

# Mechanizmy dostosowywania EZD do potrzeb urzędów

<b>Problem</b>	<b>3</b>
<b>Przegląd ogólnych rozwiązań</b>	<b>3</b>
Hooki	3
Webhooki	4
Konfigurowalność aplikacji	4
Mechanizm zestandaryzowanych wtyczek	4
Wstrzykiwanie zależności	4
Mikroserwisy	4
Wstrzykiwanie kodu	4
<b>Jak testować dostosowywalny kod?</b>	<b>4</b>
Przykład w języku Lisp (Clojure)	5
<b>Rozwiązania dla EZD</b>	<b>6</b>

# Problem

Urzędy będą miały potrzebę dostosowywania instancji EZD do własnych potrzeb.

Przykładowo, urząd do spraw sepulek może chcieć:

1. zintegrować się z bazą typów sepulek, umożliwiając przypisywanie urzędnikom typów, w których się oni specjalizują;
2. wyświetlać zdjęcia sepulek na spisie spraw zapoczątkowanych przez formularz XYZ w ePUAP (który to formularz zawiera zdjęcie sepulki);
3. nadawać inny bieg sprawom dotyczącym sepulek o określonej charakterystyce (bo np. objęte są one odrębnymi przepisami);
4. automatycznie odrzucać wnioski o zgodę na sepulenie składane przez osoby samotne;
5. automatycznie dekretować wnioski o zgodę na sepulenie do konkretnych urzędników w zależności od:
  - a. adresu zamieszkania wnioskodawcy;
  - b. typu sepulki (wg specjalizacji urzędnika);
6. wyszukiwać sprawy wg danych określonych w punkcie 5.;
7. po wydaniu zgody na sepulenie wysyłać informację o tym do Krajowego Rejestru Sepulenia;
8. pobierać z Krajowego Rejestru Sepulenia informacje o sepulkach posiadanych przez wnioskodawcę i dołączać ją do akt sprawy;
9. przetwarzać załączone we wnioskach zdjęcia sepulek w oprogramowaniu, które obliczy z nich parametry sepulek, zapisać te parametry w bazie danych, wyświetlić je w oddzielnej kolumnie na liście spraw i umożliwić wyszukiwanie po nich;
10. tworzyć własne typy metadanych;
11. tworzyć własne wzory pism;
12. tworzyć własne raporty;
13. tworzyć własne rejestry.

## Przegląd ogólnych rozwiązań

W świecie programistycznym funkcjonują różne mechanizmy dostosowywania aplikacji do własnych, specyficznych potrzeb. Poniżej znajduje się przegląd niektórych z nich.

### Hooki

Zaczepy umożliwiają wykonanie w wybranych punktach wykonywania programu funkcji zdefiniowanych przez użytkownika. Do tych funkcji mogą być niekiedy przekazywane ustalone parametry. Zob.:

- [„What is meant by the term »hook« in programming?”](#)
- [hooki w Emacsie](#)
- [hooki w Redmine](#)
- [Active Record Callbacks](#)
- [hasło “hooking” w Wikipedii](#)

## Webhooki

Webhooki polegają na wykonywaniu w określonych punktach programu ustalonych zapytań HTTP. Zazwyczaj ich działanie jest jednostronne, tj. program po wykonaniu zapytania i otrzymaniu potwierdzenia jego przetworzenia nie wykonuje żadnych dodatkowych działań. Zob. [hasło "webhook" w Wikipedii](#).

## Konfigurowalność aplikacji

Kod aplikacji jest jednolity, natomiast jej działanie różnicuje się w zależności od wartości parametrów konfiguracyjnych, do których szeregowi użytkownik nie ma dostępu.

## Mechanizm zestandaryzowanych wtyczek

Polega na umożliwieniu dostarczania kawałków kodu, który jest wykorzystywany w określonych miejscach aplikacji.

## Wstrzykiwanie zależności

Polega na wykorzystywaniu wybranych fragmentów kodu nie bezpośrednio, lecz jako parametrów posiadających ustandaryzowane API, co pozwala na podmianę tych fragmentów na inne, spełniające te same zadania.

## Mikroserwisy

Podobnie jak przy wstrzykiwaniu zależności, ale przekazywane są nie fragmenty kodu, lecz adresy końcówek ustandaryzowanego API sieciowego.

## Wstrzykiwanie kodu

Polega na modyfikacji, w tym zamianie oraz rozszerzaniu, istniejącego kodu aplikacji bez podejmowanych explicite działań w tym kierunku po stronie tegoż kodu. W zależności od języka i stylu programowania istnieją różne sposoby na osiągnięcie tego celu, m.in.:

- [monkey patching](#) — ogólne określenie na dynamiczne modyfikowanie cudzego kodu;
- [alias method chain](#) w Ruby on Rails;
- [opakowywanie funkcji w Emacs Lispie](#);
- [biblioteka Robert Hooke](#) w Clojure;
- [wtyczki w Ruby on Rails](#)

## Jak testować dostosowywalny kod?

Przy pisaniu testów istotne jest niezakładanie konkretnego brzmienia fragmentów kodu, które mogą podlegać zmianie w wyniku działania wtyczek lub innych w/w mechanizmów.

## Przykład w języku Lisp (Clojure)

Przyjmijmy, że testujemy następującą funkcję, przypisującą sprawę do komórki organizacyjnej.

```
(defn sprawa-w-komorce [sprawa komorka]
  ;; Jeśli sprawa ma już prowadzącego z danej komórki, nie zmieniamy
  go.
  (if (contains? (:czlonkowie komorka) (:prowadzacy sprawa))
      sprawa
      (assoc sprawa :prowadzacy (domyslny-prowadzacy-sprawe sprawa
komorka))))
```

Chcemy sprawdzić, czy funkcja ta, w przypadku gdy sprawa nie ma jeszcze prowadzącego z danej komórki, ustawia go zgodnie z funkcją `domyslny-prowadzacy-sprawe`. Przyjmijmy, że implementacja tej funkcji zawarta w podstawowym systemie EZD jest następująca:

```
(defn domyslny-prowadzacy-sprawe [_sprawa komorka]
  (:naczelnik komorka))
```

Kiepsko napisany test mógłby w tej sytuacji wyglądać następująco:

```
(deftest test-sprawa-w-komorce
  (testing "użycie domyślnego prowadzącego"
    (let [jan-kowalski {:imie "Jan" :nazwisko "Kowalski"}
          jozef-nowak {:imie "Józef" :nazwisko "Nowak"}
          komorka {:czlonkowie #{jan-kowalski jozef-nowak}
                  :naczelnik jan-kowalski}]
      (testing "w przypadku braku prowadzącego"
        (let [sprawa {:znak "XYZ" :prowadzacy nil}]
          (is (= jan-kowalski
                  (:prowadzacy (sprawa-w-komorce sprawa
komorka))))))))))
```

Aby dopuścić modyfikacje funkcji `domyslny-prowadzacy-sprawe`, należałoby zapisać w teście jej konkretną implementację:

```
(deftest test-sprawa-w-komorce
  (with-redefs [domyslny-prowadzacy-sprawe (fn [_ komorka]
(:naczelnik komorka))]
    (testing "użycie domyślnego prowadzącego"
      (let [jan-kowalski {:imie "Jan" :nazwisko "Kowalski"}
            jozef-nowak {:imie "Józef" :nazwisko "Nowak"}
            komorka {:czlonkowie #{jan-kowalski jozef-nowak}
                    :naczelnik jan-kowalski}]
        (testing "w przypadku braku prowadzącego"
          (let [sprawa {:znak "XYZ" :prowadzacy nil}]
            (is (= jan-kowalski
                    (:prowadzacy (sprawa-w-komorce sprawa
komorka))))))))))
```

## Rozwiązania dla EZD

Z uwagi na szerokie spektrum problemu, najprawdopodobniej konieczne będzie zastosowanie różnych technologii pozwalających na dostosowanie EZD do potrzeb urzędu. Wybór technologii może być niekiedy podyktowany możliwościami oferowanymi przez zastosowany język programowania / szkielet programistyczny. Przykład takiego wyboru można znaleźć w odniesieniu do narzędzia do zarządzania projektami Redmine i [jego mechanizmu wtyczek](#): “Due to the fact that it is so easy to extend models and controllers the Ruby way (via including modules), Redmine shouldn't (and doesn't) maintain an API for overriding the core's models and/or controllers. Views on the other hand are tricky (because of Rails magic) so an API for overriding them is way more useful (and thus implemented in Redmine).” Istotne jest takie wyspecyfikowanie warunków zamówienia dla EZD, aby uwzględniały one możliwości dostosowania systemu do własnych potrzeb urzędu.

Potencjalne obszary zastosowań wymienionych mechanizmów dostosowawczych w EZD można przedstawić w poniższej tabeli. Tabela może nie uwzględniać wszystkich możliwości z uwagi na elastyczność niektórych mechanizmów z jednej oraz niepraktyczność ich zastosowań w różnych obszarach z drugiej strony.

Nr obszaru	hooki	webhooki	konfigurowalność	zestandardyzowane wtyczki	wstrzykiwanie zależności	mikroserwisy	wstrzykiwanie kodu
1				✓			✓
2	✓	✓		✓			✓
3	✓	✓		✓			✓
4	✓	✓		✓			✓
5	✓	✓		✓			✓
6	✓	✓	✓	✓			✓
7	✓	✓		✓			✓
8	✓	✓		✓			✓
9				✓			✓
10			✓	✓			
11			✓	✓			
12			✓	✓			
13			✓	✓			