# Singleton

Wzorzec czy antywzorzec. Oto jest pytanie?

Napewno jest przydatny, gdy nie boimy się “złego” designu... Ale sam osobiście tego wzorca unikam jak ognia. Do czego służy singleton?

Singleton zapewnia istnienie tylko jednej klasy jednocześnie, w ten sposób, by nie dało się wywołać określonej klasy więcej niż jeden raz. Dzięki temu klasa ta może np. zapisywać wszystkie informacje z dowolnych miejsc i zwracać je, niezależnie od tego, gdzie jesteśmy...

Problem polega tylko na tym, że to naprawdę nie jest dobre rozwiązanie. Wzorzec ten ma więcej wad niż zalet. Po pierwsze łamie zasadę pojedynczej odpowiedzialności - o której mówiłem Ci w kursie SOLIDa, po drugie może doprowadzić do łamania barier. Określone elementy kodu nie powinny wiedzieć o innych elementach, jeśli nie jest to uzasadnione.

Dodatkowo wzorzec ten działa słabo w środowisku wielo-wątkowym. Choć PHP takim nie jest, to jednak stosownie Swoole czy ReactPHP może to umożliwić i wtedy... nie będzie to najlepsze rozwiązanie.

Na koniec testy - te też mogą zachowywać się dziwnie.

Podsumuję to tak. Za chwilę pokażę Ci wzorzec, który warto byś znał, ale nie warto żebyś nagminnie stosował.

Nasz problem jest następujący - potrzebujemy stworzyć miejsce, do którego będziemy zapisywać logi. Nie ważne skąd - na końcu po prostu chcemy je wyświetlić.

W normalnej sytuacji moglibyśmy zapisać je do pliku, ale z jakiegoś powodu zdecydowaliśmy się trzymać je w pamięci, w różnych miejscach.

To od czego musimy zacząć to upewnienie się, że nasza klasa nie ma możliwości tworzenia nowych instancji. To zadanie jest bardzo proste - wszystko co wystarczy zrobić to po prostu zmiana zasięgu konstruktora na private.

```
class GlobalLogger
{
    private static ?GlobalLogger $instance = null;
```

```
    private function __construct()
    {
    }
}
```

Z tak obecnie wyglądającą klasą możemy zrobić... Nic! Dlatego musimy zapewnić jakąś możliwość tworzenia instancji. W tym wypadku korzystamy ze statycznych metod.

```
class GlobalLogger
{
    private static ?GlobalLogger $instance = null;

    private function __construct()
    {
    }

    public static function getInstance(): GlobalLogger
    {
        if (self::$instance === null) {
            self::$instance = new GlobalLogger();
        }

        return self::$instance;
    }
}
```

Za każdym razem, gdy pobierzemy instancję, będziemy sprawdzać czy wcześniej nie została stworzona. Jeśli została - zwrócimy ją. Jeśli nie, utworzymy jej nową instancję.

I to w zasadzie tyle, choć nasz kod jeszcze nie realizuje głównego zadania - czyli logowania. Teraz do klasy dodajmy możliwość dodania i odczytania logów.

```
class GlobalLogger
{
```

```
    private static ?GlobalLogger $instance = null;
    private array $logs = [];

    private function __construct()
    {
    }

    public static function getInstance(): GlobalLogger
    {
        if (self::$instance === null) {
            self::$instance = new GlobalLogger();
        }

        return self::$instance;
    }

    public function log(string $message): void
    {
        $this->logs[] = $message;
    }

    public function getLogs(): array
    {
        return $this->logs;
    }
}
```

Wszystko tak proste, że każdy junior to zrozumie. Singleton jest jednym z łatwiejszych wzorców i nie wiele ma wspólnego z abstrakcją.

Jednak czym by był nasz kod bez faktycznej praktyki. Dodajmy zatem kilka klas używających loggera i użyjmy ich.

```
class SomeAction
{
    public function doSomething()
    {
        $logger = GlobalLogger::getInstance();
        $logger->log("Did something");
```

```
    }
}

class AnotherAction
{
    public function doSomethingElse()
    {
        $logger = GlobalLogger::getInstance();
        $logger->log("Did something else");
    }
}

$someAction = new SomeAction();
$someAction->doSomething();
$anotherAction = new AnotherAction();
$anotherAction->doSomethingElse();
```

Jak widzisz, wszystko jest współdzielone. Wywołując kod:

```
$logger = GlobalLogger::getInstance();
var_dump($logger->getLogs());
```

Mamy dostęp do wszystkich logów dostępnych w aplikacji.

I teraz pytanie - czy w praktycznym kodzie produkcyjnym, to się może przydać?

Tak. Może. Nie jest to zbyt częste, a nadużywanie tego może powodować problemy, jednak jeśli chcesz by w kodzie istniał jeden, ogólnodostępny obiekt danej klasy, to singleton jest jedynym sensownym wyjściem.

A kiedy takie zastosowanie ma naprawdę sens? Choćby w przypadku Laravela. Bo tak - owszem. Działa on w oparciu o singletona i wrapuje całą aplikację. To dzięki temy mamy dostęp do wielu elementów, do których normalnie trzeba byłoby taki dostęp stworzyć. To dlatego fasady, do których dojdziemy - zapewniają nam dane o użytkowniku, nawet jeśli wykonywany kod nie otrzymał requestu a fasadę Auth. I to właśnie za to Laravel jest często krytykowany.

Wierzę jednak, że Ty doskonale wiesz, jak posługiwać się frameworkami i że w Laravelu można pisać zarówno zły, jak i naprawdę świetny kod źródłowy. A

skoro tu jesteś, to chcesz pisać ten dobry i czysty kodzik.

No właśnie.

Poza samym wzorcem, jeśli potrzebujesz z niego skorzystać, to możesz do tego wykorzystać również to, co oferuje Laravel Service Container. Już w kursie SOLIDa mówiliśmy o odwracaniu zależności. Teraz czas dodać, że jeśli chcesz, nie musisz stosować bind, a zamiast tej metody użyć singleton, która automatycznie zaaplikuje na określonym interfejsie, właśnie tą metodę.

```
$this->app->singleton(Abstract::class, Concrete::class);
```