Reservation Service [Reference]

Introduction

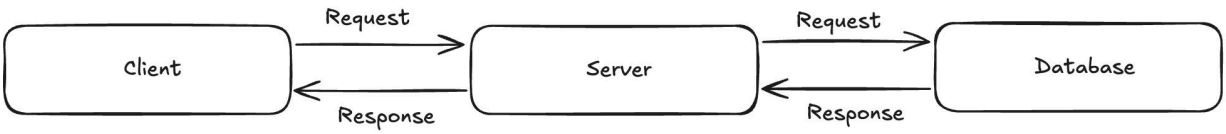
This reservation service system is currently in its initial implementation phase, focusing on administrative service setup per office. The system allows administrators to select an office and manage services (create, edit, delete) under that office.

Core Scope Includes:

- Office selection interface
- Dynamic service category management
- CRUD operations for services within each office
- Appointment booking (Opening Day / Timeslot)
- Pagination operation
- Search or filter functionality
- Error / Loading state handling

Architecture Overview:

- The client-side application handles the user interface and sends requests to the backend API.
- The server-side processes these requests, interacts with the database, and returns responses accordingly.



Frontend Capabilities

Tech Stack:

- **React + TypeScript:** Component-based UI with type safety.
- **Tailwind CSS:** Utility-first CSS framework for rapid UI design.
- **ShadCN UI:** Pre-built, accessible UI components with Tailwind.
- **React Hook Form:** Simplifies form handling and validation.
- **Zod:** Type-safe schema validation for forms.
- **Zustand:** Lightweight state management for React.
- **Axios:** HTTP client for API request
- **Tanstack Query:** Efficient data fetching, caching, and syncing.

Folder Structure:

```
/src
│
├── components/        # UI components (shadcn/ui and project-specific)
├── content/           # Static/mock data for UI rendering
├── hooks/             # Custom React Hooks
├── lib/               # Utility/helper functions
├── schemas/           # Zod schemas for form validation
├── stores/            # Global state management (e.g., Zustand)
├── types/             # Global TypeScript types and interfaces
├── App.tsx            # Root component
└── main.tsx           # Application entry point
```

Component Structure:

```
<App>
└── <QueryClientProvider client={queryClient}>
    └── <SidebarProvider>
        ├── <AppSidebar />
        └── <main>
            ├── <Container>
            │   ├── <SidebarTrigger />
            │   └── <Content />
            ├── <FormDialog />
            └── <ToastContainer />
```

- **QueryClientProvider** — Provides context for Tanstack Query (manages data fetching and caching globally).

- **SidebarProvider** — Provides context to control the Sidebar's open/close state across components.

- **AppSidebar** — Displays the left-hand sidebar (typically for navigation or menu).

- **Container** — A layout wrapper component (controls padding, margin, and layout spacing).

- **SidebarTrigger** — A toggle button for opening the sidebar (visible only on mobile screens).

- **Content** — Displays the main page content.

- **FormDialog** — A modal popup form (used for adding or editing data).

- **ToastContainer** — Displays toast notifications (via React Toastify).

Stores:

- **useOfficeIdStore** — Holds the currently selected officeId to use in service-related requests specific to that office.

```
{
  currentOfficeId: string;
  setOfficeId: (id: string) => void;
}
```

- **useFormDialogStore** — Controls open/close state of a modal (typically for create/edit forms), and stores the id of the item being edited.

```
{
  isOpen: boolean;
  id: string;
  openDialog: (id: string) => void;
  closeDialog: () => void;
}
```

- **useSearchStore** — Stores the current search keyword used for filtering or searching services.

```
{
  search: string;
  setSearch: (value: string) => void;
}
```

- **usePaginationStore** — Stores pagination-related state used for querying data and displaying pagination UI.

```
{
  skip: number;
  take: number;
  totalCount: number;
  setSkip: (value: number) => void;
  setTake: (value: number) => void;
  setTotalCount: (value: number) => void;
}
```

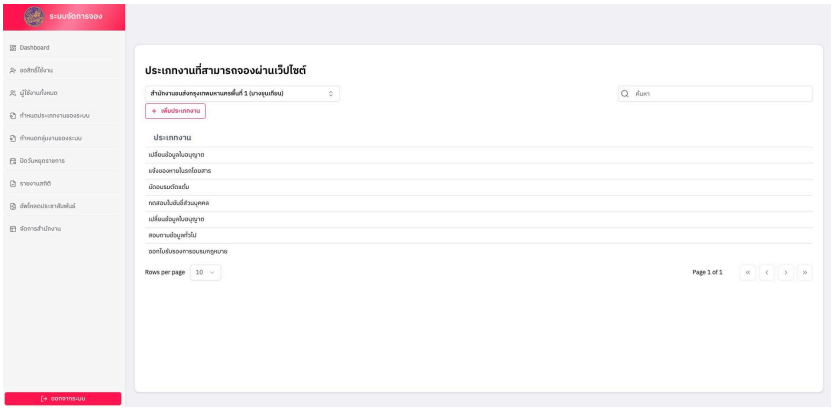
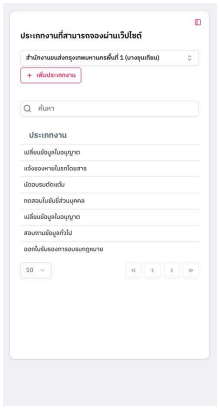
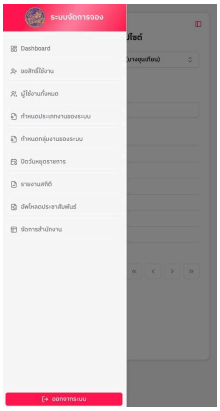
CustomHooks:

- **use-api** — Contains reusable logic for fetching and mutating data from various API endpoints.

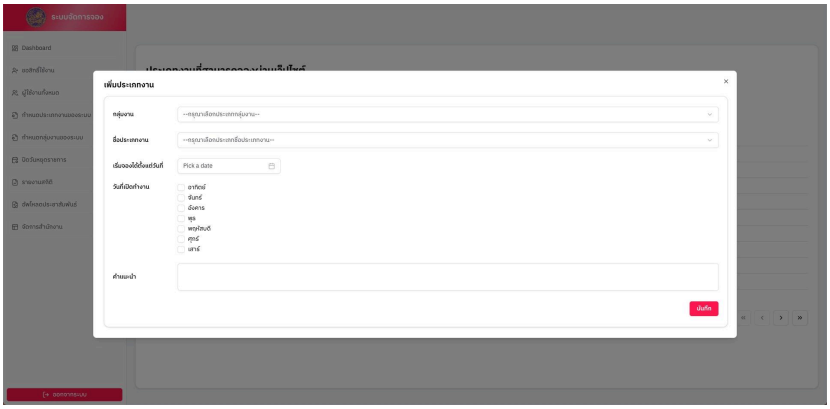
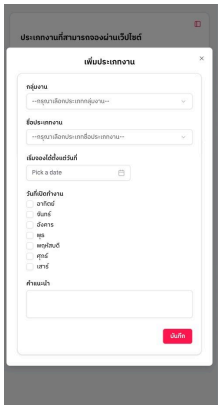
- **use-pagination-controls** — To manage the internal logic for pagination in the Handle page component.

- **use-service-form** — Encapsulates the form handling logic for the Form dialog component.

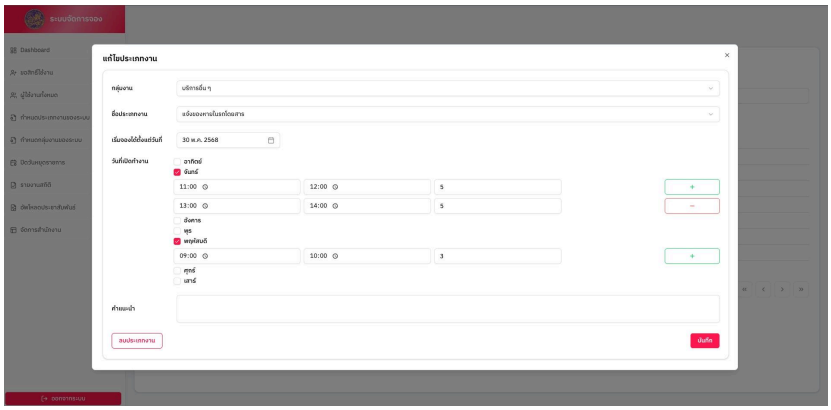
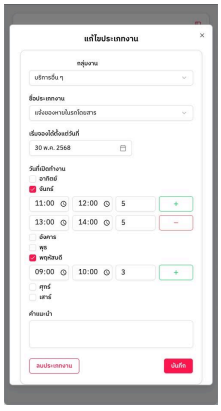
Home

Create

Edit

Backend Capabilities

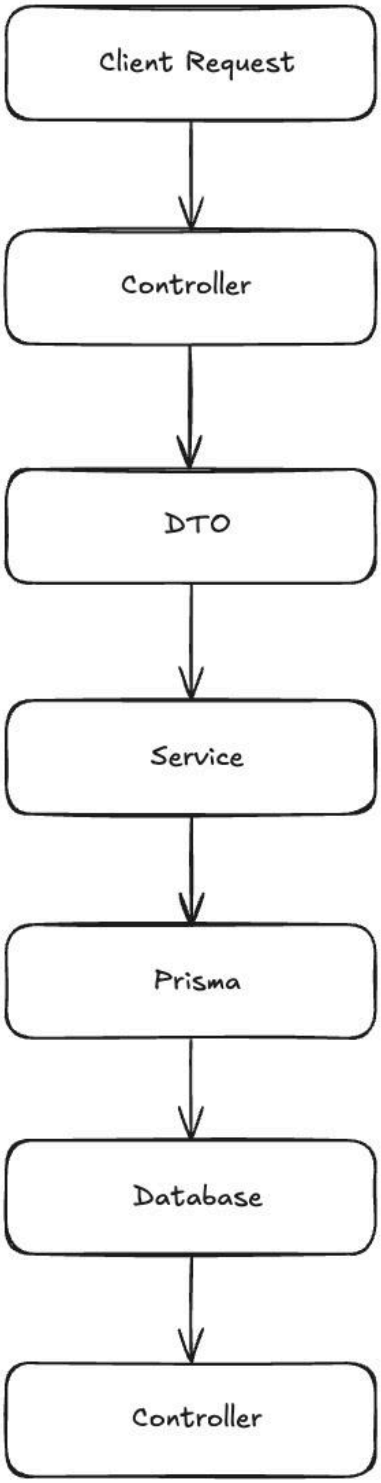
Tech Stack:

- **NestJS + TypeScript:** Modular architecture with strong typing.
- **Prisma ORM:** Type-safe database interaction with PostgreSQL.
- **PostgreSQL:** Relational database used for managing structured data.
- **Class-validator & class-transformer:** Validate and transform incoming data.

Folder Structure:

```
server/
├── prisma/                    # Prisma ORM (DB schema & seeding)
│   ├── schema.prisma          # Prisma schema (data models, relations)
│   └── seed.ts                # Seeding script for mock data
│
├── src/
│   ├── common/
│   │   └── filters/           # Global exception filter
│   │       └── http-exception.filter.ts
│   │
│   ├── offices/               # Office module (DTOs, Controller, Service)
│   ├── services/              # Services module
│   ├── service-categories/    # Service Categories module
│   ├── service-names/         # Service Names module
│   │
│   ├── app.module.ts          # Root application module
│   ├── main.ts                # App entry point
│   └── prisma.service.ts      # Injectable Prisma service
```

Main Work Flow:



Controller - Handles incoming request.

DTO - Validate & shapes input.

Service - Contains business logic

Prisma - ORM for Database access.

Controller - Sends response to client.

Example : Route for create offices

DTO

- Used to define and validate the shape of incoming request data.
- @IsString(), @IsOptional() are decorators from class-validator.
- Ensures incoming POST data has the correct structure.

```
export class CreateOfficeDto {
  @IsString()
  readonly name: string;

  @IsOptional()
  @IsString()
  readonly address: string;
}
```

Controller

- Handles HTTP requests and delegates logic to the service layer.
- Uses @Post() to handle POST requests.
- Extracts the body with @Body() and passes it to the service.

```
@Post()
create(@Body() createOfficeDto: CreateOfficeDto) {
  return this.officesService.createOffice(createOfficeDto);
}
```

Service

- Contains business logic and interacts with the database via Prisma.
- Checks if the office already exists (business rule).
- Uses PrismaService to create the record in the DB.
- Returns a structured response.

```
async createOffice(officeData: CreateOfficeDto) {
  const officeExists = await this.prisma.office.findUnique({ where: { name: officeData.name } });
  if (officeExists) throw new ConflictException("Office must be unique");

  const result = await this.prisma.office.create({ data: officeData });

  return {
    statusCode: 201,
    message: "Office created",
    data: result,
  };
}
```

PrismaService

- Used for database communication.
- Abstracts the interaction with the database.
- Used within services.

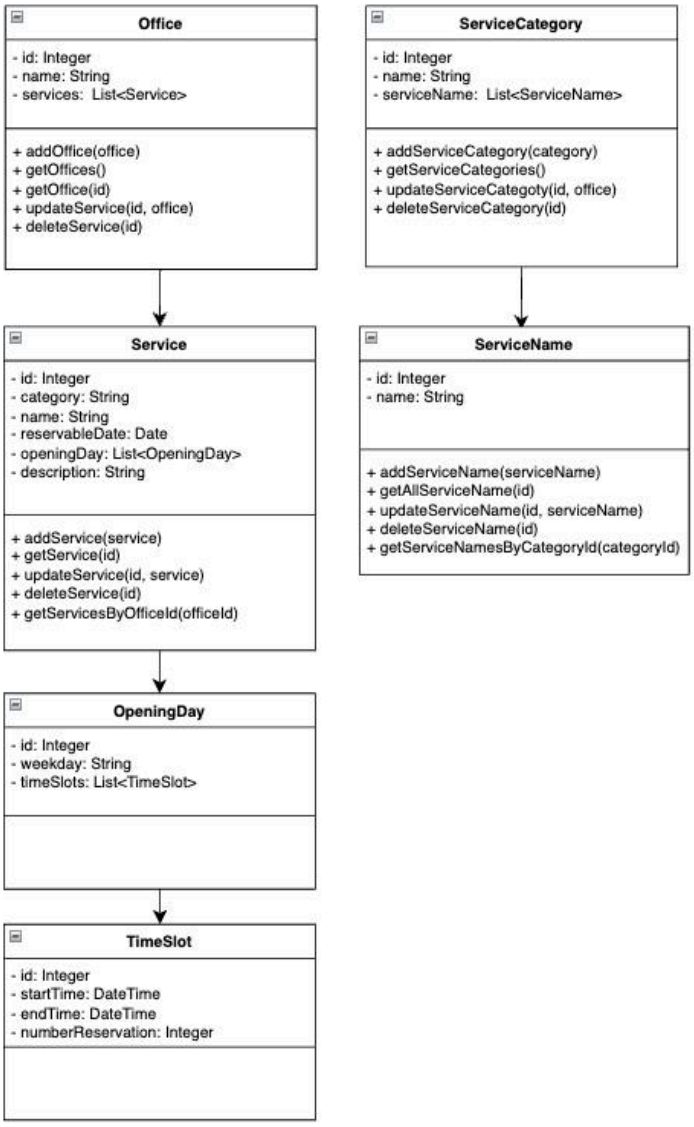
```
this.prisma.office.findUnique(...)
this.prisma.office.create(...)
```

Module

- Link everything together.
- Registers controller and services.
- Ensure dependency injection works properly.

```
@Module({
  controllers: [OfficesController],
  providers: [OfficesService, PrismaService],
})
export class OfficesModule {}
```

Database Design:



**** Note:** Relation of schemas.

- Office (1) ——< (∞) Service

- Service (1) ——< (∞) OpeningDay

- OpeningDay (1) ——< (∞) TimeSlot

- ServiceCategory (1) ——< (∞) ServiceName

API Design:

The API follows RESTful design principles with a focus on clean and scalable architecture.

- **Resource-Based Routes**: The API is organized around resources (e.g., offices, services), where each resource is identified by a URL path.

- **HTTP Methods**: Standard HTTP methods are used to perform CRUD operations.

Offices API Details

Method	Example Endpoint	Description
POST	/offices	Create a new Office
GET	/offices	Retrieve a list of all offices
GET	/offices/:id	Get detailed of a specific office
PATCH	/offices/:id	Update an existing office
DELETE	/offices/:id	Delete an office
GET	/offices/:id/services?skip=0&take=10&search=text	Retrieve paginated list of services (specific office)

Example: body

```
{
   "name": "สำนักงานขนส่งพื้นที่1(บางขุนเทียน)",
    "address": ""
}
```

Example: response

```
{
    "statusCode": 201,
    "message": "Office created",
    "data": {
        "id": "dfea41f0-74d3-4949-a2a8-9d5277939b5c",
        "name": "กรมขนส่งทางบก(อาคาร8)",
        "address": "somewhere"
    }
}
```

Services API Details

Method	Example Endpoint	Description
POST	/services	Create a new service for an office
GET	/services/:id	Retrieve a details of a specific service
PUT	/services/:id	Update an existing service
DELETE	/services/:id	Delete a service

Example: body


```
{
  "category": "งานทะเบียน",
  "name": "ใบแทนแผ่นป้ายทะเบียน",
  "reservableDate": "2025-05-12T00:00:00.000Z",
  "description": "บริการออกใบแทนกรณีแผ่นป้ายทะเบียนสูญหายหรือชำรุด",
  "officeId": "658e1159-7eb8-43c6-a52f-f7d8ecbf094c",
  "openingDays": [
    {
      "day": 3,
      "timeSlots": [
        {
          "startAt": "08:30",
          "endAt": "09:30",
          "available": 4
        }
      ]
    },
    {
      "day": 4,
      "timeSlots": [
        {
          "startAt": "11:00",
          "endAt": "12:00",
          "available": 5
        }
      ]
    }
  ]
}
```

Example: response


```
{
    "statusCode": 201,
    "message": "Service created",
    "data": {
        "id": "67256eba-b35e-436f-938c-be87a873d0d7",
        "category": "งานทะเบียน",
        "name": "ใบแทนแผ่นป้ายทะเบียน",
        "reservableDate": "2025-05-12T00:00:00.000Z",
        "description": "บริการออกใบแทนกรณีแผ่นป้ายทะเบียนสูญหายหรือชำรุด",
        "officeId": "658e1159-7eb8-43c6-a52f-f7d8ecbf094c",
        "openingDays": [
            {
                "id": "4eafe8f9-2749-40c8-8dba-332ad2143cc4",
                "day": 3,
                "serviceId": "67256eba-b35e-436f-938c-be87a873d0d7",
                "timeSlots": [
                    {
                        "id": "d5744fa5-0efa-4b95-aebc-d77c5ad8ef28",
                        "startAt": "08:30",
                        "endAt": "09:30",
                        "available": 4,
                        "openingDayId": "4eafe8f9-2749-40c8-8dba-332ad2143cc4"
                    }
                ]
            },
            {
                "id": "d30725b8-369a-45e9-826d-cd81d8a37611",
                "day": 4,
                "serviceId": "67256eba-b35e-436f-938c-be87a873d0d7",
                "timeSlots": [
                    {
                        "id": "771827d6-ada6-48ef-b43b-6dda4c80504e",
                        "startAt": "11:00",
                        "endAt": "12:00",
                        "available": 5,
                        "openingDayId": "d30725b8-369a-45e9-826d-cd81d8a37611"
                    }
                ]
            }
        ]
```

Service Categories API Details

Method	Example Endpoint	Description
POST	/service-categories	Create a new service category
GET	/services-categories	Retrieve a list of all service categories
PUT	/services-categories/:id	Update an existing service category
DELETE	/services-categories/:id	Delete a service category
GET	/services-categories/:id/service-names	Retrieve list of service names (specific category)

Example: body


```
{
    "name":"REGISTRATION"
}
```


Example: response


```
{
    "statusCode": 201,
    "message": "Service category created",
    "data": {
        "id": "bc6a6010-7491-4a1d-8d2a-2890fc443343",
        "name": "REGISTRATION"
    }
}
```

Service names API Details

Method	Example Endpoint	Description
POST	/service-names	Create a new service name for an service category
GET	/service-names	Retrieve a list of all service names
PUT	/service-names/:id	Update an existing service name
DELETE	/service-names/:id	Delete a service name

Example: body

```
{
    "name":"TAX_PAYMENT",
    "serviceCategoryId":"bc6a6010-7491-4a1d-8d2a-2890fc443343"
}
```

Example: response

```
{
    "statusCode": 201,
    "message": "Service name created",
    "data": {
        "id": "a0693487-97bc-4639-8ce9-85bafe6591a8",
        "name": "TAX_PAYMENT",
        "serviceCategoryId": "bc6a6010-7491-4a1d-8d2a-2890fc443343"
    }
}
```

- **HTTP Status Code:** Proper use of HTTP status codes is employed to represent the result of API requests.
    - 200 OK for successful requests.
    - 201 Created for successful creation.
    - 400 Bad Request for invalid input.
    - 404 Not Found for non-existent resources.
    - 409 Conflict with an existing resource(e.g. duplicate entry)
    - 500 Internal Server Error for unexpected server issues.

- **Error Handling**: Custom error messages are provided in a structured format, making it easier for developers to debug issues.


```
{
    "statusCode": 409,
    "message": "Office must be unique",
    "path": "/offices"
}
```