G-Point: Professional Appointment Management System

1. Project Overview and Value Proposition

The **G-Point** system is a modern, full-stack solution designed to streamline the appointment booking and management process for small to medium-sized service-based businesses (e.g., clinics, salons, consulting firms).

Value Proposition:

- **Efficiency:** Automates scheduling, reducing administrative overhead and minimizing no-shows through automated reminders.
- **Customer Experience:** Provides customers with a seamless, 24/7 online booking experience accessible from any device.
- **Data Insight:** Centralizes service, slot, and client data, enabling better resource planning and business intelligence.
- **Modern Technology:** Built on a robust, scalable, and maintainable technology stack (.NET 9, React 18, TypeScript).

2. Key Features

Feature Category	Description
Appointment Booking	Real-time scheduling that respects service durations, staff availability, and existing bookings. Supports customer self-service booking.
User Management	CRUD operations for internal staff (Admins, Service Providers) and external clients. Role-based access control (RBAC) ensures data security.
Service Catalog	Defines and manages the catalog of available services, including names, descriptions, duration, and pricing.
Slot Management	Allows administrators and service providers to define recurring weekly availability, blackout dates (holidays, personal leave), and capacity limits.
Authentication & Authorization	Secure token-based (JWT) authentication for API access. Supports user registration, login, and role-based permissions throughout the application.
API Testing	Provides a dedicated test suite (e.g., Swagger/OpenAPI) for developers to interact with and validate all backend endpoints easily.
Extensibility	Layered backend architecture allows for easy swapping of components (e.g., changing the database provider) and integration of new services (e.g., external payment processing).

3. Technologies Used

Backend (ASP.NET Core / C#)

- **Framework:** ASP.NET Core (.NET 9)
- **Data Access:** Entity Framework Core (EF Core)
- **Language:** C#
- **Authentication:** JWT Bearer Tokens
- **Database:** PostgreSQL (or any relational database supported by EF Core)

Frontend (React / TypeScript)

- **Framework:** React 18 (with Hooks and functional components)
- **Language:** TypeScript
- **Build Tool:** Vite
- **Styling:** Tailwind CSS (or similar utility-first framework)
- **HTTP Client:** Axios

Tooling

- **Package Manager:** npm / yarn / pnpm (Frontend), dotnet CLI (Backend)
- **API Documentation:** Swagger/OpenAPI

4. Architecture and Folder Structure

Backend Architecture: Layered Approach

The backend utilizes a clean **Layered Architecture** (often referred to as Onion or Clean Architecture) to separate concerns and enforce dependencies, ensuring maximum testability and maintainability. Dependencies flow inwards (API depends on Business, Business depends on Data, Data depends on Domain).

Layer (Project)	Purpose	Dependencies
GPoint.API	Presentation Layer. Houses Controllers, DTOs, and configuration (CORS, Auth). Entry point of the application.	Depends on Business
GPoint.BusinessLogic	Core Business Logic. Contains Services, Validators, and Handlers. Implements the use cases (e.g., `BookAppointmentService`).	Depends on Data and Domain
GPoint.DataAccess	Persistence Layer. Contains the EF Core `DbContext`, Migrations, and Repository implementations. Handles all interaction with the database.	Depends on Domain
GPoint.Domain	Core Entities. Contains primary entity models (e.g., `Appointment`, `User`, `Service`), Value Objects, and Enums. No external dependencies.	None

Frontend Folder Structure

The React application is organized into logical modules for better navigation and scalability.

```
src/
├── api/             # Shared API clients (e.g., axios setup, GPointClient)
├── assets/          # Static files, images, icons
├── components/      # Reusable, application-wide UI components (e.g., Button, Modal)
├── features/        # Feature-specific modules (each with its own logic, components, t
│   ├── appointments/  # All components/logic related to booking and viewing appointmer
│   ├── auth/          # Login, Register forms and logic
│   └── services/      # Service catalog management components
├── pages/           # Components that represent entire application routes/views (e.g.,
├── types/           # Global TypeScript interfaces and types
└── App.tsx          # Root component, routing setup
```

5. Setup and Installation Instructions

Prerequisites

You must have the following software installed:

1. **Backend:** .NET 9 SDK (or later)
2. **Database:** PostgreSQL (or other chosen database service)
3. **Frontend:** Node.js (LTS recommended) and npm/yarn/pnpm

5.1. Backend Setup

1. **Clone the Repository:**

   ```
   git clone [repository-url]
   cd [repository-name]/src/backend
   ```

2. **Configure Database:**
   - Locate `appsettings.Development.json` in the `GPoint.API` project.
   - Update the `DefaultConnection` string to point to your local PostgreSQL instance or desired database.
3. **Run EF Core Migrations:**
   - Open your terminal in the `src/backend/` directory.
   - Create the database schema:

     ```
     dotnet ef database update --project GPoint.DataAccess --startup-project GPoint.
     ```

4. **Run the Backend API:**

   ```
   dotnet run --project GPoint.API
   ```

The API will typically start on `https://localhost:5001`. The Swagger documentation will be available at `https://localhost:5001/swagger`.

5.2. Frontend Setup

1. **Navigate to Frontend Directory:**

```
cd [repository-name]/src/frontend
```

2. **Install Dependencies:**

```
npm install
# or yarn install / pnpm install
```

3. **Configure API Endpoint:**
   - Create a `.env.local` file in the `src/frontend/` directory.
   - Add the environment variable pointing to your running backend API:

```
VITE_API_BASE_URL=https://localhost:5001
```

4. **Run the Frontend Application:**

```
npm run dev
# or yarn dev / pnpm dev
```

The application will typically start on `http://localhost:5173`.

6. Usage Instructions

6.1. Authentication

- **Registration:** Navigate to `/register` and create an account. The first registered account is typically provisioned as an administrator by default (can be adjusted in the database or initial seed data).
- **Login:** Use the credentials to log in via the `/login` page. A JWT is securely stored for subsequent API requests.

6.2. Entity Management (Admin/Provider Users)

Authenticated users with appropriate roles can manage business entities via the main application dashboard:

Entity	Management Route	Actions

Services	`/admin/services`	Create, Read, Update, Delete (CRUD) service catalog items.
Users	`/admin/users`	Manage staff accounts, update roles (Admin, Provider, Client).
Availability	`/provider/schedule`	Define working hours, block out holidays, and set recurring weekly slots.

6.3. Appointment Booking Flow (Client/Customer)

1. **Select Service:** Client navigates to the booking page (`/book`) and chooses a service from the catalog.
2. **Select Provider:** Client selects a preferred service provider (optional).
3. **Select Date/Time:** The system fetches available slots based on the service duration and the provider's defined availability. The client selects a slot.
4. **Confirmation:** The client reviews the booking details and confirms. An email/system notification is typically sent.

7. Contribution Guidelines

We welcome contributions from the developer community. Please follow these guidelines:

7.1. Development Workflow

1. **Fork** the repository and clone your fork.
2. **Create a Branch:** Create a new branch for your feature or fix: `git checkout -b feature/my-new-feature` or `git checkout -b bugfix/issue-123` .
3. **Commit Changes:** Write clear, concise commit messages.
4. **Push:** Push your changes to your fork.
5. **Pull Request (PR):** Open a Pull Request against the `main` branch of the original repository.
6. **Review:** Ensure all automated tests pass and address any code review comments promptly.

7.2. Coding Standards

- **C#:** Follow standard C# naming conventions (PascalCase for classes/methods, camelCase for local variables). Adhere to the layered architecture boundaries.
- **TypeScript/React:** Use functional components and hooks. Prefer explicit typing in TypeScript. Adhere to the modular feature structure.
- **Tests:** New features or complex bug fixes should include corresponding unit or integration tests in the relevant project (`.Tests`).

8. License

This project is licensed under the **MIT License**.

```
MIT License

Copyright (c) 2024 G-Point Developers
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

9. Contact and Support

For questions, support, or general inquiries regarding the G-Point system, please reach out via the following channels:

- **Email Support:** support@gpointapp.com
- **Issue Tracker:** [Link to GitHub Issues Page]
- **Developer Contact:** devteam@gpointapp.com (For technical architecture and contribution