Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Elektryczny

Łukasz Aniszewski

nr albumu: 36705
kierunek studiów: Automatyka i Robotyka
forma studiów: studia stacjonarne

APLIKACJA NA SMARTPHONE PREZENTUJĄCA PARAMETRY REJSU JEDNOSTKI PŁYWAJĄCEJ

SMARTPHONE APPLICATION THAT PRESENTS THE PARAMETERS OF THE VESSEL'S VOYAGE

Praca dyplomowa inżynierska

napisana pod kierunkiem:

dr inż. Robert Krupiński

Katedra Przetwarzania Sygnałów i Inżynierii Multimedialnej

Data wydania tematu pracy: 28.02.2020 r.

Data dopuszczenia pracy do egzaminu: .....

Szczecin, 2020

OŚWIADCZENIE
AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska pn.
„Aplikacja na smartphone prezentująca parametry rejsu jednostki pływającej”
napisana pod kierunkiem:
dr inż. Robert Krupiński
jest w całości moim samodzielnym autorskim opracowaniem, sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych.
Złożona w Dziekanacie Wydziału Elektrycznego treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej.

Oświadczam ponadto, że złożona w Dziekanacie praca dyplomowa, ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

.....
podpis dyplomanta

Szczecin, dn. ..... r.

Streszczenie pracy

Celem pracy były stworzenie aplikacji prezentującej aktualne parametry jednostki pływającej. Zakres wykonanych prac obejmował przegląd aktualnie stosowanych rozwiązań, zapoznanie się i opis protokołu NMEA (National Marine Electronics Association) oraz część oprogramowania aplikacji mobilnej. Dla testów skorzystano z dostępnego w sieci symulatora jednostki wysyłającej dane zakodowane protokołem NMEA. Głównymi z napotkanych problemów były – ilość oraz dokumentacja komend protokołu NMEA, przygotowanie oprogramowania pod łatwy rozwój o wsparcie dla kolejnych komend. Podczas wyboru technologii kierowano się głównie osobistym doświadczeniem oraz umiejętnościami.

Słowa kluczowe

NMEA, aplikacja mobilna, wizualizacja

Abstract

The aim of the work was to create a mobile application for presenting actual parameters of the vessel's voyage. The scope of work included an overview of actually used solutions, acquiring the knowledge about NMEA (National Marine Electronics Association) protocol and describing it, creating software for mobile application. For testing purposes, NMEA Simulator (available on the internet) which sends NMEA encoded data was used. The main encountered issues were – the amount and documentation of NMEA protocol sentences, preparing the software for easy expansion by additional sentences. The choice of the technological stack was mainly based on personal experience and skills.

Keywords

NMEA, mobile application, visualization

Spis treści

Wprowadzenie

Komunikacja między urządzeniami elektronicznymi jest obecnie standardem. Zależnie od obszaru działań wspomnianych urządzeń stosowane są odmienne protokoły. Jednostki pływające, których dotyczy ta praca korzystają ze standardu *NMEA*. Brak jednak narzędzi pozwalających w łatwy sposób przedstawić człowiekowi dane zakodowane tym protokołem. Obecnie aplikacje mobilne stanowią ogromną część rynku oprogramowania. Niemal każdy posiada smartphone i korzysta z aplikacji dla systemów takich jak *Android* czy *iOS*. W kolejnych rozdziałach został przedstawiony proces tworzenia takiej aplikacji. Równie dużą, a nawet większą, rolę odgrywa obecnie komunikacja sieciowa, większość znanych nam stron i aplikacji bazuje na architekturze *REST*, popularność zyskuje *GraphQL*, a do określonych zadań stosuje się protokoły takie jak *WebSocket*. Właśnie ten ostatni został wykorzystany w pracy, co zostanie przedstawione w kolejnych rozdziałach.

Cel pracy

Celem pracy jest stworzenie aplikacji mobilnej prezentującej aktualne parametry jednostki pływającej.

Zakres pracy

Zakres pracy obejmuje przegląd aktualnie stosowanych rozwiązań w zakresie parametrów jednostek pływających oraz metod ich prezentacji, zapoznanie się i opisanie standardu dla protokołu *NMEA* (National Marine Electronics Association) oraz stworzenie aplikacji mobilnej na Smartphone wraz z *GUI* (Graphical User Interface) prezentującej aktualne parametry jednostki pływającej z wykorzystaniem sieci *WiFi* jednostki pływającej. Celem dodatkowym jest przetestowanie aplikacji w warunkach rzeczywistych na jednostce pływającej.

ROZDZIAŁ 1

Protokół NMEA

Komunikacja między morskimi urządzeniami elektronicznymi wykorzystuje protokół *NMEA*. Wyróżniamy dwie wersje tego protokołu – *NMEA 0183* oraz *NMEA 2000* [2, 3]. Obecnie interfejs *NMEA 0183* zastępowany jest przez nowszy *NMEA 2000*, jednak praca dotyczy będącego przez lata standardem oraz ciągle wykorzystywanego starszego protokołu. Odniesienia do protokołu *NMEA* w dalszej części pracy oznaczać będą wersję 0183.

1.1. Format komendy NMEA

Na podstawie [5] należy wydzielić podstawowe typy komend: talker sentences (źródeł danych), proprietary sentences (zdefiniowanych przez poszczególnych producentów), query sentences (zapytań). Pierwszy typ, na którym skupia się ta praca, obejmuje sygnały GPS, dane z urządzeń wbudowanych w jednostkę, czujników. Typ drugi pozwala indywidualnym producentom zdefiniować własne formaty, linie danych tego typu zaczynają się od znaków „$P" poprzedzających trzy literowy identyfikator producenta. Następnie zawarte są dane, o których decyduje producent. Ostatni typ pozwala urządzeniu zażądać wybranego rodzaju komendy od innego urządzenia. Ogólne formaty wyglądają następująco:

1. Talker Sentence - `$ttsss,d1,d2,....<CR><LF>`, gdzie „tt" oznacza identyfikator nadawcy, „sss" identyfikator komendy, a „d1, d2..." kolejne pola danych.

2. Proprietary Sentence - `$Psss,d1,d,2,...<CR><LF>`, gdzie „sss" oznacza identyfikator producenta, a „d1,d2..." kolejne pola danych.

3. Query Sentence - `$ttllQ,sss,<CR><LF>`, gdzie „tt" oznacza identyfikator żądającego, „ll" identyfikator urządzenia, do którego zostało skierowane zapytanie, a „sss" identyfikator żądanej komendy.

Każda linia danych nadana w protokole *NMEA* zaczyna się od znaku „$", dane oddzielone są przecinkami, a komenda kończy się sumą kontrolną poprzedzoną znakiem „*".

1.2. Identyfikatory nadawców

Identyfikatory nadawców rozróżniają linie danych *NMEA* ze względu na źródło informacji, o których mowa w danej komendzie.

Kod źródłowy 1.1. Identyfikatory nadawców
Źródło: Opracowanie własne [18, 14]

```
enum TalkerIdentifiers {
    AG= 'AG', AP= 'AP', CD= 'CD', CR= 'CR', CS= 'CS', CT= 'CT', CV= 'CV',
    CX= 'CX', DF= 'DF', EC= 'EC', EP= 'EP', ER= 'ER', GP= 'GP', HC= 'HC',
    HE= 'HE', HN= 'HN', II= 'II', IN= 'IN', LC= 'LC', RA= 'RA', SD= 'SD',
    SN= 'SN', SS= 'SS', TI= 'TI', VD= 'VD', DM= 'DM', VW= 'VW', WI= 'WI',
    YX= 'YX', ZA= 'ZA', ZC= 'ZC', ZQ= 'ZQ', ZV= 'ZV'
}
```

Kod źródłowy 1.2. Deskryptory nadawców
Źródło: Opracowanie własne na podstawie [18, 14]

```
const TalkerDescriptors: Record<TalkerIdentifiers, string> = {
    [TalkerIdentifiers.AG]: 'Autopilot – General',
    [TalkerIdentifiers.AP]: 'Autopilot – Magnetic',
    [TalkerIdentifiers.CD]: 'Communications – Digital Selective Calling (DSC)',
    // deskryptor uległa skróceniu w celach edytorskich
    [TalkerIdentifiers.ZC]: 'Timekeeper – Chronometer',
    [TalkerIdentifiers.ZQ]: 'Timekeeper – Quartz',
    [TalkerIdentifiers.ZV]: 'Radio Update, WWV or WWVH'
};
```

1.3. Identyfikatory komend

Lista wspieranych przez aplikację identyfikatorów komend.

Kod źródłowy 1.3. Identyfikatory komend
Źródło: Opracowanie własne na podstawie [18, 14]

```
enum SentenceIdentifiers {
    AAM= 'AAM', ALM= 'ALM', APA= 'APA', APB= 'APB', ASD= 'ASD', BEC= 'BEC',
    BOD= 'BOD', BWC= 'BWC', BWR= 'BWR', BWW= 'BWW', DBK= 'DBK', DBS= 'DBS',
    DBT= 'DBT', DCN= 'DCN', DPT= 'DPT', DSC= 'DSC', DSE= 'DSE', DSI= 'DSI',
    DSR= 'DSR', DTM= 'DTM', FSI= 'FSI', GBS= 'GBS', GGA= 'GGA', GLC= 'GLC',
    GLL= 'GLL', GRS= 'GRS', GST= 'GST', GSA= 'GSA', GSV= 'GSV', GTD= 'GTD',
    GXA= 'GXA', HDG= 'HDG', HDM= 'HDM', HDT= 'HDT', HSC= 'HSC', LCD= 'LCD',
    MSK= 'MSK', MSS= 'MSS', MWD= 'MWD', MTW= 'MTW', MWV= 'MWV', OLN= 'OLN',
    OSD= 'OSD', ROO= 'ROO', RMA= 'RMA', RMB= 'RMB', RMC= 'RMC', ROT= 'ROT',
    RPM= 'RPM', RSA= 'RSA', RSD= 'RSD', RTE= 'RTE', SFI= 'SFI', STN= 'STN',
    TLL= 'TLL', TRF= 'TRF', TTM= 'TTM', VBW= 'VBW', VDR= 'VDR', VHW= 'VHW',
    VLW= 'VLW', VPW= 'VPW', VTG= 'VTG', VWR= 'VWR', WCV= 'WCV', WDC= 'WDC',
    WDR= 'WDR', WNC= 'WNC', WPL= 'WPL', XDR= 'XDR', XTE= 'XTE', XTR= 'XTR',
    ZDA= 'ZDA', ZDL= 'ZDL', ZFO= 'ZFO', ZTG= 'ZTG'
```

```
15 }
```

Kod źródłowy 1.4. Deskryptory komend
Źródło: Opracowanie własne na podstawie [18, 14]

```
1  const SentencesDescriptions: Record<SentenceIdentifiers, string> = {
2      [SentenceIdentifiers.AAM]: 'Waypoint Arrival Alarm',
3      [SentenceIdentifiers.ALM]: 'GPS Almanac Data',
4      [SentenceIdentifiers.APB]: 'Autopilot Sentence "B"',
5      // deskryptor uległa skróceniu w celach edytorskich
6      [SentenceIdentifiers.ZDL]: 'Time & Distance to Variable Point',
7      [SentenceIdentifiers.ZFO]: 'UTC & Time from Origin Waypoint',
8      [SentenceIdentifiers.ZTG]: 'UTC & Time to Destination Waypoint'
9  };
```

1.4. Komendy ujęte w aplikacji

Z racji bardzo dużej ilości istniejących komend protokołu *NMEA* do realizacji pracy inżynierskiej zostały wybrane linie danych uwzględnione w używanym symulatorze *NMEA Simulator*. Oznacza to pełne dekodowanie oraz wyświetlanie danych w nich zawartych, jednak aplikacja została stworzona w taki sposób, aby dodawanie kolejnych komend było jak najprostsze. Szczegóły zostaną omówione w dalszym rozdziale. Dane dostarczone przez symulator opisują zarówno położenie jak i dane na temat samej jednostki. Na ich podstawie zbudowany został interfejs graficzny aplikacji. Pełna lista znajduje się w tabeli 1.1.

Tabela 1.1. Komendy *NMEA* w pełni dekodowane przez aplikacje
Źródło: Opracowanie własne

Identyfikator nadawcy	Nadawca	Identyfikator komendy	Komenda
SD	Sounder	DBT	Depth below transducer
GP	Global Positioning System (GPS)	GGA	Global Positioning System Fix Data
GP	Global Positioning System (GPS)	GSA	GPS DOP and active satellites
II	Integrated instrumentation	MTW	Mean Temperature of Water
WI	Weather Instruments	MWD	Wind Direction and Speed
WI	Weather Instruments	MWV	Wind Speed and Angle
GP	Global Positioning System (GPS)	RMC	Recommended Minimum Navigation Information
II	Integrated instrumentation	VHW	Water speed and heading

ROZDZIAŁ 2

Przegląd dostępnych rozwiązań

Gałąź przemysłu (biznesu) morskiego może być uważana za sektor, gdzie bezpieczeństwo gra odgórną rolę, stąd standard *NMEA* definiuje także warstwę elektryczną interfejsu i w tym przypadku jest to *EIA-422* - czyli połączenie z jednym masterem oraz wieloma slave'ami. Istnieje także standard, który mówi o pakowaniu ramek *NMEA* w datagram UDP - ale nie opisuje go formalnie żaden dokument.

Chcąc dokonać kategoryzacji istniejących rozwiązań należałoby zastanowić się nad ograniczeniem ich spektrum, ponieważ wiele firm oferuje rozwiązania programowo-sprzętowe, które są gateway'em między fizycznym połączeniem *EIA-422* (zdefiniowanym przez standard) - a siecią *WiFi*. Przykładem takiego rozwiązania jest produkt *WLN10 Smart* firmy *DIGITAL YACHT* [19] pozwalający na udostępnianie danych *NMEA* przez *WiFi*. Koszt tego produktu to 150£. W dalszej części omówione zostaną aplikacje wizualizujące owe dane
.

2.1. Rozwiązania darmowe

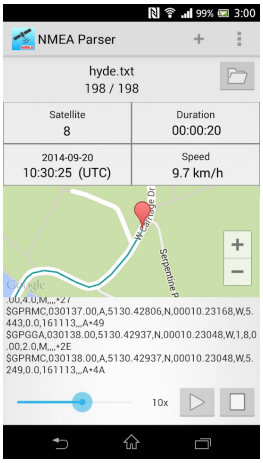
Darmowe rozwiązania dostępne dla urządzeń mobilnych są w większości przypadków mocno ograniczone, niewspierane przez twórcę lub twórców, obarczone licznymi błędami. Przykładowe aplikacje:

1. *NMEA Tools* [10] Rysunek 2.1 w wersji darmowej pozwala na odczyt danych *NMEA* z pliku tekstowego, co mocno ograniczna użyteczeność tego rozwiązania.

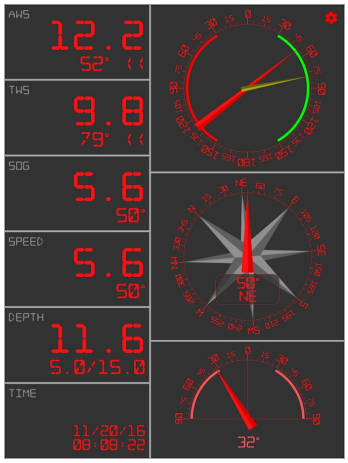
2. *NavMonitor* [6] Rysunek 2.2 wizualizuje dane dotyczące między innymi wiatru, pozycji, prędkości jednostki, czy temperatury oraz głębokości wody. Nie posiada jednak przedstawienia obiektu na mapie, co jest znaczącą wadą.

3. *SeaWi Marine* [13] Rysunek 2.3 jest jednym z najbardziej rozbudowanych rozwiązań. Aplikacja oferuje wiele funkcjonalności, w tym wizualzację danych w formie zarówno graficznej, czyli przedstawione na mapie, jak i tekstowej w postaci zdekodowanych informacji. Pozwala na tworzenie punktów kontrolnych, konfigurację

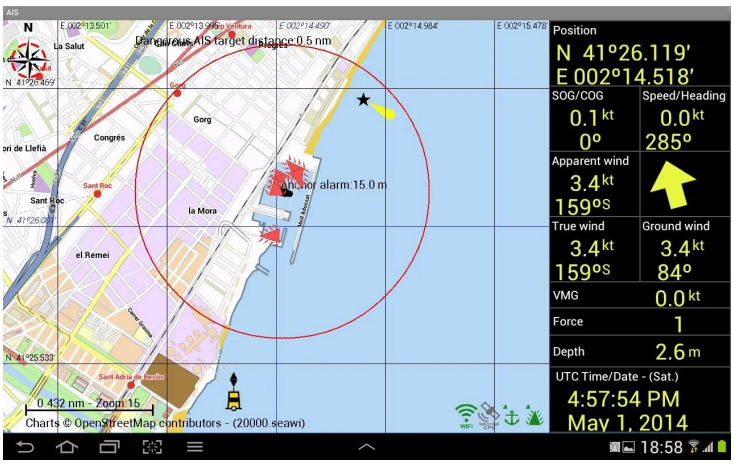
alarmów, a w tym także możliwość wysłania wiadomości SMS w przypadku ich wystąpienia, oraz wiele innych. Wadą jest niekompletny zestaw map, na chwilę obecną obejmujący Europę, Amerykę Północną, Morze Śródziemne, Morze Czarne, Karaiby, Australię, Nową Zelandię, Pacyfik oraz Afrykę Północną. Twórcy zapewniają, że mapy dla reszty świata zostaną dostarczone w przyszłości.



Rysunek 2.1. NMEA Tools
Źródło: https://play.google.com/store/apps/details?id=com.peterhohsy.nmeatools



Rysunek 2.2. NavMonitor
Źródło: https://play.google.com/store/apps/details?id=com.aboni.boatinga



Rysunek 2.3. SeaWi Marine
Źródło: https://play.google.com/store/apps/details?id=net.seawimarine.activities

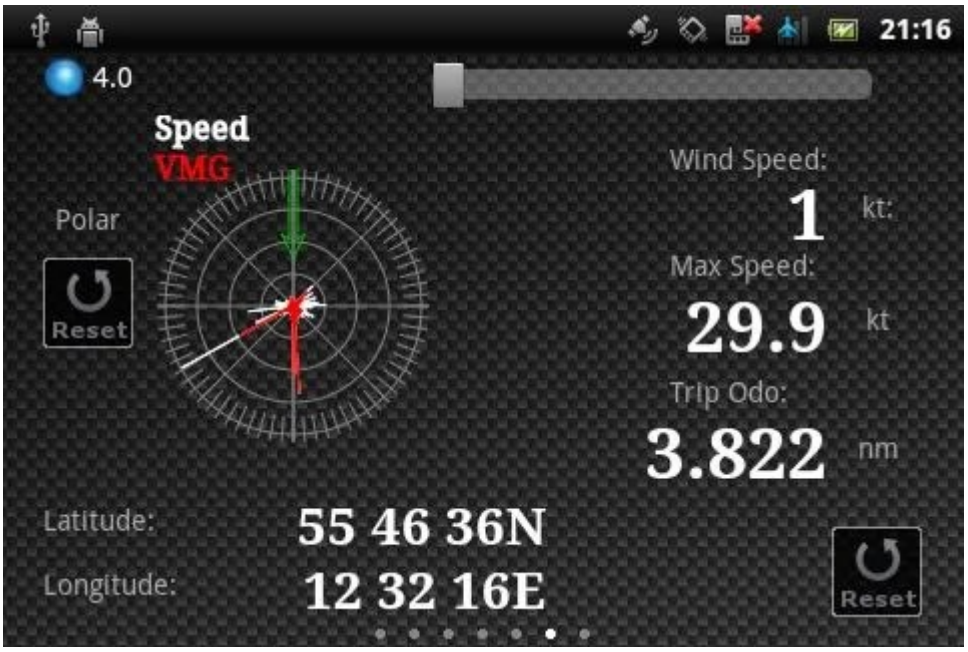
2.2. Rozwiązania komercyjne

Płatne rozwiązania zrealizowane są w modelu jednorazowej płatności za oprogramowania lub konieczności wykupienia czasowej subskrypcji, aby uzyskać dostęp do pełni funkcjonalności. Najczęściej subskrypcją objęty jest dostęp do map. Przykładowe aplikacje:

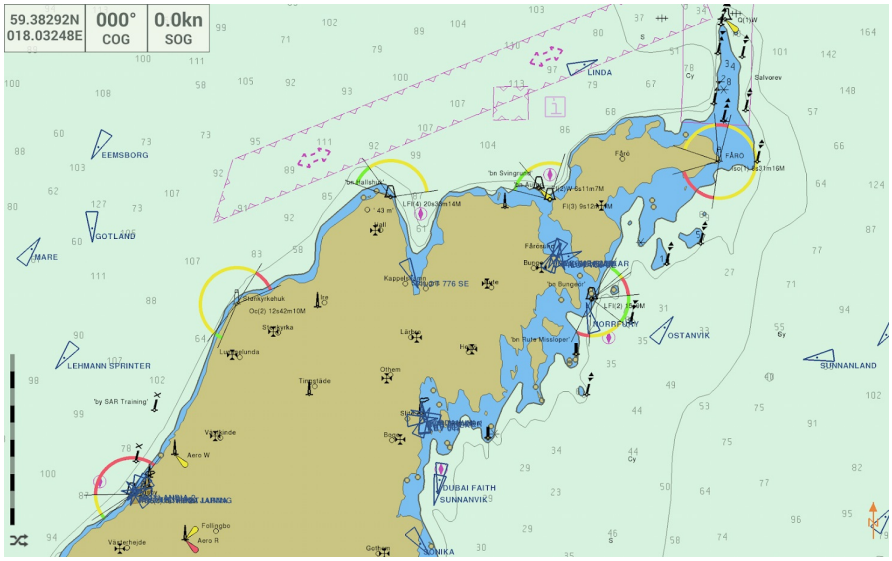
1. *iRegatta Pro* [12] Rysunek 2.4 pozwala na odczyt danych dotyczących między innymi wiatru, prędkości jednostki i jej położenia. Oprócz komunikacji sieciowej wspiera także komunikację przy wykorzystaniu *BlueTooth*. Brakuje wizualizacji obiektu na mapie. Cena tej aplikacji wynosi 69,99 złotych.

2. *Seapilot* [15] Rysunek 2.5 oferuje przede wszystkim pełen detali widok mapy, przeznaczony do specjalistycznego użycia. Podstawowe informacje na temat jednostki widoczne są w formie tekstowej. Dostęp do pełni funkcjonalności wymaga wykupienia licencji na stronie producenta. Cena wynosi 39,99$ rocznie. Dodatkowo należy wykupić mapy obejmujące interesujące użytkownika rejony, również opłacane rocznie.

3. *NMEAremote* [20] Rysunek 2.6 podobnie jak konkurencja przedstawienie zdekodowanych danych w przyjaznej dla użytkownika formie. Brakuje przedstawienia obiektu na mapie. Twórcy zapewniają listę wspieranych komend protokołu *NMEA 0183* oraz *NMEA 2000*. Jest to odpowiednio trzydzieści siedem oraz dziewiętnaście typów komend. Kosz aplikacji wynosi 17,99$.



Rysunek 2.4. *iRegatta*
Źródło: https://play.google.com/store/apps/details?id=dk.letscreate.aRegatta



Rysunek 2.5. *Seapilot*
Źródło: https://play.google.com/store/apps/details?id=com.seapilot.android

Rysunek 2.6. *NMEAremote*
Źródło: https://apps.apple.com/us/app/nmearemote/id412806204

2.3. Podsumowanie

Dostępne rozwiązania nie są perfekcyjne, jednak w przybliżeniu pokrywają się z celem pracy. Wiele dostępnych aplikacji zostało pomięte w tym rozdziale z uwagi na olbrzymią liczbę negatywnych komentarzy użytkowników, często implikujących całkowitą niezdatność oprogramowania do użytku. Aby uzyskać w pełni spersonalizowane rozwiązanie należy je stworzyć. W kolejnym rozdziale przedstawiony został sposób doboru narzędzi oraz motywy stojące za poszczególnymi wyborami, a także proces tworzenia aplikacji wraz z wyszczególnieniem głównych konceptów kodu źródłowego.

ROZDZIAŁ 3

Aplikacja Mobilna

3.1. Język programowania

Domena tworzenia oprogramowania jest bardzo obszerna i może być dzielona wedle różnych kryteriów. Pierwszą przychodzącą na myśl kategoryzacją jest podział ze względu na różne języki programowania, te najbardziej popularne z nich, czyli *C*, *C++*, *Python*, *Java*, *C#*, *JavaScript* są wystarczająco odmienne, aby było to adekwatne kryterium. Innym, prawdopodobnie najbardziej szczegółowym zbiorem kryteriów są kryteria techniczne. Najważniejsze z nich to:

1. Sposób egzekucji (czyli sposób, w jaki wykonywany jest dany program komputerowy):
    - Języki interpretowalne, czyli takie, w których środowiskiem działania jest wirtualna maszyna a kod w sposób dynamiczny wykonywany jest linijka po linijce bez translacji na kod maszynowy danego procesora
    - Języki kompilowalne, czyli takie, w których do uruchomienia programu wymagana jest zamiana jego postaci na kod wykonywalny, zrozumiały dla danej architektury
    - Języki hybrydowe, czyli będącę połączeniem powyższych dwóch - przykładowo, niektóre fragmenty kodu są kompilowane (najczęściej tam gdzie zależy nam na optymalności rozwiązania)
2. Wspierany paradygmat programowania, głównie:
    - Structural Programming (SP), czyli paradygmat, który za podstawowy element budulcowy uznaje funkcje (procedury), które są zbiorem instrukcji, które mają wpływ na stan programu. Całość jest hierarchiczna oraz sekwencyjna. Pierwszym najszerzej rozpowszechnionym, uznanym i do dzisiaj używanym językiem strukturalnym jest język *C*

- Object Oriented Programming (OOP), czyli próba opisania oprogramowania w sposób jak najbardziej oddający otaczający nas świat. Klasy oraz obiekty stanowią warstwę abstrakcji reprezentującą dane procesy oraz podmioty. Wedle definicji możemy wymienić cztery filary programowania obiektowego: abstrakcja, hermetyzacja, polimorfizm oraz dziedziczenie. Obecnie jest to dominujący na rynku paradygmat, cechujący przede wszystkim *C++*, *C#* i *Java*. Nie jest jednak pozbawiony wad oraz problemów wynikających z samych jego założeń.
- Functional Programming (FP), czyli paradygmat można rzec inspirowany matematyką. W jego podstawowych założeniach kod źródłowy składałby się z wielu przewidywalnych funkcji, nazywanych 'pure functions'. Takie funkcje muszą zawsze zwracać dokładnie taki sam rezultat przy takich samych danych wejściowych, niezależnie od jakichkolwiek innych czynników. Tak restrykcyjne założenie uniemożliwia wywoływanie jakichkolwiek efektów ubocznych (ang. side effects) w kodzie, czego najbardziej dobitnym przykładem są operacje I/O będące podstawą znakomitej części oprogramowań. Chciałbym tutaj zaznaczyć, że wbrew rozumowaniu wielu inżynierów oprogramowania, nie oznacza niemożliwości wykorzystania efektów ubocznych. W językach programowania funkcyjnego jest to w pełni możliwe, a sam paradygmat dopuszcza łamanie założeń w niezbędnych operacjach. Programowanie funkcyjne posiada wysoki próg wejścia, może wydawać się nieintuicyjne i skomplikowane oraz przestarzałe. Zyskuje jednak ponownie na popularności, powoli stając się konkurentem dla programowania obiektowego. Przykładowe języki reprezentujące ten paradygmat to *Haskell*, *Lisp*.

3. Typowanie:

    - Statyczne, inaczej zwane twardo typowanym, języki tego typu wymagają od programisty znajomości i deklaracji typów danych. Cechujące wszystkie najstarsze języki programowania, a także większość obecnie używanych. Zapewnia możliwość znalezienia błędów ludzkich na poziomie kompilacji, czyli bez konieczności uruchamiania i sprawdzania oprogramowania. Przykładowe języki: *C*, *C++*, *Java*, *C#*
    - Dynamiczne, inaczej zwane miękko typowanym, języki tego typu nie posiadają oznaczeń typów danych, takich jak np. 'string' czy 'float', lub nie wymagają jawnego ich podawania. Zaletą takiej konwencji jest mniejsza ilość kodu oraz większa przystępność dla początkujących programistów. Przykładowe języki: *Python*, *JavaScript*.

Kolejnym logicznym kryterium jest docelowa platforma, dzięki czemu wyróżnimy programowanie wbudowane, web'owe, desktop'owe, mobilne. Należy zauważyć, że w tym wypadku niektóre z wyżej wymienionych języków trafią do wspólnego zbioru. Dla przykładu zarówno *Python* jak i *JavaScript* można nazwać językami web'owymi (lecz nie tylko). Pojawia się tutaj pewien problem, którym jest skalowanie oprogramowania. Jeśli firma posiada obiecujące oprogramowanie to ograniczenie grona odbiorców do danej platformy

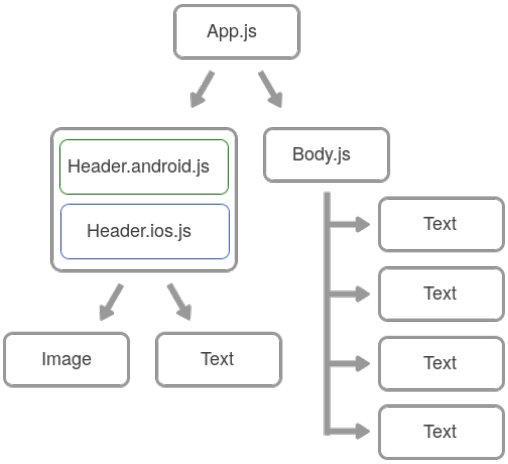
jest bardzo niekorzystne. Z między innymi tego powodu, języki programowania ewoluowały stając się wieloplatformowe. Dla przykładu *Java*, będąca jednym z najpopularniejszych języków na rynku, została stworzona z założeniem wsparcia różnych platform bez potrzeby ingerowania w kod.

Wyżej wymienione kryteria mają olbrzymie znaczenie podczas wyboru języka dla danego projektu. W przypadku aplikacji będącej podmiotem pracy inżynierskiej zdecydowano się na *TypeScript*, czyli nadzbiór języka *JavaScript*. Podobnie jak *JavaScript* wspiera on zarówno programowanie funkcyjne jak i obiektowe. Natomiast najistotniejszymi różnicami są kompilacja oraz wsparcie dla twardego typowania. Kompilowany jest on do czystego *JavaScript*, który następnie interpretowany jest przez środowisko klienckie, czyli np. *node.js* lub przeglądarka internetowa. Dzięki temu w przypadku, gdy nie jesteśmy w stanie lub nie chcemy określać typu danych - jest to całkowicie dozwolone. Przykładowo - dane dostarczane z zewnętrznej biblioteki, której twórca nie zapewnia ich struktury. W przypadku niektórych frameworków wymuszone jest użycie języka *TypeScript*, w innych jest to opcjonalne. Przykładowo jeden z najpopularniejszych frameworków do tworzenia *GUI* o nazwie *Angular* wymusza używanie tej technologii. Podobnie jak zyskujący coraz większą popularność *NestJS*, służący do tworzenia aplikacji serwerowych. Po wybraniu języka programowania przychodzi czas na wybór frameworka, który usprawni pracę.

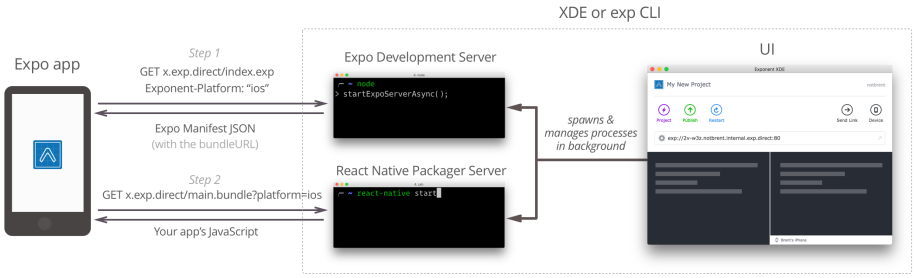
3.2. Framework

Framework dostarcza programiście szkielet aplikacji, a także metody, komponenty, serwisy oraz składnię do ich wykorzystywania. Wedle statystyki przedstawionej na Rysunek 3.3 [11] najpopularniejszym z frameworków do tworzenia aplikacji webowych po stronie klienta jest *ReactJs*, konkurujący z frameworkami nazwanymi *Angular* oraz *Vue*. Nie udostępnia on jednak możliwości tworzenia natywnych aplikacji mobilnych na systemy *Android* oraz *iOS*. Na te potrzeby powstał *ReactNative*. Jest to odpowiednik wspomnianego *ReactJs* dedykowany dla platform mobilnych. Podstawowe założenia takie jak kompozycja komponentów, przesyłanie danych, czy cykl życia komponentu pozostają bez zmian. Znaczące różnice wynikają z faktu, że w odróżnieniu do przeglądarek, aplikacje mobilne nie wspierają składni *HTML* oraz *CSS*. Na podstawie [9] w miejsce pierwszego *ReactNative* dostarcza podstawowe komponenty pozwalające na budowanie widoku. Odpowiednikiem elementu <*div /*> w składni *HTML* jest w tym wypadku element <*View /*>. Olbrzymią zaletą tego frameworka jest fakt, że w podanym przykładzie element <*View /*> zostanie wyrenderowany przy użyciu natywnego *API* platformy, co zapewnia dużo lepszą wydajność w porównaniu do przykładowo frameworku*Cordova*, który do renderowania wykorzystuje *WebView*. Rysunek 3.1 Arkuszy Styli *CSS* zastąpione są natomiast przez obiekty *Stylesheets*. Zaletą takiego podejścia jest możliwość przeniesienia aplikacji napisanej w *ReactNative* na *ReactJS* przy stosunkowo niewielkim nakładzie pracy, gdyż znaczna część kodu może zostać użyta ponownie. Dodatkowo został wykorzystany framework oraz platforma *Expo*. Wedle [7] jest to zbiór narzędzi oraz serwisów zbudowanych wokół *ReactNative*, mający na celu przyspieszenie procesu tworzenia aplikacji. Dostarcza on między innymi przydatne warstwy abstrakcji nałożone na natywne *API*, przykładowo do-

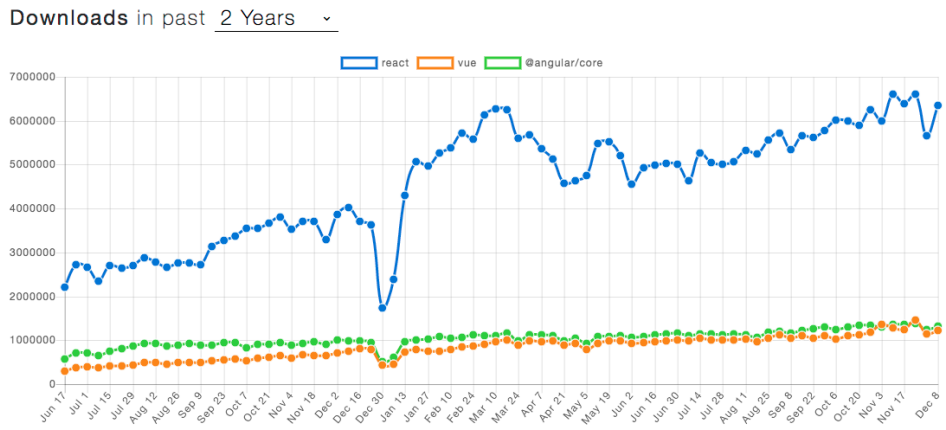
stęp do kamery, czy żyroskopu. Dodatkowo zapewnia rozbudowane *CLI* (Command Line Interface). Jedną z jego możliwości jest niezwykle proste budowanie kodu na określoną platformę oraz serwer deweloperski z funkcją 'hot-reload'. Oznacza to, że zapisanie zmiany w kodzie natychmiast wywoła proces budowy odpowiednich plików, a programista ma możliwość obserwowania zmian na bieżąco używając symulatora urządzenia mobilnego lub rzeczywistego urządzenia. Sam proces połączenia z serwerem deweloperskim jest niesłychanie prosty. *Expo* dostarcza adres url zawierający pliki źródłowe. Dla ułatwienia jest on również przedstawiony jak kod QR. Dowolne urządzenie mobilne posiadające aplikację kliencką *Expo* po zeskanowaniu tego kodu uruchomi wersję deweloperską aplikacji. Rysunek 3.2 Oczywiście takie udogodnienia nakładają pewne ograniczenia, przez co dla dużych projektów z rozbudowaną logiką biznesową Expo może okazać się niewłaściwym wyborem. Jednak analiza pod kątem potrzeb omawianej pracy pozwoliła na uzasadniony wybór tego narzędzia.



Rysunek 3.1. Przekształcenia kodu *ReactNative* na natywne API platformy
Źródło: https://reactnative.dev/



Rysunek 3.2. Schemat lokalnego działania *Expo*
Źródło: https://docs.expo.io/versions/v36.0.0/workflow/how-expo-works



Rysunek 3.3. Popularność frameworków *JavaScript*
Źródło: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

3.3. Narzędzia wspomagające

JavaScript jest niezwykle dynamicznie rozwijającym się językiem, co sprawia że wiele środowisk nie wspiera najnowszej składni. Aby rozwiązać ten problem skorzystano z transpilera *Babel*. Przekształca on najnowszą składnię języka na wspierany przez środowisko odpowiednik. Dzięki temu rozwiązaniu programista może korzystać z wszelkich udogodnień wprowadzonych wraz z kolejnymi wersjami języka nie martwiąc się przy tym o wsparcie starszych platform. Zgodnie z [4] narzędzie to posiada wsparcie dla *TypeScript* oraz składni *ReactNative*.

Kod źródłowy 3.1. Babel - przykładowe transformacje kodu
Źródło: Opracowanie własne

```
// input
enum myEnum {
  first=1,
  second=2,
  third=3
}
//output
var myEnum;

(function (myEnum) {
  myEnum[myEnum["first"] = 1] = "first";
  myEnum[myEnum["second"] = 2] = "second";
  myEnum[myEnum["third"] = 3] = "third";
})(myEnum || (myEnum = {}));

//input
const arr: number[] = [1,2,3]
arr.map(el => el * 2);
//output
var arr = [1, 2, 3];
```

```
arr.map(function (el) {
  return el * 2;
});

//input
const obj: Record<string, string> = {prop1: 'value', prop2: 'value'};
const { prop1, prop2 } = obj;
//output
var obj = {
  prop1: 'value',
  prop2: 'value'
};
var prop1 = obj.prop1,
    prop2 = obj.prop2;

//input

const templateString: string = `First variable is ${prop1} and second is ${prop2}`;
//output
var templateString = "First variable is ".concat(prop1, " and second is
").concat(prop2);

//input
const ReactComponent: React.FC = () => <div>Text</div>
//output
var ReactComponent = function ReactComponent() {
  return /*#__PURE__*/React.createElement("div", null, "Text");
};
```

Na potrzeby projektu została stworzona minimalistyczna konfiguracja zalecana przez twórców Expo.

Kod źródłowy 3.2. babel.config.js - plik konfiguracyjny
Źródło: Opracowanie własne

```
module.exports = function(api) {
  api.cache(true);
  return {
    presets: ['babel-preset-expo'],
  };
};
```

Dla zachowania standardów jakości oraz jednolitego stylu kodu zostało wykorzystane narzędzie do analizy statycznej. Za powszechny standard uznaje się oprogramowanie *ESLint*. Zapewnia ono olbrzymią ilość konfigurowalnych zasad, dotyczących zarówno stylu kodu takich jak maksymalna długość linii czy sposób indentacji, jak i jego logiki. Warto wspomnieć, że narzędzie to posiada wsparcie najpopularniejszych środowisk programistycznych. W przypadku języka *JavaScript/TypeScript* są to *Visual Studio Code*, *WebStorm*, *IntelliJ IDEA*. Dzięki temu błędy wykrywane są automatycznie podczas pisania kodu. W przypadku projektów komercyjnych tworzonych przez zespół programistów

częstą praktyką jest tworzenie tak zwanych 'git hooków', które mają na celu sprawdzenie kodu przez między innymi linter przed utworzeniem commita, a w przypadku błędów uniemożliwienie go. Omawiany projekt korzysta ze zbioru zasad dostarczonego przez firmę *Airbnb* skonfigurowanych wedle osobistych preferencji.

3.4. Komunikacja z serwerem

Sposób komunikacji z serwisami sieciowymi zależy od typu *API* danego serwisu. Najczęściej spotykany to *REST API* oparty na protokole http. Taki serwis opiera się na schemacie, w którym klient wysyła zapytanie i oczekuje na odpowiedź. Zapytanie http w momencie wywołania otwiera połączenie, a po przesłaniu odpowiedzi lub określonym (krótkim) czasie je zamyka. Ciągłe otwieranie i zamykanie połączenia jest olbrzymim problemem dla aplikacji wymagających stałej aktualizacji danych. W przypadku systemu wizualizującego możliwie aktualny stan modułów konieczna jest ciągła aktualizacja danych. Dla takich potrzeb wykorzystywany jest protokół *WebSocket*, umożliwiający otworzenie połączenia między serwerem a klientem oraz komunikację w obie strony. Dzięki temu klient jest w stanie nasłuchiwać dostarczanych wiadomości bez konieczności tworzenia i wysyłania zapytania. Korzystając z tego samego połączenia klient ma możliwość wysłania wiadomości do serwera. Wymienione cechy czynią protokół *WebSocket* idealnym wyborem dla aplikacji wymagającej możliwie najbardziej aktualnych danych. Wykorzystany do testów *NMEA Simulator* zapewnia funkcję serwera tego typu. Klient nie posiada konieczności wysyłania informacji do nadawcy, obsługuje jedynie ich odbiór. Wiersze od siódmego do dziewiątego (kod źródłowy 3.3) zawierają implementację uchytwu (ang. handler) obsługującego odbiór wiadomości.

Kod źródłowy 3.3. Metoda tworząca połączenie *WebSocket*
Źródło: Opracowanie własne

```
1   const connectWebsocket = useCallback(() => {
2         const ws = new WebSocket(state.url);
3         ws.onopen = () => {
4             dispatch(NmeaConnectorActions.setConnected(true));
5         };
6
7         ws.onmessage = (e) => {
8             dispatch(NmeaConnectorActions.setData(e.data));
9         };
10
11        ws.onerror = () => {
12            dispatch(NmeaConnectorActions.setConnected(false));
13            ws.close();
14        };
15
16        ws.onclose = () => {
17            // refresh connection after 10s
18            setTimeout(() => connectWebsocket(), 10000);
19        };
```

```
20          return (() => {
21              ws.close();
22          });
23      }, [state.url]);
```

3.5. Dekoder komend *NMEA*

Otrzymywane z serwera dane są pojedynczym ciągiem znaków zawierającym wszystkie komendy *NMEA* oddzielone znakiem końca linii. Pierwszym krokiem niezbędnym dalszego przetwarzania jest więc rozbicie otrzymanej odpowiedzi na tablicę zawierającą tekstową reprezentację pojedynczej linii danych. Następnie dla każdego z elementów wywoływana jest metoda zwracająca obiekt reprezentujący ramkę *NMEA* o określonej strukturze. Dołożono wszelkich starań, aby rozwiązanie było jak najbardziej generyczne i łatwe w rozbudowie. Proces dekodowania rozpoczyna się od walidacji sumy kontrolnej komendy (wiersze od drugiego do czwartego (kod źródłowy 3.4). W przpadku powodzenia wyodrębniane są identyfikatory nadawcy oraz linii danych (wiersze jedenaście oraz dwanaście (kod źródłowy 3.4)). Na podstawie identyfikatora komendy uzyskiwany jest jego format (wiersz czternasty kod źródłowy 3.4 oraz kod źródłowy 3.5). Format określa typ danych każdego pola, co umożliwiło stworzenie struktury określającej sposób dekodowania (kod źródłowy 3.6). Dzięki takiemu rozwiązaniu programista w łatwy sposób jest w stanie rozszerzyć obsługiwane typy pól. Posiadając informację o strukturze komendy oraz o wartościach pól możliwe jest stworzenie obiektu opisującego ramkę *NMEA*. Oznacza to także zakończenie części generycznej dla wszystkich typów komend, gdyż format danych w poszczególnych polach nie określa ich znaczenia. Aby otrzymać w pełni wartościowe informacje należy opisać strukturę poszczególnych rozkazów oraz metodę przenoszącą surowe dane do właściwych pól tej struktury. Taki interfejs oraz metoda są wykorzystywane do utworzenia obiektu pakietu (wiersz dwudziesty drugi kod źródłowy 3.4) reprezentującego w pełni odkodowaną komendę *NMEA*. Przykładowy plik dla linii danych DBT przedstawia kod źródłowy 3.7. Rozszerzenie aplikacji o kolejny typ komendy wymaga jedynie utworzenia takiego pliku oraz dodanie utworzonego interfejsu i metody do kolekcji wykorzystywanych przy parsowaniu.

Kod źródłowy 3.4. Metoda parsująca komendy *NMEA*
Źródło: Opracowanie własne

```
1  export const parseNmeaSentence = (sentence: string): Packet => {
2      if (!validateNmeaChecksum(sentence)) {
3          throw Error(`Invalid sentence: "${sentence}".`);
4      }
5      const fields = sentence.split('*')[0].split(',');
6
7      if (fields[0].charAt(1) === 'P') {
8          throw Error('Proprietary sentences not suppoeted');
9      }
10
```

```
    const talkerId = TalkerIdentifiers[fields[0].substr(1, 2)];
    const sentenceId = SentenceIdentifiers[fields[0].substr(3)];

    const formatter = SentencesFormats[sentenceId];
    const sentenceProperties = formatter.split(',').map((format, index) => {
        if (parsers[format]) {
            return parsers[format](fields[index + 1]);
        }

        return fields[index + 1];
    });
    const packet = decoders[sentenceId](sentenceProperties);

    packet.talkerId = talkerId;

    return packet;
};
```

Kod źródłowy 3.5. Obiekt definiujący formaty komend
Źródło: Opracowanie własne

```
const SentencesFormats: Record<SentenceIdentifiers, string> = {
    //obiekt skrócony w celach edytorskich
    [SentenceIdentifiers.DBT]: 'x.x,f=ft|M=m,x.x,f=ft|M=m,x.x,F',
    [SentenceIdentifiers.GGA]:
'hhmmss.ss,llll.ll,a,yyyyy.yy,a,x,xx,x.x,x.x,M,x.x,M,x.x,xxxx',
    [SentenceIdentifiers.MWD]: 'x.x,T,x.x,M,x.x,N,x.x,M',
    [SentenceIdentifiers.VHW]: 'x.x,T,x.x,M,x.x,N,x.x,K',
    }
```

Kod źródłowy 3.6. Obiekt definiujący sposób przetwarzania danych zależnie od ich rodzaju
Źródło: Opracowanie własne

```
const parsers: Object<(x: string) => any> = {
    x: (x) => parseIntSafe(x),
    xx: (x) => parseIntSafe(x),
    xxx: (x) => parseIntSafe(x),
    xxxx: (x) => parseIntSafe(x),
    xxxxx: (x) => parseIntSafe(x),
    xxxxxx: (x) => parseIntSafe(x),
    hh: (x) => parseIntSafe(x, 16),
    hhhh: (x) => parseIntSafe(x, 16),
    hhhhhh: (x) => parseIntSafe(x, 16),
    hhhhhhhh: (x) => parseIntSafe(x, 16),
    'h--h': (x) => hexWithPrefixToBytes(x),
    'x.x': (x) => parseFloatSafe(x),
    'c--c': (x) => x,
    'llll.ll': (x) => ParseCommonDegrees(x),
    'yyyyy.yy': (x) => ParseCommonDegrees(x),
    hhmmss: (x) => moment(x, 'HHmmss').format('HH:mm:ss'),
    'hhmmss.ss': (x) => moment(x, 'HHmmss').format('HH:mm:ss'),
```

```
    ddmmyy: (x) => moment(x, 'DD-MM-YY').format('DD/MM/YYYY'),
    'dd/mm/yy': (x) => moment(x, 'DD/MM/YY').format('DD/MM/YYYY'),
    'dddmm.mmm': (x) => ParseCommonDegrees(x)
};
```

Kod źródłowy 3.7. Kodek DBT

Źródło: Opracowanie własne

```
export interface DBTPacket {
    sentenceId: SentenceIdentifiers.DBT;
    sentenceName?: string;
    talkerId?: string;
    depthFeet: number;
    depthMeters: number;
    depthFathoms: number;
}

export const decodeDBT = (fields: Array<number>): DBTPacket => ({
    sentenceId: SentenceIdentifiers.DBT,
    sentenceName: SentencesDescriptions.DBT,
    depthFeet: fields[0],
    depthMeters: fields[2],
    depthFathoms: fields[4]
});
```

3.6. Zarządzanie danymi

Uzyskane dane oczywiście należy przechowywać w aplikacji. Komponenty tworzące widok zorganizowane są w hierarchicznej strukturze. Błędnym rozwiązaniem byłoby przekazywanie danych w dół hierarchii przez wszystkie komponenty pośredniczące, gdyż dostęp do nich powinny mieć jedynie elementy je wykorzystujące. W takich sytuacjach, gdy dane muszą być dostępne dla wielu komponentów na różnych poziomach zagnieżdżenia, stosuje się globalny stan aplikacji. Dużym problemem dotyczącym zagadnienia globalnie przechowywanych informacji jest możliwość ich mutacji. JavaScript przechowuje złożone struktury danych przez referencję, co umożliwia modyfikację obiektów (kod źródłowy 3.8), które powinny pozostać stałe. Może to prowadzić do ciężkich w lokalizacji błędów. Aby tego uniknąć skorzystano z wzorca stanu. Rozbudowane projekty wykorzystują w tym celu biblioteki takie jak *Redux* i *MobX*, jednak potrzeby omawianej aplikacji nie uzasadniają ich wykorzystania. Wedle [9] *ReactNative* zapewnia możliwość utworzenia kontekstu, dostępnego dla każdego komponentu w nim zagnieżdżonego. Korzystając z tej funkcjonalności utworzone zostały trzy konteksty:

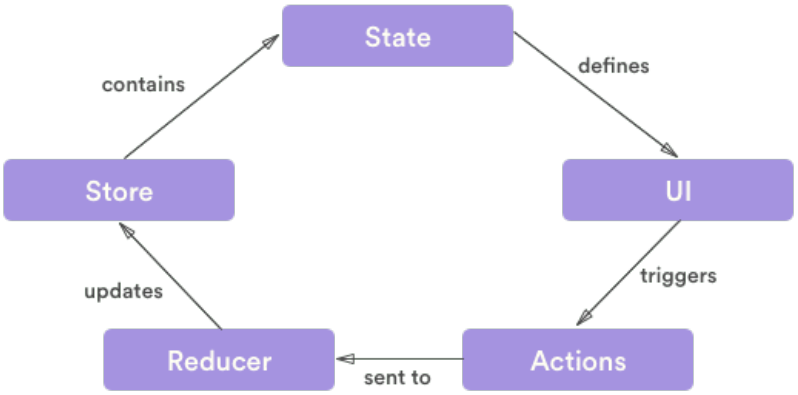
1. nmeaConnectorContext odpowiedzialny za utworzenie połączenia *WebSocket*, odbieranie informacji ze strony serwera oraz przechowywanie ich.

2. unitContext odpowiedzialny za sformatowane dane dotyczące komend *NMEA* opisujących parametry jednostki pływającej.

3. positionContext odpowiedzialny za sformatowane dane dotyczące komend *NMEA* opisujących położenie jednostki pływającej.

Do ich obsługi wykorzystano wzorzec stanu narzucany przez bibliotekę *Redux* [1, 17] (Rysunek 3.4) Podstawowe elementy składniowe tego wzorca to:

1. Action, czyli obiekt składający się z typu oraz ładunku (ang. payload). Każda akcja musi zostać wyemitowana poprzez wywołanie metody dispatch(action).
2. Reducer, czyli funkcja odpowiadająca za zmianę stanu w odpowiedzi na wyemitowanie akcji. Zaleca się, aby funkcja pełniąca rolę reducera była 'pure function'.
3. Store, czyli obiekt przechowujący stan aplikacji. Jego zmiany oraz ich przebieg definiowany jest przez reducery.



Rysunek 3.4. Cykl życia Redux
Źródło: https://dev.to/radiumsharma06/abc-of-redux-5461

Dla każdego z kontekstów oznacza to, że stan może zostać jedynie odczytany. Jakiekolwiek zmiany w stanie wywołuje jedynie reducer w odpowiedzi na określoną akcję. W funkcji reducera koniecznie należy zwrócić całkowicie nowy obiekt stanu po każdej modyfikacji. Dzięki temu możliwe jest wykonanie procedur powiązanych ze zmianą, w szczególności przerenderowanie powiązanych komponentów. Konieczność ta wynika ze wspomnianej natury języka, mutacja obiektu nie zmieni jego referencji. Co za tym idzie, porównanie zmienionego obiektu stanu z poprzednim zwróci wartość 'true' sugerującą brak zmian. Zgodnie z założeniami wzorca stanu dla każdego kontekstu (w tym wypadku pełniącego rolę 'store') utworzono wymagane akcje (kod źródłowy 3.9) oraz funckję reducera (kod źródłowy 3.10).

Kod źródłowy 3.8. Mutacja obiektu
Źródło: Opracowanie własne

```
1  const myObj = {x: 1, y: 2};
2  myObj.y = 4;
3  // myObj is now {x: 1, y:4}
```

Kod źródłowy 3.9. Akcje kontekstu konektora *NMEA*
Źródło: Opracowanie własne

```
export const NmeaConnectorActions = {

    setConnected: (flag: boolean) => createAction(ActionTypes.SET_CONNECTED, flag),
     setData: (data: any) => createAction(ActionTypes.SET_DATA, data),
     setUrl: (url: string) => createAction(ActionTypes.SET_URL, url)
};

export type NmeaConnectorActions = ActionsUnion<typeof NmeaConnectorActions>;
```

Kod źródłowy 3.10. Funkcja reducera dla konektora *NMEA*
Źródło: Opracowanie własne

```
const nmeaReducer = (state: INmeaConnectorState, action: NmeaConnectorActions):
INmeaConnectorState => {
    switch (action.type) {
        case ActionTypes.SET_CONNECTED: {
            return { ...state, connected: action.payload };
        }
        case ActionTypes.SET_DATA: {
            return { ...state, data: action.payload };
        }
        case ActionTypes.SET_URL: {
            return { ...state, url: action.payload };
        }
        default: {
            throw new Error(`Unhandled action type: ${action}`);
        }
    }
};
```

Dzięki takiemu rozwiązaniu warstwa abstrakcji definiująca strukturę globalnych danych oraz przebieg ich zmian oddzielona jest od warstwy widoku. Dane dostarczane przez konteksty są konsumowane przez zależne od nich komponenty. W tym celu utworzono tak zwane ’hooki’ [8]. Jest to wzorzec wprowadzony przez twórców biblioteki *React* oraz frameworka *ReactNative*, określający metody pozwalające na skorzystanie z wewnętrznego stanu oraz cyklu życia komponentu w ciele funkcji. Wspomniane metody dostarczane są przez framework, a ich cechą charakterystyczną jest przedrostek ’use’ występujący na początku każdej funkcji (przykładowo ’useState’, ’useEffect’). Docelowym zamysłem twórców jest wykorzystanie ich do tworzenia własnych ’hooków’ będących funkcjami zawierającymi współdzieloną logikę. Taką logiką w omawianej aplikacji jest użycie kontekstów. Każdy z nich zbudowany jest przy użyciu natywnych ’useContext’ oraz ’useReducer’. Użycie pierwszego z nich przedstawia wiersz dwudziesty czwarty kod źródłowy 3.11, gdzie ’PositionContext’ zdefiniowany jest w wierszu dwunastym 3.11. Wykorzystanie ’useReducer’ przedstawia wiersz piętnasty 3.11. Zwraca on obiekt stanu oraz metodę ’dispatch’ służące do odczytu stanu oraz emisji akcji. Wiersze od dwudziestego trzeciego do dwu-

dziestego dziewiątego 3.11 zawierają definicję własnego hooka.

Kod źródłowy 3.11. Definicja kontekstu oraz hooka pozycji jednostki
Źródło: Opracowanie własne

```
export interface IPositionState {
    GGA: GGAPacket | null;
    GLL: GLLPacket | null;
    GSA: GSAPacket | null;
}

export interface IPositionContext {
    state: IPositionState,
    dispatch: Dispatch<PositionActions>
}

const PositionContext = createContext<IPositionContext | undefined>(undefined);

const PositionProvider = ({ children }) => {
    const [state, dispatch] = React.useReducer(positionReducer, initialState);
    return (
        <PositionContext.Provider value={{ state, dispatch }}>
            {children}
        </PositionContext.Provider>
    );
};

const usePosition = (): IPositionContext => {
    const context = React.useContext(PositionContext);
    if (context === undefined) {
        throw new Error('usePosition must be used within a PositionProvider');
    }
    return context;
};
```

Posiadając w ten sposób zdefiniowany kontekst programista zyskuje prosty i intuicyjny sposób na jego wykorzystanie w dowolnym komponencie niezależnie od jego umiejscowienia w hierarchii. Przykład użycia przedstawia 3.12

Kod źródłowy 3.12. Przykładowy komponent konsumujący hook kontekstu
Źródło: Opracowanie własne

```
// funkcja stworzona w celach edytorskich
function ConsumerComponent() {
    const unit = useUnit();

    // access state
    const MTWPacket = unit.state.MTW;

    // dispatch action
    unit.dispatch(UnitActions.setFrameData({ frameType: 'MTW', frameData: data })

```

```
11      return <View><ChildComponents /></View>
```

Konteksty dotyczące jednostki oraz pozycji zawierają sformatowane komendy (wiersze od pierwszego do piątego 3.11), które użyte zostały do zbudowania GUI. Za przekształcenie otrzymywanego ciągu znaków do struktur pakietów zawartych w kontekstach odpowiada metoda przedstawiona w kodzie źródłowym 3.13). Jest ona wywoływana przez każdą zmianę stanu kontekstu 'nmeaConnectorContext', czyli przy odbiorze danych z serwera. Otrzymany ciąg znaków ulega podziałowi względem znacznika końca linii. Każdy element tablicy wynikowej tego podziału poddawany jest próbie dekodowania. Uzyskane dane przypisane zostają do ładunku akcji dla określonego kontekstu w zależności od identyfikatora komendy. Dzięki temu komponenty reprezentujące widok mogą bazować na spreparowanych danych zawartych w kontekstach 'unitContext' oraz 'positionContext', gdyż są one aktualizowane natychmiastowo po otrzymaniu wiadomości z serwera.

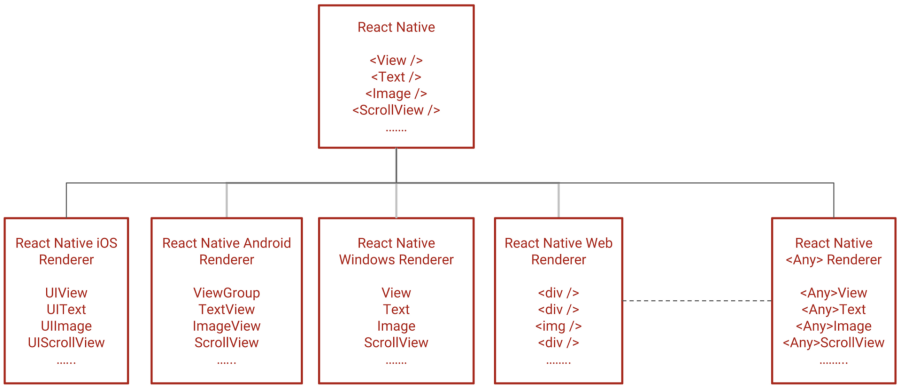
Kod źródłowy 3.13. Dekodowanie i dystrybucja otrzymanych danych
Źródło: Opracowanie własne

```
1       const nmeaConnector = useNmeaConnector();
2       const position = usePosition();
3       const unit = useUnit();
4
5       useDeepCompareEffect(() => {
6           if (nmeaConnector.state.connected &&
    Object.keys(nmeaConnector.state.data).length > 0) {
7               const frames = nmeaConnector.state.data.split(/\r?\n/);
8               frames.map((frame: string) => {
9                   try {
10                      const response = parseNmeaSentence(frame);
11                      if (isPositionPacket(response)) {
12                          position.dispatch(PositionActions.setFrameData({
    frameType: response.sentenceId, frameData: response }))
13                      }
14                      else if(isUnitPacket(response)) {
15                          unit.dispatch(UnitActions.setFrameData({ frameType:
    response.sentenceId, frameData: response }))
16                      }
17                  }
18                  catch(error) {
19                      console.error('Parsing error, invalid input data: ', error)
20                  }
21              })
22          }
23      }, [nmeaConnector.state])
```

3.7. Graphial User Interface (GUI)

Ostatnią składową aplikacji jest warstwa widoku. W przypadku projektów opartych na *ReactNative* składa się ona z hierarchicznej struktury komponentów. Dane przekazywa-

ne są z góry na dół, a każdy komponent może również posiadać własny stan wewnętrzny. Zmiana otrzymanych wartości lub wspomnianego stanu powoduje przerenderowanie komponentu. Dla zapewnienia odpowiedniej wydajności wykorzystywany jest koncept o nazwie *Virtual DOM*. Wedle [8] jest to odzwierciedlenie rzeczywistego *DOM (Document Object Model)* przechowywane w pamięci. Ich synchronizacja określona jest mianem procesu rekonsilacji, podczas którego porównywane są obydwa modele. Elementy, w których zostały wykryte zmiany zostają utworzone ponownie. Do tworzenia elementów ReactNative wykorzystuje silnik renderujący zależny od platformy, na której została uruchomiona aplikacja. Teoretycznie możliwe jest dzięki temu rozszerzenie framework'u o wsparcie dla dowolnej platformy, dla której jesteśmy w stanie zapewnić silnik renderujący [16]. Rysunek 3.5).



Rysunek 3.5. ReactNative render
Źródło: https://medium.com/@agent_hunt/introduction-to-react-native-renderers-aka-react-native-is-the-java-and-react-native-renderers-are-828a0022f433

GUI aplikacji składa się z następujących komponentów:

1. MainView odpowiedzialny za inicjalizację połączenia sieciowego oraz wywołanie procesu dekodowania danych, które następnie przypisywane są do odpowiednich kontekstów.

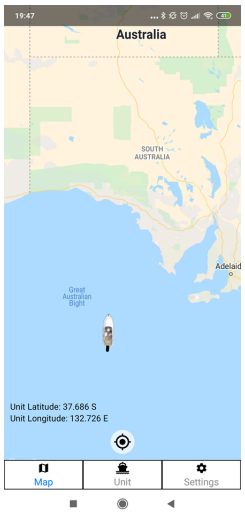
2. Navigator otrzymujący dane przekazane przez komponent MainView. Odpowiada on za rozgałęzienie hierarchii komponentów na poszczególne widoki oraz przechowywanie niezbędnych dla nich danych. W zależności od jego stanu renderowany jest odpowiedni z kolejnych wypunktowanych komponentów.

3. MapView Rysunek 3.6 odpowiedzialny za stworzenie mapy przy wykorzystaniu API firmy Google . Komponent ten nanosi jednostkę na mapę oraz przemieszcza widoczny obszar wraz z ruchem jednostki. Dodatkowo wyznacza na mapie linię obrazującą przebytą drogę. W przypadku interakcji użytkownika w postaci przesunięcia, obrócenia lub przybliżenia/oddalenia mapy funkcjonalność śledzenia jednostki zostaje wyłączona. Dzięki temu zabiegowi doświadczenia użytkownika zbliżone są

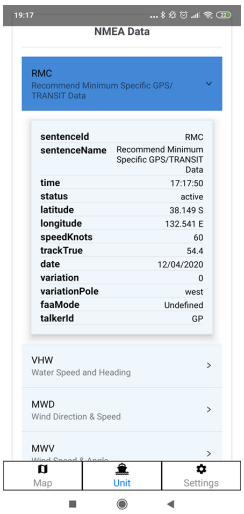
do zapewnianych przez mobilną aplikację Google Maps. Za wycentrowanie ekranu na jednostce oraz przywrócenie funkcji podążania odpowiedzialny jest przycisk umieszczony w dolnej części ekranu. Przy wykorzystaniu kontekstu pozycji w lewej dolnej części ekranu przedstawione są aktualne dane na temat położenia geograficznego. Widok ten wykorzystuje również kontekst jednostki, aby uzyskać informację o kącie skierowania. Na jej podstawie obracana jest ikonka statku reprezentująca obiekt.

4. UnitView Rysunek 3.7 odpowiedzialny za stworzenie rozwijalnych kart reprezentujących poszczególne komendy NMEA. Przeprowadzany jest w nim proces iteracji po wszystkich elementach zawartych w kontekstach jednostki oraz pozycji. Dla każdego z nich tworzy komponent FrameItem, który otrzymuje informacje dotyczące konkretnej linii danych. Przy użyciu wewnętrznego stanu określana jest ilość widocznych informacji. Domyślnie przedstawia on identyfikator komendy oraz jej deskryptor. W odpowiedzi na dotknięcie ekranu przez użytkownika komponent renderuje pozostałe informacje reprezentujące wszystkie pola oraz ich wartości. Ponowna interakcja użytkownika na tym samym elemencie powoduje ukrycie wspomnianych danych.

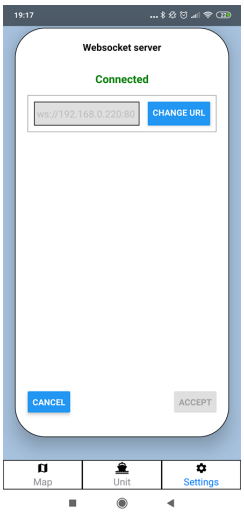
5. SettingsView Rysunek 3.8 odpowiedzialny za zarządzanie ustawieniami aplikacji. Dla omawianej aplikacji jedynym konfigurowalnym ustawieniem jest adres serwera WebSocket. Komponent ten przedstawia aktualny adres oraz stan połączenia. Zmiana adresu skutkuje próbą utworzenia połączenia. Udane nawiązanie powoduje zmianę tekstowej reprezentacji połączenia (zielony napis ”CONNECTED” lub czerwony ”DISCONNECTED”) dzięki czemu użytkownik nie jest zmuszony do zmiany widoku w celu weryfikacji.



Rysunek 3.6. MapView
Źródło: Opracowanie własne



Rysunek 3.7. UnitView
Źródło: Opracowanie własne



Rysunek 3.8. SettingsView
Źródło: Opracowanie własne

Taki podział komponentów pozwala na intuicyjne odczytywanie danych dotyczących położenia, dzięki wykorzystaniu map Google, oraz dokładnych odczytów poszczególnych składowych komend *NMEA*. Każdy nowy format dodany do aplikacji poprzez stworzenie odpowiedniego pliku kodeka automatycznie zostanie odzwierciedlony w UnitView o ile dane otrzymywane przez aplikację taką komendę zawierają. Dzięki temu interfejs użytkownika jest łatwo rozwijalny.

Zakończenie

Udało się stworzyć aplikację mobilną zgodną z wymaganiami pracy, pod niektórymi względami przewyższającą część dostępnych na rynku rozwiązań. Struktura kodu źródłowego pozwala na oddzielenie warstwy dekodowania danych od warstwy widoku. Dzięki temu możliwym byłoby dostarczenie biblioteki zajmującej się przetwarzaniem danych NMEA, możliwej do wykorzystania przez inne aplikacje. Najtrudniejszym zagadnieniem okazało się stworzenie jak najbardziej generycznej obsługi komend. Dodatkowe problemy sprawił również utrudniony dostęp do dokumentacji, większość dostępnej wiedzy stanowią źródła w postaci stron internetowych. Interfejs użytkownika można wzbogacić o wiele informacji oraz funkcjonalności, dla przykładu - dane historyczne, możliwość ustalania własnych znaczników wraz z opisami, system alarmowy. Oprócz tego warto byłoby zapewnić możliwość pracy aplikacji w trybie offline. Dla warstwy dekodującej wsparcie dla komend protokołu NMEA 2000 oraz AIS byłoby niewątpliwym atutem.

Bibliografia

[1] Abramov D.: *Dokumentacja redux*, 2020, URL: https://redux.js.org/.

[2] Association N. M. E.: *Stanard protokołu nmea 0183*, 2000, URL: https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.

[3] Association N. M. E.: *Stanard protokołu nmea 2000*, 2000, URL: https://www.nmea.org/content/STANDARDS/NMEA_2000.

[4] Babel: *Dokumentacja babel*, 2020, URL: https://babeljs.io/.

[5] Betke K.: *The nmea 0183 protocol*, maj 2000, URL: https://www.tronico.fi/OH6NT/docs/NMEA0183.pdf.

[6] Boni A.: *Aplikacja navmonitor*, wrz. 2019, URL: https://play.google.com/store/apps/details?id=com.aboni.boatinga/.

[7] Expo: *Dokumentacja expo*, 2020, URL: https://docs.expo.io/versions/v37.0.0/.

[8] Facebook: *Dokumentacja reactjs*, 2020, URL: https://reactjs.org/.

[9] Facebook: *Dokumentacja reactnative*, 2020, URL: https://reactnative.dev/.

[10] Ho P.: *Aplikacja nmea tools*, mar. 2020, URL: https://play.google.com/store/apps/details?id=com.peterhohsy.nmeatools.

[11] Krotoff T.: *Front-end frameworks popularity (react, vue and angular)*, grud. 2019, URL: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190.

[12] Ltd M. M. P.: *Aplikacja iregatta*, mar. 2018, URL: https://play.google.com/store/apps/details?id=dk.letscreate.aRegatta.

[13] Marine S.: *Aplikacja seawi marine*, kw. 2019, URL: https://play.google.com/store/apps/details?id=net.seawimarine.activities.

[14] Raymond E. S.: *Strona internetowa*, mar. 2019, URL: https://gpsd.gitlab.io/gpsd/NMEA.html.

[15] Seapilot: *Aplikacja seapilot*, czer. 2019, URL: https://play.google.com/store/apps/details?id=com.seapilot.android.

[16] Shailesh: *Introduction to react native renderers aka react native is the java and react native renderers are the jvms of declarative ui*, czer. 2018, URL: https://medium.com/@agent_hunt/introduction-to-react-native-renderers-aka-react-native-is-the-java-and-react-native-renderers-are-828a0022f433.

[17] Sharma R.: *Abc of redux*, mar. 2018, URL: https://dev.to/radiumsharma06/abc-of-redux-5461.

[18] *Strona internetowa*, 2020, URL: https://www.gpsinformation.org/dale/nmea.htm.

[19] YACHT D.: *Opis produktu wln10 smart firmy digital yacht*, 2020, URL: https://digitalyacht.co.uk/product/wln10sm/.

[20] zapfware: *Aplikacja nmearemote*, wrz. 2016, URL: https://apps.apple.com/us/app/nmearemote/id412806204.

Spis tabel

Spis rysunków

Spis kodów źródłowych