Yalantis

CHOOSING THE RIGHT DATA STORAGE SOLUTION FOR AN APPOINTMENT SCHEDULING SYSTEM

EXPLAINING DATA HANDLING WITH OBJECT-RELATIONAL MAPPING (ORM) SYSTEMS LIKE TYPEORM. CODE EXAMPLES AND POSSIBLE ENHANCEMENTS

WHAT IS IN THIS GUIDE?

WHAT IS OBJECT-RELATIONAL MAPPING (ORM) AND HOW DOES IT WORK FOR DATA MANAGEMENT?

Programmers model real-world entities using objects and classes. These objects have attributes and methods that define their properties and behaviors. In contrast, databases store data in tables, which consist of rows and columns. Object-relational mapping serves as a bridge between these two worlds.

ORM allows developers to seamlessly interact with a database using the language of objects. Instead of writing complex SQL queries to insert, retrieve, or update data, programmers can work with familiar object-oriented code. This abstraction simplifies development, reduces the need for low-level database management, and ensures data consistency and security.

WHAT IS THE BEST CHOICE OF DATA MANAGEMENT TOOLS FOR APPOINTMENT SCHEDULING SYSTEMS BUILT WITH NODE.JS?

When creating a healthcare appointment system, an SQL database may be the most suitable to start with. SQL databases allow for defining the hierarchy between key entities early on and can easily be transformed into database tables.

As for ORM tools to complement the database, Node.js offers a few options, with the most popular being Prisma, TypeORM, and Sequelize. Let's explore how ORM systems can be useful.

ORM LIKE TYPEORM: SIMPLIFYING DATA HANDLING

Object-relational mapping offers a user-friendly way to handle complex database operations. Instead of dealing directly with intricate SQL queries, developers can work with objects and classes, making the process intuitive and streamlined.

For example, say we have entities called appointment and doctor and we want to get a list of all patients along with their doctor appointments.

First, we need to describe the entities:

JavaScript

```
// appointment.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne } from 'typeorm';
import { Patient } from './patient.entity';
import { Doctor } from './doctor.entity';

@Entity()
export class Appointment {
 @PrimaryGeneratedColumn()
 id: number;

 @Column()
 date: Date;

 @ManyToOne(() => Patient, patient => patient.appointments)
 patient: Patient;

 @ManyToOne(() => Doctor, doctor => doctor.appointments)
 doctor: Doctor;
}
```

JavaScript

```
// doctor.entity.ts
import { Entity, Column, PrimaryGeneratedColumn, OneToMany } from 'typeorm';
import { Appointment } from './appointment.entity';

@Entity()
export class Doctor {
 @PrimaryGeneratedColumn()
```

```
  id: number;

  @Column()
  name: string;

  @OneToMany(() => Appointment, appointment => appointment.doctor)
  appointments: Appointment[];
}
```

Now, to get a list of patients along with their doctor appointments, let's use TypeORM:

JavaScript

```
// patient.service.ts
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Patient } from './patient.entity';

@Injectable()
export class PatientService {
 constructor(
  @InjectRepository(Patient)
  private readonly patientRepository: Repository<Patient>,
 ) {}

 async getAllPatientsWithAppointments(): Promise<Patient[]> {
  return this.patientRepository.find({ relations: ['appointments'] });
 }
}
```

Using ORM, such code provides us with a straightforward way to retrieve patients along with their doctor appointments. A similar SQL query with joins would be more complex and less readable.

Unset

```
SELECT patients.id, patients.firstName, patients.lastName, appointments.id AS
appointmentId, appointments.date
FROM patients
LEFT JOIN appointments ON patients.id = appointments.patientId;
```

By using ORM, clinicians can reduce the number of errors when working with patients' data and can easily manage their schedules to avoid conflicts.

FURTHER ENHANCEMENTS: MONGODB FEATURES FOR DATA MANAGEMENT

Along with the basic stages for choosing a database and ORM, you'll likely need to store a patient's history, doctor visits, etc. For this, you might need to use tools for things like data versioning or event sourcing. The former is a good choice for early stages of development, as it avoids complicating the system. However, this inevitably raises the issue of data retrieval and speed.

In this case, you might want to consider NoSQL databases. For instance, MongoDB or other cloud databases record the most up-to-date patient data with all necessary additional relationships, denormalizing the data. Periodic synchronization can then be performed. This approach is labor-intensive, but it provides fast data retrieval.

Here is a simple example of what doctor data might look like in MongoDB:

```javascript
// "doctors" in MongoDB
{
 "_id": ObjectId("doctor1"),
 "name": "Dr. Smith",
 "appointments": [
  {
   "_id": ObjectId("appointment1"),
   "date": ISODate("2023-01-01T08:00:00Z")
  },
  {
   "_id": ObjectId("appointment2"),
   "date": ISODate("2023-01-02T10:30:00Z")
  }
 ]
}
```

BENEFITS OF BUILDING CUSTOM HEALTHCARE SOLUTIONS WITH YALANTIS

- Expertise in creating HIPAA-compliant systems
- Fully tailored solutions – we build only what you need
- Assistance in achieving technological maturity and change management
- Constraint management strategies to minimize business risks
- Operational flexibility and visibility

YALANTIS IN BRIEF

- 15+ years of experience
- 500+ IT experts aboard
- 35+ active clients

Project inquiries:

- hello@yalantis.com

Phone number:

- + 1 213 4019311

Our offices:

- Poland
- Ukraine
- Cyprus
- Estonia

COLLABORATE WITH US TODAY!
CONTACT US

Yalantis