

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ W LUBLINIE

Wydział Matematyki, Fizyki i Informatyki

Lublin 2017

Praca licencjacka

napisana w Zakładzie Informatyki Stosowanej

pod kierunkiem dra Rajmunda Kuduka

Automatyzacja procesu wytwarzania oprogramowania na przykładzie narzędzia Yeoman

Automation of software development based on Yeoman tool

Kierunek: Informatyka

Michał Żmudziak

nr albumu: 258347

# Spis treści

[Spis treści 2](#_Toc487021861)

[Wstęp 3](#_Toc487021862)

[1. Podstawy teoretyczne 5](#_Toc487021863)

[1.1. Automatyzacja pisania kodu 5](#_Toc487021864)

[1.1.1. Programowanie automatyczne 5](#_Toc487021865)

[1.1.2. Automatyczne generowanie kodu źródłowego 6](#_Toc487021866)

[1.1.3. Generowanie kodu w IDE 7](#_Toc487021867)

[1.1.4. Ryzyko wynikające z automatyzacji 8](#_Toc487021868)

[1.2. Yeoman – historia frameworku i jego rozwój 9](#_Toc487021869)

[1.2.1. Historia frameworku 9](#_Toc487021870)

[1.2.2. Początki frameworku i podstawowe funkcjonalności 9](#_Toc487021871)

[1.2.3. Korzyści płynące z używania narzędzia Yeoman 10](#_Toc487021872)

[1.2.4. Przypadki użycia generatora 12](#_Toc487021873)

[1.2.5. Przykładowe generatory 13](#_Toc487021874)

[2. Środowisko Yeoman 14](#_Toc487021875)

[2.1. Stosowane technologie 14](#_Toc487021876)

[2.1.1. Node.js 14](#_Toc487021877)

[2.1.2. npm 14](#_Toc487021878)

[2.1.3. Gulp 15](#_Toc487021879)

[2.1.4. Bower 15](#_Toc487021880)

[2.1.5. Inquirer.js 15](#_Toc487021881)

[2.2. Sposób działania generatora Yeoman 15](#_Toc487021882)

[2.2.1. Metody jako elementy bazowe 15](#_Toc487021883)

[2.2.2. Pętla programowa 16](#_Toc487021884)

[2.2.3. Generatory zagnieżdżone 18](#_Toc487021885)

[2.2.4. Zdarzenia synchroniczne 18](#_Toc487021886)

[2.2.5. Interakcja z systemem plików 19](#_Toc487021887)

[3. Własna implementacja 21](#_Toc487021888)

[3.1. Generator i pod-generator 21](#_Toc487021889)

[3.1.1. Generator główny - genapi 22](#_Toc487021890)

[3.2.2 Generowane pliki 23](#_Toc487021891)

[3.1.2. Generator poboczny - genapi:entity 24](#_Toc487021892)

[3.2. Interakcja z użytkownikiem 26](#_Toc487021893)

[3.3 Zapisywanie plików na dysku 28](#_Toc487021894)

[3.4. Używanie generatora 31](#_Toc487021895)

[3.4.1. Instalacja 31](#_Toc487021896)

[3.4.2. Uruchomienie 31](#_Toc487021897)

[3.5. Możliwości rozwoju 32](#_Toc487021898)

[4. Podsumowanie 33](#_Toc487021899)

[Bibliografia 34](#_Toc487021900)

# Wstęp

Dzisiejszy rynek wytwarzania oprogramowania staje się coraz bardziej wymagający. Oprócz jakości i krótkiego czasu dostarczenia gotowego produktu, na ważności zyskują także procesy związane z pracą nad projektem. W celu osiągnięcia jak najwyższej niezawodności i powtarzalności, wdrażane są rozmaite metodologie i procesy optymalizujące pracę. Jednym z takich procesów jest właśnie automatyzacja pisania kodu źródłowego.

W nowoczesnych środowiskach programistycznych dostępne są na przykład rozbudowane moduły generujące skrawki kodu (ang. *code* *snippets*) lub podstawy nowego projektu. Dostępne są również samodzielne programy tworzące pliki źródłowe od podstaw, a także pozwalające na ciągłą pracę z projektem poprzez generowanie nowych jego elementów. Przykładem takiego oprogramowania są napisane we frameworku Yeoman generatory, w pewnym stopniu wyręczające programistów z pracy nad pisaniem samego kodu. W ramach przedstawianej pracy opisane zostały podstawy automatyzacji wytwarzania kodu źródłowego, pokazany został proces działania generatora Yeoman, sposób jego tworzenia oraz zastosowane w nim technologie i rozwiązania.

W pierwszym rozdziale przedstawiona została krótka historia automatyzacji pisania oprogramowania, generowania kodu oraz frameworku Yeoman. Pokazane są też korzyści płynące z włączenia generatorów w swój tryb pracy, a także przykładowe generatory i ich funkcjonalności.

W drugim rozdziale zawarte są informacje o narzędziach stosowanych przez framework oraz przedstawiony sposób jego działania. Wytłumaczone zostały również podstawowe pojęcia.

Trzeci rozdział zawiera szczegóły dotyczące implementacji własnego generatora, przechodząc kolejno przez warstwy jego struktury, odzwierciedlając tok pracy generatora. Pokazane jest, jak można używać generatorów Yeoman oraz zakreślone są możliwości dalszego rozwoju aplikacji tworzonej w ramach pracy.

Celem napisania niniejszej pracy dyplomowej było pokazanie, jak pomocna może być automatyzacja w postaci generowania kodu oraz przybliżenie frameworku Yeoman.

# 1. Podstawy teoretyczne

## 1.1. Automatyzacja pisania kodu

### 1.1.1. Programowanie automatyczne

Jest to typ programowania, w którym stosowane są mechanizmy generowania programu, pozwalając programiście pracować na wyższym poziomie abstrakcji. Jedną z pierwszych wzmianek o programowaniu automatycznym jest artykuł Saula Gorna[[1]](#footnote-1) z 1940 roku, czyli czasów, kiedy kod źródłowy był zapisywany na podziurkowanych kartach. Autor artykułu odnosi się w ten sposób do asemblerów pozwalających programiście pisać kod, który po interpretacji przez komputer wybijany był automatycznie na perforowanych kartach w postaci otworów. Oznaczało to, że komputer automatycznie wykonuje zadania, które normalnie wykonywałby człowiek. Później, termin programowania automatycznego odnosił się do używania języków jak FORTRAN, w których programista specyfikował co chce osiągnąć, a komputer interpretował jego specyfikację generując kod maszynowy. Programowanie automatyczne oznaczało w końcu to, co dzisiaj rozumiemy poprzez programowanie w nowoczesnych językach z wysokim poziomem abstrakcji.

David Lorge Parnas[[2]](#footnote-2), profesor Carnegie Mellon University w Kanadzie, w jednym ze swoich listów [1] w których zastanawia się nad użyciem oprogramowania w systemach broniących przed atakami nuklearnymi uważa, że programowanie automatyczne niezbyt nadaje się do wykorzystania w systemach czasu rzeczywistego, z ograniczonym czasem na reakcję. Podnosi, iż kod pisany na wyższym poziomie abstrakcji jest mało efektywny i niedokładny, co powoduje jego nieużyteczność w systemach zarządzania bitewnego. David Parnas trafnie przewidział przyszłość, gdyż do dziś najważniejsze systemy militarne pisane są w językach niskiego poziomu z bezpośrednim dostępem do sprzętu, jednak programowanie automatyczne w postaci języków z wysokim poziomem abstrakcji z powodzeniem używane jest w wielu systemach o kluczowej wartości.

Ciekawe wyjaśnienie pojęcia programowania automatycznego podała Mildred Koss: [2]

*„Writing machine code involved several tedious steps—breaking down a process into discrete instructions, assigning specific memory locations to all the commands, and managing the I/O 30 Grey Room 18 buffers. After following these steps to implement mathematical routines, a sub-routine library, and sorting programs, our task was to look at the larger programming process. We needed to understand how we might reuse tested code and have the machine help in programming. As we programmed, we examined the process and tried to think of ways to abstract these steps to incorporate them into higher-level language. This led to the development of interpreters, assemblers, compilers, and generators—programs designed to operate on or produce other programs, that is, automatic programming”*

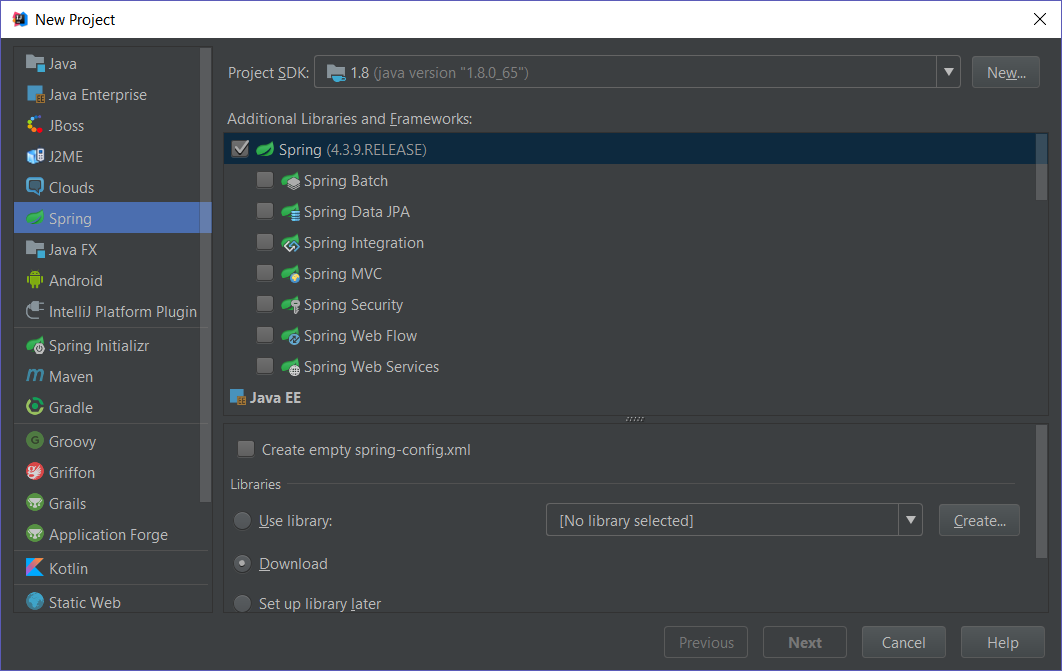
Programiści często dążą do zwiększania swojej produktywności poprzez automatyzację zadań. Takie podejście doprowadziło do stworzenia języków programowania jakie znamy dzisiaj, kompilatorów kodu z jednego języka do drugiego, a także generatorów tworzących cały kod źródłowy projektu od nowa. Programowanie automatyczne to abstrakcja, pozwalająca na pisanie kodu będącego jednocześnie zrozumiałego dla człowieka i komputera. Sprawia, że programowanie jest skupione na rozwiązaniu problemu i pozwala zapomnieć o maszynie. Pozwala na uruchamianie programu na więcej niż jednej maszynie - cecha będąca dzisiaj naturalną dla programowania [3]. Programowanie automatyczne w dzisiejszym znaczeniu oznacza w dużej mierze samo generowanie kodu źródłowego, transformację abstrakcyjnych modeli do kodu, a czasami także automatyczną refaktoryzację kodu w środowiskach programistycznych.

### 1.1.2. Automatyczne generowanie kodu źródłowego

Z automatycznego generowania kodu wynika wiele korzyści, jedną z oczywistych jest przyspieszenie pracy. Interfejs użytkownika jest zwykle w aplikacjach sieciowych jednolity, co oznacza, że ekrany są do siebie bardzo podobne. Gdyby programista posiadał pewien wzorzec, mógłby uzupełniać go automatycznie, zaoszczędzając sobie tym samym pisania wielu linii kodu. Automatycznie generowany kod zwykle jest dobrze napisany, co ułatwia jego utrzymanie i pracę z nim. Takie podejście pozwala programiście na kreatywną pracę, koncentrację nad rzeczywistym problemem i niemartwienie się o powtarzalne czynności. Ostatecznie, generowanie kodu źródłowego zmniejsza koszty poprzez pozostawienie mniej miejsca na nieporozumienia i ludzkie błędy. Naprawianie błędów popełnionych we wczesnym stadium pracy nad projektem zwykle kosztuje najwięcej, tworzenia projektu w sposób automatyczny pozwala zazwyczaj na uniknięcie tego rodzaju pomyłek.

### 1.1.3. Generowanie kodu w IDE

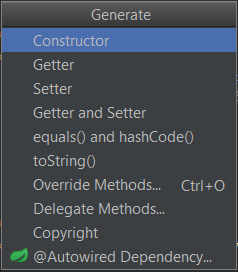
Funkcjonalność automatycznego tworzenia kodu źródłowego dostępna jest dzisiaj w każdym powszechnie używanym zintegrowanym środowisku programistycznym. Trudno jest wyobrazić sobie nowoczesne narzędzie, nieposiadające takich możliwości. Zazwyczaj proces generowania kodu w IDE jest interaktywny, co pozwala użytkownikowi dobrze dobrać potrzebne mu elementy. Zwykle dostępnych jest wiele archetypów, z których programista może wybrać typ swojej aplikacji (*Rys. 1*). Powszechnie używane są kreatory interfejsów graficznych użytkownika, w których programista może dobrać wygląd swojego programu lub jego elementów. Takiego typu aplikacje są w istocie generatorami kodu źródłowego.



*Rys. 1. Oferujący wiele archetypów nowego projektu kreator w IntelliJ IDEA.*

Oprócz klasycznej automatyzacji poprzez generowanie projektu, IDE często wyposażone są w potężne mechanizmy refaktoryzacji kodu. Na uwagę zasługuje środowisko IntelliJ IDEA, potrafiące np. odnajdować duże i skomplikowane pętle oraz wyrażenia logiczne, sugerując zamianę ich w bardziej optymalne wyrażenia lambda[[3]](#footnote-3). Potrafi przeprowadzać mądrą refaktoryzację, wydzielając nowe funkcje, zmienne oraz klasy. Możliwa jest również transkompilacja kodu[[4]](#footnote-4) z jednego języka programowania do drugiego. Są to główne powody, dla których użytkownicy wybierają to środowisko programistyczne, co dobrze pokazuje jak ważną jest to kwestią.

Środowiska programistyczne oferują często generowanie skrawków kodu przy użyciu szablonów w postaci metod, klas lub całych plików (*Rys. 2*.). Umożliwiają także rozszerzanie tej funkcjonalności przy użyciu pluginów, jak *Acceleo* dla środowiska Eclipse, zamieniający pliki w formacie EMF (*Eclipse Modeling Framework*) na kod źródłowy w wybranym języku, lub funkcja *Live Templates* w IntelliJ IDEA pozwalająca na definiowanie własnych struktur używanych do generowania kodu.



*Rys. 2. Menu kontekstowe pozwalające na generowanie skrawków kodu w IntelliJ IDEA*

### 1.1.4. Ryzyko wynikające z automatyzacji

Z automatyzacji pisania kodu wynika na pewno wiele korzyści, ale istnieje również pewne ryzyko. Tworzenie oprogramowania w sposób automatyczny nie jest idealne, służące do tego narzędzia pisane są przecież przez zwykłych programistów. Mogą zdarzyć się przypadki, w których wygenerowany kod jest niekompletny lub wręcz błędny. Takie elementy łatwo jest jednak wychwycić i poprawić, gdyż zwykle takie pomyłki są oczywiste. Niekiedy program może generować zbyt ogromną ilość kodu, niezrozumiałą dla początkującego programisty. O ile nowo wygenerowane projekty są z zasady dobrze ustrukturyzowane i minimalne, niektóre bardziej zaawansowane narzędzia mogą iść za daleko i tworzyć niepotrzebne pliki, które nigdy nie zostaną użyte - wprowadzając tym samym zamęt. Nadmiarowe generowanie plików w takich narzędziach wynika często z potrzeby obsłużenia wielu przypadków granicznych i zapewnienia zadowolenia wielu użytkowników. Jest to realne zagrożenie, mogące negatywnie wpłynąć na efektywność programu, a także na jego jakość. Jednak o ile do generowania użyty zostanie program dobrze przetestowany i powszechnie używany, zazwyczaj nie ma się czego obawiać. Interaktywność takich rozwiązań pozwala trafnie dobrać elementy do swoich potrzeb, unikając nadmiarowości. Korzyści wynikające z automatyzacji pisania kodu z pewnością przewyższają płynące z niej ryzyko.

## 1.2. Yeoman – historia frameworku i jego rozwój

### 1.2.1. Historia frameworku

Yeoman został po raz pierwszy przedstawiony na konferencji Google I/O w 2012 roku. W swoim początku był to zintegrowany zestaw dobrze współdziałających narzędzi, upraszczając tryb pracy nad tworzeniem aplikacji internetowych. Pozwalał na utrzymywanie aktualnych wersji zależności aplikacji, generowanie kodu *boilerplate*[[5]](#footnote-5), udostępniał płynny proces budowania programu z trybem automatycznego odświeżania wprowadzanych zmian. Został bardzo dobrze przyjęty, od razu zostając jedną z dominujących technologii w zakresie generowania kodu. Jest projektem *open* source, zainicjowanym przez Addy’ego Osmani, inżyniera firmy Google. Kod źródłowy projektu jest dostępny na oficjalnym repozytorium w serwisie GitHub.com[[6]](#footnote-6).

Przez okres kilku lat, z pomocą wielu programistów projekt ewoluował. Obecnie skupia się głównie na ekosystemie generatorów, nadal jednak udostępniając swoje pierwotne funkcjonalności. Aktualnie na oficjalnej stronie projektu[4] dostępnych jest publicznie ponad 6000 generatorów, pomagających wystartować z projektem w ulubionej technologii. *Yo,*narzędzie potrzebne do uruchamiania generatorów zostało dotychczas pobrane z repozytorium *npm* ponad 3 miliony razy, w tym ponad 100 000 razy w miesiącu maj 2017.

### 1.2.2. Początki frameworku i podstawowe funkcjonalności

Yeoman został zainspirowany używanym często przez programistów języka Ruby systemem generatorów frameworku *Ruby on Rails*, użytecznym podczas tworzenia aplikacji sieciowych. Wprowadził warstwę, która pozwala programistom na tworzenie generatorów niezależnych od docelowej technologii. Po uprzednim zidentyfikowaniu powtarzających się elementów kodu źródłowego aplikacji, programista może w prosty sposób napisać własny generator. Wiele takich elementów zostało już zidentyfikowanych, co zaowocowało dostępnością implementacji dla praktycznie każdej popularnej dziś technologii.

Rys. . Logo frameworku Yeoman

Generatory potrafią stworzyć np. podstawowe szablony HTML, szkice skryptów języka JavaScript, jak i kompletne projekty łączące w sobie wiele technologii i rozwiązań, na przykład aplikację internetową z interfejsem użytkownika gotową do uruchomienia lokalnie lub w chmurze. Na szczególną uwagę zasługuje możliwość wybierania w interaktywny sposób potrzebnych elementów, często implementowana w generatorach. W zależności od potrzeb, wygenerowana aplikacja może być tylko przykładowym szkieletem lub w pełni funkcjonalną implementacją. Z racji, że większość udostępnianych przez programistów generatorów jest dostępna na bazie oprogramowania *open source* na odpowiedniej licencji, użytkownik może w prosty sposób modyfikować działanie używanego generatora na swoje potrzeby.

### 1.2.3. Korzyści płynące z używania narzędzia Yeoman

Dobrze przygotowany generator potrafi błyskawicznie na podstawie interakcji z użytkownikiem utworzyć kompletny, kompilujący się i gotowy do uruchomienia program. Jednym z głównych założeń, które programista powinien mieć na uwadze pisząc własny generator jest to, aby nie powodował on w docelowym programie żadnych usterek i nie psuł jego struktury. Zawsze przed momentem, w którym pliki są zmieniane lub tworzone, użytkownik może dokładnie przejrzeć zmiany jakie zajdą w jego programie. Używanie generatorów w kompatybilnych projektach jest z reguły bezpieczne i nie powinno powodować żadnych negatywnych, nieoczekiwanych efektów.

Tworząc pliki w jasno określony sposób, Yeoman pozwala zadbać o dobrą strukturę katalogów i podział projektu na moduły, w pewien sposób wymuszając na programiście dobre praktyki i przyspieszając pracę z projektem. Generator do odpowiedniego działania musi wiedzieć na przykład, gdzie spodziewać się plików konfiguracyjnych, gdzie znajdują się pliki klas i jak wygląda struktura programu. Takie warunki jego działania, pomagają bez przerwy dbać o dobrą strukturę projektu, chociażby dla zachowania samej jego kompatybilności z przydatnym generatorem.

Yeoman to nie tylko szybki start z projektem i dobre praktyki, ale także dynamiczne implementowanie konkretnych funkcjonalności. Wiele zadań w toku pracy nad oprogramowaniem jest powtarzalnych, dodawane kolejno nowe moduły często są kopią poprzednich implementacji, dotycząc tylko innych danych. Przykładowo w aplikacjach internetowych zapisujących obiekty w bazie danych, dla każdej nowej encji (klasy-modelu) zwykle implementuje się operacje dodawania, usuwania, edycji czy pobierania listy obiektów. Dodawanie takich funkcjonalności jest czynnością wysoce powtarzalną i rzadko różniącą się czymś więcej, niż na przykład nazwami używanych klas lub miejscem, w którym znajdują się pliki źródłowe. Jest to idealne zastosowanie dla generatora, który mógłby przyjmować jako dane wejściowe nazwę wspomnianej encji i jej podstawowe atrybuty, generując:

* Klasę modelu encji,
* klasę DAO (*Data Access Object)* z podstawowymi metodami,
* serwis będący miejscem na logikę biznesową,
* kontroler stanowiący interfejs do interakcji z zewnętrznymi systemami.

Zgodne z konwencją i odpowiednio nazwane wygenerowane klasy umieszczone zostałyby w odpowiednich katalogach oraz zapewnione byłoby ich odpowiednie współdziałanie. Używanie generatora zostawia mało miejsca na pomyłkę programisty, z której mogłyby wyniknąć problemy mające wpływ np. na kompilowanie się projektu. Zaoszczędzony czas pozwala programiście nie skupiać się na powtarzalnych czynnościach i od razu implementować logikę biznesową, bez obaw o tworzenie w odpowiedniej, działającej struktury. Większość powtarzalnych czynności da się zaimplementować w generatorze, co prawdopodobnie zostało już zrealizowane w jednym z publicznie dostępnych generatorów. Takie nowoczesne podejście pozwala na uniknięcie błędów i znacznie przyspiesza pracę.

Częstym elementem pracy, o którym programistom zdarza się zapominać podczas tworzenia oprogramowania jest pisanie testów jednostkowych zapewniających o poprawnym działaniu programu. Zazwyczaj sytuacja taka wynika z niejasnej, nie działającej odpowiednio lub nieistniejącej struktury środowiska testowego w projekcie. W przypadku używania generatorów do tworzenia konkretnych funkcjonalności, realnym scenariuszem staje się również generowanie podstawowych testów ilustrujących działanie programu, a także nawet automatyczne uruchamianie ich od razu po wygenerowaniu kodu, zapewniając o jego jakości.

### 1.2.4. Przypadki użycia generatora

Yeoman jest narzędziem efektywnym w praktycznie każdym używanym dziś języku programowania. Generowany kod może mieć dowolną formę, co powoduje, że Yeoman znajduje dziś wiele różnych zastosowań:

* tworzenie nowego projektu,
* tworzenie nowych elementów istniejącego projektu,
* tworzenie modułów i pakietów,
* prototypowanie i przygotowywanie wersji demo aplikacji,
* wymuszanie na programistach dobrych praktyk,
* generowanie przykładowego kodu w do użytku w poradnikach,
* tworzenie prezentacji multimedialnych.

Sam framework, bez odpowiedniej implementacji nie podejmuje żadnych decyzji odnośnie generowanego kodu. Fakt, że wszystko zależne jest od programisty pozwala na elastyczność i tworzenie generatorów nie związanych nawet ściśle z programowaniem. Ciekawym przypadkiem użycia jest stworzony przez Adriana Bateman (Program Manager w zespole *Microsoft Edge Web Platform*) generator specyfikacji technicznej opartej na bibliotece *ReSpec*, przyspieszający tworzenie nowych dokumentów technicznych [5]. W swoim początku, Yeoman promowany był jako narzędzie mające na celu pomóc w pracy głownie programistom front‑end, jednak w miarę upływu czasu znalazł zastosowanie w większej ilości dziedzin programowania. Używany jest na przykład w wielu poradnikach on-line, pozwalając autorom przekazać w łatwy sposób swoim czytelnikom potrzebne pliki, zachęcając do interaktywnej nauki.

### 1.2.5**.** Przykładowe generatory

Generatory Yeoman zwykle tworzone są z myślą o szerszym gronie użytkowników, dlatego na rynku dostępnych jest wiele ciekawych implementacji, powszechnie używanych, dobrze przetestowanych i ciągle rozwijanych:

* JHipster - bez wątpienia najbardziej popularny, bardzo dobrze realizujący założenia dobrze napisanego generatora i pokazujący jak bardzo użyteczny może być Yeoman. Generuje w pełni funkcjonalną aplikację internetową w języku Java (Spring Boot + Angular), zarówno w architekturze monolitycznej[[7]](#footnote-7), jak i mikro serwisów[[8]](#footnote-8). Pozwala na stworzenie samego projektu, jak i ciągłą z nim pracę przy użyciu podgeneratorów. Używany jest codziennie w wielu renomowanych firmach, jak Adobe, Bosch, Google, Siemens czy Orange. Oferuje szereg dodatkowych narzędzi, jak np. wtyczki do IDE czy podgeneratory konfiguracji chmurowej. Posiada dużą różnorodność w wyborze używanych technologii oraz duże wsparcie społeczeństwa programistycznego i wielu fanów.

Logo JHipster

* Angular – generator aplikacji jednostronnych[[9]](#footnote-9) Angular, pozwalający na utworzenie kodu w języku JavaScript, CoffeeScript lub TypeScript. Posiada szereg podgeneratorów dla popularnych elementów aplikacji AngularJS, jak kontrolery, widoki, dyrektywy i filtry. Utworzony został przez zespół dostarczający rozwijający framework Yeoman. Jest często używany w dostępnych w sieci poradnikach prezentujących framework Angular.
* generator – generator generatorów Yeoman. Przygotowuje strukturę plików potrzebną do sprawnego rozpoczęcia implementacji własnego generatora, tworzy pliki zwykle dodawane w przypadku pracy z repozytorium kodu i Node.js.
* release – ułatwiający publikację aplikacji w repozytorium NPM, przeglądający zmiany dokonane od ostatniej publikacji, tworzący na ich podstawie dziennik zmian i pomagający w utrzymaniu poprawnego wersjonowania aplikacji.

# 2**.** Środowisko Yeoman

Proces instalacji środowiska, którego główną ideą jest przyspieszanie i ułatwianie pracy, musi być prosty i jednoznaczny. Oczywistym wyborem docelowej platformy uruchomieniowej dla Yeoman stał się więc realizujący te założenia Node.js, wraz z intuicyjnym menadżerem do zarządzania pakietami, jakim jest NPM. W zespole technologii udostępnianych w ramach tej platformy dostępnych jest wiele narzędzi idealnie pasujących do celu, jaki stawia sobie Yeoman – automatyzacji.

## 2.1. Stosowane technologie

### 2.1.1. Node.js

Środowisko uruchomieniowe utworzone na podstawie silnika JavaScript V8 dostarczonego przez Google. Jest to platforma sterowana zdarzeniami[[10]](#footnote-10), zaprojektowana w celu tworzenia wysoce skalowalnych aplikacji internetowych. Nieblokujący model I/O [6] pozwala na kontynuację wykonywania programu, obsługując zdarzenia poprzez zwrotne wywołania funkcji (ang. c*allback).* Posiada prosty system modułowości, pozwalający na ładowanie i używanie funkcjonalności z osobnych modułów.

const fs = require('fs');

fs.readFile('/file.md', (err, data) => {

if (err) {

throw err;

}

this.log(data);

});

*Listing 1. Przykład asynchronicznej, nieblokującej obsługi otwierania pliku używając modułu fs*

### 2.1.2. npm

Repozytorium pakietów środowiska Node.js, a zarazem narzędzie wiersza poleceń umożliwiające ich pobieranie i instalację. Pozwala na automatyzację zarządzania zależnościami aplikacji poprzez deklarację używanych modułów w pliku *package.json.* W repozytorium NPM znajdują się setki tysięcy różnych pakietów gotowych do pobrania.  
NPM umożliwia pobieranie konkretnych wersji dostępnych pakietów i pozwala w łatwy sposób publikować swoje moduły innym. Jest używany zarówno w procesie tworzenia własnego generatora Yeoman, jak i do pobrania zależności utworzonej aplikacji po jej wygenerowaniu

### 2.1.3. Gulp

Narzędzie do automatyzacji powtarzających się czynności w programowaniu. Pozwala na definiowanie zadań posiadających poszczególne kroki, uruchamianych z linii wiersza poleceń. Będąc kompatybilnym z wieloma zewnętrznymi bibliotekami znajduje bardzo wiele zastosowań.

Jest narzędziem elastycznym i łatwo konfigurowalnym dla optymalnego procesu pracy z oprogramowaniem. Po wygenerowaniu kodu przez Yeoman, odpowiednie zadanie Gulp może na przykład przygotowywać serwer aplikacji, kompilować jej kod źródłowy, minifikować[[11]](#footnote-11) i umieszczać go w odpowiednich katalogach w zoptymalizowanej formie, przy okazji zapewniając automatyczne odświeżanie projektu w przeglądarce po wykryciu zmian.

### 2.1.4. Bower

Narzędzie do automatycznego zarządzania zależnościami aplikacji. Umożliwia zarówno pobieranie wybranych bibliotek używając linii wiersza poleceń, jak i deklarację oraz automatyczne zarządzanie nimi przy użyciu pliku *bower.json*. Używany jest głównie do pobierania zależności front-endowych, jak jQuery lub Backbone. W połączeniu z resztą narzędzi używanych przez Yeoman, umożliwia minimalną pracę przy instalacji projektu po jego wygenerowaniu oraz automatyczne zarządzanie jego zależnościami.

### 2.1.5. Inquirer.js

Używany w toku pracy generatora, w fazie odpytywania użytkownika. Udostępnia kilka podstawowych typów pytań i pozwala na warunkowe ich wywoływanie. Umożliwia podpowiadanie odpowiedzi, a także ich walidację i filtrowanie. Używany jest również w fazie zapisywania plików na dysku, pozwalając użytkownikowi zobaczyć i interaktywnie zatwierdzić zmiany, jakie zajdą w jego systemie.

## 2.2. Sposób działania generatora Yeoman

### 2.2.1. Metody jako elementy bazowe

Funkcje języka JavaScript uznawane są przez generator Yeoman jako podstawowe elementy, z których korzysta programista definiując jego działanie. Rozszerzając udostępniany prototyp *Generator* o swoje metody, definiować możńa różne działania, z których każde uznawane jest za oddzielne zadanie do wykonania dla generatora i uruchamiane w sekwencji. Pozwala to na poukładanie toku pracy programu w odpowiedni sposób i kontrolę nad nim. Niektóre z nazw metod są zarezerwowane, tworząc pętlę programową. Istnieje również możliwość zadeklarowania metod prywatnych, nie wykonywanych automatycznie przez framework. Takie metody mogą służyć dodatkowej organizacji kodu i definiuje się je na trzy sposoby:

* poprzez prefiks podkreślenia w nazwie metody *\_privateMethod*,

\_privateMethod: function () {

this.log('This method wont run automatically');

},

* poprzez metody związane z konkretną instancją,

install: function () {

this.instanceMethod = function () {

this.log('This method wont run automatically');

};

},

* poprzez rozszerzanie generatora nadrzędnego.

class GenAPI extends Generator {

helperMethod() {

this.log('This method wont run automatically');

}

}

Użycie podczas tworzenia własnej implementacji generatora metod prywatnych, pozwala na zachowanie czystości kodu i odpowiedni podział na funkcjonalności. W ten sposób w pętli głównej programu znajdą się tylko metody wykonywane po kolei, jasno określając przebieg pracy. Pozwala to dokładnie zrozumieć proces zachodzący podczas generowania kodu. Jedne z metod mogą odpowiadać na przykład za odpytywanie użytkownika o dane, inne za odpowiednie wnioskowanie na podstawie jego odpowiedzi, kolejne zaś za weryfikację poprawności wprowadzonych danych lub użycie utworzonych obiektów podczas faktycznego generowania kodu. Yeoman organizuje takie metody-funkcjonalności w kolejkę, tworząc podstawową pętlę programową.

### 2.2.2**.** Pętla programowa

Wywoływanie metod generatora po kolei sprawdza się w większości przypadków, niekiedy jednak aby zapewnić niezawodność wymaganym jest, aby jeszcze dokładniej określić porządek ich wywoływania. Umożliwione zostało to przez framework poprzez udostępnienie pętli programowej, używając modułu *Grouped-queue[[12]](#footnote-12)* i specjalnych, prototypowych metod o zarezerwowanych nazwach. Dopóki zadeklarowana przez użytkownika metoda ma nazwę odpowiadającą jednej z zarezerwowanych, jej wykonanie zostanie umieszczone na specjalnej kolejce. Ponadto, jeśli metoda nie odpowiada żadnej z priorytetowych nazw, zostanie dodana do grupy wykonania domyślnego *default*.

Poniżej przedstawione zostały nazwy priorytetowych metod zarezerwowanych, w kolejności ich wykonania przez framework:

1. initializing – faza, w której w programie można zainicjalizować potrzebne zmienne, załadować konfigurację lub np. sprawdzić kontekst wykonania programu,
2. prompting – odpytywanie użytkownika generatora, zapisując jego odpowiedzi w zmiennych programu do późniejszego użycia,
3. configuring – miejsce na logikę, w którym na podstawie wcześniej zadeklarowanych odpowiedzi użytkownika można odpowiednio skonfigurować zmienne programu, na przykład operacje na napisach lub dostosowanie zmiennych do docelowego języka programowania,
4. default – umieszczane są tutaj wszystkie funkcje, których nazwy nie pasują do priorytetowych i wykonywane po kolei,
5. writing – faza, w której program faktycznie zapisuje pliki na dysku, kopiując je z szablonów a także operując na nich przy użyciu zmiennych zadeklarowanych wcześniej,
6. conflicts – używana wewnętrznie przez framework, ale jednak dostępna do rozszerzenia przez programistę faza, w której rozwiązywane są konflikty na plikach już istniejących. Generator odpytuje użytkownika o wprowadzane zmiany, pokazując dokładnie różnice i pozwalając na zastanowienie się,
7. install – etap, w którym powinno umieszczać się kod odpowiedzialny za instalację wygenerowanego projektu, uruchamianie zadań związanych z pobieraniem zależności i inne, zwykle z użyciem zewnętrznych bibliotek (npm, bower, grunt, gulp),
8. end – ostatni element pracy generatora, w którym można np. posprzątać po wykonywanych zadaniach, wyświetlić wiadomość pożegnalną.

Stworzenie generatora zgodnie z zasadami, stosując odpowiednio zarezerwowane nazwy metod priorytetowych, pozwala na odpowiednie działanie generatorów w momencie ich zagnieżdzania.

### 2.2.3**.** Generatory zagnieżdżone

Yeoman udostępnia przy użyciu mechanizmu zagnieżdzania (ang. *composability*) możliwość uruchamiania jednych generatorów podczas działania drugich*.* Operacja ta polega na wywołaniu metody *generator.composeWith()* uruchamiającej pożądany generator. Pobiera ona dwa parametry, pierwszym z nich jest ścieżka do uruchamianego generatora, a drugim obiekt *options*, zawierający argumenty przekazywane dalej. Możliwe jest przekazanie do komponowanego generatora dowolnych parametrów potrzebnych dla jego optymalnej pracy (np. nazwa generowanego modułu, używane zmienne). Umożliwia to tworzenie dowolnie złożonych implementacji, realizujących różne skomplikowane założenia.

this.composeWith('common', {

options: {

'skip-messages': false,

jshintrc: true,

gitignore: true,

}

});

Listing Przykład wywołania generatora z parametrami

Użycie zagnieżdżonych generatorów pozwala na tworzenie większej całości z mniejszych części. Zachowanie konwencji odpowiedniego nazewnictwa kluczowych metod z pętli programowej pozwala, aby framework zadbał o poprawne wykonanie powierzonej pracy. Możliwe jest również wywoływanie generatorów dostępnych publicznie w repozytorium NPM, przekazując jako ścieżkę generatora wywołanie funkcji pobierającej zależność: *require.resolve(‘generator’)*. Aby używać generatora dostępnego w NPM, należy zadeklarować jego użycie w pliku *package.json* generatora głównego jako *peerDependencies*.

### 2.2.4**.** Zdarzenia synchroniczne

Z faktu, iż Yeoman oparty jest na środowisku działającym z natury asynchronicznie, obsługa zdarzeń synchronicznych może powadzić do pewnych problemów. W toku pracy czasami wynika sytuacja, w której trzeba dokonać pauzy w działaniu generatora. Jedną z takich sytuacji jest odpytywanie użytkownika w sekwencji o kilka różnych rzeczy. Pytania powinny pojawiać się po kolei, czekając na udzielenie odpowiedzi – nie mogą wyświetlać się wszystkie naraz. Odpowiedzi na jedne pytania mogą mieć wpływ na pojawianie się lub treść kolejnych. Jest to działanie z synchroniczne. Istnieje kilka sposobów realizacji takiego scenariusza, jednym z nich może być używanie *promise[[13]](#footnote-13),* elementu nowoczesnego języka JavaScriptwprowadzonego w ramach standardu ECMAScript *2015[[14]](#footnote-14)* lub używanie funkcji *this.async()* .

Takie wywołanie powoduje zwrócenie funkcji, którą można wywołać ponownie po zakończeniu działania. Funkcja działająca asynchronicznie, tuż przed zakończeniem swojego działania wywołuje przekazaną funkcję *done()* oznaczając zakończenie operacji. Kontrola zwracana jest wtedy do funkcji wywołującej, która wznawia swoje działanie. Jeśli do wywołania funkcji kończącej działanie asynchroniczne przekazany zostanie parametr błędu, pętla programowa zatrzyma się powodując wyjątek.

function synchronousFunction() {

var done = this.async();

asynchronousOperation(done);

}

*Listing 3 Wywołanie operacji synchronicznej w funkcji asynchronicznej*

### 2.2.5**.** Interakcja z systemem plików

Działanie na plikach zrealizowane jest na idei, że zawsze dostępne są dwa konteksty operacji na plikach. Odpowiada to dwóm katalogom na dysku, z których jeden służy do zapisywania, a drugi do odczytywania danych. Pierwszy kontekst jest kontekstem *docelowym*. Jest to katalog, w którym zazwyczaj uruchomiony został generator i w którym zapisywane będą pliki. Istnieje również możliwość zmienienia kontekstu zapisywania, poprzez zadeklarowanie ścieżki do katalogu docelowego, umieszczając w nim plik *.yo-rc.json*. Jest to plik konfiguracyjny Yo, który pomaga w zachowaniu konsekwentności w uruchamianiu generatorów. Z racji, iż Yeoman może być używany do tworzenia bardzo małych, nowych funkcjonalności w istniejących projektach, istnienie tego pliku pomaga zlokalizować ścieżkę, w której tworzone mają być pliki. Ścieżkę można w generatorze pobrać przy użyciu funkcji *this.destinationRoot()*.

Drugim kontekstem interakcji z systemem plików jest kontekst *szablonów*. Domyślnie jest to katalog o ścieżce */templates/*, w którym przechowywane są szablony, na podstawie których tworzone są pliki. Analogicznie do kontekstu zapisywania można zmienić katalog, z którego pobierane są szablony przy użyciu funkcji *this.templatePath()*.

Yeoman udostępnia wewnątrz pamięciowy system, w którym pliki są przygotowywane, a następnie użytkownik ma możliwość potwierdzenia zmian w każdym z nich. Może dokładnie obejrzeć zmiany, odrzucić niektóre z nich lub po prostu zaakceptować wszystkie. Po zakończeniu pracy generatora i zaakceptowaniu wprowadzanych zmian, pliki zapisywane są asynchronicznie na dysku. Takie podejście umożliwia uniknięcie niepotrzebnych konfliktów i błędów. Podczas kopiowania plików możliwe jest użycie mechanizmu szablonowania o składni opartej na EJS, pozwalającego dynamicznie zmieniać treść. Możliwe jest także edytowanie istniejących plików, jednak należy wtedy zachować szczególną ostrożność i często wcześniej odpowiednio przygotować pliki, na przykład dodając odpowiednie linijki rozpoznawane przez program. Dodanie specyficznych linii w kluczowych miejscach pozwala na zamianę ich na kod z szablonu, rozszerzając istniejące pliki o dodatkowe funkcjonalności. Oczywistym przypadkiem użycia takiego mechanizmu jest rozszerzanie klas o kolejne pola i metody, np. w celu ustalenia relacji z innymi.

# 3**.** Własna implementacja

## 3.1. Generator i pod-generator

Aplikacja zaimplementowana w ramach przedstawianej pracy jest zespołem dwóch generatorów Yeoman. Podstawową jej funkcjonalnością jest generowanie prostego serwera opartego na frameworku Spring w języku Java, udostępniającego sieciowy interfejs REST API[[15]](#footnote-15). Pierwszy z nich, nazwany *genapi,* służy do utworzenia działającego szkieletu aplikacji udostępniającej taki serwer, a drugi, będący podgeneratorem *genapi:entity*, umożliwia uzupełnienie uprzednio utworzonej aplikacji o nowe klasy:

* model encji,
* repozytorium dostępu do bazy danych,
* kontroler REST.

Tworzone pliki są trzema podstawowymi elementami wymaganymi do poprawnego działania interfejsu REST współpracującego z bazą danych [7]. Wygenerowane klasy umożliwiają operację dodawania, usuwania, edycji i pobierania obiektów encji z bazy danych przy użyciu zapytań protokołu HTTP. Istnieje możliwość wielokrotnego wywołania podgeneratora w celu rozszerzania aplikacji o kolejne funkcjonalności (modele encji wraz z obsługującymi je klasami). Po kilku kliknięciach i uruchomieniu programu w środowisku deweloperskim, dostępna staje się aplikacja potrafiąca otrzymywać zapytania HTTP, zapisywać i odczytywać informacje z bazy danych oraz zwracać je użytkownikowi. Czasami przy implementowaniu aplikacji klienckich przydatna jest dostępność serwera, przy użyciu którego można byłoby przetestować implementowane funkcjonalności. Umożliwienie szybkiego stworzenia rozszerzalnego szkieletu aplikacji zdolnej obsługiwać zapytania REST było głównym celem napisania generatora *genapi.* Może być on również przydatny w nauce, gdyż tworzona jest minimalnie działająca, nie posiadająca zbędnego kodu źródłowego wersja, pozwalająca zrozumieć istotę działania takiej aplikacji.

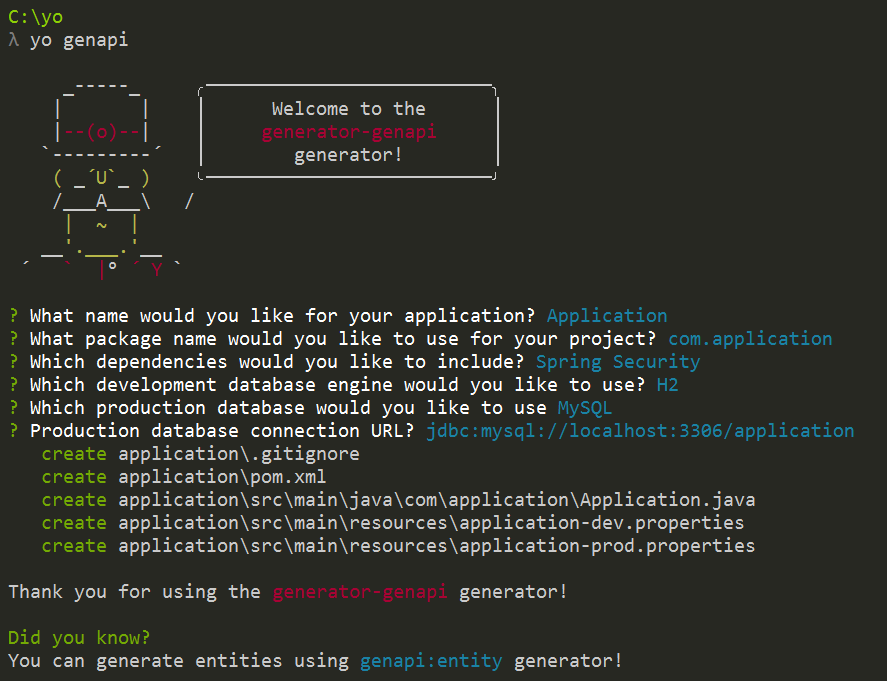
Implementacja generatorów jest dość podstawowa, ale dobrze pokazuje powody, dla których warto jest wdrożyć w swój tok pracy elementy automatyzacji pisania kodu.

### 3.1.1**.** Generator główny - genapi

#### 3.1.1.1**.** Funkcjonalności

Generator główny służy utworzeniu podstawowych plików potrzebnych do uruchomienia aplikacji, jego funkcjonalności najlepiej przedstawiają pytania stawiane kolejno użytkownikowi w toku jego pracy. Są to pytania o podstawowe elementy aplikacji (*Rys. 4*):

* nazwę aplikacji,
* nazwę pakietu Java,
* zależności aplikacji z listy kilku polecanych,
* silnik bazy danych w profilu deweloperskim,
* silnik bazy danych w profilu produkcyjnym,
* adres dostępu do bazy danych.



*Rys. 4. Ekran wyświetlany użytkownikowi po udzieleniu odpowiedzi na wszystkie pytania*

Generator pozostawia użytkownikowi decyzję co do zawartości aplikacji, interaktywnie odpytując go o jej elementy składowe. Odpowiedzi użytkownika na jedne z pytań mają wpływ na kształt innych. Przykładem dynamicznego dostosowywania pytań jest sytuacja, w której wpisana w odpowiedzi nazwa aplikacji pojawia się w podpowiadanej nazwie pakietu. Także jeżeli zadeklarujemy użycie zewnętrznej bazy danych MySQL, generator pytając o adres dostępu do niej, podpowie użytkownikowi adresem zaczynającym się od „*jdbc:mysql://localhost:3306/”,* podpowiadanym również podczas domyślnej instalacji tego typu bazy. Taki przepływ pracy, w którym odpowiedzi użytkownika mają wpływ na pojawiające się pytania, pozwala na oszczędzenie dodatkowej pracy potrzebnej na konfigurację projektu po jego wygenerowaniu. Niektóre z pytań mogą w ogóle nie zostać zadane, jeśli poprzednie odpowiedzi wykluczają możliwość użycia związanych z nimi funkcjonalności. Dla przykładu, jeśli jako silnik bazy danych wybrana zostanie wewnątrz pamięciowa baza H2, niepotrzebne jest odpytywanie użytkownika o adres dostępu do niej. Do poprawnego działania takiego typu bazy danych wystarczająca jest tylko jej deklaracja w zależnościach aplikacji, co zostaje zapewnione już po samym jej wybraniu.

Po zakończeniu pracy generatora wyświetlana jest informacja o możliwości uruchomienia pod-generatora w celu rozszerzenia funkcjonalności aplikacji. Generator główny ma mniejszą rolę niż poboczny, ponieważ używany jest tylko raz podczas tworzenia projektu, nie posiada zbyt dużej ilości funkcjonalności czy logiki.

### 3.2.2 Generowane pliki

Wynikiem pracy generatora głównego jest utworzenie na dysku kilku plików źródłowych i konfiguracyjnych aplikacji. Taki program można od razu po skompilowaniu uruchomić. Lista plików tworzonych przez generator główny*:*

* Application.java – klasa główna aplikacji,
* pom.xml – plik z deklaracjami zależności aplikacji,
* application-dev.properties – plik konfiguracyjny profilu deweloperskiego,
* application-prod.properties – plik konfiguracyjny profilu produkcyjnego,
* .gitignore – plik często dołączany w przypadku pracy z repozytorium kodu *git,* pozwalający na ignorowanie niektórych zmian.

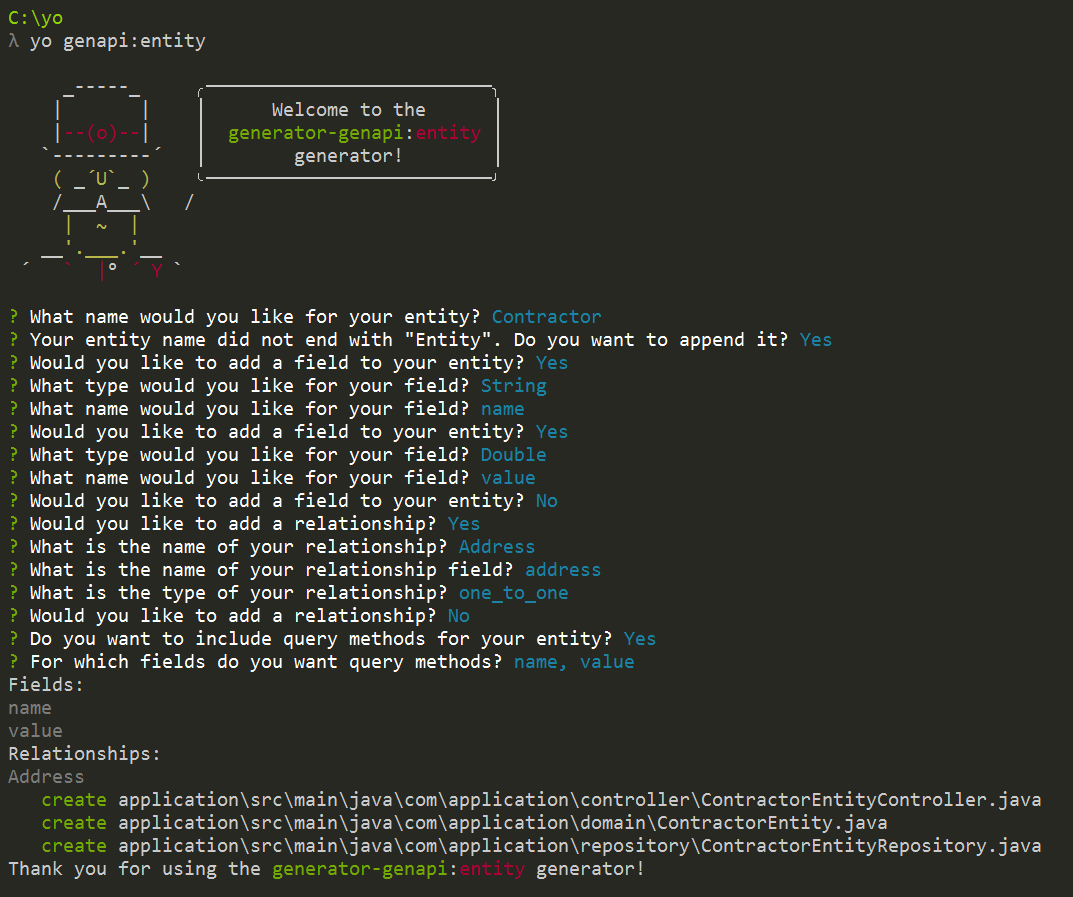
### 3.1.2**.** Generator poboczny - genapi:entity

*Genapi:entity* jest podgeneratorem służącym do rozszerzania aplikacji o kolejne modele encji. Fakt, iż jest to generator poboczny może mylnie sugerować, że ma mniejsze znaczenie. W rzeczywistościskupia on większość funkcjonalności aplikacji i można go używać wielokrotnie. Uruchomiony z poziomu katalogu z uprzednio wygenerowanym projektem, sam wywnioskuje, gdzie powinny zostać dodane nowe pliki.

#### 3.1.2.1. Funkcjonalności

Przy użyciu *genapi:*entity użytkownik ma możliwość dodania do projektu plików encji, kontrolera stanowiącego interfejs sieciowy i repozytorium dostępu do bazy danych. Podobnie jak w przypadku generatora głównego, możliwości najlepiej zilustrowane są na przykładzie pytań zadawanych w toku jego pracy przedstawione na rys. 5. Są to pytania pozwalające na wskazanie:

* nazwy encji,
* typów i nazw atrybutów,
* typów i nazw relacji,
* atrybutów, na podstawie których można będzie wyszukiwać encje w bazie danych.

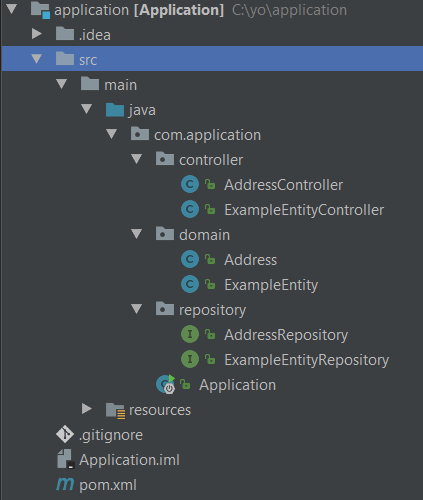


*Rys. 5. Tok pracy podgeneratora genapi:entity*

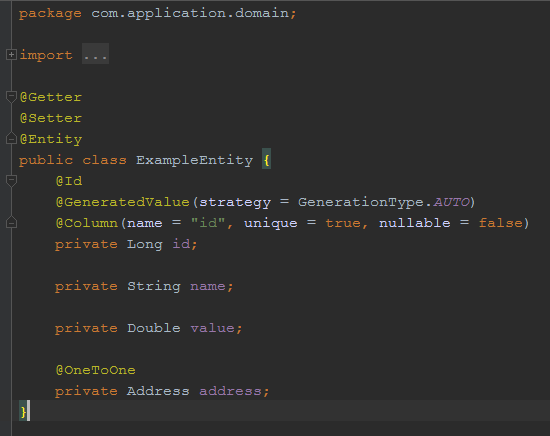
#### 3.1.2.2. Generowane pliki

Plików tworzonych w ramach pracy generatora encji jest niewiele, lecz dodają one do programu faktyczną wartość. Każdy z nich tworzony jest dynamicznie na podstawie szablonów używających odpowiedzi użytkownika, co zapewnia ich bezproblemową współpracę. Dla przykładu, jeśli zdecydujemy się na utworzenie metod pozwalających na wyszukiwanie obiektów w bazie danych według atrybutów encji, odpowiednie metody utworzą się zarówno w interfejsie repozytorium, jak i w klasie kontrolera REST, pozwalając na dostęp do danych poprzez interfejs sieciowy przy użyciu zapytań HTTP.

Po wygenerowaniu dwóch przykładowych encji ExampleEntity oraz Address, katalogi aplikacji wyglądają jak na rys. 6, a kod źródłowy wygenerowanej przykładowej klasy ExampleEntity został przedstawiony na rys. 7.



*Rys. 6. Katalogi aplikacji po wygenerowaniu dwóch encji*

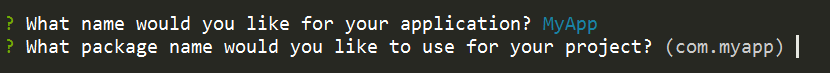


*Rys. 7. Wygenerowana klasa ExampleEntity*

## 3.2. Interakcja z użytkownikiem

Interakcja z użytkownikiem jest główną fazą działania generatora. Jest zrealizowana przy użyciu frameworku Inquirer.js [8], pozwalającego definiować zadawane pytania, obrabiać i walidować odpowiedzi użytkownika, oraz definiować warunki ich wyświetlania. Programista może zdefiniować zapytanie poprzez skonstruowanie i przekazanie do odpowiedniej funkcji specjalnego obiektu, zawierającego jako właściwości wartości kluczowe zdefiniowane przez framework. Zarezerwowanymi słowami kluczowymi rozpoznawanymi przez Inquirer.js są:

* type – pozwala na zdefiniowanie typu pytania, np. *input, list, checkbox*,
* name – nazwa pytania, używana jako odwołanie przy pobieraniu odpowiedzi użytkownika,
* message – wiadomość wyświetlana użytkownikowi,
* choices – lista możliwych do wybrania przez użytkownika opcji,
* default – wyświetlana wartość domyślna,
* validate – definicja funkcji używanej w celu walidacji odpowiedzi, pobierającej dane wpisane przez użytkownika i zwracającej wartość logiczną
* filter – definicja funkcji umożliwiającej zmianę odpowiedzi użytkownika po jej wpisaniu
* when – definicja funkcji lub wartość logiczna umożliwiająca warunkowe wyświetlanie pytania

Jako przykład posłużyć może pytanie o nazwę pakietu Java przedstawione na rys. 8, którego kod źródłowy przedstawiony jest na listingu 4.

*Rysunek 8. Pytanie o nazwę pakietu*

function askForPackageName() {

var done = this.async();

var prompts = [{

type: 'input',

name: 'package',

message: 'What package name would you like to use for your project?',

default: () => 'com.' + \_.toLower(this.appName),

filter: (answer) => \_.toLower(answer)

}];

this.prompt(prompts).then((answers) => {

this.package = answers.package;

done();

});

}

*Listing 4. Funkcja odpytująca o nazwę pakietu Java*

Na początku przy użyciu funkcji *this.async()* przypisywana jest zmienna *done* pozwalająca na synchroniczne wykonanie zapytania. Do czasu wykonania funkcji *done()*, bieg pracy programu został spauzowany, co pozwala poczekać na odpowiedź użytkownika. Obiekt pytania posiada szczególne właściwości:

* typ inputpozwalający wpisać użytkownikowi odpowiedź,
* nazwę package używaną do pobrania odpowiedzi i zapisania jest w kontekście,
* przyjazną wiadomość dla użytkownika informującą o oczekiwanej odpowiedzi,
* funkcję default podpowiadającą nazwę aplikacji zapisaną małymi literami z prefiksem,
* funkcję filter zmieniającą odpowiedź użytkownika na małe litery, jak zwykle nazywane są pakiety w języku Java.

Tak przygotowany obiekt przekazany zostaje do wywołania funkcji *this.prompt()* udostępnianej przez framework, a później w wywołaniu zwrotnym odpowiedź zapisywana może być w kontekście generatora. Po wywołaniu funkcji *done()* generator wznawia pracę, przechodząc do kolejnego pytania. W taki sposób zdefiniowane są wszystkie pytania zadawane użytkownikowi, a jego odpowiedzi zapisywane w kontekście. Kontekst ten pod postacią zmiennej thisprzekazywany jest później funkcjom zapisującym pliki na dysku, udostępniając im tym samym odpowiedzi użytkownika. Zmienne zapisane w ten sposób często wymagają obróbki w celu przystosowania ich do docelowego języka programowania. Można to zrealizować zarówno w funkcji filter obiektu pytania, jak i później – nawet po zakończeniu fazy odpytywania, w fazie configuring pętli programowej generatora*.* Istnieje również możliwość zapisywania odpowiedzi w pliku konfiguracyjnym na dysku, np. w celu przekazania ich do generatorów zagnieżdżonych. Można to zrobić przy użyciu funkcji *this.config.set(‘name’, value)*, jak to pokazane jest na Listingu 5.

this.config.set('packagePath', this.packagePath);

this.config.set('package', this.package);

this.config.set('lowercaseAppName', this.lowercaseAppName);

{

"generator-genapi": {

"packagePath": "com/myapp",

"package": "com.myapp",

"lowercaseAppName": "myapp"

}

}

*Listing 5. Zapisywanie zmiennych w pliku konfiguracyjnym i ich reprezentacja*

## 3.3 Zapisywanie plików na dysku

Po fazie zadawania pytań użytkownikowi i zapisywaniu jego odpowiedzi w odpowiedniej formie w kontekście generatora, następnym stadium jest zapisanie plików na dysku. Przed rzeczywistą interakcją z systemem plików, wszystkie zmiany dokonywane przez generator ładowane są do pamięci podręcznej. Cała operacja działania z systemem plików zrealizowana jest za pomocą modułu mem-fs-editor udostępnianego przez framework, dostępnego pod zmienną this.fs. Udostępnia on szereg metod umożliwiających operację na plikach, między innymi:

* copy *–* funkcja służąca do kopiowania plików statycznych,
* copyTpl *–* funkcja służąca do kopiowania plików na podstawie wzorca, używając zmiennych w kontekście aplikacji do jego modyfikacji.

O ile niektóre z plików szablonowych kopiowanych przez generator są statyczne, zawartość większości z nich dynamicznie zmienia się w zależności od kontekstu. Moduł operacji na plikach szablonowych używa składni języka *EJS[[16]](#footnote-16)* do osiągnięcia jak najbardziej spersonalizowanej zawartości. Składnia ta opiera się na pewnych znacznikach umieszczanych w plikach, które pozwalają na zawarcie w nich logiki. Znaczniki zawarte w składni opisane są w tabeli 1.

*Tabela 1 Znaczniki składni EJS*

|  |  |
| --- | --- |
| <% | Oznacza skrypt, pozwala na zawarcie warunków logicznych |
| <%= | Wypisuje zdekodowaną[[17]](#footnote-17) wartość do pliku |
| <%- | Wypisuje niedekodowaną wartość do pliku |
| <%# | Oznacza komentarz, nie wykonuje żadnych operacji |
| <%% | Wypisuje wartość „<%” |
| %> | Zwykły znacznik kończący |
| -%> | Znacznik kończący usuwający znak nowej linii |

Zawieranie w plikach warunków logicznych umożliwia dynamiczne tworzenie i warunkowanie pojawiania się niektórych ich elementów. Do modułu wykonującego szablonowanie przekazywany jest kontekst aplikacji, co powoduje, że dostępne są wszystkie zawarte w nim zmienne. W generatorach implementowanych w ramach przedstawianej pracy, szablonowanie wykorzystywane jest głównie przez *genapi:entity* podczas tworzenia klas Java, co przedstawiono na listingu 6. Tworzone są trzy pliki klas, umieszczane w dynamicznie generowanych katalogach i odpowiednio nazywane. Do funkcji *this.fs.copyTpl()* przekazane zostają trzy parametry:

* nazwa pliku z szablonem (według konwencji zaczynająca się od „\_”)
* ścieżka docelowa dla wygenerowanego pliku
* kontekst aplikacji poprzez zmienną *this*

Ścieżka docelowa tworzona jest na podstawie zmiennych pobranych wcześniej z pliku konfiguracyjnego *.yo-rc.json*, tworzonego przez generator. Znajduje się tam nazwa aplikacji zapisana małymi literami, ścieżka pakietu Java (wywnioskowana na podstawie jego nazwy) oraz nazwa encji. Pozwala to na poprawne zapisywanie plików w odpowiednich katalogach, także przy okazji kolejnych uruchomień generatora.

this.fs.copyTpl(

this.templatePath('\_EntityController.java'),

this.destinationPath('./' + this.lowercaseAppName + '/src/main/java/' +

this.packagePath + '/controller/' + this.entityName + 'Controller.java'),

this

);

this.fs.copyTpl(

this.templatePath('\_Entity.java'),

this.destinationPath('./' + this.lowercaseAppName + '/src/main/java/' +

this.packagePath + '/domain/' + this.entityName + '.java'),

this

);

this.fs.copyTpl(

this.templatePath('\_EntityRepository.java'),

this.destinationPath(

'./' + this.lowercaseAppName + '/src/main/java/' +

this.packagePath + '/repository/' + this.entityName + 'Repository.java'),

this

);

*Listing 6. Dynamiczne tworzenie klas Java*

Wykorzystanie mechanizmu szablonowania i jego składni w aplikacji przedstawione zostało na listingu 7, pokazującym kod odpowiadający za szablon repozytorium bazy danych. W pętli definiowane są tymczasowe zmienne camel*,* upperCameloraz type pozwalające na uporządkowanie logiki. Jeśli użytkownik w fazie odpowiadania na pytania wybierze wygenerowanie metod do pobierania obiektów po jego atrybutach, odpowiednie obiekty dodane zostaną do zmiennejqueryMethods, co spowoduje późniejszą iterację po nich w szablonie, generując deklaracje metod pobierających dane w interfejsie.

package <%= package %>.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import <%= package %>.domain.<%= entityName %>;

<% if (queryMethods.length > 0) { %>

import java.util.List;

<% } %>

public interface <%= entityName %>Repository extends JpaRepository<<%= entityName %>, Long> {

<% for (index in queryMethods) {

    var camel = queryMethods[index].camel;

    var upperCamel = queryMethods[index].upperCamel;

    var type = queryMethods[index].type;

%>

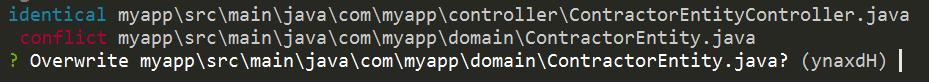
List<<%= entityName %>> findBy<%= upperCamel %>(<%= type %> <%= camel %>);

<% } %>

}

*Listing 7. Szablon klasy repozytorium bazy danych*

W przypadku ewentualnych konfliktów, użytkownik może obejrzeć zmiany jakie mają zajść w jego plikach, odrzucić je lub zaakceptować. Konflikty najczęściej występują w przypadku generowania plików o takiej samej nazwie jak już istniejące i nie jest to zjawisko częste. Przeglądanie zmian jest funkcjonalnością udostępnianą przez sam framework Yeoman. Wiadomość przekazywana użytkownikowi w razie wystąpienia konfliktów pokazana została na rysunku 9. Program oczekuje na reakcję użytkownika w postaci wpisana litery odpowiadającej komendzie.



*Rys. 9. Pytanie o reakcje użytkownika po wystąpieniu konfliktów w plikach*

Dopiero po zatwierdzeniu wszystkich pytań związanych z konfliktami, pliki zostaną zapisane na dysku. Użytkownik może zareagować na występujące konflikty, wpisując literę odpowiadającą konkretnej akcji:

* y – nadpisz aktualny plik
* n – nie nadpisuj
* a – nadpisz aktualny plik i wszystkie inne
* x – zakończ pracę
* d – pokaż różnicę pomiędzy wersją starą a nową
* h – wyświetl pomoc (listę komend)

## 3.4. Używanie generatora

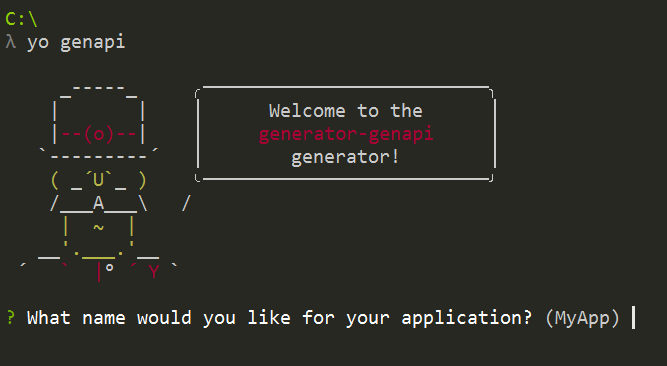
Aplikacja tworzona w ramach generatora jest pełnoprawnym serwerem obsługującym zapytania REST. Przy jego pomocy można wysyłając zapytanie HTTP zapisać, odczytać i usunąć informacje w bazie danych. Wygenerowana aplikacja dobrze pokazuje minimalną strukturę projektu, jaka potrzebna jest do działania takiego serwera. Podgenerator encji poprzez swoją dynamiczność pokazuje, jak to, czego oczekuje użytkownik przekłada się na kod źródłowy.

### 3.4.1. Instalacja

Do uruchamiania generatorów potrzebne jest narzędzie *yo*. Jest to stworzona w tym celu aplikacja linii wiersza poleceń, możliwa do pobrania z repozytorium *npm*. Sam generator również został opublikowany na tym repozytorium, co umożliwia bardzo proste jego zainstalowanie. Aby przeprowadzić instalację, należy najpierw zainstalować środowisko Node.js, razem z którym udostępniane jest narzędzie *npm* służące do instalacji aplikacji z repozytorium. Instalację można przeprowadzić za pomocą komendy uruchamianej z poziomu wiersza poleceń: *npm install -g yo generator-genapi*. Użycie flagi -g pozwala na zainstalowanie aplikacji globalnie, co umożliwi nam używanie generatora z dowolnego katalogu na komputerze. Po krótkiej chwili w konsoli wyświetla się nazwa zainstalowanej aplikacji wraz z drzewem jej zależności, które również zostały pobrane

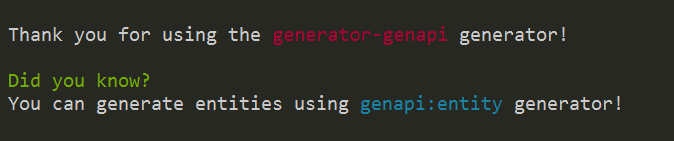
### 3.4.2**.** Uruchomienie

Generator jest aplikacją linii wiersza poleceń, i tak też należy go uruchamiać. Można to zrobić wywołując aplikację *yo* z nazwą generatora. Przykład uruchomienia generatora głównego przedstawiony został na rysunku 10.



*Rys. 10. Uruchamianie generatora głównego*

Po uruchomieniu generatora od razu użytkownik od razu zaczyna odpowiadać na pytania odnośnie swojej aplikacji. Pytań jest niewiele, dzięki czemu cały proces tworzenia projektu w ten sposób trwa kilka sekund. Po zakończeniu generowania użytkownik dostaje informację o możliwości użycia podgeneratora *genapi:entity* do generowania encji, co zostało przedstawione na rysunku 11. Użytkownik instaluje globalnie jedną aplikację (genapi) będącą w rzeczywistości zespołem generatorów. Od razu po utworzeniu plików można zaimportować nowy projekt do ulubionego środowiska programistycznego i zacząć pracę.



*Rys. 11. Zachęta do uruchomienia generatora encji*

## 3.5**.** Możliwości rozwoju

Generator tworzony w ramach przedstawianej pracy jest kompletny i dobrze spełnia swoje zadania, ale oczywiście istnieją możliwości jego rozszerzenia. Podstawową rzeczą możliwą do zrealizowania jest dodanie większej ilości opcji do wyboru w momencie wybierania zależności aplikacji. W obecnej wersji programu, wybranie jednej z zależności powoduje jedynie dodanie jej deklaracji do pliku pom.xml, co skutkuje pobraniem potrzebnych bibliotek. Użytkownik generatora może się spodziewać, że wybory czynione podczas wybierania zależności mają większy wpływ na jego aplikację. Możliwym kierunkiem rozwoju generatora jest dodanie plików szablonowych odpowiadających za podstawową implementację związaną z oferowanymi zależnościami. Przykładowo, jeśli użytkownik wybierze zależność Spring Security, czyli modułu pomagającego w zabezpieczeniu aplikacji, oferowana może być możliwość generowania klas implementujących autoryzację użytkowników (np. przy użyciu tokenów). Podobnie, jeżeli użytkownik dodatkowo wybierze zależność Spring Social, generowana może być konfiguracja pozwalająca na integrację aplikacji z serwisami społecznościowymi. Implementacja takich funkcjonalności jest zwykle powtarzalna i może być wyniesiona do szablonu.

Poza generowaniem klas związanych z modułami oferowanymi jako zależności aplikacji, generator rozszerzyć można również o dodatkowe funkcjonalności związane z obsługa bazy danych. W obecnej wersji tworzone są podstawowe encje oraz repozytoria pozwalające na pobieranie danych według jednego z atrybutów klasy modelu. Potrzebną i łatwą do zaimplementowania funkcjonalnością wydaje się możliwość pobierania danych według wielu atrybutów naraz, np. przy użyciu zapytań SQL. Generator posiadając wiedzę o atrybutach klasy, może w sposób dynamiczny tworzyć takie zapytania pozwalając na dogodne filtrowanie danych.

Kolejną rzeczą, jaką można zrobić w celu ulepszenia generatora, a niezwiązaną z samymi efektami jego pracy jest przejście na nowoczesne podejście do pisania aplikacji Node.js i JavaScript. Zdarzenia asynchroniczne obecnie obsługiwane są poprzez wywołania funkcji *this.async()*, choć obecnie lepszym i polecanym rozwiązaniem jest używanie wspomnianych wcześniej w pracy obiektówpromise. Dodatkowo istnieje również możliwość pisania generatorów Yeoman w języku TypeScript, nadzbioru języka JavaScript oferującego statyczne typowanie i używanie klas. Przejście na ten język programowania ułatwiłoby pracę nad generatorem w przyszłości.

# 4**.** Podsumowanie

Podstawowym celem przyświecającym podczas pisania niniejszej pracy było przedstawienie koncepcji automatyzacji w kontekście generowania kodu źródłowego w teorii oraz pokazanie jej działania w praktyce przy użyciu generatora Yeoman. Uważam, że cel pracy został zrealizowany i aplikacja pisana w ramach pracy dobrze pokazuje czytelnikowi, jak bardzo automatyzacja zadania, jakim jest pisanie kodu może pomóc w pracy nad programem. Wciąż nieocenionym zasobem jest sam programista, który wie i rozumie co chce osiągnąć, lecz bardzo dobrym narzędziem służącym przyspieszeniu pracy i poprawie jej jakości są również generatory napisane we frameworku Yeoman. Koncepcja automatyzacji tworzenia oprogramowania coraz częściej gości w świadomości programistów, co jest zjawiskiem dobrym i pożądanym. Automatyzacja prowadzi do czystszego kodu, podążania za dobrymi praktykami, przyspieszenia pracy i poprawy jakości – jest więc oczywistym kierunkiem, w którym zmierza dzisiejszy świat programowania.

# Bibliografia

|  |  |
| --- | --- |
| [1] | D. L. Parnas, „Software aspects of strategic defense systems,” *Communications of the ACM,* tom 28, nr 12, pp. 1326-1335, 1985. |
| [2] | A. M. Koss, „Programming on the Univac 1: A Woman's Account,” *IEEE Annals of the History of Computing,* tom 25, nr 1, p. 56, 2003. |
| [3] | W. H. K. Chun, „On Software, or the Persistence of Visual Knowledge,” *Grey Room,* nr 18, p. 30, 2004. |
| [4] | [Online]. Dostępne: http://yeoman.io. [Data uzyskania dostępu: 05 07 2017]. |
| [5] | A. Bateman, „Using Yeoman to start writing technical specifications with ReSpec,” 14 03 2015. [Online]. Dostępne: https://adrianba.net/2015/03/14/using-yeoman-to-start-writing-technical-specifications-with-respec/. [Data uzyskania dostępu: 05 07 2017]. |
| [6] | „Overview of Blocking vs Non-Blocking,” [Online]. Dostępne: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/. [Data uzyskania dostępu: 05 07 2017]. |
| [7] | „Building REST services with Spring,” [Online]. Dostępne: https://spring.io/guides/tutorials/bookmarks/. [Data uzyskania dostępu: 05 07 2017]. |
| [8] | S. Boudrias, „Inquirer.js,” [Online]. Dostępne: https://github.com/SBoudrias/Inquirer.js/. [Data uzyskania dostępu: 05 07 2017]. |
| [9] | T. Slezak, J. Bravo, A. Kurmanbayev i A. Sheet, „Automated Code Generation,” [Online]. Dostępne: http://cse360fall12.wikispaces.asu.edu/AutomatedCodeGroup38. [Data uzyskania dostępu: 05 07 2017]. |
| [10] | J. Wilcox, „Paying Too Much for Custom Application Implementation,” 11 03 2011. [Online]. Dostępne: https://blog.edgewater.com/2011/03/11/paying-too-much-for-custom-application-implementation-code-generation/. [Data uzyskania dostępu: 05 07 2017]. |
| [11] | "Writing your own Yeoman generator," [Online]. Dostępne: http://yeoman.io/authoring/. [Accessed 05 07 2017]. |
| [12] | F. J. Budinsky, M. A. Finnie, J. M. Vissides i P. S. Yu, „Automatic code generation from design patterns,” *IBM Systems Journal,* tom 35, nr 2, pp. 151-171, 1996. |

1. Saul Gorn – amerykański pionier informatyki, profesor Moore School na Uniwersytecie Pensylwanii, pracował nad pierwszymi komputerami: ENIAC i EDVAC, [↑](#footnote-ref-1)
2. David Lorge Parnas – jeden z pionierów programowania, twórca koncepcji modularyzacji oprogramowania i ukrywania informacji, promotor etyki w wytwarzaniu oprogramowania. Członek panelu ds. użycia komputerów we wsparciu systemów bitewnych, odchodząc z którego dołączył do listu rezygnacyjnego osiem krótkich esejów o użyciu oprogramowania w strategicznych systemach obronnych [↑](#footnote-ref-2)
3. Wyrażenia *lambda* – wprowadzona w Java SE 8 możliwość programowania w sposób bardziej funkcyjny [↑](#footnote-ref-3)
4. IntelliJ IDEA umożliwia na przykład zamianę kodu w języku Java na kod w języku Kotlin, zachowując jego logikę [↑](#footnote-ref-4)
5. *boilerplate code* – często powtarzający się kod źródłowy, zwykle kopiowany i uzupełniany [↑](#footnote-ref-5)
6. <https://github.com/> - repozytorium kodu źródłowego, w którym programiści zwykle zamieszczają źródła swoich programów open source. [↑](#footnote-ref-6)
7. *Aplikacja monolityczna* – aplikacja typu wszystko-w-jednym, realizująca zarówno dostęp do bazy danych, zawierająca logikę biznesową, jak i prezentująca interfejs użytkownika [↑](#footnote-ref-7)
8. *Architektura mikroserwisowa* – architektura aplikacji podzielonej na luźno połączone moduły realizujące jasno określone funkcje, pozwalająca na niezależną pracę nad nimi przez wiele zespołów programistycznych [↑](#footnote-ref-8)
9. *Aplikacja jednostronna* – aplikacja, w której cała treść jest dynamicznie ładowana zawarta na jednej stronie [↑](#footnote-ref-9)
10. Programowanie sterowane zdarzeniami – paradygmat zakładający sterowanie programem poprzez zdarzenia, na które odpowiada aplikacja. Aplikacja reaguje tylko wtedy, kiedy nadejdzie nowe zdarzenie [↑](#footnote-ref-10)
11. minifikacja – operacja mająca na celu zmniejszenie rozmiaru pliku z kodem źródłowym, np. poprzez zmianę nazw na krótsze i usunięcie niepotrzebnych znaków białych ignorowanych przez interpreter [↑](#footnote-ref-11)
12. https://github.com/SBoudrias/grouped-queue – system kolejki wewnątrz pamięci, stworzony przez Simona Boundrias, pozwalający na poukładanie zadań   
     [↑](#footnote-ref-12)
13. *promise* – specjalny obiekt języka JavaScript, pozwalający operować na zadaniach asynchronicznych w sposób synchroniczny [↑](#footnote-ref-13)
14. http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects [↑](#footnote-ref-14)
15. REST - Representational State Transfer – styl architekturalny tworzenia nowoczesnych aplikacji internetowych, definiujący zasady, które powinny spełniać [↑](#footnote-ref-15)
16. EJS, Embedded JavaScript – prosty język szablonowania używający czystego języka JS do kontroli przebiegu. [↑](#footnote-ref-16)
17. Dekodowanie HTML*, HTML escaping* – mechanizm opuszczania znaków specjalnych HTML w celu używania ich jako samej zawartości pliku [↑](#footnote-ref-17)