



Uniwersytet im. Adama Mickiewicza w Poznaniu

Wydział Matematyki i Informatyki

kierunek: Informatyka

Praca inżynierska

Rozwój open-source'owego frameworka do tworzenia aplikacji - "Sealious" (cz. 3)

Extending capabilities of Sealious - an open-source
application-development framework (part 3)

Adrian Wydmański

Numer albumu: 384082

Promotor:
prof. Marek Nawrocki

Poznań, 2016

Poznań, dnia

Oświadczenie

Ja, niżej podpisany **Adrian Wydmański**, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt:

Rozwój open-source'owego frameworka do tworzenia aplikacji - "Sealious" (cz. 3)

napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[]* - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[]* - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

.....

Spis treści

| | |
|---|-----------|
| Wstęp..... | 7 |
| I Rozwiązania ułatwiające szybkie rozwijanie aplikacji | 9 |
| 1 Problemy związane z rozwijaniem projektów programistycznych | 13 |
| 2 Organizacja pracy – spotkania, komunikacja, narzędzia | 17 |
| 3 System kontroli wersji..... | 19 |
| 4 Zarządzanie zadaniami | 23 |
| 5 Wersjonowanie i npm..... | 25 |
| 6 Testowanie oprogramowania i ciągła integracja..... | 29 |
| Podsumowanie części I..... | 35 |
| II Rozwiązanie wspomagające pracę docelowych programistów | 39 |
| 1 Deklaratywność | 43 |
| 2 Dokumentacja..... | 47 |
| 3 Komunikaty błędów | 51 |
| Podsumowanie części II..... | 55 |
| III Załączniki | 57 |
| IV Bibliografia | 73 |

Wstęp

Sealious jest to open-source'owy, wysoce deklaracyjny framework umożliwiający tworzenie aplikacji webowych i desktopowych. Naszym celem jest stworzyć użyteczne narzędzie, które pozwala docelowemu programiście skupić się na tym **jaki** ma być końcowy efekt, a nie **jak** go osiągnąć.

Pragniemy, aby Sealious był tworzony nie tylko „przez programistów, dla programistów”, ale przede wszystkim przez ludzi, dla ludzi. Jest on wyrazem naszego dążenia do zwiększenia wolności wyboru dla konsumentów oraz obniżania progu wejścia dla aspirujących programistów. Chcemy także, aby stał Sealious się przydatnym narzędziem również dla nas, jako zespołu, do przyszłego tworzenia oprogramowania.

*„Cóż bowiem za korzyść odniesie człowiek,
choćby cały świat zyskał,
a na swej duszy szkodę poniósł?”[1]*

Część I

Rozwiązania ułatwiające szybkie rozwijanie aplikacji

Praca nad projektem inżynierskim uzmysłowiła naszemu zespołowi jakie problemy towarzyszą procesowi tworzenia nowych rozwiązań programistycznych oraz jakich narzędzi oraz usług używać, aby usprawnić pracę nad kodem. Zdobyte doświadczenie pozwoliło nam wypracować dobrze działający *workflow*, który wspomaga naszą pracę i zwiększa naszą produktywność. Poniższa część zawiera opis następujących punktów:

1. **Problemy związane z rozwijaniem projektów informatycznych** – opis trzech problemów związanych z rozwijaniem projektów informatycznych, które moim zdaniem przeszkadzają w produktywnej pracy nad projektem informatycznym.
2. **Organizacja pracy** – dzięki konsekwencji w spotkaniach oraz przyjaznym otoczeniu, jesteśmy w stanie pomagać sobie nawzajem i dodawać sobie motywacji.
3. **System kontroli wersji** – serwis przeznaczony do *hostowania* projektów informatycznych wykorzystujący system kontroli wersji Git. Na nim przechowywane jest repozytorium naszego kodu, na nim również pojawiają się zgłoszenia o błędach.
4. **Zarządzanie zadaniami** – opis narzędzia do produktywności **Trello**, które pomaga zorganizować propozycje zmian w projekcie.
5. **Wersjonowanie i npm** - domyślny menadżer modułów dla środowiska Node.js, na którym opublikowany jest Sealious, narzuca również wersjonowanie projektu.
6. **Testy oprogramowanie i ciągła integracja** – dobrze napisane testy kodu źródłowego pomagają walidować istniejący kod oraz zmiany, które mają zostać dodane. Usługa ciągłej integracji pozwala zautomatyzować ten proces.

Rozdział 1

Problemy związane z rozwijaniem projektów programistycznych

W trakcie tworzenia projektu inżynierskiego wystąpiło kilka problemów, które utrudniały pracę nad nim. Wbrew pozorom, największe kłopoty nie były związane z kwestiami technicznymi, takimi jak brak systemu kontroli wersji, czy nie napisanie testów jednostkowych, dzięki którym można sprawdzić, czy zmiany w kodzie nie destabilizują cały projekt. Oczywiście te problemy powinny zostać rozwiązane jak najszybciej, jednak są one stosunkowo łatwe do rozwiązania.

O wiele poważniejsze problemy to te, które nazwałbym problemami „miękkimi”, tj. problemami z organizacją pracy i jej przebiegiem.

W tym rozdziale zajmę się trzema problemami, które powinny być jak najszybciej rozwiązane:

1. **Brak zaangażowania członków grupy** – bez zaangażowania projekt nie posiada taką ilość uwagi jaką powinien, przez to efekty pracy są niewystarczające.
2. **Brak osoby, która zarządza projektem** – osoba, która posiada zdolności przywódcze i koordynacyjne bardzo często pomaga zmobilizować zespół, dzięki temu projekt przebiega o wiele sprawniej.
3. **Brak umiejętnego zarządzania zadaniami** – niepraktyczne rozplanowanie kolejnych etapów tworzenia oprogramowania znacznie utrudnia rozwój projektu.

1. Brak zaangażowania członków grupy

Brak osób chętnych do wykonania jakiegokolwiek projektu jest największym problemem, którego wbrew pozorom nie jest tak łatwo rozwiązać. Bardzo często osoby, które są przymuszane do zrobienia danej rzeczy, robią to niestarannie i źle. Powoduje to irytację wśród osób, które są zaangażowane w pracę nad projektem, przez co spada motywacja grupy jako całości. Zazwyczaj w takich sytuacjach tworzą się podgrupy w grupie projektowej, co już samo w sobie jest niepożądane w zespole: ci, którym zależy i ci, którym nie zależy. Zazwyczaj ci pierwsi są w mniejszości, wykonują większość pracy i mają złe zdanie o tych drugich.

Nie ma jednej recepty na brak zaangażowania. Najłatwiejszym i najbardziej oczywistym rozwiązaniem jest podziękowanie za współpracę, jednak nie zawsze jest to możliwe. Poza tym nie zawsze jest to dobre rozwiązanie problemu – eliminuje się osoby, które zawadzają, jednak niektórzy mają problem z jasnym wyrażeniem swojego oburzenia i bardzo często takie osoby są traktowane jako „kula u nogi” i są trzymane w grupie do końca trwania projektu.

Inną możliwością jest prywatna rozmowa przeprowadzona przez lidera, bądź osobę odważną na tyle, by z osobą niezaangażowaną porozmawiać. Istnieje wtedy możliwość poprawy zachowania, wpływa też pozytywnie na integrację zespołu (wywoływanie poczucia odpowiedzialności za innych).

Zazwyczaj ta druga opcja powinna być wykonana najpierw, a dopiero potem można przejść do radykalnych środków.

2. Brak osoby, która zarządza projektem

Ten problem pojawia się najczęściej wtedy, gdy nie ma w zespole dwóch elementów:

1. Osoby, która posiada posłuch wśród członków zespołu i swoją charyzmą potrafi przekonać innych do swojej racji.
2. Temat jest narzucony z góry.

Kiedy w grupie nie ma osoby, która staje się liderem – osobą, która postanawia zarządzać projektem, praca staje się uciążliwa i trudna do wykonania, ponieważ nie ma autorytetu, który byłby siłą napędową grupy. Demokracja jest przydatna wtedy, kiedy odbywa się „burza mózgów”

i proponowane są rozwiązania danego problemu, natomiast samo prowadzenie projektu powinno spocząć na barkach osoby, która posiada posłuch wśród członków grupy. Wtedy praca staje się efektywna i łatwiejsza.

Osoba, która bierze na siebie odpowiedzialność za projekt i czuje pewne zobowiązanie do jego wykonania, jeżeli również posiada charyzmę i posłuch wśród członków zespołu, potrafi stać się kimś, kto zapewni, że praca nad projektem będzie trwała. Taka osoba potrafi zmotywować innych do działania, nawet jeżeli brakuje im umiejętności potrzebnych do wykonania danego im zadania.

Zazwyczaj osoba, która wychodzi z inicjatywą projektu jest tą osobą, która nią zarządza, tj. wie jak projekt ma wyglądać, wie jaki ma być końcowy wynik i stara się forsować kolejne zadania tak, aby końcowy efekt, był jak najbliżej tego, co było zakładane na początku. Więcej o liderze w grupie projektowej znajduje się w rozdziale drugim.

3. Brak umiejętnego zarządzania zadaniami

Niestety zaangażowanie członków grupy jak i posiadanie lidera, który potrafi swoją osobowością trzymać projekt w ryzach nie wystarcza, aby praca nad projektem była łatwa i przyjemna. Największym problemem jaki dotyka niemal wszystkie grupy projektowe jest brak odpowiedniego planu wykonania projektu. Umiejętne zarządzanie zadaniami jest tak samo ważne dla projektu, jak ich wykonanie.

Parafrazując Henry'ego Forda „Nic nie jest szczególnie trudne do wykonania, jeżeli podzieli się pracę na małe zadania” [2] — te słowa sprawdziły się w czasie produkcji pierwszych samochodów, sprawdzają się też podczas zarządzania projektami. Rozłożenie projektu na poszczególne kroki, czy kamienie milowe, sprawia, że łatwiej skupić się na szczegółach, które tworzą całość. Wykonanie pomniejszych zadań podnosi również morale, gdyż pojawia się możliwość częstszego celebrowania sukcesów.

Rozdział 2

Organizacja pracy – spotkania, komunikacja, narzędzia

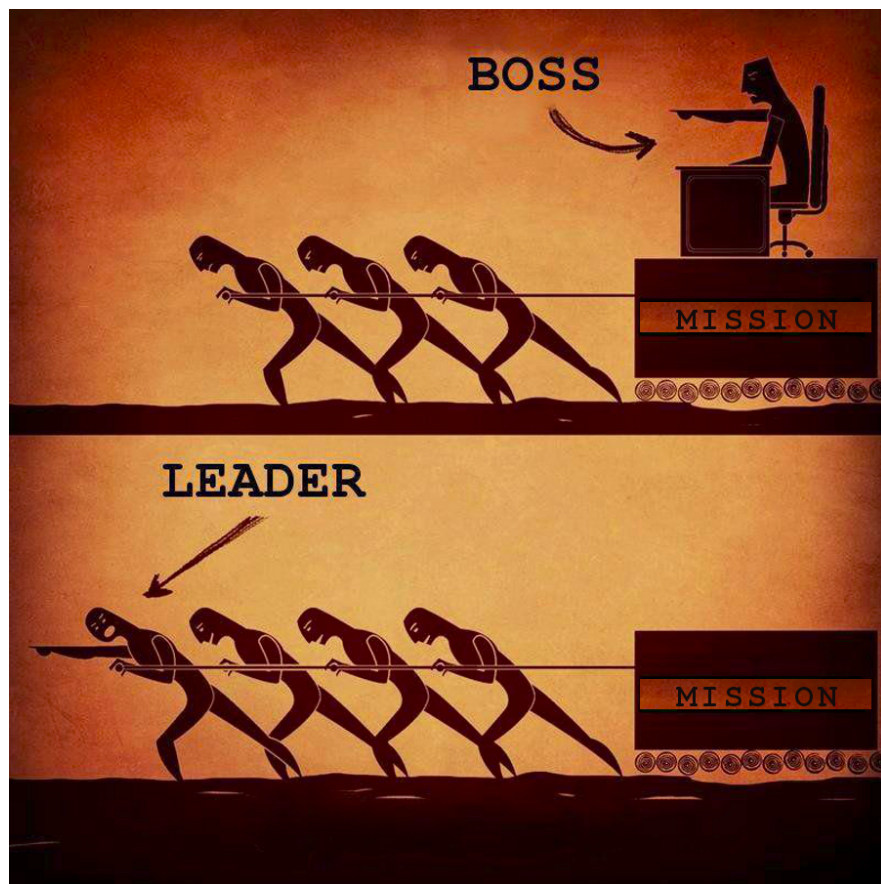
Zorganizowana grupa, to skuteczna grupa. Z moich obserwacji oraz z efektów pracy naszego zespołu wynika, że jedną z najważniejszych rzeczy o jakiej należy pomyśleć zaraz po zrzeszeniu ludzi o podobnych poglądach i zaangażowaniu, jest dobre przemyślenie jak zorganizować grupę tak, aby praca była jak najefektywniejsza, ale i satysfakcjonująca. Nie należy też zapominać o integracji, która pomaga członkom grupy nawiązywać bliższe relacje, co ułatwia komunikację i motywuje do dalszej pracy.

Nic nie jest bardziej zniechęcające niż nieprzyjemna atmosfera w miejscu pracy. Naciskanie na efekty pracy, brak wyrozumiałości, egoizm i zarozumiałość – to wszystko niszczy dobre relacje między członkami grupy i ma negatywny wpływ na efekty pracy. Dlatego podstawą jest zapewnienie luźnych, swobodnych relacji wewnątrz grupy.

W czasie formowania się grupy, w sposób naturalny wyłania się jej lider. Zazwyczaj jest to pomysłodawca, który jest na tyle pewny swojego pomysłu, że angażuje innych ludzi do projektu. Problem jaki może w tym miejscu wystąpić jest bycie „szefem”, a nie jego liderem. Różnicę między szefem a liderem pokazuje Rysunek 2.1 [3]

Innym potencjalnym problemem jest niedostateczne zaangażowanie innych członków, bądź nieumiejętne przekazanie idei projektu reszcie. Wtedy, jeżeli z jakiegoś powodu, lider grupy nie może uczestniczyć w spotkaniu, reszta grupy ma problem z kontynuowaniem projektu.

W przypadku naszego projektu, sposób zorganizowania pracy zmieniał się w czasie. W drugim semestrze projektu inżynierskiego spotkania odbywały się we wtorki i trwały średnio pięć godzin. W tym czasie dyskutowaliśmy co nam się udało zrobić w ciągu tygodnia, robiliśmy *code-review* (o którym więcej w rozdziale trzecim), ustanawialiśmy cele, rozwiązywaliśmy



Rysunek 2.1: Boss vs Leader

problemy kolegów. Nie zabrakło też elementów integracji – luźnych rozmów i dyskusji na tematy niezwiązane z projektem, żartów i oglądaniu krótkich filmów w internecie.

Rozdział 3

System kontroli wersji

Jednym z najważniejszych elementów ułatwiających pracę programistyczną jest wdrożenie systemu kontroli wersji, czyli „oprogramowania służącego do śledzenia zmian głównie w kodzie źródłowym oraz pomocy programistom w łączeniu zmian dokonanych w plikach przez wiele osób w różnych momentach czasowych”. [4] Obecnie praca bez takiego systemu jest znacznym problemem, ponieważ bez niego o wiele trudniej synchronizować kod pomiędzy programistami działającymi w danym projekcie

Jedną z najpopularniejszych systemów kontroli wersji jest Git napisany przez Linusa Torvaldsa, który został wydany w 2005. Jego prosta i łatwość w przyswojeniu przyczyniła się do ogólnoświatowej popularności.

Praca nad naszym projektem inżynierskim była w znacznym stopniu przyspieszona dzięki użyciu systemu **Git**. Udało nam się również przetestować jaki rodzaj *workflow* jest dla nas bardziej efektywny.

Początkowo, od rozpoczęcia projektu do sierpnia 2015, stosowaliśmy następującą konwencję:

- Dwa główne *branche*:
 - *master* — stabilna wersja kodu, dostępna poprzez ekosystem **npm**.
 - *dev* — nowe funkcjonalności oraz zmiany, które mają w przyszłości stać się stabilną wersją poprzez operację *git merge* do „master”, dostępna poprzez ekosystem **npm**.
- *Branche* zawierające nowe funkcjonalności, bądź zmiany w kodzie nazywane były w następujący sposób: *issue#xxx_description*, gdzie *xxx* to numer *issue* na **GitHubie**¹, a *description* to krótki opis dokonywanej zmiany. Po ukończeniu zadania tworzony był *pull request*, który był rozpatrywany przez inną niż autor osobę z zespołu. Jeżeli zmiana została zatwierdzona, dokonywana była operacja *git merge* do *brancha*, w którym miała

¹<https://github.com/> - najpopularniejsza platforma przechowująca repozytoria Git.

być przeprowadzona zmiana.

Nazewnictwo mniejszych *branchy* zostały wymyślone przez nas, aby ułatwić rozwiązywanie zadań. Początkowo wszystkie propozycje zmian oraz zgłaszanie błędów było dodawane poprzez *issue* na platformie **GitHub**.

Pomiędzy czerwcem, a sierpniem 2015, pojawiły się w zespole propozycje zmiany systemu pracy. Padła wtedy propozycja, aby *branch* „master” zmienić nazwę na „latest”, a *branch* „dev” na „next”. Choć nowe nazewnictwo powodowało początkowo trudności w zrozumieniu i zapamiętaniu, postanowiliśmy je przyjąć, aby przetestować jego użyteczność.

Dodatkowo skorzystaliśmy z narzędzia o nazwie **Trello**, o którym mowa w rozdziale 4, na które przenieśliśmy wszystkie propozycje zmian tak, aby sekcja *issue* na **GitHubie** zawierała tylko zgłoszenia błędów. W tej formie konwencja wyglądała następująco:

- Dwa główne *branche*:
 - *latest* — stabilna wersja kodu, dostępna poprzez ekosystem **npm** (więcej o dostępności poprzez ekosystem **npm** w rozdziale 5.),
 - *next* — nowe funkcjonalności oraz zmiany, które mają w przyszłości stać się stabilną wersją, dostępna poprzez ekosystem **npm**.
- Branche zawierające nowe funkcjonalności zaproponowane na *Trello* posiadają nazwę *trello#description*, a te które naprawiają błędy zgłoszone w *issue* mają nazwę *issue#xxx_description*. Po ukończeniu zadania tworzony był *pull request*, który był rozpatrywany przez inną niż autor osobę z zespołu. Jeżeli zmiana została zatwierdzona, dokonywana była operacja *git merge* do *brancha*, w którym miała być przeprowadzona zmiana.

Na przełomie listopada i grudnia 2015 roku, na platformie **Trello** przeprowadzona została dyskusja o zmianie nazewnictwa głównych *branchy*, która ostatecznie przerodziła się w zatwierdzenie nowego *workflow*, które ma wejść w życie w styczniu 2016 roku, tym razem opierającego się na **trzech** głównym *branchach*:

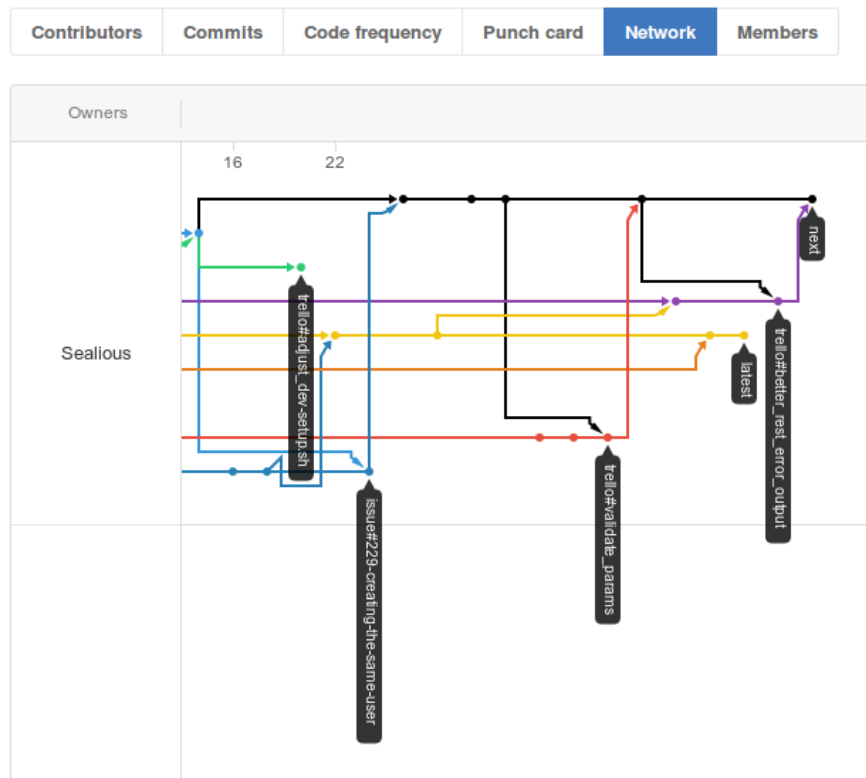
- - *stable* — stabilna wersja kodu (poprzednio „latest”), dostępna poprzez ekosystem **npm**,
 - *beta* — wersja zawierająca wszystkie zaplanowane zmiany w danej wersji, udostępniona użytkownikom do testów (poprzednio brak), dostępna poprzez ekosystem **npm**,
 - *alpha* — każda nowa funkcjonalność, bądź poprawa istniejących funkcji, dostępna jedynie poprzez **GitHub**.

- Nazwy pozostałych *branchy* są niezmiennione.

Sama praca z systemem **Git** wygląda następująco:

- Tworzony jest nowy branch, na którym przeprowadzone są zmiany, ich konwencja nazw podana jest wyżej.
- Po dokonaniu zmian, autor tworzy *pull request*, który jest rozpatrzony przez członka zespołu niebędącym współautorem zmian.
- Jeżeli kod nie zawiera błędów, dokonywana jest operacja *git merge* do *brancha*, w którym ma zajść zmiana.

Poniższa grafika przedstawia fragment grafu *commitów* na platformie **GitHub**. Widać na nim główne *branchy* - „next” i „latest”, oraz inne, pomniejsze, które zawierają zmiany w kodzie.



Rysunek 3.1: Drzewo commitów na platformie GitHub

Rozdział 4

Zarządzanie zadaniami

Rozplanowane i dobrze przemyślane pomysły powinny zostać umieszczone w miejscu, do którego każdy członek zespołu ma dostęp. Czasem zanotowanie ich na tablicy i przepisanie na kartkę papieru, bądź napisanie maila do wszystkich zainteresowanych osób nie wystarczy. Na szczęście istnieje wiele dostępnych narzędzi do produktywności i jedno z nich zastosowaliśmy w naszym projekcie – **Trello**¹. Daje ono możliwość stworzenia list do przechowywania kart, na których można zapisać zadania, pomysły, propozycje zmian, dodać do nich ankietę, przypisywać członków grupy, czy też ustawić *deadline*. Strona ta umożliwia również dodawanie komentarzy do kart, dzięki czemu każda kwestia może być przedyskutowana przez każdego z członków zespołu.

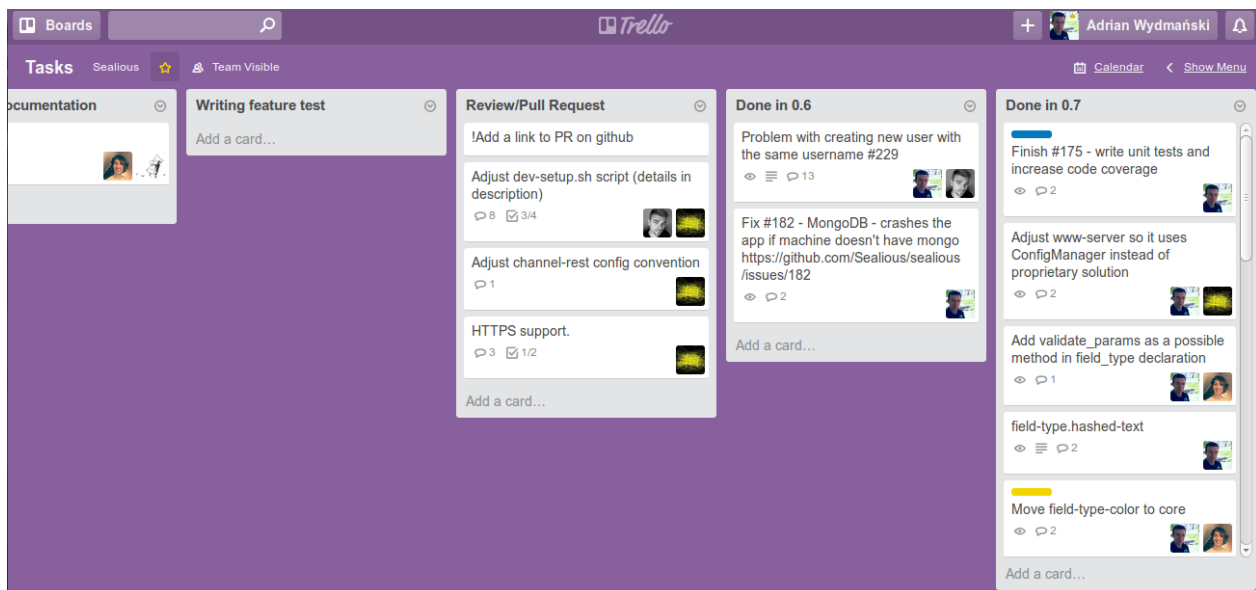
Poprzednio wszystkie propozycje zmian znajdowały się w naszym repozytorium na platformie **GitHub** jako *issue*. Zrezygnowaliśmy z takiego podejścia, ponieważ uznaliśmy, że zwykły użytkownik nie będący w zespole projektowym nie musi być na bieżąco ze wszystkimi zmianami oraz wewnętrznymi dyskusjami o projekcie. Dlatego postanowiliśmy skorzystać z **Trello** do lepszego zarządzania zadaniami.

Obecnie (grudzień 2015) posługujemy się kilkunastoma listami, ich układ jest eksperymentalny i ciągle podlega zmianom. Oto najciekawsze z nich:

- **To do in the future** — zmiany, które chcemy wprowadzić w Sealiousie w przyszłości.
- **To do in next version** — zmiany, które chcemy wprowadzić w Sealiousie w następnej wersji.
- **Hotfixes in the stable version** — zmiany, które mają być wprowadzone w obecnej wersji projektu.

¹<https://trello.com/>

- **Soft tasks** — propozycje zmian niezwiązanych z programowaniem (np. zmiana nazewnictwa *branchy*).
- **Ongoing** — zadania, które w danej chwili są wykonywane.
- **Review/Pull Request** — zadania, które zostały wykonane i są do rozpatrzenia (*code review*).
- **Done in 0.6** — zadania wykonane w wersji 0.6.
- **Done in 0.7** — zadania wykonane w wersji 0.7.
- **Otherwise closed** — wykonane zadania, które nie są bezpośrednio związane z programowaniem.



Rysunek 4.1: Trello

Rozdział 5

Wersjonowanie i npm

npm¹ jest to domyślny menadżer pakietów, zwanych również modułami, dla środowiska Node.js, które automatycznie go instaluje od wersji 0.6.3. Menadżer ten jest dostępny z poziomu linii komend i używany jest do zarządzania zależnościami w aplikacji. Aby zainstalować dany moduł wystarczy wpisać *\$ npm install moduleName*.

Ponieważ nasz projekt jest pisany w języku JavaScript z użyciem Node.js, postanowiliśmy opublikować go na platformie **npm**, która wymaga zastosowania następującego wersjonowania [5]:

Zakładając, że projekt ma wersję 1.0.0:

- Naprawa błędów i inne pomniejsze zmiany: „Patch release” — zwiększenie ostatniej cyfry o jeden: 1.0.1.
- Nowe zmiany, które nie kolidują z istniejącymi funkcjonalnościami: „Minor release” — zwiększenie środkowej cyfry o jeden: 1.1.0.
- Zmiany, które kolidują z istniejącymi funkcjonalnościami (np. brak kompatybilności wstecznej): „Major release” — zwiększenie pierwszej cyfry o jeden: 2.0.0.

Wersje projektu zmieniane są w pliku *package.json*, który zawiera podstawowe informacje wykorzystywane przez **npm**. Plik ten można najbezpieczniej utworzyć wpisując komendę *\$ npm init* w katalogu, w którym chcemy tworzyć projekt.

W naszym projekcie istniało kilka sposobów wersjonowania związanych z *workflow* wymienionych w rozdziale 3:

1. Workflow z głównymi branchami „master” i „dev”:

¹<https://www.npmjs.com/>

- Zmiany w branchu „master” trafiały na platformę **npm** ze zwiększoną środkową cyfrą po operacji *git merge*.
- Zmiany w branchu „dev” nie trafiały na platformę **npm**.

Przykład: Stabilna wersja projektu: 0.7.0. Branch „dev” zawiera nową funkcjonalność, której nie zawiera branch „master”. Po dokonaniu operacji *git merge* z „dev” do „master”, zwiększona zostaje środkowa cyfra – 0.8.0, zmiany są publikowane.

2. Workflow z głównymi branchami „latest” i „next”:

- Branch „latest” zawierał ostatnią wersję z brancha „next” przed operacją *git merge* z „next” do „latest”. Po dodaniu ewentualnej łatki, zwiększona zostaje ostatnia cyfra i zmiany są publikowane na platformie **npm**.
- Zmiany w branchu „next” trafiały na platformę **npm** ze zwiększoną ostatnią cyfrą po każdej zmianie.

Przykład 1: Praca nad branchem „next”, wersja projektu 0.7.8. Po dodaniu nowej funkcjonalności zwiększona zostaje ostatnia cyfra – 0.7.9, projekt jest publikowany.

Przykład 2: Praca nad branchem „next”, wersja projektu 0.7.15. Po dodaniu wszystkich funkcjonalności w danej wersji, zostaje dokonana operacja *git merge* z „next” do „latest”. Po tej operacji, wersja projektu na branchu „latest” to 0.7.15, natomiast na branchu „next” rozpoczynają się prace nad wersją 0.8.0.

3. Workflow z głównymi branchami „stable”, „beta”, „alpha”.

- Branch „stable” w założeniu ma być ustabilizowany i nie zawierać zmian. W razie pojawienia się łatki, zwiększona zostanie ostatnia cyfra i zmiana zostanie opublikowana na platformie **npm**.
- Zmiany w branchu „beta” trafiają na platformę **npm** ze zwiększoną środkową cyfrą po każdej zmianie.
- Zmiany w branchu „alpha” nie trafiają na platformę **npm** (dostęp tylko poprzez GitHub).

Przykład 1: W brancha „alpha” dodana jest nowa funkcjonalność. Wersja projektu się nie zmienia.

Przykład 2: Wersja na branchu „beta”: 0.8.2. Po dodaniu wszystkich funkcjonalności do brancha „alpha”, zostaje dokonana operacja *git merge* z „alpha” do „beta”. Wersja projektu na branchu „beta” zmienia się do 0.9.0.

Przykład 3: Wersja projektu na branchu „beta”: 0.9.0. Zostaje zgłoszony błąd,

który zostaje naprawiony. Wersja projektu zmienia się do 0.9.1, zmiana zostaje opublikowany.

Przykład 4: Wersja projektu na branchu „stable”: 0.8.2, na branchu beta: 0.9.1. Po uznaniu, że wersja na branchu „beta” jest stabilna, dokonana zostaje operacja *git merge* z „beta” do „stable”. Wersja projektu na branchu „stable”: 0.9.1.

Ostatni sposób wersjonowania jest ze wszystkich najbardziej optymalny, bowiem użytkownik dostaje dwa gotowe produkty: jeden stabilny, drugi zawierający najnowsze funkcjonalności, które przeszły testy wewnętrzne (choć mogące zawierać błędy, których nie udało się wychwycić wcześniej).

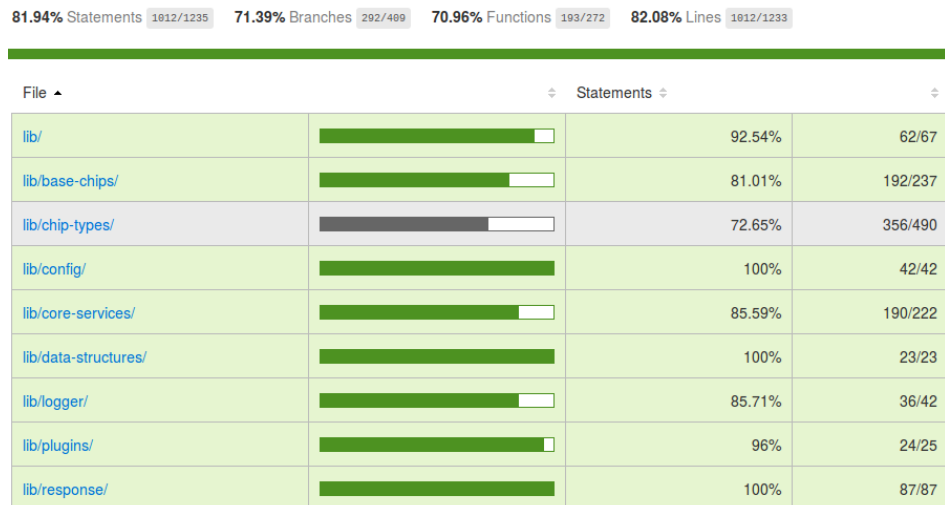
Rozdział 6

Testowanie oprogramowania i ciągła integracja

Testowanie kodu jest przydatnym narzędziem pozwalającym zweryfikować poprawność napisanego kodu. Ustalenie jak kod powinien wyglądać, jak ma się zachować, jakie wyniki ma zwracać i zaimplementowanie tego w postaci testów pokrycia pomaga na bieżąco walidować nowe zmiany w projekcie oraz zapewniają, że testowany program działa tak jak powinien. Napisanie testów pokrycia pomaga również wykryć istniejące błędy jeszcze w fazie tworzenia oprogramowania.

Testy powinny być pisane zawsze, do każdego projektu. Ręczne sprawdzenie, czy kod działa poprawnie jest zajęciem żmudnym i podatnym na błędy, łatwo bowiem pominąć, bądź zapomnieć o przypadku użycia. Automatyzacja jest w tym przypadku jak najbardziej pożądana.

W naszym projekcie inżynierskim testy jednostkowe są kluczowym elementem sprawdzania poprawności zmian. Jeżeli proponowana zmiana nie przechodzi poprawnie wszystkich testów, tj. wynik końcowy jest inny od oczekiwanego, oznacza to, że zmiana ta jest niezgodna z obowiązującymi w danej wersji funkcjonalnościami i powinna być poprawiona tak, aby testowany kod wykonywał się poprawnie. Obecnie w naszym projekcie znajduje się 155 testów, które pokrywają 82% kodu źródłowego, obejmujące przede wszystkim funkcjonalności odpowiedzialne za obsługę zasobów:



Rysunek 6.1: Pokrycie kodu testami jednostkowymi

Testy jednostkowe pisane są za pomocą *frameworka* **MochaJS**¹, który znacząco ułatwia sam proces tworzenia testów:

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      [1,2,3].indexOf(5).should.equal(-1);
      [1,2,3].indexOf(0).should.equal(-1);
    });
  });
});
```

Powyższy przykład sprawdza, czy tablica [1,2,3] zawiera elementy 5 i 0. Każdy test powinien być przyporządkowany do odpowiedniej kategorii tak, aby użytkownik przeprowadzający test bez problemu wiedział jaki obszar kodu w dalej chwili podlega testom. Służy do tego funkcja `describe()`, która jako pierwszy argument posiada nazwę kategorii, do której zostanie przypisany test. Drugim argumentem jest funkcja `callback`, w której mogą znajdować się wielokrotne wywołania funkcji `describe()`.

Funkcją odpowiedzialną za samo przeprowadzenie testu jest `it()`, która jako pierwszy argument powinna przyjmować dokładny opis przeprowadzanego testu, zaś drugim argumentem jest funkcja `callback`, w której dokonuje się testu. Może ona również zawierać argument, który określa koniec testu.

¹<https://mochajs.org/>

Przykłady testów jednostkowych w Sealiousie

`FieldType.Email` jest to deklaracja typu pola zasobu “email”, który akceptuje tylko ciągi znaków mające strukturę adresu *email*:

```
var Sealious = require("sealious");
var Promise = require("bluebird");

var field_type_email = new Sealious.ChipTypes.FieldType({
  name: "email",
  get_description: function(context, params){
    return "Email address, like something@something.sth"
  },
  is_proper_value: function(accept, reject, context, params, value){
    var address = value;

    var regex = /^[^<>()[\]\.\,;\:\\s@\"']+
      (\\. [^<>()[\]\.\,;\:\\s@\""]+)*|(\\".+\\")@((\\[[0-9]{1,3}\\
      [0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\)|
      (([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))$/;

    if (!regex.test(address)) {
      reject(address + " is not valid e-mail address.");
    } else {
      accept();
    }
  }
});
```

Do powyższego fragmentu kodu zostały napisane trzy testy jednostkowe, pokrywające każdą funkcjonalność:

```
describe("FieldType.Email", function() {
  it("should return the description of the field type", function(done) {
    if (typeof field_type_email.declaration.get_description() === "string")
      done();
    else
      done(new Error("But it didn't"));
  });
  it("should check if is_proper_value works correctly" +
    "(given correct date format)", function(done) {
    field_type_email.is_proper_value(new Sealious.Context(), {}, "test@mail.com")
      .then(function() {
        done();
      });
  });
});
```

```

    })
    .catch(function(error) {
        done(new Error(error));
    })
});
it("should check if is_proper_value works correctly" +
    "(given incorrect date format)", function(done) {
    field_type_email.is_proper_value(new Sealius.Context(), {}, "test")
    .then(function() {
        done(new Error("It worked correctly"));
    })
    .catch(function(error) {
        if (error.type === "validation")
            done();
        else
            done(new Error(error));
    })
});
});

```

- Pierwszy test jednostkowy sprawdza, czy deklaracja typu pola jest ciągiem znaków, jeżeli jest, to test wykona się poprawnie, jeżeli nie, to zostanie zwrócony błąd:

```

it("should return the description of the field type", function(done) {
    if (typeof field_type_email.declaration.get_description() === "string")
        done();
    else
        done(new Error("But it didn't"));
});

```

- Drugi test jednostkowy sprawdza, czy weryfikacja adresu email `test@mail.com` przeszła poprawnie, jeżeli nie, to zwrócony zostanie błąd:

```

it("should check if is_proper_value works correctly" +
    "(given correct date format)", function(done) {
    field_type_email.is_proper_value(new Sealius.Context(), {}, "test@mail.com")
    .then(function() {
        done();
    })
    .catch(function(error) {
        done(new Error(error));
    })
});

```

- Trzeci test jednostkowy sprawdza, czy typ pola odrzuci podany adres email `test`, jeżeli

go zaakceptuje, to zwracany jest błąd:

```
it("should check if is_proper_value works correctly" +
  "(given incorrect date format)", function(done) {
  field_type_email.is_proper_value(new Sealius.Context(), {}, "test")
  .then(function() {
    done(new Error("It worked correctly"));
  })
  .catch(function(error) {
    if (error.type === "validation")
      done();
    else
      done(new Error(error));
  })
});
```

Wywołanie powyższych testów w konsoli spowoduje wypisanie ich rezultatów:

```
FieldType.Email
  should return the description of the field type
  should check if is_proper_value works correctly(given correct date format)
  should check if is_proper_value works correctly(given incorrect date format)
```


W przypadku błędnego rezultatu, w konsoli zostanie wypisany komunikat:


```
1) FieldType.Email should check if is_proper_value works correctly(given incorrect date format):
   Error: It worked correctly
```


Mechanizmem ułatwiającym wykonywanie testów jednostkowych w naszym projekcie jest usługa ciągłej integracji **Travis CI**², który wykrywa zmiany w repozytorium na platformie **GitHub**, uruchamia nową wersję projektu i wykonuje testy jednostkowe oraz ostatecznie wysyła maila z rezultatem do osoby, która dokonała zmiany w repozytorium:

²<https://travis-ci.org/>



 **Build #727 passed.**

 39 seconds

 theadimar21

5226954 Changeset →

0.6.19


System message:

Happy Holidays from **Travis CI**! From Dec 19th to Jan 4th, 2016, most of the **Travis CI** team is taking a break to be with family and enjoy the new year. For urgent issues, please email support@travis-ci.com, but response times may be slower.

Want to work for **Travis CI**? We're growing our dev team. Come [join us!](#)

Want to know about upcoming build environment updates?

Would you like to stay up-to-date with the upcoming **Travis CI** build environment updates? We set up a mailing list for you! Sign up [here](#).




[Documentation](#) about **Travis CI**

For help please join our IRC channel [irc.freenode.net#travis](irc://freenode.net/#travis).

Choose who receives these build notification emails in your [configuration file](#).

Would you like to test your private code?

[Travis CI for Private Projects](#) could be your new best friend!



Rysunek 6.2: Raport mailowy wysłany przez Travis CI

Podsumowanie części I

Początki pracy nad projektem nie były łatwe, ponieważ wówczas nie mieliśmy wystarczająco dużo doświadczenia, aby wiedzieć jak efektywniej i wydajniej pracować. Zaczęliśmy eksperymentować i próbować nowych rzeczy, aby przekonać się, co okaże się dla nas najlepszym rozwiązaniem. Z czasem udało nam się stworzyć działający *workflow*, który obecnie jest wydajny i spełnia swoje zadania. Mimo tego ciągle jesteśmy nastawieni na zmiany i szukanie nowych, lepszych rozwiązań. Obecnie praca nad projektem wygląda następująco:

1. Organizacja pracy – spotkania i komunikacja:

- Cotygodniowe spotkania projektowe, na których omawiamy bieżące sprawy.
- Praca we własnym zakresie w ciągu tygodnia.
- Komunikacja mailowa.

2. Obecnie obowiązująca konwencja *branchy* w systemie kontroli wersji **Git**:

1. Główne *branche*:

- *stable* — stabilna wersja kodu, dostępna poprzez ekosystem **npm**.
- *beta* — wersja zawierająca wszystkie zaplanowane zmiany w danej wersji, udostępniona użytkownikom do testów, dostępna poprzez ekosystem **npm**.
- *alpha* — każda nowa funkcjonalność, bądź poprawa istniejących funkcji, dostępna jedynie poprzez **GitHub**.

2. Nazewnictwo *branchy* w których dokonywane są zmiany w kodzie:

- Jeżeli dane zadanie pochodzi z **Trello**: *trello#opis_zadania*.
- Jeżeli dane zadanie pochodzi z *issues* na platformie **GitHub**: *issue#xxx_opis_zadania*, gdzie *xxx* to numer *issue*.

3. Rozwój projektu – dodanie nowej funkcjonalności:

1. Wybór zadania z **Trello**.
2. Utworzone nowego *brancha* w systemie **Git** z odpowiednią nazwą, w którym będzie dokonana zmiana.

3. Praca nad zadaniem.
4. Ukończenie zadania, założenie *pull requesta*, przesunięcie do go listy „Review/Pull Request”.
5. Rozpatrzenie zadania przez innego członka zespołu, jeżeli kod jest poprawny, *git merge* do głównego *brancha*.
6. Zamknięcie *pull requesta*, *issue* (jeżeli zadania zostało również zgłoszone na platformie **GitHub**), przesunięcie zadania do listy „Done in <version>”.

4. Rozwój projektu – naprawienie błędu:

1. Wybór zadania z *issue* na **GitHub**.
2. Utworzenie nowego *brancha* w systemie **Git** z odpowiednią nazwą, w którym będzie dokonana zmiana.
3. Praca nad błędem.
4. Ukończenie zadania, założenie *pull requesta*, jeżeli błąd został wspomniany na **Trello** – przesunięcie zadania do listy „Review/Pull Request”.
5. Rozpatrzenie zadania przez innego członka zespołu, jeżeli kod jest poprawny, *git merge* do głównego *brancha*.
6. Zamknięcie *pull requesta*, *issue*, przesunięcie zadania do listy „Done in <version>” (jeżeli błąd został wspomniany na **Trello**).

5. Rozwój projektu – opublikowanie nowej wersji:

- *Przykład 1:* Wersja projektu na branchu „beta”: 0.8.2. W branchu „alpha” znajdują się wszystkie zmiany w kodzie źródłowym projektu, które miały być dodane w ramach nowej wersji. Dokonywana jest operacja *git merge* z „alpha” do „beta”. Wersja projektu w branchu „beta” zmienia się na: 0.9.0, projekt zostaje opublikowany na platformie **npm**.
- *Przykład 2:* Wersja projektu na branchu „beta”: 0.9.0. Zostaje zgłoszony błąd, który następnie zostaje naprawiony. Wersja projektu zmienia się na 0.9.1, nowa wersja zostaje opublikowana na platformie **npm**.
- *Przykład 3:* Do *brancha* „alpha” zostaje dodana nowa funkcjonalność. Wersja projektu się nie zmienia, projekt nie jest publikowany na platformie **npm**.
- *Przykład 4:* Projekt w *branchu* „beta” ma wersję 0.9.3. Zespół uznaje, że wersja ta jest stabilna. Zostaje dokonana operacja *git merge* z „beta” do „stable”.

6. Rozwój projektu – testy jednostkowe i pokrycia:

- Do każdej funkcjonalności dopisywane są testy jednostkowe, które weryfikują

jej działanie, pomagają wykryć błędy i sprawdzają, czy oprogramowanie działa poprawnie.

- W Sealiousie korzystamy z *frameworka* **MochaJS**, która ułatwia szybkie i dokładne pisanie testów.

Część II

Rozwiązanie wspomagające pracę docelowych programistów

W dzisiejszych czasach przeciętny programista nie musi posiadać dogłębnej wiedzy programowania niskopoziomowego, aby napisać funkcjonalne, efektywne i przede wszystkim spełniające swoje zadanie programy. Powodem tego stanu rzeczy jest ogromna ilość narzędzi programistycznych, które ułatwiają pisanie kodu. Codziennie grupy developerskie wypuszczają nową bibliotekę albo frameworka, które ułatwiają życie setkom, jeżeli nie tysiącom, programistów na całym świecie.

I chociaż nie można zaprzeczyć użyteczności owych narzędzi, głównym problemem, który dotyka wielu użytkowników, jest problem z wdrożeniem się w daną technologię.

Wynika on z niestaranego napisania dokumentacji technicznej, niedoprecyzowania kroków instrukcji obsługi, czy braku sekcji „Najczęściej zadawane pytania” (FAQ).

Brak podstawowych informacji o produkcie, z którego użytkownik korzysta, napawa frustracją oraz zniechęca do korzystania z niego – jest to podstawowa zasada każdego przedmiotu, z którego korzysta człowiek, nie ważne, czy to pralka, telewizor, program komputerowy, czy samolot. Dlatego, w kontekście urządzeń elektronicznych, w latach dziewięćdziesiątych powstała myśl filozoficzna *User Experience*, którą można zdefiniować jako „całość wrażeń, jakich doświadcza użytkownik podczas korzystania z produktu interaktywnego”. [6]

Obecnie każda renomowana marka posiada specjalistów od User Experience, którzy mają za zadanie upewnić się, czy dany produkt jest wystarczająco użyteczny, łatwy do przyswojenia i intuicyjny dla przeciętnego użytkownika. Powstaje więc pytanie, czy twórcy narzędzi programistycznych też powinni upewnić się, że to co dają programistom jest użyteczne, łatwe do przyswojenia i intuicyjne?

Odpowiedź brzmi: **tak**.

Poniższa część zawiera trzy główne elementy, które mają ułatwić docelowym programistom pracę z naszym projektem:

1. **Deklaratywność** — skupienie się końcowym rezultacie, a nie na szczegółowej sekwencji kroków, które mają do niego doprowadzić.
2. **Dokumentacja** — coś co każdy projekt powinien posiadać, jednak nie zawsze twórcy oprogramowania zdają sobie sprawę, jak powinna ona wyglądać i jakie problemy powinna rozwiązywać.
3. **Komunikaty błędów** — dobrze zakomunikowany błąd powinien być sam w sobie wystarczającą informacją pomagającą wykryć programiście co poszło nie tak.

Rozdział 1

Deklaratywność

Sealious jest *frameworkiem* deklaratywnym, co pozwala użytkownikowi skupić się na tym jaki ma być wynik działania aplikacji, jak ma się ona zachowywać i jakie możliwości ma ona dawać. Developerowi zostaje narzucony dobrze przemyślanie *workflow*, przez co nie musi on się zajmować takimi kwestiami technicznymi jak ustanawianie połączenia z bazą danych, projektowania tabel bądź kolekcji, tworzenie zapytań o zasoby, czy ustanawiania kanału komunikacji klient-serwer. Dzięki temu można napisać w pełni działającą aplikację w kilkanaście linii kodu.

Przykład aplikacji

```
var Sealious = require("sealious");

Sealious.init();

new Sealious.ChipTypes.ResourceType({
  name: "owner",
  fields: [
    {name: "first-name", type: "text", required: true},
    {name: "last-name", type: "text", required: true},
    {name: "address", type: "text", required: true},
    {name: "phone-number", type: "int", required: true},
    {name: "email", type: "email"}
  ]
});

new Sealious.ChipTypes.ResourceType({
  name: "pet",
```

```

    fields: [
      {name: "species", type: "animald", required: true},
      {name: "name", type: "text", required: true},
      {name: "age", type: "int", required: true},
      {name: "diagnosis", type: "text", params: {max_length: 200}, required: true}
    ]
  });

Sealious.start();

```

Powyższej znajduje się przykład aplikacji napisanej z użyciem *frameworka* Sealious. W 26 liniach kodu aplikacja wykonała:

1. Połączenie z domyślną nierelacyjną bazą danych **TingoDB**.
2. Zadeklarowanie dwóch kolekcji (bądź tabel w przypadku zastosowania bazy SQL).
3. Zdefiniowanie pól w kolekcji (bądź tabeli) o określonym typie (opcjonalnie można zadeklarować dodatkowe ograniczenia do pola, np. maksymalna długość ciągu znaków).
4. Uruchomienie serwera HTTP wraz ze ścieżkami REST `api/v1/<nazwa zasobu>` (w tym przypadku `api/v1/owner` oraz `api/v1/pet`) oraz automatycznie obsłużenie zapytań GET, POST, PUT, DELETE na tych ścieżkach.

Dokładną analizę przykładu można znaleźć w instrukcji “Sealious Handbook”, o której mowa w rozdziale “Dokumentacja”.

Innym aspektem deklaratywności jest możliwość dodania, wymiany lub modyfikacji poszczególnych składowych *frameworka*:

1. **Kanał komunikacji** — aktualnie jedynym obsługiwanym kanałem komunikacji klient-serwer jest REST, jednak sama struktura projektu została skonstruowana tak, aby nie była przywiązana do jednego rozwiązania. Dlatego możliwe jest napisanie własnego kanału komunikacji, np. WebSocket, i podłączenie go do Sealiousa.
2. **Typy zasobów, strategia dostępu** — każde pole zasobu musi posiadać swój określony typ, np. “int”, “text”, “boolean”, a sam zasób może posiadać strategię dostępu, czyli zestaw reguł opisujących dostępność zasobu. Sealious umożliwia dodanie własnego typu zasobu, bądź strategii dostępu co umożliwia dostosowanie aplikacji do swoich potrzeb.
3. **Baza danych** — domyślną bazą danych jest baza NoSQL **TingoDB**, jednak dzięki wyabstrahowaniu warstwy zarządzania bazodanowego istnieje możliwość napisania własnego sterownika obsługującego inną bazę i podpięcie go do *frameworka*.

Dzięki temu developer może przystosować nasz projekt do swoich aktualnych potrzeb, skupiając się na tym jaki ma być wynik działania, a nie jak ma wyglądać przebieg pracy

aplikacji.

Rozdział 2

Dokumentacja

Dokumentacja projektu jest podstawową rzeczą, o którą należy zadbać, jeżeli chcemy, aby nasz produkt był łatwy do zrozumienia dla osób trzecich. Jednak niewielu twórców oprogramowania rozumie, że sama dokumentacja techniczna, czyli opis każdej funkcjonalności wystarcza.

W naszym projekcie jednym z głównym założeń było to, aby każdy użytkownik mógł z łatwością wdrożyć w projekt, dlatego postanowiłem się zająć tworzeniem instrukcji obsługi, którą nazwaliśmy „Sealious Handbook” (załącznik). Zawiera on podstawowe informacje o tym jak napisać prostą aplikację w oparciu o nasz projekt inżynierski oraz wyjaśnia podstawowe pojęcia takie jak „FieldType”, czy „AccessStrategy” oraz odpowiada na pytania związane z działaniem projektu. Instrukcja ta jest ciągle rozwijana, ponieważ pragnę, aby użytkownik nie musiał myśleć o tym **jak** ma stworzyć swój program, ale **co** dokładniej chce mieć i **jaki** powinien być efekt działania.

Aby lepiej zrozumieć jak należy napisać „handbook”, aby czytelnik mógł znaleźć w niej to co potrzebuje w łatwy i prosty sposób, przeczytałem książkę „Don’t make me think” autorstwa Steve’a Kruga [7], która choć napisana jest przede wszystkim dla twórców stron internetowych, może być cenną nauką dla wszystkich, którzy starają się tworzyć coś dla innych ludzi. Oto najważniejsze zasady jakie zastosowałem w instrukcji obsługi (wszelkie cytaty są przetłumaczone i/lub sparafrazowane z angielskiej wersji książki):

1. „My nie czytamy stron [internetowych]. My je skanujemy.” — jeżeli poszukujemy szybkiej odpowiedzi w internecie na nasze pytania, w większości przypadków nie czytamy dokładnie, słowo w słowo, każdego wyniku wyszukiwania, czy każdego artykułu, który otworzymy. Szukamy raczej słów kluczowych, które mogą okazać się

potencjalną odpowiedzią, bądź wskazówką do uzyskania odpowiedzi.

2. „Często używaj nagłówków” — jeżeli coś nadaje się do tego, aby stać się nagłówkiem, to najprawdopodobniej powinno nim być. Należy również pamiętać, że jest wiele wielkości nagłówków, które dzielą tekst na części, sekcje, rozdziały i tym podobne. Ponadto nagłówek powinien być bliżej tekstu, do którego wprowadza niż do tego, który go poprzedza.
3. „Skracaj akapity” — długi akapit sprawia, że czytelnik ma do czynienia z tak zwaną „ścianą tekstu”, co znacznie utrudnia czytanie, przez co prawdopodobieństwo pominięcia go wzrasta.
4. „Jeżeli coś może być wypunktowane, a nie jest: wypunktuj to” — Dobrymi kandydatami do wypunktowania są rzeczy wymieniane po przecinku, czy średniku. Dobrą praktyką jest też ustawianie dodatkowej przestrzeni pomiędzy każdym punktem. Zwiększa to czytelność tekstu.
5. „Wyróżniaj kluczowe terminy” — skanowanie tekstu oznacza wyszukiwanie słów, które mają potencjał doprowadzenia nas do pożądanej odpowiedzi. Pomaga w tym wyróżnienie poprzez pogrubienie, podkreślenie, czy kursywę wyrazów, czy nawet całych zdań, które są w danej sekcji tekstu kluczowe.
6. „Usuń niepotrzebne słowa ze zdań i zdania z akapitów” — używanie nadmiarowych słów zwiększa tekst objętościowo, jednak nie przynosi żadnej wartości merytorycznej. Dlatego należy formułować zdania i akapity tak, aby były konkretne i zwarte. Dzięki temu, że zdania i akapity są krótsze, użytkownik może przeskanować więcej tekstu bez potrzeby przewijania strony.

Obecnie „Sealious Handbook” zawiera sekcje:

1. *Getting started* — opisuje co jest potrzebne, aby pracować z użyciem naszego projektu.
2. *Sample Sealious app* — opisuje jak napisać przykładową aplikację opartą na Sealiousie.
 - Declaring our first ResourceType
 - Adding more resources
 - Final result
3. *Base chips* – sekcja ta opisuje czym są base chips i jak je wykorzystać w projektach.
 - Access Strategy
 - What is an access strategy
 - How to use access strategy
 - Access strategies in Sealious

- Creating a new access strategy
 - Common questions and errors
- Field Types
 - What are field types
 - How to use field types
 - Field types in Sealious
 - Creating a new field type
 - Common questions and errors

Treść instrukcji jest ciągle dodawana i zmieniana, aby upewnić się, że będzie wykonywała swoje zadanie, jednak prawdziwym sprawdzianem jej użyteczności będzie oficjalne wypuszczenie naszego projektu. Chcemy zachęcać użytkowników do tego, aby dawali nam uwagi odnośnie użyteczności „Sealious Handbook”, ponadto oferujemy im możliwość posiadania własnego wkładu w tę pracę, ponieważ całość jest hostowana na serwisie **GitHub**.

Obecna wersja instrukcji znajduje się w załączniku.

Rozdział 3

Komunikaty błędów

Błędy w kodzie są tak zwanym „chlebem powszednim” w życiu każdego programisty. Pojawiają się najczęściej wtedy, gdy zaczynamy pracę z nowym, nieznanym dotąd przez nas środowiskiem. I chociaż komunikaty informujące nas o problemie, który wystąpił w trakcie tworzenia oprogramowania mogą być irytujące, błąd którego nie rozumiemy jest o wiele większym problemem.

Jeżeli tworzy się produkt dla programistów, poprawność komunikatów jest czymś, co należy zwrócić szczególną uwagę. Dlatego w naszym projekcie inżynierskim dokładamy starań, aby każdy komunikat był czytelny i jasny w swoim przekazie. Ponadto, w „Sealious Handbook”, o którym mowa w poprzednim rozdziale, zawiera sekcje „Najczęstsze błędy i pytania”, które mają na celu pomóc w razie błędu.

W Sealiousie definiujemy kilka stworzonych przez nas rodzajów błędów. Część odpowiada za błędy związane z przetwarzaniem zasobów, część informuje o błędzie w pracy serwera, a jeszcze inne są wynikiem błędu programisty korzystającym z naszego projektu, jednak każdy błąd jest inny i jest sam w sobie komunikatem informującym o wystąpieniu wyjątku.

Ponieważ obecnie jedynym dostępnym kanałem komunikacyjnym pomiędzy serwerem, a klientem jest REST, każdy z błędów zawiera jako parametr adekwatny kod HTTP, zwracany klientowi przy każdym zapytaniu. Dołożyliśmy starań, aby odpowiedź na każde zapytanie zawierała poprawny kod, zgodnie ze stylem architektury oprogramowania REST. Niestety często się zdarza, że serwer HTTP wyświetla stronę WWW z komunikatem „Nie znaleziono” wraz z kodem **200**, który oznacza „OK”, podczas gdy powinien zwrócić kod **404**, oznaczający „*Not found*”. Rozbieżność ta jest niezgodna z konwencją i jest niestety powszechną praktyką wśród programistów.

Specyfikacja standardu HTTP, **RFC 2616** [8], wydana w 1999 roku, opisuje wszystkie

wówczas ustandaryzowane kody statusów HTTP jako trzycyfrowe numery dołączone do odpowiedzi HTTP. Umożliwiają one szybkie zakomunikowanie klientowi jak daną operację odczytał i przetworzył serwer. Obecnie, w wyniku rozwoju internetu, zostało wydanych wiele doprecyzowań i uaktualnień do standardu HTTP, które czynią RFC 2616 przestarzałym. Dlatego dobrym podsumowaniem wszystkich zmian jest książka „RESTful Web APIs” autorstwa Leonarda Richardsona, Mike’a Amundsena i Sama Ruby’ego [9], która zawiera aktualny spis wszystkich kodów statusów HTTP, wraz z częstotliwością ich używania.

Kody statusów używane w Sealiousie (ich definicja jest tłumaczeniem i/lub sparafrazowaniem fragmentów książki „RESTful Web APIs”:

- **200** (*OK*) — operacja przebiegła pomyślnie, klient otrzymał poprawną odpowiedź na zapytanie.
- **201** (*Created*) — serwer wysłał ten kod, kiedy utworzył nowy zasób po zapytaniu klienta.
- **204** (*No Content*) — serwer pomyślnie wykonał operację, ale odmawia wysłania reprezentacji zasobu, bądź opisu operacji. Zazwyczaj używany przy operacji DELETE lub PUT.
- **400** (*Bad Request*) — klient wysłał zapytanie w poprawnym formacie, ale serwer nie potrafi go odczytać.
- **401** (*Unauthorized*) — klient wysłał zapytanie o chroniony zasób bez ówczesnej autoryzacji, która może wynikać z podania błędnych danych (loginu i hasła, klucza API, tokenu autoryzacji), bądź nie podania ich wcale.
- **403** (*Forbidden*) — zapytanie klienta jest poprawnie sformułowane, ale serwer nie chce je wykonać. Nie chodzi tu o brak autoryzacji, gdyż wtedy mamy do czynienia z kodem 401, lecz bardziej o dostępność zasobu tylko w określonych przedziałach czasowych, bądź z określonego IP.
- **404** (*Not Found*) — serwer nie jest w stanie zlokalizować zasobu, o który prosi klient. Czasem w celach bezpieczeństwa jest wykorzystywany zamiast kodów 401 i 403, które mówią o tym, że zasób istnieje, ale klient nie ma do niego dostępu.
- **409** (*Conflict*) — zapytanie nie może być przetworzone z powodu konfliktu stanu zasobu (np. zapytanie o utworzenie zasobu, który już istnieje).
- **418** (*I’m a teapot*) — kod ten nie istnieje w standardzie, ale jest zaimplementowany w projekcie jako „easter egg”. Klient wysłał prośbę o kawę, jednak serwer nie może jej utworzyć, ponieważ jest czajnikiem (imbrykiem) do herbaty.
- **500** (*Internet Server Error*) — błąd w działaniu serwera.

W Sealiousie definiujemy kilka stworzonych przez nas rodzajów błędów:

- *ValidationError* — pojawia się, gdy podana wartość jest nieprawidłowa, kod **403**.
- *ValueExists* — pojawia się, gdy chcemy dodać wartość, która już istnieje, kod **409**.
- *InvalidCredentials* — pojawia się, gdy dane logowania są nieprawidłowe, kod **401**.
- *NotFound* — pojawia się, gdy nie znaleziono pożądanego zasobu, kod **404**.
- *InternalConnectionError* — pojawia się, gdy wystąpił błąd wewnątrz serwera, kod **500**.
- *DependencyError* — pojawia się, gdy wymagany jest moduł, bądź składowa projektu, która nie została znaleziona, kod **500**.
- *UnauthorizedRequest* — pojawia się, gdy wykonane jest zapytanie o zasób, do którego użytkownik nie ma dostępu, kod **401**.
- *DeveloperError* — pojawia się, gdy programista korzystający z Sealioua wykona nieprawidłową operację, brak kodu.
- *BadContext* — pojawia się, gdy kontekst, wewnątrz którego rozpatrywane jest zapytanie, jest nieprawidłowy, kod **401**.

Jak łatwo zauważyć, niektóre z tych błędów są bardzo konkretne, a niektóre ogólne. Wynika to z częstotliwości ich występowania, przykładowo o wiele częściej użytkownik podaje nieprawidłowe dane do logowania, niż występuje błąd pracy serwera.

W Sealiusie każde zapytanie przechodzi przez specjalną funkcję `custom_reply_function()`, która “buduje” odpowiedź w zależności od argumentów, które są jej dostarczone:

```
function custom_reply_function(original_reply_function, request_details, obj, status_code){  
    var client_ip = request_details.info.remoteAddress;  
    var mime_type = request_details.mime;  
    var request_description = "\t" + request_details.method +  
        " " + request_details.path + "\n\t\t\tfrom: " + client_ip +  
        ", mime: " + mime_type + "\n\t\t\tresult: ";  
  
    var ret;  
  
    if(request_details.headers["x-http-method-override"]){  
        request_details.method = request_details.headers["x-http-method-override"];  
    }  
  
    if(request_details.payload && request_details.payload["x-http-method-override"]){  
        method = request_details.payload["x-http-method-override"];  
        delete request_details.payload["x-http-method-override"];  
    }  
  
    if(obj==undefined){  
        obj={};  
    };  
  
    if(obj.is_sealious_error || obj.is_error){
```

```

    var res = Sealious.Response.fromError(obj);
    Sealious.Logger.error(request_description+"failed - "+obj.status_message);
    ret = original_reply_function(res);
    ret.code(obj.http_code);
    console.log(obj.stack);
  }else if(obj instanceof Error){
    Sealious.Logger.error(request_description);
    Sealious.Logger.error(obj);
    console.log(obj.stack);
    var err = Sealious.Errors.Error("Internal server error")
    var res = Sealious.Response.fromError(err);
    ret = original_reply_function(res).code(err.http_code);
  }else{
    Sealious.Logger.info(request_description + "success!");
    var processed_obj = process_sealious_response(obj)
    ret = original_reply_function(processed_obj);
  }
  return ret;
}

```

Kluczowe są linijki 16-37, które sprawdzają, czy obiekt, o którego pyta klient jest błędem, czy nie i w zależności od tego tworzona jest odpowiedź na zapytanie.

W przypadku błędu w konsoli pojawia się następujący komunikat:

```

20:21:12.825 - error:   PUT /api/v1/places/ehuyta04pkfff
                    from: 127.0.0.1, mime: multipart/form-data
                    result:
20:21:12.825 - error:  TypeError: Cannot read property 'toString' of undefined
                    ...
                    /*stos błędów*/
                    ...

```

Podsumowanie części II

Pomoc przyszłym użytkownikom naszego projektu traktujemy bardzo poważnie. Dlatego staramy się dostarczyć im narzędzie, które będzie wspierało ich pracę, a nie przeszkadzało w niej. Trzema głównymi kategoriami, na których się skupiamy to:

1. **Deklaratywność** — dzięki dobrze przemyślanej strukturze projektu, developer pisząc aplikację wykorzystującą Sealiousa, skupia się tylko na tym co chce uzyskać i jaki ma być efekt końcowy, nie przejmując się takimi zagadnieniami technicznymi jak połączenie i obsługa bazy danych, zapytanie o zasoby, czy ustanawianie kanału komunikacji klient-serwer. Jest to znaczne ułatwienie dla użytkownika, któremu zostanie jedynie kwestia reprezentacji danych.
2. **Dokumentacja** — zależy nam, aby wdrożenie się w nasz projekt odbywało się bez problemów, dlatego chcemy dostarczyć przyszłym użytkownikom materiały potrzebne do realizacji tego celu. Można je podzielić na dwie kategorie:
 1. Dokumentacja techniczna — opis każdej funkcjonalności, która istnieje w Sealiousie.
 2. Instrukcja obsługi – dokument napisany w bezpośrednim języku niezawierającym skomplikowanych terminów, informujący czytelnika co jest potrzebne do pracy z naszym projektem i jak z niego korzystać. „Sealious Handbook”, bo taką ta instrukcja ma nazwę, w chwili obecnej zawiera trzy sekcje:
 1. *Getting started* — opisuje co jest potrzebne, aby pracować z użyciem naszego projektu.
 2. *Sample Sealious app* — opisuje jak napisać przykładową aplikację Sealious’ową.
 3. *Base chips* — opisuje czym są base chips i jak je wykorzystać w projektach opartych na Sealiousie.
3. **Komunikaty błędów** — niejasny i źle opisany błąd jest znacznym utrudnieniem dla programisty. W tym celu dokładamy starań, aby użytkownik po zobaczeniu

komunikatu wyjątku wiedział, gdzie leży wina, z czego wynika i jak można ją naprawić. Ponadto, wykorzystując REST jako kanał komunikacji między serwerem a klientem, każda odpowiedź posiada swój własny kod statusu HTTP, zgodnie ze standardem.

Obecnie Sealious obsługuje następujące kody:

- **200** (*OK*) — operacja przebiegła pomyślnie, klient otrzymał poprawną odpowiedź na zapytanie.
- **201** (*Created*) — serwer wysyła ten kod, kiedy utworzył nowy zasób po zapytaniu klienta.
- **204** (*No Content*) — serwer pomyślnie wykonał operację, ale odmawia wysłania reprezentacji zasobu, bądź opisu operacji. Zazwyczaj używany przy operacji DELETE lub PUT.
- **400** (*Bad Request*) — klient wysłał zapytanie w poprawnym formacie, ale serwer nie potrafi go odczytać.
- **401** (*Unauthorized*) — klient wysłał zapytanie o chroniony zasób bez ówczesnej autoryzacji, która może wynikać z podania błędnych danych (loginu i hasła, klucza API, tokenu autoryzacji), bądź nie podania ich wcale.
- **403** (*Forbidden*) — zapytanie klienta jest poprawnie sformułowane, ale serwer nie chce je wykonać. Nie chodzi tu o brak autoryzacji, gdyż wtedy mamy do czynienia z kodem 401, lecz bardziej o dostępność zasobu tylko w określonych przedziałach czasowych, bądź z określonego IP.
- **404** (*Not Found*) — serwer nie jest w stanie zlokalizować zasobu, o który prosi klient. Czasem w celach bezpieczeństwa jest wykorzystywany zamiast kodów 401 i 403, które mówią o tym, że zasób istnieje, ale klient nie ma do niego dostępu.
- **409** (*Conflict*) — zapytanie nie może być przetworzone z powodu konfliktu stanu zasobu (np. zapytanie o utworzenie zasobu, który już istnieje).
- **418** (*I'm a teapot*) — kod ten nie istnieje w standardzie, ale jest zaimplementowany w projekcie jako „easter egg”. Klient wysłał prośbę o kawę, jednak serwer nie może jej utworzyć, ponieważ jest czajnikiem (imbrykiem) do herbaty.
- **500** (*Internet Server Error*) — błąd w działaniu serwera.

Część III

Załączniki

Table of contents

1. Getting started
2. Sample Sealious app
 - Declaring our first ResourceType
 - Adding more resources
 - Final result
3. Base chips
 - Access Strategy
 - What is an access strategy
 - How to use access strategies
 - Access strategies in Sealious
 - Creating a new access strategy
 - Common questions and errors
 - Field Types
 - What are field types
 - How to use field types
 - Field types in Sealious
 - Creating a new field type
 - Common questions and errors

Getting started

Before starting your journey with Sealious, make sure Node.js is installed and set. Also we'll be using MongoDB in our examples.

1. Open command line in your desired directory.
2. Type in `npm init`. This is create `package.json` file, which will hold all crucial data about your project.
 - You will be prompted several times, you can just ENTER through.
3. Once `package.json` is set, type in `npm install --save sealious`. This will download Sealious from npm (Node Package Manager) and save the dependency in your `package.json`.
4. We need to have some means of communication with the server. Type in `npm install --save sealious-www-server` and `npm install --save sealious-channel-rest`.
5. To set the database, type in `npm install --save sealious-datastore-mongo`.

And that's it. Note that Sealious reads its dependencies from `package.json`, so don't forget to create it.

Simple Sealious app

In this section we will show you how to create a fully functional and working back-end application, with REST routes and database handling set, in just **28** lines:

```
1  var Sealious = require("sealious");
2
3  Sealious.init();
4
5  require("./field-type.animal.js");
6
7  new Sealious.ChipTypes.ResourceType({
8    name: "owner",
9    fields: [
10      {name: "first-name", type: "text", required: true},
11      {name: "last-name", type: "text", required: true},
12      {name: "address", type: "text", required: true},
13      {name: "phone-number", type: "int", required: true}
14      {name: "email", type: "email"}
15    ]
16  });
17
```

```

18 new Sealius.ChipTypes.ResourceType({
19   name: "pet",
20   fields: [
21     {name: "species", type: "animald", required: true},
22     {name: "name", type: "text", required: true},
23     {name: "age", type: "int", required: true},
24     {name: "diagnosis", type: "text", params: {max_length: 200}, required: true}
25   ]
26 });
27
28 Sealius.start();

```

We'll call this project **Veterinarian Clinic**. Basically it's main task is to store information about pets and their owners.

Declaring our first ResourceType

ResourceType is the core concept of Sealius. Without it, Sealius has nothing to work on.

Let's create a new resource named **owner**.

In the real world, a pet must have a name and some age. Here's how we'll represent it in Sealius:

```

1  var Sealius = require("sealius");
2
3  Sealius.init();
4
5  new Sealius.ChipTypes.ResourceType({
6    name: "owner",
7    fields: [
8      {name: "first-name", type: "text", required: true},
9      {name: "last-name", type: "text", required: true},
10     {name: "address", type: "text", required: true},
11     {name: "phone-number", type: "int", required: true}
12     {name: "email", type: "email"}
13   ]
14 });
15
16 Sealius.start();

```

This app consists of four parts:

- `var Sealius = require("sealius");` - a reference to Sealius,

- `Sealious.init();` - loads Sealious components,
- `new Sealious.ChipTypes.ResourceType({})` - defines a new resource-type that will be our data model,
- `Sealious.start();` - prepares REST routes, sets the database, loads dependencies - gets the whole thing started.

This is all we need to start a new Sealious app.

From now on we can communicate with the server through REST routes (the default Sealious channel):

- GET on URL `api/v1/owner` returns all `owner` resources,
- GET on URL `api/v1/owner/{owner_id}` returns a specific `owner` resource,
- POST on URL `api/v1/owner` with body `{name: <text>, age: <int>}` creates a new `owner` resource with given body,
- PUT and PATCH on URL `api/v1/owner/{owner_id}` with body `{name: <text>, age: <int>}` modifies the `owner` resource,
- DELETE on URL `api/v1/owner/{owner_id}` deletes a specific `owner` resource.

Q: *How can I communicate with the server if I don't want to use REST?*

A: We give the developers channel REST by default. If you want to use other channel you are free to write one yourself. See X for more information.

Q: *I have no idea what's happening in this `ResourceType` thing...*

A: Let's walk through it step by step:

1. `new Sealious.ChipTypes.ResourceType({` - this line creates a new `ResourceType` object, that will enable you to work on your resources.
2. `name: "owner",` - this is the name of our resource, that we can work with.
3. `fields: [` - we define `fields` array that holds information about what type of data our resource can store.
4. `{name: "first-name", type: "text", required: true},` - this is the field used by our resource. The name of the field is `name`, its type is `text` and it must be declared (`required: true`).
5. `{name: "last-name", type: "int", required: true}` - this is the another field used by our resource. The name of the fields is `age`, its type is `int` and it must be declared (`required: true`).

6. `{name: "email", type: "email"}` - this defines the field named `email` with type `email` - it only accepts strings that look like and email. Note that `required: true` is absent - that means, that you don't have to include this field and the request will still be parsed correctly.

And that's it.

If you want to know more about `FieldType`, click [here](#).

Adding more resources

You can add as many resources as you need. In our example, we'll add `pet` resource.

```
1  var Sealious = require("sealious");
2
3  Sealious.init();
4
5  new Sealious.ChipTypes.ResourceType({
6    name: "owner",
7    fields: [
8      {name: "first-name", type: "text", required: true},
9      {name: "last-name", type: "text", required: true},
10     {name: "address", type: "text", required: true},
11     {name: "phone-number", type: "int", required: true}
12     {name: "email", type: "email"}
13   ]
14 });
15
16 new Sealious.ChipTypes.ResourceType({
17   name: "pet",
18   fields: [
19     {name: "species", type: "text", required: true},
20     {name: "name", type: "text", required: true},
21     {name: "age", type: "int", required: true},
22     {name: "diagnosis", type: "text", params: {max_length: 200}, required: true}
23   ]
24 });
25 Sealious.start();
```

As you can see, the `pet` resource has *four* fields:

1. `species`,
2. `name`,

3. age,
4. diagnosis.

The new thing here is `params: {max_length: 200}` in field `diagnosis`. This parameter says, that only strings with length up to 200 characters.

And that's it. We've finished our **Veterinarian Clinic**!

Q: *But wait, the first code block you showed us had line `require("./field-type.animal.js");`. What's with this?*

A: Nice eye. Here's the explanation:

Sealious let's you define your own field-types, that will validate, encode or decode (and many more - see API reference) the values that's in the field.

For our example, we've defined a new field-type, that checks if given animal species is acceptable by our clinic:

```
1  var Sealious = require("sealious");
2
3  new Sealious.ChipTypes.FieldType({
4    name: "animal",
5    get_description: function(context, params){
6      return "Only accepts dogs, cats and parrots.";
7    },
8    is_proper_value: function(accept, reject, context, params, new_value){
9      var acceptable_animals = ['dog', 'cat', 'parrot'];
10     var result = acceptable_animals.find( x => x === new_value);
11
12     result ? accept() : reject("This species of animal is not accepted");
13   }
14 });
```

If you want to know more about `FieldType`, click [here](#).

Now if we include our newly created field-type, we can check what's the species of the pet we want to add to our resources.

Final result:

```
1  var Sealious = require("sealious");
2
3  Sealious.init();
```



```

4
5  require("../field-type.animal.js");
6
7  new Sealius.ChipTypes.ResourceType({
8      name: "owner",
9      fields: [
10         {name: "first-name", type: "text", required: true},
11         {name: "last-name", type: "text", required: true},
12         {name: "address", type: "text", required: true},
13         {name: "phone-number", type: "int", required: true}
14         {name: "email", type: "email"}
15     ]
16 });
17
18 new Sealius.ChipTypes.ResourceType({
19     name: "pet",
20     fields: [
21         {name: "species", type: "animald", required: true},
22         {name: "name", type: "text", required: true},
23         {name: "age", type: "int", required: true},
24         {name: "diagnosis", type: "text", params: {max_length: 200}, required: true}
25     ]
26 });
27
28 Sealius.start();

```

Sealius base chips

Access Strategy

I. What is an access strategy?

Access strategy is a function that takes a context as an argument and based on it either allows or denies access to certain resources or operations.

II. How to use access strategies

One of the possible ways to use an access strategy is when defining a new **resource**.

```

new Sealius.ResourceType({
    name: "nobody_can_update_me",
    fields: [{ name: "value", type: "text" }],
    access_strategy: {

```

```

        update: "noone"
    }
});

```

In this example we use Access Strategy `noone` that rejects any `update` request.

III. Access strategies in Sealious

Sealious comes with *three* pre-defined access strategies. that are located in `lib/base_chips`. File `lib/base-chips/_base-chips.js` defines the order of access strategies initilizing.

1. Just owner:
 - only the owner of the resource can modify it
2. Noone:
 - noone can modify the resource
3. Public:
 - Everybody has access to the resource.

IV. Creating a new access strategy

V. Common questions and errors

Field-Types

I. What are field types?

Each “field” in a `ResourceType` must have a field-type assigned. Field-types describe which values can and which cannot be assigned to a field. Field-type’s behaviour can be adjusted using field-type parameters.

A field-type can accept or reject a value, with appropriate error message.

It’s a field-type’s responsibility to describe how to store it’s values in a datatore.

II. How to use field types?

The example below shows a simple Sealious app:

```

1  var Sealious = require("sealious");
2
3  Sealious.init();
4

```

```

5  new Sealious.ChipTypes.ResourceType({
6      name: "person",
7      fields: [
8          {name: "name", type: "text", params: {max_length: 25}, required: true},
9          {name: "age", type: "int", required: true},
10         {name: "hair-color", type: "color"}
11     ]
12 });
13
14 Sealious.ChipManager.get_chip("channel", "rest").set_url_base("/api/v1");
15
16 Sealious.start();

```

Several things to note:

1. There are *three* fields declared:
 - name, which is of `text` type,
 - age, which is of `int` type,
 - hair-color, which is of `color` type.
2. Each field-type describes how the field behaves and what value it contains. For example, `field-type.color` accepts values like:
 - "black",
 - "#000000",
 - {r: 0, g: 0, b: 0}, but rejects values such as "this is my color".
3. Field `name` has defined an object called `params`, which provides additional information about the field-type. In this case, `name` can have no more than 25 characters. Every field can have `params` object to specify what values can be accepted and what rejected.

Q: *But how can I actually use those field-types?*

A: Good question.

Sealious uses channel REST by default. You can send HTTP request on URL `http://localhost/api/v1/<resource_name>`. In our case, `<resource_name>` is `person`.

This means that you can send, for example, HTTP POST request on URL `http://localhost/api/v1/person` with body:

- name: Maurice,
- age: 21,
- hair-color: blue,

because of the fields defined in our app.

The screenshot shows a REST client interface with the following details:

- Method: POST
- URL: http://localhost/api/v1/person
- Buttons: Params, Send, and a save icon.
- Tabs: Authorization, Headers (0), Body, Pre-request script, Tests.
- Body Type: form-data (selected), x-www-form-urlencoded, raw, binary.
- Form Fields:
 - name: Maurice (Text field)
 - age: 21 (Text field)
 - hair-color: blue (Text field)
- Table with columns: Key, Value, and a dropdown menu (Text).

Rysunek 3.1: HTTP POST request

Sending this request will result in adding a new user named **Maurice**, who is **21** years old and has **blue** hair color to the database.

III. Field types in Sealious

Sealious comes with *eleven* pre-defined field-types. that are located in `lib/base_chips`. File `lib/base-chips/_base-chips.js` defines the order of field-type initializing.

1. Boolean
 - boolean value: `true` or `false`,
 - can be a string: `"true"` or `"false"`,
 - as well as numeric: 1 or 0.
2. Color
 - `"black"`,
 - `"#000000"`,
 - `{r: 0, g: 0, b: 0}`.
3. Date
 - date standard ISO 8601 YYYY-MM-DD,
 - rejects other date formats.
4. Datetime
 - timestamp - amount of miliseconds since epoch.
5. Email
 - email address, like `something@something.sth`.
6. File
7. Float
 - a float number.
8. Hashed-text

- hash with chosen algorithm (default md5),
- designed to hash passwords:
 - can specify 1 or more required numbers,
 - can specify 1 or more required capital letters,
 - can specify 1 or more required numbers and capital letters.

9. Int

- an integer number.

10. Reference

11. Text

- can specify maximum string length,
- can specify minimum string length.

IV. Creating a new field type

In this section we will show you step by step how to create your own field-type. We will use `field-type.color` for this example.

1. Create your field-type file in `lib/base-chips/`.
2. Add reference to `lib/base-chips/_base-chips.js`:

```
// some requires...
require("./field_type.color.js"); //add this
require("./field_type.file.js");
// some requires...
```

3. Write the skeleton of your new field-type.

```
1  var Sealious = require("sealious");
2
3  new Sealious.ChipTypes.FieldType({
4
5  });
```

4. The most basic form of field-type consists of `name` property and `is_proper_value` method. Let's add them!

```
1  var Sealious = require("sealious");
2
3  new Sealious.ChipTypes.FieldType({
4    name: "color", // important! This is the name of your field-type
5    is_proper_value: function(accept, reject, context, params, new_value){
6      // checks if `new_value` is correct. If not, it rejects the request.
7    }
8  });
```

Note: is_proper_value method checks if new_value is correct. If not, it rejects the request.

5. Since we want to create a field-type that will accept colors, we have to implement is_proper_value method properly, so that it will accept values like "black" or "#123FA5" and reject values like "silly sealy". In this example we will use a module from npm called color.

```
1  var Sealious = require("sealious");
2  var Color = require("color");
3
4  new Sealious.ChipTypes.FieldType({
5    name: "color", // important! This is the name of your field-type
6    is_proper_value: function(accept, reject, context, params, new_value){
7      // checks if `new_value` is correct. If not, it rejects the request.
8      try {
9        if (typeof (new_value) === "string")
10          Color(new_value.toLowerCase());
11        else
12          Color(new_value);
13      } catch (e){
14        reject("Value `" + new_value + "` could not be parsed as a color.");
15      }
16      accept();
17    },
18  });
```

And now, is_proper_value will accept those colors, that can be parsed correctly by color module. If the parsing cannot be done, the method will reject the new_value argument.

6. Voila! We have created a new field-type that accepts colors!

Q: *Is it all?*

A: Yes, this is how you create a new field-type. But there **can** be more to it.

Let's say, that we want to store some colors in the database. This may be troublesome, because of several ways to define a color:

- black (explicitly),
- {r: 10, g: 2, b: 200} (RGB object),
- "#115DFA" (hex value).

So the safer approach would be parsing the colors to one, standardized form.

This is what `encode` ensures - it transforms (or encodes) the value that was given to `is_proper_value` to what we want.

```
1  var Sealious = require("sealious");
2  var Color = require("color");
3
4  new Sealious.ChipTypes.FieldType({
5    name: "color", // important! This is the name of your field-type
6    is_proper_value: function(accept, reject, context, params, new_value){
7      // checks if `new_value` is correct. If not, it rejects the request.
8      try {
9        if (typeof (new_value) === "string")
10          Color(new_value.toLowerCase());
11        else
12          Color(new_value);
13      } catch (e){
14        reject("Value `" + new_value + "` could not be parsed as a color.");
15      }
16      accept();
17    },
18    encode: function(context, params, value_in_code){
19      var color = Color(value_in_code);
20      return color.hexString();
21    }
22  });
```

And that's it. Now you can use `color` as a field-type in your app field declaration.

V. Common questions and errors

Q: I created a new field-type and I want to use it in my app. But when I create a new field with my field-type and start the app, I get this error:

Error: In declaration of resource type 'person': unknown field type 'my-new-field-type' in field 'name'

A: There are several causes that may throw this error (**not** including typos):

1. Did you remember to add your new field-type reference to `/lib/base-chips/_base-chips.js`?

```
// some requires
require("./field_type.color.js");
require("./field_type.file.js");
require("./your-new-field-type.js") // your field-type
// some requires
```

2. Did you remember to include `name` property in your field-type declaration?

```
new Sealious.ChipTypes.FieldType({
  name: "my-new-field-type", // important! This is the name of your field-type
  //rest of the declaration
});
```

Q: *I used my new field-type in my app declaration, everything is okay when the app starts, but when I try to use HTTP POST request, nothing happens. No error, no response, nothing.*

A: Make sure that in your field-type declaration, the `is_proper_value` method uses `accept()` argument.

```
is_proper_value: function(accept, reject, context, params, number){
  // checks if `new_value` correct. If not, it rejects the request.
  var test = parseFloat(number);
  if (test === null || test === NaN || isNaN(number) === true) {
    reject("Value `" + number + "` is not a float number format.");
  } else {
    accept(); // remember to add this
  }
}
```

Q: *I try to create a new field-type that inherits (extends) from an already-existing field-type. I added the reference to `/lib/base-chips/_base-chips.js`, but all I get is this error:*

Error: ChipManager was asked to return a chip of type ``field_type`` and name ``<already-existing``

A: File `/lib/base-chips/_base-chips.js` contains a list of currently used field-types. Those field-types are loaded one after another, sequentially. This error occurs, when you want to load your field-type **before** the already existing field-type you want to inherit from.

Make sure that Sealious **first** loads that existing field-type, and **then** yours:

```
// some requires
require("./already-existing-field-type.js") // this is above your field-type
require("./your-field-type.js"); // this is your field-type
// some requires
```


Część IV

Bibliografia

Bibliografia

- [1] Ewangelia Matusza 16:26, Biblia Warszawska, Brytyjskie i Zagraniczne Towarzystwo Biblijne w Polsce, Czerwiec 1975.
- [2] Henry Ford, Data nieznana, „*Nothing is particularly hard if you divide it into small jobs*” <http://www.goodreads.com/quotes/31505-nothing-is-particularly-hard-if-you-divide-it-into-small>, (dostęp: 14.01.2016).
- [3] Autor nieznany, Data nieznana, *Boss vs Leader* <http://i.imgur.com/EZb2lxu.jpg>, (dostęp: 27.12.2015).
- [4] Praca zbiorowa (Wikipedia), 19.11.2015, *System kontroli wersji* https://pl.wikipedia.org/wiki/System_kontroli_wersji, (dostęp: 27.12.2015).
- [5] npm, Data nieznane, *Semantic version*, <https://docs.npmjs.com/getting-started/semantic-versioning>, (dostęp: 27.12.2015).
- [6] Praca zbiorowa (Wikipedia), 15.11.2015, *User Experience* https://pl.wikipedia.org/wiki/User_experience, (dostęp: 27.12.2015).
- [7] Steve Krug, *Don't make me think, Revisited. A common sense approach to Web usability* New Riders, 2014.
- [8] W3C, *RFC 2616* <https://www.ietf.org/rfc/rfc2616.txt>, Czerwiec 1999.
- [9] Leonard Richardson, Mike Amundsen, Sam Ruby, *RESTful Web APIs* O'Reilly, Beijing - Cambridge - Farnham - Koln - Sebastopol - Tokyo, *Appendix A. The Status Codex*