COFFEETIME

Step-by-Step Build Guide

A Father-Son Coding Project with Claude Code

30 Small Steps from Zero to Working App

How to Use This Guide

This guide breaks down building CoffeeTime into 30 small, manageable steps. Each step has:

- A prompt to copy-paste into Claude Code
- A check-in to verify things work before moving on
- Teaching moments explaining what just happened
- Discussion questions to explore concepts together

The Workflow

1. Read the step together

2. Copy the green prompt box into Claude Code

3. Watch Claude Code work (it will ask permission for commands)

4. Complete the check-in together

5. Discuss the teaching moment

6. Move to the next step

Before You Start

- Install Claude Code on your computer
- Have a Google account ready
- Create a folder for your project (e.g., ~/projects/coffeetime)
- Set aside about 4-6 hours total (can be split across multiple sessions)

Phase 1: Project Setup

Getting all our tools ready and creating the foundation

Step 1	**Initialize the Project**	10 min

First, we need to create a new Next.js project. This is the framework that will power our app.

```
Create a new Next.js 14 project called 'coffeetime' with:
- TypeScript enabled
- Tailwind CSS enabled
- App Router (not pages router)
- ESLint enabled
- src/ directory disabled (use app/ at root)

After creating, just confirm it worked. Don't add anything else yet.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ A new folder called 'coffeetime' was created
- □ You can see files like package.json, app/page.tsx, tailwind.config.ts
- □ Claude Code didn't report any errors

💡 **TEACHING MOMENT: What is Next.js?**

Next.js is a framework built on top of React (a library for building user interfaces). It handles a lot of complicated stuff for us: routing between pages, optimizing performance, and making our app work well on phones. Think of it like getting a pre-built car frame instead of welding metal yourself.

🗣️ *DISCUSS: What other kinds of apps have you used that might be built with similar technology?*

Step 2	**Run the App for the First Time**	5 min

Let's start the development server and see our app running in a browser!

```
Start the Next.js development server and tell me what URL to open
in my browser to see the app.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ The terminal shows 'Ready' with a URL (usually http://localhost:3000)
- □ Opening that URL in Chrome shows a Next.js welcome page

□ The page has some default Next.js content

💡 **TEACHING MOMENT: What is a Development Server?**

When building websites, we run a 'development server' on our own computer. It's like having a private preview of our app that only we can see. Every time we change code, the server automatically updates so we can see changes instantly. When we're done, we'll 'deploy' to a real server so anyone can access it.

🗣️ *DISCUSS: Why do you think developers test on their own computer first instead of putting changes directly on the internet?*

Step 3	**Install shadcn/ui**	10 min

shadcn/ui gives us beautiful, pre-built components (buttons, forms, cards) so we don't have to design everything from scratch.

```
Initialize shadcn/ui in this project with:
- Default style
- Slate as the base color (we'll customize later)
- CSS variables enabled

Then install these components: button, input, card, dialog,
dropdown-menu, form, label, select, textarea, tabs, toast

Don't modify any other files yet.
```

✓ **CHECK-IN: Before moving on, verify:**

□ A 'components/ui' folder was created with multiple files
□ You can see button.tsx, card.tsx, input.tsx, etc.
□ No error messages appeared

💡 **TEACHING MOMENT: What are UI Components?**

A component is a reusable piece of interface—like a button or a card. Instead of writing the code for a button every time we need one, we write it once and reuse it everywhere. shadcn/ui gives us professionally designed components that look great and work well on all devices.

🗣️ *DISCUSS: Can you think of UI elements that appear over and over in different apps? (Hint: think about buttons, menus, forms...)*

Step 4	**Set Up the Dark Theme**	10 min

Our app will have a dark theme with warm coffee colors. Let's configure our color palette.

```
Update the Tailwind and CSS configuration for a dark-mode-first theme
with these custom colors:

Background: #0A0A0B
Surface/Cards: #141416
Surface Hover: #1C1C1F
Border: #2A2A2E
Text Primary: #FAFAFA
Text Secondary: #A1A1AA
Accent/Primary: #D4A574 (warm brown)
Accent Hover: #C4956A

Update globals.css and tailwind.config.ts appropriately.
Make dark mode the default.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ globals.css has new color variables
- □ tailwind.config.ts has custom color definitions
- □ The app in your browser now has a dark background

💡 **TEACHING MOMENT: CSS Variables and Theming**

Colors in our app are stored as 'variables'—named values we can reference anywhere. Instead of typing '#D4A574' every time we want our accent color, we just say 'accent'. If we ever want to change the color, we change it in one place and it updates everywhere. This is called 'theming'.

🗣️ *DISCUSS: Why did we choose warm brown (#D4A574) as our accent color for a coffee app?*

Step 5	**Create the Basic Layout Structure**	15 min

Every screen in our app will share the same basic structure: a header at the top and navigation at the bottom. Let's create that skeleton.

```
Create the basic app layout structure:

1. Update app/layout.tsx with:
   - Dark background color
   - Inter font from Google Fonts
   - Proper meta tags for PWA

2. Create a components/layout/BottomNav.tsx component with:
```

```
   - Fixed bottom navigation bar
   - 4 nav items: Log (home icon), History (clock icon),
     Analytics (chart icon), Settings (gear icon)
   - Use Lucide React icons
   - Highlight active route

3. Create app/(main)/layout.tsx that wraps pages with BottomNav

4. Create placeholder pages:
   - app/(main)/page.tsx (Log - shows 'Log Your Brew')
   - app/(main)/history/page.tsx (shows 'Brew History')
   - app/(main)/analytics/page.tsx (shows 'Analytics')
   - app/(main)/settings/page.tsx (shows 'Settings')

Each page should just show its title for now.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ The browser shows a dark page with 'Log Your Brew' text
- □ A navigation bar appears at the bottom with 4 icons
- □ Clicking each icon navigates to a different page
- □ The active page's icon is highlighted

💡 **TEACHING MOMENT: Routing and Navigation**

In Next.js, the folder structure IS the routing structure. A file at app/(main)/history/page.tsx automatically becomes the /history URL. The (main) folder with parentheses is a 'route group'—it lets us share a layout (like our bottom nav) without adding to the URL path.

🗣️ *DISCUSS: Open the browser's developer tools (F12) and click the 'Network' tab. What happens when you navigate between pages? Does the whole page reload?*

Phase 2: Firebase Setup

Connecting to our database and authentication

Step 6	**Create a Firebase Project**	15 min

This step happens in your web browser, not Claude Code. You'll create a Firebase project on Google's website.

📋 **MANUAL STEPS (do these in your browser):**

1. Go to console.firebase.google.com
2. Click 'Create a project' (or 'Add project')
3. Name it 'coffeetime' (or similar)
4. Disable Google Analytics (not needed)
5. Click 'Create project' and wait
6. Click the Web icon (</>) to add a web app
7. Nickname it 'coffeetime-web'
8. Check 'Also set up Firebase Hosting'
9. Copy the firebaseConfig object shown—you'll need it!

✓ **CHECK-IN: Before moving on, verify:**

- □ You have a Firebase project created
- □ You copied the config object (has apiKey, authDomain, projectId, etc.)
- □ The Firebase console shows your new project

💡 **TEACHING MOMENT: What is Firebase?**

Firebase is Google's 'backend as a service.' Instead of setting up our own servers and databases (which is complicated and expensive), Firebase provides them ready to use. It handles storing data, user accounts, file uploads, and hosting—all for free at our scale.

🗣️ *DISCUSS: What are some things that need to happen on a server that can't happen just in your browser?*

Step 7	**Enable Firebase Services**	10 min

Now we need to turn on the specific Firebase features we'll use.

📋 **MANUAL STEPS (in Firebase Console):**

Enable Authentication:

1. Click 'Authentication' in left sidebar

2. Click 'Get started'
3. Click 'Google' under Sign-in providers
4. Toggle 'Enable', add your email, click Save

Enable Firestore:

1. Click 'Firestore Database' in left sidebar
2. Click 'Create database'
3. Choose 'Start in test mode' (we'll secure it later)
4. Select a location close to you, click Enable

✓ CHECK-IN: Before moving on, verify:

□ Authentication shows Google as an enabled provider
□ Firestore Database shows an empty database
□ No error messages appeared

💡 **TEACHING MOMENT: Authentication vs Database**

Authentication answers 'WHO is this user?' Database answers 'WHAT data do they have?' They work together: when you sign in, Firebase gives you a unique user ID. We use that ID to organize your data in the database so you only see your own brews, not everyone else's.

Step 8	Add Firebase to Our Project	10 min

Now we connect our app code to Firebase using the config you copied earlier.

```
Set up Firebase in our project:

1. Install firebase package

2. Create lib/firebase/config.ts with the Firebase initialization.
   Use environment variables for the config values.

3. Create a .env.local file with placeholders for:
   NEXT_PUBLIC_FIREBASE_API_KEY
   NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN
   NEXT_PUBLIC_FIREBASE_PROJECT_ID
   NEXT_PUBLIC_FIREBASE_STORAGE_BUCKET
   NEXT_PUBLIC_FIREBASE_MESSAGING_SENDER_ID
   NEXT_PUBLIC_FIREBASE_APP_ID

4. Create a .env.example with the same keys (no values)

Tell me what values I need to fill in from my Firebase config.
```

📋 **MANUAL STEP: Copy your Firebase config values into .env.local**

✓ **CHECK-IN: Before moving on, verify:**

- □ lib/firebase/config.ts exists
- □ .env.local has your Firebase values filled in
- □ .env.example exists (without real values)
- □ No errors when saving files

💡 **TEACHING MOMENT: Environment Variables**

Environment variables store sensitive info (like API keys) outside our code. Why? Because code often gets shared (like on GitHub), but we don't want to share our secret keys! The .env.local file stays on your computer, while .env.example shows others what variables they need without revealing your actual values.

🗣️ *DISCUSS: What could happen if someone got access to your Firebase API key?*

Step 9	Create the Auth Context	15 min

We need a way to know if a user is signed in throughout our whole app. React Context lets us share this information everywhere.

```
Create the authentication system:

1. Create lib/firebase/auth.ts with:
   - signInWithGoogle function (using popup)
   - signOutUser function
   - onAuthStateChange listener

2. Create lib/context/AuthContext.tsx with:
   - AuthProvider component
   - useAuth hook
   - Track: user object, loading state, error state
   - Listen to auth state changes on mount

3. Wrap the app with AuthProvider in app/layout.tsx

Don't create any UI yet, just the logic.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ lib/firebase/auth.ts has signInWithGoogle and signOutUser functions
- □ lib/context/AuthContext.tsx exports AuthProvider and useAuth
- □ app/layout.tsx wraps children with AuthProvider
- □ No console errors in the browser

💡 **TEACHING MOMENT: React Context**

Normally, if you want to pass data from a parent component to a deeply nested child, you'd have to pass it through every component in between (called 'prop drilling'). Context creates a 'shortcut'—any component can access shared data directly. It's like a bulletin board that any component can read from.

🗣️ *DISCUSS: What other pieces of information might we want to share across our entire app? (Hint: think about settings, theme...)*

Step 10	**Build the Login Screen**	15 min

Now let's create a beautiful login screen that appears when users aren't signed in.

```
Create the login experience:

1. Create app/(auth)/login/page.tsx with:
   - App logo/title 'CoffeeTime' with coffee emoji ☕
   - Tagline: 'Track your perfect brew'
   - 'Sign in with Google' button using shadcn Button
   - Clean, centered layout
   - Dark theme styling

2. Create a components/auth/AuthGuard.tsx that:
   - Shows loading spinner while checking auth
   - Redirects to /login if not authenticated
   - Shows children if authenticated

3. Wrap the (main) layout with AuthGuard

4. On the login page, redirect to / if already signed in

Make it look professional and polished.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ Opening the app shows the login screen (you're not signed in yet)
- □ The login screen looks clean with the CoffeeTime title
- □ There's a 'Sign in with Google' button
- □ The dark theme is applied

💡 **TEACHING MOMENT: Protected Routes**

A 'protected route' is a page that requires authentication. Our AuthGuard component acts like a bouncer at a club—it checks if you're authenticated before letting you in. If not, it redirects you to the login page. This pattern keeps private data safe.

🗣️ *DISCUSS: What would happen if we didn't have AuthGuard and someone went directly to /history without signing in?*

Step 11	**Test Sign In**	10 min

Let's test that signing in actually works!

```
Add a temporary sign-out button to the Settings page so we can
test the full sign-in/sign-out flow. The button should:
- Call signOutUser when clicked
```

```
- Show the user's email when signed in
- Use our accent color
```

📋 **TEST IT: Click 'Sign in with Google' and complete the flow**

✓ **CHECK-IN: Before moving on, verify:**

- □ Clicking 'Sign in with Google' opens a popup
- □ After signing in, you're redirected to the main app
- □ The Settings page shows your email
- □ Clicking 'Sign out' returns you to the login screen

💡 **TEACHING MOMENT: OAuth and Sign-in Popups**

When you 'Sign in with Google,' you're using OAuth—a secure way to log in without giving our app your password. Google handles the authentication, then tells our app 'yes, this person is who they say they are' along with basic profile info. This is why you see Google's login page, not ours.

🗣️ *DISCUSS: Why is it safer to sign in with Google instead of creating a new username/password for every app?*

Phase 3: Data Layer

Creating the structure for storing our brewing data

Step 12	**Define TypeScript Types**	10 min

Before we store any data, we define exactly what shape that data will have. TypeScript helps us catch mistakes.

```
Create lib/types/index.ts with TypeScript interfaces for our data
model:

Coffee: id, name (required), roaster, origin, region, farm, process,
        variety, elevation, roastLevel, roastDate, flavorNotes
(array),
        photoURL, notes, isActive, createdAt, updatedAt

Grinder: id, name (required), brand, model, type, burrType, burrSize,
         settingsMin, settingsMax, settingsType, notes, isActive,
         createdAt, updatedAt

Brewer: id, name (required), brand, type, material, capacityMl,
        filterType, notes, isActive, createdAt, updatedAt

CoffeeTime: id, timestamp (required), coffeeId, coffeeName,
grinderId,
         grinderName, brewerId, brewerName, doseGrams, waterGrams,
         ratio, waterTempF, waterTempC, grindSetting,
bloomTimeSeconds,
         bloomWaterGrams, totalTimeSeconds, techniqueNotes,
tastingNotes,
         rating (1-10), photoURL, rawVoiceInput, createdAt, updatedAt

All fields except those marked 'required' should be optional
(nullable).
Use Timestamp type from firebase for date fields.
```

✓ CHECK-IN: Before moving on, verify:

- □ lib/types/index.ts exists with Coffee, Grinder, Brewer, CoffeeTime interfaces
- □ Required fields don't have '?' after them
- □ Optional fields have '?' after them
- □ Date fields use Timestamp type

💡 TEACHING MOMENT: TypeScript Interfaces

An interface is like a contract that says 'data of this type MUST have these fields.' When you try to use the data incorrectly, TypeScript warns you immediately—before you even run the code. It's like spell-check for your data structure.

🗣️ *DISCUSS: Why did we make most fields optional? What happens if we're at a coffee shop and don't know the water temperature?*

Step 13	**Create Firestore Service - Coffees**	15 min

Now we create functions to read and write coffee data to Firestore. We'll start with just Coffees.

```
Create lib/services/coffeeService.ts with these functions:

- getCoffees(userId): Get all active coffees for a user
- getCoffee(userId, coffeeId): Get a single coffee
- addCoffee(userId, coffee): Add a new coffee, return the new ID
- updateCoffee(userId, coffeeId, updates): Update a coffee
- archiveCoffee(userId, coffeeId): Set isActive to false
- deleteCoffee(userId, coffeeId): Permanently delete

Collection path: users/{userId}/coffees/{coffeeId}

Include proper error handling and TypeScript types.
Use serverTimestamp() for createdAt/updatedAt.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ lib/services/coffeeService.ts exists
- □ All 6 functions are defined
- □ Functions use the correct Firestore collection path
- □ Error handling is included

💡 **TEACHING MOMENT: Firestore Collections and Documents**

Firestore organizes data in 'collections' (like folders) containing 'documents' (like files). Each document has an ID and contains fields with values. Our structure is: users collection → user document → coffees collection → coffee documents. This keeps each user's data separate.

🗣️ *DISCUSS: Why do we 'archive' coffees instead of deleting them? What would happen to old brew logs that reference a deleted coffee?*

Step 14	**Create Remaining Services**	15 min

Let's create the same service pattern for grinders, brewers, and brew logs.

```
Create the remaining service files following the same pattern as
coffeeService:

1. lib/services/grinderService.ts
   - Same CRUD functions as coffee
   - Path: users/{userId}/grinders/{grinderId}

2. lib/services/brewerService.ts
   - Same CRUD functions as coffee
   - Path: users/{userId}/brewers/{brewerId}

3. lib/services/brewLogService.ts
   - getCoffeeTimes(userId, options): Support filtering by date
range,
     coffeeId, brewerId, grinderId, minRating. Support pagination.
   - getCoffeeTime(userId, brewLogId): Get single brew
   - addCoffeeTime(userId, brewLog): Add new brew
   - updateCoffeeTime(userId, brewLogId, updates): Update brew
   - deleteCoffeeTime(userId, brewLogId): Delete brew
   - Path: users/{userId}/brewLogs/{brewLogId}

4. Create lib/services/index.ts that exports everything
```

✓ **CHECK-IN: Before moving on, verify:**

- □ All 4 service files exist in lib/services/
- □ lib/services/index.ts exports all services
- □ brewLogService has filtering options
- □ All services follow the same pattern

💡 **TEACHING MOMENT: Service Layer Pattern**

We put all database operations in 'service' files instead of directly in our components. This separation has benefits: (1) Components stay simple, (2) We can change how data is stored without changing components, (3) We can reuse the same functions across multiple components.

🗣️ *DISCUSS: What if we later wanted to switch from Firebase to a different database? How would the service layer pattern help?*

Step 15	**Create React Hooks for Data**	15 min

React hooks let us easily use our services in components with loading states and error handling.

```
Create custom React hooks for each data type:

1. lib/hooks/useCoffees.ts:
   - useCoffees(): Returns { coffees, loading, error, refetch }
   - useCoffee(id): Returns single coffee
   - Use real-time listener (onSnapshot) for live updates

2. lib/hooks/useGrinders.ts - same pattern

3. lib/hooks/useBrewers.ts - same pattern

4. lib/hooks/useCoffeeTimes.ts:
   - useCoffeeTimes(options): With filtering support
   - useCoffeeTime(id): Single brew log
   - Support pagination with useInfiniteCoffeeTimes if possible

All hooks should:
- Get userId from useAuth()
- Return loading and error states
- Clean up listeners on unmount
```

✓ **CHECK-IN: Before moving on, verify:**

- □ All 4 hook files exist in lib/hooks/
- □ Hooks use onSnapshot for real-time updates
- □ Hooks return loading and error states
- □ Hooks clean up listeners (return unsubscribe in useEffect)

💡 **TEACHING MOMENT: Custom Hooks**

Custom hooks let us extract and reuse stateful logic. Instead of copying the same 'load data, handle loading, handle errors' code into every component, we write it once in a hook and use it everywhere. The 'use' prefix is a React convention that indicates it's a hook.

🗣️ *DISCUSS: What does 'real-time updates' mean? What happens if you open the app on two devices and log a brew on one?*

Phase 4: Equipment Management

Building the screens to manage coffees, grinders, and brewers

Step 16	Build the Settings Page Structure	10 min

The Settings page will have sections for managing equipment and app preferences.

```
Update the Settings page (app/(main)/settings/page.tsx) with:

1. User profile section at top:
   - User's profile photo (from Google)
   - User's name and email
   - Sign out button

2. Equipment section with 3 cards:
   - 'My Coffees' - shows count, links to coffee list
   - 'My Grinders' - shows count, links to grinder list
   - 'My Brewers' - shows count, links to brewer list

3. Each card should show:
   - Icon
   - Title
   - Count of items
   - Arrow indicating it's clickable

Don't create the sub-pages yet, just the Settings page.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ Settings page shows your Google profile photo and name
- □ Three equipment cards are visible
- □ Sign out button is present
- □ The dark theme looks good

💡 **TEACHING MOMENT: Component Composition**

Notice how the Settings page is made up of smaller pieces: a profile section, equipment cards, a sign-out button. This is composition—building complex UIs from simple, reusable parts. If we need a similar card elsewhere, we can reuse the same component.

🗣️ *DISCUSS: Looking at the Settings page, what smaller components could we extract that might be useful elsewhere?*

Step 17	Create the Coffee List Screen	15 min

Let's build the screen that shows all your coffees.

```
Create app/(main)/settings/coffees/page.tsx with:

1. Header with back button and 'My Coffees' title

2. 'Add Coffee' button (floating action button style)

3. List of coffees using the useCoffees hook:
   - Show loading spinner while loading
   - Show empty state if no coffees
   - Each coffee card shows: name, roaster, origin (if exists)
   - Cards are tappable (we'll add navigation later)

4. Handle the empty state nicely:
   - Coffee cup illustration or emoji
   - 'No coffees yet' message
   - 'Add your first coffee' button

Don't create the Add/Edit form yet.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ Navigating to Settings → My Coffees shows the coffee list page
- □ If no coffees, an empty state appears
- □ An 'Add Coffee' button is visible
- □ Loading state shows briefly

💡 **TEACHING MOMENT: Conditional Rendering**

Our page shows different things based on the state: a spinner when loading, an empty state when there's no data, and a list when there is data. This is conditional rendering—using JavaScript conditions to decide what to display. It makes the UI respond appropriately to every situation.

🗣️ *DISCUSS: What would the user experience be like if we didn't show a loading spinner? Why is feedback important?*

Step 18	Create the Coffee Form	20 min

Now we build the form to add and edit coffees. This is a full-screen modal.

```
Create components/equipment/CoffeeForm.tsx as a full-screen modal
form:

1. Props: isOpen, onClose, existingCoffee (optional for edit mode)

2. Form fields (all optional except name):
   - Name (text input, required)
   - Roaster (text input)
   - Origin (text input)
   - Region (text input)
   - Process (select: washed, natural, honey, anaerobic, other)
   - Variety (text input)
   - Roast Level (select: light, light-medium, medium, medium-dark,
dark)
   - Roast Date (date picker)
   - Flavor Notes (tag input - user can add multiple)
   - Notes (textarea)

3. Form behavior:
   - Pre-fill fields if editing existing coffee
   - Show 'Add Coffee' or 'Edit Coffee' in header based on mode
   - Save button calls addCoffee or updateCoffee service
   - Show loading state while saving
   - Close and refresh list on success
   - Show error toast on failure

4. Add the form to the coffees page:
   - 'Add' button opens form in add mode
   - Tapping a coffee card opens form in edit mode

Use shadcn form components with proper validation.
```

✓ CHECK-IN: Before moving on, verify:

- □ Clicking 'Add Coffee' opens the form modal
- □ All form fields are present and styled nicely
- □ You can fill in fields and click Save
- □ After saving, the coffee appears in the list
- □ Clicking a coffee opens it for editing
- □ Editing and saving updates the coffee

💡 **TEACHING MOMENT: Forms and Validation**

Forms are how users input data. Good forms have: clear labels, appropriate input types (text, date, select), validation (checking the data is correct), and feedback (showing errors or success). We're using shadcn's form components which handle a lot of this complexity for us.

🗣️ *DISCUSS: Try adding a coffee with no name. What happens? Why do we make the name required but other fields optional?*

Step 19	**Create Grinder and Brewer Forms**	15 min

Let's repeat the pattern for grinders and brewers. Since the pattern is similar, Claude Code can do this quickly.

```
Create the list pages and forms for grinders and brewers,
following the same pattern as coffees:

1. app/(main)/settings/grinders/page.tsx
2. components/equipment/GrinderForm.tsx
   Fields: name*, brand, model, type (electric/hand),
   burrType (flat/conical), burrSize, settingsMin,
   settingsMax, settingsType (stepped/stepless), notes

3. app/(main)/settings/brewers/page.tsx
4. components/equipment/BrewerForm.tsx
   Fields: name*, brand, type (pour over/immersion/espresso/
   drip/french press/other), material, capacityMl,
   filterType (paper/metal/cloth/none), notes

Make sure the Settings page links work correctly.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ Settings → My Grinders shows grinder list
- □ Settings → My Brewers shows brewer list
- □ Can add/edit grinders with all fields
- □ Can add/edit brewers with all fields
- □ Empty states work for both

💡 **TEACHING MOMENT: DRY Principle (Don't Repeat Yourself)**

Notice how similar these three forms are? A more advanced technique would be to create a single generic 'EquipmentForm' component that works for all three types. This is the DRY principle—avoiding code duplication. For learning, it's okay to have some repetition so you can see the pattern.

🗣️ *DISCUSS: If you wanted to add a fourth equipment type (like 'Kettles'), what files would you need to create?*

Step 20	**Add Sample Data**	5 min

Let's add some of your real equipment so we have data to work with!

📋 **MANUAL STEP: Add your actual equipment!**

Add at least:

- 2 coffees you have at home
- Your grinders (Fellow Ode, any hand grinders)
- Your brewers (AeroPress, Chemex, V60, etc.)

✓ **CHECK-IN: Before moving on, verify:**

- □ You've added real coffees from your cabinet
- □ Your grinders are in the system
- □ Your brewers are in the system
- □ You can see the counts on the Settings page

💡 **TEACHING MOMENT: Testing with Real Data**

Using real data (your actual equipment) instead of fake test data helps you understand if the app actually works for your needs. You might realize you need a field we forgot, or that a dropdown doesn't have the right options.

🗣️ *DISCUSS: Did you notice any fields missing that would be useful? Any options you wish were available?*

Phase 5: Brew Logging

The core feature—logging your brews!

Step 21	Build the Log Brew Screen - Layout	15 min

This is the main screen users see when they open the app. Let's build the structure.

```
Rebuild the Log Brew page (app/(main)/page.tsx) with:

1. Header showing current date and time

2. Collapsible sections (use shadcn Collapsible or Accordion):

   EQUIPMENT section:
   - Coffee dropdown (from useCoffees)
   - Grinder dropdown (from useGrinders)
   - Brewer dropdown (from useBrewers)

   PARAMETERS section:
   - Dose (number input, grams)
   - Water (number input, grams)
   - Ratio (auto-calculated display, e.g., '1:16.7')
   - Water temp (number input, °F, with toggle for °C)
   - Grind setting (text input)

   TIMING section:
   - Bloom time (number input, seconds)
   - Bloom water (number input, grams)
   - Total time (number input, seconds OR mm:ss format)

   NOTES section:
   - Technique notes (textarea)
   - Tasting notes (textarea)
   - Rating (1-10 slider or button group)

3. Fixed 'Save Brew' button at bottom

Don't wire up saving yet, just build the UI.
```

✓ CHECK-IN: Before moving on, verify:

- □ Log page shows all four sections
- □ Sections can expand/collapse
- □ Equipment dropdowns show your added equipment
- □ Ratio auto-calculates when you enter dose and water
- □ All inputs are styled consistently

💡 **TEACHING MOMENT: Calculated Fields**

The ratio field is calculated automatically from dose and water (ratio = water / dose). This is called a 'derived' or 'computed' value—it depends on other values and updates automatically. We don't store it separately; we calculate it when needed.

🗣️ *DISCUSS: Why might we want to show the ratio instead of making users calculate it themselves?*

Step 22	Wire Up Brew Saving	15 min

Now let's make the Save button actually save brews to the database.

```
Connect the Log Brew form to the database:

1. Create a custom hook or component state to manage form data

2. When saving:
   - Validate that at least timestamp is set (auto-set to now)
   - Denormalize names: copy coffeeName, grinderName, brewerName
     from the selected items
   - Call addCoffeeTime service
   - Show success toast
   - Reset form to defaults
   - Stay on the Log page (user might want to log another)

3. Handle errors:
   - Show error toast if save fails
   - Don't clear form on error

4. Show loading state on Save button while saving

5. Make empty fields null, not empty strings or 0
```

✓ **CHECK-IN: Before moving on, verify:**

- □ Fill in some brew parameters and click Save
- □ A success toast appears
- □ The form resets (ready for another brew)
- □ Check Firebase Console → Firestore to see the saved brew

💡 **TEACHING MOMENT: Denormalization**

We're storing both coffeeId AND coffeeName in each brew log. This is 'denormalization'—intentionally duplicating data. Why? It makes reading data faster (we don't have to look up the coffee name every time), and it preserves history (if you rename a coffee later, old brews still show the original name).

🗣️ *DISCUSS: Look at the saved brew in Firebase Console. Can you identify all the fields? Which are null?*

Step 23	**Log Your First Real Brew!**	10 min

Time to make some coffee and log it!

☕ **MANUAL STEP: Actually make coffee and log the brew!**

1. Make a pour over, AeroPress, or whatever you want
2. Measure your parameters as you go
3. Log everything in the app
4. Taste the coffee and add tasting notes + rating
5. Log 2-3 more brews over the next day or two

✓ **CHECK-IN: Before moving on, verify:**

- □ You've logged at least one real brew
- □ All the fields you used saved correctly
- □ The experience felt usable (note any frustrations!)

💡 **TEACHING MOMENT: User Testing**

You just did 'user testing'—trying the app as a real user would. This always reveals problems that weren't obvious when coding. Maybe a field is hard to reach, the save button is awkward, or you wish you could do something differently. This feedback is gold for making the app better.

🗣️ *DISCUSS: What was frustrating or awkward about logging that brew? What would make it faster/easier?*

Phase 6: History & Detail View

Viewing and searching past brews

Step 24	Build the History List	15 min

The History page shows all your past brews so you can look back and learn.

```
Build the History page (app/(main)/history/page.tsx):

1. Search bar at top (we'll wire up search later)

2. Brew cards showing:
   - Date and time
   - Coffee name (or 'Unknown coffee' if none)
   - Brewer name
   - Key stats: dose → water (ratio), time
   - Rating as stars or number
   - Tasting notes preview (truncated)

3. Load brews using useCoffeeTimes hook
   - Sort by timestamp descending (newest first)
   - Show loading state
   - Show empty state if no brews

4. Cards should be tappable (we'll add detail view next)

5. Consider infinite scroll or 'Load more' button

Make the cards look great with our dark theme!
```

✓ CHECK-IN: Before moving on, verify:

- □ History page shows your logged brews
- □ Brews are sorted newest first
- □ Each card shows date, coffee, brewer, stats
- □ Empty state shows if you delete all brews
- □ Cards are visually appealing

💡 **TEACHING MOMENT: Data Presentation**

Notice how we don't show every field on the card—just the most important ones. Good UI design means showing the right information at the right time. The card shows enough to recognize a brew; the detail view (next step) shows everything.

🗣️ *DISCUSS: If you had 500 brews, would loading them all at once be a problem? How does 'Load more' help?*

Step 25	Create Brew Detail View	15 min

Tapping a brew card should open a detailed view where you can see everything and make edits.

```
Create the brew detail view as a full-screen modal or page:

Option A: app/(main)/history/[id]/page.tsx (new page)
Option B: components/brew/BrewDetail.tsx (modal)

Choose whichever feels right, then:

1. Show all brew data in organized sections:
   - Header: Date, time, rating
   - Equipment: Coffee, grinder, brewer (with full names)
   - Parameters: Dose, water, ratio, temp, grind setting
   - Timing: Bloom time/water, total time
   - Notes: Technique and tasting notes (full text)

2. Edit button that opens the brew in edit mode
   (can reuse a form similar to the Log page)

3. Delete button with confirmation dialog

4. Back button to return to history

5. Handle missing data gracefully (show '-' or skip field)
```

✓ CHECK-IN: Before moving on, verify:

- □ Tapping a brew in History opens detail view
- □ All logged data is displayed
- □ Missing fields show gracefully (not 'null' or errors)
- □ Edit mode works
- □ Delete with confirmation works
- □ Can navigate back to History

💡 **TEACHING MOMENT: Dynamic Routes**

If you chose Option A, you used a dynamic route: [id] in the folder name means the ID comes from the URL. Visiting /history/abc123 makes 'abc123' available as a parameter. This is how apps show different content based on the URL.

🗣️ *DISCUSS: What are the pros and cons of opening a new page vs. showing a modal?*

Step 26	**Add Search and Filters**	15 min

Let's make it easy to find specific brews.

```
Add search and filtering to the History page:

1. Search bar filters brews by:
   - Coffee name (contains search text)
   - Roaster name
   - Tasting notes
   (Client-side filtering is fine for now)

2. Filter chips/buttons for:
   - Date range: Today, This Week, This Month, All Time
   - Coffee: Dropdown of your coffees
   - Brewer: Dropdown of your brewers
   - Rating: 7+, 8+, 9+ filter

3. Show active filter count

4. 'Clear filters' button when filters are active

5. Show 'No results' when search/filters match nothing
```

✓ CHECK-IN: Before moving on, verify:

- □ Search bar filters brews as you type
- □ Date range filters work
- □ Coffee/Brewer dropdown filters work
- □ Rating filter works
- □ Filters can be combined
- □ Clear filters resets everything

💡 **TEACHING MOMENT: Client-side vs Server-side Filtering**

We're filtering on the client (in the browser) after loading all brews. This is simple and fast for small amounts of data. With thousands of brews, we'd need server-side filtering—sending the filters to Firestore and only getting matching results. For a personal app, client-side is usually fine.

🗣️ *DISCUSS: If you had 10,000 brews, why would loading them all and filtering locally be slow?*

Phase 7: Analytics

Understanding your brewing patterns

Step 27	**Build Analytics Dashboard**	20 min

Charts and stats help you learn from your brewing history.

```
Build the Analytics page (app/(main)/analytics/page.tsx):

1. Date range selector at top: 7 days, 30 days, 90 days, All time

2. Summary stats cards:
   - Total brews
   - Average rating
   - Most used coffee
   - Most used brewer

3. Charts using Recharts library:

   Brews Over Time (line or bar chart):
   - X axis: dates
   - Y axis: number of brews

   Rating Distribution (bar chart):
   - X axis: ratings 1-10
   - Y axis: count of brews with that rating

   Brews by Coffee (donut/pie chart):
   - Shows which coffees you brew most

4. Install Recharts: npm install recharts

5. Use our warm accent color (#D4A574) in charts

6. Show loading states and empty states appropriately
```

✓ CHECK-IN: Before moving on, verify:

- □ Analytics page shows summary stats
- □ Brews Over Time chart displays
- □ Rating Distribution chart displays
- □ Brews by Coffee chart displays
- □ Date range selector changes the data
- □ Charts use our color scheme

💡 TEACHING MOMENT: Data Visualization

Charts turn numbers into pictures that are easier to understand at a glance. A line going up tells you something is increasing without reading any numbers. Choosing the right chart type matters: line charts show trends over time, bar charts compare quantities, pie charts show parts of a whole.

🗣️ *DISCUSS: What insights can you already see from your charts? Any surprises?*

Phase 8: PWA Setup

Making the app installable on your phone

Step 28	Configure PWA Manifest	15 min

A PWA needs a 'manifest' file that tells the phone how to install and display the app.

```
Set up PWA configuration:

1. Install next-pwa: npm install next-pwa

2. Create public/manifest.json with:
   - name: 'CoffeeTime'
   - short_name: 'CoffeeTime'
   - description: 'Track your perfect brew'
   - start_url: '/'
   - display: 'standalone'
   - background_color: '#0A0A0B'
   - theme_color: '#D4A574'
   - icons: array of icon sizes (we'll add icons next)

3. Update next.config.js to use next-pwa

4. Add manifest link to app/layout.tsx <head>

5. Add meta tags for PWA:
   - apple-mobile-web-app-capable
   - apple-mobile-web-app-status-bar-style
   - theme-color
```

✓ CHECK-IN: Before moving on, verify:

- □ public/manifest.json exists
- □ next.config.js has PWA config
- □ No build errors

💡 TEACHING MOMENT: Web App Manifest

The manifest is a JSON file that describes your app to the operating system: its name, colors, icons, and how it should behave when installed. It's what allows a website to be 'installed' and appear in your app drawer alongside native apps.

🗣️ *DISCUSS: Why does the manifest need multiple icon sizes?*

Step 29	Create App Icons	10 min

The app needs icons at various sizes for different devices and contexts.

```
Create app icons:

1. Generate a simple icon for CoffeeTime:
   - Coffee cup emoji style or simple coffee icon
   - Brown/cream color scheme on dark background
   - Clean, minimal design

2. Create these sizes in public/icons/:
   - icon-192x192.png
   - icon-512x512.png
   - apple-touch-icon.png (180x180)

   (You can use a single large icon and resize,
   or create an SVG that exports to multiple sizes)

3. Update manifest.json with icon paths

4. Add apple-touch-icon link to layout.tsx

Note: If you can't generate images, use placeholder
colored squares and we'll replace them later.
```

✓ CHECK-IN: Before moving on, verify:

- □ Icons exist in public/icons/
- □ manifest.json references the icons
- □ No console errors about missing icons

💡 TEACHING MOMENT: App Icons

Apps need icons at many sizes: the home screen uses one size, the app switcher uses another, notifications use another. The operating system picks the best size for each context. That's why we provide multiple sizes in the manifest.

🗣️ *DISCUSS: What makes a good app icon? Think about apps you use—what makes their icons recognizable?*

Step 30	**Test PWA Installation**	10 min

Let's install the app on a phone!

📋 **MANUAL STEPS:**

First, we need to access the app from your phone:

1. Your computer and phone need to be on the same WiFi
2. Find your computer's local IP address
3. On phone, open Chrome and go to http://[YOUR-IP]:3000
4. Chrome should show 'Add to Home Screen' in the menu
5. Tap it to install the app!

```
Help me find my computer's local IP address so I can access
the app from my phone. Also check that the PWA is configured
correctly by looking at the manifest.
```

✓ **CHECK-IN: Before moving on, verify:**

- □ You can access the app from your phone via IP address
- □ Chrome offers to 'Add to Home Screen' or 'Install'
- □ After installing, the app appears in your app drawer
- □ Opening the installed app goes full-screen (no browser UI)
- □ The app icon shows (even if it's a placeholder)

💡 **TEACHING MOMENT: PWA Installation**

When you 'install' a PWA, your phone downloads the web app and caches it locally. It gets an icon in your app drawer and opens without browser controls (like a native app). The app still loads from the web, but cached files make it fast, and service workers can enable offline access.

🗣️ *DISCUSS: How does using the app as a PWA feel different from using it in the browser?*

🎉 Congratulations!

You've built a real, working app! CoffeeTime can now:

- ✅ Sign in with your Google account
- ✅ Manage your coffees, grinders, and brewers
- ✅ Log brews with all your parameters
- ✅ View and search your brew history
- ✅ See analytics about your brewing patterns
- ✅ Install on your phone as a PWA

Continue Building: Sharing Features

The core app is done! Now you can add social and sharing features. These are more advanced but follow the same patterns you've learned.

Phase 9: Share Individual Brews (Steps 31-35)

- Create a public /shared/[id] page anyone can view
- Add Share button to brew detail screen
- Generate shareable links with short codes

Phase 10: Follow/Subscribe System (Steps 36-42)

- Create user profiles with share codes
- Follow other coffee enthusiasts
- See a feed of brews from people you follow

Phase 11: Coffee Circles (Steps 43-49)

- Create collaborative groups (like a coffee club)
- Invite friends via code or link
- Share brews with your group

Phase 12: Google Drive Export (Steps 50-56)

- Export your data to Google Drive
- Choose CSV, JSON, or Google Sheets format
- Share exports using Google Drive sharing

Each of these phases follows the same pattern: define the data model, create the service, build the hooks, then create the UI. You've already mastered this pattern!

Other Future Ideas

Even more features you could add:

- Voice input: Speak your brew and have AI parse it
- Photo recognition: Photograph coffee bags to auto-fill data
- Deploy to the internet: Host on Firebase Hosting
- Timer feature: Built-in brew timer

Concepts You Learned

- How web apps work (frontend, backend, database)
- React components and hooks
- TypeScript interfaces and type safety
- Firebase authentication and database
- The service layer pattern
- Routing and navigation
- Forms and data validation
- Data visualization with charts
- Progressive Web Apps
- Public vs private data and security rules
- Social features: following, groups, sharing

☕ Happy Brewing! ☕