GAROONO FREE DOC · CAREER

Crack your Flutter, React Native and full stack interview

The questions that keep coming up, short answers you can say out loud, and a prep plan that fits next to a full time job

THE SHORT ANSWER

Most of these interviews ask the same 15 to 20 things per stack. Lifecycle, state, async, performance, how the framework draws to the screen, and how you'd build a small app.

Knowing the answer isn't enough. They want to hear **why, and what you'd trade off.** Learn the core questions, practice saying them out loud, build one small app the way a take-home would ask, and get three stories about your own work ready.

What's inside

01	How these interviews actually run, round by round
02	A 3 week prep plan for people with a day job
03	Flutter: the questions that keep coming up, with short answers
04	React Native: same, including the New Architecture
05	Full stack: HTTP, auth, databases, caching, JavaScript
06	System design, mobile and backend
07	Live coding and take-home tasks
08	How to talk about the things you built
09	Mistakes that cost offers, and a one page checklist

01 · THE ROUNDS

How these interviews actually run

Every company mixes it differently, but the pieces are the same. Small companies often do two or three of these. Bigger ones do most of them.

ROUND	WHAT HAPPENS	WHAT THEY'RE CHECKING
Recruiter call	A short call about your experience, notice period and salary	Can you explain your work simply, and do you fit the role
Quiz or screen	Multiple choice or quick questions on the framework	The basics, fast
Tech interview	Lifecycle, state management, async, performance, testing	Do you know why things work, not just what to type
Live coding	A small problem or a small UI, in an online editor, DartPad or Expo Snack	How you think, talk and test while you code
Take-home	Build a small app or API in a few hours or a few days	Structure, error states, tests, a clear README
System design	Design a feed, a chat, an offline app or a backend service	Tradeoffs, and whether you ask the right questions first
Project and HR	Your past work, a hard bug, a disagreement, a failure	Ownership, honesty, how you work with people

Read the interview reports people post on Glassdoor for the company you're applying to. For Flutter roles they keep mentioning key types, the stateful widget lifecycle, state management, the levels of testing and a small task in DartPad. For React Native, the bridge, hooks like `useMemo` and `useCallback`, Redux versus Context, and native modules. It's the same short list again and again. That's good news.

> **WHAT THE BIG QUESTION LISTS SKIP**
>
> Search for "flutter interview questions" and you get very long lists with one line answers. Useful, but they skip the parts that decide offers: a plan you can follow with a job, the take-home, mobile system design, and how to talk about your own work. This doc covers those too.

02 · THE PLAN

3 weeks, with a day job

About 45 minutes on weekdays and a 2 to 3 hour block on each weekend day. That's it. If you have a fourth week, use it for your second stack.

WHEN	WHAT TO DO
Week 1 Basics	Weekdays: 5 questions a day from your stack's section below. Read the answer, close the doc, say it out loud in your own words. Record yourself once and listen back. Add one easy coding problem a day. Weekend: the language core. Dart async and null safety, or JavaScript closures, the event loop and TypeScript basics
Week 2 Build	Build one small app the way a take-home would ask: a list screen from a real API, a detail screen, loading, error and empty states, a search box, a simple offline cache, and 3 to 5 tests. Write a short README. Weekend: one system design question each day, on paper, against a timer
Week 3 Practice	Two mock interviews, with a friend or a recording. Write your three project stories (page 16). Read the company's app or product, use it, and note two things you'd improve. Last two days: the checklist on page 17 and nothing new

A normal weekday, 45 minutes

MINUTES	DO THIS
0 to 15	Three or four questions, answered out loud with no notes
15 to 35	One coding problem, or one small feature on your practice app
35 to 45	Write down what you got wrong. Tomorrow starts with those

✎ FROM MY SIDE

I'm a senior software engineer with a full time job, and I've shipped 14 apps on the side that over 100,000 people use. Next to a job, that only works in small daily blocks, not by waiting for a free weekend. Interview prep works the same way. Short and daily beats long and once.

Flutter questions you'll get

Say each answer in under a minute. Then add one line on when you'd do it differently

1 Stateless vs Stateful widget?

A StatelessWidget only depends on what it's given and what it reads from above. A StatefulWidget has a separate `State` object that lives across rebuilds and can call `setState`. Widgets themselves are cheap and get recreated all the time. The State is what sticks around.

2 Walk me through the State lifecycle

`createState`, then `initState` once, then `didChangeDependencies`, then `build`. When the parent passes a new config, `didUpdateWidget` runs, then `build`. On removal, `deactivate`, and `dispose` if it's gone for good. Start things in initState, clean them up in dispose.

```
@override
void initState() {
  super.initState();
  _sub = widget.ticks.listen((t) => setState(() => _t = t));
}

@override
void dispose() {
  _sub.cancel(); // skip this and you leak, then call setState after dispose
  super.dispose();
}
```

3 Widget, Element and RenderObject. What's the difference?

Widgets are throwaway descriptions of the UI. Elements are the long lived instances in the tree that hold the State and decide what to reuse. RenderObjects do the real work of layout, painting and hit testing. That's why rebuilding widgets is cheap: Flutter compares the new widget with the old one and only updates what changed.

4 What happens in one frame?

Build, then layout, then paint, then composite, then the engine rasterizes it. In layout, **constraints go down, sizes go up, and the parent sets the position.** Say that line. It explains most "why is my widget the wrong size" bugs, like an unbounded height error from a ListView inside a Column.

5 What is BuildContext?

A handle to the widget's place in the tree. You use it to look things up above you: Theme, Navigator, a Bloc. Classic bug: using it after an `await` when the widget may already be gone.

```
await repo.save(note);
if (!context.mounted) return;
Navigator.of(context).pop();
```

6 How does data get down the tree without passing it everywhere?

`InheritedWidget`. Widgets below can read it, and they rebuild when it changes. Theme, MediaQuery, Provider and BlocProvider all sit on top of this idea.

7 When do you need keys?

When widgets of the same type move, get added or get removed in a list and they hold state. Without a key Flutter matches by position, so the state can end up on the wrong item. `ValueKey` for an id, `ObjectKey` for an object, `UniqueKey` to force a fresh widget, `GlobalKey` to reach a widget's state from outside (rarely, it costs more).

```
for (final todo in todos) TodoTile(key: ValueKey(todo.id), todo: todo)
```

8 Why use const constructors?

A `const` widget is created once at compile time and reused. When the parent rebuilds, Flutter sees it's the exact same instance and skips rebuilding that part. Free speed.

9 Which state management do you use, and why?

Name one you've shipped and say why, then show you know the others exist. Mine is **flutter_bloc**: UI sends events or calls methods, the bloc emits new states, the UI just draws the state. Easy to test and easy for a team to follow. `Cubit` is the lighter version with plain methods instead of events.

```
sealed class AuthEvent {}
final class LoginPressed extends AuthEvent {
  LoginPressed(this.email, this.password);
  final String email, password;
}

class AuthBloc extends Bloc<AuthEvent, AuthState> {
  AuthBloc(this._repo) : super(AuthInitial()) {
    on<LoginPressed>((event, emit) async {
      emit(AuthLoading());
      try {
        emit(AuthSuccess(await _repo.login(event.email, event.password)));
      } catch (e) {
        emit(AuthFailure('$e'));
      }
    });
  }
  final AuthRepository _repo;
}
```

10 BlocBuilder, BlocListener, BlocConsumer, BlocSelector?

- **BlocBuilder** rebuilds UI when the state changes
- **BlocListener** runs one-off actions like a snackbar or navigation, no rebuild
- **BlocConsumer** is both in one widget
- **BlocSelector** rebuilds only when one picked value changes

Use `context.read` to call a method, `context.watch` to rebuild on change.

11 Future vs Stream?

A Future gives you one value later, like an API call. A Stream gives you many values over time, like location updates or a chat. `await` a Future, `listen` to a Stream or use `StreamBuilder`, and always cancel subscriptions you start.

12 Dart is single threaded. How does async work then?

An event loop with two queues. Your sync code runs first, then every microtask, then the next event like a timer or I/O. Guess the output, it's a favorite:

```
print('1');
Future(() => print('4'));
scheduleMicrotask(() => print('3'));
print('2');
// 1 2 3 4
```

13 When do you use an isolate?

When work is heavy enough to drop frames: parsing a big JSON, image processing, crypto. Isolates don't share memory, they pass messages. `Isolate.run` is the simple way. Flutter's `compute` does the same on mobile, but on the web it runs on the main thread because isolates aren't supported there.

```
final items = await Isolate.run(() => parseItems(rawJson));
```

14 Your app janks. What do you do?

- Reproduce it in **profile mode** on a real device, never in debug
- Open DevTools, look at the frame chart, see if it's the UI thread or the raster thread
- UI side: too much rebuilt. Push state down, add const, use `ListView.builder` for long lists
- Raster side: expensive painting. Avoid `Opacity` and clipping on big areas, add `RepaintBoundary` around things that animate
- Measure again to prove the fix

15 What is Impeller?

Flutter's newer rendering engine. It prepares its shaders ahead of time, which removes the first-run shader jank Skia had. Per the Flutter docs it's the only engine on iOS and the default on Android API 29 and up since Flutter 3.27. The web still uses Skia.

16 How do you call native code?

Platform channels. A `MethodChannel` for request and reply, an `EventChannel` for a stream from native. Messages get encoded on the way in and out. For bigger APIs, **Pigeon** generates type safe code for both sides.

```
const _ch = MethodChannel('app/battery');
final level = await _ch.invokeMethod<int>('getLevel');

// Kotlin side
MethodChannel(flutterEngine.dartExecutor.binaryMessenger, "app/battery")
  .setMethodCallHandler { call, result ->
    if (call.method == "getLevel") result.success(readBattery())
    else result.notImplemented()
  }
```

17 Debug, profile and release builds?

Debug runs with JIT so hot reload works, with asserts on and slow speed. Profile is close to release speed but keeps some tools for measuring. Release is AOT compiled to native code, fastest and smallest. Never judge performance in debug.

18 Hot reload vs hot restart?

Hot reload puts new code into the running app and keeps the state. Hot restart starts the app over and loses the state. Changes to `initState`, globals or `main` usually need a restart.

19 How do you test a Flutter app?

Three levels. **Unit tests** for logic and blocs (the `bloc_test` package makes this short). **Widget tests** to pump one screen and tap things without a device. **Integration tests** to run the real app on a device or emulator. Mock the repository, not the HTTP client inside every test.

```
blocTest<CounterCubit, int>(
  'emits 1 after increment',
  build: CounterCubit.new,
  act: (c) => c.increment(),
  expect: () => [1],
);
```

20 How do you handle navigation and deep links?

For small apps, `Navigator.push` is fine. For deep links, web URLs and auth redirects, use a router like `go_router` where each screen has a path and a redirect can send logged out users to login.

21 What's new in Dart that you actually use?

Null safety (`?`, `!`, `late`, and why `!` is a smell), records, patterns and `sealed` classes. Sealed classes plus a `switch` make the compiler check you handled every bloc state:

```
return switch (state) {
  AuthLoading() => const CircularProgressIndicator(),
  AuthSuccess(:final user) => HomeView(user: user),
  AuthFailure(:final message) => ErrorView(message),
  AuthInitial() => const LoginForm(),
};
```

22 How do you keep a large app organized?

Split by feature, not by type. Inside each feature: a data layer (API and local storage behind a repository), the bloc, and the UI. The UI never talks to the API directly. Dependencies are passed in, so tests can swap them.

SENIOR LEVEL EXTRAS

Expect follow ups on app size (`--split-debug-info` and `--obfuscate`), flavors for dev and prod, secure storage for tokens, crash reporting, and how you'd move a big app from one state management to another without stopping feature work.

React Native questions you'll get

Versions move fast here. Check the release notes for the version the company uses

1 How does React Native draw the screen?

Your React code runs in a JavaScript engine, Hermes by default. It describes the UI, and React Native turns that into real native views, a real `UIView` on iOS and a real `View` on Android. It's not a web page inside the app. Flutter is different: it draws every pixel itself.

2 Old bridge vs the New Architecture?

The old bridge sent everything between JS and native as serialized messages, async, in batches. That could choke under load. The New Architecture replaces it:

- **JSI** lets JS hold a reference to a C++ object and call it directly, no serializing
- **Fabric** is the new renderer. It can render synchronously when needed and supports React's concurrent features
- **TurboModules** are native modules that load lazily, only when first used
- **Codegen** reads your typed JS spec and generates the native glue, so both sides agree on types

Per the React Native docs it became the default in 0.76, and the 0.82 release made it the only architecture.

3 What's Hermes and why does it matter?

A JavaScript engine built for React Native. Your JS is compiled to bytecode at build time, so the phone doesn't parse it at launch. The docs say that for many apps this means faster start, less memory and a smaller app than JavaScriptCore.

4 Which threads matter?

The **JS thread** runs your React code and logic. The **UI thread** draws native views and handles native scrolling. To hit 60 fps each frame has about 16.67 ms (from the performance docs). If the JS thread is busy, taps and JS driven animations lag even while native scrolling stays smooth.

5 Expo or bare React Native?

The official docs now recommend starting new apps with a framework like Expo. You can still add your own native code with config plugins and prebuild. Say you know what Expo handles for you (builds, updates, native setup) and when you'd need to go deeper.

6 useMemo, useCallback and React.memo?

`React.memo` skips re-rendering a component when its props didn't change. But a new function or object on every render is a "changed" prop, so `useCallback` keeps a function the same and `useMemo` keeps a computed value the same. Use them where you measured a problem, not on everything.

7 A long list is slow. Fix it

Use `FlatList` (or FlashList), never a ScrollView with a map. Then: a stable `keyExtractor`, a memoized row, a stable `renderItem`, `getItemLayout` if rows have a fixed height, and tune `initialNumToRender` and `windowSize`. Small images, not 4000 px ones.

```
const Row = memo(({ msg }: { msg: Msg }) => <MessageRow msg={msg} />);
const renderItem = useCallback(
  ({ item }: { item: Msg }) => <Row msg={item} />,
  [],
);

<FlatList
  data={messages}
  keyExtractor={(m) => m.id}
  renderItem={renderItem}
  getItemLayout={(_, i) => ({ length: 72, offset: 72 * i, index: i })}
  initialNumToRender={12}
  windowSize={7}
/>
```

8 What can go wrong with useEffect?

Missing dependencies give you stale values. Missing cleanup gives you leaks, and race conditions when the id changes fast and an old response lands last:

```
useEffect(() => {
  const ctrl = new AbortController();
  fetch(`/api/users/${id}`, { signal: ctrl.signal })
    .then((r) => r.json())
    .then(setUser)
    .catch((e) => { if (e.name !== 'AbortError') setError(e); });
  return () => ctrl.abort();
}, [id]);
```

9 Redux, Context, Zustand or React Query?

Split it in two. **Server data** (lists from an API) belongs in something like TanStack Query, which handles caching, refetching and loading states. **App state** (theme, cart, logged in user) goes in Context for small things, or Redux Toolkit or Zustand when it grows. Context re-renders every consumer when its value changes, so don't put fast changing data in one big Context.

10 How do animations stay smooth?

Keep them off the JS thread. With `Animated`, set `useNativeDriver: true` (works for transform and opacity). For gestures and complex animations, Reanimated runs small functions called worklets on the UI thread.

11 Navigation?

React Navigation with a native stack for real native transitions, or Expo Router, which builds routes from your file names. Know how deep links map to screens and how you'd guard screens that need login.

12 How do you write a native module today?

A Turbo Native Module. Write a typed spec in TypeScript, Codegen generates the native interface, then you implement it in Kotlin or Swift/Objective-C. Write one when there's no library for what you need, or when a native SDK must be called directly.

```
// specs/NativeBattery.ts
import type { TurboModule } from 'react-native';
import { TurboModuleRegistry } from 'react-native';

export interface Spec extends TurboModule {
  getLevel(): Promise<number>;
}

export default TurboModuleRegistry.getEnforcing<Spec>('NativeBattery');
```

13 Where do you store a token?

Not in AsyncStorage, it's plain unencrypted storage. Use the Keychain on iOS and the Keystore on Android, through something like `expo-secure-store` or `react-native-keychain`. Same answer in Flutter with `flutter_secure_storage`.

14 Over the air updates?

Tools like EAS Update can swap the JS bundle and assets without a store release. They can't change native code. Use them for bug fixes, not to sneak in features that change what the store reviewed.

15 How do you debug a crash or a slow screen?

React Native DevTools for JS, the React profiler for re-renders, Xcode and Android Studio for native crashes and memory. Always test speed in a **release build**. Dev mode is much slower and lies to you. Strip `console.log` in production builds too, the performance docs call it out as a big JS thread cost.

16 Platform specific code?

`Platform.OS` or `Platform.select` for small differences, and files named `.ios.tsx` and `.android.tsx` when a whole component differs.

17 How do you test it?

Jest for logic, React Native Testing Library to render a component and press things like a user would, and Detox or Maestro for end to end flows on a simulator.

> **THE ANSWER THAT SOUNDS SENIOR**
>
> If they ask "will the New Architecture make our app fast?", don't just say yes. Say you'd find the slow screen first, check if it's the JS thread, the UI thread, the network or memory, fix that one thing, and prove it on a release build. That's the difference they listen for.

Full stack questions you'll get

Mobile devs get these too, the moment a role says "and some backend"

1 What happens when you type a URL and press enter?

DNS finds the IP. The browser opens a connection and does the TLS handshake for HTTPS. It sends an HTTP request, the server (often behind a CDN and a load balancer) sends a response. The browser parses HTML, fetches CSS and JS, builds the page and paints it.

2 REST or GraphQL?

REST: one URL per resource, simple, and HTTP caching works out of the box. GraphQL: one endpoint, the client asks for exactly the fields it needs, handy when screens need different shapes of the same data. Costs: harder caching, and resolvers can cause N+1 queries unless you batch them, for example with DataLoader.

3 Which HTTP methods are idempotent? And 401 vs 403?

GET, PUT and DELETE are idempotent: sending them twice leaves the same result. POST isn't, so for payments or orders the client sends an idempotency key and the server ignores repeats. **401** means "I don't know who you are", **403** means "I know who you are, and you're not allowed".

4 Sessions or JWT?

- **Session:** the server stores the session and the client holds an id in a cookie. Easy to log someone out, the server just deletes it
- **JWT:** a signed token the server can check without a lookup. Signed, not encrypted, so anyone can read what's inside. Hard to revoke before it expires

The usual answer: short lived access token plus a refresh token that rotates. On the web, keep tokens in an `HttpOnly`, `Secure`, `SameSite` cookie so JavaScript can't read them. On mobile, secure storage.

5 How does "Sign in with Google" work?

OAuth 2 with OpenID Connect on top. The user signs in on Google, Google sends back a code, the app swaps it for tokens. Mobile apps and single page apps use **PKCE**, because they can't keep a client secret safe.

6 What is CORS?

A browser rule. A page on one origin can't read responses from another origin unless that server says it's allowed with `Access-Control-Allow-Origin`. For some requests the browser first sends an `OPTIONS` preflight. It doesn't protect your server, curl and mobile apps ignore it. Your auth does that.

7 Name the security basics

Parameterized queries against SQL injection. Escape output against XSS. CSRF protection for cookie auth. Check permissions on the server for every request, never trust the client. Hash passwords with bcrypt or argon2. Rate limit login. The OWASP Top 10 is the list they have in mind.

8 SQL or NoSQL?

SQL when data has clear relations and you need joins and transactions: users, orders, payments. NoSQL when the shape changes a lot, or you read whole documents by key at large scale. Most apps are fine on Postgres, and saying why sounds senior.

9 How does an index help, and what does it cost?

An index is a sorted structure (usually a B-tree) that lets the database jump to rows instead of scanning all of them. It costs disk space and slows down writes a little. A composite index works for queries that use its leftmost columns.

```
-- slow on a big table: reads every row
SELECT * FROM orders WHERE user_id = 42 ORDER BY created_at DESC LIMIT 20;

CREATE INDEX idx_orders_user_created ON orders (user_id, created_at DESC);
-- then run EXPLAIN ANALYZE on the query to prove it uses the index
```

10 What's the N+1 problem?

One query to get a list, then one more query for every item in it. 100 posts become 101 queries. Fix it with a join, an include, or by batching the ids into one query.

```
// N+1
const posts = await db.post.findMany();
for (const p of posts) {
  p.author = await db.user.findUnique({ where: { id: p.authorId } });
}

// one round trip
const posts = await db.post.findMany({ include: { author: true } });
```

11 What's a transaction?

Writes that all happen or none do (ACID: atomic, consistent, isolated, durable). Moving money takes from one account and adds to another. If the second write fails, the first rolls back.

12 Where can you cache, and what's the hard part?

- **Browser:** `Cache-Control` and `ETag` headers
- **CDN:** static files and public pages close to the user
- **App cache:** Redis in front of the database. Read the cache, on a miss read the DB and fill the cache

The hard part is knowing when cached data is stale. Give everything a TTL and clear the key when the data changes.

13 Offset or cursor pagination?

Offset (`?page=3`) is simple but slow on deep pages, and items shift when new rows come in. Cursor (`?after=last_id`) stays fast and stable, which is why feeds use it.

14 Your app gets 100 times more users. What changes?

Measure first to find what breaks. Then: stateless servers behind a load balancer, indexes and caching, a queue for slow work like emails, read replicas for heavy reads, and rate limits.

JavaScript and TypeScript core

These come up in React, React Native and Node interviews alike. Expect at least one "what does this print?"

15 Explain the event loop

Same idea as Dart. Sync code runs first, then all microtasks (promise callbacks), then the next task (timers, I/O). See MDN's execution model page.

```
console.log('A');
setTimeout(() => console.log('D'), 0);
Promise.resolve().then(() => console.log('C'));
console.log('B');
// A B C D
```

16 What's a closure? Then the classic loop

A function that remembers the variables from where it was created, even after that code finished. `var` is one shared variable for the whole loop, `let` gives each round its own.

```
for (var i = 0; i < 3; i++) setTimeout(() => console.log(i));
// 3 3 3     with let: 0 1 2
```

17 Quick fire round

- `===` compares without type conversion. Use it by default
- `null` is "empty on purpose", `undefined` is "never set"
- `Promise.all` fails on the first error, `Promise.allSettled` waits for all
- Arrow functions don't have their own `this`
- In TypeScript, `unknown` forces you to check the type before using it, `any` turns checking off
- `type` vs `interface`: mostly the same for objects. Interfaces can be extended and merged, types can do unions

18 Why not use the array index as a React key?

Keys tell React which item is which between renders. If you insert or reorder items, the index of each item changes, so React mixes up their state and inputs. Use a stable id. Same idea as Flutter keys.

19 Write a debounce

Asked in live coding all the time, usually for a search box.

```
function debounce<A extends unknown[]>(fn: (...args: A) => void, ms: number) {
  let timer: ReturnType<typeof setTimeout> | undefined;
  return (...args: A) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), ms);
  };
}
```

06 · SYSTEM DESIGN

Use the same 5 steps every time

Nobody expects a perfect design. They want to see you ask good questions, make choices and say what each choice costs. In a 45 minute round, keep moving through these steps:

STEP	WHAT YOU DO
1. Ask	Who uses it, the main features, how many users, does it need to work offline, real time or not
2. Data and API	The main objects and the few endpoints you need, with pagination
3. Boxes	Draw the parts: app, API, database, cache, queue, storage for files
4. Go deep	Pick the hardest part and spend most of your time there
5. Break it	What happens when the network drops, a server dies, or traffic jumps

Mobile version: "Design an offline first notes app"

- **The local database is the source of truth.** The UI reads only from it (SQLite, Drift, Isar, WatermelonDB). The network fills it
- **Writes go into a queue first.** Save locally, show it right away, then a sync job sends the queue when the device is online
- **Conflicts:** last write wins is fine for simple fields. Say so, and say when you'd need something smarter, like two people editing one note
- **Ids made on the device** (UUIDs), so new notes don't wait for the server
- **Lists:** cursor pagination, cache images on disk, load thumbnails, not full images
- **The phone itself:** the app can be killed in the background, the network is slow and battery matters. Sync in batches, retry with backoff

✎ PRACTICE SET

Pick one for each weekend day in week 2, and the rest after: a news feed, a chat app, an offline notes app, a URL shortener (backend), image upload with progress. Same 5 steps every time, timer on.

07 · CODING ROUNDS

Live coding: think out loud

Silence fails more people here than wrong answers. They can only give credit for thinking they can hear.

- **Repeat the problem** in your words and ask about edge cases: empty input, duplicates, very large input
- **Say the simple solution first**, even if it's slow. Then improve it
- **Run one example by hand** before you say you're done
- **Know the usual tools:** a hash map for "have I seen this", two pointers, a sliding window, sorting first, BFS for shortest path in a grid

For mobile roles the live task is often a small screen instead of a puzzle: fetch a list from an API, show it, add search. Have this ready in your head: **loading, error, empty and data states**, a debounce on the search box, and pull to refresh.

Take-home: what actually gets scored

AREA	DO THIS
README	How to run it, what you skipped and why, what you'd do with more time
Structure	Feature folders, a repository between UI and API, state outside the widgets
All the states	Loading, error with retry, empty, no internet. Most people skip these
Tests	A few that matter: the bloc or reducer, one widget or component test
Git history	Small commits with clear messages, not one big "done" commit
Time	Stay inside the time they gave. Over-building looks like poor scoping

> **REUSE YOUR WEEK 2 APP**
>
> The week 2 app is your take-home starter: same folders, same states, same test setup. When the real task arrives, you start from something that works.

Talk about what you built like an owner

"Tell me about a project" is where people either stand out or ramble. Use the STAR order: Situation, Task, Action, Result. Keep it to about two minutes, and talk about what **you** did, not "we".

FILL IN TEMPLATE

```
Situation: [The app or product] for [who uses it].
           The problem was [what was going wrong].
Task:      I owned [your part], and the goal was [what success meant].
Action:    I [2 or 3 concrete things you did and why you chose them].
           I didn't do [the other option] because [the tradeoff].
Result:    [What changed, with a real number if you have one].
           Next time I'd [one honest thing you'd do differently].
```

SAMPLE (A MADE UP APP, SWAP IN YOUR OWN)

```
Situation: A habit tracker. People said it lost their check-ins on the subway.
Task:      I owned the data layer, and the goal was no lost check-ins offline.
Action:    I made the local database the source of truth and put every write in a
           queue that syncs when the phone is back online. I kept last write wins
           because a check-in is only ever edited by one person.
Result:    Reviews about lost data stopped.
           Next time I'd add sync tests from day one, I added them late.
```

Get three of these ready: one technical win, one bug or failure you owned, one time you disagreed with someone and how it ended. Only use numbers you can back up if they ask.

✎ SIDE APPS ARE THE BEST STORIES

I build Flutter apps next to my job, and a side app gives you things most people can't talk about: you owned everything. In my apps that meant App Store rejections I had to fix (like adding in-app account deletion and the right subscription terms link on a paywall), a purchase library version that crashed on buy and had to be pinned, and analytics that quietly stopped sending data. Interviewers love these because they're real problems with a clear fix. If you have even one app in a store, lead with it.

09 · LAST MILE

Mistakes that cost offers

- **Buzzwords with nothing behind them.** If you say "clean architecture", be ready to draw it
- **"I always use X".** Say when you wouldn't
- **Not knowing your own resume.** Every line on it is a question waiting to happen
- **Guessing.** "I haven't used that, but here's how I'd figure it out" beats a confident wrong answer
- **Talking badly about a past team or company**
- **No questions at the end.** It reads as no interest

Good questions to ask them: What does the first month look like? How do you ship, and how often? What's the hardest problem in the app right now? How do you test before release?

The night before

- [ ] You can explain the lifecycle, state management and async of your stack out loud
- [ ] You can explain how your framework gets pixels on the screen
- [ ] You've written the event loop "what does this print" from memory
- [ ] Your practice app runs, with loading, error and empty states
- [ ] You've done the 5 step system design at least three times
- [ ] Three STAR stories, about two minutes each
- [ ] You used the company's app and have two ideas for it
- [ ] Three questions ready to ask them
- [ ] Editor, mic and camera tested. A glass of water nearby

Want more like this?

I'm a senior software engineer who builds and ships apps next to a full time job. Over 100,000 people use my apps, and I share what actually works, the boring parts included. Follow along wherever you hang out:

Instagram
3,500+ builders
@garoonoo →

LinkedIn
8,000+ followers
Connect →

Medium
250+ readers
Read the long versions →

See all my apps and free docs at garoono.in →

Go deeper

FLUTTER	Flutter architectural overview · Performance best practices · Isolates · Testing · bloc library
REACT NATIVE	New Architecture · Performance · Optimizing FlatList · Turbo Native Modules
WEB	MDN: HTTP caching · Cookies · CORS · OWASP Top 10
PRACTICE	roadmap.sh full stack questions · Tech Interview Handbook: behavioral rounds
MORE DOCS	garoono.in/docs The rest of the free and Pro playbooks