Pyłek

Jakiś czas temu, na swoim instagramie lenkowski_dev umieściłem post - nie daj sobie wmówić, że PHP to gorszy język.

Pod postem pojawiło się pytanie: “Skoro PHP nie jest gorszy to jak zapisać dane w pamięci RAM”.

Sposób w jaki działa PHP nie wymaga od niego zapisywania danych do RAMu, choć oczywiście za pośrednictwem REDISa, jesteśmy w stanie to zrealizować.

Czemu o tym mówię? Bo pyłek to wzorzec, który służy do optymalizacji pamięci. A w PHP nie korzystamy ze wszystkich dostępnych danych. Zwykle potrzebujemy zaledwie ich wycinka.

Pyłek jest wzorcem, który pozwala nam zmieścić więcej obiektów w danej przestrzeni RAM, ale my przecież z RAMu raczej nie korzystamy...

Czy w takiej sytuacji pyłek jest wzorcem, który może nam się w ogóle przydać?

Nasze zadanie wygląda następująco.

Otrzymaliśmy listę danych z pracownikami i firmami w których pracują. Używamy bazy relacyjnej, ale oczywiście nie wszystkie dane w tej bazie posiadamy. Naszym celem jest wyświetlenie informacji o tych pracownikach. Chcemy też zapisać dane firm, ale oczywiście tylko raz i tylko wtedy, gdy faktycznie tych danych nie ma w bazie.

Stwórzmy więc nasz agregator pyłków - FlyweightFactory.

```
class FlyweightFactory
{
    /**
     * @var CompanyAddress[]
     */
    private array $flyweights = [];

    public function __construct(array $initialFlyweights)
    {
```

```
        foreach ($initialFlyweights as $state) {
            $id = array_shift($state);
            $this->flyweights[$this->getKey($state)] = new Co
        }
    }

    private function getKey(array $state): string
    {
        ksort($state);

        return implode("_", $state);
    }

    public function getFlyweight(array $sharedState): Company
    {
        $key = $this->getKey($sharedState);

        if (!isset($this->flyweights[$key])) {
            echo "Tworzę nowy stan firmy.\n";
            $this->flyweights[$key] = new CompanyAddress(uniq
        } else {
            echo "Pobieram pobrany wcześniej stan.\n";
        }

        return $this->flyweights[$key];
    }

    public function listFlyweights(): void
    {
        $count = count($this->flyweights);
        echo "\nIlość pyłków w pamięci: $count\n";
        foreach ($this->flyweights as $key => $flyweight) {
            echo $key . "\n";
        }
    }
}
```

Zawiera on listę firm, możliwość pobrania konkretnej oraz wylistowania wszystkich.

Mamy oczywiście również klasę przechowującą konkretny pyłek, czyli CompanyAddress.

```
class CompanyAddress
{
    private string $id;
    private array $sharedState;

    public function __construct(string $id, array $sharedStat
    {
        $this->id = $id;
        $this->sharedState = $sharedState;
    }

    public function print(array $uniqueState): void
    {
        $s = json_encode($this->sharedState);
        $u = json_encode($uniqueState);
        echo "Stan współdzielony ($s) dla ID ".$this->id." st
    }
}
```

Mając taką strukturę klas, czyli kolekcję, będącą w stanie wczytywać pyłki i faktyczne pyłki, możemy sobie wyobrazić, że na start wczytujemy dane firm.

```
$factory = new FlyweightFactory([
    [uniqid(), '6793108059', 'Inpost sp. z o. o.', 'ul. PANA
    [uniqid(), '5321956443', 'Grycan', 'ul. KWITNĄCEJ WIŚNI 2
]);
```

Być może w prawdziwym życiu, nie jest to zbyt optymalne. Możemy sobie jednak wyobrazić pewną optymalizację w oparciu o lazy-loading. To jednak teraz nie jest aż tak istotne.

Chcąc wyświetlić informacje o konkretnym pracowniku użyjemy funkcji

```
function displayCompanyWorker(
    FlyweightFactory $ff, string $nip, string $company, strin
) {
    // Store company
    $flyweight = $ff->getFlyweight([$nip, $company, $address,

    // Display company extended with owner
    $flyweight->print([$workerFullName]);
}
```

A następnie ją wywołamy z pobranymi z crawlera wartościami

```
displayCompanyWorker(
    $factory,
    '9562318640',
    'Exulto sp. z o. o.',
    'ul. Włocławska 167',
    '87-100',
    'Toruń',
    'Marcin Lenkowski'
);

displayCompanyWorker(
    $factory,
    '6793108059',
    'Inpost sp. z o. o.',
    'ul. PANA TADEUSZA 4',
    '30-727',
    'Kraków',
    'Rafał Brzoska'
);

displayCompanyWorker(
    $factory,
    '5321956443',
    'Grycan',
    'ul. KWITNĄCEJ WIŚNI 2',
    '05-462',
```

```
    'Majdan',
    'Zbigniew Grycan'
);

displayCompanyWorker(
    $factory,
    '9562318640',
    'Exulto sp. z o. o.',
    'ul. Włocławska 167',
    '87-100',
    'Toruń',
    'Marta Lenkowska'
);
```

Dzięki zastosowaniu pyłka, w 2 pierwszych przypadkach, wykorzystamy dane, które mamy już załadowane. Dane te oczywiście docelowo mogły by posiadać także indentyfikator.

W przypadku 3 odkrywamy nowe dane i dodajemy je do naszej kolekcji. Generujemy tutaj uuid (no a w przypadku produkcyjnego rozwiązania, wypadałoby poszukać go jeszcze w bazie danych).

Niezależnie jednak od tego, gdy dochodzi do podania tych samych danych ponownie, ale z innym pracownikiem - pobieramy już wcześniej przygotowany obiekt. Nie tworzymy jego kopii w pamięci - właśnie dzięki pyłkowi.

Ten wzorzec nie jest zbyt popularny w PHP. Co więcej, jego zastosowanie, nawet tutaj, w przykładzie - może być na wyrost. Cel jest jednak jasny - jeśli będziemy mieć obiekty w pamięci, wyglądające tak samo - zamiast pchać do niej nowe dane, możemy skorzystać z posiadanych wcześniej wartości.

Wady i zalety

Zalety:

- Możesz zaoszczędzić mnóstwo pamięci RAM, o ile twój program tworzy mnóstwo podobnych obiektów.

Wady

- Może się zdarzyć, że oszczędność pamięci odbędzie się kosztem czasu procesora, gdyż część danych kontekstowych musi być wyliczana przy każdym wywołaniu metody pyłka.

- Kod staje się dużo bardziej skomplikowany. Nowi członkowie zespołu z pewnością będą się zastanawiać dlaczego stan czegoś został odseparowany.