Testy jednostkowe w języku Go

Testowanie oprogramowania jest bardzo ważną częścią procesu jego wytwarzania – pozwala na sprawdzenie, czy nasza aplikacja działa poprawnie i czy spełnia wymagania. W wielu współcześnie używanych językach programowania wsparcie testowania nie jest częścią ich definicji i jest dostarczane z zewnątrz – najczęściej w postaci jakiejś mutacji frameworka xUnit. W Go jego odpowiednik, pakiet testing, jest częścią biblioteki standardowej, a dostarczane z toolchainem narzędzia pozwalają na zarządzanie testami. W tym artykule, skierowanym zarówno do obecnych, jak i potencjalnych programistów języka Go, przyjrzymy się tematowi testów jednostkowych.

CO I DLACZEGO TESTUJEMY

Dawno, dawno temu, gdy zaczynałem moją pierwszą pracę, testy wykonywali testerzy – klikali po interfejsie użytkownika, sprawdzając zgodność zachowania systemu ze specyfikacją. Programiści poprawiali błędy – czasem banalne – znalezione przez testerów, co zajmowało mniej więcej połowę czasu przeznaczonego na development. Było to bardzo kosztowne – w praktyce każdy test był formalnym testem akceptacyjnym, sprawdzającym wymagania biznesowe, funkcjonalne i niefunkcjonalne, na całości aplikacji, replikując zachowania użytkowników.

Współcześnie prawie każdy przyzwoity programista spędza sporo czasu na pisaniu testów jednostkowych (ang. *unit tests*), sprawdzających na poziomie kodu źródłowego działanie pojedynczych metod, funkcji, klas czy modułów. Testy jednostkowe, z reguły automatyczne, są tanie, szybkie i pozwalają na wczesne (czyli również tanie) wykrycie i naprawienie wielu błędów.

W testach integracyjnych testujemy większe kawałki kodu, sprawdzając, jak współdziałają różne części aplikacji. Do takich testów potrzebujemy często zewnętrznych komponentów, np. jakiejś bazy danych, serwera HTTP czy mikrousługi, która musi być w określonym stanie, żeby można było przeprowadzić test. To kosztuje więcej. Testy akceptacyjne, gdzie testujemy system *end-to-end*, ze wszystkim zależnościami, są oczywiście najdroższe i najwolniejsze.

Większość definicji języków programowania w zasadzie ignoruje temat testów (dostajemy co najwyżej instrukcję lub funkcję `assert`). Nie jest to szczególnie dziwne – w końcu do napisania najprostszego testu wystarczy instrukcja warunkowa i operator porównania. Potrzeba dostarczania coraz bardziej skomplikowanych, użytecznych testów spowodowała powstanie licznych frameworków wspierających developerów w tym zadaniu, na czele z dedykowanym do Smalltalka SUnitem [1]. Jego architektura okazała się na tyle atrakcyjna, że xUnit (gdzie „x" jest zastępowane pierwszą literą lub literami nazwy używanego języka programowania, np. JUnit dla Javy czy CppUnit dla C++) stał się *de facto* standardem.

PODSTAWY TESTOWANIA W GO

Frameworki typu xUnit ([2] – nie mylić z xUnit.net) mają jedną wspólną cechę – nie są częścią definicji ani biblioteki standardowej języka, w którym zostały napisane, ani jego standardowego toolchaina. Twórcy Go poszli w innym kierunku – częścią biblioteki standardowej języka jest pakiet `testing` (podobnie jak `unittest` w Pythonie czy `Test` w Ruby), a toolchain dostarcza wszechstronnych narzędzi wspierających testowanie.

Jeśli przyjrzymy się zawartości `testing`, znajdziemy w nim przede wszystkim trzy typy – `T` (od *testing*), `B` (*benchmarking*) i `F` (*fuzzing*) wraz z zestawem powiązanych z nimi funkcji. Typ `T` jest przekazywany do funkcji testującej (o której za chwilę), służy do zarządzania stanem testów jednostkowych i pozwala na umieszczanie rezultatów w sformatowanych logach. Typ `B` związany jest z testami wydajnościowymi, pozwala na ustalenie liczby iteracji do wykonania i sterowanie pomiarem czasu. Ostatni – najmłodszy, bo wprowadzony dopiero w wersji 1.18 języka – to typ przekazywany do *fuzz testów*, do których jeszcze wrócimy. Dla porządku należy wspomnieć, że `testing` eksportuje jeszcze kilka innych typów i funkcji, ale są one częściami wewnętrznej infrastruktury testowej i implementacji polecenia `go test`, więc w praktyce programista nie ma z nimi nic do czynienia.

Go oczekuje określonej organizacji plików z testami i określonej konwencji ich nazywania. Kod testujący musi być przechowywany w osobnych plikach umieszczonych w tym samym katalogu, co kod produkcyjny, i być nazwany zgodnie ze wzorcem *_test.go. Jeśli więc chcemy testować funkcje pakietu foo umieszczone w pliku *foo.go*, to kod testujący musi trafić do pliku *foo_test.go*. Pośrednio wynika z tego, że będziemy dążyć do wyprowadzenia większości naszego kodu poza *main.go*, żeby przy wykonywaniu testów nie trzeba było uruchamiać całej aplikacji.

Posłużmy się przykładem programu w stylu „Hello, world". W naszym pakiecie mamy jedną funkcję (w pliku *hello.go*), która przyjmuje jako parametr język i wyświetla na konsoli powitanie w tym języku (wywołanie w *main.go*) – Listingi 1-2.

Listing 1. Zawartość pliku main.go

```
package main

import "fmt"

func main() {
    fmt.Println(Hello("polish"))
}
```