

POLITECHNIKA POZNAŃSKA
WYDZIAŁ INFORMATYKI
INSTYTUT INFORMATYKI



PRACA DYPLOMOWA

INŻYNIERSKA

**Zastosowanie steganografii do wymiany informacji na forum
internetowym**

Bartosz Kostaniak, Daniel Koza

Promotor: prof. dr hab. inż. Jerzy Brzeziński
Instytut Informatyki
Politechniki Poznańskiej

Poznań 2016

Spis treści

1. Wstęp.....	7
1.1 Cel i zakres pracy.....	8
1.2 Struktura pracy	8
1.3 Podział pracy	9
2. Wprowadzenie	11
2.1 Idea steganografii.....	11
2.2 Rys historyczny	11
2.3 Nośniki informacji umożliwiające zaimplementowanie steganografii	12
2.4 Narzędzia komunikacji w Internecie	12
2.4.1 Czat internetowy	12
2.4.2 Forum internetowe	13
2.5 Steganografia w plikach graficznych.....	13
3. Użyte technologie.....	15
3.1 Git 15	
3.2 JavaScript.....	15
3.3 Node.JS.....	15
3.3.1 Platforma Node.JS	15
3.3.2 Node Package Manager (NPM)	16
3.4 MongoDB	17
3.5 MeteorJS.....	17
3.5.1 Ogólny opis.....	17
3.5.2 Struktura plików.....	17
3.5.3 Modele i metody	18
3.6 AngularJS (wersja 1.x)	19
3.6.1 Ogólny opis.....	19
3.6.2 Jednostki	19
3.6.3 Two-way data binding	21
3.6.4 Angular-Meteor.....	21
3.7 Bower	22
3.8 Preprocessor.....	22
3.8.1 CoffeeScript	22
3.8.2 Jade	23
3.8.3 Less	23
3.9 Narzędzia do zarządzania projektem	24
3.9.1 TravisCI	24
3.10 Narzędzia do testowania	25
3.10.1 GulpJS.....	25
3.10.2 Jasmine.....	26
3.10.3 Karma.....	26
3.10.4 Protractor.....	27

4. Architektura aplikacji.....	29
4.1 Struktura plików i folderów	29
4.1.1 .gulp.....	29
4.1.2 .meteor.....	29
4.1.3 .stegano.....	29
4.1.4 client	31
4.1.5 lib.....	31
4.1.6 server	31
4.1.7 tests.....	32
4.1.8 Pliki w katalogu głównym	32
4.2 Modele w bazie danych.....	33
4.2.1 Post.....	33
4.2.2 Temat.....	34
4.2.3 Sekcja	35
4.2.4 Rola	36
4.2.5 Użytkownik	38
4.3 Sposób działania aplikacji.....	40
5. System uwierzytelniania i kontroli dostępu.....	41
5.1 Rejestrowanie i logowanie użytkowników.....	41
5.2 Role	41
5.3 Implementacja	42
5.4 Panel administracyjny	45
6. System ukrywania wiadomości.....	47
6.1 Treści zamieszczane na forum	47
6.2 Zabezpieczenia	48
6.3 Koncepcja widoków	48
6.4 Uzyskiwanie dostępu do modułu steganograficznego.....	49
6.5 Omówienie zaimplementowanych algorytmów steganograficznych	51
6.5.1 Dane wejściowe.....	51
6.5.2 Wyznaczanie wartości klucza steganograficznego	51
6.5.3 Kodowanie znaków tajnych wiadomości	51
6.5.4 Algorytm ukrywający wiadomość	52
6.5.5 Algorytm odkodowujący wiadomość	53
7. Testy.....	57
7.1 Testy jednostkowe.....	57
7.2 Testy integracyjne	59
7.3 Testy funkcjonalne	61
8. Podsumowanie i wnioski	63
9. Bibliografia.....	65

1. Wstęp

Wraz z wykształceniem się u człowieka umiejętności komunikacji powstał problem skutecznego przekazywania tajnych informacji, takich których zawartość mogłaby być znana tylko przez nadawcę i odbiorcę. Jak bezpiecznie przesłać list do adresata, tak aby pośrednik nie był w stanie go odczytać? Jak wysłać do króla wiadomość, której nikt inny nie byłby w stanie przechwycić? Jak przekazać polecenia dla szpiega, który znajduje się za linią frontu? Te i wiele innych pytań towarzyszy ludzkości praktycznie od powstania cywilizacji. Najczęstszym rozwiązaniem tych problemów jest szyfrowanie wiadomości przez nadawcę i odszyfrowanie jej za pomocą tajnego klucza przez odbiorcę. Dziedzinę nauki zajmującą się przekazywaniem informacji w sposób zabezpieczony przed niepowołanym dostępem nazywamy kryptologią. Dzieli się ona na kryptoografię, czyli gałąź wiedzy o utajnianiu wiadomości oraz kryptoanalizę, czyli gałąź wiedzy o przełamywaniu zabezpieczeń i deszyfrowaniu wiadomości. Przekazywanie informacji z pomocą kryptografii ma jednak jedną, ale zasadniczą wadę: obecność tajnego komunikatu jest jawna. Steganografia rozwiązuje ten problem. Jest to nauka o komunikacji w taki sposób, aby obecność komunikatu nie mogła zostać wykryta. Połączenie tych dwóch dziedzin nauki daje nam najwyższy możliwy poziom bezpieczeństwa w przekazywaniu tajnych wiadomości [Bateman 2008].

Na przestrzeni wieków zmieniały się narzędzia komunikacji. Zaczynając od prostej mowy, pisma, przechodząc przez radio, telewizję, a kończąc na Internecie człowiek zyskał bardzo wiele możliwości przesyłania dowolnej wiadomości (np. w formie pliku graficznego, filmu) na dowolną odległość. Dzięki temu implementacja steganografii w dwudziestym pierwszym wieku, jest teoretycznie ograniczona jedynie przez ludzką wyobraźnię.

Motywacją do podjęcia tematu była chęć stworzenia jak najbezpieczniejszego narzędzia do komunikacji w Internecie. Praktycznie każda większa korporacja oraz rządy państw nieustannie zbierają informacje na temat każdego użytkownika przeglądającego witryny WWW. Dzięki forum steganograficznemu przeciętny człowiek, bez większych umiejętności informatycznych, będzie w stanie wymieniać wiadomości, nie obawiając się o szpiegowanie przez strony trzecie. Alternatywą jest tzw. *Darknet* (pl. *Ciemny Internet*), czyli miejsce gdzie istnieje wolność absolutna, jednakże jest on trudny w obsłudze oraz dostępie.

1.1 Cel i zakres pracy

Celem niniejszej pracy jest stworzenie w pełni funkcjonalnego forum internetowego wraz z zaimplementowanym mechanizmem do ukrywania tajnych wiadomości w plikach graficznych. Aplikacja powinna umożliwiać rejestrację i logowanie użytkowników, tworzenie, edycję, przeglądanie i usuwanie sekcji, tematów oraz postów. Posty powinny należeć do tematów, a tematy do sekcji. Należy umożliwić wgranie pliku graficznego i przypisanie go do danego posta.

1.2 Struktura pracy

W rozdziale 2. Czytelnik znajdzie szersze wprowadzenie do tematu steganografii wraz z przykładami, jak była ona realizowana na przestrzeni wieków. Zawarto w nim również krótki opis możliwości wykorzystania technik steganograficznych do ukrywania informacji w plikach graficznych.

Spis narzędzi, technologii oraz języków programowania wykorzystanych do stworzenia aplikacji, o której traktuje ta praca znajduje się w rozdziale 3.

Rozwinięcie tematu zastosowanych technicznych rozwiązań stanowi rozdział 4., w którym opisana została architektura całej aplikacji. Jego lektura pozwoli zrozumieć organizację kodu aplikacyjnego w poszczególnych folderach. Przedstawione zostały także encje bazy danych wykorzystane w implementacji.

Rozdział 5. przedstawia szczegóły implementacji systemu uwierzytelniania użytkowników aplikacji i kontroli dostępu do zasobów oraz poszczególnych akcji możliwych do wykonania.

Wykorzystane techniki steganograficzne, ich implementację wraz z aspektami bezpieczeństwa przedstawia w detalach rozdział 6. Opisany został w nim także sposób uzyskiwania dostępu do funkcji steganograficznych, gdyż bezpośrednio nie ma do nich dostępu żaden użytkownik.

Rozdział 7. opisuje wraz z przykładami rodzaje testów, którym poddawane było opisywane forum w trakcie tworzenia.

Rozdział 8. stanowi podsumowanie pracy oraz formalne zakończenie. Po nim, na ostatnich stronach znajduje się także bibliografia.

1.3 Podział pracy

Efektywna praca w ramach grupy nad określonym projektem wymaga ustalenia jasnego podziału zadań i odpowiedzialności poszczególnych jej członków. Niniejsza praca nie jest wyjątkiem od tej zasady, dlatego jeszcze przed rozpoczęciem realizacji poszczególne zadania przypisano do autorów w sposób, który przedstawiony został w tabeli 1.

Tab. 1. Podział zadań pomiędzy autorów

Bartosz Kostaniak	Daniel Koza
Implementacja modułu steganograficznego Implementacja modułu do wgrywania plików graficznych Implementacja projektu graficznego aplikacji Zaprojektowanie struktur w bazie danych Testy integracyjne Testy jednostkowe	Implementacja systemu uwierzytelniania i autoryzacji użytkowników Testy jednostkowe Testy funkcjonalne Implementacja podstawowej funkcjonalności forum internetowego Zaprojektowanie warstwy wizualnej aplikacji

2. Wprowadzenie

2.1 Idea steganografii

Steganografia to nauka/zbiór technik umożliwiających ukrywanie wiadomości tajnej w innej, mało istotnej wiadomości, tzw. informacji nośnej. Ma to umożliwić bezpieczną wymianę informacji, jednak w odróżnieniu od kryptografii, która służy temu samemu celowi, niewidoczny jest sam fakt prowadzenia wymiany wrażliwych informacji [Garbaczuk, Kopniak 2005]. O ile bowiem dane takie zaszyfrowane za pomocą algorytmów kryptograficznych mogą być całkowicie nieczytelne dla atakującego, to jednak może on się łatwo domyślić, że dane komunikaty są w rzeczywistości interesującymi go informacjami, a do ich wydobywania konieczna jest jedynie odpowiednio duża moc obliczeniowa. Wzmacnianie zabezpieczeń kryptograficznych często zatem sprowadza się do zwiększania długości kluczy szyfrujących. W przypadku steganografii ktoś potencjalnie zainteresowany przechwyceniem informacji, do których nie powinien mieć dostępu nie jest świadomy, które komunikaty, w jaki sposób i czy w ogóle zawierają jakąś interesującą go treść. Jest to zatem pójdzie "o krok dalej" w zakresie ochrony wymiany danych.

2.2 Rys historyczny

Jako przykłady najstarszych wzmianek użycia steganografii podaje się zwykle [Bateman 2008] sytuacje opisywane przez Herodota z Halikarnasu w jego pismach z V wieku p.n.e. Pierwsza technika miała polegać na użyciu drewnianych tablic pokrywanych woskiem, które w jego czasach były powszechnie używane. Podczas normalnego użytkowania znaki nanoszone były na warstwę wosku, kiedy jednak wryto je w samym drewnie, które następnie pokryto woskiem, to tabliczka taka mogła ukrywać fakt prowadzenia komunikacji bez wzbudzania podejrzeń.

Kolejnym przykładem była historia Histiajosa, tyrana Miletu, który będąc w niewoli u perskiego władcy Dariusza chciał przekazać w bezpieczny sposób wiadomość swemu zięciowi Arystagorasowi. Wytatuował ją więc na ogolonej głowie posłańca, odczekał aż włosy odrosły, po czym posłał go w drogę z mało ważnym listem. Dzięki temu mógł on wyruszyć w drogę mimo pilnujących go strażników.

W czasie II wojny światowej niemiecki wywiad Abwehra dysponował technologią mikrokropek. Oznacza to, że możliwe było umieszczanie punktów o wielkości ok. 1 mm zawierających zdjęcia, szczegółowe plany techniczne czy zminiaturyzowany tekst. Taka mikrokropka może z powodzeniem zastąpić zwykłą kropkę nad literą "i" lub kończącą zdanie bez zauważalnej różnicy dla człowieka. W ten sposób szpiegzy niemieccy przesyłali zdobyte informacje dotyczące wojsk alianckich.

Współcześnie, dzięki rozwojowi techniki komputerowej, w zakresie sposobów na ukrycie jednej wiadomości wewnątrz innej głównym ograniczeniem jest wyobraźnia. Można sobie bowiem wyobrazić ukrywanie kolejnych bitów wiadomości jako na przykład standardowa lub nieco zwiększona/zmniejszona interlinia między wierszami tekstu, zastosowanie różnych rodzajów kropek w tekście czy inne, jeszcze wymyślniejsze metody.

2.3 Nośniki informacji umożliwiające zaimplementowanie steganografii

Tradycyjne rodzaje informacji nośnej zostały wymienione w poprzednim punkcie. Jednak wraz z rozwojem techniki obliczeniowej i ciągle zwiększającej się dostępności komputerów przed steganografią otworzyły się nowe możliwości. Oto bowiem każdy na swoim komputerze osobistym posiada dziś tysiące (jeśli nie miliony) różnego rodzaju plików. Mogą to być zdjęcia, pliki dźwiękowe, tekstowe czy inne. Praktycznie wszystkie stanowią potencjalne nośniki tajnych wiadomości, a dzięki wystarczającej mocy obliczeniowej komputerów możliwe jest ukrywanie takich wiadomości w szybki i niebudzący podejrzeń sposób.

2.4 Narzędzia komunikacji w Internecie

Jedną z głównych funkcji Internetu niemal od jego początków pozostaje funkcja komunikacyjna. Realizowana jest najczęściej na różnego rodzaju forach i czatach, do których dołączyć i w których aktywnie się udzielać może każdy człowiek. Ze względu na ich powszechność możliwe jest ich wykorzystanie do wymiany wrażliwych danych właśnie za pomocą technik steganograficznych w sposób niebudzący podejrzeń. Nawet jeśli ktoś jest świadomy, że takie usługi internetowe mogą służyć realizacji tajnej wymiany informacji, to w Internecie istnieją miliony forów i czatów, z których większość (jak możemy podejrzewać) nie stosuje opisywanych tutaj technik. Daje to dogodną okazję do ukrycia się w tłumie podobnych aplikacji.

2.4.1 Czat internetowy

Tradycyjny czat internetowy to aplikacja oferująca użytkownikom możliwość komunikacji za pomocą krótkich wiadomości tekstowych. Mogą być one wysyłane prywatnie (tylko do określonego użytkownika), tylko do określonej grupy użytkowników lub publicznie (widoczne dla wszystkich).

2.4.2 Forum internetowe

Fora internetowe służą do wymiany myśli, poglądów, doświadczeń użytkowników. Klasyczne fora umożliwiają wstawianie postów, tzn. wiadomości tekstowych z opcjonalnymi załącznikami, najczęściej plikami graficznymi. Posty widoczne są dla wszystkich zalogowanych użytkowników. Możliwość dołączania do swoich postów plików graficznych jest typowa (co oznacza w praktyce brak podejrzeń o celowość implementacji takiej funkcji) i to ona przesądziła o wyborze forum jako platformy, na której najłatwiej pokazać praktyczne zastosowanie steganografii.

2.5 Steganografia w plikach graficznych

Ukrywanie wiadomości w plikach graficznych jest najpopularniejszą metodą użycia steganografii. Jest tak dlatego, że nawet niewielki taki plik zawiera w sobie ogromną ilość informacji. Pojedynczy kolorowy obrazek zapisany z 32-bitową głębią koloru o wymiarach kilkaset na kilkaset pikseli (najbardziej typowych) może zajmować na nośniku nawet kilkaset kilobajtów pamięci. Jest to ilość wystarczająca do ukrycia w niej kilkunastu stron tekstu.

Z jednej strony mamy więc możliwość zawarcia w informacji nośnej jaką jest plik graficzny sporej ilości wrażliwych wiadomości, z drugiej strony im większa jest informacja nośna, a mniejsza tajna, tym mniejsze i w konsekwencji trudniejsze do wykrycia są modyfikacje oryginalnego pliku. Taka elastyczność stanowi mocną stronę stosowania steganografii właśnie z plikami graficznymi.

Najpopularniejszym sposobem ukrywania wiadomości w plikach graficznych jest jej zakodowanie na najmniej znaczących bitach kanałów koloru (RGB) poszczególnych pikseli. Jest to tzw. metoda najmniej znaczącego bitu (*ang. Least Significant Bit*) [Garbarczuk, Kopniak 2005]. Możliwe jest także umiejscowienie tajnej informacji np. w nagłówku pliku. W tej pracy zdecydowano się na wykorzystanie pierwszej metody, choć z pewnymi usprawnieniami. Problemem w przypadku tej metody okazuje się bowiem sytuacja, gdy ktoś postanawia ukryć wiadomość w pliku graficznym udostępnionym w Internecie za pomocą tej najpowszechniejszej metody. Porównując bowiem oryginał wraz z wersją zmodyfikowaną, ludzkie oko nie zauważy co prawda różnicy, lecz analiza komputerowa obu obrazów wskaże dokładnie, na których pikselach występują różnice koloru. Z występowania takiej różnicy można już wnioskować o obecności tajnej wiadomości, a także szacować jej długość czy wręcz ją zdekodować.

Szczegóły implementacji zastosowanego algorytmu ukrywającego wiadomości w plikach graficznych podane zostały w rozdziale 6.5.

3. Użyte technologie

3.1 Git

Git jest rozproszonym systemem kontroli wersji, stworzonym przez Linusa Torvaldsa jako narzędzie wspomagające rozwój jądra Linux. Umożliwia jednoczesną pracę nad projektem przez wiele osób, bez konieczności ciągłego dostępu do sieci. Każde zadanie do wykonania może być przypisane do innej gałęzi projektu (ang. *branch*), przy czym w danej chwili może istnieć wiele takich gałęzi. Dowolność łączenia poszczególnych gałęzi oraz opcjonalnej synchronizacji lokalnego repozytorium z centralnym umożliwia efektywne rozwijanie projektu. Użytkownik w dowolnym momencie może wykonać migawkę aktualnego stanu projektu (ang. *commit*), opisujący zmiany względem poprzedniej wersji. Dzięki możliwości odtworzenia stanu projektu opisanego przez dowolną migawkę, możliwe jest chronologiczne śledzenie wszystkich zmian, jakie zaszły od początku trwania projektu.

3.2 JavaScript

JavaScript jest skryptowym językiem programowania stosowanym najczęściej na stronach internetowych, jednakże dzięki różnym bibliotekom coraz częściej jest wykorzystywany także w innych środowiskach. Łączy w sobie następujące paradygmaty: obiektowy, funkcyjny oraz imperatywny. Jest to język dynamiczny i słabo typowany, czyli typy zmiennych są nadawane w czasie wykonywania skryptu, a konwersje na różne typy danych następują automatycznie. W języku tym występują obiekty, takie jak: *Object*, *Array* oraz typy prymitywne: *Boolean*, *Number*, *String*, *Null*, *Undefined*. Standardem JavaScript jest ECMAScript, czyli ustandaryzowany przez organizację ECMA skryptowy, obiektowy język programowania. Specyfikacja ta oznaczona jest jako ECMA-262 oraz ISO/IEC 16262.

3.3 Node.JS

3.3.1 Platforma Node.JS

Node.JS jest platformą umożliwiającą uruchamianie skryptów JavaScript w innym środowisku niż przeglądarka internetowa. Korzysta ona z silnika V8, z otwartym kodem źródłowym, który został stworzony i jest rozwijany przez firmę Google. Wykorzystuje go popularna przeglądarka internetowa Google Chrome. Node.JS korzysta między innymi z biblioteki *libuv*, która zapewnia asynchroniczne powiadamianie o zdarzeniach, jak i również z wielu innych bibliotek powszechnie dostępnych w Internecie, dzięki czemu Node.JS jest środowiskiem programistycznym

wykorzystywanym do tworzenia wysoce skalowalnych, sterowanych zdarzeniami i wykorzystujących asynchroniczny system wejścia-wyjścia aplikacji internetowych.

3.3.2 Node Package Manager (NPM)

W platformę Node.JS wbudowany jest system modułów nazywany Node Package Manager, w skrócie NPM. Umożliwia on doinstalowywanie zewnętrznych modułów do aplikacji stworzonej w środowisku Node.JS. Za pomocą jednej komendy programista jest w stanie ściągnąć konkretną wersję danej biblioteki. Dodatkowo jest możliwość zapisania listy używanych bibliotek do pliku *package.json*, który znajduje się w głównym katalogu projektu, dzięki czemu inny programista będzie mógł bez problemów ściągnąć wszystkie potrzebne zależności, które nie będą się znajdowały w zdalnym repozytorium. W pliku tym możliwe jest również zapisanie podstawowych informacji o projekcie, zdefiniowanie prostych skryptów oraz zarządzanie wersjami.

Listing 3.1. Przykładowy plik *package.json*

```
{
  "name": "angular-unit-testing-helpers",
  "version": "0.0.6",
  "description": "A collection of helper functions for writing
AngularJS unit tests.",
  "scripts": {
    "test": "./node_modules/karma/bin/karma start",
    "test-travis": "./node_modules/karma/bin/karma start --
single-run --browsers=Chrome_travis_ci,Firefox,PhantomJS"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/dakolech/angular-unit-
testing-helpers.git"
  },
  "author": "Daniel Koza",
  "license": "MIT",
  "homepage": "https://github.com/dakolech/angular-unit-
testing-helpers#readme",
  "devDependencies": {
    "angular": "1.5.0-rc.0",
    "angular-mocks": "^1.4.8",
    "jasmine-core": "^2.3.4",
    "karma": "^0.13.15",
    "karma-chrome-launcher": "^0.2.1",
    "karma-firefox-launcher": "^0.1.7",
    "karma-jasmine": "^0.3.6",
    "karma-ng-html2js-preprocessor": "^0.2.0",
    "karma-phantomjs2-launcher": "^0.4.0",
    "phantomjs": "^1.9.19"
  }
}
```

3.4 MongoDB

MongoDB jest otwartym, nierelacyjnym systemem zarządzania bazami danych. Charakteryzuje się dużą skalowalnością, wydajnością oraz dowolnością w strukturze tworzonych baz danych. Dane są składowane w postaci dokumentów JSON (*JavaScript Object Notation*), umożliwiając aplikacjom bardziej naturalne ich przetwarzanie, zwłaszcza przez biblioteki napisane w języku JavaScript. Dane składowane w bazie danych MongoDB tworzą tzw. dokumenty. Zbiór dokumentów opisujących podobne dane nazywa się kolekcją. Należy zaznaczyć przy tym, że struktura każdego dokumentu kolekcji może (choć nie musi) być taka sama, jak ma to miejsce w tradycyjnych, relacyjnych bazach danych.

3.5 MeteorJS

3.5.1 Ogólny opis

MeteorJS jest biblioteką wykorzystywaną do tworzenia aplikacji internetowych, jak i mobilnych. Została napisana z użyciem platformy Node.JS. Jest zintegrowana z nierelacyjnymi bazami danych MongoDB. Działa zarówno po stronie klienta (przeglądarka internetowa), jak i serwera. Wykorzystuje wzorzec projektowy publikuj-subskrybuj do automatycznego propagowania modyfikacji danych w stronę klienta, bez konieczności pisania kodu synchronizującego przez programistę. Dzięki temu stworzenie w pełni funkcjonalnej aplikacji internetowej wymaga dużo mniej pracy niż w konkurencyjnych bibliotekach, takich jak Ruby on Rails czy Django. Jedną z głównych zalet MeteorJS jest to, że cała komunikacja między przeglądarką internetową a serwerem, jest wykonywana za pomocą WebSocket'ów. WebSocket jest technologią zapewniającą dwukierunkowy kanał komunikacji za pośrednictwem jednego gniazda TCP.

MeteorJS posiada własny menadżer bibliotek i wtyczek, AtmosphereJS, który bardzo przypomina ten z Node.JS. Podobnie jak w NPM, programista może zainstalować zewnętrzną bibliotekę za pomocą jednego polecenia, a informacje o niej są zapisywane do pliku. Jednakże, w odróżnieniu od platformy Node.JS, nie są wymagane referencje do wtyczek w kodzie, albowiem wszystkie biblioteki są dostępne globalnie.

3.5.2 Struktura plików

MeteorJS posiada z góry określoną strukturę plików i kolejność ich ładowania, co ma duże znaczenie, zwłaszcza jeżeli aplikacja korzysta z dodatkowych bibliotek, takich jak na przykład AngularJS. W uproszczeniu: wszystko co znajduje się w folderze `.meteor` służy do konfiguracji wewnętrznych i zewnętrznych bibliotek oraz ich wersji, zawartość folderu `server` jest widoczna i wykonywana tylko po stronie serwera, podobnie jak katalogu `private`, pliki w folderze `client` oraz `public` natomiast są

wykonywane i widoczne tylko po stronie klienta, czyli w tym przypadku, przeglądarki internetowej, a skrypty w katalogu *tests* nie są widoczne ani przez serwer, ani przez klienta, służą tylko i wyłącznie do wykonywania testów. Kolejność ładowania plików jest następująca: wszystkie skrypty znajdujące się w katalogu *lib* ładowane są w pierwszej kolejności, następnie MeteorJS wczytuje resztę plików według kolejności alfabetycznej, najpierw z podkatalogu, a później z katalogu nadrzędnego, z wyjątkiem plików rozpoczynających się słowem *main*, które są ładowane na samym końcu.

3.5.3 Modele i metody

Dane w MeteorJS są przechowywane za pomocą kolekcji. Jako że biblioteka ta korzysta z MongoDB, struktura modeli w bazie danych nie jest określona. Kolekcje tworzy się raz i są dostępne globalnie.

Listing 3.2. Przykład inicjalizacji kolekcji Messages dostępnej globalnie.

```
Messages = new Mongo.Collection("messages");
```

W MeteorJS używany jest obiekt globalny *Meteor*, który posiada metodę *publish* oraz *methods*. *Publish* służy do wyświetlania danych bezpośrednio z MongoDB. Każda aktualizacja modelu lub kolekcji w bazie danych jest od razu widoczna w przeglądarce internetowej, bez żadnego przeładowywania strony, dzięki zastosowaniu WebSocket'ów. Oznacza, to że dane publikowane za pomocą tej metody są bezpośrednio połączone z danymi wyświetlanymi w widoku. Nie jest jednakże możliwa ich bezpośrednia modyfikacja za pomocą tej metody. *Methods* natomiast służy do definiowania akcji modyfikujących dane. Zarówno do metody *publish*, jak i *methods*, przesyłana zostaje funkcja, która, w pierwszym wypadku musi zwrócić wektor z bazy danych, a w drugim może zwrócić dowolną wartość.

Listing 3.3. Przykład metody publish, która zwraca wszystkie obiekty z kolekcji Rooms

```
Meteor.publish("rooms", function () {  
  return Rooms.find({});  
});
```

Listing 3.4. Przykład użycia methods. Podany kod tworzy dwie metody: foo oraz bar.

```
Meteor.methods({  
  foo: function (arg1, arg2) {  
    check(arg1, String);  
    return "some return value";  
  },  
  
  bar: function () {  
    return "baz";  
  }  
});
```


3.6 AngularJS (wersja 1.x)

3.6.1 Ogólny opis

AngularJS w wersji 1.x jest jedną z najbardziej popularnych bibliotek, napisanych w języku JavaScript, do tworzenia SPA (*Single Page Application*), czyli stron internetowych, które bardzo przypominają aplikacje działające natywnie w środowisku Windows, Linux czy Mac OSX. Wszelkie akcje, jakie użytkownik wykonuje, dzieją się w rzeczywistości tylko na jednej stronie, bez żadnego przeładowywania, a wszystkie dane pobierane i wysyłane do serwera są wykonywane za pomocą technologii AJAX (*Asynchronous JavaScript and XML*). Użytkownik ma wrażenie, że użytkuje normalną stronę, między innymi z powodu zmieniających się widoków i adresów URL w przeglądarce internetowej. AngularJS korzysta z popularnego wzorca projektowego MVC (*Model View Controller*), delikatnie go modyfikując. AngularJS jest stosunkowo młodą biblioteką, albowiem został opublikowany w 2012 roku przez firmę Google. Dzięki wsparciu tej korporacji stał się wiodącą technologią w dziedzinie tworzenia nowoczesnych aplikacji internetowych. W 2016 roku ma zostać wydana wersja 2.0, która w bardzo dużym stopniu różni się od aktualnej edycji. Dlatego tak ważne jest podanie, której wersji programista używa.

3.6.2 Jednostki

AngularJS opiera się na modułach. Są to podstawowe jednostki w tej bibliotece, dzięki którym organizujemy całą aplikację. Do każdego modułu możemy wstrzyknąć inny moduł, przez co tworzone jest drzewo zależności oraz zostaje usunięty problem globalnego definiowania obiektów. W bardzo dużym stopniu ułatwia to testowanie jednostkowe, ponieważ do testu wstrzykiwany jest tylko jeden moduł. Wszystkie jednostki opisane poniżej tworzone są właśnie w modułach.

Listing 3.5. Przykład inicjalizacji modułu myApp wraz ze wstrzykniętym modułem restangular

```
var myApp = angular.module('myApp', ['restangular']);
```

Rolę modelu we wzorcu MVC pełnią serwisy, takie jak: *service*, *factory*, *provider*. Mają one dwie podstawowe cechy: AngularJS inicjalizuje je wtedy i tylko wtedy, gdy jakiś komponent od nich zależy, a także korzystają ze wzorca *singleton*, czyli każdy komponent otrzymuje tę samą, pojedynczą referencję do danego serwisu. Jednostki te odpowiadają głównie za logikę biznesową aplikacji oraz za komunikację z serwerem. Do serwisu można wstrzyknąć inny serwis.

Listing 3.6. Przykład serwisu w AngularJS

```
angular.
module('myServiceModule', []).
factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
        msgs.push(msg);
        if (msgs.length == 3) {
            win.alert(msgs.join("\n"));
            msgs = [];
        }
    };
}]);
```

Jedną z podstawowych jednostek w AngularJS są kontrolery, czyli litera C we wzorcu MVC. Odpowiadają one za zarządzanie danymi w widoku oraz za wykorzystywanie serwisów. Są łącznikiem między warstwą logiki a warstwą prezentacji aplikacji. W przeciwieństwie do serwisów, które są inicjalizowane tylko raz w trakcie używania aplikacji, kontrolery są tworzone za każdym razem, gdy dany widok od nich zależy i niszczone, gdy są już nieużywane. Ma to ogromne znaczenie w procesie projektowania aplikacji. Kontrolery powinny zawierać jak najmniej kodu i nie zarządzać logiką biznesową, ze względu na to, że nie przechowują danych (są niszczone przy każdej zmianie widoku) i nie da się ich wstrzyknąć do innego komponentu.

Listing 3.7. Przykład kontrolera ze wstrzykniętym serwisem \$scope w AngularJS

```
myApp.controller('DoubleController', ['$scope',
function($scope) {
    $scope.double = function(value) {
        return value * 2;
    };
}])
```

Esencją AngularJS są dyrektywy. Dodają one dodatkowe elementy lub atrybuty do widoku, rozszerzając dość ograniczone możliwości dokumentów HTML. AngularJS posiada 69 wbudowanych dyrektyw (w wersji 1.4), które zaczynają się od liter ng, na przykład: *ng-click="funkcja()"*, *ng-if="false"* czy *ng-controller="DoubleController"*. Pierwsza służy do przypisania podanej funkcji do zdarzenia kliknięcia na daną część strony, druga usuwa element z dokumentu HTML, gdy wyrażenie jest nieprawdziwe, a ostatnia przypisuje podany kontroler do widoku. Biblioteka umożliwia także tworzenie nowych dyrektyw. Najprostszy przykład znajduje się w listingach 3.8-3.10. Oczywiście jest możliwość tworzenia bardziej skomplikowanych elementów, które będą przyjmować różne dane wejściowe, posiadać swój własny kontroler, czy korzystać z serwisów.

Listing 3.8. Przykład dyrektywy w AngularJS

```
myApp.directive('redName', function() {  
  return {  
    restrict: 'AE',  
    template: '<div class="red"><span> My name </span></div>'  
  };  
});
```

Listing 3.9. Przykład użycia dyrektywy z listingu 3.8 w dokumencie HTML

```
<red-name></red-name>  
<div red-name></div>
```

Listing 3.10. Wygenerowany dokument HTML z listingu 3.9 w przeglądarce internetowej

```
<div class="red">  
  <span>  
    My name  
  </span>  
</div>  
<div class="red">  
  <span>  
    My name  
  </span>  
</div>
```

3.6.3 Two-way data binding

Jedną z najważniejszych cech, jeśli nie najważniejszą, jest tak zwane dwukierunkowe wiązanie danych (ang. *two-way data binding*). Wszystkie dane przypisane do serwisu *\$scope*, są bezpośrednio powiązane z danymi występującymi w dokumencie HTML. Oznacza to, że każda zmiana modelu, jest od razu widoczna w widoku, jak i każda aktualizacja danej, na przykład w formularzu, jest od razu dostępna w serwisie *\$scope*. AngularJS sprawdza zmiany w modelach przez porównanie wartości z wartościami zgromadzonymi we wcześniejszym procesie, co jest nazywane pętlą *digest* (ang. *digest loop*). Cała aplikacja jest podzielona na wiele zakresów (ang. *scope*): główny *\$rootScope*, dostępny na całej stronie, oraz jego potomków. Pętla *digest* rozpoczyna się od najmłodszego potomka i aktualizuje wszystkie wartości z poszczególnych zakresów, przechodząc przez rodziców, a kończąc na *\$rootScope*.

3.6.4 Angular-Meteor

Angular-Meteor jest to moduł do biblioteki AngularJS umożliwiający jego współpracę z platformą MeteorJS. Dodaje on między innymi serwis *\$meteor*, który ułatwia komunikację z serwerem. Dzięki niemu w aplikacji zostaje wykorzystane nawet trójkierunkowe wiązanie danych (ang. *three-way data binding*), dane z bazy danych są

bezpośrednio połączone z danymi w serwisach AngularJS, jak i w dokumencie HTML, przez co każda aktualizacja obiektów na serwerze jest od razu widoczna w przeglądarce internetowej. Moduł ten dodaje także wiele innych funkcji, takich jak narzędzia uwierzytelniające współpracujące z wtyczką *Accounts* do MeteorJS, czy obsługę kolekcji z MongoDB i metod z obiektu globalnego *Meteor*.

3.7 Bower

Bower jest narzędziem podobnym do NPM, służy głównie do zarządzania bibliotekami używanymi po stronie klienta. Pliki źródłowe ściągane są z zdalnego repozytorium i zapisywane do katalogu *bower_components*. Konfiguracja narzędzia Bower znajduje się w dokumencie *bower.json*.

3.8 Preprocessor

3.8.1 CoffeeScript

CoffeeScript jest jednym z pierwszych preprocesorów jaki powstał do języka JavaScript. Dodaje on tak zwany lukier składniowy, aby zwiększyć czytelność i skrócić objętość kodu. Jego twórcy w dużym stopniu inspirowali się językiem Ruby oraz Python. W CoffeeScript słowo kluczowe *function* zostało zastąpione *->* lub *=>*, które dodatkowo wstrzykuje do funkcji obiekt *this* z wyższego bloku kodu. Nie są wymagane nawiasy okrągłe, ani nawiasy klamrowe. Podobnie jak w Python'ie, ciało funkcji jest wyznaczane przez wcięcie w kodzie. Nie jest potrzebne pisanie słowa kluczowego *return*, gdyż, podobnie jak w Ruby, funkcja zwraca ostatnie wyrażenie w jej ciele. Program napisany w CoffeeScript jest kompilowany do JavaScript.

Listing 3.11. Przykład kodu w JavaScript

```
function listen (el, event, handler) {
  if (el.addEventListener) {
    return el.addEventListener(event, handler);
  } else {
    return el.attachEvent("on" + event, function() {
      return handler.call(el);
    });
  }
}
```

Listing 3.12. Kod z listingu 3.11 napisany w CoffeeScript

```
listen = (el, event, handler) ->
  if el.addEventListener
    el.addEventListener event, handler
  else
    el.attachEvent 'on' + event, ->
      handler.call el
```

3.8.2 Jade

Jade jest preprocesorem dla dokumentów HTML. Jego składnia cechuje się dużą prostotą w stosunku do popularnego języka znaczników używanego głównie przez przeglądarki internetowe. Klasy oraz identyfikatory przypisuje się tak jak w CSS (zaczynając od kropki lub znaku #), wcięcia oznaczają potomne elementy, atrybuty znajdują się w nawiasach okrągłych, a tagi nie posiadają znaków < oraz >. Szablon Jade jest kompilowany do czystego HTML.

Listing 3.13. Przykład kodu w HTML

```
<!doctype html>
<html>

<head>
  <link type="text/css" rel="stylesheet" href="/site.css" />
  <title>Hello</title>
</head>

<body>
  <h1>Hello world!</h1>
</body>

</html>
```

Listing 3.14. Kod z listingu 3.13 napisany w Jade

```
doctype html
html
  head
    link(type='text/css', rel='stylesheet', href='/site.css')
    title Hello
  body
    h1 Hello world!
```

3.8.3 Less

Less jest preprocesorem do kaskadowych arkuszy stylów (CSS). Stara się on wprowadzić elementy z normalnych języków programowania, takie jak na przykład zmienne, funkcje czy pętle. Jedną z jego najważniejszych cech jest zagnieżdżanie reguł CSS. Less umożliwia także importowanie stylów z różnych plików, dzięki czemu programista może zdefiniować kolejność wczytywania arkuszy. Less jest kompilowany do CSS.

Listing 3.15. Przykład kodu w CSS

```
.box {
  color: #fe33ac;
  border-color: #fdcdea;
}
.box div {
  -webkit-box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
}
```

Listing 3.16. Kod z listingu 12 napisany w Less

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
  -webkit-box-shadow: @style @c;
  box-shadow: @style @c;
}
.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha)) {
  .box-shadow(@style, rgba(0, 0, 0, @alpha));
}
.box {
  color: saturate(@base, 5%);
  border-color: lighten(@base, 30%);
  div { .box-shadow(0 0 5px, 30%) }
}
```

3.9 Narzędzia do zarządzania projektem

3.9.1 TravisCI

TravisCI jest narzędziem wykorzystywanym do ciągłej integracji. Każdy projekt dostępny na platformie *Github.com* ma możliwość dodania pliku *.travis.yml*, który odpowiada za konfigurację. Podstawową funkcją TravisCI jest możliwość budowania oraz testowania aplikacji w zamkniętym środowisku i przesyłania informacji o statusie na skrzynkę pocztową właściciela projektu. Takie zadania mogą być wykonywane dla wybranych gałęzi oraz Pull Request'ów. Przy odpowiedniej konfiguracji narzędzie to jest w stanie wysłać kod źródłowy na serwer produkcyjny w przypadku pozytywnie zakończonych wszystkich poleceń. TravisCI jest wykorzystywany przez wiele projektów z otwartym kodem źródłowym, ponieważ jest płatny tylko dla prywatnych repozytoriów.

Listing 3.17. Przykładowy plik .travis.yml

```
language: node_js
node_js:
  - '5'
script:
  - npm run-script test-travis
before_script:
  - npm install
```

3.10 Narzędzia do testowania

3.10.1 GulpJS

GulpJS jest systemem automatyzacji pracy dostępny na platformie Node.JS. Za pomocą tego narzędzia programista jest w stanie zdefiniować często wykonywane przez niego zadania na plikach, takie jak kompilacja plików LESS, CoffeScript, kopiowanie, usuwanie, wykonywanie testów czy sprawdzanie składni w kodzie. Cała konfiguracja znajduje się w pliku *gulpfile.js*.

Listing 3.18. Przykładowy plik gulpfile.js

```
var gulp = require('gulp');
var sass = require('node-sass');
var rename = require('gulp-rename');
var replace = require('gulp-replace');
var rimraf = require('gulp-rimraf');

gulp.task('build', ['images'], function() {
  var css = sass.renderSync({
    file: 'template/scss/styles.scss'
  });
  gulp.src('template/screenshot.jpg').pipe(gulp.src('build'));
  return gulp.src('template/responsive.html')
    .pipe(replace('app.css', css))
    .pipe(rename('index.html'))
    .pipe(gulp.dest('build'));
});
gulp.task('clean', function() {
  return gulp.src('build', {
    read: false
  }).pipe(rimraf());
});
```

3.10.2 Jasmine

Jasmine jest biblioteką wspomagającą testowanie, dostępną na platformie Node.JS. Testy są wykonywane w funkcjach *it*, które można grupować w blokach *describe*. Istnieją także funkcje *beforeEach* oraz *afterEach*, wykonywane odpowiednio przed i po każdym teście. Walidacja poprawności wykonania danego testu odbywa się w funkcji *expect*, która posiada takie atrybuty jak *toBe*, *toEqual*, oznaczające, że wyrażenie podane jako argument w *expect* ma być takie samo jak podany obiekt. W Jasmine istnieją także specjalne obiekty *spyOn* (pl. szpieguj na), które sprawdzają czy dana metoda została wywołana i z jakimi argumentami. Jest możliwość także nadpisania istniejących funkcji, tak aby zachowywały się tak, jak program tego oczekuje.

Listing 3.19. Przykładowe testy w Jasmine

```
describe("Hello world", function() {
  it("says world", function() {
    expect(helloWorld()).toContain("world");
  });
});

describe("Person", function() {
  it("calls the sayHello() function", function() {
    var fakePerson = new Person();
    spyOn(fakePerson, "sayHello");
    fakePerson.helloSomeone("world");
    expect(fakePerson.sayHello).toHaveBeenCalled();
  });
});
```

3.10.3 Karma

Karma umożliwia automatyczne wykonywanie testów na różnych przeglądarkach internetowych, takich jak Firefox, Chrome, czy PhantomJS (narzędzie zastępujące prawdziwą przeglądarkę). Biblioteka ta została stworzona przez twórców AngularJS. Posiada wiele wtyczek, dzięki którym można na przykład obliczyć pokrycie kodu testami w aplikacji, czyli sprawdzić jaka część aplikacji posiada odpowiadający jej kod testujący. Plikiem odpowiadającym za konfigurację narzędzia jest najczęściej *karma.config.js*.

Listing 3.20. Przykładowy plik karma.config.js

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [
      'node_modules/angular/angular.js',
      'node_modules/angular-mocks/angular-mocks.js',
      'test-helpers.js',
      'test/**/*.js',
      'examples/**/*.js',
      'examples/**/*.html'
    ],
    preprocessors: {
      'examples/**/*.html': ['ng-html2js']
    },
    reporters: ['progress'],
    port: 9876,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['PhantomJS2'],
    singleRun: false
  })
}
```

3.10.4 Protractor

Protractor jest narzędziem wykorzystywanym do wykonywania testów *end-to-end* (funkcjonalnych) dla aplikacji napisanej w AngularJS. Biblioteka ta wykonuje swoje zadania na stronie internetowej, naśladując zachowanie użytkownika w przeglądarce. Wykorzystuje do tego oprogramowanie *Selenium*. Wszelka konfiguracja znajduje się najczęściej w pliku *protractor.config.js*. Protractor, podobnie jak Karma może używać biblioteki Jasmine do tworzenia testów.

Listing 3.21. Przykładowy test w protractor

```
'use strict';
describe('Main scenario', function() {
  it('should load the main page', function() {
    browser.get('/');

    expect(browser.isElementPresent(By.css('body'))).toBe(true);
  });
});
```

Listing 3.22. Przykładowy plik protractor.config.js

```
exports.config = {
  jasmineNodeOpts: {
    showColors: true,
    defaultTimeoutInterval: 300000
  },
  specs: ['**/*.scenario.js'],
  seleniumArgs: ['-browserTimeout=60'],
  seleniumAddress: 'http://localhost:4444/wd/hub',
  multiCapabilities: [{
    'browserName': 'chrome'
  }]
};
```

4. Architektura aplikacji

4.1 Struktura plików i folderów

4.1.1 .gulp

Niestety MeteorJS nie współpracuje z NPM (w wersji 1.2.1), dlatego żeby użyć GulpJS, potrzebny jest ukryty katalog *.gulp* (każdy plik lub katalog zaczynający się kropką jest niewidoczny dla aplikacji). Znajdują się w nim pliki *.bowerrc*, *bower.json* oraz *packages.json* służące do zarządzania zewnętrznymi bibliotekami. Za pomocą *gulpfile.coffee* skonfigurowane zostało narzędzie GulpJS, używane do generowania algorytmu steganograficznego (rozdział 6.), jak i do startowania testów jednostkowych oraz funkcjonalnych (włączenie Selenium oraz Protractor).

4.1.2 .meteor

Folder *.meteor* jest generowany automatycznie podczas tworzenia nowego projektu w MeteorJS. Wszelkie pliki znajdujące się w tym katalogu powstają i są aktualizowane automatycznie, bez ingerencji programisty. W *.id* znajduje się unikalny identyfikator dla aplikacji. Wszystkie używane wtyczki lub biblioteki zapisywane są do dokumentu *packages*, a numery ich wersji oraz zależności znajdują się w *versions*. *platforms* odpowiada za platformy docelowe aplikacji, w tym przypadku jest to *browser* (przeglądarka internetowa) oraz *server* (serwer). W pliku *release* znajduje się numer wersji samej platformy MeteorJS.

4.1.3 .stegano

Folder ten zawiera kod modułu steganograficznego w oryginalnej postaci, to znaczy przed poddaniem go minifikacji i umieszczeniu w kodzie serwera aplikacji. Moduł ten składa się z kilkunastu plików z kodem JavaScript umieszczonych w podkatalogach *js/algorithm/*, *js/events/*, *js/helpers/*, *js/image/*, *js/integration* oraz *js/lib*. Zostaną teraz opisane zadania, za które odpowiada kod znajdujący w poszczególnych podkatalogach. Kluczowe dla aplikacji są 3 pliki w podkatalogu *js/algorithm*, ponieważ znajdują się w nich kody 3 algorytmów:

- algorytmu służącego do ukrywania wiadomości w podanym pliku graficznym za pomocą określonego klucza steganograficznego
- algorytmu pozwalającego na wydobywanie tajnej wiadomości z określonego pliku graficznego za pomocą określonego klucza steganograficznego

- algorytmu *xorshift1024**, będącego generatorem liczb pseudolosowych posiadającym 1024 bity stanu oraz maksymalny okres wynoszący $2^{1024} - 1$

Z kolei w podkatalogu *js/events* znajduje się tylko 1 plik: *common.js*, w którym zdefiniowane są funkcje odpowiedzialne za wysyłanie wiadomości z ukrytą treścią oraz za przełączanie widoku tematu ze zwykłego na “tajny” oraz na odwrót.

Funkcje pomocnicze, np. funkcja pobierająca dane o pliku graficznym wybranym przez użytkownika jako informacja nośna, zdefiniowane są w *js/helpers*.

Podkatalog *js/image* zawiera kod funkcji ładujących określony plik graficzny na element *canvas*, będący częścią specyfikacji języka znaczników HTML5 oraz funkcje umożliwiające pobieranie/ustawianie koloru określonego piksela obrazu załadowanego na element *canvas*. Są one wykorzystywane bezpośrednio przez algorytmy: ukrywający oraz wydobywający tajne informacje z obrazów zamieszczonych na forum.

W celu zintegrowania modułu steganograficznego z aplikacją konieczne było napisanie funkcji wzbogacających automatycznie oryginalny widok tworzenia nowego posta na forum o dodatkowy element *input* języka HTML, pozwalający na podanie przez użytkownika treści tajnej wiadomości, element *canvas* umożliwiający manipulacje kolorem na poziomie pojedynczych pikseli załadowanego na nim obrazu, oraz przełącznik (*ang. switch*), czyli element interfejsu służący użytkownikowi do przełączania się pomiędzy dwoma trybami widoku listy postów danego tematu:

- widok normalny – użytkownik widzi oryginalną treść postów oraz dołączone do nich ewentualnie pliki graficzne
- widok tajny – użytkownik widzi jedynie treści będące w widoku normalnym ukryte w plikach graficznych; nie widzi natomiast oryginalnych treści postów oraz samych obrazów

Tworzeniem oraz obsługą wyżej wymienionych elementów zajmują się skrypty umieszczone w podkatalogu *js/integration*.

Ostatnimi skryptami są te zlokalizowane w podkatalogu *js/lib*. Pierwszy z nich, *long.min.js* jest wstępnie zminifikowaną wersją biblioteki *long.js*, pozwalającej na operowanie liczbami 64-bitowymi w JavaScriptcie – funkcjonalność ta okazała się konieczna przy implementowaniu generatora liczb pseudolosowych *xorshift1024**. Drugim jest *main.js*, odpowiadający za załadowanie opisanych powyżej skryptów w poprawnej kolejności.

4.1.4 client

W katalogu *client* znajduje się cała aplikacja klienta napisana w AngularJS. Każda nazwa pliku *.coffee*, opisuje jego zawartość, i tak skrypty z końcówką *module.coffee* oznaczają definicję modułu, *directive.coffee* – dyrektywy, *service.coffee* – serwisu, a *controller.coffee* – kontrolera. Pliki z widokami posiadają końcówkę *ng.jade*. Jest to spowodowane wymaganiami wtyczki do zarządzania szablonami Jade dla AngularJS. Główny dokument *index.jade* nie potrzebuje *ng* w nazwie. Ze względu na brak możliwości ustawienia kolejności ładowania skryptów w MeteorJS, wszystkie definicje modułów muszą się znajdować w folderach *lib*. Jako, że w tych katalogach może się znajdować więcej plików (na przykład skrypty konfiguracyjne), to nazwy dokumentu z modułem zaczynają się literą *a* (lub *an* w przypadku gdy nazwa rozpoczyna się samogłoską). Dzięki temu zabiegowi, AngularJS bez problemów wczytuje wszystkie moduły w odpowiedniej kolejności. Architektura aplikacji klienckiej jest zorganizowana w sposób modułowy, to znaczy, że każdy moduł posiada osobny katalog. Ułatwia to w znacznym stopniu zidentyfikowanie zastosowania danego pliku.

4.1.5 lib

W folderze tym znajdują się między innymi definicje kolekcji, dzięki czemu są one dostępne globalnie. Plik *filters.coffee* zawiera inicjalizację globalnego obiektu *Filter*, który służy do filtrowania danych pobieranych z MongoDB.

4.1.6 server

Katalog ten odpowiada za część serwerową aplikacji i podobnie jak część kliencka, jest podzielony na moduły. W każdym module możemy znaleźć pliki z końcówką *methods.coffee* oznaczające skrypty z metodami obiektu Meteor, *publish.coffee* – funkcje publikujące dane z MongoDB, *seed.coffee* – kod do tworzenia przykładowych obiektów w bazie danych oraz *functions.coffee* – funkcje używane w innych częściach części serwerowej. Plik *app.coffee* jest głównym skryptem inicjalizującym. Ze względu na to, że MeteorJS w wersji 1.2.1 nie wspiera eksportowania i importowania obiektów z innych plików (wszystko jest globalne), to zaimplementowany został prosty system *require/module.exports*, podobny do tego używanego na platformie Node.JS. Dzięki temu uniknięto globalnego definiowania większości funkcji.

4.1.7 tests

Folder ten zawiera pliki wykonujące testy. W katalogu *unit* znajdują się skrypty do wykonywania testów jednostkowych aplikacji klienckiej, posiadają one końcówkę *spec.coffee*. W *e2e* mieszczą się scenariusze do testów funkcjonalnych, posiadają w nazwie słowo kluczowe *scenario*. Folder *jasmine/server* zawiera skrypty wykonujące testy integracyjne serwera. *jasmie/client/integration* powstał tylko w celu uruchomienia bliźniaczej aplikacji do testów funkcjonalnych.

4.1.8 Pliki w katalogu głównym

.codeclimate.yml – plik konfiguracyjny dla platformy *CodeClimate.com*, która sprawdza między innymi pokrycie aplikacji testami oraz jakość kodu.

.gitignore – dokument, dzięki któremu możliwe jest ignorowanie wybranych folderów/plików przez repozytorium Git.

.node-version – określa wersję platformy Node.JS

.travis.yml – plik konfiguracyjny dla platformy TravisCI.

README.md – plik zawierający podstawowe informacje o projekcie.

build.variables.sh – prosty skrypt pomocny przy opisywaniu wersji aplikacji dostępnej na serwerze produkcyjnym/testowym.

deploy.exp – skrypt umożliwiający automatyczne wgranie aplikacji na serwer produkcyjny/testowy po pomyślnym przejściu testów na platformie TravisCI.

4.2 Modele w bazie danych

_id – unikalny identyfikator obiektu, wygenerowany automatycznie przez platformę MeteorJS. Posiada on postać skomplikowanego łańcucha znaków, na przykład: *wjQyQ6sGjzvNMDLiJ*

id – identyfikator obiektu unikalny w danej kolekcji, wygenerowany automatycznie. Posiada on postać liczby naturalnej, inkrementowanej przy tworzeniu nowego obiektu.

4.2.1 Post

Post	
<i>_id</i> :	string
<i>id</i> :	string
<i>topic_id</i> :	string
<i>image_id</i> :	string
<i>text</i> :	string
<i>createdAt</i> :	Date
<i>updatedAt</i> :	Date
<i>userId</i> :	string

Rys. 4.1. Schemat modelu posta w bazie danych

topic_id – identyfikator tematu, do którego należy post

image_id – identyfikator obrazka, który się znajduje w poście

text – zawartość postu

createdAt – data utworzenia postu

updatedAt – data zmodyfikowania postu

userId – identyfikator użytkownika, do który utworzył post

4.2.2 Temat

Topic	
<code>_id:</code>	string
<code>id:</code>	string
<code>section_id:</code>	string
<code>name:</code>	string
<code>description:</code>	string
<code>createdAt:</code>	Date
<code>updatedAt:</code>	Date
<code>userId:</code>	string

Rys. 4.2. Schemat modelu tematu w bazie danych

section_id – identyfikator sekcji, do której należy temat

name – nazwa tematu

description – opis tematu

createdAt – data utworzenia tematu

updatedAt – data zmodyfikowania tematu

userId – identyfikator użytkownika, do który utworzył temat

4.2.3 Sekcja

Section	
_id:	string
id:	string
name:	string
description:	string
createdAt:	Date
updatedAt:	Date

Rys 4.3. Schemat modelu sekcji w bazie danych

name – nazwa sekcji

description – opis sekcji

createdAt – data utworzenia sekcji

updatedAt – data zmodyfikowania sekcji

4.2.4 Rola

Role	
_id:	string
name:	string
default:	boolean
can:	object
create:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
read:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
update:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
remove:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean

Rys 4.4. Schemat modelu roli w bazie danych

name – unikalna nazwa roli

default – flaga oznaczająca, która rola będzie nadawana nowemu użytkownikowi, tylko jeden obiekt posiada to pole z wartością ustawioną na *true*

can – obiekt definiujący wszystkie prawa, jakie dana rola posiada. Dzieli się na cztery podstawowe czynności, które użytkownik jest w stanie wykonać na obiektach w bazie danych: *create* – stwórz, *read* – czytaj, *update* – modyfikuj oraz *remove* – usuń. *Post*, *topic*, *section*, *user* oznaczają kolekcje znajdujące się w aplikacji. Flaga *create.post* ustawiona na *true* daje uprawnienia do stworzenia postu. Obiekt *in* posiada dwie tablice zawierające identyfikatory (*topic* – tematów, *section* – sekcji), oznaczające w których częściach aplikacji użytkownik może wykonać daną czynność. Jeżeli w tablicy *read.in.topic* znajduje się 2 oznacza, że rola pozwala na czytanie, w tym przypadku postów, w temacie o identyfikatorze równym 2. Obiekt *self*, zawiera flagi wyznaczające czy dane czynności mogą być wykonywane przez autora danego postu, tematu lub sekcji. *update.self.post* równe *true* daje użytkownikowi możliwość modyfikowania swojego postu.

4.2.5 Użytkownik

User	
_id:	string
username:	string
password:	string
role:	string
emails:	array
:	object
address:	string
verified:	boolean
profile:	object
firstname:	string
lastname:	string
catname:	string
catbreed:	string
catyears:	string
dogname:	string
dogbreed:	string
dogyears:	string
can:	object
create:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
read:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
update:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean
remove:	object
post:	boolean
topic:	boolean
section:	boolean
user:	boolean
in:	object
topic:	array
section:	array
self:	object
post:	boolean
topic:	boolean
section:	boolean

Rys 4.5. Schemat modelu użytkownika w bazie danych

username – unikalna nazwa użytkownika

password – hasło użytkownika

role – nazwa roli, którą użytkownik posiada

email – kolekcja zawierające obiekty, które posiadają pole *address* – adres e-mail użytkownika oraz flagę *verified*, oznaczającą czy dany adres został zweryfikowany (nieużywana w aplikacji). W aplikacji użytkownik może posiadać tylko jeden adres e-mail.

profile – obiekt zawierający dodatkowe informacje o użytkowniku, takie jak: *firstname* – imię, *lastname* – nazwisko, *catname* – imie kota, *catbreed* – rasa kota, *catyears* – wiek kota, *dogname* – imię psa, *dogtbreed* – rasa psa, *dogyears* – wiek psa

can – obiekt odziedziczony po roli, którą dany użytkownik posiada

4.3 Sposób działania aplikacji

Opisywana aplikacja forum internetowego nie ujawnia w żaden sposób przeciętnemu użytkownikowi faktu, że może być wykorzystywana jako tajny kanał bezpiecznej komunikacji. Już na początku procesu projektowania podjęto decyzję, aby nie wyróżniała się ona od innych, dostępnych powszechnie w Internecie stron i aplikacji pod względem funkcjonalności oferowanej zwykłym użytkownikom. Implikacją tego założenia jest umożliwienie zarejestrowania się i korzystania z niej każdemu chętnemu. Oczywiście w takim modelu większość użytkowników nie jest świadoma prawdziwej intencji istnienia takiego forum i nigdy nie wykorzysta go do przekazywania wrażliwych danych, lecz dla pozostałych, świadomych możliwości aplikacji, stanowi to dobry kamuflaż. Dodając do tego nieformalną tematykę, np. forum dla właścicieli zwierząt domowych lub hobbystów wędkarstwa, uzyskuje się aplikację, której nie sposób podejrzewać o wykorzystywanie do celów np. szpiegowskich.

Dodatkowe funkcje realizowane za pomocą steganografii nie są domyślnie dostępne ze względów bezpieczeństwa. Aby móc z nich skorzystać należy wykonać pewną tajną, ściśle określoną sekwencję ruchów takich jak napisanie posta czy jest edycja. Takie rozwiązanie problemu dostępu do ukrytej funkcjonalności zostało zastosowane w niniejszej pracy, a jego szczegóły zostaną opisane w rozdziale 6. Poprawne wykonanie zdefiniowanej sekwencji skutkuje przesłaniem przez serwer do klienta kodu modułu steganograficznego, odpowiedzialnego za rozszerzenie możliwości działania całej aplikacji.

Wspomniany wcześniej moduł umożliwia ukrywanie wiadomości w dołączanych do postów plikach graficznych. Wykorzystywany jest algorytm nadpisywania wartości najmniej znaczących bitów koloru wybranych pikseli obrazu. Szczegóły działania oraz implementacji omówione zostaną w rozdziale 6.

5. System uwierzytelniania i kontroli dostępu

5.1 Rejestrowanie i logowanie użytkowników

System uwierzytelniania użytkowników został oparty na wtyczce do MeteorJS o nazwie *accounts-password* dostępnej na platformie AtmosphereJS. Bardzo dobrze współpracuje ona z biblioteką Angular-Meteor, dzięki czemu ilość kodu potrzebna do konfiguracji środowiska nie była duża.

Każdy użytkownik chcący się zarejestrować na forum musi podać w formularzu swój adres e-mail, unikalną nazwę użytkownika oraz hasło. Jeżeli wszystkie dane są poprawne, użytkownik od razu zostanie zalogowany do aplikacji, w przeciwnym wypadku wyświetlony zostanie błąd, opisujący problem. Wylogowywanie polega na naciśnięciu przycisku *Sign out*, znajdującego się w górnym menu. Logowanie odbywa się na podobnej zasadzie, co rejestracja: użytkownik musi podać poprawny adres e-mail lub nazwę użytkownika oraz odpowiadające im hasło. W przypadku poprawnych danych użytkownik zostanie pomyślnie zalogowany, w przeciwnym zostanie wyświetlony błąd.

Implementacja tego systemu polega na stworzeniu odpowiednich formularzy i napisania do nich logiki wykorzystującej metody serwisu *\$meteor*. Logika odpowiadająca za powyższe akcje znajduje się w pliku: *client/users/services/login.service.coffee*.

5.2 Role

Rozbudowana struktura ról (opisana w rozdziale 4.2.4) pozwala przyznawać szerokie spektrum uprawnień dla poszczególnych użytkowników. Dzięki zastosowaniu wartości boolowskich w atrybutach zarządzanie takimi obiektami jest dużo czytelniejsze w kodzie programu.

Obiekt *can* w strukturze użytkownika jest identyczny jak ten w roli. Jest on automatycznie przypisywany w trakcie tworzenia nowego konta lub przy edycji uprawnień danej osoby. Wynika to z faktu, że platforma MeteorJS wysyła do aplikacji klienckiej obiekt użytkownika i jest on dostępny od razu po zalogowaniu. Dodatkowe zapytanie o rolę nastąpiło by dopiero po tej czynności, czyli w trakcie kompilacji widoku, przez co obsługa kolejnego zapytania wiązałaby się z dodatkową pętlą *\$digest* w bibliotece AngularJS. Innym atutem współdzielenia obiektu *can* jest mniejsza ilość kodu po stronie aplikacji klienckiej.

5.3 Implementacja

Listing 5.1 Autoryzacja użytkownika w aplikacji serwerowej

```
checkAuth = (what, name, object, userId) ->
  if !userId
    throw new (Meteor.Error)('notLogged')
    return false

  user = Meteor.users.findOne({_id: userId})

  if user.can[what][name]
    return true

  if name == 'post'
    if object.topic_id in user.can[what].in.topic
      return true

    foundTopic = Topics.findOne({ id: object.topic_id })
    if !!foundTopic
      sectionId = foundTopic.section_id
      if sectionId in user.can[what].in.section
        return true

  if name == 'topic'
    if object.section_id in user.can[what].in.section
      return true

  if user.can[what].self[name] && userId == object.userId
    return true

  throw new (Meteor.Error)('notAuthorized')
  return false
```


Listing 5.2 Autoryzacja użytkownika w aplikacji klienckiej

```
can: (what, name, object = {}) ->
  user = $rootScope.currentUser
  sectionId = object.section_id || $stateParams.section_id
  topicId = object.topic_id || $stateParams.topic_id

  if !user
    if name == 'section' && what == 'read'
      return true
    else
      return false

  if !!what && !!name && user.can[what][name]
    return true

  if name == 'post'
    if !!topicId && topicId in user.can[what].in.topic
      return true
    if !!sectionId && sectionId in
user.can[what].in.section
      return true

  if name == 'topic'
    if !!sectionId && sectionId in
user.can[what].in.section
      return true

  if !!what && user.can[what].self[name] && user._id ==
object.userId
    return true

  return false
```

Zwrócenie wartości *true* oznacza przyznania uprawnień, *false* odmowę.

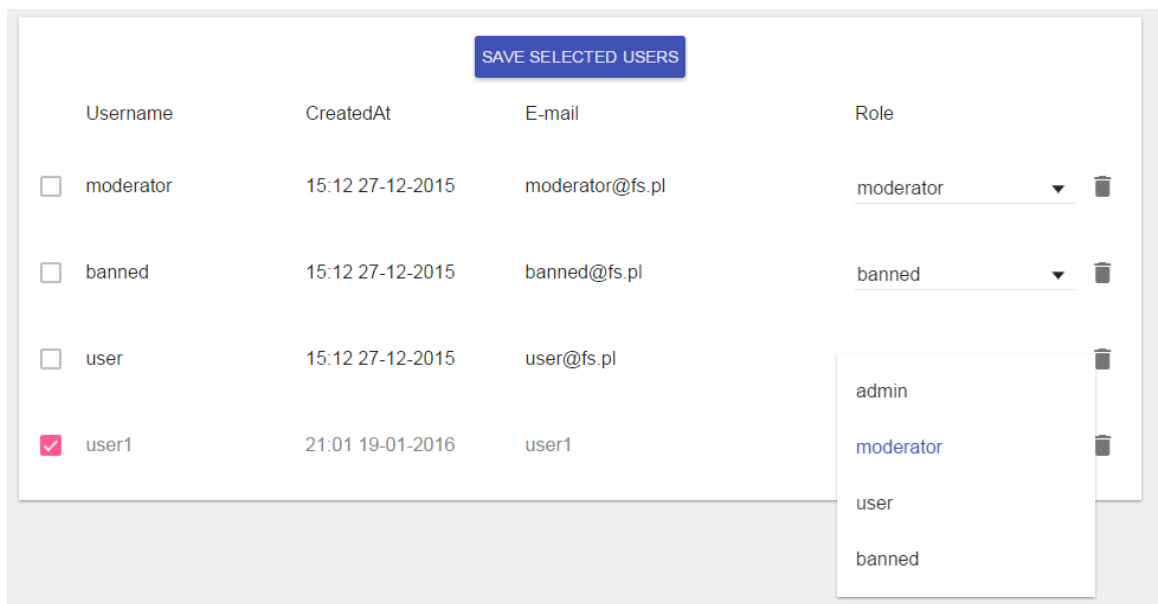
Kody źródłowe przedstawione w listingu 5.1 oraz w 5.2 są do siebie bardzo podobne z powodu współdzielenia struktury użytkownika przez całą aplikację. Argumentami obu funkcji są: *what* – czynność, którą chce wykonać użytkownik, *name* – nazwa obiektu, którego dostęp jest sprawdzany, *object* – właściwy obiekt. Aplikacja serwera posiada jeszcze *userId*, czyli identyfikator użytkownika, którego uprawnienia są sprawdzane. Jest on pobierany z bazy danych metodą *findOne* na obiekcie *Meteor.users*. Po stronie klienta użytkownik jest przechowywany w *\$rootScope.currentUser*. Dodatkowo w listingu 5.1 jest rzucaany jest wyjątek, gdy nie istnieje *userId*, co oznacza, że dana osoba nie została zalogowana. Zaraz po pobraniu użytkownika w listingu 5.2, pobierane są także identyfikatory sekcji i tematu, jeżeli takie istnieją. Informacje o nich przechowywane są w adresie URL i zarządzane przez serwis *\$stateParams*. Może się też tak zdarzyć, że obiekt, który jest sprawdzany, posiada w sobie zapisany identyfikator tematu lub sekcji.

Pierwszy warunek w aplikacji klienckiej sprawdza czy niezalogowany użytkownik chce wyświetlić listę sekcji, które są dostępne dla każdego. Jeżeli jest to inna czynność wtedy zwracaną wartością jest *false*. Drugi warunek, a pierwszy w aplikacji serwerowej, sprawdza czy dana osoba może wykonać daną akcję bezpośrednio. Następnie badane są uprawnienia dla tematu oraz sekcji, do których należy dany post. W przypadku listingu 5.1 należy pobrać odpowiedni temat z bazy danych, z którego można odczytać identyfikator sekcji, ponieważ post nie posiada w sobie o niej informacji. Kolejna część kodu sprawdza, czy użytkownik może wykonywać daną czynność w sekcji, jeżeli sprawdzanym obiektem jest temat. Ostatni warunek bada, czy dana osoba może wykonać akcję na obiekcie stworzonym przez nią samą. Jeżeli wcześniej nie zostanie zwrócona żadna wartość, wtedy funkcja wyrzuca błąd o braku autoryzacji w przypadku aplikacji serwerowej oraz zwraca *false* w obu przypadkach.

5.4 Panel administracyjny

Do zarządzania systemem autoryzacji powstał panel administracyjny znajdujący się w części klienckiej. Dostęp do niego posiada tylko i wyłącznie użytkownik, do którego przypisana jest rola *admin*. Generowana jest ona podczas pierwszego uruchomienia aplikacji i jest niemodyfikowalna z poziomu przeglądarki internetowej. Dzięki temu nie jest możliwe przypadkowe pozbawienie niektórych uprawnień dla tej roli.

Pierwszym widokiem przedstawionym na rysunku 5.1 jest lista wszystkich użytkowników zarejestrowanych na forum. Ze względów bezpieczeństwa nie jest wyświetlana aktualnie zalogowana osoba. Przycisk *SAVE SELECTED USERS* jest odblokowany tylko wtedy gdy zmodyfikowany został chociaż jeden użytkownik. Widok ten umożliwia zmianę roli danej osoby poprzez wybór opcji z elementu *select* oraz usunięcie użytkownika za pomocą kliknięcia w ikonę kosza na śmieci. Zmiany nie są zapisywane, dopóki administrator nie naciśnie przycisku *SAVE SELECTED USERS*.



Rys. 5.1. Widok listy użytkowników w panelu administracyjnym

Drugi widok przedstawiony na rysunku 5.2 służy do zarządzania rolami w aplikacji. Na samej górze znajduje się element *select* umożliwiający zmianę standardowej (ang. *default*) roli, która jest nadawana dla nowych użytkowników. Poniżej znajduje się lista istniejących ról w bazie danych wraz z ich atrybutami przedstawionymi w postaci *checkbox*’ów. Po prawej stronie umieszczone są ikonki kosza na śmieci służące do usuwania danych obiektów. *ADD NEW ROLE* dodaje nową rolę z odznaczonymi wszystkimi atrybutami. Przyciski *SAVE DEFAULT ROLE* oraz *SAVE SELECTED ROLES* są zablokowane, gdy nie jest zmodyfikowany żaden obiekt. Zmiany nie są zapisywane, dopóki administrator nie naciśnie jednego z przycisków *SAVE*.

Default role: user SAVE DEFAULT ROLE

ADD NEW ROLE SAVE SELECTED ROLES

Name	Create	Read	Update	Delete
<input checked="" type="checkbox"/> banned	Posts: <input type="checkbox"/> Topics: <input type="checkbox"/> Sections: <input type="checkbox"/> Self: <input type="checkbox"/>	Posts: <input type="checkbox"/> Topics: <input type="checkbox"/> Sections: <input checked="" type="checkbox"/> Self: <input type="checkbox"/>	Posts: <input type="checkbox"/> Topics: <input type="checkbox"/> Sections: <input type="checkbox"/> Self: <input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> moderat	Posts: <input checked="" type="checkbox"/> Topics: <input checked="" type="checkbox"/> Sections: <input checked="" type="checkbox"/> Self: <input checked="" type="checkbox"/>	Posts: <input checked="" type="checkbox"/> Topics: <input checked="" type="checkbox"/> Sections: <input checked="" type="checkbox"/> Self: <input checked="" type="checkbox"/>	Posts: <input checked="" type="checkbox"/> Topics: <input checked="" type="checkbox"/> Sections: <input checked="" type="checkbox"/> Self: <input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/> user	Posts: <input checked="" type="checkbox"/> Topics: <input checked="" type="checkbox"/> Sections: <input type="checkbox"/> Self: <input checked="" type="checkbox"/>	Posts: <input checked="" type="checkbox"/> Topics: <input checked="" type="checkbox"/> Sections: <input checked="" type="checkbox"/> Self: <input checked="" type="checkbox"/>	Posts: <input type="checkbox"/> Topics: <input type="checkbox"/> Sections: <input type="checkbox"/> Self: <input checked="" type="checkbox"/>	<input type="checkbox"/>

Rys. 5.2. Widok listy ról w panelu administracyjnym

6. System ukrywania wiadomości

6.1 Treści zamieszczane na forum

Przy projektowaniu aplikacji przyjęto założenie, że posty umieszczane przez użytkowników będą składać się z dwóch rodzajów treści:

- wiadomości tekstowej (opcjonalna)
- zamieszczonego pliku graficznego (maksymalnie 1 na post, opcjonalnie)

Oznacza to, że każdy post składa się z wiadomości tekstowej, obrazka lub obu naraz. Dodatkowo, użytkownik posiadający dostęp do modułu steganograficznego może zamieszczać na forum posty zawierające ukrytą wiadomość (wówczas do posta musi dołączyć plik graficzny, w którym będzie ukryta wiadomość oraz wypełnić pole formularza, którego zawartość będzie ukrywana w obrazku). Użytkownik taki zachowuje jednak możliwość pisania zwykłych postów, bez ukrytych wiadomości – wystarczy, że nie dołączy do posta pliku graficznego lub pozostawi puste pole formularza przeznaczone do wpisania tajnej wiadomości. Różnice w jego wyglądzie w zależności od tego, czy aktywowany został moduł steganograficzny, pokazują rysunki 6.1 oraz 6.2.

Add new post



Rys. 6.1. Formularz dodawania nowego posta na forum (brak dodatkowego pola i przełącznika)

Add new post



Rys. 6.2. Formularz dodawania nowego posta po rozszerzeniu go o dodatkowe pole i przełącznik przez moduł steganograficzny

6.2 Zabezpieczenia

Opisywane w niniejszej pracy forum internetowe służyć ma określonej grupie ludzi do wymiany wrażliwych informacji. Przyjmujemy założenie, że grupa ta jest w stanie ustalić między sobą tajne hasło, które używane będzie do ukrywania wiadomości w obrazkach. Stanowi ono główne zabezpieczenie mające zapewnić, że nikt niepowołany nie będzie w stanie odczytać wymienianych przez tę grupę informacji.

Dodatkowo, z każdym obrazem, w którym ukryto jakąś wiadomość jest skojarzona w bazie danych specjalna wartość 1024-bitowa, inna dla każdego pliku graficznego. W celu odczytania informacji ukrytej w obrazie jest ona łączona za pomocą operacji bitowej *xor* z opisywanym wyżej tajnym hasłem, a wynik tej operacji stanowi dopiero właściwy klucz używany w algorytmie odkodowującym wiadomość. Szczegóły tego procesu zostaną przedstawione w rozdziale 6. Należy zaznaczyć, że wartość ta nie musi być specjalnie chroniona, ponieważ jej znajomość nie mówi nic o wartości operacji *xor* przeprowadzonej na niej oraz na tajnym hasle, bit po bicie. Dlatego pełni ona rolę analogiczną do klucza publicznego w algorytmach kryptograficznych.

Oczywiście forum nie oferuje od razu każdemu użytkownikowi możliwości ukrywania wiadomości. W celu skorzystania z tej funkcji należy wykonać specjalną sekwencję ruchów, znaną z założenia tylko wybranym użytkownikom. Zagadnienie to jest szerzej opisane w rozdziale 6.4.

6.3 Koncepcja widoków

W celu przeglądania postów forum udostępnia dwa rodzaje widoków: **normalny**, który dostępny jest dla każdego zalogowanego użytkownika, wyświetlający treść postów wraz z dołączonymi do nich plikami graficznymi, oraz **tajny**, który dostępny jest jedynie po aktywowaniu modułu steganograficznego za pomocą specjalnej sekwencji. Widok ten pomija posty bez ukrytej treści, pokazując jedynie wiadomości odkodowane z obrazów.

Wygodne zmienianie rodzaju widoku umożliwia specjalny element interfejsu, tzw. przełącznik (*ang. switch*), który tworzony i obsługiwany jest przez moduł steganograficzny, tzn. zmiana widoku możliwa jest tylko po uzyskaniu dostępu do tego modułu. Wzbogaca on widok dodawania nowego posta o pole formularza do wpisania tajnej wiadomości oraz właśnie przełącznik do zmiany widoku. Jego wygląd został przedstawiony na rysunku 6.3.



Rys. 6.3. Przełącznik – po lewej w stanie “widok normalny”, po prawej “widok tajny”

6.4 Uzyskiwanie dostępu do modułu steganograficznego

Jak zostało wcześniej napisane, uzyskanie dostępu do modułu steganograficznego wymaga wykonania na forum ściśle określonej sekwencji ruchów. Zdarzenia wchodzące w skład tej sekwencji są rejestrowane przez serwer aplikacji osobno dla każdego użytkownika, dzięki czemu wie on w danej chwili w którym kroku wykonywania sekwencji jest każdy użytkownik.

Pożądanymi cechami, które posiadać powinna dobrze określona sekwencja są odpowiedni poziom skomplikowania poszczególnych kroków oraz ich niezbyt mała liczba (przynajmniej kilka). W konkretnym przypadku aplikacji będącej częścią tej pracy zaimplementowano następującą sekwencję:

1. Należy, jako zalogowany użytkownik, napisać na forum post w dowolnym temacie, o dowolnej treści.
2. Następnie, w ciągu 10 sekund konieczne jest wyedytowanie go bez zmiany treści i zapisanie. Postronni użytkownicy nie zauważą żadnej zmiany, lecz serwer aplikacji zarejestruje takie zdarzenie edycji jako wykonanie drugiego kroku.
3. Ostatnią czynnością jaką należy wykonać jest przejście do zakładki “Profile” w górnym panelu strony a następnie uzupełnienie pola formularza o nazwie “How old?” w sekcji odnoszącej się do cech kota użytkownika forum o liczbę wyrazów, z których składa się post napisany w punkcie 1.

Wykonywanie poszczególnych kroków zostało przedstawione na rysunkach 6.4 – 6.6.

Add new post

Przykładowa treść posta, który ma składa się z 10 wyrazów.

SUBMIT

ADD IMAGE

Rys. 6.4. Tworzenie nowego posta na forum (krok pierwszy)

Przykładowa treść posta, który ma składa się z 10 wyrazów.

SAVE

Rys. 6.5. Aktualizacja tego samego posta bez zmiany treści w ciągu 10 sekund (krok drugi)

Do you have a cat?

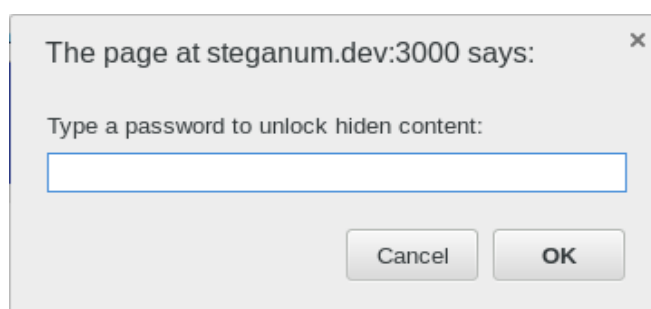
<input type="text" value="Cat Name"/>	<input type="text" value="Cat Breed"/>	<input type="text" value="How old?"/>
		<input type="text" value="10"/>

Or dog?

<input type="text" value="Dog Name"/>	<input type="text" value="Dog Breed"/>	<input type="text" value="How old?"/>
---------------------------------------	--	---------------------------------------

Rys. 6.6. Uzupełnienie w formularzu pola "Wiek Twojego kota" (krok trzeci)

Po poprawnym wykonaniu całej sekwencji serwer automatycznie przekaże aplikacji zminifikowany kod modułu steganograficznego, a aplikacja po stronie klienta sama rozpozna, że otrzymała kod JavaScript, na którym wykona następnie operację *eval*. Następuje wówczas inicjalizacja modułu steganograficznego, częścią której jest wyświetlenie użytkownikowi niewielkiego okna z pytaniem (*ang. prompt*) o tajne hasło, które zostanie użyte do odkodowywania/ukrywania wiadomości. Etap ten pokazany został na rysunku 6.7.



Rys. 6.7. Okno z zapytaniem o tajne hasło, pokazujące się po inicjalizacji modułu steganograficznego

Podane przez użytkownika hasło nie jest w żaden sposób weryfikowane; jeśli poda on hasło inne niż to użyte do ukrycia wiadomości w obrazach na forum, to nie będzie w stanie odczytać żadnej tajnej wiadomości. Celem takiego zachowania modułu jest

zapewnienie maksymalnego bezpieczeństwa użytkownikom korzystającym z modułu steganograficznego, gdyż znane tylko im hasło nie jest w żadnej postaci przesyłane w sieci komunikacyjnej.

6.5 Omówienie zaimplementowanych algorytmów steganograficznych

6.5.1 Dane wejściowe

Algorytmy: ukrywający wiadomość w pliku graficznym oraz odkodowujący ją z pliku wymagają takich samych danych wejściowych. Są to:

- plik graficzny w formacie z kompresją bezstratną (BMP lub PNG)
- wartość 1024-bitowa pobrana z serwera, skojarzona z danym obrazkiem
- tajne hasło podane przez użytkownika w trakcie inicjalizacji modułu steganograficznego

Oczywiście do algorytmu ukrywającego wiadomość w pliku graficznym musimy dostarczyć również treść tej wiadomości.

6.5.2 Wyznaczanie wartości klucza steganograficznego

Klucz steganograficzny wyznaczany jest jako wynik operacji bitowej *xor* wartości 1024-bitowej skojarzonej z danym plikiem i pobranej z serwera oraz tajnego hasła wpisywanego przez użytkownika. Uzyskana wartość stanowi tzw. ziarno (*ang. seed*) dla generatora liczb pseudolosowych, zaimplementowanego zgodnie z algorytmem *xorshift1024**. Sytuacja taka zachodzi dla obu algorytmów, zarówno ukrywającego, jak i odkodowującego wiadomość z pliku graficznego.

6.5.3 Kodowanie znaków tajnych wiadomości

Każdy znak wiadomości zapisywany jest w kodowaniu UTF-16, dlatego możliwe jest używanie liter diakrytyzowanych z wielu alfabetów świata. Algorytm zakłada wykorzystanie maksymalnie 3 bitów w każdym pikselu użytym do zapisania części wiadomości, po jednym, najmniej znaczącym bicie z każdego kanału koloru: czerwonego, zielonego i niebieskiego. Dzięki takiemu rozwiązaniu zmiana koloru piksela wykorzystanego do zapisania w nim części wiadomości będzie całkowicie niezauważalna dla ludzkiego oka.

Oznacza to także, że do zapisania 1 litery tajnej wiadomości potrzeba wyznaczyć 6 różnych pikseli, w których ukrywane będą kolejne bity znaku w kodowaniu UTF-16. Kodowanie to, jak wiadomo, mapuje liczby dwubajtowe, czyli 16-bitowe na określone

znaki (*ang. characters*). Tak więc do ukrycia 16 bitów, zakładając możliwość ukrycia do 3 bitów na piksel potrzeba minimum 6 pikseli oryginalnego obrazu.

6.5.4 Algorytm ukrywający wiadomość

6.5.4.1 Zniekształcanie obrazu na najmniej znaczących bitach

Etapem wstępnym algorytmu jest ustawianie wygenerowanych w sposób pseudolosowy zer i jedynek na najmniej znaczące pozycje kolorów czerwonego, zielonego i niebieskiego w każdym z pikseli oryginalnego obrazu. Ma to na celu maksymalne utrudnienie odczytania ukrywanej wiadomości przez osoby nieupoważnione. Nawet jeśli wiadomość zostałaby ukryta w ogólnodostępnym pliku graficznym (np. zdjęciu umieszczonym publicznie w Internecie), to porównując oryginalny obraz ze steganogramem nie będzie możliwe wskazanie pikseli, w których ukryto informację. Stanie się tak dlatego, że większość z nich będzie się różnić na najmniej znaczących bitach od oryginału jeszcze przed etapem ukrywania tajnego przekazu.

6.5.4.2 Wyznaczanie kolejnych pikseli, na których zapisana będzie wiadomość

Mając podany klucz steganograficzny, algorytm przystępuje do etapu wyznaczania kolejnych pikseli, w których zapisane zostaną kolejne bity tajnej wiadomości. Generator ustawiony ziarnem o wartości klucza steganograficznego wyznacza kolejne liczby pseudolosowe z zakresu od 0 do $n - 1$, gdzie n to liczba pikseli oryginalnego obrazu. W przypadku gdy generator wyznaczy jakikolwiek piksel ponownie, podaje on kolejne wartości, aż do uzyskania takiej, która jeszcze nie wystąpiła (niepoprawne byłoby zapisanie danych więcej niż raz w tym samym pikselu, ponieważ za każdym kolejnym razem zniszczeniu ulegałaby poprzednia informacja).

6.5.4.3 Ukrywanie kolejnych bitów wiadomości

Każdy znak wiadomości, którą algorytm ukrywa znajdzie się w 6 pikselach oryginalnego obrazu, przy czym bity ukrywane są trójkami, począwszy od najmniej znaczących aż do najbardziej znaczących w każdej liczbie 16-bitowej będącej reprezentacją znaku w kodowaniu UTF-16.

Implementację w języku JavaScript funkcji odpowiedzialnej za wybór kolejnych 3 bitów 16-bitowej wartości, które należy ukryć przedstawia listing 7.1.

Listing 6.1. Implementacja funkcji pobierającej określoną trójkę bitów 16-bitowej wartości

```

/* charCode - 16-bitowa wartość będąca kodem UTF-16
 * charPixelNumber - liczba od 0 do 5, numer trójki bitów
 */
function _getNextBits(charCode, charPixelNumber) {
    // przesun w lewo i pobierz 3 najmłodsze bity
    return ((charCode >> 3*charPixelNumber) & 0x7);
}

```

Z kolei kluczowy fragment funkcji `_hide3Bits` zapisującej trójkę bitów wyznaczoną wcześniej przez funkcję `_getNextBits` przedstawia listing 6.2.

Listing 6.2. Kluczowy framgent funkcji ukrywającej 3 bity w pojedynczym pikselu

```

/* data - trzy bity, które mają zostać ukryte w danym pikselu
 * pixelNum - numer piksela, w którym następuje ukrycie bitów
 */
function _hide3Bits(data, pixelNum) {
    /* ... */
    var pixel = stegano.module('image').getPixel(image, x, y),
        color = {
            r: (pixel.r & 0xFE) | ((data & 0x04) >> 2),
            g: (pixel.g & 0xFE) | ((data & 0x02) >> 1),
            b: (pixel.b & 0xFE) | (data & 0x01)
        };
    /* ... */
}

```

Jak widać, najbardziej znaczący bit z trójki zostaje zapisany na najmniej znaczącym bicie składowej czerwonej koloru, środkowy – składowej zielonej a najmniej znaczący – składowej niebieskiej.

6.5.4.4. Łańcuch zakończony zerem

Wiadomość zostaje ukryta w obrazie w celu późniejszego odekodowania i odczytania jej przez upoważnione osoby, dlatego aby ułatwić pracę algorytmowi odekodującemu algorytm ukrywający dodaje znak zerowy jako ostatni znak ukrywany w obrazie. Dzięki temu algorytm odekodujący może w łatwy sposób określić kiedy odczytał już całą wiadomość i zakończyć się po odczytaniu z obrazu samych zer jako kolejnego znaku wiadomości.

6.5.5 Algorytm odekodujący wiadomość

6.5.5.1 Wyznaczanie kolejnych pikseli, z których odczytywana będzie wiadomość

Algorytm odczytujący wiadomość po wyznaczeniu klucza steganograficznego od razu przechodzi do wyznaczania pikseli, na których ukryta została wiadomość. Proces ten zachodzi w taki sam sposób jak w algorytmie ukrywającym wiadomość.

6.5.5.2 Odczytywanie kolejnych bitów wiadomości

Z każdego odwiedzonego przez algorytm piksela odczytywany jest jego kolor. Pobierane z niego są 3 bity, tj. po jednym najmniej znaczącym bicie z każdego kanału koloru: czerwonego, zielonego i niebieskiego. Bity te łączone są w wartość 3-bitową w tej kolejności. Po wykonaniu sześciu iteracji takiej procedury algorytm odczytał 18 bitów danych, z których 2 ostatnie nie mają znaczenia i należy je odrzucić. Pozostałe 16 bitów tworzy kod pojedynczego znaku tajnej wiadomości w UTF-16. Wartość ta zamieniana jest na odpowiadającą jej literę i dołączana do wynikowego łańcucha, zawierającego odkodowaną wiadomość.

Implementacje funkcji: odczytującej odpowiednie bity z wartości koloru danego piksela oraz głównej funkcji algorytmu przedstawiają listingi 7.3 i 7.4.

Listing 6.3. Implementacja funkcji odczytującej 3 bity wiadomości z pojedynczego piksela

```
/* pixelNum - numer piksela, którego kolor należy odczytać
 */
function _decodeBitsFrom(pixelNum) {
    // pobranie danych obrazu
    var imageData = stegano.module('image').getData();

    // wyznaczenie współrzędnych piksela o danym numerze
    // oraz pobranie wartości jego koloru
    var x = pixelNum % imageData.width,
        y = Math.floor(pixelNum / imageData.width),
        pixel = stegano.module('image').getPixel(imageData, x, y);

    // odczyt trójki bitów
    return ((pixel.r & 0x01) << 2) | ((pixel.g & 0x01) << 1) |
        (pixel.b & 0x01);
}
```

Wartością funkcji jest odczytana trójka bitów w odpowiedniej kolejności (takiej samej, w jakiej była zapisana w obrazie, tzn. najstarszy bit trójki pochodzi ze składowej czerwonej, środkowy z zielonej, a najmniej znaczący z niebieskiej).

Listing 6.4. Główna funkcja algorytmu odkodowującego wiadomość

```
/*  imageElement - element drzewa DOM będący obrazem,
    z którego algorytm ma wydobyć wiadomość
*/
function _retrieving(imageElement) {
    var decodedText = '', security = 0, usedPixels = [];
    do {
        var decodedLetter = 0;
        for (var i = 0; i < 6; ++i) {
            var _nextPixel = _getNextPixel(usedPixels),
                _nextDecodedBits = _decodeBitsFrom(_nextPixel);

            // dołącz kolejną trójkę bitów do odczytywanej
            // obecnie wartości
            decodedLetter |= _nextDecodedBits << 3*i;
        }
        decodedText += String.fromCharCode(decodedLetter);
        ++security;
    } while (decodedLetter !== 0 &&
        security < MAX_SECRET_MESSAGE_LENGTH);
    /* ... */
}
```

Wspomnieć należy o celu zastosowania zmiennej *security* oraz stałej *MAX_SECRET_MESSAGE_LENGTH* w powyższym kodzie. Jeśli użytkownik poda w momencie inicjalizacji modułu steganograficznego, po otrzymaniu go z serwera, hasło inne niż to, którego użyto do ukrycia wiadomości w danym pliku graficznym, to algorytm odczytując wiadomość za pomocą błędnego klucza normalnie mógłby nigdy się nie zakończyć. Prawidłowe zakończenie algorytmu i wyjście z pętli *do { ... } while (...)* następuje po odczytaniu znaku zerowego, który umieszczany jest na końcu wiadomości przez algorytm ukrywający. Błędny klucz powoduje odczyt nieodpowiednich pikseli przez co algorytm mógłby bardzo długo odczytywać losowe dane bez napotkania znaku zerowego. Właśnie dlatego istnieje możliwość przerwania działania algorytmu jeśli odczytano już przynajmniej *MAX_SECRET_MESSAGE_LENGTH* znaków. Wartością zmiennej *security* jest liczba odczytanych dotychczas znaków wiadomości, natomiast stała *MAX_SECRET_MESSAGE_LENGTH* ustawiona została na wartość 1000. Oznacza to m.in. niemożliwość zapisania w pliku graficznym przez powyższy algorytm tajnej wiadomości o długości większej niż 1000 znaków, lecz jest to wartość wystarczająca na potrzeby forum internetowego.

7. Testy

7.1 Testy jednostkowe

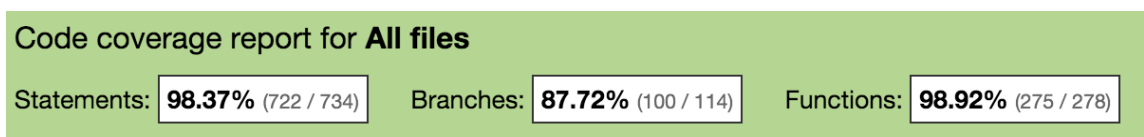
Testy jednostkowe są pierwszym sposobem na sprawdzenie poprawności działania aplikacji. Polegają one na testowaniu jednostek, najczęściej są to klasy w językach programowania zorientowanych obiektowo. W przypadku biblioteki AngularJS są to serwisy, kontrolery, dyrektywy oraz moduły [Dayley 2015]. Niestety platforma MeteorJS nie posiada w rzeczywistości żadnych jednostek, dlatego testy te zostały wykonane tylko dla aplikacji klienckiej.

Do wystartowania testów za pomocą konsoli systemowej użyto narzędzia Karma, do ich tworzenia biblioteki Jasmine, a do wygenerowania raportu dotyczącego pokrycia kodu testami posłużyła wtyczka *karma-coverage*.

Zostało napisanych łącznie 298 testów jednostkowych, pokrywających 98,37% kodu, 87,72% możliwych ścieżek warunków (ang. *branches*) oraz 98,92% funkcji. Całość, wraz z plikiem konfiguracyjnym narzędzia Karma, znajduje się w folderze *tests/client*. Struktura plików i folderów naśladuje tę z aplikacji klienckiej.

```
INFO [karma]: Karma v0.12.37 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.8 (Linux 0.0.0)]: Connected on socket XRNwXynkeZtcJ0-CNWJe with id 41756997
PhantomJS 1.9.8 (Linux 0.0.0): Executed 125 of 298 SUCCESS (0 secs / 0.533 secs)
PhantomJS 1.9.8 (Linux 0.0.0): Executed 298 of 298 SUCCESS (1.152 secs / 1.097 secs)
[21:51:55] Finished 'unit-tests' after 3.66 s
[21:51:55] Finished 'default' after 4.51 s
```

Rys. 7.1. Zrzut ekranu przedstawiający pomyślnie zakończone testy jednostkowe



Rys. 7.2. Raport karma-coverage dotyczący testów jednostkowych aplikacji klienckiej

Testy te polegały głównie na sprawdzeniu czy dane funkcje zwracają odpowiednie wartości oraz czy wywołują metody danego serwisu z odpowiednimi parametrami. W przypadku dyrektyw testowanie były najczęściej wygenerowane widoki i ich zachowanie na różne zdarzenia, takie jak kliknięcie, czy wpisanie wartości do pola w formularzu. Oczywiście wszystkie zależności zostały zastąpione swoimi odpowiednikami naśladującymi zachowanie oryginałów i przeznaczonymi do przeprowadzania testów (*ang. mock object*). Oznacza to, że nie zostały użyte prawdziwe serwisy, a stworzono nowe, zachowujące się podobnie na dane wejściowe, ale nie posiadające dokładnej implementacji. Dzięki temu testy danych jednostek są od siebie niezależne.

Listing 7.1 Przykład testu jednostkowego kontrolera *ShowSectionCtrl*

```
describe 'ShowSection controller', ->
  $scope = { }
  $controller = { }
  ShowSectionCtrl = { }
  mockedSectionsServ =
    getTopics: angular.noop
  mockedStateParams =
    section_id: 96
  mockedState =
    goBack: angular.noop
  successCallback = {}
  topics = [
    'Super Best Friends'
    'Better Super Best Friends'
  ]

  beforeEach module 'sections'

  beforeEach () ->
    spyOn(mockedSectionsServ, 'getTopics').and.returnValue topics

  beforeEach inject ($injector) ->
    $controller = $injector.get('$controller')
    $scope = $injector.get('$rootScope').$new()
    ShowSectionCtrl = $controller 'ShowSectionCtrl',
      $scope: $scope
      sectionsServ: mockedSectionsServ
      $stateParams: mockedStateParams

  it 'should initialize', ->
    expect(ShowSectionCtrl).toBeDefined()

  it 'should call getTopics method on sectionsServ with section.id', ->
    expect(mockedSectionsServ.getTopics).toHaveBeenCalledWith(mockedStateParams.s
ection_id)

  it 'should set topics from sectionsServ.getTopics to vm.topics', () ->
    expect(ShowSectionCtrl.topics).toBe topics

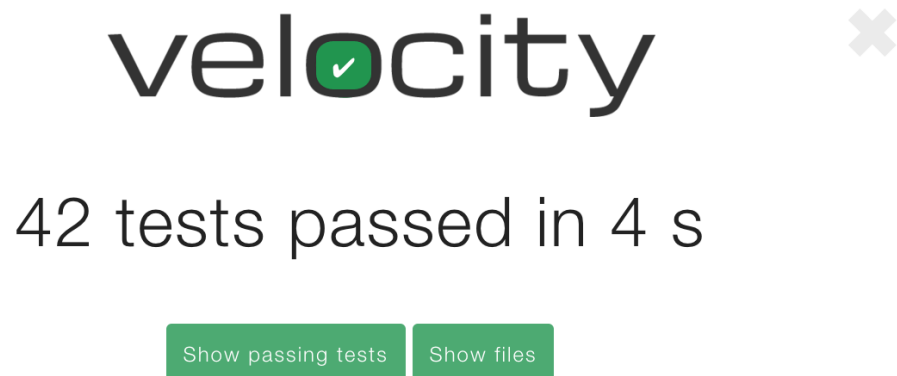
  it 'should set section_id from mockedStateParams to vm.id', () ->
    expect(ShowSectionCtrl.id).toBe mockedStateParams.section_id

  it 'should set sectionsServ to vm.section', () ->
    expect(ShowSectionCtrl.section).toBe mockedSectionsServ
```


7.2 Testy integracyjne

Testy integracyjne są drugim sposobem weryfikowania poprawności napisanego kodu. Polegają one na sprawdzeniu części aplikacji większej niż jednostka. W przypadku biblioteki AngularJS może to być na przykład cały moduł. Dla platformy MeteorJS powstało narzędzie *Velocity* służące do tworzenia testów, w tym przypadku integracyjnych. Tworzy ono tak zwany serwer lustro (ang. *mirror*), działający na innym porcie i rozpoczyna na nim wykonywanie testów. Wyniki są dostępne w głównej aplikacji (działającej najczęściej na porcie 3000) w prawym górnym rogu. Zielona, okrągła ikonka oznacza pomyślne wykonanie wszystkich testów.

Testy te skonstruowane są w taki sposób, że sprawdzają metody publiczne serwera poprzez zwracane przez nie wartości oraz przez weryfikację stanu bazy danych. Znajdują się one w folderze *tests/jasmine/server/integration* i struktura folderów jest podobna do tej w aplikacji serwerowej. Zostały napisane 42 testy integracyjne pokrywające najważniejsze części serwera, takie jak zarządzanie postami, tematami sekcjami, rolami, logowanie oraz rejestracja użytkowników.



Rys. 7.3. Raport Velocity dotyczący testów integracyjnych aplikacji serwerowej

Listing 7.2 Przykład testu integracyjnego metod publicznych dotyczących tematów

```
Helpers = require 'Helpers'

describe 'Topics', ->
  topic =
    section_id: '1'
    name: 'Example topic name'
    description: 'Description of example topic'

  describe 'addTopic method', ->
    it 'should throw not-authorized exception for not logged user', ->
      Helpers.logout()
      Meteor.call 'addTopic', topic, (error) ->
        expect(error).toEqual(new Meteor.Error('notLogged'))

    it 'should add new topic after login', ->
      Helpers.login('admin')
      topics_before = Topics.find({}).count()
      Meteor.call 'addTopic', topic
      expect(Topics.find({}).count()).toEqual(topics_before + 1)
      expect(Topics.find({ name: 'Example topic name'
    })).count().toBeGreaterThan(0)

  describe 'updateTopic method', ->
    beforeEach ->
      Helpers.login('admin')

    it 'should update topic name and description in database', ->
      local_topic = Topics.findOne({ name: 'Example topic name' })
      expect(local_topic).not.toBe(undefined)

      local_topic.name = 'Updated topic name'
      local_topic.description = 'Updated description of example topic'

      Meteor.call 'updateTopic', local_topic

      updated_topic = Topics.findOne({ id: local_topic.id })
      expect(updated_topic).not.toBe(undefined)
      expect(updated_topic.name).toBe('Updated topic name')
      expect(updated_topic.description).toBe('Updated description of
example topic')

      expect(updated_topic.updatedAt).toBeGreaterThan(local_topic.updatedAt)

  describe 'deleteTopic method', ->
    beforeEach ->
      Helpers.clear()
      Helpers.seed.topic(topic)
      Helpers.login('admin')

    it 'should delete given topic', ->
      local_topic = Topics.findOne({ name: 'Example topic name' })
      expect(local_topic).not.toBe(undefined)

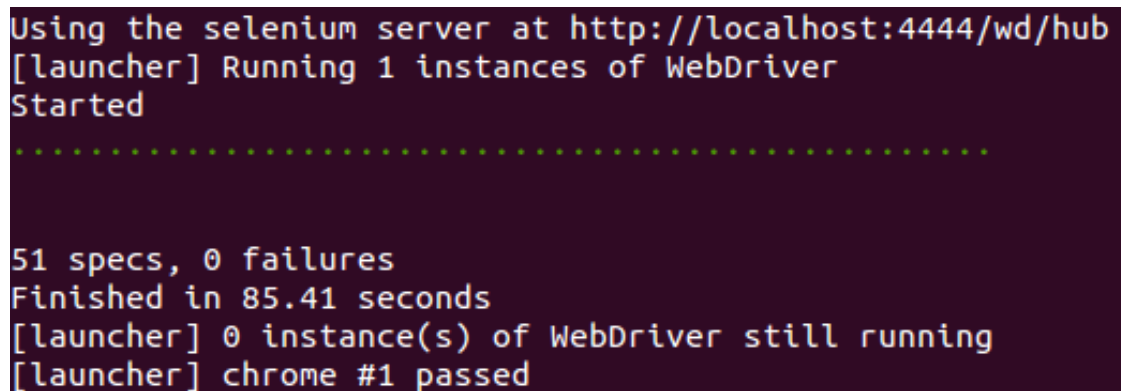
      Meteor.call 'deleteTopic', local_topic.id

      local_topic = Topics.findOne({ name: 'Example topic name' })
      expect(local_topic).toBe(undefined)
```

7.3 Testy funkcjonalne

Ostatnim sposobem na zweryfikowanie poprawności napisanego kodu są testy funkcjonalne. Polegają one na automatycznym wykonywaniu akcji użytkownika na gotowej aplikacji i sprawdzenie jej odpowiedzi widocznej w przeglądarce internetowej.

Do wykonywania testów użyto narzędzia Protractor, a do ich tworzenia wykorzystano możliwości *ECMAScript* w wersji 6. Całość znajduje się w folderze *test/e2e*. Głównym plikiem zarządzającym kolejnością wykonywania testów jest *main.scenario.js*. Importuje on inne scenariusze (czyli pliki z rozszerzeniem *scenario.js*), które posiadają strukturę klasy i zwracają jej nowy obiekt. Każda taka klasa dziedziczy metody z innych klas znajdujących się w tym samym folderze. Istnieje jeszcze główna klasa (*lib/main.class.js*) posiadająca ogólne metody wykorzystywane w wielu scenariuszach. Zostało napisanych w sumie 51 testów jednostkowych sprawdzających główne założenia forum steganograficznego.

A screenshot of a terminal window with a dark background and light-colored text. The text shows the execution of Protractor tests using Selenium WebDriver. It starts with the Selenium server URL, then indicates that 1 instance of WebDriver was launched. After a separator line of dots, it reports that 51 specifications were run with 0 failures, taking 85.41 seconds. It concludes by stating that 0 instances of WebDriver are still running and that the Chrome browser instance #1 passed the tests.

```
Using the selenium server at http://localhost:4444/wd/hub
[launcher] Running 1 instances of WebDriver
Started
.....

51 specs, 0 failures
Finished in 85.41 seconds
[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #1 passed
```

Rys. 7.4. Screen przedstawiający pomyślnie zakończone testy funkcjonalne

Listing 7.3 Przykład testu funkcjonalnego dotyczący tematów

```
let TopicClass = require('./topic.class')
let AuthClass = require('./user/auth.class')
let AuthHelpers = new AuthClass;
class TopicsScenario extends TopicClass {
  constructor() { super(); }
  begin() {
    AuthHelpers.loginAs('admin', 'admin');
    browser.get('/sections');
    browser.sleep(500);
    this.sections.first().click();
  }
  run() {
    describe('Topics scenario', () => {
      let newTopic = {
        name: 'someTopicName',
        description: 'someTopicDescription';
      }
      let editTopic = {
        name: 'updatedTopicName',
        description: 'updatedTopicDescription';
      }
      it('begin', () => {
        this.begin();
      });
      describe('Topics page', () => {
        let topicsCountBefore;
        beforeEach(() => {
          this.topics.count().then((count) => {
            topicsCountBefore = count;
          });
        });
        it('should add a new topic', () => {
          this.addTopic(newTopic.name, newTopic.description);
          this.addTopic(newTopic.name, newTopic.description);
          expect(this.topics.count()).toBe(topicsCountBefore + 2);
          expect(this.topics.first().getText()).toContain(newTopic.name);
          expect(this.topics.first().getText())
            .toContain(newTopic.description);
        });
        it('should edit the first topic', () => {
          this.editTopic(0, editTopic.name, editTopic.description);
          expect(this.topics.first().getText()).toContain(editTopic.name);
          expect(this.topics.first().getText())
            .toContain(editTopic.description);
        });
        it('should delete the last topic', () => {
          this.deleteLast();
          this.cancelDelete();
          this.deleteLast();
          this.confirmDelete();
          expect(this.topics.count()).toBe(topicsCountBefore - 1);
          expect(this.topics.first().getText()).toContain(editTopic.name);
          expect(this.topics.first().getText())
            .toContain(editTopic.description);
        });
      });
      it('should logout', () => {
        AuthHelpers.logout();
      });
    });
  }
}
module.exports = new TopicsScenario;
```

8. Podsumowanie i wnioski

W ramach niniejszej pracy inżynierskiej zrealizowano forum internetowe, mogące pełnić funkcję tajnego kanału komunikacyjnego dla użytkowników potrafiących uzyskać dostęp do modułu steganograficznego. Warty odnotowania jest fakt, że dla ludzi nieświadomych jego możliwości ukrywania wiadomości w obrazach nadal stanowi praktyczną platformę wymiany informacji, dokładnie tak jak tradycyjne forum internetowe.

Sposób implementacji aplikacji bierze pod uwagę aspekty bezpieczeństwa: aby uzyskać dostęp do funkcjonalności steganograficznej na forum należy wykonać odpowiednią sekwencję ruchów w ściśle określonym czasie oraz podać hasło umożliwiające wydobyć ukrytej treści z obrazów, które mogą być załączane opcjonalnie do każdego posta.

Wprowadzanie losowych wartości "0" i "1" na najmniej znaczących pozycjach wartości kolorów czerwonego, zielonego i niebieskiego we wszystkich pikselach obrazu, poprzedzające właściwe ukrywanie informacji, ma za zadanie maksymalnie utrudnić atakującemu wytypowanie tych pikseli, które faktycznie zawierają kolejne bity tajnej wiadomości. Stosowanie innego klucza dla każdego pliku graficznego zawierającego ukrytą informację zapewnia, że sekwencja pikseli, z których należy odczytać kolejne bity jest za każdym razem inna.

Stworzenie aplikacji typu SPA (ang. *Single Page Application*) pozwoliło zwiększyć wygodę użytkownika forum. Brak pełnych przeładowań strony podczas użytkowania skutkuje skróconym czasem odpowiedzi aplikacji na akcje wykonywane przez użytkownika. Pozwala także na przechowywanie raz uzyskanego modułu steganograficznego w zmiennej globalnej kodu JavaScript. W razie konieczności szybkiego wyłączenia tego modułu użytkownik może po prostu wymusić przeładowanie (odświeżenie) zawartości strony w przeglądarce. Dodatkowa funkcjonalność nie będzie już dostępna.

Przedstawiona aplikacja, mimo że spełnia wymagania postawione na wstępie pracy, może być w przyszłości wciąż ulepszana i rozbudowywana. W celu zwiększenia bezpieczeństwa możliwe byłoby na przykład szyfrowanie wiadomości jeszcze przed etapem ukrywania jej w obrazie [Bateman 2008]. W oparciu o opisywane forum łatwe byłoby także stworzenie analogicznego czatu internetowego.

Mając na celu użycie steganografii do przekazania zaufanej stronie wrażliwych informacji nie jest jednak konieczne konstruowanie od zera własnej aplikacji, która to umożliwia. Wyizolowanie z forum samego modułu steganograficznego i jego

niewielkie zmodyfikowanie pozwoliłyby na ukrywanie wiadomości w plikach graficznych bez wzbudzania podejrzeń na wielu popularnych serwisach internetowych, które z założenia służą użytkownikom do zamieszczania obrazków. Przykładami takich portali mogą być na przykład *9gag.com* oraz *4chan.org*. Cała procedura polegałaby na tym, aby jedna osoba, korzystając z przeglądarki internetowej przygotowywała plik z ukrytą wiadomością, po czym zamieściła go na jednym z ww. serwisów. Inne osoby, które wiedzą na jakie obrazki, w jakie dni i w jakich godzinach czekać mogłyby następnie użyć tego samego modułu do wydobywania tajnej wiadomości. Jednocześnie trudno spodziewać się, aby ktoś analizował wszystkie pliki graficzne zamieszczane przez wszystkich użytkowników na portalach, które są do tego przeznaczone, pod kątem obecności ukrytych komunikatów.

Steganografia okazała się interesującą alternatywą (bądź wzmocnieniem jeśli ukrywana treść będzie dodatkowo szyfrowana) dla kryptografii. Wykorzystana odpowiednio jest w stanie zapewnić bezpieczeństwo komunikacji choć, paradoksalnie, informacje nośne (w opisywanym przypadku pliki graficzne dołączane do postów na forum) są publicznie i łatwo dostępne. Postępowanie takie wychodzi z założenia, że "najciemniej jest pod latarnią", a atakującemu, chcącemu wykraść wrażliwe informacje, do których nie ma dostępu nie przyjdzie do głowy szukać ich na pospolitym forum internetowym. Nawet w przypadku chęci dokładnego przeszukania treści umieszczanej w takich miejscach jak fora w Internecie każdego dnia, byłoby to zadanie niemożliwe do wykonania ze względu na zbyt dużą ilość danych konieczną do przeanalizowania.

Podsumowując, wzrastająca wciąż w wykładniczym tempie ilość informacji produkowanej każdego dnia przez ludzkość tworzy sprzyjające warunki dla stosowania steganografii. Zakładając utrzymanie się takiego trendu wnioskować należy zwiększone zainteresowanie i wykorzystywanie tej dziedziny nauki w przyszłości, jako rozsądnej alternatywy lub dodatku dla kryptografii w zakresie bezpiecznej wymiany danych.

9. Bibliografia

- [1] Bateman B., “Image Steganography and Steganalysis”, <http://personal.cs.surrey.ac.uk/personal/st/H.Schaathun/projects/Past/phil-msc.pdf>, dostęp 24.01.2016, 2008
- [2] Dayley B., “Node.js, MongoDB, AngularJS. Kompendium wiedzy”, Wydawnictwo Helion, Gliwice, 2015
- [3] Garbaczuk W., Kopniak P., “Steganologia: współczesne metody ochrony informacji (przegląd)”, *Pomiary Automatyka Kontrola*, Wydawnictwo PAK, numer 3/2005, strony 21-25
- [4] Green B., Seshadri S., “AngularJS”, Wydawnictwo Helion, Gliwice, 2014
- [5] <http://www.angular-meteor.com>, dostęp: 23.01.2016
- [6] <http://www.docs.meteor.com/#/full/>, dostęp: 23.01.2016
- [7] <https://docs.angularjs.org/api>, dostęp: 23.01.2016
- [8] Stefanov S., “JavaScript Wzorce”, Wydawnictwo Helion, Gliwice, 2012