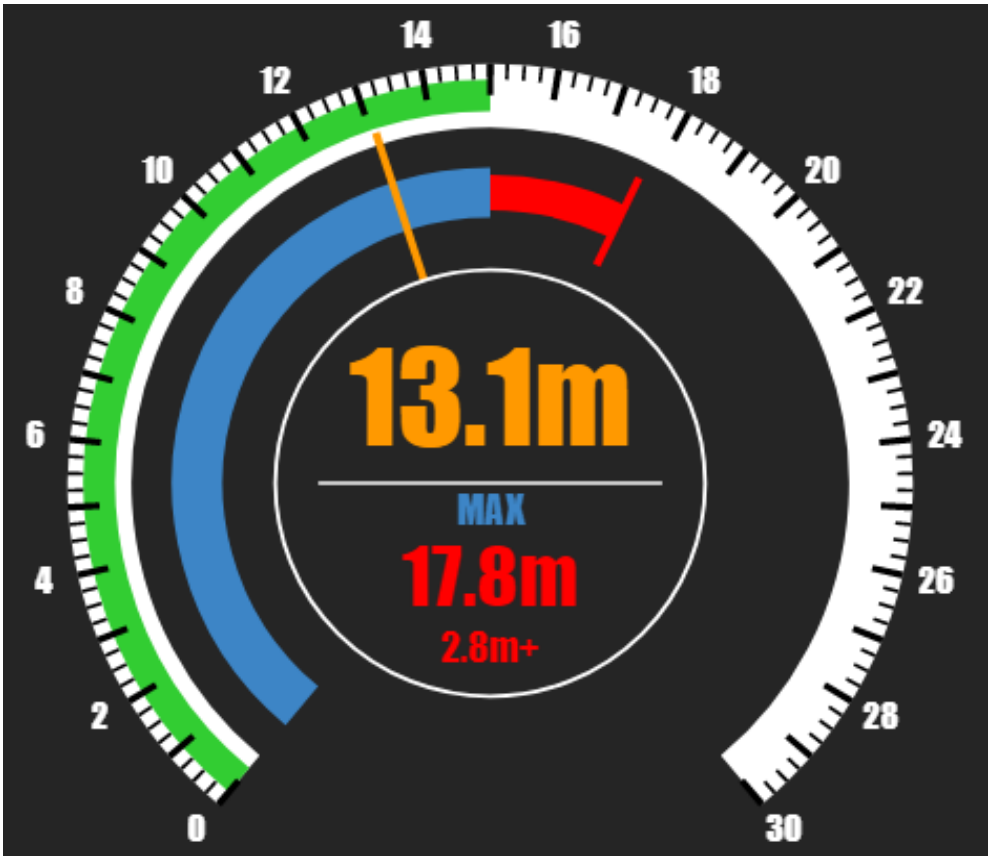
CHALMERS




# Applikation för planering och bevakning av dykeriarbete

Examensarbete inom Data- och Informationsteknik

Joel Petersson

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

EXAMENSARBETE

# Kravspecifikation och utveckling av applikation för användning vid yrkesdykning

Joel Petersson

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET

Göteborg 2020

**Kravspecifikation och utveckling av applikation för användning vid yrkesdykning**

Joel Petersson



Examinator: Peter Lundin

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
412 96 Göteborg
Telefon: 031-772 1000



Omslag:
Klockmätare för djup

Institutionen för Data- och Informationsteknik
Göteborg 2020

# SAMMANFATTNING

För att underlätta bevakning, planering och dokumentation av dykeriarbete har en applikation utvecklats i samarbete med Poseidon Diving Systems AB. Funktionalitet för att skapa dykplaner anpassade efter arbetsmiljöverkets föreskrifter för kommersiell dykning och sedan spara dessa i en lokal databas har implementeras samt bevakning av dyk genom att utrusta en yrkesdykare med en djupsensor som kontinuerligt rapporterar data till en pc på land via en CAN-länk. Applikationen representerar sedan informationen i ett realtidsgränssnitt med från grunden skapade grafiska komponenter, vilket låter en dykledare bevaka pågående dyk.

Applikationen har utvecklats med hybrid-ramverket Ionic med JavaScript biblioteket React för att skapa användargränssnittet. Genom att skapa abstraktioner för plattformsspecifika moduler har goda förutsättningar skapats för att kunna stödja nya plattformar.

Projektets resultat blev en lyckad prototyp som möter majoriteten av de krav som ställts med god potential för vidareutveckling.

**Nyckelord:** Applikationsutveckling, Hybrida Applikationer, Ionic, React, TypeScript, Dykeriarbete, Design, Användargränssnitt, NeDB

# FÖRORD

Jag vill tacka Thomas Oskarsson och Filip Petersson på Poseidon Diving Systems för deras stöd och rådgivning under detta projekt. Jag vill också tacka Poseidon Diving System för att de, trots svåra omständigheter, erbjudit både plats och utrustning på sitt kontor.

# Innehåll

# Figurer

# 1. Introduktion

## 1.1 Bakgrund

Yrkesdykning är en av de farligaste yrken som existerar med en dödlighet på 20-40 per 100,000 [1]. Det är en dödlighet som är 12.3 till 24.7 gånger högre än resten av anläggningsindustrin, som i sig är en av de farligaste industrierna. En av de vanligaste skadeorsakerna är tryckfallssjuka som kan inträffa vid för snabb uppstigning från botten eller när gränserna för expositionstid vid ett visst djup har överskridits. Många av dessa dödsfall och olyckor anses kunna undvikas med bättre rutiner och verktyg.

Innan dykeriearbete får påbörjas skall enligt arbetsmiljöverket ett dyklag utses[2]. I varje sådant skall en dykledare finnas. Det är dykledarens uppgift att planera, leda och övervaka dykeriarbetet, samt att göra en riskbedömning inför varje dyk. Det finns mycket som kan gå fel under ett dyk, och de verktyg som idag används för monitorering av en yrkesdykare under ett dyk är relativt primitiva, bestående av främst röstkommunikation mellan dykaren och dykledaren.
Utöver den vanliga arbetsledningen som sker via denna röstlänk måste dykledaren kontinuerligt fråga dykaren (eller dykarna) vilket djup de befinner sig på för att säkerställa att de inte överskrider det planerade maxdjupet. I yrkesdykning, till skillnad från sportdykning där man använder datorer, används tabeller för att beräkna hur länge en dykare får stanna vid ett specifikt djup samt hur länge de måste vänta för att få dyka igen efter uppstigning. Dessa tabeller visar hur mycket kväve som kroppens vävnad absorberat under dyket genom att specificera en gruppbeteckning (generellt en bokstav mellan A till O) som sedan kan användas med en annan tabell för att beräkna hur länge dykaren behöver vänta tills kvävenivån har återställts.

Detta projekt utfördes i samarbete med Poseidon Diving Systems i Göteborg och ämnar skapa en applikation som kan bidra till att yrkesdykning kan ske säkrare och effektivare.

## 1.2 Syfte

Syftet med detta projekt är att skapa en applikation som kan assistera en dykledare med att planera,bevaka och dokumentera yrkesdykning genom att i realtid rapportera djup,tid och temperatur och efter avslutat dyk spara informationen för att underlätta dokumentation.

Det primära fokuset för detta projekt är att skapa ett grafiskt användargränssnitt som tydligt rapporterar den information som har högst prioritet för dykledaren.

## 1.3 Mål

Målen med detta projekt är att:

- Undersöka slutanvändarens behov och utforma en kravspecifikation för applikationen.
- Designa en visuell presentation av data som levereras i realtid från en sensor.
- Utveckla en applikation, för en lämplig plattform och ramverk, som uppfyller kravspecifikationen. Applikationen skall designas så att byte av plattform möjliggörs.
- Logga den insamlade informationen i en databas för dokumentation.

## 1.4 Avgränsningar

Projektet kommer endast att utveckla en prototyp. I projektet ingår det ej att vattensäkra hårdvaran.
För att lyckas hålla tidsramen för projektet så kommer endast dyk med direktuppstigning stödjas av applikationen. Funktionalitet som assisterar med så kallad etappuppstigning där en dykare måste spendera tid vid olika djup vid uppstigningen kommer därmed inte implementeras. Applikationen kommer endast utvecklas och testas med Windows som platform.

# 2. Teknisk bakgrund

I detta kapitel beskrivs kort tekniker som använts eller varit relevanta under projektets gång.

## 2.1 Hårdvara

Här följer en översikt av den hårdvara som användes i detta projekt.

### 2.1.1 Tryckmätare

Den mätare som användes för att samla in djupdata är en sensor som Poseidon Diving Systems tillhandahöll. Den består av ett kretskort där diverse sensorer är anslutna till en mikroprocessor. Mätaren har egen mjukvara som har ansvar för att läsa av sensorerna och sända ut informationen via en CAN-buss.
Denna modul har inget batteri och behöver därför bli försörjd med spänning via CAN-anslutningen.

### 2.1.2 CAN

CAN (Controller Area Network) är en typ av seriell kommunikationsteknik som används främst i fordonsindustrin. En CAN-buss tillåter flera noder i ett nätverk att skicka information till varandra på ett enkelt och tillförlitligt sätt. För att detta skall vara möjligt krävs det att varje nod är en mikrodator som har en mjukvaruarkitektur som består av: ett fysiskt lager som omvandlar digital information till signaler att skicka ut på en länk, ett kommunikationslager som har ansvar för att sändning och mottagning sker på ett korrekt sätt samt ett applikationslager som läser av sändare och aktiverar aktuerare[3].

## 2.2 Hybrida applikationer

En hybrid applikation är ett mellanting mellan en nativ-applikation, där kod skrivs och kompileras för specifikt den valda plattformen, och en webapplikation som skrivs i kod som kan tolkas och visas i alla webläsare. Hybrida applikationer låter utvecklare skriva sin kod så portabel som möjligt genom att använda webtekniker för att utveckla applikationen utan att ge upp möjligheten att komma åt plattformsspecifika detaljer så som filsystem, kamera och blåtand [4].

Detta möjliggörs genom att den applikation som skapas är en nativ applikation för den valda plattformen med en webläsarvy där användargränssnittet rendreras samt en javascript runtime-miljö. Applikationen körs på så sätt som en lokalt sparad websida som har tillgång till alla resurser som en nativ applikation har.

### 2.2.1 Ionic

Ionic är en uppsättning mjukvaruverktyg för utveckling av hybrida applikationer[5]. Utöver utvecklingsverktygen så tillhandahåller Ionic även en mängd webkomponenter som låter applikationen anpassa sig grafiskt till den plattformen den körs på. Detta ger applikationen ett utseende som annars endast nativa applikationer lyckas åstadkomma.

### 2.2.2 Capacitor

Capacitor är Ionics runtimemiljö som tillåter webapplikationer att köras i en webläsarvy med nativ åtkomst till en plattforms resurser[6]. Capacitor stödjer utveckling av mobilapplikationer på både Android och iOS samt Linux, Mac och Windows via Electron.

Capacitor använder sig av ett pluginsystem för att exponera nativ funktionalitet till applikationen. En plugin består av ett gränssnitt som utvecklare kan implementera i nativ kod för varje plattform. Detta gränssnittet kan sedan anropas från huvudapplikationen genom att Capacitor skapar en brygga mellan webläsarvyn och den underliggande nativa applikationen. På så sätt krävs det endast att den plattformsspecifika koden skrivs flera gånger medans koden för användargränssnitt och grundläggande applikationslogik kan vara gemensam för alla plattformar.

### 2.2.3 TypeScript

TypesScript är ett open-source programmeringsspråk som bygger på JavaScript genom att lägga till statiska typdefinitioner[7]. Med hjälp av statiska typer kan objekt beskrivas och verifieras innan någon kod körs vilket underlättar felsökning och dokumentation. Detta tillägg är valfritt att använda, så JavaScripts typinferens fungerar fortfarande med hjälp av typen 'any'.
TypeScript transkompileras till JavaScript som sedan kan köras i en webläsare eller som en serverapplikation via Node.

## 2.3 React

React (eller React.js/ReactJS) är ett JavaScript-bibliotek som underlättar skapandet av användargränssnitt för webben. Det är skapat, och underhålls, främst av Facebook som ett open-source projekt[8].

### 2.3.1 Komponenter

Ett användargränsnitt skrivet i React är uppbyggt att diskreta komponenter. Dessa komponenter kan vara en av två typer: funktionella komponenter, vilket innebär att de inte kan ha något inre tillstånd, och klasskomponenter som kan innehålla internt beständigt tillstånd.
Varje komponent har en renderingsfunktion som returnerar HTML-element och/eller andra komponenter. Det är dessa element, tillsammans med css filer, som beskriver utseendet av komponenten. Genom att ta emot parametrar (så kallade 'props') kan en komponent ändra sina interna element, och samma komponent kan användas flera gånger på samma sida men med olika innehåll. En komponent kan inte själv förändra sina props, så för data som behöver kunna uppdateras internt används vad som kallas 'state'. Detta interna tillstånd kan sedan skickas som props till subkomponenter inuti komponenten.

### 2.3.2 Virtuelt DOM-träd

I React byggs användargränssnittet upp helt programatiskt i JavaScript (eller JSX som är en påbyggnad till JavaScript som gör syntaxen för React mer hanterbar) medans sidans HTML-dokument endast består av ett rot-element. Sidan fylls genom att React för in de relevanta komponenterna genom att manipulera sidans DOM-träd.
För att undvika onödig om-rendering använder React sig av ett virtuellt DOM-träd, vilket är en datastruktur som speglar användargränssnittets tillstånd. När en förändring sker i något gränssnittstillstånd skapas ett nytt virtuellt DOM-träd och en differens mellan det gamla och det nya beräknas.
När detta gjorts beräknas det mest optimala sättet att uppdatera de berörda noderna, och sedan appliceras förändringarna på det faktiska DOM-trädet. Genom denna process kan manipulering av DOM-trädet minimeras vilket ökar prestandan.

## 2.4 NeDB

NeDB är ett lättviktigt databassystem som är skrivit i JavaScript[9]. Det kan köras både på webben eller lokalt och använder sig av ett API som är designat för att likna det som MongoDB använder. NeDB lämpar sig för bäst för små mängder data, men tack vara likheten med MongoDB är det trivialt att byta till denna när datamängden överskrider vad NeDB kan hantera. NeDB lagrar information i json-format.

# 3. Metod

Detta kapitel beskriver metoden som användes för att utföra projektet i enlighet med de målen beskrivna i introduktionen.

## 3.1 Förstudie

Som inledande arbete för projektet utfördes en förstudie i två faser som utfördes parallellt. Den första genomfördes för att skapa en förståelse för bakgrunden till den problembeskrivning som ställts. Detta gjordes för att kunna skapa en kravspecifikation för applikationen med lämpliga avgränsningar, vilket krävde en djupare inblick i den kommersiella dykindustrin.
Den andra fasen bestod av besluta vilken plattform som var lämplig för applikationen samt en kortare teknisk utvärdering av potentiella verktyg och utvecklingsmiljöer.

### 3.1.1 Insamling av domänkunskaper

Förstudien inleddes med ett möten och diskussioner med anställda på Poseidon för att skaffa en överblick över dykindustrin samt att få en förståelse för skillnaderna mellan sportdykning och kommersiell dykning. En genomgång av den påbörjade förstudie som utförts i samarbete med en av Poseidons kunder genomfördes med förslag på avgränsningar.

#### 3.1.1.1 YRGO

Ett besök på YRGO Yrkesdykarskola genomfördes som bestod av en demonstration av hur kommersiell dykning fungerar i Sverige, vilka verktyg som används samt vilka roller som ingår i ett dyklag. Efter demonstrationen utfördes en intervju med lärare från utbildningen där en prioritering av på potentiella funktioner skapades vilken sedan användes som grund för kravspecifikationen.

### 3.1.2 Val av plattform och utvecklingsverktyg

En analys av vilka krav som den hårdvara som skall användas måste uppfylla utfördes, och utifrån denna undersöktes utbudet på marknaden för att på så sätt skapa en uppfattning om vilka plattformar som är bäst lämpad för denna typ av applikation.

Efter att valet av plattform gjorts påbörjades en utvärdering av de tillgängliga utvecklingsverktygen. Både nativa och hybrida applikationer undersöktes, och ett be-

slut togs att använda utvecklingsramverket Ionic för att skapa applikationen. Detta beslut baserades på det faktum att Ionic tillät utveckling av hybrida applikationer som med relativ enkelhet kan anpassas till flera olika plattformar. Poseidon var också intresserade av att ha tillgång till de komponenter som skulle skapas för ett annat projekt som använder Ionic.
Som alternativ till Ionic undersöktes applikationsramverket Qt som också erbjuder funktionalitet för att utveckla användargränssnitt som fungerar på flera olika plattformar.

## 3.2 Kravspecifikation

I samband med förstudiens avslut påbörjades utformningen av en kravspecifikation där en detaljerad beskrivning av applikationens funktionella och icke-funktionella krav beskrevs.
Genom att utgå från den insamlade informationen från förstudien specificerades applikationens huvudfunktioner, och en så kallad användarpersona skapades som kort beskrev vem den tänkta användaren är och varför de skulle vilja använda applikationen. En lista över applikationens olika vyer samt navigationsflödet mellan dessa utformades, samt beskrivningar av vyernas olika grafiska komponenter.
Kravspecifikationen utvärderades och förändrades i samspel med anställda på Poseidon och yrkesdykare med lång erfarenhet.

## 3.3 Designprocess

En stor del av projektet bestod av att utforma grafiska komponenter specifikt designade att förmedla dykinformation på ett så effektivt sätt som möjligt. På grund av det smala användningsområdet valde vi därför att skapa dessa komponenter från grunden istället för att utgå från och modifiera existerande biblioteks liknande komponenter.
Denna process bestod av att använda digitala ritverktyg för att formge förslag på möjliga utseende för varje komponent. Dessa skisser utvärderades sedan i likhet med kravspecifikationen, och förslag på förändringar mottogs. Genom att arbeta iterativt på detta sätt kunde de ritningar som låg till grund för den faktiska implementationen färdigställas tidigt i projektets utveckling.

## 3.4 Testning

På grund av begränsningar i utformningen av sensorn Poseidon tillhandahöll var det omöjligt att utföra undervattenstester. För att kunna testa systemet vid höga tryck användes därför istället en akryltryckkammare som anslöts till en tryckluftsflaska. Med hjälp av reglage för att kontrollera trycket i kammaren kunde på så sätt dyk simuleras.

# 4. Specifikation

I detta kapitel beskrivs kravspecifikationen som togs fram för detta projekt.

## 4.1 Sammanfattning

DMS är en applikation som hjälper dykledare inom yrkesdykning att planera, genomföra och dokumentera dyk med direktuppstigning. Applikationen skall agera som ett verktyg som underlättar dykledarens arbete genom att i realtid visa information om dykaren under dykets gång och grafiskt visa om dykplanen följs. Dykplanen skall utformas efter de dyktabeller som idag används i Sverige vid yrkesdykning. Efter genomfört dyk skall den insamlade datan sparas i en lokal databas som har möjlighet att exportera till en csv fil.

## 4.2 Användarpersona

### 4.2.1 Vem?

Användaren är en dykledare som arbetar antingen på en myndighet eller ett företag som sysslar med yrkesdykning.

### 4.2.2 Varför vill de använda applikationen?

Genom att effektivisera planeringen och i realtid rapportera information som tidigare behövde levereras via röstkommunikation minskar mängden information dykledaren behöver hålla i huvudet samtidigt och låter denne fokusera på att ta informerade beslut.

## 4.3 Planerade dyk

En lista av redan skapade dykplaner med möjlighet att lägga till nya. När användaren väljer att skapa en ny plan skall denne fylla i en huvudplan samt två backup-planer för varje dykare. Dessa skall bestå av följande:

- Dykares namn/id.
- Dykskötares namn/id.
- Dykplats.
- Vilken dekompressionstabell som skall användas.

- Vilken gruppbeteckning dykaren har innan dyket.
- Planerat dykdjup.
- Planerad dyktid.
- Typ av vatten

När dykplanen fyllts i skall applikationen rapportera dykarens förväntade gruppbeteckning om dyk sker enligt plan.
När en redan skapad dykplan väljes skall en överblick av planen visas, möjlighet att förändra planen ges samt ett alternativ att starta ett dyk med den valda planen.

## 4.4 Under dyk

När användaren valt att starta ett dyk skall en ny skärm visas med information som skickas från en sensor på dykaren som applikationen skall visa grafiskt. Informationen skall även loggas i en lokal databas. Information som skall visas grafiskt är följande:

### 4.4.1 Dykplan

Vilken dykplan som används samt möjligheten att byta till någon av backup planerna. Om byte av plan sker skall gränsvärdena i de andra grafiska komponenterna uppdateras.

### 4.4.2 Djupmätare

Djupmätaren skall visa djup just nu samt uppnått maxdjup. Mätaren skall ha markeringar som visar var gränserna för det planerade dyket går. Om planerat maxdjup överskrids skall detta visas grafiskt med färg samt med varningstext.

### 4.4.3 Djup-tid-graf

En graf som i realtid visar hur den faktiska djupdatan förhåller sig till dykplanen. Dykplanen ritas upp i förväg och sedan fylls grafen med djupdata över tid. Värden som ligger inom/utanför dykplanen skall markeras med olika färg.

### 4.4.4 Uppstigningsmätare

Mätare som visar hur snabbt dykarens uppstigning sker och varnar om den överskrider rekommenderade säkerhetsgränser.

### 4.4.5 Tidtagare

Timers som visar hur länge dyket har pågått samt hur lång tid det är kvar tills uppstigning. Dessa skall startas manuellt av dykledaren när dyket sätts igång.

## 4.5 Efter dyk

Efter dyket skall en en summering av dyket visas. Varje dykare skall ha en egen summering. Summeringen skall innehålla följande:

### 4.5.1 Gruppbeteckning

Föreslagen gruppförteckning utifrån den tabellen som valts i planeringen.

### 4.5.2 Djup-tid-graf

En mer detaljerad graf med genomsnittsdjup utmarkerat som en horisontell linje.

## 4.6 Genomförda dyk

Här skall en lista av genomförda dyk visas med möjlighet att välja ett dyk för att få se mer detaljerad information. Skärmen som visar den detaljerade informationen skall bestå av samma komponenter som summeringen som visas efter ett genomfört dyk.

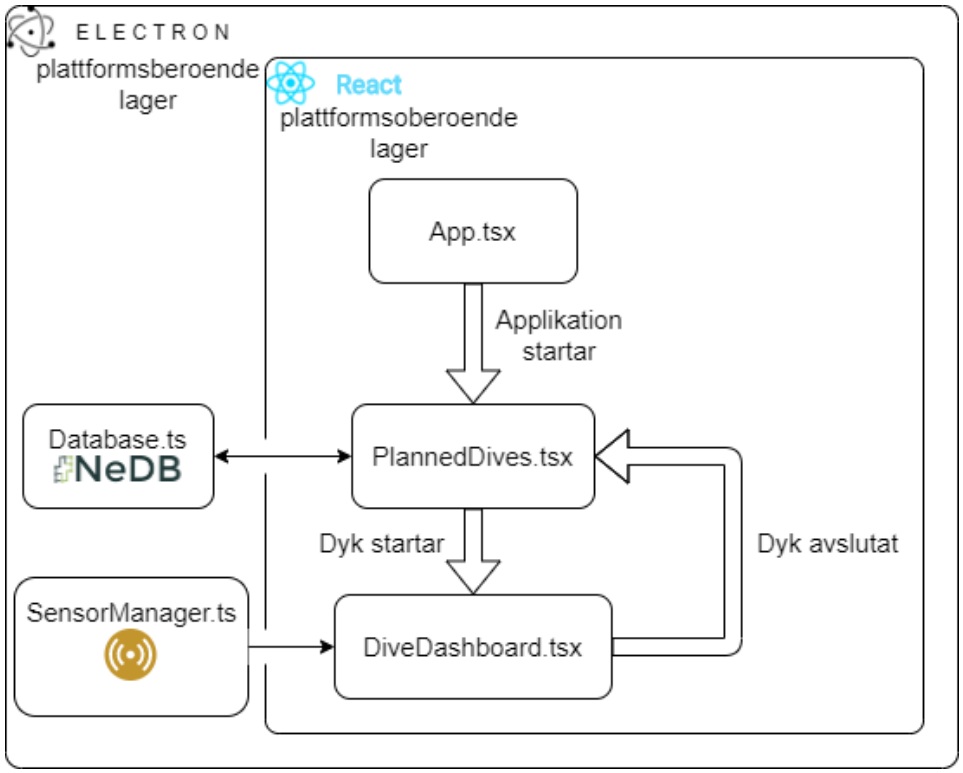
# 5. Implementation

I detta kapitel beskrivs implementationen av applikationen enligt kravspecifikation.

## 5.1 Applikationens struktur

Den grundläggande strukturen för applikationen skapades med hjälp av Ionics CLI med flaggor satta för att använda React som ramverk. Detta verktyg skapar en mappstruktur för projektet, filer som är gemensamma för alla React applikationer samt installerar de moduler som projektet beror på. För att kunna komma åt hårdvaran för den valda plattformen (Windows) konfigurerades byggsystemet till att använda Electron.

Applikationen är uppdelad i två huvud-vyer, `PlannedDives` och `DiveDashboard`, som kan navigeras mellan via en `<IonReactRouter>` komponent. Var och en av dessa består av en React-komponent som är den överste i den vyns hierarki av komponenter och äger den delen av applikationens tillstånd. Detta tillstånd delas sedan ut till vyns subkomponenter via props.
Dessa vyer är uppbyggda på två olika sätt: `PlannedDives` är implementerad med ett antal färdiga komponenter via Ionics gränssnittsbibliotek medans `DiveDasboard`innehåller komponenter som implementerats från grunden.




**Figur 5.1:** Mjukvaruarkitektur

## 5.2 Planerade dyk

`PlannedDives` är startsidan för applikationen och visar en lista med de dykplaner som användaren har skapat samt tillåter skapandet av nya dykplaner. När användaren markerar en av dykplanerna i lista dyker en knapp upp som vid ett tryck startar ett dyk med den valda planen.

En dykplan är en lista av en datastruktur bestående av ett `DiverID`, ett `TenderID` och tre instanser av `DiveProfile` som i sin tur innehåller två värden: planerat maxdjup för dyket och den planerade expositionstiden vid det djupet. Längden på listan avgör hur många dykar/dykskötar-par som skall delta i dyket.
De dykplaner som användaren skapat sen tidigare hämtas från en lokal databas, implementerad med Node-modulen NeDB, som representeras av klassen `Database`. Denna klass exponerar metoderna `getAllDivePlans()` och `insertDivePlan()` som tillåter hämtning och insättning av dykplaner. När vyn `PlannedDives` laddas instansieras denna klass och de dykplaner som finns sparade hämtas och läggs till i en lista. Varje dykplan i listan skickas sedan som prop till varsin `DivePlanEntry` som renderas i ett listfönster.
`DivePlanEntry`beräknar med hjälp av informationen i dykplanen den mottagit den resulterande gruppbeteckning för varje dykprofil och visar detta för användaren. När komponenten anropar funktionen som beräknar gruppbeteckning måste den dekompressionstabell som skall användas specificeras som ett argument. Under projektet implementerades endast försvarsmaktens dekompressionstabell för dyk med

direktuppstigning[2] (vilken är den som skall användas vid dykeriarbete i Sverige) men genom att utforma funktionaliteten för beräkning av gruppbeteckning så generellt som möjligt är utökning av tabeller trivialt.

Skapandet av nya dykplaner utformades så att endast värden som är godkända enligt den valda dekompressionstabellen tillåts. Användaren får välja hur många dykar/dykskötar-par som planen skall innehålla, och när alla fält är ifyllda kan läggs dykplanen till i databasen och listfönstret uppdateras med den nya planen.

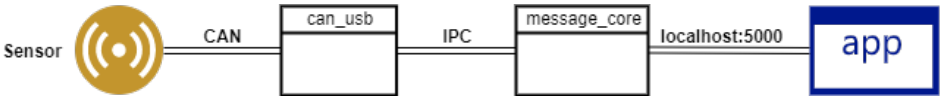
## 5.3 Realtidsgränssnitt

Större delen av arbetet med projektet kretsade kring realtidsgränssnittet, som enligt den avgränsningen som satts i samråd med Poseidon var huvudfunktionaliteten för applikationen.

Realtidsgränssnittet består av en hierarki av React-komponenter där `DiveDashboard` är roten. Denna innehåller sedan, beroende på vilken dykplan som valts, en eller två instanser av `DiveInterfaceContainer`vilken är den komponent som tar emot, sätter och delar ut förändringar i tillstånd.

### 5.3.1 Djupsensor

Den djupsensor som användes under projektets utveckling är en del av en annan av Poseidons produkter och inte är den tilltänkta slutliga hårdvaran. på grund av detta implementerades inte avläsningen av denna inuti applikationen. Avläsningen skedde istället i en separat process som läste av CAN-paket som sensorn skickade via en seriell port. Efter att processen mottagit ett paket skickades detta till än annan process via IPC som sedan i sin tur skickade ut datan som ett tcp-paket på en lokal socket-anslutning. Avkodningen av paketen implementerades dock, då varje steg beskrivit ovan endast vidarebefordrade CAN-paketen utan att processa dem.




**Figur 5.2:** Kommunikation med sensor.

Denna implementation för transporten av sensordata var en tillfällig lösning och skulle vara för omständig att använda i en slutprodukt.
På grund av detta valde vi att skapa abstraktionen `SensorManager` för applikationens hanteringen av indata. Denna abstraktion bestod av att låta `SensorManager` vara en subklass till `EventEmitter`; en klass som levererar data i form av events, och som exponerar ett API där ägaren av objektet kan registrera lyssnar-funktioner som exekverar i respons till dessa (Node.js implementation av observatör designmönstret). Läsning av indata från valfri källa (i vårat fall en lokal socket-anslutning) implementerades sedan i två publika klassmetoder, `connect()` och `disconnect()`.

Genom att sända ut events vid tryck- eller temperaturförändring levererades den relevanta datan på så sätt till rätt komponenter med minimal koppling.
På detta sätt kunde implementationsdetaljerna för inmatning av data kapslas och separeras från resten av applikationskoden, vilket utöver att möjliggöra förändring i sensorhårdvara underlättar för en potentiell utökning av stödda plattformar.

### 5.3.2 Grafiska komponenter byggda med svg

Efter att ha utvärderat ett antal befintliga bibliotek för skapandet av digitala instrumentbrädor tidigt i utvecklingen blev det uppenbart att komponenter specifikt designade för dykändamål var nödvändiga för att skapa en användarupplevelse i enighet med önskemålen från kunder och samarbetspartners.

Detta innebar att varje komponent behövdes byggas grafiskt från grunden. För detta ändamål valde vi att använda HTML-elementet `<svg>`, som låter användare rita vektorgrafik programatiskt med hjälp av ett antal primitiva former. Att använda denna teknik istället för `<canvas>`, som ger en ännu mer finkornig kontroll över pixlarna på skärmen, var till stor del på grund av hur React som ramverk är uppbyggt.

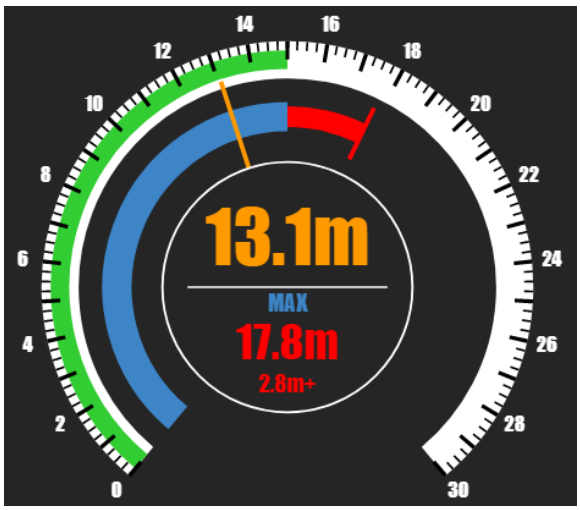
Genom att skapa komponenter vars `render()` funktion innehåller ett `<svg>`-element, med komponentens grafiska beståndsdelar beskrivna deklarativt i kod, kan man via Reacts prop-system förändra utseendet när en attribut till komponenten förändras.

```
SvgChildComponent extends React.Component<{radiusProp: number},{}> {
    render() {
        return (
            <svg viewBox="0 0 100 100" xmlns="http://www.w3.org/2000/svg">
                <circle cx="50" cy="50" r={this.props.radiusProp}/>
            </svg>
        );
    }
}
```

```
SvgParentComponent extends React.Component<{},{radiusState: number}> {
    render() {
        return (
            <div>
            <SvgChildComponent radiusProp={this.state.radiusState} />
            <button onClick={() => this.setState({radiusState: radiusState+1})}>
                Increase Radius
            </button>
            </div>
        );
    }
}
```

**Figur 5.3:** Reactexempel som illustrerar hur grafiska komponenter byggs med svg-element.

I exemplet ovan skickar `SvgParentComponent` sitt interna tillstånd `radiusState` som prop till subkomponenten `SvgChildComponent`. När knappen trycks ner uppdaterar `SvgParentComponent` sitt tillstånd vilket signalerar till React att `SvgChildComponent` måste uppdateras.
Denna metod låter på så vis förändringar i tillstånd som levererats via props påverka de värden inuti komponenten som påverkar dess utseende. Alla de grafiska komponenter vars uppgift var att rapportera information om det pågående dyket använde sig av denna teknik.

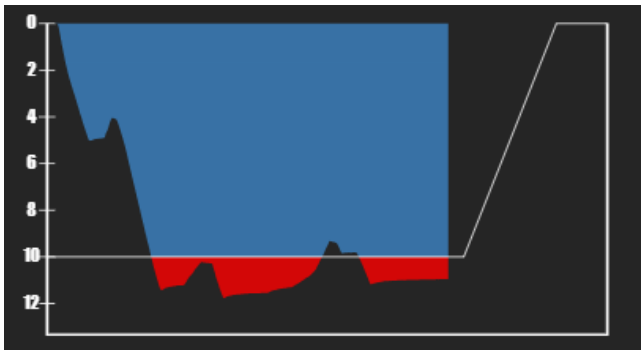



**Figur 5.4:** DepthGauge - Djupmätare

`DepthGauge` är den komponent som levererar den viktigaste informationen till dykledaren och behövde därför vara den del i användargränssnittet som är tydligast och tar mest plats. Designen som framtogs är inspirerad av klassiska klockmätare som är vanliga vid tryckmätning.
Då det inte finns något svg-element som enkelt uttrycker cirkelbågar implementerades subkomponenten `SvgArc` för att kunna skapa tryckmätarens olika delar. Denna

komponent tar via props emot värden för: radie, star- och slutvinkel, tjocklek, färg samt hur många procent av cirkelbågen som skall fyllas.

`DepthGauge` tar emot ett max- och minvärde för mätaren, antal värdemarkeringar som skall ritas, det högsta tillåtna värdet innan komponenten varnar samt värden för det nuvarande djupet och det maxdjup som uppnåtts under dyket.
Under ett pågående dyk uppdateras det nuvarande och max uppnådda djupet kontinuerligt. Resten av `DepthGauge`s props initieras med den första dykprofilen i den valda dykplanen, men kan ändras genom att användaren byter vilken dykprofil som skall användas.




**Figur 5.5:** DepthChart - Djup-tid-graf

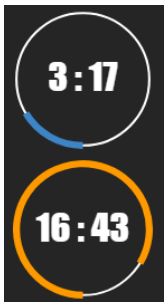
`DepthChart` är en komponent som visar förändring av djup över tid med hjälp av en graf. Den valda dykplanen är visad med en linje och värden som går över det planerade maxdjupet visas med röd färg. Inritad är också ett uppstigningsförslag som är anpassat efter försvarsmaktens rekommendation på 9 meter per minut.
Varje förändring i tryck sparas av `DiveInterfaceContainer` tillsammans med en tidsstämpel i en lista, men för att minimera antalet gånger `DepthChart` behöver ritas om filtreras denna listan så att endast ett djupvärde per pixel rapporteras till komponenten.
Detta görs genom att beräkna tidsintervallet en pixel i `DepthChart` representerar, sätta en variabel till den aktuella tiden plus detta intervall och sedan vänta med att rapportera förändringar tills det att tidsstämpeln överskrider denna variabel. För att rapportera den mest konservativa bilden av dyket så är det ett lokalt maximum från intervallet som skickas till komponenten.

När djup eller tid överskrider grafens kanter skalas komponenten dynamiskt om för att rymma informationen och djupmarkörerna anpassas för att stämma överens med denna förändring.


**Figur 5.6:** AssenRateGauge - Uppstigningsmätare

`AssenRateGauge` är en simpel komponent vars funktion är att visa med vilken hastighet uppstigning ur vattnet sker och att varna användaren när en uppstigning sker

för snabbt. Markeringarna i komponenten fylls i takt med att hastigheten stiger, och när den rekommenderade uppstigningshastigheten överskrids signaleras detta genom att byta färg.




**Figur 5.7:** Tidtagarkomponent

JavaScripts inbyggda funktioner för att hantera tid garanterar inte korrekthet utan kan variera beroende systemets belastning. För att motverka denna osäkerheten implementerades en tidtagarklass `AdjustingInterval` som kompenserar intervallet som används i JavaScript-funktionen `setTimeout()` genom att beräkna felet mellan varje anrop.

```
step() {
    let drift = Date.now() - this.expected;
    this.expected += this.interval;
    //utför arbete här
    this.timeout = setTimeout(this.step, this.interval - drift);
}
```

**Figur 5.8:** Rekursiv tidtagarmetod som kompenserar för avdrift.

I exemplet ovan är `expected` den förväntade tidstämplen. Genom att jämföra den med den faktiska tidstämpeln `Date.now()` så beräknas avdriften. Det önskade intervallet adderas till `expected` för att få nästa förväntade tidstämpel och sedan anropas `setTimeout()` med intervallet minus avdriften.
Den grafiska representationen av tidtagerkomponenten implementerades genom att använda `SvgArc` komponenten med ett 360 graders vinkelspann.

# 6. Resultat

I detta kapitel följer en redogörelse för projektets resultat, vilka mål som uppfyllts samt hur väl applikationen som producerades möter de krav som ställdes i specifikationen.

## 6.1 Uppfyllda mål

**• Undersöka slutanvändarens behov och utforma en kravspecifikation för applikationen.**

En kravspecifikation baserad på inhämtad domänkunskap producerades. En god kontakt hölls med experter inom dykområdet vilket hjälpte denna växa fram iterativt genom intervjuer, kravförslag och sedan återkopplingar på dessa.

**• Designa en visuell presentation av data som levereras i realtid från en sensor.**

Visuella representationer av all realtiddata realiserades i form av komponenter i vyn `DiveDashboard`. Dessa komponenter byggdes från grunden för att så tydligt som möjligt förmedla den viktigaste informationen till dykledaren.

**• Producera en applikation för en vald plattform med hjälp av ett lämpligt ramverk som ger möjlighet för byte av plattform.**

Applikationen utvecklades med Windows som huvudplattform, men kan med mindre modifikationer anpassas till att köras på alla de plattformar som Ionic-ramverket stödjer (iOS, Android, Linux och Mac).

## 6.2 Ej uppfyllda mål

**• Logga den insamlade informationen i en databas för dokumentation.**

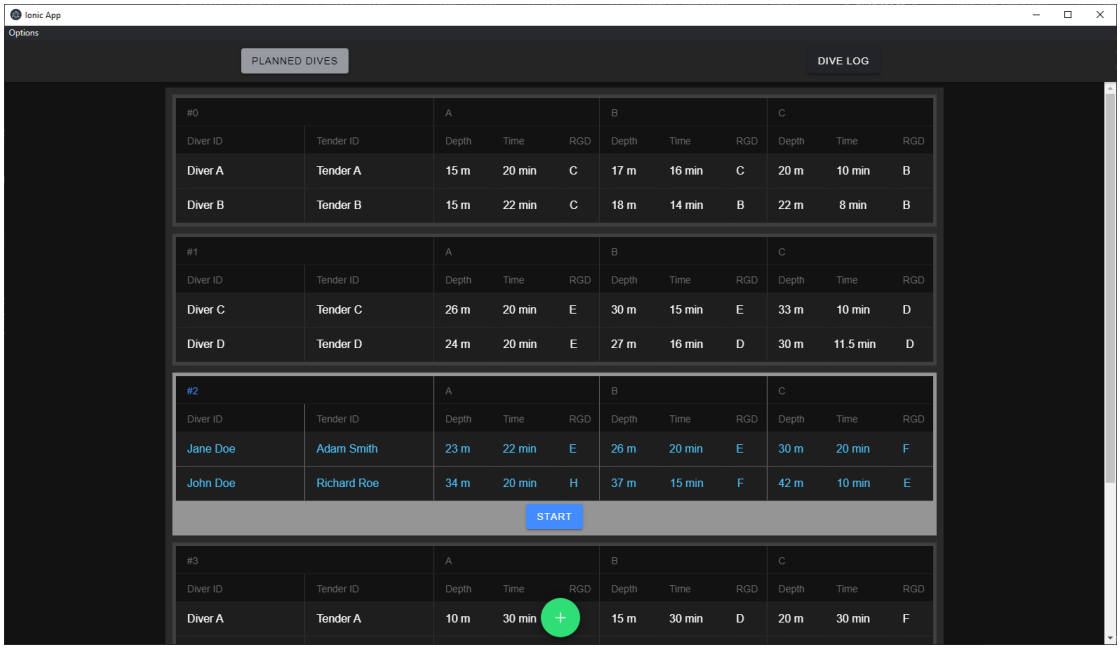
En lista med loggar av utförda dyk implementerades ej på grund av tidsbrist.

## 6.3 Uppnådda krav

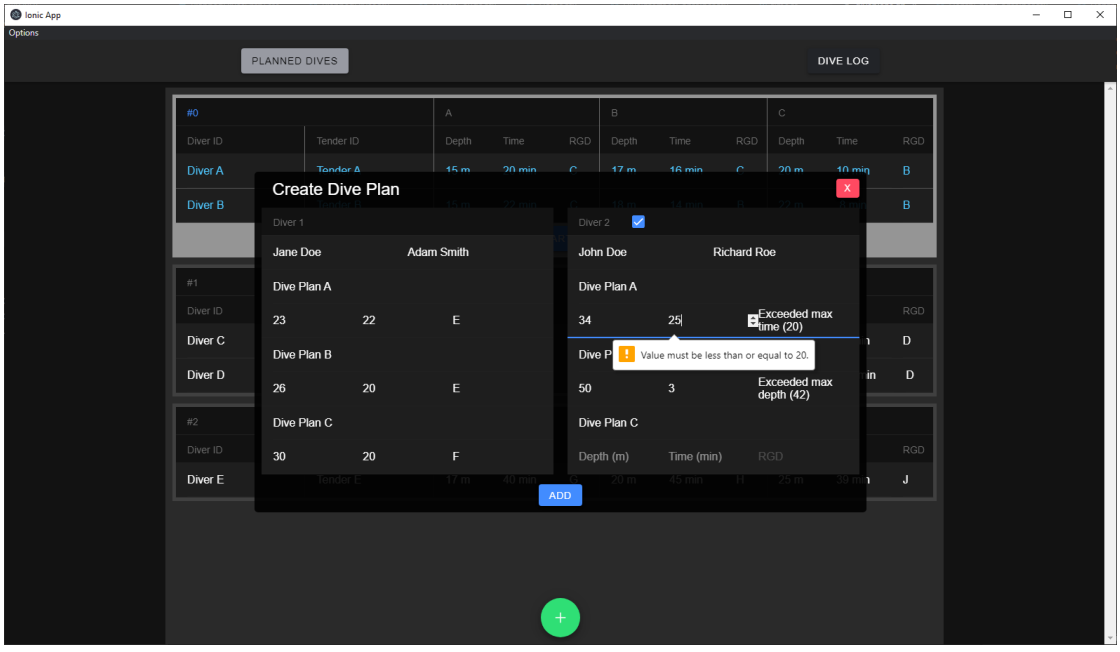
Den applikation som producerats uppnår de flesta, men inte alla de funktionella krav som sattes av kravspecifikationen.

### 6.3.1 Planerade dyk

Vyn för planerade dyk implementerades enligt kraven med undantag för dykplanens utformning. Dykplats, gruppbeteckning innan dyk samt typ av vatten inkluderades ej i den slutgiltiga applikationens version av en dykplan.



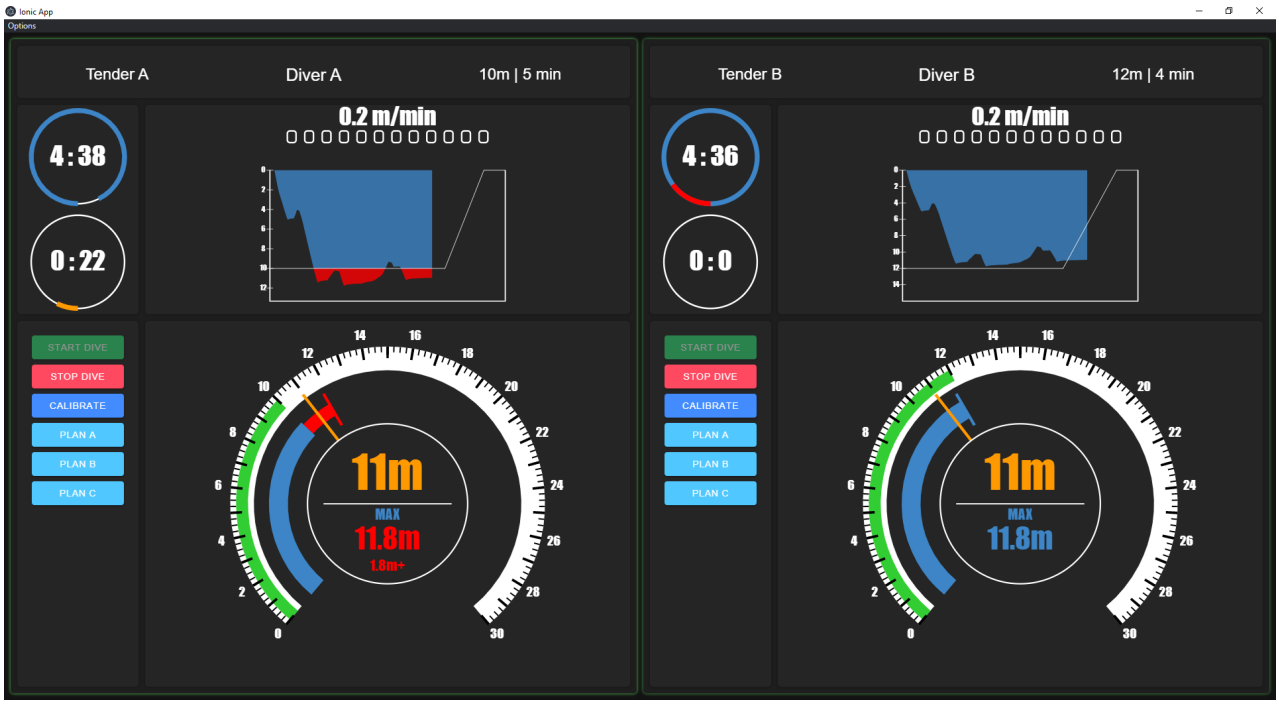

**Figur 6.1:** Planerade dyk




**Figur 6.2:** Skapa dykplan

### 6.3.2 Realtidsgränssnitt

Användargränssnittet som dykledaren ser under ett pågående dyk implementerades helt enligt kravspecifikationen. Utöver detta skapades också funktionalitet för att kalibrera djupsensorn.




**Figur 6.3:** Realtidsgränssnitt

# 7. Diskussion

Den applikation som utvecklades blev väldigt lyckad trots att den inte levde upp till all de satta målen. Att alla mål inte lyckades uppfyllas berodde till stor del på omfattningen av förarbetet som utfördes innan implementationen mjukvaran påbörjades. Förstudie, framtagning av en kravspecifikation och skissande av grafiska komponenter pågick i ett antal veckor innan någon kod började skrivas. Trots att detta förarbete lämnade otillräcklig tid för att implementera den applikation som specificerats var det en extremt lärorik process som gav en inblick i alla stadier av utveckling. Att kommunicera med slutanvändaren för att förstå det problem som applikationen försöker lösa, att arbeta i samarbete med domänkunniga för att komma fram till en kravspecifikation som är smal nog för den satta tidsramen men som ändå skapar tillräckligt med värde för användaren och att analysera plattformar och hårdvara från ett kundperspektiv var alla värdefulla erfarenheter.

## 7.1 Tillståndshantering med React

Reacts tillståndshantering är ett väldigt enkelt system tillstånd lever inuti komponenter och uppdateras därifrån. Detta system fungerar utmärkt när komponenter är få och har diskreta ansvarsområde där de inte behöver dela särskilt mycket data mellan varandra. Delning av tillstånd är inte omöjligt, men då måste tillståndet lyftas upp från komponenterna till en överordnad komponent som sedan kan dela ut detta till sina subkomponenter.
Detta skapade inte några större bekymmer i början av projektet; varje vys komponentträd var då relativt grunt. Med tiden blev dessa träd djupare och det skapades emellertid större och större problem som bottnade i var tillstånd var lagrat.
Exempel är kontrollkomponenter som behövde ha möjlighet att uppdatera tillstånd som låg flera lager ovanför, vilket krävde att funktioner för att hantera dessa interaktioner behövdes skickas ner igenom trädet via en kedja av props för att nå den ämnade komponenten.
Det största problemet var dock tillstånd som behövde delas mellan vyer. Det var möjligt, men på ett väldigt problematiskt sätt som vid uppskalning inte skulle vara hållbart.

Detta problem har sedan länge varit en känd svag punkt hos React, och det har därför utvecklats en rad bibliotek ämnade att lösa just detta. Det vanligaste av dessa - Redux - låter utvecklaren använda ett helt separat system för hantering av tillstånd. I stället för att låta komponenter äga sitt eget tillstånd skapas ett

globalt tillståndslager varifrån alla komponenter kan komma åt det tillstånd de är intresserade av.

Denna typ av lösning bör tillämpas innan vidare utveckling av applikationen sker, men det kommer innebära en signifikant refaktorisering.

## 7.2 Diskussion av arbetsmetod

Arbetsmetoden fungerade bra, men skulle möjligen behövts struktureras upp mer genom att sätta upp tydliga delmål. Att i ett projekt som utvecklas av en person åta sig att göra förstudie, kravspecifikation och implementering av en prototyp var svårt att hinna med.

## 7.3 Etik och arbetsmiljö

Detta projekt har i stor del utvecklats utan att betänka de etiska konsekvenser av de val som gjorts, både tekniska och designmässiga. Med tanke på domänens säkerhetskritiska natur är detta en aspekt som i allra högsta grad skulle vara lämplig att beakta.
Om ett fel skulle inträffa i applikationen så är potentialen för mänsklig skada (beroende självklart på felets karaktär) mycket stor. Som utvecklare har man därför ett etiskt ansvar att minimera dessa risker, vilket kan göras på en rad olika sätt.

- Mätbara säkerhetskrav kan inkluderas i mjukvarans specifikation.
- Utförliga tester kan utföras för att verifiera mjukvaran.
- Val av plattform, ramverk och designmönster kan utgå från ett krav på säkerhet.

Utöver den tekniska aspekten bör också andra perspektiv betänkas. Även om applikationen fungerar helt felfritt enligt specifikationen så kanske designval som gjort påverkar dykeriarbetet indirekt. En av målen med applikationen var att ta bort behovet av att ständigt fråga dykaren om dennes nuvarande djup via röstlänken med motiveringen att lämna mer plats för kommunikation om det faktiska arbete som utförs. Det är dock inte säkert att detta får den önskade effekten. Kanske är den konstanta kommunikationen om nuvarande djup en viktig del för att upprätthålla en pågående kontakt till ytan, och utan den så förmedlas i slutändan mindre information.

## 7.4 Hållbar utveckling

Applikationen som utvecklades har i sig inte någon större påverkan på miljö. En förhoppning med ett projekt som detta kan vara att det gör dykbranchen mer öppen för annan ny teknik som kan minska de miljöpåverkningar som speciellt anläggningsdykningen står för idag.

# Litteraturförteckning


[1] European Diving Technology Committee, "EDTC Annual Assembly,"2016. [Online]. Tillgänglig: http://edtc.org/wp-content/uploads/2020/05/Minutes-of-Tallinn-HR.pdf, hämtad: 2020-06-02.

[2] AFS 2010:16. Dykeriarbete: Arbetsmiljöverkets föreskrifter om dykeriarbete samt allmänna råd om tillämpningen av föreskrifterna.[Online]. Tillgänglig: https://www.av.se/globalassets/filer/publikationer/foreskrifter/dykeriarbete-foreskrifter-afs2010_16.pdf, hämtad: 2020-06-02.

[3] CAN Specification: Version 2.0, Stuttgart, Tyskland: Bosch, 1991. [Online]. Tillgänglig: http://esd.cs.ucr.edu/webres/can20.pdf, hämtad: 2020-06-02.

[4] C. Griffith, Åhat is Hybrid App Development?". [Online]. Tillgänglig: https://ionicframework.com/resources/articles/what-is-hybrid-app-development, hämtad: 2020-06-02.

[5] Drifty Co, Milwaukee, WI, "Ionic framework,"2020. [Online]. Tillgänglig: https://ionicframework.com/, hämtad: 2020-06-02.

[6] Drifty Co, Milwaukee, WI, "Capacitor: A Cross-platform App Runtime,"2020. [Online]. Tillgänglig: https://capacitor.ionicframework.com/docs/, hämtad: 2020-06-02.

[7] Microsoft Corporation, Redmond, WA, TypeScript Documentation,"2020. [Online]. Tillgänglig: https://www.typescriptlang.org/docs/home.html, hämtad: 2020-06-02.

[8] Facebook, Inc., Menlo Park, CA, React,"2020. [Online]. Tillgänglig: https://reactjs.org/, hämtad: 2020-06-02.

[9] L. Chatriot, NeDB,"2020. [Online]. Tillgänglig: https://github.com/louischatriot/nedb, hämtad: 2020-06-09.