Build a Quiz App using Angular 18, Signals for reactivity, and standalone components.

Angular Signals are a modern, fine-grained reactivity system introduced in Angular 16+. They're a simpler alternative to RxJS when:

Your state is local, not streaming  
You want fine control over updates  
You want a zone-free, high-performance app  
You want declarative UI logic like computed() and effect()

**Step 1: Add Angular Material to App (Optional)**

Install Material (for styled buttons or radio inputs):

ng add @angular/material

**Step 2: Set Up Global State with Signals**

Create src/app/state/quiz.store.ts

Using signal() makes your state reactive and local — no need for boilerplate RxJS.

import { signal, computed } from '@angular/core';

export interface Question {

id: number;

question: string;

options: string[];

answer: string;

}

const \_questions = signal<Question[]>([]);

const \_currentIndex = signal(0);

const \_selectedAnswer = signal('');

const \_score = signal(0);

export const quizStore = {

questions: computed(() => \_questions()),

currentIndex: computed(() => \_currentIndex()),

selectedAnswer: computed(() => \_selectedAnswer()),

score: computed(() => \_score()),

loadQuestions: (qs: Question[]) => \_questions.set(qs),

selectAnswer: (ans: string) => \_selectedAnswer.set(ans),

next: () => {

const q = \_questions();

const i = \_currentIndex();

if (\_selectedAnswer() === q[i]?.answer) \_score.update(s => s + 1);

\_selectedAnswer.set('');

\_currentIndex.set(i + 1);

},

reset: () => {

\_questions.set([]);

\_currentIndex.set(0);

\_selectedAnswer.set('');

\_score.set(0);

}

};

**Step 3: Quiz API Service**

Create src/app/services/quiz.service.ts

Separates API logic from UI logic.

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Question } from '../state/quiz.store';

@Injectable({ providedIn: 'root' })

export class QuizService {

private readonly apiUrl = 'http://localhost:3000/questions'; // JSON server or your backend

constructor(private http: HttpClient) {}

getQuestions() {

return this.http.get<Question[]>(this.apiUrl);

}

}

**Step 4: Create QuizComponent**

ng generate component pages/quiz —standalone

**In quiz.component.ts, replace contents with:**

import { Component, OnInit, computed, effect } from '@angular/core';

import { Router } from '@angular/router';

import { quizStore } from '../../state/quiz.store';

import { QuizService } from '../../services/quiz.service';

import { NgIf, NgFor } from '@angular/common';

@Component({

standalone: true,

selector: 'app-quiz',

templateUrl: './quiz.component.html',

styleUrls: ['./quiz.component.scss'],

imports: [NgIf, NgFor]

})

export class QuizComponent implements OnInit {

quizStore = quizStore;

currentQuestion = computed(() => this.quizStore.questions()[this.quizStore.currentIndex()]);

constructor(private quizService: QuizService, private router: Router) {

effect(() => {

const selected = this.quizStore.selectedAnswer();

if (selected) console.log('[Signal Effect] Selected answer:', selected);

});

}

ngOnInit(): void {

this.quizService.getQuestions().subscribe(qs => this.quizStore.loadQuestions(qs));

}

selectAnswer(option: string) {

this.quizStore.selectAnswer(option);

}

nextQuestion() {

this.quizStore.next();

if (this.quizStore.currentIndex() >= this.quizStore.questions().length) {

this.router.navigate(['/result']);

}

}

}

Then in quiz.component.html:

<div \*ngIf="currentQuestion() as question">

<h2>{{ question.question }}</h2>

<ul>

<li \*ngFor="let option of question.options"

(click)="selectAnswer(option)"

[class.selected]="option === quizStore.selectedAnswer()">

{{ option }}

</li>

</ul>

<button (click)="nextQuestion()">Next</button>

</div>

**Step 5: Result Page**

ng generate component pages/result —standalone

**In result.component.ts:**

import { Component, computed } from '@angular/core';

import { quizStore } from '../../state/quiz.store';

import { NgIf } from '@angular/common';

import { RouterModule } from '@angular/router';

@Component({

standalone: true,

selector: 'app-result',

templateUrl: './result.component.html',

styleUrls: ['./result.component.scss'],

imports: [NgIf, RouterModule]

})

export class ResultComponent {

quizStore = quizStore;

result = computed(() => ({

score: this.quizStore.score(),

total: this.quizStore.questions().length

}));

}

**In result.component.html:**

<div \*ngIf="result() as res">

<h2>Your Score: {{ res.score }} / {{ res.total }}</h2>

<p \*ngIf="res.score === res.total">🎉 Perfect score!</p>

<p \*ngIf="res.score < res.total && res.score > 0">👍 Good job!</p>

<p \*ngIf="res.score === 0">😅 Better luck next time!</p>

<button routerLink="/quiz" (click)="quizStore.reset()">Try Again</button>

</div>

**Step 6: Add Routes**

Create or update src/app/app.routes.ts:

import { Routes } from '@angular/router';

import { QuizComponent } from './pages/quiz/quiz.component';

import { ResultComponent } from './pages/result/result.component';

export const routes: Routes = [

{ path: '', redirectTo: 'quiz', pathMatch: 'full' },

{ path: 'quiz', component: QuizComponent },

{ path: 'result', component: ResultComponent }

];

**Step 7: Root App Component**

**Create app.component.ts:**

import { Component } from '@angular/core';

import { RouterOutlet } from '@angular/router';

@Component({

selector: 'app-root',

standalone: true,

imports: [RouterOutlet],

template: `<router-outlet></router-outlet>`

})

export class AppComponent {}

**Step 8: Bootstrap the App**

**Update src/main.ts:**

import { bootstrapApplication } from '@angular/platform-browser';

import { provideRouter } from '@angular/router';

import { provideHttpClient } from '@angular/common/http';

import { routes } from './app/app.routes';

import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {

providers: [provideRouter(routes), provideHttpClient()]

});

**Step 9: Run It**

Start your server:

ng serve

If your questions come from a local db.json, also run:

npx json-server --watch db.json

**With Signals, We Can Make It Zoneless**

**Step 1: Disable Zone.js**

In your angular.json, remove or comment out Zone.js:

"scripts": [

// "node\_modules/zone.js", ← remove or comment this out

]

Also remove the Zone.js import from polyfills.ts, if it's there:

// zone.js import — delete this:

import 'zone.js'; // REMOVE THIS LINE

**Step 2: Use ngZone: 'noop' in main.ts**

Update your bootstrapApplication call in main.ts:

import { bootstrapApplication } from '@angular/platform-browser';

import { provideRouter } from '@angular/router';

import { provideHttpClient } from '@angular/common/http';

import { routes } from './app/app.routes';

import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {

providers: [provideRouter(routes), provideHttpClient()],

// This disables Zone.js entirely

ngZone: 'noop'

});

**Result**

* Your app is now zoneless, and Angular won’t patch async APIs.
* Signals become responsible for reactivity.
* Use signal(), computed(), and effect() for updates with no fallback to Zone.js.

**Why Go Zoneless?**

|  |  |
| --- | --- |
| **Benefit** | **Explanation** |
| Faster change detection | Only updates what's actually changed |
| Cleaner integration | No monkey-patching of browser APIs |
| Predictable rendering | UI reacts *only* to state tracked in Signals |
| Better for modern frameworks | Aligns Angular with React, SolidJS, etc. |

**The Significance of Zoneless Angular**

“Zoneless” Angular means running Angular **without Zone.js**, the legacy change detection mechanism. This shift is **foundational** — and here’s why it matters:

**What is Zone.js (and why it was used)?**

Zone.js is a library Angular traditionally used to monkey-patch all async behavior (like setTimeout, fetch, addEventListener) to automatically know when to trigger change detection.

**But:**

* It’s inefficient — it triggers *global* change detection on every tiny async event
* It’s not granular — Angular has no idea *what* actually changed
* It patches native browser APIs, which can break third-party libraries or tools
* It hides cause and effect, making debugging harder

**Zoneless Angular (with Signals) — What It Means**

1. **No Zone.js**
   * You remove the old global monkey-patching.
   * Angular stops "watching everything" just in case.
2. **You control reactivity** with:
   * signal(): defines reactive state
   * computed(): derived values that update only when their dependencies do
   * effect(): run side-effects when signals change

**Key Benefits of Zoneless Angular**

|  |  |
| --- | --- |
| **Benefit** | **Why It Matters** |
| **Performance** | Reactivity is scoped. Instead of updating the whole page, Angular only re-renders parts tied to specific signals. |
| **Debuggability** | You know exactly what caused the UI to update — it’s tied to specific signals, not random async triggers. |
| **Simplicity** | You no longer need ChangeDetectorRef, NgZone.run(), or manual change detection tricks. |
| **Better DX** | It feels more like React or Solid.js — clean, state-driven UI that only updates when it needs to. |
| **No monkey-patching** | Zone.js can break third-party code, especially native Web APIs or complex event-driven libraries. Without it, your app plays better with the ecosystem. |
| **Modern & Future-Ready** | Angular’s long-term vision is zoneless + signals. You’re aligning with where the framework is going. |

**Real-World Example**

|  |  |  |
| --- | --- | --- |
| **Action** | **Zone.js Angular** | **Zoneless + Signals** |
| setTimeout fires | Triggers **full change detection** tree | **No update** unless a signal() changes |
| A user selects an option | Triggers full check of the app | Only updates UI **tied to that signal** |
| An animation finishes | Angular checks *everything* again | Angular **does nothing** unless a signal changes |

**Summary:**

Zoneless Angular shifts you from "watch everything" to "update only when something you care about changes."

* Faster
* Cleaner
* More scalable
* And more like modern reactive frameworks (React, Solid, Svelte)