BO·TT·EGA
IT minds

Program szkolenia:

Test Driven Development w Node.js

Informacje:

Nazwa:	**Test Driven Development w Node.js**
Kod:	**craft-test-Node**
Kategoria:	Testowanie automatyczne
Odbiorcy:	developerzy, architekci
Czas trwania:	2 dni
Forma:	10% wykłady / 90% warsztaty

TDD - praktyka, narzędzie, strategia wytwarzania oprogramowania. Pomaga utrzymać skupienie i projektować dobre API w skali mikro. Jak każde inne narzędzie czy praktyka sprawdza się w wybranych kontekstach.

W trakcie 2 dniowego szkolenia Test-Driven Node.js nauczysz się projektować aplikacje Node.js z użyciem testów. Będziemy praktykować tzw. Londyńską szkołę TDD znaną również jako Outside-in development.

- 1 dnia projektujemy testami od zera aplikację command-line do przetwarzania danych.
- 2 dnia projektujemy testami API webowe oparte Express.js i bazę MongoDB.

Orientacyjna wielkość problemu to ok. 150 linii kodu produkcyjnego i ok. 300 linii testów dla każdej z aplikacji. Nauka odbywa się w 90% przez ćwiczenia praktyczne, wprowadzamy tylko minimum teorii niezbędnej do rozwiązywania problemów.

Nie będzie żadnych skrótów, szkolenie jest symulacją codziennej pracy z TDD w rzeczywistych projektach.

Czego się nauczysz

- Projektować czysty i testowalny kod sterowany małymi, skupionymi na jednej rzeczy testami.
- Patrzeć na testy przez pryzmat ROI (Return on Investment), unikać testowania tego samego po kilka razy i nadmiernej specyfikacji.
- Projektować testowalny kod z “trudnymi” zależnościami tj. system plików, HTTP czy standardowe wyjście.
- Wybierać najbardziej efektywny sposób testowania kodu asynchronicznego, gdzie część programu wykonuje się teraz, a część później.
- Uzupełniać taktyczne decyzje TDD, strategicznymi decyzjami przy tablicy oraz zbierać potrzebne informacje za pomocą spike’ów.
- Słuchać feedback’u od testów i odpowiednio poprawiać strukturę kodu oraz API które wystawiamy naszym kolaboratorom.
- Zastępować biblioteki i narzędzia, praktykami i technikami dostępnymi na poziomie mechanizmów języka i platformy Node.js. Np. mock, stub, spy bez narzędzi/bibliotek.
- W łatwiejszy sposób przekazywać intencje kodu produkcyjnego oraz kodu testów z wykorzystaniem nowych elementów ES6 i decydować kiedy ich użycie ma sens.
- Stosować najbardziej przystępne elementy programowania funkcyjnego aby zmaksymalizować nasze opcje i wyjść poza programowanie z this/new/klasami.

- Kwestionować wiele mainstreamowych narzędzi i praktyk. Wytłumaczę np. dlaczego nie używam bibliotek tj. nock, sinon, chai albo dlaczego DRY w testach jest przereklamowane.

Zalety szkolenia:

- Londyńska szkoła TDD
- Duży nacisk na elementy programowania funkcyjnego
- Nietrywialne przykłady

BOTTEGA
IT minds

Szczegółowy program:

1. TDD

1.1. Praktyka, Narzędzie, Strategia - patrzenie na TDD z różnych perspektyw

1.2. TDD jako narzędzie taktyczne - projektowanie prostych i rozszerzalnych API w skali mikro

1.3. TDD a decyzje strategiczne - czyli dlaczego TDD nie zastępuje a wzbogaca tablicę i markery

1.4. Spikes - rozpoznawanie natury problemu zanim zaczniemy pisać testy

2. Testy

2.1. ROI testów - patrzenie na testy przez pryzmat zysku z inwestycji i jak pisać testy które są tanie w utrzymaniu

2.2. Testy jako mechanizm feedbacku - jak testy podpowiadają nam które fragmenty kodu poprawić

2.3. Testy jednostkowe vs testy integracyjne - praktyczne przykłady pokazujące jak dobierać rodzaj testu w zależności od problemu

2.4. Happy path vs unhappy path - testowanie sytuacji wyjątkowych i przypadków brzegowych

2.5. Porównanie Londyńskiej szkoły TDD i szkoły z Chicago/Detroit

3. Trudne zależności

3.1. System plików

3.2. Klient HTTP

3.3. Baza danych

3.4. Zamienniki testowe tj. mock, spy, stub

3.5. Zarządanie danymi testowymi w testach integracyjnych z bazą

4. TDD a kod asynchroniczny

4.1. Projektowanie testowalnych API asynchronicznych

4.2. Promisy i async/await - pisanie testów kodu asynchronicznego, które respektują sekwencyjne działanie naszego mózgu

5. TDD z elementami programowania funkcyjnego

BOTTEGA IT minds

5.1. Myślenie w kategorii przepływu danych

5.2. Funkcje wyższego rzędu

5.3. Domknięcia

5.4. Kompozycja

5.5. Pure functions

5.6. Programowanie bez this/class/prototype

6. Narzędzia

6.1. Jak ograniczyć użycie narzędzi do testowania i jak sam język daje nam bardzo przydatne mechanizmy do testów

6.2. Mocha

6.3. Node.js assert

6.4. Supertest