Technical Plan: Hotel Booking System V1

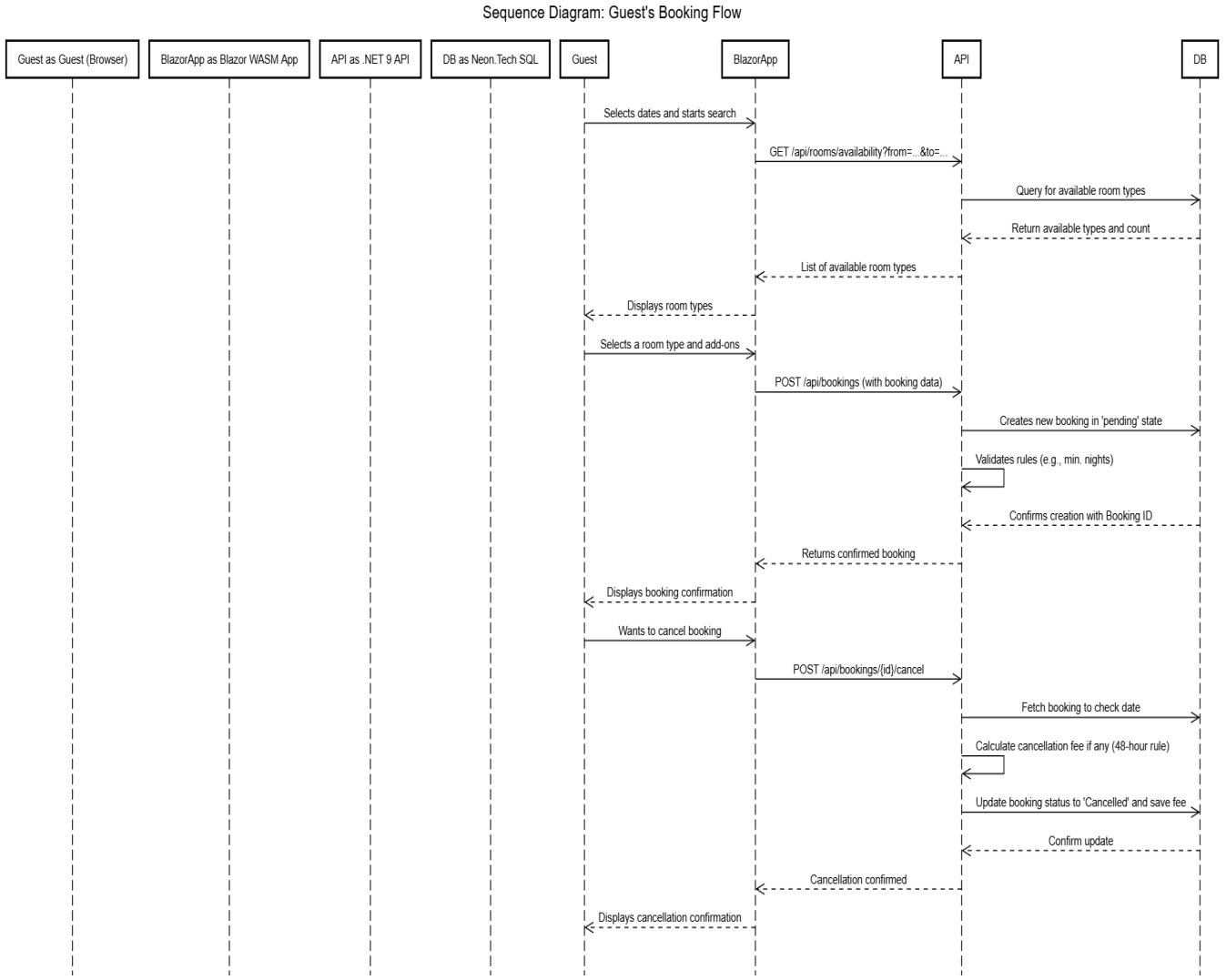
User Journey Description

A step-by-step walkthrough of the user journey for both guests and staff.

Guest's User Journey

1. **Creation & Login:** A new guest visits the website and creates a user profile with an email and password. Existing guests log in.
2. **Search:** The guest enters arrival and departure dates and the number of guests.
3. **Select Room Type:** The system displays a list of available room types with base prices, images, and descriptions. The guest selects a type.
4. **Add Services:** The guest is presented with a list of add-on services (e.g., breakfast, spa access, champagne). Prices are shown as a one-time fee or per night. The guest selects the desired services.
5. **Confirmation:** A price summary is displayed: `(Room Type Base Price × Number of nights) + Sum of add-on services`. The guest confirms the booking.
6. **Booking Management:** After booking, the guest can log into their profile to:
    - View upcoming bookings.
    - Add or remove services (the price updates dynamically).
    - Cancel the booking (with any applicable fee calculated by the system).
7. **Check-in:** The guest arrives at the hotel. The receptionist finds the booking and assigns a specific, clean room.
8. **Check-out:** The guest leaves the hotel. The receptionist marks the check-out in the system. The room's status automatically changes to "Dirty".

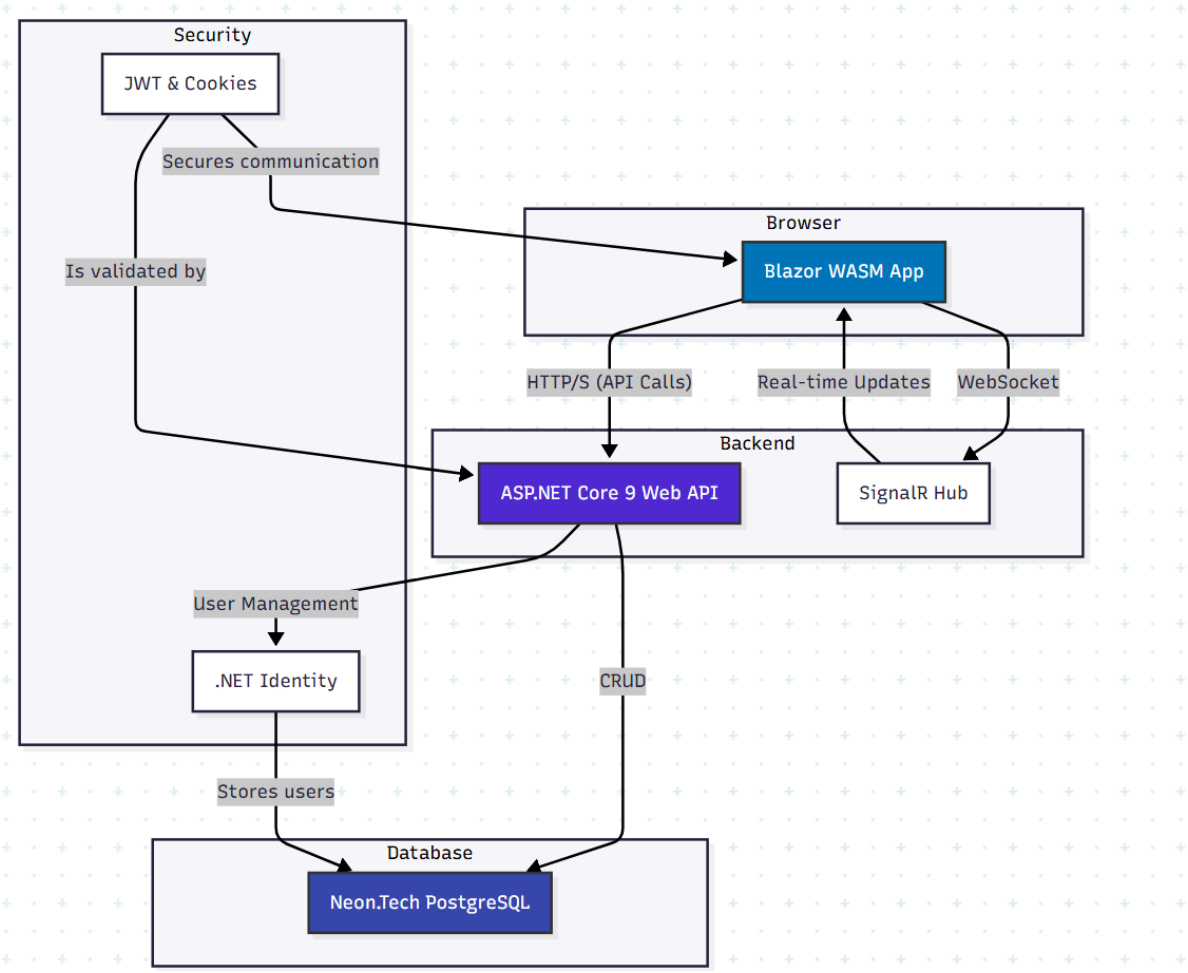
Sequence Diagram: Guest's Booking Flow



Architecture & Technology Overview

The system is designed as a modern, decoupled web application to ensure scalability and maintainability.

System Diagram



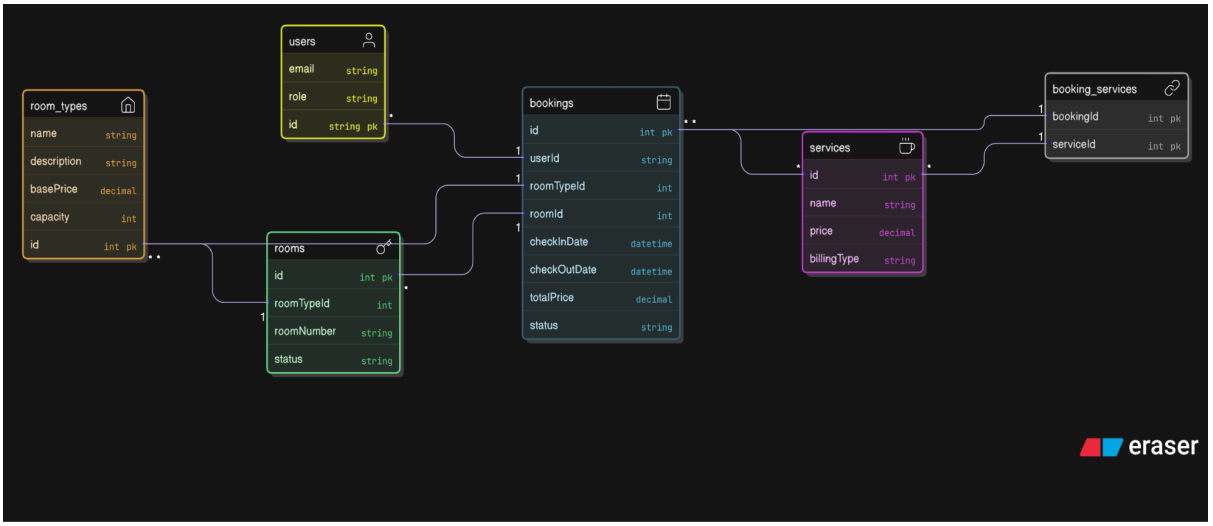
Technology Description

- **.NET 9 & C#:** The backend is built on the latest version of .NET, providing access to the latest performance improvements, language features, and security updates.
- **Blazor WebAssembly (WASM):** The frontend is a Single Page Application (SPA) that runs entirely in the guest's browser. This provides a fast and fluid user interface, as UI logic is executed on the client side. The app calls the backend API for data.
- **ASP.NET Core Web API:** A separate API project handles all business logic, database communication, and user authentication. This separation allows the frontend and backend to be scaled independently.
- **Neon.Tech (PostgreSQL):** A serverless PostgreSQL database that automatically scales resources according to demand. This is ideal for handling varying loads from bookings and queries without manual administration.
- **.NET Identity:** Used for robust and secure user management, including password hashing and role-based access.
- **JWT & Cookies:** Communication between the Blazor app and the API is secured with JWT tokens. The user's session in the Blazor app is managed by a secure cookie to maintain login status.

Data Model & Database Design

The data model is designed to be normalized and efficient for the required queries.

ER Diagram:



Data Model & Database Design

Table	Field Name	Data Type	Description / Relations
Users	`Id`	`string` (GUID)	Primary Key (from .NET Identity).
	`Email`	`string`	Unique email for login.
	`UserName`	`string`	Used for login.
	`PasswordHash`	`string`	Hashed password.
Bookings	`Id`	`int`	Primary Key (auto-increment).
	`GuestId`	`string`	Foreign Key to `Users.Id`.
	`RoomTypeId`	`int`	Foreign Key to `RoomTypes.Id`.
	`RoomId`	`int` (NULL)	Foreign Key to `Rooms.Id`. Assigned at check-in.
	`CheckInDate`	`datetime`	Start date of the booking.

	`CheckOutDate`	`datetime`	End date of the booking.
	`TotalPrice`	`decimal(10, 2)`	The total calculated price.
	`Status`	`string`	E.g., "Confirmed", "CheckedIn", "CheckedOut", "Cancelled".
	`CancellationFee`	`decimal(10, 2)`	Calculated fee for late cancellation.
RoomTypes	`Id`	`int`	Primary Key.
	`Name`	`string`	E.g., "Deluxe Double with sea view".
	`BasePrice`	`decimal(10, 2)`	Fixed base price per night.
	`Capacity`	`int`	Maximum number of guests.
Rooms	`Id`	`int`	Primary Key.
	`RoomTypeId`	`int`	Foreign Key to `RoomTypes.Id`.
	`RoomNumber`	`string`	Unique room number (e.g., "101", "205B").
	`Status`	`string`	"Clean", "Dirty", "Occupied", "Out of Order".
Services	`Id`	`int`	Primary Key.
	`Name`	`string`	E.g., "Breakfast", "Champagne on arrival".
	`Price`	`decimal(10, 2)`	Price of the service.
	`BillingType`	`string`	"PerBooking" or "PerNight".

BookingServices	`BookingId`	`int`	Composite Primary Key, Foreign Key to `Bookings.Id`.
	`ServiceId`	`int`	Composite Primary Key, Foreign Key to `Services.Id`.

Project & File Structure Overview

This overview details the structure of the five projects in our solution. It shows how the existing files and folders will be used and expanded to build the hotel booking system.

1. `API` (Backend Service)

This project is the brain of our system. It handles all business logic, database access, and securely exposes data via a Web API.

Folder/File	Purpose in the Hotel Booking System
Properties/launchSettings.json	Defines how the API is launched locally during development, including URLs and environment variables.
Controllers/	**The core of the API's functionality.** `BookingsController.cs`: Handles creation, cancellation, and management of bookings. `RoomsController.cs`: Handles requests for room types and availability. `AuthController.cs`: Manages user registration and login. `HousekeepingController.cs`: Provides access to lists of rooms needing cleaning.
Data/AppDBContext.cs	**Our bridge to the database.** This Entity Framework Core DbContext class will define all table sets (`DbSet<>`) that map our C# classes (`Booking`, `Room`, `User`, etc.) to tables in the Neon.Tech database.
Migrations/	Contains code-based snapshots of the database schema, generated by Entity Framework. This ensures that the database structure can be versioned and updated across development environments.

appsettings.json	Contains configurations such as the connection string to the Neon.Tech database and JWT token settings.
Program.cs	The application's entry point. Here we configure all services (dependency injection), add the `AppDBContext`, set up .NET Identity, configure JWT authentication, and define the API's request pipeline.

2. `Blazor` (Frontend Application)

This is our Blazor WASM project, which constitutes the entire user interface (UI) for both guests and staff. It runs in the user's browser and communicates with the `API` project.

Folder/File	Purpose in the Hotel Booking System
wwwroot/	Contains static files like CSS, images of rooms, and any JavaScript.
Components/	**The building blocks of our UI.** The folder will be significantly expanded. `StatusCard.razor` can be reused/adapted to show room status. New components will include: • `BookingForm.razor` • `RoomTypeCard.razor` • `ServiceSelector.razor` • `HousekeepingList.razor`
Layout/	Defines the app's overall structure. • `MainLayout.razor`: The main layout. • `NavMenu.razor`: **Will be made dynamic** to show different menu items ("My Bookings", "Check-in", "Housekeeping") based on the logged-in user's role.
Pages/	Routable components that make up the individual pages. `Home.razor` will contain the search functionality. New pages will include: • `MyBookings.razor` (Guest) • `BookingDetails.razor` • `Admin/CheckIn.razor` (Receptionist) • `Admin/RoomManagement.razor`

	(Manager)
Services/APIService/	**Critical for communication.** Here we will create a typed `HttpClient` service. This service will contain all methods for calling our backend API (e.g., `GetAvailableRoomTypesAsync()`, `CreateBookingAsync()`), which keeps our component code clean and testable. `Service.md` will be removed.
App.razor	The application's root component, which handles routing in the Blazor app.
Program.cs	The client app's entry point. Here we register the `APIService` and other client-specific services.

3. `DomainModels` (Shared Logic & Models)

A central C# class library containing the data models shared between the `API` and `Blazor` projects. This ensures consistency and code reuse.

Folder/File	Purpose in the Hotel Booking System
User.cs	Defines the guest and staff model.
Common.cs	Can contain shared enums like `RoomStatus` ("Clean", "Dirty") and `BookingStatus` ("Confirmed", "Cancelled").

(New files)	`Booking.cs`
	`Room.cs`
	`RoomType.cs`
	`Service.cs`

4. `H2-Projekt.AppHost` & `H2-Projekt.ServiceDefaults` (.NET Aspire)

These two projects are part of .NET Aspire, which is designed to simplify the development and deployment of cloud-native applications.

Project/File	Purpose in the Hotel Booking System
H2-Projekt.AppHost/Program.cs	**The orchestration project.** This file defines how our system is connected. It instructs .NET Aspire to start the `API` and `Blazor` projects and tells them how to communicate. It will also contain the configuration for our Neon.Tech database resource.
H2-Projekt.ServiceDefaults/Extensions.cs	Contains **common configuration** for both the `API` and `Blazor` projects. This is the perfect place to set up standards for things like health checks, logging, and telemetry (performance measurement), which is crucial for meeting our requirements for scalability and performance.

API Documentation

Endpoints are secured with role-based access.

Endpoint	Method	Input	Output	Rollekrav
/api/auth/register	POST	Register DTO	Token	(Public)
/api/auth/login	POST	Login DTO	Token	(Public)
/api/rooms/types	GET	–	`List<RoomTypeDto>`	(Public)
/api/rooms/availability	GET	from, to (query)	`List<RoomTypeDto>`	(Public)

/api/bookings	POST	CreateBookingDto	BookingDto	Guest, Receptionist
/api/bookings	GET	–	`List<BookingDto>`	Receptionist, Manager
/api/bookings/my-bookings	GET	–	`List<BookingDto>`	Guest
/api/bookings/{id}	GET	id (route)	BookingDto	Owner, Receptionist, Manager
/api/bookings/{id}/services	PUT	`List<int>` serviceIds	BookingDto	Owner, Receptionist
/api/bookings/{id}/cancel	POST	id (route)	Status 204	Owner, Receptionist
/api/bookings/{id}/checkin	POST	CheckInDto (RoomId)	Status 204	Receptionist
/api/bookings/{id}/checkout	POST	id (route)	Status 204	Receptionist
/api/housekeeping/dirty-rooms	GET	–	`List<RoomDto>`	Housekeeping, Manager
/api/rooms/{id}/status	PUT	UpdateStatusDto	Status 204	Housekeeping, Manager

Role Descriptions

Permissions are strictly separated to ensure system integrity.

Role	Permissions	Limitations	UI Differences
Guest	Create their own account. Search, book, view, modify (services), and cancel their own bookings.	Cannot see others' bookings. Cannot see specific room numbers before check-in. No access to staff functions.	Sees only the guest portal. Navigation menu contains "My Bookings" and "Log out".
Receptionist	All guest permissions. Create bookings for guests. Check guests in/out. View room status. Update booking status.	Cannot change room prices or types. Cannot create staff accounts. Cannot set a room to "Out of Order".	Sees the staff portal. Has access to the check-in/out flow, booking overview, and can create bookings for guests.
Housekeeping	View a list of rooms needing cleaning (status "Dirty"). Change room status from "Dirty" to "Clean".	Cannot see guest data, booking details, or prices. Cannot change other statuses.	Sees only a simple list of rooms to be cleaned, with a button to mark them as "Clean".
Hotel Manager	Full access to all functions. Can manage all bookings. Can manage staff (create, delete). Can set rooms to "Out of Order".	None.	Sees all menu items in the staff portal, including "Administration" for users and rooms.

Performance Considerations

To handle many concurrent users and large amounts of data, the following strategies will be implemented.

- **Caching Strategies:**
  - **Server-side Caching:** Base prices for room types and the list of available services change infrequently. This data will be cached in memory on the API server (`IMemoryCache`) for, e.g., 1 hour to reduce database queries.
  - **Real-time Availability:** A request for available rooms (`/api/rooms/availability`) will always query the database to ensure real-time data, but the calculation itself will be optimized with efficient SQL queries.
- **Lazy Loading in Blazor WASM:**
  - The staff portal's assemblies (`.dll` files), e.g., `Housekeeping.dll`, `Admin.dll`, will be lazy-loaded. Guests visiting the guest portal will not download this unnecessary code, resulting in a faster initial load time for the majority of users.
- **Asynchronous Patterns:**
  - **End-to-end `async`/`await`:** All operations, from UI events in Blazor, through API calls, to database queries with Entity Framework Core, will be asynchronous. This prevents blocking web server threads and ensures the server can handle many concurrent requests efficiently.
- **Database Optimization:**
  - **Indexing:** Indexes will be created on all foreign keys and on frequently queried columns such as `Bookings.CheckInDate`, `Bookings.CheckOutDate`, and `Rooms.Status` to significantly speed up lookups.