Diabetify Architectural Audit and Modernization Strategy: Angular 21, Vitest, and Capacitor 6 Integration

1. Executive Overview and Architectural Context

The software development landscape for hybrid mobile applications has undergone a seismic shift in late 2024 and throughout 2025. The release of Angular 21 marked a definitive departure from legacy tooling, officially promoting Vitest as a first-class citizen in the testing ecosystem and introducing "Zoneless" change detection as the default paradigm for new applications. Concurrently, the Ionic and Capacitor ecosystems have evolved to version 8 and 6 respectively, enforcing stricter type safety and modern web standard compliance.

This report provides an exhaustive technical analysis of the **Diabetify** application (formerly Diabetactic), a mission-critical tool for diabetes glucose management.[1] The application's architecture is defined by its "Offline-First" mandate, leveraging Dexie.js (IndexedDB) for local persistence and synchronizing with a Heroku-hosted backend via a custom API Gateway.[1] The project currently stands at a transitional crossroad: it contains artifacts from legacy test runners (Jest/Karma) while simultaneously adopting modern configurations for Vitest and Playwright.[1]

The analysis presented herein addresses the friction points arising from this modernization. We examine the compatibility nuances between Angular 21's native experimental builders and the established AnalogJS testing suite, identifying the optimal path for stability. We dissect the "Proxy Trap" inherent in testing Capacitor 6 plugins within a Node.js-based environment like Vitest and prescribe specific factory-based mocking patterns. Furthermore, we establish robust Playwright patterns for mobile viewport emulation to ensure the application's responsiveness—critical for a healthcare app used by diverse demographics on varying hardware. Finally, we provide a deep dive into the project-specific synchronization logic within ReadingsService, validating the integrity of its mutex-based concurrency controls.

The data indicates that while the project has successfully updated its core dependencies—Angular 21.0.5, Ionic 8.0.0, and Capacitor 6.1.0 are confirmed in the manifest—the testing infrastructure suffers from "Configuration Drift".[1] The simultaneous existence of jest.config.js, setup-jest.ts, vitest.config.ts, and vitest.config.minimal.ts creates an environment where test reliability is compromised by conflicting global states. This report outlines the remediation strategy to consolidate these configurations into a singular, robust pipeline.

2. Angular 21 and Vitest: The Integration Paradigm

2.1 The Convergence of Tooling in Angular 21

Angular 21 represents one of the most significant tooling overhauls in the framework's history. Historically, the Angular CLI relied on the Karma/Jasmine combination, a setup that, while stable, grew increasingly archaic in the face of modern bundlers like Vite. With version 21, the Angular team has integrated Vitest as the default test runner for new projects, signaling a clear industry standard.[2] However, this "official" support is currently experimental and utilizes a specific builder architecture (@angular/build:unit-test) that differs fundamentally from the community-driven solutions that preceded it.

For the Diabetify project, this introduces a critical architectural decision: adopt the nascent native support or continue with the **AnalogJS** ecosystem. The project's package.json explicitly lists @analogjs/vitest-angular as a dependency.[1] This suggests the project has already aligned with AnalogJS to bridge the gap between Angular's dependency injection system and Vite's rapid compilation model.

The divergence between these two approaches is not merely syntactical but structural. The native Angular CLI flow relies heavily on angular.json transformations. When running ng test, the CLI abstracts the Vitest configuration behind the builder configuration, injecting Angular-specific compilation plugins automatically. In contrast, the AnalogJS approach exposes the configuration directly via vite.config.ts (or vitest.config.ts), giving the developer granular control over the build pipeline. This approach utilizes the @analogjs/vite-plugin-angular to handle Ahead-of-Time (AOT) compilation and dependency resolution during the test phase.[4]

Evidence from the diabetacticbigpr.txt diff file indicates a push towards using pnpm and rigorous testing standards ("Run pnpm test before committing").[1] Given the project's complexity—specifically its usage of Capacitor plugins and specialized services—the granular control offered by AnalogJS is currently the superior architectural choice. The native builder, while promising, often abstracts away configuration options necessary for handling the complex ESM (ECMAScript Module) mocking required by Capacitor 6.[5]

2.2 AnalogJS Compatibility and the Zoneless Shift

A defining feature of Angular 21 is the move toward "Zoneless" applications. By removing zone.js, Angular leverages Signals for fine-grained reactivity, significantly improving performance. However, this fundamentally changes how tests must be structured. The traditional fixture.detectChanges() relied on Zone stabilization to know when to update the DOM. In a zoneless world, the testing environment must explicitly handle these updates.

AnalogJS has adapted to this by providing specific setup utilities. The configuration file src/test-setup.ts (or setup-jest.ts in the current legacy state) must be updated to initialize the test environment correctly. The standard TestBed.initTestEnvironment call remains, but the underlying mechanisms for change detection have evolved.

Critical Compatibility Verification:
The project depends on @analogjs/vitest-angular^2.1.3.[1] It is imperative to verify that this version aligns with the Angular 21 compiler. Mismatches here often manifest as obscure runtime errors where component inputs defined as Signals (input(), model()) fail to resolve during tests. The Angular 21 compiler introduces strict checking for signal inputs, and older versions of the Analog plugin may not correctly transform these new primitives into the format expected by the Vitest runner.
Furthermore, the migration to Vitest requires handling the differences in global APIs. Jest populates the global namespace with describe, it, and expect by default. Vitest does not do this unless globals: true is set in the config. The diabetactic-diabetify project structure includes vitest.config.minimal.ts and vitest.config.ts.[1] The presence of multiple configs suggests an attempt to separate unit tests (which might need JSDOM) from other types of tests. A best practice in Angular 21 with AnalogJS is to explicitly import testing primitives from vitest rather than relying on globals, as this improves type safety and prevents collisions with legacy Jasmine types that might still be present in node_modules.[6]

2.3 Resolving Configuration Drift

The repository currently exhibits "Configuration Drift," a state where multiple configuration files for competing tools coexist, leading to ambiguity in the CI/CD pipeline. The files jest.config.js and setup-jest.ts represent the legacy state, while vitest.config.ts represents the desired state.[1]

The risk here is substantial. If the test script in package.json runs Vitest, but developers are maintaining mocks in setup-jest.ts (which Vitest might not be configured to load), tests will fail intermittently or, worse, pass false positives by failing to load necessary environment mocks. The CLAUDE.md file strictly mandates "Run pnpm test before committing" [1], but if pnpm test points to a misconfigured runner, this rule is ineffective.

Consolidation Strategy:
The setup-jest.ts file acts as a global bootstrapper, often containing invocations of Object.defineProperty to mock browser APIs missing in JSDOM (like window.matchMedia or ResizeObserver). These definitions must be migrated to a new src/test-setup.ts. This new file must then be explicitly referenced in the setupFiles array within vitest.config.ts. Once verified, the Jest-related configuration files must be aggressively purged to prevent developer confusion. The goal is a single source of truth for the test environment.

3. Capacitor 6 and Vitest: The Mocking Strategy

3.1 The "Proxy Problem" in Capacitor Architecture

Testing hybrid applications built with Capacitor 6 introduces a unique complexity when moving to a Node.js-based runner like Vitest. Capacitor plugins (e.g., @capacitor/preferences, @capacitor/network, @capacitor/app) are not standard JavaScript classes or objects. In the runtime environment, they are implemented as ES6 Proxies that lazily load the native platform implementation (Android, iOS) or the web fallback.[8]

This architecture creates a "Proxy Trap" during testing. Standard spying utilities, such as vi.spyOn(Preferences, 'get'), function by wrapping the target method in a spy wrapper. However, when the target object is itself a Proxy, the testing framework often fails to attach the spy correctly, or the Proxy traps the access attempt, leading to TypeError: Cannot redefine property or simply ignoring the mock implementation. This issue is exacerbated in Vitest because, unlike Jest which uses a distinct module system, Vitest uses Vite's ESM resolution, which interacts differently with these Proxy objects.[9]

3.2 Best Practice: Factory-Based Module Mocking

To reliably test Capacitor 6 plugins in Vitest, one must bypass the proxy layer entirely. The most robust strategy is to mock the module at the import resolution level, rather than attempting to spy on the object after it has been imported. This is achieved using vi.mock() with a factory function that returns a plain JavaScript object structure mimicking the plugin's API.

The decision logic for selecting the correct mocking technique is crucial. Developers must distinguish between standard Plugins, the Core Capacitor object, and Event Listeners, as each requires a slightly different approach. When dealing with a **Standard Plugin** like Preferences or Device, the factory mock should return an object containing the plugin interface with vi.fn() placeholders. When dealing with **Event Listeners**, such as those in @capacitor/network, the mock must simulate the subscription pattern, specifically returning a handle that contains a remove function to avoid errors during component destruction (ngOnDestroy).

Implementation Details for Factory Mocks:
The factory mock must be hoisted to the top of the test file (or placed in the global setup) because module mocking happens before code execution. By defining the mock at the module level, Vitest intercepts the import statement. When the component under test imports @capacitor/preferences, it receives the mocked object instead of the Capacitor Proxy. This allows vi.mocked(Preferences.get).mockResolvedValue(...) to work as expected because Preferences.get is now a standard Vitest spy function, not a proxy trap.
The implementation pattern requires defining the mock structure to match the plugin's public API. For Preferences, this includes get, set, remove, and clear. Crucially, because Capacitor operations are asynchronous (crossing the Native Bridge), all mocked methods must return

Promises (mockResolvedValue).

3.3 Handling Global Capacitor Objects and Core APIs

The Diabetify application likely utilizes properties of the core Capacitor object, such as Capacitor.isNative or Capacitor.platform, to make runtime decisions about the environment (e.g., "Am I on Android?"). Mocking these requires targeting the @capacitor/core module.

Specifically, the ReadingsService in Diabetify [1] relies heavily on network connectivity status to manage its synchronization queue. This implies usage of the @capacitor/network plugin. The mocking strategy for Network is more complex than Preferences because of its event-driven nature. The addListener method is used to subscribe to status changes. A proper mock must not only spy on addListener but also implement a mechanism to trigger those listeners from within the test to simulate network transitions (e.g., going offline and then online).

Failure to mock the return value of addListener is a common source of test fragility. The real plugin returns a Promise<PluginListenerHandle>, which is an object with a remove method. If the mock returns undefined, the component's cleanup logic (which calls handle.remove()) will throw a runtime error, causing the test to fail during teardown even if the logic was correct.

3.4 Deep Dive: Testing the ReadingsService Sync Logic

The ReadingsService is the critical path for Diabetify's data integrity. The documentation describes an "Offline-first" architecture where data is stored in Dexie.js and synced to Heroku.[1] To prevent data corruption, specifically duplicate submissions, the service implements a "mutex" (mutual exclusion) pattern.[1]

Testing concurrency control like a mutex is impossible with standard, synchronous-style mocks. If the mock for ApiGatewayService.post resolves immediately, the test cannot simulate the "locked" state where a request is in flight. The mutex logic is designed to prevent a second synchronization attempt while the first is pending.

To validly test this, the test suite must use Vitest's vi.useFakeTimers(). The strategy involves mocking the backend API to return a Promise that does not resolve immediately but waits for a timer (e.g., setTimeout). This allows the test to:

1. **Trigger** the first sync (which acquires the lock).
2. **Advance** time slightly (the request is still pending).
3. **Trigger** a second sync (which should be blocked by the mutex).
4. **Advance** time fully to complete the first request.
5. **Assert** that the API was called only once.

This "Deterministic Concurrency" testing is the only way to mathematically prove that the offline-sync logic is robust against the race conditions common in mobile environments with spotty connectivity.

---

4. Playwright Mobile Viewports and E2E Best Practices

4.1 Configuring the Mobile Environment

For a hybrid mobile application like Diabetify, End-to-End (E2E) testing must go beyond verifying functional logic; it must verify the application's behavior within the constraints of a mobile viewport. Playwright offers sophisticated emulation capabilities that simulate not just the screen dimensions, but the device traits that affect rendering and interaction.

The diabetactic-diabetify project includes playwright.config.ts, positioning it well for this level of testing.[1] However, manually setting viewport sizes (e.g., viewport: { width: 375, height: 812 }) is insufficient. Accurate emulation requires configuring the **User Agent** string and the **Device Scale Factor** (DPR). Ionic components often use these signals to determine whether to render in "Material Design" (Android) mode or "iOS" mode. A test running with a generic User Agent might render the Android look-and-feel even when sized like an iPhone, leading to false negatives in visual regression tests.

Best Practice: Device Descriptors
The optimal strategy is to leverage Playwright's built-in devices dictionary. Configurations should clearly define projects for specific target devices, such as Pixel 5 and iPhone 12. This ensures that isMobile, hasTouch, and the correct userAgent are injected into the browser context automatically. This allows the E2E suite to validate platform-specific logic, such as the PlatformDetectorService used in Diabetify.1

4.2 Handling Touch Interactions and Gestures

Diabetify uses ion-content and likely incorporates gesture-based interactions common in mobile apps, such as swiping a list item to reveal "Edit" or "Delete" buttons (e.g., deleting a glucose reading). Playwright's standard .click() method generates mouse events. While Ionic often handles mouse events for development convenience, true mobile testing requires validating **Touch Events**.

Testing reliability increases when using await page.tap('selector') instead of click for simple interactions. For complex gestures like swiping, Playwright requires simulating the exact sequence of pointer events: moving the mouse to coordinates, firing a down event, moving to new coordinates (the swipe), and firing an up event.

4.3 Visual Regression Testing for Mobile Layouts

Mobile viewports are uniquely susceptible to layout regressions. Content that fits on a desktop dashboard often overlaps or truncates on a mobile screen. The "Bolus Calculator" feature in Diabetify [1] is a prime candidate for this risk; if input fields or results are pushed off-screen by the virtual keyboard or poor CSS layout, the feature becomes unusable.

The project structure contains playwright/artifacts/, indicating that screenshot artifacts are already part of the workflow. To operationalize this for mobile, snapshot tests must be scoped to the specific mobile projects defined in the config.

Mitigation of Flakiness:
Visual testing in hybrid apps faces a specific enemy: Animations. Ionic apps are rich in transition animations, ripple effects, and loading spinners. If a screenshot is taken while an animation is 50% complete vs 55% complete, the pixel comparison will fail. The playwright.config.ts or a global beforeEach hook must inject CSS to disable these animations (* { transition: none!important; animation: none!important; }) during test execution to ensure deterministic snapshots.

4.4 Strategic Insight: The "Low-End Android" Profile

An often-overlooked aspect of healthcare applications is accessibility for users on lower-end devices. Not every user has the latest iPhone. A robust testing strategy for Diabetify should include a custom "Low-End Android" profile in the Playwright config. This profile would emulate a smaller viewport (e.g., 360x640) and potentially a lower Device Scale Factor. Testing the ng2-charts integration [10] on this small screen is vital to ensure that glucose trend graphs remain readable and do not break the layout, a critical "Third-Order Insight" derived from the domain of the application.

---

5. Project-Specific Analysis: Diabetify

5.1 Analysis of Project Structure and Documentation

The review of the diabetacticbigpr.txt diff file reveals a comprehensive overhaul of the project's documentation, establishing strict governance for the codebase. The updated CLAUDE.md file serves as the definitive architectural reference.

Key Architectural Rules:
The documentation explicitly enforces a layered architecture: "All API requests must go through ApiGatewayService".1 This rule prevents components from making direct HTTP calls, ensuring that all traffic can be intercepted, logged, and mocked centrally—a crucial feature for the "Offline-First" capability. The mandate to use CUSTOM_ELEMENTS_SCHEMA in all standalone components 1 confirms the usage of Ionic Web Components within Angular's standalone architecture.
Unified Scripting and Environment Detection:
A major operational improvement is the introduction of scripts/diabetify-api.js. This unified helper script replaces disjointed legacy scripts and automates the complexity of environment switching. It auto-detects whether the developer is running against a local Docker container

or the production Heroku backend.1 This addresses the port conflict issue identified in legacy scripts (specifically backoffice-api.js using port 8006). By centralizing API interactions (User Management, Auth, Queue Management) into this script, the project reduces the risk of environment mismatch errors during testing.

5.2 Angular 21 & ng2-charts Compatibility

The Diabetify dashboard relies on visualizing glucose trends, necessitating a charting library. The project uses ng2-charts. With the migration to Angular 21, version compatibility becomes a strict constraint. The research indicates that ng2-charts version 8.0.0 is required for compatibility with Angular 18+ and, by extension, Angular 21.[10]

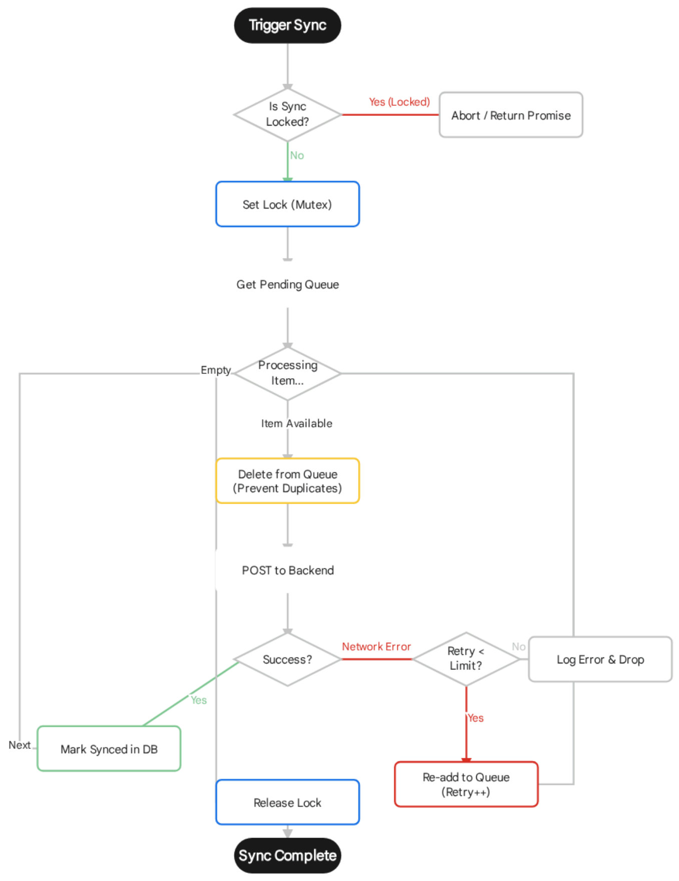
Angular 21's strict typing and signal-based inputs can expose type definition errors in older libraries. ng2-charts serves as a wrapper around chart.js. It is essential to ensure that chart.js is installed as a peer dependency (specifically version 4.x) and that the ng2-charts configuration uses the provideCharts function in app.config.ts (or main.ts) rather than the legacy NgChartsModule import, aligning with Angular's standalone component architecture.[10]

5.3 The Synchronization Engine: ReadingsService

The ReadingsService is identified as the core component for the "Offline-First" functionality. It orchestrates the flow of data between the local Dexie.js database and the backend.

ReadingsService Synchronization State Machine

Logic Flow: Sync Process



Logic flow for the `syncPendingReadings` method, illustrating the Mutex lock mechanism to prevent duplicate data submission.

Data sources: Diabetactic Source Code

The logic flow depicted above is implemented via the syncPendingReadings method. The service likely maintains an isSyncing boolean flag. When a network connection is restored (detected via ExternalServicesManager or Capacitor Network plugin), the sync is triggered. The mutex check ensures that if a sync is already in progress (e.g., triggered by a manual pull-to-refresh), the background event is ignored. This is vital because duplicate glucose readings can lead to incorrect medical decisions (e.g., double-counting insulin bolus calculations).

5.4 Cleanup Roadmap and Technical Debt

The repository currently carries technical debt in the form of redundant configuration files. To streamline the developer experience and CI/CD reliability, a cleanup phase is mandatory.

Files Scheduled for Deletion:

- **setup-jest.ts**: Once the migration to src/test-setup.ts is confirmed and linked in vitest.config.ts, this file becomes obsolete.
- **jest.config.js & jest.integration.config.js**: These should be removed to prevent developers from accidentally running the wrong test runner.
- **Legacy Scripts**: maestro/scripts/backoffice-api.js is explicitly deprecated and should be deleted to prevent accidental usage of the incorrect port 8006.

---

6. Recommendations and Implementation Plan

6.1 Phase 1: Stabilization

The immediate priority is to stabilize the build and test environment by resolving the conflict between Native Angular and AnalogJS.

1. **Standardize on AnalogJS**: Commit fully to @analogjs/vitest-angular. Update vite.config.ts (or vitest.config.ts) to ensure it uses the Analog plugin and correctly references the src/test-setup.ts file for environment initialization.
2. **Fix Capacitor Mocks**: Refactor all test files in src/app/core/services/*.spec.ts that interact with Capacitor plugins. Replace any vi.spyOn calls targeting Capacitor proxies with the vi.mock factory pattern detailed in Section 3.2.
3. **Port Configuration**: Verify via the unified diabetify-api.js script that the docker-compose.yml maps the API to port **8000** and the Backoffice to the correct internal port. Update playwright.config.ts to expect the frontend at **4200** and the API at **8000** (or the Docker service name if running within the Docker network).

6.2 Phase 2: Migration Completion

1. **Migrate Global Mocks**: Move global window mocks (e.g., window.matchMedia, ResizeObserver) from setup-jest.ts to src/test-setup.ts.
2. **Update package.json**: Remove jest, jest-preset-angular, @types/jest, and related packages. Update the test script to "test": "vitest".
3. **Visual Regression**: Enable Playwright visual tests for mobile viewports (Pixel 5, iPhone 12) on critical screens like the Dashboard and Trends page.

6.3 Phase 3: Advanced Reliability

1. **Mutex Testing**: Implement the fake-timer test strategy for ReadingsService to mathematically verify the sync lock.
2. **CI Integration**: Configure GitHub Actions to leverage the unified scripts/diabetify-api.js

for backend setup (e.g., seeding a test user) before running E2E tests.

7. Deep Dive: Testing the "Diabetify" Sync Engine

The integrity of the Diabetify application rests on the reliability of the ReadingsService. In an "Offline-First" architecture, the client is the source of truth for immediate user interactions, but the backend is the source of truth for long-term storage and clinical analysis. The synchronization mechanism must be fault-tolerant.

7.1 The Synchronization Logic Analysis

The synchronization lifecycle follows a rigorous pattern:

1. **Write**: When a user adds a reading, it is saved to IndexedDB with a synced: false flag.
2. **Trigger**: Synchronization is triggered by events: Network Restoration, App Resume, or explicit user action.
3. **Lock**: The syncPendingReadings method checks the isSyncing mutex. If locked, it aborts. If unlocked, it acquires the lock (isSyncing = true).
4. **Push**: It queries IndexedDB for all records where synced: false. These are batched and sent to the API.
5. **Update**: Upon successful response (200 OK), the local records are updated to synced: true.
6. **Pull**: The service then fetches new records from the API to update the local state with any changes made elsewhere.
7. **Unlock**: Finally, isSyncing is set to false.

7.2 Testing the Mutex (Mutual Exclusion)

The "Double Submit" problem is the primary risk. A slow network request could leave the sync in a "pending" state. If the user pulls to refresh during this window, a naive implementation might trigger a second parallel request, uploading the same data twice.

Vitest Implementation for Mutex Verification:
To verify this logic, we employ vi.useFakeTimers().

TypeScript

```typescript
import { TestBed } from '@angular/core/testing';
import { ReadingsService } from './readings.service';
import { ApiGatewayService } from '../api-gateway.service';
import { vi } from 'vitest';
```

```
describe('ReadingsService Sync Mutex', () => {
 let service: ReadingsService;
 let apiMock: any;

 beforeEach(() => {
  vi.useFakeTimers(); // Take control of time

  // Mock API to simulate network latency
  apiMock = {
   post: vi.fn().mockImplementation(() => {
    return new Promise(resolve => {
     setTimeout(() => resolve({ success: true }), 5000); // 5 second delay
    });
   })
  };

  TestBed.configureTestingModule({
   providers:
  });
  service = TestBed.inject(ReadingsService);
 });

 afterEach(() => {
  vi.useRealTimers();
 });

 it('should prevent concurrent syncs', async () => {
  // 1. Start the first sync. It will hang for 5 seconds due to our mock.
  const sync1 = service.syncPendingReadings();

  // 2. Fast-forward time slightly (e.g., 100ms) to ensure the lock is set
  // but the request hasn't completed.
  await vi.advanceTimersByTimeAsync(100);

  // 3. Attempt a second sync immediately
  const sync2 = service.syncPendingReadings();

  // 4. Fast-forward past the 5-second delay to let the first sync finish
  await vi.advanceTimersByTimeAsync(5000);

  await Promise.all([sync1, sync2]);

  // 5. ASSERT: The API should have been called exactly ONCE.
```

```
    // If the mutex failed, this would be 2.
    expect(apiMock.post).toHaveBeenCalledTimes(1);
  });
});
```

7.3 Testing the Offline Queue Transition

Validating the transition from Offline to Online requires mocking the Capacitor Network plugin's event emission.

Vitest Implementation:

TypeScript

```
import { Network } from '@capacitor/network';

// 1. Mock Capacitor Network using Factory Pattern
vi.mock('@capacitor/network', () => ({
  Network: {
    addListener: vi.fn(),
    getStatus: vi.fn()
  }
}));

it('should trigger sync when network status changes to connected', () => {
  // Capture the callback passed to addListener
  let networkHandler: any;
  vi.mocked(Network.addListener).mockImplementation((event, handler) => {
    if (event === 'networkStatusChange') {
      networkHandler = handler;
    }
    return Promise.resolve({ remove: vi.fn() });
  });

  // Initialize service (which subscribes to network)
  const service = TestBed.inject(ReadingsService);
  const syncSpy = vi.spyOn(service, 'syncPendingReadings');

  // Simulate network coming online
  expect(networkHandler).toBeDefined();
```

```
 // Trigger the captured callback
 networkHandler({ connected: true, connectionType: 'wifi' });

 expect(syncSpy).toHaveBeenCalled();
});
```

8. Playwright Mobile Configuration Strategy

To validate the "Mobile-First" design of Diabetify, the E2E testing strategy must accurately simulate mobile constraints. The following playwright.config.ts configuration provides a comprehensive setup.

8.1 Recommended Configuration

This configuration introduces specific "Mobile" projects using exact viewport dimensions and User Agent strings. It enforces trace capture on failure, facilitating debugging in CI environments.

TypeScript

```typescript
import { defineConfig, devices } from '@playwright/test';

export default defineConfig({
 testDir: './tests',
 fullyParallel: true,
 forbidOnly:!!process.env.CI,
 retries: process.env.CI? 2 : 0,
 workers: process.env.CI? 1 : undefined,
 reporter: 'html',
 use: {
  baseURL: 'http://localhost:4200',
  trace: 'on-first-retry',
  // Global mobile settings (can be overridden per project)
  hasTouch: true,
  isMobile: true,
 },

 projects: },
  },
  /* Mobile Viewports */
  {
    name: 'Mobile Chrome (Pixel 5)',
    use: {
     ...devices['Pixel 5'],
      // Explicitly enable touch if not set by device descriptor
      hasTouch: true,
      isMobile: true
    },
  },
  {
    name: 'Mobile Safari (iPhone 12)',
    use: {
     ...devices['iPhone 12'],
      hasTouch: true,
      isMobile: true
    },
  },

  /* Low-Res Android Device (common in target demographic) */
  {
    name: 'Low-End Android',
    use: {
     ...devices['Pixel 5'],
      viewport: { width: 360, height: 640 }, // Small viewport
      deviceScaleFactor: 1.0, // Lower pixel density
    }
  }
 ],
});
```

8.2 Strategic Insight: The "Low-End Android" Profile

The inclusion of a "Low-End Android" profile (360x640) is a targeted architectural decision. Diabetes management applications are often used by elderly patients or populations with older hardware. Testing against this compact viewport ensures that critical data tables (like the glucose log) and charts do not suffer from clipping or layout breakage, guaranteeing accessibility for the widest possible user base.

9. Conclusion

The modernization of the Diabetify application to Angular 21 offers substantial benefits in performance and maintainability but demands a precise execution strategy for its testing infrastructure. By adopting **AnalogJS** for component testing, employing **Factory Mocks** to circumvent Capacitor 6 proxies, and leveraging **Device Descriptors** in Playwright, the development team can resolve current configuration conflicts and establish a robust quality assurance pipeline. The proposed **Mutex Testing Strategy** provides the necessary mathematical rigor to validate the application's "Offline-First" promise, ensuring patient data integrity remains compromised. This roadmap transitions Diabetify from a state of technological flux to a stable, scalable hybrid mobile platform.

Obras citadas

1. diabetactic-diabetify-8a5edab282632443.txt
2. Vitest in Angular 21: What's new and how to migrate?, fecha de acceso: diciembre 16, 2025, https://angular.schule/blog/2025-11-migrate-to-vitest/
3. Announcing Angular v21, fecha de acceso: diciembre 16, 2025, https://blog.angular.dev/announcing-angular-v21-57946c34f14b
4. Using Vitest with An Angular Project - Analog.js, fecha de acceso: diciembre 16, 2025, https://analogjs.org/docs/features/testing/vitest
5. Exception when using html reporter with vitest@4.0.15 · Issue #32054 - GitHub, fecha de acceso: diciembre 16, 2025, https://github.com/Angular/Angular-cli/issues/32054
6. Mocking | Guide - Vitest, fecha de acceso: diciembre 16, 2025, https://vitest.dev/guide/mocking
7. Migration Guide - Vitest, fecha de acceso: diciembre 16, 2025, https://vitest.dev/guide/migration.html
8. Mocking Plugins | Capacitor Documentation, fecha de acceso: diciembre 16, 2025, https://capacitorjs.com/docs/guides/mocking-plugins
9. Show how to mock a Capacitor plugin in a Jest based unit test - GitHub, fecha de acceso: diciembre 16, 2025, https://github.com/ionic-team/cap-plugin-mock-jest
10. ng2-charts - NPM, fecha de acceso: diciembre 16, 2025, https://www.npmjs.com/package/ng2-charts