elevate that static prototype into a fully dynamic, interactive, and data-driven single-page application (SPA) using the latest stable version of the Angular framework.

The journey ahead involves a methodical deconstruction of the original prototype and a reconstruction using Angular's powerful, component-based architecture. This process is designed not only to replicate the functionality but to enhance it with robust state management, scalable form handling, and seamless integration with a powerful third-party service: the Google Calendar API.

This report will cover the entire development lifecycle, beginning with the foundational steps of setting up a professional development environment and scaffolding the project architecture. It will then proceed to the hands-on implementation of the user interface, translating each step of the booking process into discrete, reusable Angular components powered by the Angular Material library. The core of this project lies in connecting the front-end application to the Google Calendar API, a complex but rewarding process involving Google Cloud Platform configuration, OAuth 2.0 for secure user authentication, and direct API interaction to check for availability and create appointments. Finally, the guide will address refinement through styling and animations and chart a clear course for mobile deployment, demonstrating how the Angular web application can be transformed into native iOS and

Android apps using Capacitor.

Part I: Project Foundation and Architecture

A successful software project is built on a solid foundation. Before writing application-specific code, it is imperative to establish a correct development environment, choose a modern project structure, and define a clear architectural strategy. This section details the setup of the Angular environment, the creation of the initial application, the integration of the Angular Material component library, and the architectural blueprint that will guide the development process.

1. Setting the Stage: Environment and Dependencies

1.1. Establishing the Angular Development Environment

The Angular framework, like most modern web development platforms, relies on a specific set of tools to be present on the developer's local machine. These tools form the bedrock of the development, build, and deployment processes.

Node.js: The Engine of Modern Web Development
A prerequisite for any Angular development is the installation of Node.js.2 Node.js is a JavaScript runtime environment that allows JavaScript to be executed outside of a web browser. For Angular, its role is twofold: it provides the runtime for the local development server and build tools, and, more importantly, it includes the Node Package Manager (npm). npm is the world's largest software registry and the default package manager for the JavaScript ecosystem, used to install and manage project dependencies, including the Angular framework itself.4
Angular requires an active Long-Term Support (LTS) or maintenance LTS version of Node.js. As of late 2025, this includes versions such as 20.x and 22.x.[5] It is crucial to install a compatible version from the official Node.js website to ensure stability and access to the latest features.

Installing the Angular CLI
The Angular Command Line Interface (CLI) is an indispensable tool for professional Angular

development.6 It is a command-line utility that automates and simplifies a vast range of tasks, from creating new projects and components to building, testing, and deploying applications. By abstracting away complex webpack configurations and build processes, the CLI allows developers to focus on writing application code.2
The Angular CLI is installed globally on the system using npm. Open a terminal or command prompt and execute the following command:

Bash

```
npm install -g @angular/cli
```

The -g flag signifies a global installation, making the ng command available from any directory on the system.[2]

Verifying the Installation
After the installation processes are complete, it is essential to verify that all tools were installed correctly. This can be done by running the following commands in the terminal and checking their output against the expected versions 4:

- node -v: Displays the installed Node.js version.
- npm -v: Displays the installed npm version.
- ng --version: Displays a detailed report of the installed Angular CLI version, along with the versions of Node.js and other key dependencies it uses.

1.2. Scaffolding the Application with ng new

With the environment prepared, the Angular CLI can now be used to generate a new, modern Angular project. The ng new command creates a workspace with a well-structured starter application, configured with modern best practices.[8]

To create the project, navigate to a suitable parent directory in the terminal and run the following command:

Bash

```
ng new nail-booking-app --standalone --style=scss --routing
```

Each flag in this command configures the project in a specific, deliberate way:

- --standalone: This is arguably the most important flag for a new project. Introduced in Angular v14 and made the default in v17, standalone components represent a paradigm shift in Angular development.[10] This approach eliminates the need for NgModules, which were a source of complexity and boilerplate for many developers. By using standalone components, the application becomes more modular, and the dependency graph of each component is explicitly declared within the component itself, simplifying the learning curve and improving maintainability.[8]
- --style=scss: This flag sets the default stylesheet format to SCSS (Sassy CSS). SCSS is a superset of CSS that adds powerful features like variables, nesting, and mixins, which lead to more organized and maintainable styles, especially in larger applications.
- --routing: This generates a separate set of files for handling application navigation. For a multi-step application like a booking system, a robust routing setup is essential, and this flag provides the necessary foundation.[9]

Upon execution, the CLI will create a new directory named nail-booking-app, install all necessary npm packages, and generate a file structure. Key files include angular.json (the workspace configuration file), package.json (listing project dependencies), src/main.ts (the application's entry point), and the src/app directory, which contains the root application component and routing configuration.[8]

1.3. Integrating the Angular Material Component Library

To accelerate UI development and ensure a high-quality, accessible, and consistent user experience, integrating a component library is a standard practice. Angular Material is Google's official implementation of the Material Design specification and is the premier choice for Angular applications.

Instead of a simple npm install, the Angular CLI provides a more powerful ng add command for libraries that support it. This command not only installs the package but also runs a "schematic"—a script that can modify the project's code and configuration to properly integrate the library.[12]

To add Angular Material, run the following command from within the project directory:

Bash

```
ng add @angular/material
```

This will initiate an interactive prompt to configure the library's integration [12]:

- **Theme Selection:** It will offer a choice of pre-built themes (e.g., "Indigo/Pink") or a "custom" option. For this project, selecting a pre-built theme is the fastest way to get started. The chosen theme's CSS file will be automatically added to the angular.json configuration.[13]
- **Global Typography:** It will ask whether to set up global Material Design typography styles. Answering "yes" ensures that the application's text elements will adhere to the Material Design guidelines.
- **Animations:** It will prompt to include and enable animations. Selecting "yes" is crucial, as many Material components, such as the stepper, dialogs, and menus, rely on the @angular/animations module to function correctly. This schematic will automatically configure the provideAnimations() function in the application's bootstrap configuration (app.config.ts), which is the modern, standalone equivalent of importing BrowserAnimationsModule.[13]

This automated setup process is a significant advantage of the Angular ecosystem, ensuring that complex libraries are configured correctly with minimal manual intervention.

The "Standalone by Default" Paradigm Shift

The decision to build this application using standalone components is a direct reflection of the evolution of the Angular framework. An analysis of Angular's version history reveals a clear trajectory away from the NgModule system. Standalone components were introduced as a developer preview in version 14, became stable in version 15, and were established as the new default for projects generated via the CLI in version 17.[10]

This was a deliberate strategic decision by the Angular team. NgModules served multiple purposes—compilation context, dependency injection configuration, and organization—which often made them a point of confusion for newcomers and a source of boilerplate for experienced developers. The standalone paradigm simplifies this model significantly. Each component, directive, and pipe now explicitly declares its own dependencies in its decorator's imports array. This makes components more self-contained, easier to reason about, and promotes better tree-shakability, potentially leading to smaller application bundle sizes.

For a developer learning Angular today, starting with the standalone-first approach is essential. It aligns their skills with the current and future direction of the framework, ensuring that the knowledge they gain is durable and relevant. Building this guide around the legacy NgModule architecture would teach an outdated pattern and would be a disservice to the learning objective. Consequently, all code examples, dependency injection patterns (using provide functions in app.config.ts), and structural decisions in this report will be based on this modern, streamlined approach.

Package Name	Purpose	Installation Command
@angular/material	The core Angular Material UI component library.	ng add @angular/material
@abacritt/angularx-social-login	Simplifies OAuth 2.0 integration with providers like Google.	npm install @abacritt/angularx-social-login
googleapis	Google's official Node.js client library for their APIs.	npm install googleapis
@types/gapi & @types/gapi.auth2	TypeScript type definitions for the Google API client.	npm install --save-dev @types/gapi @types/gapi.auth2
@capacitor/core & @capacitor/cli	Core libraries for turning the web app into a native mobile app.	npm install @capacitor/core @capacitor/cli
@capacitor/android & @capacitor/ios	Native platform runtimes for Capacitor.	npm install @capacitor/android @capacitor/ios

2. Architectural Blueprint: Components and State

With the project scaffolded, the next step is to define its internal architecture. This involves deconstructing the user interface into a logical hierarchy of components and establishing a robust strategy for managing the application's state as the user progresses through the

booking flow.

2.1. Deconstructing the UI: Mapping the Prototype to Angular Components

The core principle of Angular is building applications as a tree of components. A component encapsulates the template (HTML), styling (CSS/SCSS), and logic (TypeScript) for a specific piece of the UI. The monolithic nails.txt file must be broken down into a set of such components, each with a single, well-defined responsibility.[1]

This process of "componentization" leads to a more modular, reusable, and maintainable codebase. The following component structure is proposed for this application:

- AppComponent: The root component of the application, generated by the CLI. It acts as the main application shell.
- BookingComponent: A new top-level component that will contain the primary UI for the booking process, most notably the Angular Material Stepper.
- ServiceSelectionComponent: A child component responsible for displaying the list of available services (Step 1).
- DateTimePickerComponent: A child component responsible for the calendar and time slot selection UI (Step 2).
- UserDetailsComponent: A child component containing the form for user details (Step 3).
- ConfirmationComponent: A child component to display the final booking summary (Step 4).

Each of these components can be generated using the Angular CLI command ng generate component <component-name>, which creates the necessary .ts, .html, and .scss files in a dedicated folder.[11]

2.2. The Application Shell: Implementing the Material Stepper

The custom JavaScript logic in the prototype that shows and hides div elements to create a step-by-step flow will be replaced by a more powerful and idiomatic solution: the Angular Material Stepper.[1] The

<mat-stepper> component provides a wizard-like workflow out of the box, handling the visual presentation of steps, navigation, and state.[17]

The <mat-stepper> will be implemented in the BookingComponent's template. Each step of

the booking process will be represented by a <mat-step> element. The child components defined above (ServiceSelectionComponent, DateTimePickerComponent, etc.) will be placed inside these <mat-step> elements, effectively nesting our component architecture within the stepper's structure.[19]

To match the required user flow, the stepper will be configured as linear. A linear stepper requires the user to complete each step before they can proceed to the next one, which is the desired behavior for a booking process where each step depends on the previous one.[17]

2.3. The Single Source of Truth: Creating a BookingService for State Management

A critical architectural decision is how to manage and share data between components. In the original prototype, a global JavaScript object called bookingState was used to store the user's selections.[1] While functional for a simple script, this pattern is an anti-pattern in a component-based framework like Angular, where components are designed to be isolated and self-contained.

The bookingState object in the prototype serves as a global, in-memory store for the booking flow's data. As the user selects a service or a time, functions directly mutate this global object. In Angular, components are encapsulated and cannot directly access the state of their siblings. This presents a classic problem: how does the DateTimePickerComponent know which service was selected in the ServiceSelectionComponent?

Angular's solution to this is the injectable service combined with the Dependency Injection (DI) framework. A service is a class designed for a specific purpose, such as fetching data, logging, or, in this case, managing state. When a service is "provided" at the root level of the application, Angular's DI system creates a single instance of it (a singleton). Any component in the application can then "inject" this single instance into its constructor, gaining access to its properties and methods.

This makes the Angular service the direct architectural equivalent of the prototype's global bookingState object. It acts as a shared, in-memory "single source of truth" that is accessible to all components in a structured, maintainable, and testable way. This parallel provides a clear learning path from familiar JavaScript patterns to core Angular concepts.

A BookingService will be created using the CLI (ng generate service services/booking). This service will be responsible for holding the complete state of the appointment being booked. To manage this state reactively, it will use Angular Signals, a modern reactivity model introduced in Angular v16.[15] Signals provide a fine-grained and efficient way to track state changes and automatically update the UI wherever that state is displayed. The service will

contain signals for the selected service, the chosen date and time, and the user's personal details. It will also expose public methods (

setService(), setDateTime(), etc.) that components can call to update the state in a controlled manner.

nails.txt Element (id)	Angular Component	Responsibility
booking-container	BookingComponent	Hosts the mat-stepper and orchestrates the overall booking flow.
step-1	ServiceSelectionComponent	Displays available services and emits the user's selection.
step-2	DateTimePickerComponent	Renders a calendar and time slots, interacts with Google API, emits selection.
step-3	UserDetailsComponent	Contains the reactive form for collecting the user's name, email, and phone.
step-4	ConfirmationComponent	Displays a summary of the booked appointment from the BookingService.
navigation-buttons	BookingComponent (Template)	The matStepperNext and matStepperPrevious buttons are part of the stepper's UI.

Part II: Building the User Interface

This section transitions from architectural planning to hands-on implementation. Each

component defined in the blueprint will be built, focusing on creating the user interface with Angular Material, binding it to the application state, and handling user interactions. The goal is to create a fully functional, client-side prototype before integrating the Google Calendar API.

3. Step 1: Crafting the Service Selection View

The first step in the booking process is for the user to select a service. The ServiceSelectionComponent will be responsible for this view.

The UI from the nails.txt prototype, which features a grid of service "buds," will be recreated within the service-selection.component.html file.[1] Instead of being hardcoded or injected by JavaScript, the services will be defined as an array of objects in the component's TypeScript file (

service-selection.component.ts). The new @for control flow block, introduced in Angular v17, will be used to iterate over this array and render a clickable element for each service. This approach is more declarative and performant than the traditional *ngFor directive.

The data flow will be managed through the BookingService. The ServiceSelectionComponent will inject the BookingService via its constructor. When a user clicks on a service, a method within the component will be called. This method will do two things:

1. It will call the corresponding setService() method on the injected BookingService instance, updating the application's central state.
2. It will use an @Output() property with an EventEmitter to signal to its parent component (BookingComponent) that a selection has been made. The BookingComponent will listen for this event and programmatically advance the mat-stepper to the next step.

4. Step 2: Engineering the Date and Time Picker

This step is the most complex from a UI perspective, requiring both a calendar for date selection and a list of available time slots.

The custom-built calendar from the prototype's JavaScript will be replaced by the Angular Material Datepicker (<mat-datepicker>).[1] This component provides a fully-featured, accessible, and themeable calendar UI, saving significant development effort. It will be bound to a property in the

DateTimePickerComponent.

For the time slots, a grid of Angular Material buttons (<button mat-button>) will be used. These buttons will be generated dynamically using another @for block that iterates over an array of available time slots. The state of these buttons (enabled or disabled) will depend on the availability for the selected date. Initially, this availability will be driven by mock data defined within the component. In Part III, this mock data will be replaced by live data fetched from the Google Calendar API.

The data flow for this step is interactive. When the user selects a date from the <mat-datepicker>, a (dateChange) event will fire. The component will handle this event by triggering the logic to fetch (or, for now, generate) the available time slots for that specific day. When the user then clicks on a time slot button, a method will be called to update the selectedDateTime signal in the BookingService.

5. Step 3: Managing User Input with Reactive Forms

Collecting user details is a critical part of the booking process. Angular provides a powerful system for managing user input called Reactive Forms. This approach is a significant upgrade from the simple DOM manipulation used in the prototype, offering better scalability, testability, and more explicit control over the form's data model and validation logic.[1]

Inside the UserDetailsComponent, a FormGroup will be created in the TypeScript file. A FormGroup is a collection of form controls that tracks the value and validation status of the group. This FormGroup will contain three FormControl instances, one for each input field: name, email, and phone.

In the component's template, the [formGroup] directive will be used to bind the HTML <form> element to the FormGroup instance. Each <input> element will then be bound to its corresponding FormControl using the formControlName directive.

A key advantage of Reactive Forms is built-in validation. Validators can be added directly to the FormControl definition. For example, Validators.required will be added to the name and email controls, and Validators.email will be added to the email control to ensure it's a valid email format. The component's template will include logic to check the status of these controls (e.g., form.get('email').invalid && form.get('email').touched) and display user-friendly error messages when validation fails. This provides immediate feedback to the user.

Once the form is filled out and valid, the user will click the "Confirm Booking" button. The component will handle this click by reading the values from the FormGroup and calling a

method on the BookingService to update the userDetails state.

6. Step 4: Displaying the Booking Confirmation

The final step is a simple confirmation screen. The ConfirmationComponent will be a "display-only" or "presentational" component. Its sole responsibility is to display the final state of the booking.

It will inject the BookingService in its constructor. In its template, it will read the final booking details—the selected service, date, time, and the user's name—directly from the signals within the service. Angular's data binding syntax (interpolation, e.g., {{ bookingService.selectedService().name }}) will be used to display these values. This demonstrates the power of the centralized state management service: the ConfirmationComponent does not need to receive any data via inputs; it can simply connect to the single source of truth and display the current state.

Part III: Integrating with Google Calendar

This section details the integration of the Angular application with the Google Calendar API. This is the most intricate part of the project, transforming the application from a client-side-only tool into a system that interacts with a powerful external service. The process involves configuring a project in the Google Cloud Platform, implementing secure user authentication via OAuth 2.0, and creating a dedicated Angular service to handle all API communications.

7. GCP Configuration: Enabling the Calendar API and OAuth 2.0

Before the application can make any requests to Google's APIs, it must be registered within the Google Cloud Platform (GCP). This registration process identifies the application to Google, enables the necessary APIs, and configures the authorization mechanism.

The step-by-step process is as follows:

1. **Create a GCP Project:** All Google Cloud resources are organized into projects. The first

step is to navigate to the Google Cloud Console and create a new project. This project will serve as the container for all APIs, credentials, and settings related to the booking application.[21]

2. **Enable the Google Calendar API:** Within the new project, navigate to the "APIs & Services" > "Library" section. Here, one can search for and enable the "Google Calendar API." This action makes the API's endpoints available to the project and links it to billing and usage monitoring.[21]
3. **Configure the OAuth Consent Screen:** Because the application will be accessing user data (their calendars), it must request their permission. The OAuth consent screen is the dialog box that Google shows to users when the application first requests access. This screen must be configured with the application's name, a user support email, and, most importantly, the "scopes" of access it requires.[23] During development, it is critical to set the "Publishing status" to "Testing" and add specific Google accounts to the "Test users" list. This allows development to proceed without needing to go through Google's lengthy app verification process.[26]
4. **Create OAuth 2.0 Client ID:** The final configuration step is to create the credentials the application will use to identify itself. In the "APIs & Services" > "Credentials" section, create a new "OAuth 2.0 Client ID." The application type must be set to "Web application." This is where the connection to the Angular front-end is established. In the "Authorized JavaScript origins" field, http://localhost:4200 must be added to allow requests from the local development server. The "Authorized redirect URIs" field should also be configured, although for the client-side flow used here, the origin is the more critical setting.[24] Upon creation, Google will provide a
Client ID, which is the public identifier for the application and will be used directly in the Angular code.

The underlying principle of this entire process is OAuth 2.0, a security standard for delegated authority. The application should never ask for or handle a user's Google password. Instead, the user authenticates directly with Google. If they consent, Google provides the application with a short-lived, limited-permission access token. This token acts as a key that allows the application to make API requests on the user's behalf.

The "scopes" are the heart of this permission model. They define exactly what the access token is allowed to do, adhering to the security principle of least privilege. By requesting only the necessary scopes, the application minimizes its access to user data, which builds user trust and is a requirement for passing Google's verification process. The consent screen transparently communicates these requested permissions to the user, empowering them to make an informed decision. Understanding this context elevates the configuration from a series of clicks to a deliberate security design process.

Scope URL	Permission Granted	Why We Need It

https://www.googleapis.com/auth/calendar.events	Full, read/write access to events on the user's calendars.	Required to create (insert) a new appointment event into the user's primary calendar.
https://www.googleapis.com/auth/calendar.readonly	Read-only access to the user's calendars.	Required to query for freebusy information to determine available time slots.
openid, email, profile	Basic OpenID Connect scopes.	Required by the social login library to get the user's name and email for authentication.

8. Secure Authentication: Implementing Google Sign-In

Handling the OAuth 2.0 flow manually in a client-side application can be complex and error-prone. It is standard practice to use a well-vetted library to manage this process.

For this project, the @abacritt/angularx-social-login library will be used. This is the community-maintained, modern version of the popular angularx-social-login library, updated to support recent versions of Angular and Google's newer sign-in APIs.[28]

The library is configured within the application's main configuration file, app.config.ts. Here, a provider for SocialAuthServiceConfig is set up. This configuration object specifies the OAuth providers the application will use. For Google, a GoogleLoginProvider is instantiated, and the Client ID obtained from the GCP console in the previous step is passed to its constructor.[28]

To encapsulate the authentication logic, a new AuthService will be created (ng generate service services/auth). This service will be the single point of interaction for authentication-related tasks in the application. It will inject the SocialAuthService from the library and expose simple methods like signIn() and signOut(). The service will also subscribe to the library's authentication state observable, which emits a SocialUser object upon successful login. The AuthService will store this user object (containing the user's name, email, and, crucially, the access token) in a signal, making the current user's identity and authentication status available reactively to any component in the application.

9. The Bridge to Google: The CalendarApiService

With authentication handled, a separate service is needed to manage the actual API calls to the Google Calendar endpoints. This separation of concerns is a key architectural principle: the AuthService knows *who* the user is, and the CalendarApiService knows *what* to do with their authorized access.

A CalendarApiService will be created (ng generate service services/calendar-api). This service will be responsible for initializing the Google API client library (gapi) and constructing the authenticated API requests. The gapi library is a JavaScript library provided by Google that simplifies making API calls. It will be loaded by adding a <script> tag to the main index.html file.[24]

The CalendarApiService will inject the AuthService to get access to the authenticated user's access token. This token will be used to initialize the gapi client, ensuring that all subsequent API requests made through it are authenticated on behalf of the logged-in user.[32]

9.1. Querying Availability with the freebusy Endpoint

To determine which time slots are available, the service will use the freebusy.query endpoint of the Google Calendar API. This endpoint is designed specifically for this purpose. It takes a time range (timeMin and timeMax) and a list of calendar IDs and returns a list of time blocks during which those calendars are busy.[33]

A getFreeBusy(date) method will be implemented in the CalendarApiService. This method will:

1. Construct a request body. The timeMin will be the start of the selected day, and timeMax will be the end of the selected day. The items array will contain an object with the id of the calendar to check, which will be 'primary' to represent the user's main calendar.[33]
2. Execute the API call using gapi.client.calendar.freebusy.query(requestBody).
3. Process the response. The API returns an object containing a calendars property, which in turn contains a list of busy time intervals.[33] The service will contain logic to transform this list of busy blocks into a list of available time slots (e.g., by taking a list of all possible slots and filtering out those that overlap with the busy intervals). This transformed list of available slots is what will be returned to the DateTimePickerComponent.

9.2. Creating Appointments with the events.insert Endpoint

Once the user has filled out all their details and clicks the final confirmation button, the application needs to create the event in their calendar. This is done using the events.insert endpoint.

An createAppointment() method will be implemented in the CalendarApiService. This method will:

1. Inject the BookingService to get the final, consolidated booking details (service name, date, time, user details).
2. Construct a Google Calendar Event resource object. This is a JSON object that describes the event to be created. It will include properties like summary (the event title, e.g., "Classic Manicure"), description, start.dateTime, end.dateTime (in ISO 8601 format), and attendees (an array containing the user's email).[31]
3. Execute the API call using gapi.client.calendar.events.insert({ calendarId: 'primary', resource: eventResource }). The 'primary' calendar ID again refers to the user's default calendar.[31]

9.3. Wiring the UI to the API Service

The final step of the integration is to connect the UI components to the new CalendarApiService.

- In the DateTimePickerComponent, the logic that generates mock time slots will be replaced. Now, when a user selects a date, the component will call the calendarApiService.getFreeBusy(selectedDate) method. It will then use the returned list of available slots to dynamically render the time slot buttons.
- In the UserDetailsComponent, the click handler for the "Confirm Booking" button will be updated. It will now first call calendarApiService.createAppointment(). Only after this API call returns successfully will it navigate the stepper to the final confirmation screen. This ensures that the user only sees the confirmation page after the event has been successfully created in their Google Calendar.

Part IV: Refinement and Mobile Deployment

With the core functionality of the web application complete and integrated with the Google Calendar API, this final part focuses on polishing the user experience and fulfilling the request to make the application compatible with iOS and Android. This involves applying custom styling, recreating the smooth transitions from the original prototype using Angular's native animation capabilities, and using the Capacitor framework to package the web application for mobile deployment.

10. Enhancing the User Experience with Styling and Animations

10.1. Applying Custom Themes and Component-Specific Styles

The aesthetic of the original prototype is a key part of its appeal, featuring a soft, dusty rose color palette and elegant typography.[1] This styling will be migrated into the Angular application.

The SCSS styles defined in the <style> block of nails.txt will be broken down and moved into the component-specific .scss files generated by the CLI. For example, styles related to the service selection "buds" will go into service-selection.component.scss, and form input styles will go into user-details.component.scss. This practice, known as scoped styling, is a core feature of Angular's component architecture. It ensures that a component's styles do not leak out and affect other parts of the application, preventing unintended side effects and making the styles easier to manage. For global styles, such as the body background color, the main src/styles.scss file will be used.

10.2. Recreating UI Transitions with the Angular Animations Module

The original prototype uses the anime.js library to create a fluid "fade and slide" effect when transitioning between steps.[1] While it is possible to use third-party animation libraries in Angular, the framework provides its own powerful and highly integrated animation system,

@angular/animations. Using the native Angular animations ensures that animations are performed within Angular's change detection cycle, leading to better performance and a more

cohesive development experience.[14]

The animation logic will be defined within the animations metadata property of the BookingComponent's decorator. The implementation will involve several key functions from the @angular/animations package [37]:

- trigger(): This function defines a named animation that can be attached to an element in the template. For instance, a trigger named 'routeAnimation' will be created.
- transition(): This function specifies the animation that should run when the state of the trigger changes. For example, transition('* <=> *') will apply the animation whenever the route changes.
- query(): This function is used to find elements within the animating element.
- style(): This function defines the CSS styles for an element at a particular point in the animation (e.g., starting styles like opacity: 0).
- animate(): This function defines the timing of the animation, including its duration, delay, and easing function (e.g., '500ms ease-in-out').[38]

By combining these functions, a sequence can be created that replicates the prototype's effect: when a new step's content enters the view, it will start with an opacity of 0 and be slightly translated, then animate to an opacity of 1 and its final position. This provides the polished and professional user experience envisioned in the original design.

11. Beyond the Browser: Preparing for Mobile with Capacitor

A key requirement of the project is the ability to create iPhone and Android apps from the same codebase. Capacitor is a modern tool designed for exactly this purpose. It acts as a native runtime for web applications, allowing a standard web app (like the one just built) to be packaged as a native mobile application that can be submitted to the App Store and Google Play.[39]

11.1. Introduction to Capacitor for Cross-Platform Development

Capacitor works by creating a native mobile project (for Xcode on iOS, and Android Studio on Android) that contains a web view. This web view loads the built Angular application. Capacitor then provides a bridge that allows the web application's JavaScript code to call native device APIs (like the camera, GPS, or file system) through a consistent, cross-platform plugin system.[41] This "write once, run anywhere" approach significantly reduces the time and

effort required to target multiple platforms.

11.2. Adding and Configuring Capacitor for iOS and Android

Integrating Capacitor into an existing Angular project is a straightforward process handled via the command line.

1. **Installation:** First, the Capacitor CLI and core libraries are installed as development dependencies in the project [42]:
   Bash
   ```
   npm install @capacitor/cli @capacitor/core
   ```

2. **Initialization:** Next, the init command is run. This command creates the main Capacitor configuration file, capacitor.config.ts, and prompts for the app name and a unique bundle ID (e.g., com.nailsalon.bookingapp).[39]
   Bash
   ```
   npx cap init
   ```

   After initialization, the capacitor.config.ts file must be edited. The webDir property needs to be set to the output directory of the Angular build process, which is typically dist/<project-name> (e.g., 'dist/nail-booking-app').[40]
3. **Adding Platforms:** With the configuration in place, the native platforms can be added. The following commands create top-level ios and android directories in the project, which contain the full, native Xcode and Android Studio projects, respectively.[42]
   Bash
   ```
   npm install @capacitor/ios @capacitor/android
   npx cap add ios
   npx cap add android
   ```

11.3. The Native Build and Run Workflow

The development workflow for building and running the mobile app involves a simple, repeatable three-step process [39]:

1. **Build the Web App:** The first step is always to create a production-ready build of the Angular application. This is done with the standard Angular CLI command, which compiles the TypeScript, optimizes the assets, and places the output in the

dist/nail-booking-app directory.
```
Bash
ng build
```

2. **Sync Web Assets:** The sync command is the core of the Capacitor workflow. It copies the built web assets from the dist folder into the native ios and android projects. It also updates any native dependencies for Capacitor plugins that have been installed.[42]
```
Bash
npx cap sync
```

3. **Open the Native IDE:** Finally, the open command launches the appropriate native Integrated Development Environment (IDE)—Xcode for iOS or Android Studio for Android.
```
Bash
npx cap open ios
# or
npx cap open android
```

From within the native IDE, the application can be run on a simulator/emulator or deployed to a physical device for testing. The final steps, such as configuring app icons, splash screens, and preparing for app store submission, are all handled within these standard native development tools.

Conclusion: Your Full-Stack Booking System and Future Directions

This comprehensive guide has detailed the complete process of transforming a static HTML/CSS prototype into a dynamic, full-stack web application with a clear path to mobile deployment. By following these steps, a developer can successfully build a functional appointment booking system powered by Angular and the Google Calendar API.

The project journey has covered a wide array of modern web development skills. It began with establishing a professional development environment using Node.js and the Angular CLI. A robust and scalable architecture was designed, centered on standalone components and a centralized state management service. The user interface was implemented using the Angular Material library, incorporating powerful features like the linear stepper and reactive forms for a superior user experience.

The most significant achievement is the successful integration with a major third-party

service. This involved navigating the Google Cloud Platform to configure APIs and credentials, implementing a secure OAuth 2.0 authentication flow to protect user data, and making authenticated API calls to query calendar availability and create new events. Finally, the project was prepared for the future by integrating Capacitor, enabling the single Angular codebase to be deployed as native iOS and Android applications.

With this foundation in place, several exciting future directions are possible:

- **Deployment:** The web application can be deployed to a cloud hosting service like Firebase Hosting, Netlify, or Vercel to make it publicly accessible.
- **Backend Development:** A custom backend (e.g., using Node.js/Express or Firebase Functions) could be developed to manage multiple service providers, each with their own calendar, and to store business-specific information.
- **Enhanced Features:** More advanced features could be added, such as sending email or SMS reminders for appointments, allowing for cancellations, or integrating a payment gateway.
- **Native Functionality:** The mobile app's capabilities could be expanded by using more Capacitor plugins to access native device features, such as push notifications for appointment reminders.

The completion of this project marks a significant milestone in the journey of learning Angular, providing not just theoretical knowledge but tangible, real-world experience in building a modern, feature-rich application from concept to completion.

Works cited

1. nails.txt
2. Setting up the local environment and workspace - Angular, accessed September 15, 2025, https://angular.dev/tools/cli/setup-local
3. How To Install Angular on Windows, macOS, and Linux - Kinsta, accessed September 15, 2025, https://kinsta.com/blog/install-angular/
4. Step by Step Guide to Install Angular on Windows - Radixweb, accessed September 15, 2025, https://radixweb.com/blog/how-to-install-angular-on-windows
5. Version compatibility • Angular, accessed September 15, 2025, https://angular.dev/reference/versions
6. CLI Overview and Command Reference - Angular, accessed September 15, 2025, https://angular.io/cli
7. @angular/cli - npm, accessed September 15, 2025, https://www.npmjs.com/package/@angular/cli
8. Create a new project - Angular, accessed September 15, 2025, https://angular.io/tutorial/tour-of-heroes/toh-pt0
9. ng new - Angular, accessed September 15, 2025, https://angular.dev/cli/new
10. Angular (web framework) - Wikipedia, accessed September 15, 2025, https://en.wikipedia.org/wiki/Angular_(web_framework)

11. How to Create an Angular Project from Scratch ? - GeeksforGeeks, accessed September 15, 2025, https://www.geeksforgeeks.org/angular-js/how-to-create-an-angular-project-from-scratch/
12. Getting started - Angular Material, accessed September 15, 2025, https://material.angular.dev/guide/getting-started
13. Getting started | Angular Material, accessed September 15, 2025, https://v5.material.angular.dev/guide/getting-started
14. Introduction to Angular animations, accessed September 15, 2025, https://v17.angular.io/guide/animations
15. Angular Versions - GeeksforGeeks, accessed September 15, 2025, https://www.geeksforgeeks.org/angular-js/angular-versions-and-releases/
16. How to Create New Angular Project: Step-by-Step Guide | TinyMCE, accessed September 15, 2025, https://www.tiny.cloud/blog/create-new-angular-project/
17. Stepper | Angular Material, accessed September 15, 2025, https://material.angular.dev/components/stepper/overview
18. Angular Material Stepper - GeeksforGeeks, accessed September 15, 2025, https://www.geeksforgeeks.org/angular-js/angular-material-stepper/
19. Stepper | Angular Material, accessed September 15, 2025, https://v5.material.angular.dev/components/stepper
20. Stepper | Angular Material, accessed September 15, 2025, https://v6.material.angular.dev/components/stepper
21. Enable and disable APIs - API Console Help - Google Help, accessed September 15, 2025, https://support.google.com/googleapi/answer/6158841?hl=en
22. How to Integrate Google Calendar in Node.js ? - GeeksforGeeks, accessed September 15, 2025, https://www.geeksforgeeks.org/node-js/how-to-integrate-google-calendar-in-node-js/
23. Go quickstart | Google Calendar, accessed September 15, 2025, https://developers.google.com/workspace/calendar/api/quickstart/go
24. Using the Google Calendar API to Create Calendar Events (with Javascript examples), accessed September 15, 2025, https://endgrate.com/blog/using-the-google-calendar-api-to-create-calendar-events-(with-javascript-examples)
25. JavaScript quickstart | Google Calendar, accessed September 15, 2025, https://developers.google.com/workspace/calendar/api/quickstart/js
26. How to Integrate Google Calendar API Into Your App - OneCal, accessed September 15, 2025, https://www.onecal.io/blog/how-to-integrate-google-calendar-api-into-your-app
27. OAuth 2.0 for Client-side Web Applications - Google for Developers, accessed September 15, 2025, https://developers.google.com/identity/protocols/oauth2/javascript-implicit-flow
28. Integrating Google Single Sign-On with Angular - C# Corner, accessed September 15, 2025, https://www.c-sharpcorner.com/article/integrating-google-single-sign-on-with-a

ngular/
29. abacritt/angularx-social-login - NPM, accessed September 15, 2025, https://www.npmjs.com/package/@abacritt/angularx-social-login
30. Google login in Angular14 with @abacritt/angularx-social-login and the newer Google GIS library - Stack Overflow, accessed September 15, 2025, https://stackoverflow.com/questions/75498800/google-login-in-angular14-with-abacritt-angularx-social-login-and-the-newer-goo
31. CS249 | Google Calendar API, accessed September 15, 2025, https://cs.wellesley.edu/~mashups/pages/am4calendar.html
32. Angular: How to integrate Google Calendar API? - Stack Overflow, accessed September 15, 2025, https://stackoverflow.com/questions/77380238/angular-how-to-integrate-google-calendar-api
33. Freebusy: query | Google Calendar, accessed September 15, 2025, https://developers.google.com/workspace/calendar/api/v3/reference/freebusy/query
34. Why does the 'Free/Busy' time for Google Calendar API come back 'Undefined'?, accessed September 15, 2025, https://stackoverflow.com/questions/25734575/why-does-the-free-busy-time-for-google-calendar-api-come-back-undefined
35. Integrating Google Calendar API in Node.JS: A Guide to Event Creation and Meeting Scheduling | by Gaurav Lohar | Medium, accessed September 15, 2025, https://medium.com/@lohargaurav00/integrating-google-calendar-api-in-node-js-a-guide-to-event-creation-and-meeting-scheduling-8fb2d0a99871
36. Angular Animations: Learn the Basics - Brian Treese, accessed September 15, 2025, https://briantree.se/angular-animations-tutorial-learn-the-basics/
37. Angular Animations: A Beginner's Guide with Real-World Examples | by Sehban Alam, accessed September 15, 2025, https://medium.com/@sehban.alam/angular-animations-a-beginners-guide-with-real-world-examples-fa76d4248223
38. Angular 13 Animations. 1. Introduction to Animations | by Krishnakumar | Medium, accessed September 15, 2025, https://medium.com/@krishsurya1249/angular-13-animations-f0d47ba8c1ba
39. Building Cross-Platform Android Apps from Angular with Capacitor: A Comprehensive Guide | by Shivam Shukla | Medium, accessed September 15, 2025, https://medium.com/@shivam29feb/building-cross-platform-android-apps-from-angular-with-capacitor-a-comprehensive-guide-03db41a651b4
40. Building Mobile Apps with Angular and Capacitor - Capgo, accessed September 15, 2025, https://capgo.app/blog/angular-mobile-app-capacitor/
41. Using Capacitor with Angular, accessed September 15, 2025, https://capacitorjs.com/solution/angular
42. Installing Capacitor | Capacitor Documentation, accessed September 15, 2025, https://capacitorjs.com/docs/getting-started