i want to create a simple app which by inputting a persons name and егн to receive notifications about birthdays and namedays

Got it You want a simple app where you input a person’s **name** and **ЕГН** (Bulgarian personal number), and the app should notify you about their **birthday** and **nameday**. Let’s break it down into the core features:

Key Features

1. **Input & Storage**
   - Form to enter:
     - Name
     - ЕГН (from which you can extract date of birth).
   - Store the data (simple file, SQLite, or cloud DB depending on how big you want it).
2. **Birthday Extraction**
   - From ЕГН → extract birthdate (first 6 digits = `YYMMDD`, with special rules for months depending on century).
   - Set a yearly reminder for that date.
3. **Nameday Calculation**
   - Namedays in Bulgaria are tied to specific names & church calendar dates.
   - You’ll need a **mapping** between common names and their namedays.
   - Store this in a dictionary or database (e.g., `{"Ivan": "07-01", "Georgi": "06-05", ...}`).
4. **Notifications**
   - On the exact date → push notification / email / system alert.
   - Could also remind a day before.

Tech Options

- **Mobile (Android/iOS):**
    - **Flutter** (cross-platform, simple UI).
    - **React Native** (if you prefer JS/TS).
    - Native Android (Kotlin/Java) or iOS (Swift).
- **Desktop/Web:**
    - Simple **web app with Node.js/Express + SQLite**.
    - Or just a **Python app with Tkinter/PyQt** and scheduled checks.

⚙ Implementation Steps

1. **Parse ΕΓΗ → Birthday**
    - Extract date from ΕΓΗ.
    - Handle months:
        - 01–12 = 1900–1999
        - 21–32 = 1800–1899 (subtract 20 from month)
        - 41–52 = 2000–2099 (subtract 40 from month).
2. **Nameday Database**
    - Start small → only add the names you care about.
    - Later, expand with full list (there are nameday APIs/lists online).
3. **Scheduler/Notifications**
    - On mobile → use **local notifications** (e.g., `flutter_local_notifications`).
    - On desktop/web → use cron jobs, background timers, or email reminders.

Quick Example (Python, minimal):

```
pythonCopy codefrom datetime import datetime

# Example EGN: 0445123456 -> 2004-05-15
def get_birthday_from_egn(egn):
    year = int(egn[0:2])
    month = int(egn[2:4])
    day = int(egn[4:6])

    if 1 <= month <= 12:
        year += 1900
    elif 21 <= month <= 32:
        year += 1800
        month -= 20
    elif 41 <= month <= 52:
        year += 2000
        month -= 40

    return datetime(year, month, day).date()

print(get_birthday_from_egn("0445151234"))  # -> 2004-05-15
```

Then you just check daily if today matches someone’s birthday/nameday and send a notification.

Do you want me to sketch this as a **mobile app with notifications (Flutter/React Native)**, or as a **desktop/web app** that just shows reminders?

Do you think I can do it with c# cause I know it may not be the most optimal language for this but it's the one I am versitale enough to maintain by myself or java or c++

Yes □ you can absolutely do this in **C#**, **Java**, or even **C++** — the difference will mainly be in how easy it is to handle things like **UI, date parsing, and notifications**.

Here’s how they compare for your use case (small personal app with birthdays & namedays):

▯ C#

- Great choice if you want **Windows desktop app**.
- You can build it with **WPF** or **WinForms** (simpler, but older).
- Storing data: `SQLite`, JSON, or just a text file.
- Notifications:
    - Windows → toast notifications with `Windows.UI.Notifications`.
    - Cross-platform → use **.NET MAUI** (replacement for Xamarin) to make it run on Android/iOS/Windows with one codebase.

▯ Since you said you're versatile in C#, this might be your **sweet spot**.

▯ Java

- Great for **Android app** development (native).
- Easy to set up notifications on Android.
- Can also make a simple **Swing/JavaFX desktop app**, but feels heavier than C#.
- Birthday calculation logic is the same.

▯ C++

- Doable, but overkill 😅.
- You'd need Qt (for UI & cross-platform support).
- Notifications aren't as straightforward.
- More maintenance overhead compared to C#/Java.

▯ My Suggestion

- If you want a **Windows-only desktop tool** → go with **C# (WinForms/WPF)**.
- If you want **cross-platform desktop & mobile** → use **.NET MAUI (C#)**.
- If you want a **mobile app only** → go with **Java (Android)** or **Kotlin (better modern Android choice)**.

□ Since you already know C#, you could quickly get an MVP working with:

1. **Input form** → TextBoxes for Name + ΕΓΗ.
2. **Birthday parser** → small method to extract birthdate from ΕΓΗ.
3. **Nameday dictionary** → hardcode a few names first.
4. **Storage** → JSON file with all people.
5. **Daily check** → Timer that runs once a day and compares today’s date.
6. **Notification** → messagebox (basic), or Windows toast if you want it fancy.

□ Do you want me to sketch a **basic C# WinForms version** of this (with JSON storage + daily birthday/nameday check), so you can expand later?

If you ignore my desire to be c# which language do you think would be best

If we set C# aside and just look for the **best tool for the job**, I’d weigh it like this:

□ Best Choices

1. **Kotlin (Android)**
    - If your goal is mainly **mobile notifications** (birthdays & namedays pinging you on your phone).
    - Kotlin is the modern standard for Android.
    - Easy access to **system notifications**, background jobs, alarms.
    - Can later publish the app for your own use or even on Google Play.
2. **Flutter (Dart)**
    - If you want **cross-platform** (Android + iOS + desktop).
    - One codebase, rich UI, and good libraries for local notifications.

    - Easy to make it look nice with little effort.

⚖ Middle Ground

- **Java** → still solid for Android, but Kotlin is replacing it.
- **Python** → very fast to prototype, but not great for polished apps with real notifications.
- **Web app (JavaScript/TypeScript + Node.js/React)** → great if you’re fine with browser/email notifications, but not as smooth as native notifications.

▯ Not Ideal

- **C++/Qt** → too heavy for such a simple project, notifications are tricky.
- **Plain Java desktop (Swing/JavaFX)** → feels old, not mobile-friendly.

▯ My ranking ignoring C#:

1. **Kotlin (Android, best for native mobile notifications).**
2. **Flutter (if you want it cross-platform and pretty).**
3. **Java (Android, but less modern).**

Do you imagine this app more as something you’ll **carry on your phone** (so you always get reminders), or more as a **desktop tool** that runs on your computer?

How easy is it to use kotlin and hypothetically would be possible with react and c#. These are 2 separate questions

Great two questions ▯ let’s split them clearly:

1 How easy is it to use **Kotlin**?

- **Learning curve:**
  - If you’ve used **Java or C#**, Kotlin will feel very familiar.
  - Syntax is cleaner and more concise than Java.
  - Example:

    ```
    kotlinCopy codeval name = "Ivan"    // immutable
    var age = 30          // mutable
    ```

    (compared to more verbose Java).
- **Development setup:**
  - Install **Android Studio** → create new project → choose Kotlin.
  - Android Studio has templates for activities, forms, and notifications.
  - Google’s official docs + tons of tutorials make it beginner-friendly.
- **Notifications in Kotlin:**
  - Built-in Android APIs let you easily set alarms, reminders, and push notifications.
  - Example: scheduling a birthday notification is just a few lines with `AlarmManager` or `WorkManager`.

 For someone with C#/Java background, you can get productive in **a few weeks**.

---

2 Would it be possible with **React** and **C#**?

Yes, in two different flavors:

- **React (JS/TS):**
  - If you mean **React Native**, then yes → you can build a mobile app.
  - React Native supports local notifications via libraries (e.g., `react-native-push-notification`).
  - Advantage: cross-platform (Android + iOS).

  - Downside: you’d need to manage name/EGN parsing logic in JavaScript/TypeScript, and mobile notifications require some native setup.
- **C# (with .NET MAUI):**
  - Lets you build **one project that runs on Android, iOS, Windows, macOS**.
  - Supports local notifications with community libraries.
  - Familiar to you since you already know C#.
  - Downsides: ecosystem is younger than React Native/Flutter, so sometimes more setup work.

⚖ Comparison

- **Kotlin** → best if you want a **native Android app**, fewer dependencies, very direct.
- **React Native** → best if you want **cross-platform** but are comfortable with JavaScript/TypeScript.
- **C# (.NET MAUI)** → best if you want cross-platform **but prefer to stay in your comfort zone** (C#).

 If you wanted to start tomorrow with your current skills, C# (.NET MAUI or WinForms) would be fastest.
 If you wanted the *cleanest long-term Android app*, Kotlin is easier than it looks.
 If you wanted to reach both iOS + Android fast, React Native is viable.

Do you want me to sketch **what Kotlin code for a birthday notification might look like** so you can see how simple/complex it feels?

If I need both android app and a website and I want to use kotlin ?

Good question If you need **both an Android app AND a website**, Kotlin can actually cover both — but with different approaches:

For Android App

- Kotlin is the **first-class language** for Android.
- You’d use **Android Studio** and Jetpack libraries.
- Notifications, background jobs, UI → all straightforward.

For Website

Kotlin has two main paths:

1. **Ktor (Kotlin Web Framework)**
    - Like Express.js but in Kotlin.
    - Can build REST APIs, backends, even full websites.
    - Example: you could have `POST /users` to add a person with name + ΕΓΗ, and `GET /reminders` to fetch upcoming birthdays/namedays.
2. **Kotlin + React (via Kotlin/JS)**
    - You can actually write frontend in Kotlin that compiles to JavaScript.
    - Not as common, but possible if you want to stick with Kotlin everywhere.
    - Example: use Kotlin Wrappers for React

3. **Classic Approach: Kotlin Backend + JS Frontend**
    - Write the backend in Kotlin (Ktor, Spring Boot).
    - Write the frontend in React/Angular/Vue (JS/TS).
    - This is the most common real-world setup.

How They Work Together

- **Backend (Ktor or Spring Boot in Kotlin):**
    - Handles storage (SQLite/Postgres/MySQL).
    - Stores users (name, ΕΓΗ, birthday, nameday).
    - Sends data to both the Android app and website.
    - Could also send email reminders or push notifications.
- **Android App (Kotlin):**
    - Talks to backend API.
    - Displays notifications locally.
- **Website:**
    - Could be a React app consuming the same backend API.
    - Or a server-side rendered website with Ktor templates.

⚖ Pros of going Kotlin for both

- One language → less context switching.
- Strong type-safety, modern features (coroutines, null safety).
- JetBrains (the makers of IntelliJ/Android Studio) back Kotlin heavily, so it’s not going anywhere.

⚠ Cons

- Frontend web with Kotlin/JS isn’t as popular → fewer resources.
- If you want a slick website, React/Angular (in TypeScript) still has the edge.

▯ If your **main goal is Android + a simple companion website**, then:

- **Android App** → Kotlin (native).
- **Backend** → Kotlin (Ktor or Spring Boot).
- **Frontend** → either Kotlin/JS React wrappers (if you want *all Kotlin*), or plain React (JS/TS) if you want easier hiring/community support later.

Do you want me to sketch a **possible architecture diagram** for the Kotlin app + website setup, so you see how the pieces fit?

i asked my clients(family) what exactly do they need what we found it is this: it doesnt matter if its a phone app or web app or desktop app or a system its important that they either receive a notification in the day that some of their clients have bday or nameday or the system automatically send email/viber message wishing them something generated. currently they have an excel table storing info about the clients i am thinking of migrating this to a database which would terminate the use of this excel sheet and we expand our project a little by allowing them to also add new clients to the database or find info fast about an existing clients

Perfect ▯ Thanks for clarifying — this makes it much more of a **real-world CRM-lite system** for your family. Let me summarize and then give you a roadmap.

▯ Requirements (from what you said)

1. **Data source migration**
    - Right now → Excel file with clients.
    - Future → Database (so you can query, add, update easily).
2. **Reminders / automation**
    - Daily check → “who has birthday/nameday today”.
    - Notify the user (your family) **OR** send automatic greeting.
3. **Communication channels**

    - Option A → System notification (if it’s a desktop app).
    - Option B → Email greeting (easiest automation).
    - Option C → Viber message (requires API integration).

4. **Additional features**
    - Add/edit/remove clients in the DB.
    - Quick search for clients (by name, ΕΓΗ, phone, etc.).

⚙ Core System Architecture

You basically need 3 layers:

1. **Database**
    - Holds client info:
        - Full name
        - ΕΓΗ → used for birthday extraction
        - Nameday info (or deduced from name)
        - Contact (email, phone, viber ID)
    - Tech options:
        - SQLite (simple, file-based, good for desktop/small web).
        - PostgreSQL/MySQL (if you want scalability).
2. **Business logic (app/service)**
    - Imports data from Excel once.
    - Scheduler checks every morning → compares today’s date to birthdays/namedays.
    - Generates greetings.
    - Sends either email or Viber.
3. **Interface**
    - Could be **desktop app (C# WPF/WinForms)**, **web app (React/Angular frontend + Kotlin/Java backend)**, or even **mobile app**.

    - Lets family add new clients, search, and browse.

□ Notifications / Auto Messages

- **Email**:
    - Easiest. Use SMTP (Gmail, Outlook, or a transactional service like SendGrid).
    - You can format birthday greetings nicely.
- **Viber**:
    - Needs integration with **Viber API** (Viber Business Messages).
    - You’d need to register a business account.
    - More setup, but more personal.
- **Local notification (desktop/mobile)**:
    - Just reminds the user → they send the greeting themselves.
    - Easiest technically, but less automated.

□ Suggested Stack (based on your skills + needs)

Since you know **C#**, and this is for your **family business**, I’d recommend:

- **Backend + Database:**
    - C# with .NET 8 (ASP.NET Core Web API).
    - Use **SQLite** (simple, portable) or PostgreSQL (if you expect growth).
- **Frontend / UI:**
    - Start with **desktop app (WinForms/WPF)** → fastest for you to build.
    - Later, you can add a **web frontend** that talks to the same API.
- **Notifications:**
    - First stage → system pop-up + log of today’s birthdays/namedays.

    - Second stage → email integration.
    - Optional advanced stage → Viber integration.

Migration from Excel

1. Parse the existing Excel (easy with `ClosedXML` or `EPPlus` in C#).
2. Insert rows into DB with schema like:

```
sqlCopy codeCREATE TABLE Clients (
    Id INTEGER PRIMARY KEY,
    FullName TEXT,
    EGN TEXT,
    BirthDate DATE,
    Nameday TEXT,
    Email TEXT,
    Phone TEXT
);
```

3. Once DB is set, you no longer need Excel.

Development Roadmap

1. **MVP** (few weeks)
    - Migrate Excel → DB.
    - Build CRUD (Create/Read/Update/Delete) interface.
    - Add birthday/nameday calculation.
    - Daily “today’s events” popup.
2. **Phase 2**
    - Add email sending (SMTP).
    - Generate greeting messages automatically.
3. **Phase 3 (optional)**
    - Add Viber Business integration.
    - Build a small web frontend (Blazor, React, or Angular).

⚖ So:

- If you want something **quickly usable** → C# desktop app (WPF + SQLite).
- If you want **expandability (web + mobile in future)** → ASP.NET Core Web API backend + web frontend later.

Would you like me to sketch a **basic C# project structure (with database + daily birthday check)** so you can start prototyping this system?

i prefer we spent a bit more time but make it more maintanable and scalable for the future also could you tell me more about vber's api like pricing usage and such

Nice — going for something maintainable & scalable is smart. Below I outline what that means for your system architecture & tech choices, *and* what I found out about Viber’s Business Messages API (pricing, limits, pros/cons) so you can decide if it’s worth including.

⚙ Making the system more maintainable & scalable

Here are design choices & practices to invest in up front so your system can grow without becoming a nightmare:

Area	Key Considerations	Suggestions
Architecture	Decouple components so each does one job (UI, backend, scheduler, messaging)	Use a REST API + separate UI(s). The backend handles database, logic (birthday/nameday detection), and communication with external services (email, Viber).
Database	Reliable, supported, easy to migrate	Use PostgreSQL or SQL Server. If

Area	Key Considerations	Suggestions
		small-scale at first, SQLite is OK but harder to scale. Support migrations.
Scheduler / jobs	Sending reminders automatically	Use something like a scheduled job system: e.g. CRON on a server, Windows Scheduled Tasks, or a background service. Or in web apps: Hangfire (C#), Quartz, etc.
Modularity / configurability	Externalize configuration	Things like email/Viber credentials, message templates, scheduling times are stored in config or DB, not hardcoded.
Logging & monitoring	For debugging & operations	Log what messages got sent, failures, delivery status. Maybe simple dashboards.
Testing	To avoid regressions	Unit tests for date/ΕΓΗ parsing, nameday logic; integration tests for sending messages (can mock).
Security / Privacy	since personal data involved	Secure the database, encrypt communication (HTTPS), handle consent (especially for messaging), follow local laws (e.g. GDPR if in EU).

Area	Key Considerations	Suggestions
API abstraction	If you use Viber (or email, or other channels)	Define an interface (e.g. IMessageSender) so you can swap/augment channels without rewriting logic.

If you build with these in mind, even though it takes more time now, future changes (adding mobile client, more messaging channels, etc.) will be smoother.

□ Viber’s Business Messages API: What I Found

Here are features, uses, pricing, limits, and stuff to watch out for about Viber’s API.

□ What the API offers / key features

- You can send **transactional**, **promotional**, **conversational messages**. forbusiness.viber.com+2developers.cm.com+2
- Supports rich media: text, images, videos, buttons, etc. Infobip+2Infobip+2
- Messages show up in a **Business Inbox** for users, so business messages are separated from regular chats. help.viber.com+1
- Has **delivery reports / status callbacks** to know if messages were delivered. Mobica+2developers.cm.com+2
- Can have fallback: if a user doesn’t have Viber or message isn’t delivered, some providers support sending SMS fallback. Mobica+1

□ Pricing & Minimum Fees

This is variable, depending on country, volume, message type (promotional vs transactional), and the provider / partner you use. Some specifics:

- **Infobip** – Viber Business Messaging: has a **minimum monthly commitment fee**. E.g. for destinations in certain countries (Bulgaria is one), the minimum monthly fee per active Sender/Service ID is **150 €**. Infobip

- **Edna.io** has rates for various countries. For Bulgaria, delivered-message cost for **Service (transactional)** messages ~ **€0.023** per message, for **Promotional** ~ **€0.039**, and a **minimum monthly fee** ~ **€195**. edna
- **Messaggio**: example for Bulgaria: transactional about **€0.0207**, promotional about **€0.0345**, with a monthly minimum commitment of ~ **€175**. SMSBAT
- Some providers like local ones (e.g. Connectix in Bulgaria) will have different pricing tiers depending on how many messages you send per month. connectix.bg

---

⚠ Usage Limits, Policies & Other Constraints

- **Fair Usage**: Viber has a “Fair Usage Policy” for small/micro businesses. If you exceed certain thresholds (e.g. number of messages / business size), they may limit or charge differently. Viber
- **Session limits**: If there’s an ongoing conversation (user initiated), there may be rules (window of time, number of messages) governing how many messages you can send in that session. E.g. with Infobip, there are “session-based” charges. Infobip+1
- **Approval of templates**: For transactional messages or templated messages, the content is often reviewed/approved by Viber or via partner. Templates must follow Viber's guidelines. If you send something not approved, you may be charged at the promotional rate or message may be rejected. Infobip+1
- **Minimum monthly spend**: Even if you send few messages, you might need to pay a minimum fee for having the sender ID / service active. Infobip+2SMSBAT+2
- **Billing only for delivered messages**: Most providers only charge for messages that are confirmed delivered. If not delivered (due to user not on Viber, offline, etc.), either it’s dropped or fallback (like SMS) may apply. Mobica+2connectix.bg+2

Is it viable for your use case?

Let’s map this against what you described (birthday/nameday greetings, likely relatively low volume, mostly automated):

Pros:

- Very direct and personal channel. Messages through Viber are likely to be seen quickly.
- If you automate greeting messages, this gives “set-and-forget” for that channel.
- If recipients are using Viber, cost per message might be reasonable compared to SMS or paid alternative.

Challenges:

- The monthly minimum fees might be nontrivial, especially if volume is low. If you only send e.g. 30–100 messages/month, you may still pay a fee as if you sent more.
- Template approval adds overhead. You can’t just send arbitrary content unless it's approved.
- If recipients don’t use Viber, you’ll need fallback (SMS, email) which complicates logic & cost.
- Management of sender ID, account setup, compliance takes some time & maybe costs.

□ What I’d Suggest (Given Your Situation)

Based on your needs, here’s a strategy:

1. **Start with email + possibly SMS fallback** for greetings. Email is almost always cheaper, simpler, and sufficient for many.
2. **Evaluate Viber as optional channel** once you see usage / demand. If many clients do use Viber and the cost is acceptable, implement Viber sending as another “outbound channel” in your architecture.
3. When doing Viber:
    - Use a provider (Infobip, Messaggio, or local provider in Bulgaria) to simplify setup, template registration, sender ID, etc.
    - Make sure the system checks for message delivery and supports fallback (if not delivered, send email/SMS).
    - Keep messaging templates flexible but consistent (e.g. “Happy Birthday, {Name}!”) to ease approval.

4. Always compute: **estimated costs** for say 500 messages/month, then compare that to budget / benefit (if people prefer Viber messages vs email, etc.).

If you like, I can pull together **a sample cost estimate** for your projected volume (say 100-500 automated greetings/month) using Bulgarian providers for email, Viber, SMS so you can see which mix makes sense. Do you want me to do that?

okay lets right now focus on starting this project and i gotta tell you i am not at all fond of winform apps so if we can avoid using them that would be great

Got it ▯ no WinForms — that makes perfect sense if you want scalability, maintainability, and a modern stack.

Here's a structured way to **start your project**:

---

1. **Architecture Choice**

Since you want something future-proof and not tied to an old desktop tech like WinForms, I'd recommend:

- **Backend**:
    - ASP.NET Core (C#) → REST API or gRPC.
    - Handles clients, birthdays, notifications, etc.
    - Database integration (SQL Server, PostgreSQL, or even SQLite for start).
- **Frontend Options**:
    - **Web app (React, Angular, or Blazor)** → Best if you want cross-device access without installing apps.
    - **Mobile app (React Native or Kotlin/Swift)** → Better UX but more dev effort.
- **Notifications**:
    - Email → SMTP server or service like SendGrid.

    - Viber/WhatsApp → Use Infobip, Twilio, or Viber Business API.

So your family would just open a web app (or later install a mobile app if you expand).

---

2. Database Migration from Excel

- Step 1: Design a simple relational DB schema:

  ```
  scssCopy codeClients(id, name, birthday, nameday, email, phone, notes, ...)
  ```

- Step 2: Create a migration script that imports the Excel sheet once into the DB.
- Step 3: Build a CRUD interface (Add/Edit/Delete/Find clients).

---

3. Notification System

- **Scheduler** (e.g., Quartz.NET, Hangfire, or background hosted service in ASP.NET Core).
- Every morning at 9:00, check DB for birthdays/namedays.
- Send either:
    - Email → cheap/free at low volume.
    - Viber → via API (but this might require a paid service).

---

4. Development Plan

1. **Backend skeleton**:
    - ASP.NET Core Web API project.
    - Add EF Core for DB.
    - Add first model: `Client`.
2. **Database setup**:
    - Use PostgreSQL (free + scalable).
    - Migrate Excel → DB.
3. **Frontend MVP**:

    - React web app with table search + forms.

4. **Notification MVP**:
    - Add email sending (cheapest, easiest).
    - Later expand to Viber/WhatsApp/SMS.

 This way:

- You avoid WinForms.
- You start with a clean, scalable architecture.
- You can deploy it anywhere (local server, cloud, even free tiers).

 Do you want me to draft you the **first prototype structure** (like solution folders, first DB schema, and which NuGet/npm packages you'll need), so you can start coding right away?

yeah i do cause i dont yet have the excel file so i dont really now what they need to be in the DB for now

Perfect — since you don't yet have the Excel file, we'll design a **minimal but extendable schema** so you can:

1. Start coding immediately.
2. Expand later when you see the real data.

---

1. Project Structure (ASP.NET Core + EF Core + React)

```
bashCopy code/ClientNotifier
  /backend
    /ClientNotifier.API      # ASP.NET Core Web API
    /ClientNotifier.Core     # Business logic, models, services
    /ClientNotifier.Data     # EF Core DbContext, migrations,
repositories
  /frontend
    /client-notifier-ui      # React web app
```

This keeps things **modular and scalable**.

2. Database Schema (Initial Draft)

We’ll keep it simple but flexible:

Clients table

```
vbnetCopy codeId            (int, PK, auto-increment)
FirstName     (string, required)
LastName      (string, optional)
Email         (string, optional)
PhoneNumber   (string, optional)
Birthday      (date, optional)
NameDay       (date, optional)
Notes         (string, optional, free text)
CreatedAt     (datetime, default now)
```

Later, if your family needs more fields (like address, tags, categories), you just add migrations.

3. Backend (C#, ASP.NET Core)

- **NuGet packages** you’ll need:
    - `Microsoft.EntityFrameworkCore.SqlServer` (or `Npgsql` if PostgreSQL)
    - `Microsoft.EntityFrameworkCore.Tools`
    - `Swashbuckle.AspNetCore` (Swagger for API testing)
    - `MailKit` (for email notifications)
    - `Quartz` or `Hangfire` (for scheduled jobs, later)

API endpoints to start:

```
bashCopy codeGET    /api/clients          → list all clients
GET    /api/clients/{id}     → get one client
POST   /api/clients          → add client
PUT    /api/clients/{id}     → update client
DELETE /api/clients/{id}     → delete client
```

□ 4. Frontend (React)

- **npm packages**:
    - `axios` (for API calls)
    - `react-router-dom` (routing)
    - `antd` or `mui` (table + forms UI components)

Pages to start:

- `ClientsList` → searchable table of clients.
- `ClientForm` → add/edit client.

□ 5. Notifications (Future step)

Start with **email** (free/cheap), then later expand to Viber/WhatsApp.

- For now, we just log: “Today is X’s birthday, send email to …”.
- Later you wire in **SMTP → Gmail / SendGrid**.

□ With this, you can:

- Start writing code before you even see the Excel file.
- Expand the DB schema later with migrations.
- Test API + UI locally quickly.

Do you want me to **write out the first C# models + DbContext + initial migration setup** for you, so you just copy-paste and run `dotnet ef migrations add InitialCreate`?

about the name day i dont really like the idea of them putting in the date the same goes for the birthday the bday we can extract from their егн and for the name day i am not sure yet how we could implement it

Perfect, that actually simplifies the **birthday** part and raises a good point about **namedays**. Let’s break it down:

Birthday

- You **don’t need them to input the date**.
- Instead, you extract it from **ЕГН**.
  - First 6 digits of ЕГН → `YYMMDD`
  - Month adjustments based on century (as we discussed earlier).
- Store **Birthday** in the DB after extraction (so you can query easily).
- The `EGN` field becomes mandatory, the `Birthday` field can be **computed once and stored**, or **computed on the fly** if you prefer.

Nameday

Namedays are **name-based**, not date-based, but there’s no universal mapping. Options:

1. **Predefined mapping table**
   - Maintain a table of `Name → Nameday`.
   - Example:

     ```
     nginxCopy codeName        Nameday
     Ivan        07-01
     Georgi      06-05
     Maria       08-15
     ```

   - When adding a client, system automatically assigns the nameday based on `FirstName`.
   - Pros: simple, predictable.
   - Cons: must maintain/update table if you want more names.
2. **Manual input with suggestions**
   - When adding a client, the system suggests nameday based on `FirstName`.
   - User can adjust if needed.

    - Pros: flexible for uncommon names.
    - Cons: more manual effort.

3. **External API**
    - Some websites provide nameday APIs (e.g., `nameday.abalin.net`).
    - Could query `FirstName → Nameday`.
    - Pros: automatic, up-to-date.
    - Cons: depends on internet and API availability.

Recommended Approach for MVP

- **Step 1 (minimal):**
    - Maintain a small **local mapping table** of common names → namedays.
    - Automatically assign nameday when client is added.
    - Let users override if needed (optional field).
- **Step 2 (future):**
    - Consider querying an API to cover more names dynamically.
    - This keeps the system maintainable & scalable without forcing manual input.

 Database tweak for this:

Clients table

```
scssCopy codeId            (int, PK)
FirstName     (string)
LastName      (string)
EGN           (string, required)
Birthday      (date, computed from EGN)
Nameday       (date, computed from name or mapping)
Email         (string)
PhoneNumber   (string)
Notes         (string)
CreatedAt     (datetime)
```

NamedayMapping table (for now small & extendable)

```
pgsqlCopy codeId        (int, PK)
Name      (string, unique)
Nameday   (date)
```

- When adding a client:
    a. Extract birthday from EGN.
    b. Look up first name in `NamedayMapping` → assign nameday automatically.

If you want, I can **draft the C# EF Core models + code for automatic birthday extraction + nameday lookup** so that the first prototype is almost fully automatic when adding a client.

okay so we do it like that: one table for people and 1 for nameday mapping ? lets start i curently dont have the excel table but we do know what the columns would contain so lets start building the app

Exactly □ — one table for **People** and one for **NamedayMapping**. That keeps things clean and scalable. Since you want to start right away, I'll give you the **first working building blocks** of the backend in C# (ASP.NET Core + EF Core).

□ Step 1: Create the Solution Structure

```
bashCopy codedotnet new sln -n ClientNotifier
mkdir ClientNotifier.API ClientNotifier.Core ClientNotifier.Data
dotnet new webapi -n ClientNotifier.API
dotnet new classlib -n ClientNotifier.Core
dotnet new classlib -n ClientNotifier.Data
dotnet sln add ClientNotifier.API/ClientNotifier.API.csproj
dotnet sln add ClientNotifier.Core/ClientNotifier.Core.csproj
dotnet sln add ClientNotifier.Data/ClientNotifier.Data.csproj
dotnet add ClientNotifier.API reference ClientNotifier.Core
ClientNotifier.Data
dotnet add ClientNotifier.Data package
```

```
Microsoft.EntityFrameworkCore.SqlServer
dotnet add ClientNotifier.Data package
Microsoft.EntityFrameworkCore.Design
dotnet add ClientNotifier.API package Swashbuckle.AspNetCore
```

Step 2: Define Entities (in `ClientNotifier.Core/Models`)

Person.cs

```
csharpCopy codenamespace ClientNotifier.Core.Models
{
    public class Person
    {
        public int Id { get; set; }
        public string FirstName { get; set; } = null!;
        public string? LastName { get; set; }
        public string EGN { get; set; } = null!;
        public DateTime Birthday { get; set; }   // extracted from EGN
        public DateTime? Nameday { get; set; }   // looked up from
mapping
        public string? Email { get; set; }
        public string? PhoneNumber { get; set; }
        public string? Notes { get; set; }
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    }
}
```

NamedayMapping.cs

```
csharpCopy codenamespace ClientNotifier.Core.Models
{
    public class NamedayMapping
    {
        public int Id { get; set; }
        public string Name { get; set; } = null!;
        public DateTime Nameday { get; set; }
    }
}
```

□ Step 3: Setup DbContext (in `ClientNotifier.Data`)

AppDbContext.cs

```
csharpCopy codeusing ClientNotifier.Core.Models;
using Microsoft.EntityFrameworkCore;

namespace ClientNotifier.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) {}

        public DbSet<Person> People { get; set; }
        public DbSet<NamedayMapping> NamedayMappings { get; set; }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<NamedayMapping>()
                .HasIndex(n => n.Name)
                .IsUnique();
        }
    }
}
```

□ Step 4: Configure EF Core in API (`Program.cs`)

```
csharpCopy codeusing ClientNotifier.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Database connection
builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

```
// Add controllers & Swagger
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseAuthorization();
app.MapControllers();
app.Run();
```

appsettings.json

```
jsonCopy code{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=ClientNotifierDb;Trusted_Connection=True;TrustServerCertificate=True"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Step 5: Extract Birthday from ЕГН (Utility)

Create `EgnUtils.cs` in `Core/Services`:

```
csharpCopy codenamespace ClientNotifier.Core.Services
{
    public static class EgnUtils
    {
        public static DateTime ExtractBirthday(string egn)
        {
            if (egn.Length < 6)
                throw new ArgumentException("Invalid EGN");

            int year = int.Parse(egn.Substring(0, 2));
            int month = int.Parse(egn.Substring(2, 2));
            int day = int.Parse(egn.Substring(4, 2));

            // Handle century encoding in EGN
            if (month > 40) { year += 2000; month -= 40; }
            else if (month > 20) { year += 1800; month -= 20; }
            else { year += 1900; }

            return new DateTime(year, month, day);
        }
    }
}
```

Step 6: Generate Migration

```
bashCopy codedotnet ef migrations add InitialCreate -p
ClientNotifier.Data -s ClientNotifier.API
dotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

This will create your DB with `People` and `NamedayMappings`.

At this point you'll have:

- A running **ASP.NET Core API** with a connected database.
- Two tables ready for storing people and nameday mappings.
- Automatic birthday extraction logic from ЕГН.

Do you want me to **add the first Controller** (`PeopleController`) with a `POST` endpoint that automatically:

- extracts birthday from EGN,
- looks up nameday by first name,

- saves the new person?

i want to be able to follow some plan you know. for example i created a github repo new branch called feature/init but in the future i would only have the branch i mean in work we use jira tickets some documentation you know in here for now i only have the commits to go trough and somethimes they may not be enough

That’s a really good instinct □ — if you already think in terms of **Git branches, commits, Jira-style workflow, and documentation**, then we can structure this project like you’re working in a real-world dev team. That will:

- Make it easier for you to **track progress**.
- Keep the project **maintainable** over time.
- Give you a **professional workflow** that mirrors what you’ll see in internships/jobs.

□ Suggested Workflow Plan

1. Branching Strategy

- **main** → stable, production-ready code.
- **develop** → integration branch, feature branches are merged here first.
- **feature/*** → per-task branches (e.g. `feature/init-db`, `feature/people-crud`).

So:

```
bashCopy codegit checkout -b feature/init-db
# do work, commit
git push origin feature/init-db
```

Later, merge into `develop` with a PR (even if you review it yourself).

---

2. Documentation

Even if it’s just you, keep a **docs/** folder in the repo:

- `docs/requirements.md` → what the system should do (like your family's needs).
- `docs/design.md` → database schema, architecture decisions.
- `docs/todo.md` or a GitHub Project board → tasks/features to implement.
- `docs/changelog.md` → a human-friendly log of major updates.

That way, future you (or anyone else) won't have to dig only through commits.

---

3. Task Management

If you don't want full Jira, you can:

- Use **GitHub Issues** + link them to commits/PRs.
- Or keep a simple `TODO.md` where you list tasks like:

  ```
  cssCopy code- [x] Setup solution structure
  - [x] Add Person + NamedayMapping entities
  - [ ] Implement PeopleController
  - [ ] Add email notifications
  ```

This gives you a mini backlog you can close step by step.

---

4. Commit Messages

Follow a simple convention (like Conventional Commits):

```
vbnetCopy codefeat: add Person entity and DbContext
fix: correct birthday extraction from EGN
docs: update requirements with nameday logic
chore: bump EF Core package version
```

This makes the history **self-explanatory**.

---

5. Development Plan (Milestones)

Here's how I'd break down the first iterations:

M1 – Initialization

- Setup solution (API + Core + Data projects).

- Add models (Person, NamedayMapping).
- Configure DbContext + migrations.

M2 – Basic CRUD

- Add PeopleController (GET, POST, PUT, DELETE).
- Add NamedayMappingController.
- Swagger docs for testing.

M3 – Business Logic

- Auto-extract Birthday from EGN on insert.
- Lookup Nameday from mapping.

M4 – Notifications

- Add background job scheduler.
- Send email for birthdays/namedays.

M5 – Frontend (React)

- Basic UI to list/add/edit people.
- Search/filter.

M6 – Expansion

- Viber/WhatsApp integration.
- Roles/users if needed.
- Deploy to server.

With this, you’ll have both **commit history** AND a **clear progress log**, like a mini Jira/Trello board + docs.

Do you want me to set up a **docs/requirements.md** draft for you now (based on what you told me about the Excel → DB migration, birthdays from ЕГН, nameday mapping, notifications), so you already have the first documentation file to commit along with your `feature/init` branch?

how can i usegithub issues

Great question □ GitHub Issues can be your “lightweight Jira.” They let you track tasks, bugs, and enhancements directly inside your repo. Here’s how to use them effectively:

□ 1. Enable Issues

- Go to your **repo on GitHub**.
- Under **Settings → General → Features**, make sure **Issues** is checked.

Now you’ll see an **Issues** tab at the top.

□ 2. Create an Issue

1. Click **Issues → New Issue**.
2. Give it a **title** (short summary).
    - Example: `Add Person entity and DbContext`.
3. Add a **description** (details, acceptance criteria).

```
markdownCopy code## Goal
Create the initial Person model with required fields.

## Tasks
- [ ] Add Person.cs in Core/Models
- [ ] Add DbSet<Person> in AppDbContext
- [ ] Run first EF migration
```

4. Add **labels** like `feature`, `bug`, `enhancement`, `documentation`.

□ 3. Link Issues to Branches & Commits

When you work on something, create a branch named after the issue:

```
bashCopy codegit checkout -b feature/1-init-person-model
```

(`1` is the issue number).

In commit messages, reference the issue:

```
bashCopy codegit commit -m "feat: add Person entity #1"
```

- On GitHub, the commit will be linked to Issue **#1**.
- You can even auto-close issues with keywords:

  ```
  bashCopy codegit commit -m "feat: add Person entity (closes #1)"
  ```

  → When merged, GitHub automatically closes the issue.

---

4. Use Milestones & Projects (Optional)

- **Milestones** = group issues into a release (e.g. “M1 – Init”).
- **Projects** = Kanban board (like Jira).
  - You can create columns `To Do`, `In Progress`, `Done`.
  - Drag issues between them as you work.

---

5. Example Workflow

1. Create Issue: `#1 Setup database schema`.
2. Create Branch: `feature/1-db-schema`.
3. Work on it, commit with `#1` in message.
4. Open Pull Request to `develop`.
5. Merge → issue auto-closes, history is clean.

Result:

- You’ll have **tasks tracked like Jira tickets**.
- Each commit/branch/PR ties back to an **Issue**.
- Repo itself becomes both code + documentation of your process.

Do you want me to draft your **first set of GitHub Issues** (like backlog items: init DB, add Person entity, add NamedayMapping, etc.) so you can just paste them into your repo and start working in a Jira-style flow?

Is your feature request related to a problem? Please describe. A clear and concise description of what the problem is. Ex. I'm always frustrated when [...] **Describe the solution you'd like** A clear and concise description of what you want to happen. **Describe alternatives you've considered** A clear and concise description of any alternative solutions or features you've considered. **Additional context** Add any other context or screenshots about the feature request here. write me short ticket in this stile for the initial setup i currently dont have any logic written just the context models and program cs

Got it ▯ You want a **GitHub Issue ticket** in the feature request template style, but tailored to your initial setup work. Here’s a short and clean draft you can paste directly into GitHub:

Is your feature request related to a problem? Please describe.
We need to bootstrap the project structure so future features can be built in a maintainable way. Currently, there is no database, no migrations, and only minimal scaffolding exists (`Program.cs`, model classes).

Describe the solution you'd like

- Initialize solution structure (`API`, `Core`, `Data`).
- Add basic models: `Person`, `NamedayMapping`.
- Configure `AppDbContext` and connection string.
- Add initial EF migration and update database.
- Enable Swagger for API exploration.

Describe alternatives you've considered

- Directly building logic without a structured solution, but this would lead to poor maintainability and scaling issues later.

Additional context
This is the foundation milestone (M1). No business logic yet, just scaffolding and basic entities to prepare for CRUD operations in the next step.

□ Would you like me to also draft **separate smaller tickets** for each sub-task (e.g., “Add Person model”, “Setup EF Core”, “Enable Swagger”), or do you prefer to keep the **whole initialization as one Issue**?

```
dotnet ef migrations add InitialCreate -p ClientNotifier.Data -s
ClientNotifier.API   Welcome to .NET 9.0!   ---------------------   SDK Version:
9.0.305   Telemetry   ---------   The .NET tools collect usage data in order to
help us improve your experience. It is collected by Microsoft and shared with
the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using
your favorite shell.   Read more about .NET CLI Tools telemetry:
https://aka.ms/dotnet-cli-telemetry   ----------------   Installed an ASP.NET Core
HTTPS development certificate.   To trust the certificate, run 'dotnet dev-
certs https --trust'   Learn about HTTPS: https://aka.ms/dotnet-https
----------------   Write your first app: https://aka.ms/dotnet-hello-world   Find out
what's new: https://aka.ms/dotnet-whats-new   Explore documentation:
https://aka.ms/dotnet-docs   Report issues and find source on GitHub:
https://github.com/dotnet/core   Use 'dotnet --help' to see available
commands or visit: https://aka.ms/dotnet-cli
-------------------------------------------------------------------------------------   Possible
reasons for this include:   * You misspelled a built-in dotnet command.   * You
intended to execute a .NET program, but dotnet-ef does not exist.   * You
intended to run a global tool, but a dotnet-prefixed executable with this
name could not be found on the PATH.   dotnet : Could not execute because
the specified command or file was not found.   At line:1 char:1   + dotnet ef
migrations add InitialCreate -p ClientNotifier.Data -s Clie ...   +
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~   + CategoryInfo : NotSpecified:
(Could not execu... was not found.:String) [], RemoteException   +
FullyQualifiedErrorId : NativeCommandError   PM> dotnet ef database update
-p ClientNotifier.Data -s ClientNotifier.API   Possible reasons for this include:
* You misspelled a built-in dotnet command.   * You intended to execute
a .NET program, but dotnet-ef does not exist.   * You intended to run a global
tool, but a dotnet-prefixed executable with this name could not be found on
the PATH.   dotnet : Could not execute because the specified command or
file was not found.   At line:1 char:1   + dotnet ef database update -p
ClientNotifier.Data -s ClientNotifier.AP ...   +
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~   + CategoryInfo : NotSpecified:
(Could not execu... was not found.:String) [], RemoteException   +
FullyQualifiedErrorId : NativeCommandError
```

Ah, that error is classic ▯ — it just means the **EF Core CLI tools** (`dotnet-ef`) aren’t installed yet. The runtime packages are already in your `ClientNotifier.Data` project, but the CLI tool is separate.

▯ Fix

Run this once globally:

```
bashCopy codedotnet tool install --global dotnet-ef
```

Verify it worked:

```
bashCopy codedotnet ef --version
```

Now your original commands should work fine:

```
bashCopy codedotnet ef migrations add InitialCreate -p ClientNotifier.Data -s ClientNotifier.API
dotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

▯ Alternative (per-project instead of global)

If you don’t want to install globally, you can install it as a local tool in your solution:

```
bashCopy codedotnet new tool-manifest    # if you don’t have one already
dotnet tool install dotnet-ef
```

Then you can run:

```
bashCopy codedotnet tool run dotnet-ef migrations add InitialCreate -p ClientNotifier.Data -s ClientNotifier.API
```

⚡ Pro tip: Most teams prefer the **global install** so you can just use `dotnet ef` directly.

Do you want me to also give you the **exact EF Core package versions** you should add for `.NET 9`, so migrations don’t complain about version mismatches?

Your startup project 'ClientNotifier.API' doesn't reference Microsoft.EntityFrameworkCore.Design. This package is required for the Entity Framework Core Tools to work. Ensure your startup project is correct, install the package, and try again. but i have it installed

Got it That error usually comes from one of two things:

---

1. The package is in the **wrong project**

- `Microsoft.EntityFrameworkCore.Design` must be installed in the **startup project** (`ClientNotifier.API`), not only in the `ClientNotifier.Data` project.
  Check with:

  ```
  bashCopy codedotnet list ClientNotifier.API package
  ```

If it’s missing, add it:

```
bashCopy codedotnet add ClientNotifier.API package
Microsoft.EntityFrameworkCore.Design
```

---

2. It’s installed, but only as a runtime dependency

`Design` should be a *development dependency*. In `.csproj`, it should look like this:

```
xmlCopy code<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="9.0.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

If you just see a plain `<PackageReference ... />`, migrations may still fail.

---

3. Startup project mismatch

When running the command, make sure `-s ClientNotifier.API` points to the actual startup project you’ll run (`Program.cs` with `WebApplication.CreateBuilder`).

▯ Try this sequence:

```
bashCopy codedotnet add ClientNotifier.API package Microsoft.EntityFrameworkCore.Design
dotnet restore
dotnet ef migrations add InitialCreate -p ClientNotifier.Data -s ClientNotifier.API
```

Do you want me to also show you a minimal `.csproj` setup for **Data** and **API**, so you can compare with yours and be sure nothing is missing?

great i created a ticket a brannch a PR merged it everything is okay so we can continue but before we do so tell me how can i view my db i have dataGrip but is it enough to open it to follow it and also i upload my project in github but how can my mother use it for example shouldnt i contanarize it?

Nice ▯ you’re already working like a team player with tickets/branches/PRs ▯

Let me split your two questions:

---

1. Viewing your DB

- **DataGrip** is more than enough.
- Just make sure your EF Core connection string points to a real SQL instance (e.g. `localhost` with SQL Server, PostgreSQL, or SQLite).
- Once migrations are applied, you can connect to that DB from DataGrip, explore tables, and run queries.
  ▯ So yes, simply opening DataGrip and connecting to the DB is enough to follow changes.

---

2. Sharing / running the project (e.g. your mom using it)

There are a couple of approaches depending on how “user-friendly” you want this to be:

- **Local run (for developers)**
  If it’s another developer, cloning from GitHub and running `dotnet run` is fine.
- **Containerization (recommended for easy distribution)**
  - If you put the API into a Docker container, you can ship it as one command:

    ```
    bashCopy codedocker run -p 5000:5000 your-app-image
    ```

  - Add a database container (like `postgres` or `mssql`) and use **docker-compose** so anyone (even your mom) can start the whole system with:

    ```
    bashCopy codedocker-compose up
    ```

- **Deployment (for true end-users)**
  If you actually want your mom to use it like an app:
  - You’d host the API (Azure, AWS, or even a cheap VPS).
  - Then you’d give her a client UI (web or desktop) that calls your API.
  - She wouldn’t need GitHub at all — just open the web page or app.

⚖ So:

- For **development / following DB** → DataGrip is enough.
- For **sharing the project easily** → yes, containerizing is the right path (Docker + docker-compose).
- For **real end-user use** → you’ll eventually need to host/deploy.

□ Do you want me to sketch out a **minimal docker-compose** setup for your API + DB, so you can start the project anywhere with one command?

first help me setup data grip to open the db

Perfect □ let’s get you connected in DataGrip.
First I need to know **which database provider you’re using with EF Core**.

In your `ClientNotifier.Data` project, open your `DbContext` class (something like `ApplicationDbContext`) and check what’s in `OnConfiguring` or what you passed in `Program.cs`.

For example, you might see:

- `options.UseSqlServer("...")` → SQL Server
- `options.UseNpgsql("...")` → PostgreSQL
- `options.UseSqlite("...")` → SQLite
- `options.UseMySql("...")` → MySQL/MariaDB

Can you paste me your connection string or tell me which `.UseXYZ` you’re using? That way I’ll give you the exact DataGrip steps for your DB.

this

Great □ since you’re using **SQL Server**, DataGrip can connect to it directly.
Here’s how to set it up:

1. Get your connection string

In your `Program.cs` (or appsettings.json if you moved it there) you’ll see something like:

```
csharpCopy
codeoptions.UseSqlServer("Server=localhost;Database=ClientNotifier;Trusted_Connection=True;MultipleActiveResultSets=true");
```

or

```
csharpCopy codeoptions.UseSqlServer("Server=(localdb)\\MSSQLLocalDB;Database=ClientNotifier;Trusted_Connection=True;");
```

Take note of:

- **Server** (e.g., `localhost`, `(localdb)\MSSQLLocalDB`)
- **Database** (e.g., `ClientNotifier`)
- **Authentication method** (Windows Auth or SQL login).

---

2. Open DataGrip

1. Click the **Database** tool window (usually on the right).
2. Press **+** → **Data Source** → **SQL Server**.

---

3. Enter connection details

- **Host**: if you're using `localhost` → put `localhost`.
- **Port**: default is `1433`.
- **Database**: `ClientNotifier` (or whatever you called it).
- **Authentication**:
    - If `Trusted_Connection=True`, choose **Windows Authentication**.
    - Otherwise, use **SQL Server Authentication** and enter `User` + `Password`.

---

4. Test the connection

Click **Test Connection**. If DataGrip can't connect, it will offer to download the SQL Server driver — let it.

---

5. Explore your DB

Once connected, you'll see your database in the **Database** window. Expand it → **Tables** → you should see your EF Core tables (e.g., `People`, `NameDayMapping`).

□ Quick tip:
If you’re using **LocalDB** (`(localdb)\MSSQLLocalDB`), DataGrip sometimes struggles. You may need to:

- Change host to `localhost`
- Set port manually (run `sqllocaldb info MSSQLLocalDB` in PowerShell to find it).

Do you know if you’re running against **SQL Server Express / LocalDB** or a full **SQL Server instance**? That will slightly change the setup.

options.UseSqlServer("Server=localhost;Database=ClientNotifier;Trusted_Connection=True;MultipleActiveResultSets=true"); i use this

Perfect — that means you’re connecting to a **local SQL Server instance** (not LocalDB). That’s much easier □

Here’s how to hook it up in DataGrip:

□ Step 1: Add a new data source

1. Open **DataGrip**.
2. Go to the **Database** tool window → click **+** → **Data Source** → **SQL Server**.

□ Step 2: Configure the connection

- **Host**: `localhost`
- **Port**: `1433` (default for SQL Server — if you didn’t change it).
- **Authentication**:
  - Since you’re using `Trusted_Connection=True`, select **Windows Authentication**.
- **Database**: `ClientNotifier`

Step 3: Test connection

Click **Test Connection**.

- If DataGrip asks to download SQL Server JDBC driver → accept.
- If it succeeds → you'll see a green check

Step 4: Browse your tables

Once connected:

- Expand **ClientNotifier → Tables**
- You'll see the tables EF created (`People`, `NameDayMapping`, and `__EFMigrationsHistory`).
- Right-click → **Jump to Data** to view records.

⚠ If you get an error like *"Cannot connect to localhost:1433"*:

1. Make sure SQL Server is running → open **Services** → look for **SQL Server (MSSQLSERVER)**.
2. Verify TCP/IP is enabled → open **SQL Server Configuration Manager** → **SQL Server Network Configuration** → **Protocols for MSSQLSERVER** → enable **TCP/IP**.
3. Restart the SQL Server service.

Do you want me to give you the **exact DataGrip config fields filled in** for your case (with screenshots replacement in text), so you can copy-paste?

[08S01] The TCP/IP connection to the host localhost, port 1433 has failed. Error: "Connection refused: getsockopt. Verify the connection properties. Make sure that an instance of SQL Server is running on the host and accepting TCP/IP connections at the port. Make sure that TCP connections to the port are not blocked by a firewall.".

Ah, that error tells us exactly what's happening: **DataGrip can't reach your SQL Server on TCP port 1433**. With your connection string

(`Server=localhost;Trusted_Connection=True`) you’re probably using **LocalDB** (or the default SQL Server Express), which by default doesn’t listen on TCP — it only allows **named pipes / local connections**.

□ Option 1: Connect via LocalDB

Since your connection string doesn’t specify an instance name, it may be using the default LocalDB. DataGrip can connect to LocalDB like this:

1. **Change host**: `(localdb)\MSSQLLocalDB`
2. **Port**: leave empty
3. **Authentication**: **Windows Authentication**
4. **Database**: `ClientNotifier`

- This usually works without enabling TCP/IP because LocalDB uses **named pipes**.

□ Option 2: Enable TCP/IP for SQL Server

If you really want `localhost:1433` to work (like a real SQL Server instance):

1. Open **SQL Server Configuration Manager**
2. Go to **SQL Server Network Configuration → Protocols for [YourInstance]**
3. Enable **TCP/IP**
4. Right-click **TCP/IP → Properties → IP Addresses → IPAll → TCP Port** → set to `1433`
5. Restart the SQL Server service
6. Make sure **Windows Firewall** allows inbound traffic on 1433

Then DataGrip can use `localhost:1433` with Windows Auth.

□ Quickest path for you

Since you’re in development and just want to **inspect your DB**, the easiest is **Option 1**: connect using LocalDB instance `(localdb)\MSSQLLocalDB`.

- In DataGrip:

    - **Host**: `(localdb)\MSSQLLocalDB`
    - **Port**: leave blank
    - **Authentication**: Windows Auth
    - **Database**: `ClientNotifier`

You'll then see all your tables.

I can write you a **step-by-step DataGrip connection config for LocalDB** that works 100% with your current EF setup. Do you want me to do that?

lets start anew and if i need to change something in the source code for the connection

Perfect — let's reset and make sure your **DB connection works from scratch**, both in your source code and in DataGrip. I'll guide you step by step.

---

1 Decide which SQL Server you want to use

You have two main options for development:

Option	Pros	Cons
LocalDB (localdb)\MSSQLLocalDB	Lightweight, no TCP setup, works out of the box for dev	Only local, single-user
Full SQL Server / SQL Server Express	Can enable TCP/IP, share with others, closer to production	Must enable TCP/IP and firewall

For your project, **LocalDB is easiest**.

---

2 Update your connection string

In `Program.cs` or `appsettings.json`, change it to LocalDB style:

```
csharpCopy codeoptions.UseSqlServer(
    @"Server=(localdb)\
MSSQLLocalDB;Database=ClientNotifier;Trusted_Connection=True;MultipleA
ctiveResultSets=true"
);
```

- `(localdb)\MSSQLLocalDB` → points to the default LocalDB instance
- `Trusted_Connection=True` → uses Windows authentication
- `MultipleActiveResultSets=true` → EF Core requirement for multiple queries

If you want to use full SQL Server with TCP later, we can switch back easily.

---

3 Ensure the DB exists

1. Open **Command Prompt** or **PowerShell**
2. Run:

```
powershellCopy codesqllocaldb info
```

You should see `MSSQLLocalDB` listed. If not, create it:

```
powershellCopy codesqllocaldb create "MSSQLLocalDB"
sqllocaldb start "MSSQLLocalDB"
```

---

4 Update EF Core

Make sure your API project has **EF Core packages**:

```
bashCopy codedotnet add ClientNotifier.Data package
Microsoft.EntityFrameworkCore.SqlServer
dotnet add ClientNotifier.Data package
Microsoft.EntityFrameworkCore.Design
dotnet add ClientNotifier.API package
Microsoft.EntityFrameworkCore.Design
```

---

5 Apply migrations

From the solution root:

```
bashCopy codedotnet ef migrations add InitialCreate -p
ClientNotifier.Data -s ClientNotifier.API
dotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

- `-p` → project containing DbContext
- `-s` → startup project containing Program.cs

If `dotnet ef` fails, make sure the tool is installed:

```
bashCopy codedotnet tool install --global dotnet-ef
```

---

6 Connect in DataGrip

1. Open **Database → + → SQL Server**
2. **Host**: `(localdb)\MSSQLLocalDB`
3. **Port**: leave empty
4. **Authentication**: Windows Authentication
5. **Database**: `ClientNotifier`
6. **Test Connection** → should succeed
7. Expand → Tables → see `People`, `NamedayMapping`, etc.

□ At this point:

- Your API is configured to connect to LocalDB
- EF Core migrations have created the DB
- DataGrip can browse the tables and see inserted data

If you want, I can also **show a minimal Program.cs + DbContext snippet** with this LocalDB setup so your connection is ready to go without changes later.

Do you want me to do that?

[S1000][4060] Cannot open database "ClientNotifier" requested by the login. The login failed. [28000][18456] Login failed for user 'DESKTOP-FHB54D9\Nitro5'.

Ah, this error is common with **LocalDB / SQL Server** — it's basically saying:

> "The SQL Server instance exists, but the database `ClientNotifier` doesn't exist yet, and your login can't access it."

□ Step 1: Make sure the DB exists

LocalDB does **not automatically create the database**. You have two options:

Option A — Let EF Core create it

EF Core can create the database automatically when you run `dotnet ef database update`. Make sure your connection string points to the instance **without a pre-existing DB requirement**:

```
csharpCopy codeoptions.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=ClientNotifier;Trusted_Connection=True;MultipleActiveResultSets=true");
```

Then run:

```
bashCopy codedotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

- EF will create `ClientNotifier` for you.
- If this fails, it usually means the login doesn't have permissions to **create a database** in LocalDB.

Option B — Manually create the DB

1. Open **Command Prompt / PowerShell**
2. Launch SQL Server CLI:

```
powershellCopy codesqllocaldb info MSSQLLocalDB   # to ensure the instance exists
sqllocaldb start MSSQLLocalDB
```

3. Open **SQL Server Management Studio (SSMS)** or **DataGrip**, connect using **Windows Authentication**.

4. Run:

```
sqlCopy codeCREATE DATABASE ClientNotifier;
```

5. Then EF migrations should succeed because the DB exists.

□ Step 2: Verify login

Since your error mentions:

```
pgsqlCopy codeLogin failed for user 'DESKTOP-FHB54D9\Nitro5'
```

- This is your Windows account, which **should** have access to LocalDB by default.
- Make sure you connect in DataGrip using **Windows Authentication** (`Integrated Security=True`).
- If needed, in SQL Server you can run:

```
sqlCopy codeCREATE LOGIN [DESKTOP-FHB54D9\Nitro5] FROM WINDOWS;
ALTER SERVER ROLE sysadmin ADD MEMBER [DESKTOP-FHB54D9\Nitro5];
```

> Usually unnecessary for LocalDB, but it ensures full permissions.

□ Step 3: Retry

- After the DB exists and your Windows login has access, run:

```
bashCopy codedotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

- In DataGrip, connect using:
    - Host: `(localdb)\MSSQLLocalDB`
    - Windows Authentication
    - Database: `ClientNotifier`

If you want, I can give you a **ready-to-copy minimal Program.cs + connection string + DbContext** setup for LocalDB that works without needing any manual DB creation.

Do you want me to do that?

Yes — let’s make a minimal, working LocalDB setup so you can run migrations and connect from DataGrip without extra errors. Here’s a clean example:

Program.cs

```
csharpCopy codeusing ClientNotifier.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Configure EF Core with LocalDB
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(
        @"Server=(localdb)\
MSSQLLocalDB;Database=ClientNotifier;Trusted_Connection=True;MultipleA
ctiveResultSets=true"
    )
);

var app = builder.Build();

app.MapGet("/", () => "ClientNotifier API is running!");

app.Run();
```

DbContext (`AppDbContext.cs`)

```
csharpCopy codeusing Microsoft.EntityFrameworkCore;

namespace ClientNotifier.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options)
            : base(options)
        {
        }

        public DbSet<Person> People { get; set; }
        public DbSet<NamedayMapping> NamedayMappings { get; set; }
    }

    public class Person
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public string EGN { get; set; } // Bulgarian Personal Number
    }

    public class NamedayMapping
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public DateTime Nameday { get; set; }
    }
}
```

Steps to run migrations & database

1. Ensure EF Core tools are installed:

```
bashCopy codedotnet tool install --global dotnet-ef
```

2. Add migration:

```
bashCopy codedotnet ef migrations add InitialCreate -p
ClientNotifier.Data -s ClientNotifier.API
```

3. Apply migration (create database & tables):

```
bashCopy codedotnet ef database update -p ClientNotifier.Data -s
ClientNotifier.API
```

Connect from DataGrip

1. **Host:** `(localdb)\MSSQLLocalDB`
2. **Port:** leave empty
3. **Authentication:** Windows Authentication
4. **Database:** `ClientNotifier`

- Click **Test Connection** → should succeed.

- You’ll see `People`, `NamedayMappings`, and `__EFMigrationsHistory`.

This setup works **without manually creating the DB**. EF Core will create it automatically on `database update`.

If you want, I can also show a **minimal Docker version** next so your mom could run it without installing anything locally. Do you want me to do that?

Thanks for the screenshot — that confirms the problem. The error:

```
pgsqlCopy code[S1000][4060] Cannot open database "ClientNotifier" requested by the login.
```

means **the database doesn’t exist yet** on your LocalDB instance. DataGrip is trying to connect, but there’s no `ClientNotifier` database to open.

􏰀 Steps to fix

Option 1 — Let EF Core create the DB (recommended)

1. Make sure your `Program.cs` connection string points to LocalDB:

```
csharpCopy codeoptions.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=ClientNotifier;Trusted_Connection=True;MultipleActiveResultSets=true");
```

2. Run EF Core commands:

```
bashCopy codedotnet ef migrations add InitialCreate -p ClientNotifier.Data -s ClientNotifier.API
dotnet ef database update -p ClientNotifier.Data -s ClientNotifier.API
```

- `database update` will **create the `ClientNotifier` database** automatically.
- After that, DataGrip will be able to connect.

Option 2 — Manually create the database

If you want to manually create it before connecting:

1. Open **Command Prompt** or **PowerShell**.

2. Start LocalDB:

```
powershellCopy codesqllocaldb start MSSQLLocalDB
```

3. Open **SQL Server Management Studio (SSMS)** or DataGrip, but this time **connect to `(localdb)\MSSQLLocalDB` without specifying a database** (leave the database field empty).

4. Run:

```
sqlCopy codeCREATE DATABASE ClientNotifier;
```

5. Now reconnect in DataGrip with `Database = ClientNotifier`.

□ **Tip:** LocalDB databases don’t exist until you either let EF Core create them or manually create them. That’s why DataGrip shows “login failed for database” — your Windows login exists, but the DB doesn’t.

If you want, I can give you **the exact DataGrip config for LocalDB + a step-by-step EF Core command sequence** that guarantees it works without creating the DB manually. Do you want me to do that?