How Long?

*- A retirement countdown app built in Flutter*

I was having lunch with my dad, and he excitedly pulled out his phone to show me an email:

“Look! Only 5 years and 4 months until my retirement date!”

My response? **This app.**

Welcome to *How Long*,

An app designed for those who cannot wait to retire!

**Designing an idea**

This project started as a simple idea, something lightweight that would help me explore Flutter while building an app from scratch. What I didn’t expect was how much I’d enjoy documenting my experience while planning, testing and structuring the code.

Phase 1 – Foundation

**How it works:**

* User enters date of birth
* App returns how long is left in **years, months and days**

To collect the date of birth, a DatePicker is presented to the user.

To ease the complexity of calculating how long until retirement, I created a CountdownService class. This class sets a predefined constant value for retirement age, set at 67, and performs all the auxiliary transformations.

At this stage, the design was left very simple. Just enough to ensure the interaction worked and that the user could pick a date and see the predicted time displayed correctly.

What mattered most was understanding how the different layers of the app—such as the UI components, state providers, and services—would communicate with each other.

* Separation of concerns — UI and logic were kept apart from the start
* Immediate user feedback — input triggers update, shown immediately

Phase 2 – State Management

With the foundation working, it was time to take a more technical step.

I started thinking:

What if the user leaves the screen?

How can we keep the app reactive and structured?

I realised the **date of birth** needed to be stored *outside* the screen and handled separately from the UI.

**The aim:**

* Keep DOB and countdown logic external to the screen
* Trigger the countdown automatically when the DOB changes
* Let the UI just focus on displaying the result

To achieve this, I introduced the provider package and created a new file retirement\_provider.dart. This file manages both the dobProvider (stores the user’s selected DOB) and the countdownProvider (computes and updates the countdown based on the DOB).

This decision also simplified the HomeScreen itself. It no longer had to calculate or update anything – it simply read and displayed data based on the current state.

Phase 3 – Persistence

This stage was exciting because it introduced real-world usability.

The goal:

**Let the app remember the date of birth after it’s closed, so the countdown remains visible next time.**

Here’s the plan:

1. Save the DOB whenever the user selects it
2. On launch, check if there’s a stored value
3. If there is, restore it into dobProvider, and let the countdown update automatically

To do this, I created a new file in a /storage folder: dob\_storage.dart.

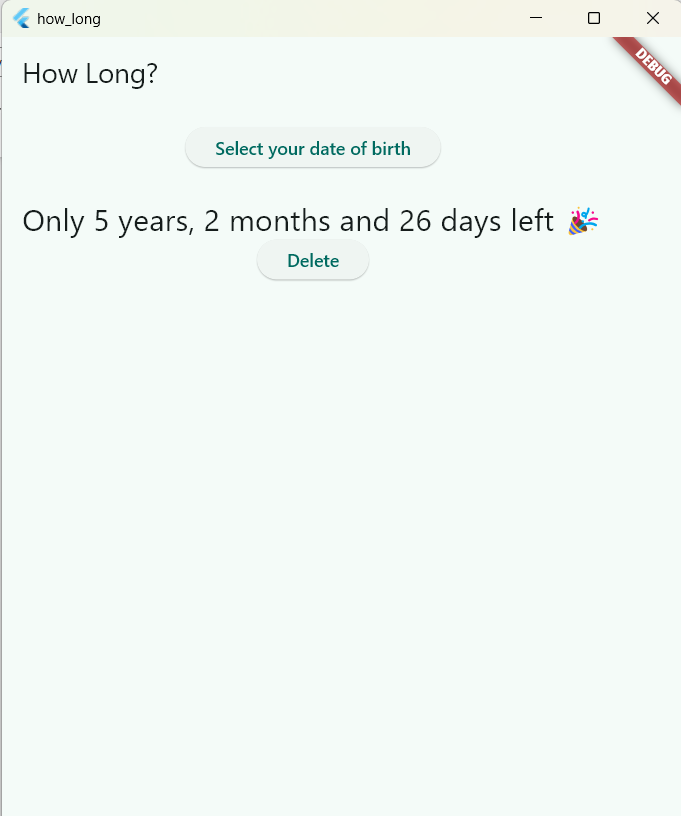
While writing it, I noticed that all methods were using the same call to SharedPreferences.getInstance(). To avoid repeating that line in every method, I decided to store it as a class-level instance, a small refactor that made the code a bit cleaner and more efficient.

Once persistence was working, I thought:

**“What if the user wants to clear their data?”**

So I added a **delete button** but only made it visible when a date was stored. It made more sense to keep the UI reactive and intuitive, only showing buttons when they were relevant.

At this point, things were working nicely, and I took a moment to breathe before heading into the final phase.



Phase 4 – UI Sparkle

At this stage, the app was functional but still looked a bit plain.

Although designing is something exciting, it goes a little beyond what I’m most confident with. After a couple of improvements, something clicked.

I realised the HomeScreen **was trying to do too much**:

* It handled DOB selection
* It showed the result
* It decided what buttons to show depending on state

It felt too conditional and messy. So, I paused and decided to refactor the flow entirely.

The app now has two separate screens:

1. **Setup Screen** — user selects their date of birth
2. **Countdown Screen** — displays the time left until retirement, with options to edit or delete the date

This change simplified the code and made the logic cleaner. Instead of asking the screen to “react” to whether or not there’s a value stored, it navigates to a new screen once the value exists.

It also made the user flow more intuitive. The app now mirrors how users think: first they set up, then they check their status.

While working on this, I also added an Edit button, something I’d originally overlooked. Instead of forcing users to delete and re-enter their DOB, they can now just update it.

And since the date selection logic was now needed in both screens, I refactored it into a small **utility class**, to avoid repeating code and keep things consistent.

Current Features

* 🎂 Date of birth input via DatePicker
* 📅 Countdown until retirement in years, months and days
* 💾 Persistent storage using SharedPreferences
* ♻️ State management with flutter\_riverpod
* ✏️ Ability to edit or delete stored date
* 📱 Clean navigation between screens based on state
* 💡 Thoughtful UI/UX — simple, purposeful, and reactive

Screens screenshot of a phone

AI-generated content may be incorrect.

Final thoughts

This was meant to be a small app, and it is. But what I’ve learned building it is anything but small. I planned it, built it from scratch, made mistakes, refactored and thought carefully about each decision. I now understand how Flutter apps are structured, how to work with state and how to separate logic in a way that scales.

I’m very happy with how this project came out. At first I wasn’t sure if I had the right skills or the greatest idea but by watching it evolve, I can’t stop thinking about other ways to make this app more useful.

And best of all, the idea came from a real-life moment with my dad. Thanks dad, only 5 more years!

Extending the idea

I thought I was done with the app, but after adding a background picture to the countdown it all felt more real. I implemented this by allowing each countdown to store an optional imagePath, which is then used to display a background image behind the countdown card. This small visual enhancement made the countdowns feel more tangible and personalised and it encouraged me to take the app a step further. So, I decided that I could extend the functionality a bit further…

**Since I have a working countdown system, why not support multiple countdowns?**

Update countdown model

What seemed straightforward at first turned out to be more complex than I anticipated. The entire logic had to be redrawn to support a now list of countdowns, as opposed to a single DateTime object.

There was also a need to introduce a title for each countdown and rethink the interaction.

State Management Evolution:

* From Simple Date to Countdown List

Originally, the app only needed to store and update a single date of birth. But once I decided to let users add their own countdowns, I had to rethink how data was stored and managed. Instead of a simple DateTime?, I introduced a full Countdown model with title, targetDate, and an optional imagePath. I also created a CountdownListNotifier class that extended StateNotifier, allowing me to store a list of countdowns that could be added, edited and deleted individually.

This was a major architectural shift. I had to adjust the provider logic to support multiple countdowns and ensure they remained reactive and persistent across launches. It was definitely a learning curve, but now the app feels much more robust and scalable.

Design Decisions:

* Balancing Simplicity and Flexibility

At first, I considered using a floating action button that opened a dialog with options to add, edit or delete countdowns. I also explored SpeedDial widgets and multi-action buttons. But after testing them, I realised that placing individual edit/delete buttons on each countdown card made the app far easier to use. The user can clearly see which countdown they're modifying and there's no extra dialog step.

This decision wasn’t just about simplicity, it was also about accessibility and user experience. Especially on mobile, reducing the number of taps and steps makes a big difference.

Reusable Widgets for UI Consistency

One technical improvement that made a big impact was extracting the countdown display card into a *reusable* CountdownCard widget. Initially, I was duplicating the same Card styling across retirement and custom countdowns. As soon as I made this a standalone widget, it became easier to keep the design consistent and avoid repetition.

This helped when implementing the retirement countdown as a special case: I could reuse the same card, just change the text and logic.

Conditional Navigation and Flow Control

The first version of the HomeScreen did everything. It handled input, updated the countdown and conditionally displayed UI elements. As the app evolved, this got messy and harder to maintain.

So I changed the logic:

* If a DOB exists, the app goes directly to the countdown screen.
* If not, the user is prompted to enter their DOB.

This decoupling made each screen responsible for a single purpose, which not only simplified the code but also improved the user experience.

Retirement Countdown: Still the Star of the Show

Even after generalising the app to support multiple countdowns, I didn’t want to lose the original idea, a retirement countdown.

To preserve it, I:

* Kept the DOB input as the first thing the user sees on first launch.
* Automatically generated the retirement countdown using the DOB.
* Displayed the retirement countdown styled like all others.
* Sorted the countdowns by end date

This way, the app still reflects the personal inspiration that started it all, while also becoming useful to a wider audience.

Challenges and Realisations

Some of the toughest parts weren’t the code, but the logic. Moving from a single value to a list of models broke a lot of things temporarily.

* I had to rethink how updates work.
* I rewrote how SharedPreferences handles storing multiple objects.
* I had to refactor existing functions to take a Countdown object instead of a date.

Sometimes I had to pause, sketch things out, or just take a walk. These moments of stepping away often led to unexpected clarity. It reminded me that progress isn’t always about typing, sometimes, it’s about thinking. But each roadblock taught me more about Flutter, state management and building real-world apps.

Final-final Thoughts

This app now feels like a small but complete product. It’s flexible, reactive, persistent and well-organised. I feel more confident in using Riverpod, managing user state and structuring apps with multiple data flows. Most importantly, I’ve started thinking like a developer: solving problems step by step and asking, "How will the user experience this?"

Best of all, my dad loves it!

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.