NITOR.BE

REALISATIEDOCUMENT

STAGE 23/02/2026 - 22/05/2026

SENNE FABRI

THOMAS MORE GEEL

Inhoudsopgave

Inleiding 2
Analyse 2
State management 2
Offline storage 2
Authenticatie 3
Kaarten 3
Development workflow 3
Coding guidelines 4
Clean code 4
Coding guidelines 4
Code Review 5
State management en architectuur 6
Scheiding van logica en UI 6
Persistentie met MMKV 8
Reactiviteit en Automatische UI-Updates 8
Cartografie met Mapbox 8
Focus op gebruiksvriendelijkheid en UX 8
Technische implementatie en interactie 9
Leercurve 11
Buiten de grenzen van de applicatie 11
Bestandsbeheer en de App Sandbox 11
De systeem-hiërarchie bij NFC 12
De fotokiezer 12
Persoonlijke groei 13
Van 'happy path' naar robuuste UX 13
Van snelle code naar duurzame kwaliteit 13
Besluit 13
Literatuurlijst 14
Verantwoording gebruik Artificiële Intelligentie 14
Bijlagen 14
Coding guidelines 14

Inleiding

Tijdens mijn laatste jaar in de richting Toegepaste Informatica – Digital Innovation loop ik stage bij Nitor, waar ik de opdracht heb gekregen om een Proof of Concept (POC) te ontwikkelen voor een mobiele applicatie binnen de bouwsector. Voor meer informatie omtrent de achtergrond van mijn stagebedrijf of stageopdracht, verwijs ik graag naar mijn “Project Plan”-document.

Dit document beschrijft in de eerste plaats de technische realisatie van mijn stageopdracht. Om de context te schetsen, herhaal ik in het hoofdstuk “Analyse” kort de belangrijkste elementen uit de analysefase. Daarna bespreek ik de kern van de opdracht en hoe ik dit heb uitgewerkt, waarna ik dieper inga op de architectuur en de technische hoogtepunten. Verder reflecteer ik op hoe dit project me heeft geleerd om breder te denken dan enkel de code van de applicatie. Tot slot bundel ik mijn ervaringen in een samenvattend besluit.

Analyse

Vooraleer ik aan de eigenlijke realisatie begonnen ben, heb ik drie weken de tijd genomen om de opdracht te analyseren. In deze periode heb ik enerzijds een design analyse gemaakt, waarin ik nagedacht heb over use cases en de datastructuur, alsook prototypes gemaakt heb.

Verder heb ik ook een technische analyse uitgevoerd. Deze was meer gefocust op de tools en werkwijzen die ik zou gebruiken. Het was belangrijk om te experimenteren met nieuwe technologieën en tegelijkertijd voldoende aan te sluiten bij de werkmethodes van Nitor. Nog belangrijker was uiteraard dat de juiste tools worden gekozen voor de aard en omvang van het specifieke POC.

In dit onderdeel benoem ik de belangrijkste zaken uit mijn technische analyse. Voor de volledige analyse, verwijs ik opnieuw graag naar mijn “Project Plan”-document.

State management

Om van een simpel iets naar een professionele app te gaan, is het belangrijk om aan state management te doen. Bij Nitor wordt hiervoor – en voor nog wat andere zaken – Redux gebruikt. Redux is jaren de absolute standaard egeweest om te gebruiken voor complexe apps. Het heeft echter een relatief hoge leercurve en vereist best veel tijd om boilerplate code te schrijven.

Daarom heb ik gebruik gemaakt van een moderner alternatief: Zustand. Deze tool wordt de laatste jaren steeds meer gebruikt. Volgens mijn research zou Zustand zelfs geschikt moeten zijn voor complexe apps. Voor deze POC is het zeker voldoende. Redux zou zelfs wat overkill zijn.

Offline storage

Jarenlang was AsyncStorage de onbetwiste standaard voor data-persistentie in React Native, en logischerwijs ook de vertrouwde keuze binnen Nitor. Hoewel deze tool voor lichte toepassingen nog steeds uitstekend voldoet, kent het architecturale beperkingen bij intensiever gebruik. AsyncStorage werkt, zoals de naam suggereert, asynchroon en communiceert via de traditionele React Native 'Bridge'. Dit betekent dat data telkens geserialiseerd moet worden en JavaScript moet wachten op de native kant. Bij het inladen of wegschrijven van grotere datasets, creëert deze vertaalslag over de bridge een bottleneck, wat kan leiden tot merkbare vertragingen in de gebruikersinterface.

Om een vlotte en betrouwbare offline-first ervaring te garanderen, heb ik gekozen voor een modernere oplossing: `react-native-mmkv`. Deze bibliotheek is onder de motorkap ontwikkeld door Tencent (het bedrijf achter WeChat, een app met een miljard gebruikers) en is ontworpen voor robuust enterprise-gebruik. In plaats van de oude bridge te gebruiken, maakt MMKV rechtstreeks

gebruik van de JavaScript Interface (JSI) en C++. Dit stelt de applicatie in staat om data direct en synchroon uit het geheugen te lezen en te schrijven, zonder wachttijd. Uit benchmarks blijkt dat MMKV hierdoor tot wel 30 keer sneller presteert dan AsyncStorage.

Mijn collega was steeds sceptisch om tools die ze al jaren gebruiken, en die vroeger ook de beste waren, te vervangen. Toch was ze onder de indruk over mijn research resultaten betreft MMKV. Tijdens mijn laatste stagemaand heeft ze het namelijk zelf onderzocht, en een project van Nitor gerefactored van AsyncStorage naar MMKV omwille van performanceproblemen.

Authenticatie

Een technische eis was gebruik maken van minstens 1 identity provider zoals Google of Meta. Supabase en firebase zijn twee opties die het inloggen met Google/Meta super makkelijk maken zonder dat ik zelf een backend hoef te schrijven.

Firebase is gemaakt door Google. Het zit helemaal in haar eigen ecosysteem en gebruikt onder andere google cloud. Het is overigens een *NoSQL* database, wat voor complexe data vaak rommelig wordt.

Een andere mogelijkheid is Supabase. Het is een Open-source alternatief. Het heeft perfecte TypeScript integratie: aangezien het automatisch de TypeScript types van de database kan genereren, wat naadloos aansluit bij mijn .tsx bestanden. Daarom heb ik Supabase gebruikt.

Kaarten

Voor het weergeven van kaarten (nodig voor het manueel aanpassen van de locatie) zijn er 2 grote opties. Google Maps en Mapbox. Het makkelijkste om te integreren is Google maps. Helaas is deze API niet goed genoeg voor mijn noden. Mapbox vult hier de gaten op, maar heeft een steilere leercurve.

Zo heeft Google maps geen functionaliteiten om kaarten te downloaden. Dit strijkt de “Offline” eis tegen de haren in. Mapbox daarentegen, heeft ingebouwde functionaliteiten voor het downloaden van kaarten.

Verder heeft Google maps minder styling opties dan Mapbox om het uiterlijk van de kaart te personaliseren. Ook heeft Mapbox (dankzij OpenStreetMap-data) meer kennis van bv. landelijke gebieden of nieuwe industrieterreinen.

Ik heb dus gebruik gemaakt van Mapbox.

Development workflow

Zoals beschreven in het “Project Plan”-document zal ik gebruik maken van het Expo framework. Nitor heeft hier nog geen ervaring mee. Ze werken met “Bare React Native”. Dit omdat frameworks zoals Expo enkele jaren geleden niet enterprice proof genoeg ware voor complexe apps.
De laatste jaren is Expo echter sterk verbeterd. Vandaag de dag is het framework voor de meeste toepassingen dan ook zeer geschikt. Daarom was ervoor gekozen dat ik wel gebruik zou maken van Expo. Dit geeft Nitor een interessante kans om te evalueren of Expo misschien toch iets kan betekenen voor het bedrijf.

Een enorme troef van Expo is hun zeer gebruiksvriendelijke development workflow. Ze bieden, onder andere, namelijk een “Expo Go” app die je rechtstreeks van de Play Store kunt installeren op je mobiele apparaat. Zo kun je zeer eenvoudig je telefoon omtoveren tot een live weergave van je app.

De manier hoe dit werkt is dat de meest gebruikte bibliotheken ingebakken zitten in deze app. Zo kan de “Expo Go” app er perfect mee overweg als je ze gebruikt. Uiteraard zitten wel niet alle bestaande

bibliotheken ingebakken in deze build. Zo is bijvoorbeeld Mapbox een zware native bibliotheek die er niet mee samenwerkt. De standaard Expo Go app bevat de C++ en Java code voor Mapbox simpelweg niet. De app zou direct crashen.

Dit zorgde ervoor dat ik niet via Expo Go kon ontwikkelen. Uiteraard vormde dat geen probleem aangezien ik via Expo prebuilds kon werken. Je kan je eigen custom “Expo Go” app maken met alles ingebakken dat je nodig hebt. Ik kende dit nog niet, maar zag het als een kans om bij te leren!

Coding guidelines

Tot slot vermeld ik nog graag mijn coding guidelines. Zoals beschreven in mijn “Project Plan”-document, was het voor mijn opdrachtgever belangrijk dat ik aandacht besteedde aan “Clean Code”. Het was dan ook een eis om coding guidelines te definiëren en automatisch af te handelen.

In de eerste plaats heb ik gebruik gemaakt van ESLint en Prettier. Eslint controleert of de code aan de afgesproken regels voldoet (bijv. geen ongebruikte variabelen). Prettier daarentegen, formatteert de code automatisch zodat alles er hetzelfde uitziet (spaties, quotes, etc.).

In het hoofdstuk “Clean code”, vermeld ik nog in meer detail hoe ik met coding guidelines omgegaan ben.

Clean code

Waar sommige jaargenoten misschien een applicatie met veel en ingewikkelde functionaliteiten hebben gebouwd, lag de hoofdfocus bij mijn opdracht ergens anders.

Zo heb ik bijvoorbeeld mogen ervaren dat, in een echt softwarebedrijf, het niet louter gaat om functionaliteiten, maar dat er enorm veel aandacht wordt besteed aan gebruikerservaring, code kwaliteit en andere details. Zo werd ik gepusht om traag te durven werken zodat ik rekening kon houden met details. Ook heb ik een unieke blik gekregen op hoe het eraan toe gaat bij klanten en partners.

Met code kwaliteit heb ik op verschillende vlakken rekening moeten houden. Voorbeelden hiervan zijn het definiëren en afhandelen van coding guidelines alsook de code review.

Coding guidelines

In de realisatiefase, tijdens de eerste sprint heb ik coding guidelines opgesteld. Deze heb ik afgestemd met het team van developers bij Nitor. Vervolgens heb ik hier een verslag van gemaakt. Dit verslag kan worden teruggevonden in de bijlagen aan het einde van dit document.

Vervolgens heb ik mijn IDE ingesteld om deze richtlijnen zo veel mogelijk automatisch af te handelen. Hiervoor maak ik gebruik van de “ESLint” en “Prettier” extensies. Ik heb voor een groot deel gebruik kunnen maken van de default configuratie. De configuratie die ik dan nog moest voorzien was vrij beperkt en gericht om perfect samen te werken met zaken zoals Expo en aanvullend om nog wat meer van mijn coding guidelines te bewaken. De codevoorbeelden hieronder zijn enkele van mijn configuratiebestanden.

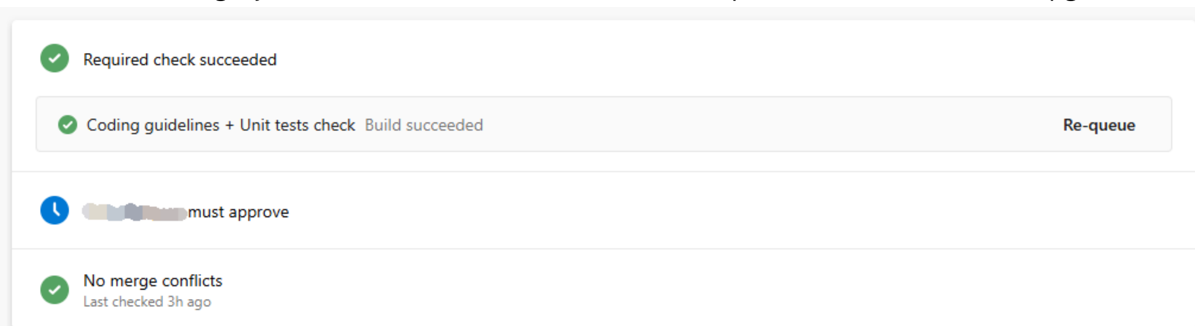
.eslintrc.js

```js
module.exports = {
  extends: ['expo', 'prettier'],
  plugins: ['prettier'],
  rules: {
    'prettier/prettier': 'error',
    'no-unused-vars': 'warn',
    'react-native/no-inline-styles': 'warn',
    'import/namespace': 'off',
  },
};
```

.prettierrc

```json
{
  "singleQuote": true,
  "trailingComma": "es5",
  "tabWidth": 2,
  "semi": true,
  "printWidth": 100,
  "endOfLine": "lf",
  "plugins": ["@trivago/prettier-plugin-sort-imports"],
  "importOrder": [
    "^react$",
    "^react-native$",
    "^expo",
    "<THIRD_PARTY_MODULES>",
    "^@/(.*)$",
    "^[./]"
  ],
  "importOrderSeparation": true,
  "importOrderSortSpecifiers": true
}
```

Om het hanteren van de coding guidelines af te dwingen, heb ik een pipeline gemaakt. Deze is multifunctioneel. Wanneer ik een Pull Request (PR) aanmaak naar een releasebranch, dan zorgt de pipeline er enerzijds voor dat Prettier en ESLint checken of aan alle coding guidelines voldaan wordt. Anderzijds worden ook alle aanwezige Unit Tests uitgevoerd. Als minstens een van deze zaken faalt, dan is het niet mogelijk om de PR af te ronden, en moeten de problemen eerst worden opgelost.



Code Review

Als laatste stap in het proces om code kwaliteit te verzekeren is er de code review. Bij Nitor wordt hier heel veel aandacht aan besteed. Telkens wanneer er een PR wordt aangemaakt moet deze door het hele developer team worden goedgekeurd. Iedereen neemt dan de tijd om door alle nieuwe code te gaan en opmerkingen te plaatsen op zaken die niet helemaal duidelijk zijn, of mogelijks efficiënter zouden kunnen.

Om dit proces te vergemakkelijken, zorgde ik bij elke PR voor een duidelijke beschrijving die eventuele belangrijke info communiceert. Telkens voegde ik bij de PR relevante screenshots van de app toe, en zorgde ik voor een release build zodat de app effectief en grondig getest kon worden.

Door deze intensieve controle kwamen er vaak nog details naar boven die mij tijdens het ontwikkelen niet waren opgevallen.

State management en architectuur

In het verlengde van de sterke focus op clean code en een onderhoudbare architectuur, was de aanpak rondom state management (het beheren van de interne status van de applicatie) een cruciaal onderdeel van de realisatiefase. Waar complexe applicaties vaak verstrikt raken in een web van doorgegeven eigenschappen of zware frameworks, werd er voor deze Proof of Concept binnen de bouwsector bewust gezocht naar een gestroomlijnde oplossing.

De keuze is hierbij gevallen op Zustand. In tegenstelling tot traditionele, zwaardere oplossingen zoals Redux, blinkt Zustand uit in zijn eenvoud en minimalisme. Het dwingt geen onnodige boilerplate (herhalende standaardcode) af, maar biedt wel de robuustheid die nodig is voor een professionele applicatie. Dit sloot perfect aan bij de filosofie van het Nitor team: houd de code zo leesbaar en efficiënt mogelijk.

Scheiding van logica en UI

Door Zustand te gebruiken, kon ik de bedrijfslogica volledig lostrekken van de visuele componenten. Ik heb specifieke, afgebakende *stores* opgebouwd, zoals een incidentStore voor het beheren van de incidenten en een toastStore voor de globale notificaties.

Hierdoor werden de React-componenten zelf aanzienlijk "schoner". Ze hoefden zich enkel nog bezig te houden met het correct renderen van de data en het doorgeven van gebruikersinteracties, zonder zelf verantwoordelijk te zijn voor complexe data-manipulaties. Een voorbeeld van hoe zo'n store er uitziet, is hieronder weergegeven:

incidentStore.ts

```ts
import { create } from 'zustand';
import { StateStorage, createJSONStorage, persist } from 'zustand/middleware';

import { storage } from '../lib/storage';
import { MOCK_INCIDENTS } from '../mocks/incidents.mock';
import { Incident } from '../types/database';

const zustandStorage: StateStorage = {
  setItem: (name, value) => storage.set(name, value),
  getItem: (name) => storage.getString(name) ?? null,
  removeItem: (name) => storage.remove(name),
};

interface IncidentState {
  incidents: Incident[];
  addIncident: (incident: Incident) => void;
  updateIncident: (updatedIncident: Incident) => void;
  seedIncidents: () => void;
}

export const useIncidentStore = create<IncidentState>()(
  persist(
    (set, get) => ({
      incidents: [],

      addIncident: (incident) => set((state) => ({ incidents: [...state.incidents,
incident] })),

      updateIncident: (updatedIncident) =>
        set((state) => ({
          incidents: state.incidents.map((incident) =>
            incident.id === updatedIncident.id ? updatedIncident : incident
          ),
        })),

      seedIncidents: () => {
        const currentIncidents = get().incidents;
        if (currentIncidents.length === 0) {
          set({ incidents: MOCK_INCIDENTS });
        }
      },
    }),
    {
      name: 'incident-storage',
      storage: createJSONStorage(() => zustandStorage),
    }
  )
);
```

State management is niet het enige aspect waar ik bij de architectuur heb stil gestaan. Bij het implementeren van de NFC-functionaliteit bijvoorbeeld, stond het principe van Separation of Concerns ook centraal, geheel in lijn met de Clean Code richtlijnen. In eerste instantie is het

verleidelijk om de logica voor het uitlezen van de chip direct in het visuele component te plaatsen. Echter, om de code schaalbaar en onderhoudbaar te houden, is er bewust gekozen voor een strikte scheiding.

Alle directe communicatie met de hardware van de telefoon en het decoderen van de binaire payloads van de tags is ondergebracht in een geïsoleerde nfcService. Deze service levert pure, gestructureerde data af. Het React-component (nfc.tsx) is vervolgens uitsluitend verantwoordelijk voor de presentatie: het tonen van de juiste vertalingen, het aansturen van de laad-animaties en het afhandelen van de navigatie. Deze architectuur voorkomt niet alleen race conditions bij het inladen van taalbestanden, maar maakt de code ook veel beter te begrijpen.

Persistentie met MMKV

Hoewel Zustand standaard de state in het geheugen beheert, zou de data bij een app-herstart of een crash verloren gaan. Voor een applicatie die in de bouwsector wordt gebruikt, is betrouwbaarheid en offline-beschikbaarheid echter een harde eis. Om dit op te lossen, heb ik de persist middleware van Zustand geïntegreerd in combinatie met MMKV.

Door een 'custom storage bridge' te schrijven, fungeert MMKV als de persistente laag die de state van de stores direct naar het flashgeheugen van het toestel schrijft. Hierdoor blijft de gebruikerservaring vloeiend, zelfs bij het verwerken van grotere datasets of bijlagen.

Reactiviteit en Automatische UI-Updates

Een van de grootste voordelen van deze setup is de naadloze reactiviteit binnen de gehele applicatie. Zustand maakt gebruik van het observer-pattern, wat betekent dat componenten zich "abonneren" op specifieke delen van de state via custom hooks.

Zodra een actie wordt aangeroepen, bijvoorbeeld het toevoegen van een nieuwe opvolging aan een incident, wordt de state in de store bijgewerkt. Op dat exacte moment detecteert Zustand welke componenten die specifieke data gebruiken en triggert de app automatisch een re-render van uitsluitend die onderdelen. Voor mij als developer betekent dit dat ik geen handmatige "refresh"-logica of complexe event-listeners hoef te schrijven. De data fungeert als de Single Source of Truth: of een gebruiker nu een wijziging doorvoert in een detailvenster of een status aanpast op de kaart, de rest van de applicatie is onmiddellijk en automatisch up-to-date.

Cartografie met Mapbox

Voor de visuele weergave van incidenten en het handmatig kunnen aanpassen van locaties op een bouwterrein, was de integratie van een betrouwbaar kaartensysteem essentieel. Zoals beschreven in de analysefase heb ik hiervoor Mapbox gebruikt.

Focus op gebruiksvriendelijkheid en UX

Bij de integratie van de kaart is er nadrukkelijk nagedacht over de gebruikerservaring (UX). Een veelvoorkomende frustratie bij mobiele applicaties is dat het scrollen door een

pagina onbedoeld resulteert in het in- of uitzoomen op een kaart, of omgekeerd. Om dit te voorkomen, is de kaartsectie in de leesmodus standaard statisch gemaakt. Pas wanneer de gebruiker expliciet op 'Bewerken' klikt, wordt de kaart bedienbaar. Om conflicten te vermijden, wordt op dat moment het scrollen van de volledige onderliggende pagina tijdelijk vergrendeld. Zodra de bewerkingsmodus wordt afgesloten, keert de pagina direct terug naar zijn normale, vloeiende scrollgedrag.

Daarnaast is er een bewuste keuze gemaakt voor het standaard kaarttype. In de bouwsector veranderen locaties enorm snel. Nieuwe industrieterreinen of werven zijn op een standaard wegenkaart vaak nog een blinde vlek. Daarom opent de kaart standaard in satellietweergave, wat veel meer visuele herkenningspunten biedt om accuraat te navigeren op een bouwplaats. Omdat vrijheid voor de gebruiker belangrijk is, is er ook een toggle voorzien waarmee direct geschakeld kan worden naar de standaard wegenkaart. Zo kan de gebruiker, afhankelijk van de exacte situatie en persoonlijke voorkeur, altijd de meest overzichtelijke weergave kiezen.

Technische implementatie en interactie

De implementatie is gerealiseerd met de @rnmapbox/maps bibliotheek. Het hart van deze functionaliteit bevindt zich in de MapSection-component. Hierbij wordt gebruikgemaakt van een MapView die gekoppeld is aan de gekozen stijl, wat gebruikers een realistisch beeld geeft van de huidige situatie op de werf.

MapSection.ts

```
<View style={styles.mapWrapper}>
  <MapView
    style={styles.map}
    styleURL={
      isSatellite
        ? "mapbox://styles/mapbox/standard-satellite"
        : "mapbox://styles/mapbox/streets-v12"
    }
    projection="globe"
    scaleBarEnabled={false}
    logoPosition={
      Platform.OS === "android" ? { bottom: 3, left: 3 } : undefined
    }
    attributionPosition={
      Platform.OS === "android" ? { bottom: 3, right: 3 } : undefined
    }
    scrollEnabled={isMapActive}
    zoomEnabled={isMapActive}
    pitchEnabled={isMapActive}
    rotateEnabled={isMapActive}
    onPress={handleMapPress}
    onRegionWillChange={() => setIsMoving(true)}
    onRegionDidChange={() => setIsMoving(false)}
  >
    <Camera
      centerCoordinate={[longitude, latitude]}
      zoomLevel={16}
      pitch={45}
      animationMode="flyTo"
      animationDuration={1000}
    />

    {hasAssignedLocation && (
      <PointAnnotation
        id="incident-marker"
        coordinate={[longitude, latitude]}
        anchor={{ x: 0.5, y: 1 }}
      >
        <View style={styles.markerContainer}>
          <Ionicons name="location" size={32} color={colors.errorText} />
        </View>
      </PointAnnotation>
    )}
  </MapView>

  {isMapActive && !isMoving && (
    <View style={styles.instructionOverlay}>
      <Text style={styles.instructionText}>
        {isAssignMode ? t("tapMapToAssign") : t("tapMapToEdit")}
      </Text>
    </View>
  )}
</View>;
```

De kern van het weergeven en wijzigen van de locatie werkt via een samenspel tussen de Camera en PointAnnotation componenten.

- **Locatie-weergave:** Bij het openen van een incident zorgt de Camera component voor een vloeiende "flyTo" animatie naar de exacte coördinaten van het incident. De locatie wordt gemarkeerd door een PointAnnotation met een herkenbare icoon-marker.
- **Locatie-wijziging:** Om de nauwkeurigheid te waarborgen, heb ik een interactieve modus gebouwd. Wanneer de gebruiker de 'Bewerken'-stand activeert, wordt de kaart ontgrendeld. Door simpelweg op de gewenste plek op de kaart te tikken, vangt de onPress handler de nieuwe Longitude en Latitude op. Deze coördinaten worden vervolgens direct teruggekoppeld naar Zustand, waarna de marker zich onmiddellijk verplaatst naar de nieuwe positie.

Leercurve

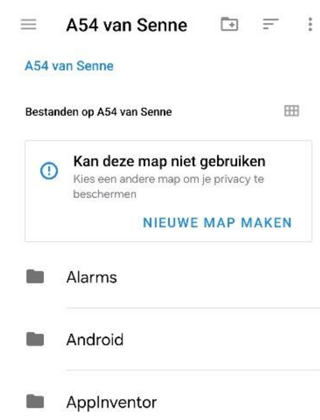
Waar ik vooraf verwachtte dat de technische integratie van de Mapbox API de grootste uitdaging zou zijn, bleek de werkelijke leercurve ergens anders te liggen: het meedogenloos bewaken van de UX en het proactief voorkomen van bugs. Het inbouwen van een kaart viel mee. Mapbox is tot heel veel in staat. Meer geavanceerde zaken waarin Mapbox uitblinkt hebben vaak een redelijke leercurve, maar ik had enkel de basis nodig. De moeilijkheid bleek uiteindelijk te liggen in ervoor zorgen dat deze feilloos samenwerkt met de rest van het formulier. Dat vereist een scherp oog voor detail.

Een goed voorbeeld hiervan is de wisselwerking tussen de 'readonly' weergave en de interactieve bewerkingsmodus. Als een gebruiker de algemene bewerkingsmodus van de pagina annuleert via het formulier, was het cruciaal om ervoor te zorgen dat ook de interne state van de Mapbox-component perfect werd gereset. Zonder deze verfijning zou de kaart onbedoeld interactief kunnen blijven op de achtergrond.

Buiten de grenzen van de applicatie

Tijdens de ontwikkeling van deze applicatie besefte ik al snel dat een mobiele app nooit in een vacuüm draait. Waar je bij traditionele webapplicaties voornamelijk te maken hebt met de browser, ben je als mobiele applicatie te gast op het besturingssysteem van de gebruiker. Dit vereist een fundamenteel andere manier van denken. Je moet buiten de veilige grenzen van je eigen code treden en communiceren met Android of iOS over hardware, bestanden en permissies.

Bestandsbeheer en de App Sandbox



De eerste grote uitdaging waarbij ik met het besturingssysteem moest communiceren, was bestandsbeheer. In de applicatie moeten gebruikers bijlagen van opvolgingen (zoals PDF-documenten) en foto's van incidenten kunnen opslaan op hun apparaat.

Moderne besturingssystemen hanteren een strikt sandboxing-principe. Een app mag standaard alleen in zijn eigen, afgeschermde mapje kijken en schrijven. Om een foto daadwerkelijk in de publieke galerij van de telefoon op te slaan, of een document in de algemene 'Downloads' map te plaatsen, moest ik gebruikmaken van specifieke

OS-API's. Dit vereist het zorgvuldig aanroepen van de juiste File System en Media Library protocollen, waarbij de complexiteit schuilt in het feit dat iOS en Android dit onder de motorkap op compleet verschillende manieren afhandelen.

De systeem-hiërarchie bij NFC

Een andere uitdaging ontstond bij de integratie van de NFC-scanner. De complexiteit zat hierbij niet in het eenmalig uitlezen van een tag, maar in het beheren van de component lifecycle in combinatie met de hardware. Een NFC-chip is een gedeelde bron op een smartphone. Als applicatie moet je precies aangeven wanneer je deze claimt en weer vrijgeeft.

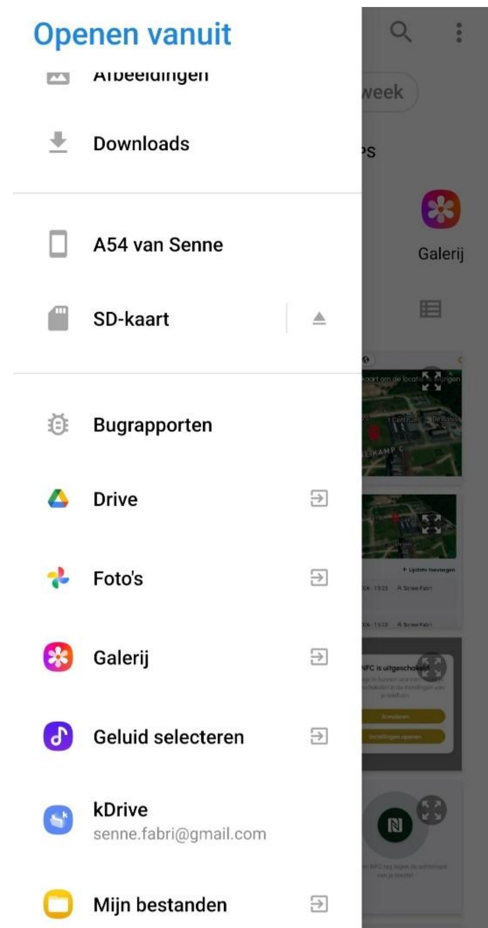
Het testen bracht interessante edge cases aan het licht. Wat gebeurt er als een gebruiker de app minimaliseert, via de instellingen van de telefoon de NFC-antenne uitschakelt, en vervolgens terugkeert naar de app? Door slim gebruik te maken van achtergrond-intervallen en useRef-hooks, is de applicatie nu in staat om dit soort externe hardware-wijzigingen direct te detecteren en de gebruikersinterface hier naadloos op aan te passen, zonder dat de app vastloopt.

Daarnaast botste ik op een klassiek platform-specifiek probleem op Android. Wanneer de applicatie succesvol een tag had gelezen en tijdelijk stopte met actief scannen, probeerde het Android-besturingssysteem de regie over te nemen door de standaard systeem-app voor NFC te openen bij een volgende scan. Dit verstoorde de gebruikerservaring aanzienlijk. Om dit op te lossen, heb ik gebruikgemaakt van Foreground Dispatch. Hiermee dwingt de applicatie af dat, zolang het NFC-scherm geopend is, alle hardware-scans passief worden afgevangen door onze app, waardoor het besturingssysteem niet onbedoeld kan ingrijpen.

Het robuust afhandelen van dit soort externe interrupts was een cruciale eerste stap in het leren denken als een volwaardige mobiele developer.

De fotokiezer

De meest leerzame iteratie op het gebied van OS-interactie en UX vond plaats bij het toevoegen van foto's aan een incident. Initieel maakte ik gebruik van de traditionele methode. De app vroeg de gebruiker om toestemming tot de volledige mediabibliotheek.



Op moderne Android-versies (vanaf Android 13) resulteert dit echter in een complexe en frustrerende gebruikerservaring. Gebruikers krijgen een intimiderende pop-up en kunnen ervoor kiezen om slechts "beperkte toegang" te geven tot een aantal geselecteerde foto's. Als de gebruiker later een andere foto wil toevoegen, moest de app telkens opnieuw ingrijpen in die beperkte permissies, wat leidde tot een verwarrende en ongebruiksvriendelijke flow.

Om dit op te lossen, heb ik de architectuur omgegooid en ben ik gebruik gaan maken van de native System Photo Picker. In plaats van permissies te vragen om zelf in de bestanden te zoeken, roept de app nu een interface op die als een overlay door het besturingssysteem zelf wordt gegenereerd. De gebruiker bladert in zijn eigen, vertrouwde galerij, selecteert een foto, en het OS geeft

uitsluitend dat ene specifieke bestand door aan de app. Het resultaat is een enorme overwinning voor de UX: de gebruiker wordt niet meer lastiggevallen met enge permissie-dialogen, de privacy is 100% gewaarborgd omdat de app geen toegang heeft tot de rest van de galerij, en de flow voelt compleet natuurlijk en "native" aan.

Persoonlijke groei

Mijn stage heeft gezorgd voor een belangrijke paradigmaverschuiving in mijn denkwijze. Ik leerde dat de beste gebruikerservaring vaak ontstaat door juist niet alles zelf te willen bouwen of forceren binnen de app, maar door slim gebruik te maken van de tools die het besturingssysteem aanbiedt.

Van 'happy path' naar robuuste UX

De verschuiving van "Hoe schrijf ik code om alle foto's uit te lezen?" naar "Hoe ontwerp ik een flow waarbij de gebruiker zich veilig voelt en geen fouten kan maken?" typeert mijn groei tijdens dit project. Ik denk nu veel breder na over UX, waarbij ik de systeem-dialogen, permissies en privacy-overwegingen niet langer zie als obstakels, maar als integrale onderdelen van een professionele app-ervaring.

Het afvangen van edge cases, zoals de bugs bij het implementeren van de kaart, heeft mijn denkwijze als developer ook veranderd. Ik test niet alleen of een 'happy path' functionaliteit werkt, maar denk meer dan ooit actief na over hoe onverwachte navigatie of component-lifecycles tot UX-problemen kunnen leiden. Het heeft me geleerd dat de ware kwaliteit van een applicatie vaak schuilt in deze verfijnde, onderliggende details die samen een vlekkeloze, bugvrije ervaring garanderen.

Van snelle code naar duurzame kwaliteit

Waar ik op school vaak gefocust was op het zo snel mogelijk werkend krijgen van functionaliteiten, leerde ik nu om al vroeg in de sprint te vertragen en langer stil te staan bij details. Door het strikt hanteren van coding guidelines, pipelines en intensieve code reviews (waar op school minder tijd voor was), leerde ik dat de ware kwaliteit van software in de onderhoudbaarheid en de details zit.

Besluit

Mijn stageopdracht omvatte hoofdzakelijk het realiseren van een Proof of Concept voor de bouwsector, een project dat fundamenteel over meer ging dan enkel het schrijven van code. In plaats van een eenzijdige focus op het zo snel mogelijk opleveren van een breed scala aan functionaliteiten, stond dit traject volledig in het teken van softwarekwaliteit en een schaalbare architectuur. De nadruk lag op het creëren van een applicatie die in een veeleisende omgeving niet alleen betrouwbaar, maar ook veilig, onderhoudbaar en intuïtief in gebruik is.

Om dit niveau van kwaliteit te bereiken, zijn er tijdens de realisatiefase weloverwogen technische keuzes gemaakt. Van het opzetten van een gestroomlijnd, offline-first databeheer en de integratie van interactieve kaarten, tot de feilloze afhandeling van hardware-interrupts bij de NFC-scanner. Een van de meest waardevolle lessen hierbij was het leren denken buiten de veilige grenzen van de applicatie zelf. De succesvolle navigatie door de complexe *sandboxes* en permissie-systemen van mobiele besturingssystemen vormde de sleutel tot een app die naadloos en natuurlijk aanvoelt op het apparaat van de gebruiker.

Deze technische beslissingen stonden echter nooit op zichzelf, maar stonden steeds in dienst van de gebruikerservaring (UX). Het meedogenloos bewaken van deze UX werd de rode draad van het project en vereiste een ware paradigmaverschuiving. Het traditionele 'happy path'-denken werd

losgelaten ten gunste van het proactief afvangen van *edge cases*, het integreren van *clean code* principes en het slim inzetten van native besturingssysteem-tools, zoals de fotokiezer.

Terugkijkend vormt deze stageopdracht voor mij de ultieme brug tussen de theorie en de professionele praktijk. Dit project wordt dan ook niet enkel afgesloten met de succesvolle oplevering van een robuuste applicatie, maar markeert voor mij persoonlijk een veel belangrijkere mijlpaal. Het heeft de transformatie in gang gezet van een student naar een professionele software developer met een scherp oog voor detail. Dit traject heeft daarmee een onmisbaar fundament gelegd voor mijn verdere loopbaan.

Literatuurlijst

ESLint. (z.d.). *Find and fix problems in your JavaScript code*. Geraadpleegd op 22 mei 2026, van https://eslint.org/docs/latest/

Expo. (z.d.). *Expo Documentation*. Geraadpleegd op 22 mei 2026, van https://docs.expo.dev/

Mapbox. (z.d.). *Mapbox Documentatie*. Geraadpleegd op 22 mei 2026, van https://docs.mapbox.com/

Meta Platforms, Inc. (z.d.). *React Native*. Geraadpleegd op 22 mei 2026, van https://reactnative.dev/docs/getting-started

mrousavy. (z.d.). *react-native-mmkv*. GitHub. Geraadpleegd op 22 mei 2026, van https://github.com/mrousavy/react-native-mmkv

Supabase. (z.d.). *Supabase Documentation*. Geraadpleegd op 22 mei 2026, van https://supabase.com/docs

Zustand. (z.d.). *Zustand Documentation*. pmndrs. Geraadpleegd op 22 mei 2026, van https://zustand-demo.pmnd.rs/

Verantwoording gebruik Artificiële Intelligentie

Tijdens de totstandkoming van dit realisatiedocument is gebruikgemaakt van artificiële intelligentie. De AI is uitsluitend ingezet als ondersteunend hulpmiddel voor het stroomlijnen van teksten en het structureren van alinea's om de algemene leesbaarheid en professionaliteit van het verslag te verhogen. Alle inhoudelijke analyses, architectuurkeuzes, technische implementaties en persoonlijke reflecties zijn volledig gebaseerd op mijn eigen werk, code en bevindingen tijdens de stageperiode.

Bijlagen

Coding guidelines

Coding guidelines report

Page • Stage • 1 backlink

Dit document beschrijft de coding guidelines, architectuurkeuzes en kwaliteitscontroles voor mijn opdracht. Het doel van deze setup is om een schaalbare, schone en uniforme codebase te garanderen, ongeacht welke developer in de toekomst aan het project werkt.

Om menselijke fouten te minimaliseren en gebruiksgemak te garanderen, is de configuratie zo ingericht dat het zware werk zoveel mogelijk geautomatiseerd wordt door tooling en CI/CD pipelines.

1. Geautomatiseerde Code Opmaak (Tooling)

Ik maak gebruik van **Prettier** als de absolute bron van waarheid voor de opmaak van de code. Ik hoef me zo geen zorgen te maken over spaties, enters of het sorteren van imports. Dit wordt lokaal automatisch afgehandeld bij het opslaan van een bestand (via VS Code) en afgedwongen in de pipeline. Door dat laatste kunnen de guidelines niet genegeerd of vergeten worden.

Onze Prettier configuratie forceert het volgende:

- **Quotes & Komma's:** Enkele aanhalingstekens (`singleQuote: true`) en een trailing comma waar geldig in ES5 (`trailingComma: "es5"`).
- **Regellengte & Inspringen:** Maximaal 100 karakters per regel (`printWidth: 100`) en 2 spaties voor inspringen (`tabWidth: 2`).
- **Line Endings (LF):** Om conflicten tussen Windows en Mac/Linux te voorkomen, dwingt de configuratie af dat alle enters worden opgeslagen als `LF` (Unix-standaard). Dit wordt ondersteund door een `.gitattributes` bestand (`* text=auto eol=lf`).
- **Automatische Import Sortering:** Met behulp van de `@trivago/prettier-plugin-sort-imports` plugin worden imports automatisch gegroepeerd en alfabetisch gesorteerd. De hiërarchie is altijd: `React` ➔ `React Native` ➔ `Expo` ➔ `Externe Libraries` ➔ `Lokale bestanden`. Tussen de externe en lokale imports wordt automatisch een witregel geplaatst voor de leesbaarheid.

2. Kwaliteitscontrole & CI/CD Pipeline

Om te voorkomen dat slordige of foutieve code in de belangrijkste branches (`develop` , `release` , `main`) terechtkomt, hanteer ik een strikte controle via ESLint, Prettier en Azure DevOps.

Hoewel ik lokaal bewust geen gebruik maak van afdwingende pre-commit hooks (zoals Husky) om de ontwikkelvrijheid en snelheid op lokale feature-branches te behouden, fungeert de server als de ultieme poortwachter. Ik ben als developer zelf verantwoordelijk voor het lokaal toepassen van de juiste formattering (bijvoorbeeld via de automatische *format-on-save* in mijn editor, maar de code mag pas gemerged worden als deze slaagt voor de pipeline-check.

Server controle (Azure DevOps Pipeline)

Tijdens een Pull Request naar een van de hoofdbranches treedt de CI/CD pipeline in werking. In de pipeline draait het volgende script:
`npm run lint`(geconfigureerd als: *`eslint . --ext .js,.jsx,.ts,.tsx --max-warnings=0 && prettier --check .`*)

Wat de pipeline onverbiddelijk afdwingt (en een PR direct blokkeert bij falen):

- **100% Correcte opmaak:** Prettier checkt of elk gewijzigd bestand exact aan de styling-regels voldoet. Ontbreekt er ergens een witregel of klopt de inspringing niet? Dan faalt de pipeline.
- **Geen test-logs:** Via onze ESLint Flat Config (`eslint.config.js`) is het gebruik van `console.log` strikt verboden (`"no-console": ["error"]`). Lokale test-logs moeten vóór het aanmaken van een PR verwijderd zijn. Let op: `console.warn` en `console.error` zijn wél toegestaan voor legitieme en noodzakelijke foutafhandeling in productie.
- **Geen waarschuwingen:** Dankzij de `--max-warnings=0` vlag resulteert elke afwijking of ESLint-waarschuwing direct in een gefaalde build. Er is in de PR's geen ruimte voor genegeerde waarschuwingen.

3. Handmatige Richtlijnen (Mijn Verantwoordelijkheid)

Niet alles hoeft (of moet) dichtgetimmerd te worden door de linter. Voor de volgende regels kies ik ervoor om hier tijdens het ontwikkelen en bij code reviews zelf streng op te letten, om zo de ontwikkelvrijheid niet onnodig te beperken:

- **Volgorde in Interfaces/Types:** Bij het definiëren van een data-structuur schrijf ik altijd eerst de verplichte properties, gevolgd door een blok met de optionele properties (aangeduid met een `?`). Dit verbetert de leesbaarheid.
- Ban op het **`any`** **type in Interfaces:** Het gebruik van index signatures zoals `[key: string]: any;` wordt vermeden. Enkel indien het echt nodig is (en verantwoord

kan worden), zal ik any gebruiken.

- Style properties worden onder elkaar gezet. Niet naast elkaar. Dit verhoogt de leesbaarheid.
- Voor pagina's en layouts zal ik default functions gebruiken. Expo router vereist namelijk overal *default exports*, en deze syntax is daarvoor het schoonst en het meest direct.
  Voor kleine interne componenten en helpers zal ik arrow functions gebruiken.
- Bij properties zal ik eerst alle variabelen zetten en daarna alle functies.
- Ik zal de Stylesheet altijd onderaan mijn bestand plaatsen.

4. Naming Conventions

Om de codebase uniform te houden, hanteer ik een strikte set aan naamgevingsconventies:

Variabelen, Types & Structuur

- **Variabelen:** Altijd `camelCase` (dit geldt ook voor default values binnen een interface/type).
- **Types & Interfaces:** Altijd `PascalCase`.
- **JSON Keys:** Altijd `camelCase`.

Functies

- **Naamgeving:** Functienamen moeten zo duidelijk, beknopt en kort mogelijk zijn (in `camelCase`).

Bestanden & Mappen

- Mappen binnen **`src/`**: `camelCase` (bijv. `src/utils`, `src/store`).
- **Specifieke Component-mappen:** Mappen die UI-componenten of schermen bevatten schrijf ik in `PascalCase` .
- UI/Component bestanden (**`.tsx`** **):** UI componenten schrijf ik in `PascalCase` (bijv. `PrimaryButton.tsx`).
- Logica bestanden (**`.ts`** **):** Bestanden met logica, modellen, stores of reducers schrijf ik in `camelCase` (bijv. `incidentStore.ts`).
- Overige bestanden (`.json`, **`.css`**, **etc.):** Altijd `camelCase`.

⚠️ Uitzondering: De app/ map (Expo Router)

Omdat dit project gebruikmaakt van Expo Router (waarbij de bestandsnaam direct de URL/deep-link van de applicatie bepaalt), is er één belangrijke uitzondering:

- **Routing bestanden:** Bestanden en mappen binnen de `app/` directory worden altijd in kleine letters geschreven (kebab-case indien nodig), zoals `app/(tabs)/index.tsx` of `app/incident/[id].tsx`. Dit garandeert schone en web-vriendelijke URL's.