# Ajax dla zaawansowanych. Architektura i najlepsze rozwiązania


Autor: Shawn M. Lauriat
Tłumaczenie: Radosław Meryk
ISBN: 978-83-246-1585-8
Tytuł oryginału: Advanced Ajax: Architecture and Best Practices
Format: 168x237, stron: 392

Dowiedz się:

- Jak tworzyć rozbudowane i idealnie dopasowane do potrzeb interfejsy?
- Jak zapewnić uniwersalność, skalowalność oraz łatwość eksploatacji?
- Jak zaprojektować architekturę aplikacji?

Ajax (skrót od ang. Asynchronous JavaScript and XML) to niezwykle popularna technologia tworzenia serwisów internetowych, w której połączono kilka sprawdzonych technik. Dzięki tej zintegrowanej technologii do niezbędnego minimum została ograniczona ilość danych przesyłanych pomiędzy serwerem a oknem przeglądarki użytkownika. Nie tylko to przysporzyło Ajaksowi zwolenników – jest on także bardzo dobrym narzędziem do tworzenia interaktywnych serwisów internetowych. Sprawdza się również przy przeprowadzaniu weryfikacji danych oraz rysowaniu wykresów w czasie rzeczywistym. Dzięki asynchronicznym wywołaniom umożliwia szybszą interakcję z użytkownikiem, a poszczególne sekcje mogą być wywoływane indywidualnie, dzięki czemu aplikacja sprawia wrażenie bardziej dynamicznej.

Książka „Ajax dla zaawansowanych. Architektura i najlepsze praktyki" to idealna lektura dla programisty, który miał już przyjemność pracować z Ajaksem. Podjęto tu wszystkie zagadnienia niezbędne do tworzenia dynamicznych aplikacji, niezależnie od użytych narzędzi i technologii. Na praktycznych przykładach przedstawiono sposoby wykorzystania Ajaksa do tworzenia rozbudowanych interfejsów w przeglądarce dla aplikacji internetowych, ze szczególnym uwzględnieniem ich uniwersalności, możliwości wielokrotnego wykorzystania kodu, skalowalności oraz łatwości eksploatacji. Podręcznik wskazuje zarówno sytuacje, w których Ajax jest przydatny, jak i takie, w których jego wybór nie spełni oczekiwań użytkownika.

- Planowanie interfejsów Ajaksa
- Debugowanie, walidacja i optymalizacja kodu
- Tworzenie skalowalnych interefejsów
- Architektura aplikacji po stronie serwera oraz klienta
- Bezpieczeństwo aplikacji internetowych
- Projektowanie gier

**Poznaj więcej niezwykłych możliwości Ajaksa!**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

# Spis treści

Rozdział 7

# Architektura aplikacji po stronie serwera

**W tym rozdziale:**

Projektowanie rozbudowanych aplikacji internetowych zazwyczaj koncentruje się na programach działających po stronie klienta. Ma to sens, ponieważ większość nowych technologii stosowanych w aplikacjach internetowych koncentruje się wokół obiektu JavaScript `XMLHttpRequest`. Jednak projektowanie aplikacji po stronie serwera w dalszym ciągu zasługuje na co najmniej tyle uwagi, ile wymagało ono przed upowszechnieniem się aplikacji sterowanych żądaniami Ajax. Technologie po stronie serwera nie tylko muszą w dalszym ciągu wspierać operacje ładowania kompletnych stron, ale także obsługiwać zapytania mające na celu aktualizację lub odczytywanie informacji w tle.

Aplikacje tej natury wymagają dostatecznie elastycznej architektury, tak aby można było ograniczyć ładowanie danych i obiektów oraz wykonywanie operacji dla potrzeb obsługi bieżącego żądania. Zapewnienie tego samego poziomu kontroli autoryzacji i zestawu funkcji niezależnie od tego, jaka część aplikacji ładuje się dla określonego typu żądania, jest nie lada wyzwaniem.

## 7.1. Projektowanie aplikacji dla wielu interfejsów

Aplikacje sterowane żądaniami Ajax wymagają dostępu do danych i zestawu funkcji przy użyciu co najmniej dwóch formatów odpowiedzi: XHTML i XML lub JSON. Spełnienie tego wymagania stwarza konieczność zapewnienia możliwości wyprowadzania tych samych danych bądź wyników wywołań do dwóch różnych zestawów szablonów — to implikuje potrzebę elastycznego mechanizmu obsługi szablonów. Potrzebna jest również wystarczająco elastyczna architektura aplikacji — taka, która gwarantuje, że dla wybranego żądania nie będą wykonywane zbędne operacje.

Aplikacja nie powinna zajmować się pobieraniem metadanych strony, struktur nawigacyjnych, czy też obsługą uprawnień związanych z dozwolonymi kontrolkami interfejsu tylko po to, aby zwrócić prostą listę w odpowiedzi na wywołanie `XMLHttpRequest`. Potrzebna jest jednak pewna struktura tworząca kręgosłup aplikacji, a także mechanizm zapewniający załadowanie konfiguracji, nawiązanie połączenia z bazą danych, zarządzanie buforowaniem, przesyłaniem komunikatów, uwierzytelnianiem, autoryzacją oraz ładowaniem zasobów.

Dzięki należytej abstrakcji logiki w aplikacji można zapewnić łatwiejsze dynamiczne ładowanie funkcji i ich wielokrotne wykorzystywanie w aplikacji. Dzięki temu łatwiejsza jest również praca z kodem aplikacji, ponieważ funkcje i metody mają znacznie bardziej zwięzłe definicje. Zaniedbania w odpowiedniej abstrakcji

logiki aplikacji niezwykle utrudniają pielęgnację. W przypadku konieczności aktualizacji logiki powielonej w kilku miejscach aplikacji trzeba to robić wielokrotnie. Problemy są szczególnie dotkliwe, kiedy w powielonym kodzie znajduje się błąd, którego poprawienie ma istotne znaczenie — na przykład luka w zabezpieczeniach.

Rozważmy funkcje biblioteczne języka PHP `md5()` i `sha1()`. Są to narzędzia do tworzenia skrótów (ang. *hash*) pozwalające na generowanie tokenów wykorzystywanych w plikach cookie, sesjach, operacjach weryfikacji źródeł przesyłania, nazwach plików oraz innych miejscach wymagających pozornie losowych, ale spójnych unikatowych identyfikatorów. Używa się ich zwłaszcza wtedy, gdy istnieje zagrożenie, że napastnikom uda się odgadnąć wspomniane identyfikatory i wykorzystać do realizacji różnych celów. Programiści zazwyczaj korzystają z tych funkcji bezpośrednio, ponieważ wywołanie `sha1($text)` zajmuje bardzo mało miejsca i nie pogarsza czytelności kodu.

Może się jednak zdarzyć, że programista w wywołaniach funkcji nie poda argumentu `salt`[1]. W takim przypadku w kilku plikach źródłowych należy wprowadzić zmiany w sposób, który może nieco skomplikować kod — trzeba bowiem załadować parametry globalne i zarządzać ich wartościami. W niektórych zastosowaniach narzędzi generowania skrótów trzeba posługiwać się losową wartością argumentu `salt`, w innych argument ten jest prekonfigurowany — na przykład podczas generowania skrótów haseł, aby zapewnić ich spójność. We wszystkich wynikających stąd scenariuszach w funkcjach należy uwzględnić więcej kodu, którego tam być nie powinno.

```
class User extends DBO {
    /* ... */
    public function set($field, $value) {
        if ($field == 'password') {
            global $config;
            $salt = $config['settings']['salt'];
            $hash = sha1($string . $salt);
            return parent::set($field, $hash);
        } else {
            return parent::set($field, $value);
        }
    }
    /* ... */
}
```

[1] Więcej informacji na temat generowania skrótów z argumentem `salt` można znaleźć w rozdziale 8., „Bezpieczeństwo aplikacji internetowych".

Powyższa definicja metody `User::set()` przeciąża bazową definicję `DBO::set()`. Ma to na celu ustawienie wartości pola `password` na skrót hasła zamiast jego wartości tekstowej. Zaprezentowany sposób daje dostęp do globalnych ustawień konfiguracyjnych umożliwiających skorzystanie z predefiniowanych wartości argumentu `salt`, ale obsługa tego argumentu i mechanizmu generowania skrótów wymaga miejsca wewnątrz metody `User::set()`. Praktyka ta nie ma sensu w pokazanym kontekście i zaciemnia pierwotne przeznaczenie metody — zwłaszcza jeśli w jej obrębie zachodzi potrzeba implementacji dodatkowych funkcji w dalszej fazie projektowania. Oto kolejna wersja metody `set`:

```
class User extends DBO {
    /* ... */
    public function set($field, $value) {
        if ($field == 'password') {
            return parent::set(
                $field,
                Utilities::hashWithSalt($value)
            );
        } else {
            return parent::set($field, $value);
        }
    }
    /* ... */
}
```

Powyższa definicja metody `User::set()` wymaga jedynie wywołania statycznej metody `Utilities::hashWithSalt()`, co sprawia, że jest ona znacznie łatwiejsza do pielęgnacji. W przypadku zmiany logiki związanej z generowaniem skrótów zmienić trzeba tylko definicje wewnątrz klasy `Utilities`. Dzięki temu, że jest tylko jedno miejsce wprowadzania zmian, zyskujemy pewność, że wymagana zmiana będzie wprowadzona we wszystkich fragmentach kodu, w których wykorzystano funkcje generowania skrótów.

Oddzielenie logiki aplikacji od warstwy obsługi danych i prezentacji jest niezwykle pomocne w przypadku wszystkich aplikacji internetowych, a w szczególności w przypadku aplikacji internetowych sterowanych wywołaniami Ajaksa. Na przykład logika zaimplementowana w wielu miejscach aplikacji przyczynia się do powstania zależności prezentacji od metod przechowywania danych. Zdarza się również, że logika aplikacji, a nawet kod obsługi danych, stają się zależne od interfejsu prezentowanego użytkownikom.

Zapisywanie kodu obsługi zapytań do baz danych i otrzymywanych wyników bezpośrednio w wyjściu XHTML prowadzi do powstania liniowego wyjścia o stru-

kturze siatkowej (ang. *grid-based*) z bardzo ograniczonym zestawem funkcji. Ponieważ kod niezbędny do zarządzania zapytaniami i zwracanymi wynikami już „skaził" kod związany z generowaniem zbioru znaczników strony, dodanie bardziej złożonych interfejsów jest znacznie trudniejsze do zaimplementowania, nie mówiąc już o pielęgnacji.

W przypadku elementów interfejsu sterowanych żądaniami Ajax problem mieszania interfejsu z logiką aplikacji wzrasta proporcjonalnie do liczby formatów wyniku, jakie powinna obsługiwać aplikacja — XML, JSON lub oba te formaty. Całą logikę, która powinna trafić do pierwotnej postaci kodu XHTML, należy teraz zdublować w każdej z metod wyniku obsługujących żądania Ajax. Jeśli w tym momencie, w związku z wywołaniami z poziomu kodu generującego każdy z formatów wyjścia, zdarzy się sytuacja podobna do problemu logiki generowania skrótów opisanego poprzednio, rozwiązanie problemu zajmie sporo czasu i wysiłku.

Na przykład aplikacja może wyświetlać informacje użytkownika z wykorzystaniem następującego kodu:

```
<div id="userinfo">
<?php
$query = 'SELECT `id`, `login`, `nazwisko`, `email`, `utworzono`
    FROM `uzytkownicy` where `id` = ' . $id;
if ($result = mysqli_query($query)) {
    if ($user = mysqli_fetch_assoc($result)) {
        // Wyświetlenie informacji o użytkowniku z wykorzystaniem tablicy asocjacyjne.j
    } else {
        // Użytkownika nie znaleziono.
    }
} else {
    // Błąd zapytania.
}
?>
</div>
```

W powyższym kodzie jest kilka problemów — wszystkie one mogłyby się pojawić w kodzie odpowiedzi zarówno w formacie XML, jak i JSON. Na podstawie samego kodu programiści nie są w stanie stwierdzić, czy dla zmiennej `$id` zastosowano jakiekolwiek mechanizmy filtrowania bądź „unieszkodliwiania" znaków sterujących. Jeśli wystąpi błąd zapytania, programista ma jedynie możliwość obsłużenia i wyświetlenia komunikatu o błędzie w tym konkretnym punkcie wyniku. W tym momencie spora część wyniku mogła już dotrzeć do przeglądarki użytkownika. Kod generujący poprzedni wynik musi założyć powodzenie wykonania zapytania. Problem komplikuje następująca kwestia: co się stanie, jeśli aplikacja

musi obsłużyć inne mechanizmy obsługi baz danych? Zapytanie oraz kod bezpośrednio komunikujący się z bazą danych nie może pozostać wewnątrz kodu renderowania wyniku, jeśli ma być zachowana jego łatwość pielęgnacji oraz użyteczność.

## 7.2. Wzorzec projektowy Model-Widok-Kontroler

Wzorzec projektowy MVC jest jednym z częściej używanych sposobów rozdzielania od siebie warstw logiki aplikacji, danych i prezentacji. Jak opisano w rozdziale 3., „Architektura aplikacji po stronie klienta", wzorzec ten oddziela logikę obsługi danych od logiki biznesowej oraz logiki prezentacji. Dzięki temu na przykład wiele części aplikacji może współdzielić jedną implementację logiki.

Opisany wcześniej problem wstawianych w kodzie zapytań do bazy danych i logiki biznesowej nigdy nie może się zdarzyć w aplikacji korzystającej z modelu MVC, ponieważ warstwa prezentacji z samej definicji nie ma sposobu na to, by poznać metodę przechowywania danych, nie mówiąc już o użyciu zapytań i bibliotek funkcji specyficznych dla tej metody przechowywania danych. Każdy z formatów wyniku, który należy zaimplementować w aplikacji (XHTML, XML, JSON, itd.), współdzieli tę samą logikę pobierania danych dzięki użyciu tych samych obiektów, a nie bezpośrednich wywołań. Jeśli zachodzi potrzeba modyfikacji dowolnego elementu tej logiki — na przykład jeśli aplikacja wymaga obsługi innego silnika bazy danych, można zmodyfikować odseparowaną logikę bez wpływu na kod prezentacji lub nawet logiki biznesowej. Ten sam problem dotyczy odseparowania funkcji generowania skrótów, opisanej we wcześniejszej części tego rozdziału.

### 7.2.1. Model

Logika obsługi danych w aplikacji standardowo koncentruje się wokół interakcji z pamięcią masową aplikacji, zazwyczaj z bazą danych. W rzeczywistości jednak nie ma to wielkiego znaczenia, ponieważ sama logika obsługi danych jest niezależna od typu pamięci masowej. Logika obsługi danych powinna dotyczyć tylko takich operacji, jak uprawnienia użytkowników, obsługa błędów oraz zależności. Aby to zapewnić, można wykorzystać mechanizm dziedziczenia w celu usunięcia metod obsługi pamięci masowej z logiki obsługi danych.

Podobnie jak każdy obiekt w większości języków obiektowych rozszerza klasę lub podklasę klasy `Object`[2], wszystkie obiekty danych w aplikacji mogą rozszerzać klasę bazową `DBO` (opisującą obiekty obsługi bazy danych). Klasa ta zawiera cały kod niezbędny do tworzenia, odczytywania, aktualizowania i usuwania rekordów:

```
/**
 * Klasa bazowa obiektów obsługi bazy danych.
 */
class DBO {
    // Klucze główne niektórych tabel mają inne nazwy.
    public $pk = 'id';
    // Nazwa tabeli bazy danych.
    protected $table;
    // Nazwa tabeli bazy danych po „unieszkodliwieniu" znaków sterujących.
    private $table_mysql;
    // Tablica asocjacyjna pól tabeli do przechowywania ich wartości.
    protected $fields = array('id' => null);
    // Tablica z kluczami $fields po poddaniu ich operacji „unieszkodliwiania" — tylko do
    // wewnętrznego użytku.
    private $fields_mysql;
    // Tablica opisująca ograniczenia dotyczące typu i rozmiaru poszczególnych pól.
    protected $fields_constraints = array();
    // Tablica pól zaktualizowanych w określonym egzemplarzu.
    protected $updated = array();
    // Flaga decydująca o tym, czy należy wywołać metodę insert() czy update() w momencie
    // wywołania metody save().
    protected $inserted = false;

    /**
     * Jeśli pole istnieje, metoda zwraca jego bieżącą wartość,
     * w innym przypadku zwraca false.
     */
    public function get($var) {
        if (array_key_exists($var, $this->fields)) {
            return $this->fields[$var];
        } else {
            return null;
        }
    }

    /**
     * Jeśli pole istnieje, metoda aktualizuje wartość i oznacza jego miejsce w
     * zaktualizowanej tablicy, tak by skrypt aktualizacyjny miał informacje o polach, które ma
     * przetwarzać.
```

[2] W języku PHP rolę klasy `Object` spełnia klasa `stdClass`.

```
     */
    public function set($field, $value) {
        if (array_key_exists($field, $this->fields)) {
            if ($this->fields[$field] != $value) {
                // Zgłoszenie wyjątku.
                if ($this->meetsFieldConstraints($field, $value)) {
                    $this->fields[$field] = $value;
                    $this->updated[$field] = true;
                } else {
                    return false;
                }
            }
            return true;
        } else {
            return false;
        }
    }

    /**
     * Sprawdzenie ograniczeń pola w celu stwierdzenia, czy
     * podana wartość spełnia wymagania. Metoda zwraca
     *  true, jeśli warunki ograniczeń są spełnione lub
     * zgłasza wyjątek w przeciwnym przypadku.
     */
    protected function meetsFieldConstraints($field, $value) {
        // Jeśli nie zdefiniowano ograniczeń, to nie ma czego sprawdzać.
        if (isset($this->fields_constraints[$field])) {
            // Sprawdzenie typu.
            if (isset($this->fields_constraints[$field]['type'])) {
                Utilities::assertDataType(
                    $this->fields_constraints[$field]['type'],
                    $value
                );
            }
            // Sprawdzenie rozmiaru.
            if (isset($this->fields_constraints[$field]['size'])) {
                Utilities::assertDataSize(
                    $this->fields_constraints[$field]['size'],
                    $value
                );
            }
        }

        return true;
    }

    /**
     * Metoda pomocnicza umożliwiająca ustawienie wielu pól
     * na raz dzięki użyciu tablicy asocjacyjnej.
```

```
 */
public function setAssoc($array) {
    if (is_array($array)) {
        foreach ($array as $field => $value) {
            $this->set($field, $value);
        }
    } else {
        return false;
    }
}

/**
 * Metoda save() sprawdza flagę inserted w celu podjęcia decyzji o tym, czy wstawiać
 * nowy rekord, czy też aktualizować istniejący.
 */
public function save() {
    if ($this->inserted) {
        return $this->update();
    } else {
        return $this->insert();
    }
}

/**
 * Usunięcie rekordu na podstawie jego klucza głównego.
 */
public function delete() {
    $statement = $this->database->prepare(
        'DELETE FROM ' . $this->table_mysql . ' WHERE '
        . $this->fields_mysql[$this->pk] . ' = ?'
    );
    if ($statement->execute(array($this->fields[$this->pk]))) {
        $this->inserted = false;
        return true;
    } else {
        return false;
    }
}

/**
 * Ustawienie zaktualizowanych pól rekordu na nowe wartości.
 */
protected function update() {
    if (!in_array(true, $this->updated)) {
        return true;
    }
    $qry = 'UPDATE ' . $this->table_mysql . ' SET ';
    $f = false;
    foreach ($this->updated as $field => $value) {
```

```
            if (!$f) {
                $f = true;
            } else {
                $qry .= ', ';
            }
            $qry .= $this->fields_mysql[$field] . ' = ? ';
        }
        $qry .= ' WHERE ' . $this->fields_mysql[$this->pk] . ' = ? ';
        $statement = $this->database->prepare($qry);
        // Pobiera zaktualizowane wartości pól i dodaje klucz główny
        // dla klauzuli WHERE.
        $parameters = array_push(
            array_intersect_key($this->fields, $this->updated),
            $this->fields[$this->pk]
        );
        if ($statement->execute($parameters)) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * Wprowadzenie bieżących wartości do nowego rekordu bazy danych.
     */
    public function insert() {
        $qry = 'INSERT INTO ' . $this->table_mysql . ' ('
                . implode(', ', $this->fields_mysql)
                . ') VALUES ('
                . str_repeat('?,', count($this->fields) - 1) . '?)';
        $statement = $this->database->prepare($qry);
        if ($statement->execute($this->fields)) {
            $this->inserted = true;
            $this->fields[$this->pk] = mysql_insert_id();
            return true;
        } else {
            $GLOBALS['messenger']->addError(
                $this->database->errorInfo()
            );
            return false;
        }
    }

    /**
     * Alias metody DBO::select($pk, $id);
     */
    public function load($id) {
        $fields = array($this->pk);
```

```
    $values = array($id);
    return $this->select($fields, $values);
}

/**
 * Wybór rekordu na podstawie tablicy pól w celu porównania
 * z tablicą wartości.
 */
public function select($fields, $values) {
    global $config;
    if (is_array($fields) && is_array($values)) {
        $qry = 'SELECT ('
            . implode(', ', $this->fields_mysql)
            . ') FROM ' . $this->table_mysql . ' WHERE ';
        $f = false;
        foreach ($fields as $i => $field) {
            if (isset($this->fields_mysql[$field])) {
                if (!$f) {
                    $f = true;
                } else {
                    $qry .= ' AND ';
                }
                $qry .= $this->fields_mysql[$field] . ' = ? ';
            }
        }
        $statement = $this->database->prepare($qry);
        if ($statement->execute($values)) {
            if ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
                $this->fields = $row;
                $this->inserted = true;
                return true;
            }
        } else {
            $error = $statement->errorInfo();
            $GLOBALS['messenger']->add($error[2], 'error');
        }
    }
    return false;
}

/**
 * Ponieważ rozszerzenie PDO nie „unieszkodliwia" identyfikatorów tabeli i pól,
 * ta metoda tworzy prywatną, „nieszkodliwą" i ujętą w cudzysłów kopię
 * identyfikatorów tabeli i pól do wykorzystania w zapytaniach SQL.
 */
protected function escapeIdentifiers() {
    $this->table_mysql = $this->escapeTable($this->table);
    foreach ($this->fields as $field => $value) {
        $this->fields_mysql[$field] = $this->escapeIdentifier($field);
```

```
        }
    }

    /**
     * Nazwy tabel podlegają różnym ograniczeniom.
     * Dodatkowo w bazie danych MySQL nazwy tabel nie mogą kończyć się spacją ani
     * zawierać znaków "/", "\" lub "."
     */
    protected function escapeTable($string) {
        // Nazwy tabel w systemie MySQL podlegają nieco innym
        // ograniczeniom.
        $temp = preg_replace('/[\/\\.]/D', '', $string);
        $temp = str_replace('`', '``', $temp);
        return '`' . trim($temp) . '`';
    }

    /**
     * W nazwach pól należy „unieszkodliwić" wszystkie lewe apostrofy (ang. backtick).
     */
    protected function escapeIdentifier($string) {
        return '`' . str_replace('`', '``', $string) . '`';
    }

    /**
     * Kiedy obiekt wywołujący podaje identyfikator ID, należy wywołać metodę DBO::load()
     * w celu załadowania rekordu.
     */
    public function __construct($id = null) {
        global $controller;
        $this->database = $controller->getDatabaseHandle();
        if (!is_null($id)) {
            $this->load($id);
        }
        $this->escapeIdentifiers();
    }
}
```

Zaprezentowana klasa `DBO` implementuje wszystkie podstawowe metody niezbędne do zarządzania odpowiednikami rekordów bazy danych w postaci obiektów danych w PHP. Klasa zamyka w sobie bezpośrednie operacje z bazą danych potrzebne do zarządzania indywidualnymi rekordami. Dzięki wykorzystaniu rozszerzenia PDO ich pisanie jest znacznie łatwiejsze, zwłaszcza kiedy aplikacja wymaga zapewnienia przenośności do innych systemów baz danych. Dla zapytań również należałoby zapewnić mechanizm obsługi konkretnych baz danych. Przenośność aplikacji do innych systemów baz danych bez wielkiego wpływu na

architekturę aplikacji oraz logikę w warstwie modelu implementacji wzorca MVC można zapewnić dzięki wykorzystaniu biblioteki generowania zapytań, biblioteki zapisanych wstępnie zapytań lub dowolnej innej metody abstrakcji kodu SQL.

> Zazwyczaj cała warstwa obsługi bazy danych jest umieszczona pomiędzy obiektami danych a operacjami z bazą danych — w klasach dostępu do danych. Modyfikacje schematu, obsługa dodatkowych silników baz danych, a nawet przechowywanie danych poza bazami danych są dzięki temu o wiele łatwiejsze. W niniejszej książce nie opisano tego zagadnienia, głównie dlatego, aby na architekturze strony serwerowej aplikacji sterowanych wywołaniami Ajaksa skoncentrować się w tym rozdziale.

Dzięki rozszerzeniom tego obiektu fragmenty warstwy modelu w aplikacji mogą zawierać dokładnie tyle logiki, ile potrzeba bez konieczności zaśmiecania kodu źródłowego kodem obsługi pamięci masowej. Dzięki temu można implementować dodatkowe własności, takie jak zamieszczona poniżej klasa `Session` będąca rozszerzeniem klasy `DBO`. Jej celem jest umożliwienie dostępu do danych sesji za pośrednictwem tego samego ogólnego interfejsu przy jednoczesnym zapewnieniu zarządzania rekordem sesji w bazie danych:

```
// W aplikacji należy włączyć ten plik tylko raz
// i rozpocząć sesję tylko raz.
session_start();

/**
 * Klasa Session zarządza danymi w tabeli user_sessions ,
 * przy czym jej podstawowym przeznaczeniem jest zarządzanie sesją PHP.
 * Strukturę tę można by równie dobrze wykorzystać do zapisania wszystkich danych sesji
 * w tabeli bazy danych, zamiast używania wbudowanych mechanizmów obsługi sesji języka PHP,
 * bez konieczności zmian w interfejsie obiektowym.
 */
class Session extends DBO {
    public $pk = 'session';
    // Odwołanie do zmiennej superglobalnej $_SESSION.
    protected $session;
    // Tabela łącząca dane użytkowników z tablicą $_SESSION.
    protected $table = 'user_sessions';
    protected $fields = array(
                        'user' => null,
                        'session' => null
                    );
```

```
/**
 * Odtworzenie identyfikatorów sesji poprawia bezpieczeństwo, zwłaszcza w przypadku
 * wywołania po pomyślnym zalogowaniu z wykorzystaniem danych identyfikacyjnych.
 */
public function regenerate() {
    session_regenerate_id(true);
    $this->fields[$this->pk] = session_id();
    return $this->save();
}

/**
 * Metoda Session::get() przeciąża metodę DBO::get() w celu
 * obsługi przezroczystego pobierania informacji
 * z sesji.
 */
public function get($key) {
    if ($key == 'id') {
        return session_id();
    } else if ($key == 'user') {
        return $this->fields['user'];
    } else if (isset($this->session[$key])) {
        return $this->session[$key];
    } else {
        return false;
    }
}

/**
 * Metoda Session::set() przeciąża metodę DBO::set() w celu
 * obsługi przezroczystego ustawiania informacji
 * dotyczących sesji.
 */
public function set($key, $value) {
    if ($key == 'id') {
        return false;
    } else if ($key == 'user') {
        $this->fields['user'] = $value;
    } else {
        $this->session[$key] = $value;
    }
    return true;
}

/**
 * Ponieważ wartość klucza głównego pochodzi z żądania
 * (za pośrednictwem sesji w przeglądarce), metoda Session::load
 * powinna umożliwiać automatyczną obsługę przekazywanych danych.
 */
```

```
    public function load() {
        return parent::load($this->fields[$this->pk]);
    }

    /**
     * Przeciążenie konstruktora w celu utworzenia referencji do
     * zmiennej superglobalnej $_SESSION.
     */
    public function __construct() {
        global $_SESSION;
        parent::__construct();
        $this->session = $_SESSION;
    }
}
```

Klasa `Session` ma stosunkowo prostą implementację, ponieważ jej nadrzędna klasa `DBO` implementuje wszystko, co jest potrzebne do zarządzania odpowiadającym jej rekordem bazy danych. Ponieważ egzemplarze klasy `Session` również mogą mieć prosty interfejs obiektowy, uwierzytelnianie użytkownika na podstawie jego sesji i ładowanie danych tego użytkownika do obiektu `User` można wykonać za pomocą poniższego prostego kodu umieszczonego wewnątrz metody klasy `User`:

```
if ($this->session->load()) {
    if ($userid = $this->session->get('user')) {
        if ($this->load($userid)) {
            //Uwierzytelniony użytkownik z prawidłową sesją.
        } else {
            // Nieprawidłowy lub przeterminowany rekord sesji zwracający niepoprawny
            // identyfikator użytkownika.
        }
    } else {
        // Użytkownik z anonimową sesją.
    }
} else {
    // Użytkownik z całkowicie nową sesją.
}
```

Ściślejsza kontrola błędów, rejestrowanie i informowanie użytkownika o wykonywanych operacjach poprawiłyby „elegancję" tego kodu, trzeba jednak podkreślić, że w tym przykładzie logika interakcji z egzemplarzem klasy `Session`, mająca na celu określenie jej stanu z dokładnością do czterech stopni szczegółowości, zajęła tylko pierwsze dwa wiersze. Dane sesji można by zapisać z wykorzystaniem wbudowanych metod obsługi sesji języka PHP, niestandardowej tabeli bazy danych,

tymczasowych plików XML lub po prostu w pamięci. Metody obsługi danych nie mają wpływu na interfejs obiektowy dostępu do danych, a dzięki abstrakcji operacje zapisu i odczytu danych, niezależnie od ich docelowej lokalizacji, są trywialne.

## 7.2.2. Kontroler

Kontroler w architekturze MVC zawiera całą logikę aplikacji. W jego obrębie jest cały kod dotyczący interakcji z obiektami. Kontroler obsługuje również potrzebne operacje oraz wszystkie inne działania wymagane przez aplikacje, które nie mieszczą się w zakresie zarządzania danymi oraz logiki prezentacji. Do jego zadań należą testy autoryzacyjne związane z operacjami (za te, które dotyczą danych, jest odpowiedzialna warstwa modelu), ładowanie zasobów oraz cała logika biznesowa związana z operacjami. Kontroler ładuje również zasoby wymagane przez warstwę Widok oraz przekazuje dane i zasoby potrzebne do renderowania strony.

### 7.2.2.1. Zagnieżdżone kontrolery

W celu zapobieżenia konieczności kodowania tego samego kodu architektury w każdym obiekcie kontrolera aplikacja może zawierać obiekt centralnego kontrolera, którego zadaniem jest wykonywanie tych wspólnych zadań. Każdy kontroler zagnieżdżony wewnątrz centralnego kontrolera skupia się tylko na tej logice, która go dotyczy. Dzięki temu znacznie łatwiejsze jest również wprowadzanie modyfikacji w architekturze, w przypadku kiedy zachodzi taka potrzeba na dalszych etapach projektowania.

Centralny kontroler w przykładzie zamieszczonym poniżej jest maksymalnie ograniczony i spełnia jedynie rolę solidnej osi dla aplikacji. Jego zadaniem jest zainicjowanie obsługi żądania, utworzenie połączenia z bazą danych oraz skonfigurowanie środowiska dla pozostałej części aplikacji. W dalszej kolejności centralny kontroler wyznacza kontroler wymagany dla wybranego żądania, ładuje go i przekazuje do niego żądania w celu wykonania kodu związanego z wybranym fragmentem zestawu funkcji aplikacji.

Struktura ta w maksymalnym stopniu oddziela logikę związaną z samą architekturą od właściwej logiki aplikacji, dzięki czemu kod staje się bardziej czytelny i łatwiejszy do pielęgnacji. W ten sposób aplikacja może załadować poszczególne funkcje w miarę potrzeb zamiast ładowania dużych fragmentów kodu dla każdego żądania.

Zaprezentowana poniżej klasa `CentralController` nie obsługuje logiki potrzebnej do zarządzania żądaniami w większym stopniu niż jest to konieczne do załadowania właściwego kontrolera dla tego obszaru funkcji aplikacji. Dzięki zagnieżdżaniu kontrolerów w ten sposób logika szkieletu aplikacji może pozostać we własnej, centralnej klasie, natomiast każdy z kontrolerów podrzędnych obsługuje swoją własną logikę. Dzięki tej dodatkowej warstwie abstrakcji klasy są znacznie bardziej czytelne. Poza tym (oraz dzięki prostej klasie `Utilities` lub dostępnej globalnie bibliotece funkcji) każda klasa zawiera tylko tyle kodu, ile jest niezbędne do spełnienia swojej roli.

```
class CentralController {
    // Alias tablicy konfiguracyjnej.
    protected $config;
    // Odwołania do zmiennych globalnych.
    protected $raw_get;
    protected $raw_post;
    protected $raw_request;
    protected $raw_headers;
    // Kontroler dla wybranego fragmentu.
    protected $controller;
    // Odwołanie do bieżącego użytkownika.
    protected $user;

    public function handleRequest($get, $post = null, $request = null) {
        $this->raw_get = $get;
        $this->raw_post = (is_null($post)) ? array() : $post;
        $this->raw_request = (is_null($request)) ? array() : $request;
        try {
            $this->loadUser();
            $this->loadController();
            $this->passTheBuck();
        } catch (Exception $e) {
            // Strona z komunikatem o błędzie krytycznym.
        }
    }

    protected function loadUser() {
        try {
            Utilities::loadModel('User');
            $this->user = new User();
            $this->user->authenticate();
        } catch (Exception $e) {
            exit($e->getMessage());
        }
    }
```

```
protected function loadController() {
    global $controllers;

    // Jeśli nie określono kontrolera, wykorzystanie kontrolera domyślnego.
    $controller_key = 'default';
    if (isset($this->raw_get['c'])
            && isset($controllers[$this->raw_get['c']])) {
        $controller_key = $this->raw_get['c'];
    }
    // Wyszukanie kontrolera lub zgłoszenie wyjątku.
    $controller_path = 'controllers/'
            . $controllers[$controller_key]['filename'] . '/'
            . $controllers[$controller_key]['filename'] . '.php';
    // Sprawdzenie, czy plik istnieje przed jego załadowaniem na wypadek, gdyby jego lokalizacja
    // zmieniła się od chwili generowania listy dostępnych kontrolerów.
    if (!file_exists($controller_path)) {
        throw new Exception('Kontrolera nie znaleziono');
    }
    // Załadowanie pliku i utworzenie egzemplarza obiektu kontrolera.
    include $controller_path;
    $this->controller = new $controllers[$controller_key]['class']();
    return true;
}

/**
 * Prosta metoda do ładowania nagłówków żądania HTTP.
 * Zwraca żądaną wartość, jeśli ona istnieje.
 */
public function getHeader($key) {
    // Późne ładowanie nagłówków apache.
    if (!isset($this->raw_headers)) {
        $this->raw_headers = apache_request_headers();
    }
    if (isset($this->raw_headers[$key])) {
        return $this->raw_headers[$key];
    } else {
        return false;
    }
}

protected function passTheBuck() {
    $this->controller->handleRequest(
        $this->raw_get,
        $this->raw_post,
        $this->raw_request
    );
}

public function getDatabaseHandle() {
```

```
        if (!isset($this->database)) {
            $this->loadDatabase();
        }
        return $this->database;
    }

    protected function loadDatabase() {
        $this->database = new PDO(
            $this->config['database']['dsn'],
            $this->config['database']['username'],
            $this->config['database']['password'],
            $this->config['database']['options']
        );
    }

    public function display() {
        $this->controller->display();
    }

    public function __construct() {
        include 'configuration.php';
        $this->config = $config;
    }
}
```

Ten kontroler może również rozwiązywać problem dostarczania co najmniej jednego tokenu dla każdego zagnieżdżonego kontrolera. Powinien on być unikatowy dla sesji użytkownika i zakodowany z wykorzystaniem losowego ciągu z klasy `Utilities`. Praktyka ta pozwala na to, aby zagnieżdżone kontrolery sprawdzały obecność tokenu walidacyjnego właściwego dla operacji, dla której należy zapewnić autoryzację. W rezultacie napastnikom jest o wiele trudniej przeprowadzić atak CSRF.

Logika danego fragmentu aplikacji jest oddzielona od logiki pozostałych fragmentów dzięki wykorzystaniu implementacji obiektu `CentralController`, przy czym każdy fragment uzyskuje dodatkową warstwę zabezpieczeń. Zamieszczone poniżej zmienne i metody dodane do obiektu `CentralController` umożliwiają sprawdzanie tokenu walidacyjnego za pośrednictwem wymaganego sposobu (zmiennej POST lub nagłówka HTTP) poprzez wywołanie pojedynczej metody obiektu `CentralCon` ↪`troller`:

```
// Flaga typu Boolean wskazująca na to, czy token walidacyjny jest prawidłowy.
protected $validated = false;
protected $validation_token;
```

```
/**
 * Pobranie tokenu właściwego dla bieżącego obszaru aplikacji,
 * ale tylko wtedy, gdy użytkownik przeszedł z innego obszaru.
 */
protected function generateValidationToken($area) {
    // Pobranie ostatnio przeglądanego obszaru zapisanego w sesji.
    $last_viewed = $this->user->session->get('last_viewed_area');
    // Ponowne wygenerowanie tokenu i zastosowanie go dla sesji,
    // w przypadku, gdy jest to obszar różny od bieżącego.
    if ($area != $last_viewed) {
        $session = $this->user->session->get('id');
        $this->validation_token = Utilities::generateToken($area . $session);
        $this->user->session->set('last_viewed_area', $area);
    }
}

/**
 * Walidacja tokenu z nagłówkami żądania.
 */
public function validateHeader() {
    return $this->validateToken(
        $this->getHeader(
          $this->config['settings']['validation']
        )
    );
}

/**
 * Walidacja tokenu z danymi POST.
 */
public function validatePost() {
    if (isset($this->raw_post[$this->config['settings']['validation']])) {
        return $this->validateToken(
          $this->raw_post[$this->config['settings']['validation']]
        );
    }
}

/**
 * Sprawdzenie, czy bieżący token oraz token żądania są ze sobą zgodne.
 */
public function validateToken($test) {
    return ($test === $this->validation_token);
}
```

Teraz obiekt `CentralController` na podstawie bieżącego żądania musi jedynie wywołać metodę w celu wygenerowania tokenu do wykorzystania w metodach walidacyjnych. Ponieważ w związku z tym token będzie zależny od kontrolera,

dobrym rozwiązaniem będzie wykorzystanie klucza dla kontrolera w tablicy asocjacyjnej skonfigurowanej wcześniej. Metoda `loadController()` może następnie wywołać metodę generowania tokenu po tym, gdy pomyślnie utworzy egzemplarz kontrolera z wykorzystaniem tego samego klucza:

```
protected function loadController() {
    global $controllers;

    // Jeśli nie określono kontrolera, wykorzystanie kontrolera domyślnego.
    $controller_key = 'default';
    if (isset($this->raw_get['c'])
            && isset($controllers[$this->raw_get['c']])) {
        $controller_key = $this->raw_get['c'];
    }
    // Wyszukanie kontrolera lub zgłoszenie wyjątku.
    $controller_path = 'controllers/'
            . $controllers[$controller_key]['filename'] . '/'
            . $controllers[$controller_key]['filename'] . '.php';
    // Sprawdzenie, czy plik istnieje przed jego załadowaniem na wypadek, gdyby lokalizacja
    // zmieniła się od chwili generowania listy dostępnych kontrolerów.
    if (!file_exists($controller_path)) {
        throw new Exception('Kontrolera nie znaleziono');
    }

    // Wygenerowanie tokenu walidacji żądania.
    $this->generateValidationToken($controller_key);

    // Załadowanie pliku i utworzenie egzemplarza obiektu kontrolera.
    include $controller_path;
    $this->controller = new $controllers[$controller_key]['class']();
    return true;
}
```

Teraz, kiedy obiekt `CentralController` potrafi obsłużyć wstępne żądanie, zainicjować połączenie z bazą danych, podjąć próbę załadowania i uwierzytelniania użytkownika, załadować w dynamiczny sposób kontroler ze zbioru zagnieżdżonych kontrolerów oraz zapewnić dostępną globalnie prostą ochronę przed atakami CSRF, zagnieżdżone kontrolery mogą działać na bazie tej warstwy zgodnie z własnymi wymaganiami.

Zagnieżdżony kontroler, zamieszczony poniżej, wykonuje tylko jedną operację: połączenie formularza rejestracyjnego użytkownika z obiektem `User` w celu utworzenia rekordu w bazie danych po tym, gdy użytkownik wprowadzi wszystkie potrzebne informacje. Wszystkie elementy z warstwy bazy danych są oddzielone od widoku, kontroler jedynie zaopatruje widok w dane i obsługuje jego odpowiedzi.

Umieszczenie niektórych fragmentów kodu bazowego, niespecyficznych dla procesu rejestracji, miałoby więcej sensu w nadrzędnej klasie `Controller`, którą poniższy kontroler mógłby rozszerzać. Jednak dla zapewnienia przejrzystości opisu w tym rozdziale kontroler ten zdefiniowano w postaci pojedynczej klasy:

```
Utilities::loadModel('User');

class RegistrationController {
    // Odwołania do zmiennych globalnych.
    protected $raw_get;
    protected $raw_post;
    protected $raw_request;
    // Utworzenie odwołania do obiektu użytkownika.
    protected $user;
    protected $userinfo = array(
        'login' => null,
        'name' => null,
        'email' => null,
        'password' => null
    );
    // Obiekt obsługujący wyjście.
    protected $view;
    // Sposób odpowiedzi na żądanie.
    protected $method;

    /**
     * Pobranie metody żądania z widoku, utworzenie egzemplarza silnika
     * renderowania, ustawienie kontekstu renderowania do katalogu, w którym znajduje się ten plik,
     * odfiltrowanie żądania i podjęcie próby utworzenia rekordu użytkownika na podstawie
     * danych żądania.
     */
    public function handleRequest($get, $post = null, $request = null) {
        // Utworzenie silnika renderowania.
        $this->method = View::getMethodFromRequest($get);
        $this->view = View::getRenderingEngine($this->method);
        $this->view->setContext(dirname(__FILE__));
        // Zastosowanie filtra dla żądania.
        $this->filterRequest($get, $post, $request);
        // próba utworzenia nowego rekordu użytkownika z wykorzystaniem filtrowanego żądania.
        $this->createUser();
    }

    /**
     * Akceptacja danych żądania tylko wtedy, gdy obiekt CentralController dokona walidacji
     * wartości nagłówka lub zmiennej post w przypadku operacji ładowania pełnej strony (przesłanie
     * formularza).
     */
```

```
public function filterRequest($get, $post = null, $request = null) {
    global $controller;

    if ($controller->validateHeader()
            || ($this->method == View::METHOD_XHTML
                && $controller->validatePost())) {
        $this->raw_get = $get;
        $this->raw_post = (is_null($post)) ? array() : $post;
        $this->raw_request = (is_null($request)) ? array() : $request;
    } else {
        return false;
    }
}

/**
 * Próba utworzenia rekordu użytkownika w przypadku, gdy istnieją wszystkie pola.
 * Przekazanie komunikatów o błędach do obiektu Messenger.
 */
protected function createUser() {
    global $messenger;

    $this->user = new User();
    if ($this->getUserInfo()) {
        $errors_found = false;
        foreach ($this->userinfo as $field => $value) {
            try {
                $this->user->set($field, $value);
            } catch (Exception $e) {
                if (!$errors_found) {
                    $errors_found = true;
                }
                $messenger->add($e->getMessage(), $field);
            }
        }
        if (!$errors_found) {
            try {
                $this->user->save();
            } catch (Exception $e) {
                $messenger->add($e->getMessage(), 'error');
            }
        }
    }
}

/**
 * Pobranie wartości pól wszystkich użytkowników z żądania post
 * pod warunkiem, że hasło i jego potwierdzenie
 * są ze sobą zgodne.
 */
```

```
    public function getUserInfo() {
        global $messenger;
        foreach ($this->userinfo as $field => $value) {
            if (isset($this->raw_post[$field])) {
                if ($field == 'login' && User::loginExists($value)) {
                    $messenger->add('Nazwa użytkownika jest zajęta', 'login');
                    continue;
                } else if ($field == 'password') {
                    if (!isset($this->raw_post['password_confirm'])
                            || $this->raw_post[$field]
                               != $this->raw_post['password_confirm']) {
                        $messenger->add(
                         'Hasło i jego potwierdzenie muszą być ze sobą zgodne',
                         'password'
                        );
                        continue;
                    }
                }
                $this->userinfo[$field] = $this->raw_post[$field];
            }
        }
        return in_array(null, $this->userinfo);
    }

    /**
     * Wyświetlenie wyjścia mechanizmu renderowania z wykorzystaniem
     * odpowiedniego szablonu dla wybranego formatu żądania.
     */
    public function display() {
        switch ($this->method) {
            case View::METHOD_JSON:
                $this->view->setTemplate('json.php');
                break;
            case View::METHOD_XML:
                $this->view->setTemplate('xml.php');
                break;
            case View::METHOD_XHTML:
            default:
                $this->view->setTemplate('index.php');
                break;
        }
        $this->view->display();
    }
}
```

Pokazany powyżej obiekt `RegistrationController` obsługuje logikę rejestracji użytkownika zgodnie z wcześniejszym opisem. Próba utworzenia rekordu użytkownika jest wykonywana tylko wtedy, gdy w formularzu wprowadzono wszystkie

niezbędne pola. Obiekt sprawdza, czy podana nazwa użytkownika nie jest zajęta, i przekazuje informacje o błędach do obiektu `Messenger`. Błędy są podzielone na kategorie odpowiadające polom, których dotyczy dany błąd (lub są zaliczone do ogólnej kategorii błędów metody `User::save()`). Dzięki temu warstwa widoku może odpowiednio obsłużyć wszystkie komunikaty i błędy. Własność tę można również zaimplementować w postaci ogólnej warstwy usług. Dzięki temu bez konieczności przepisywania kodu inne obiekty mogą wykonać te same testy. Takie rozwiązanie powinno sprawdzić się zwłaszcza w przypadku bardziej złożonych aplikacji.

Metoda wyświetlania sprawdza sposób obsługi żądania i przypisuje odpowiedni szablon do silnika renderowania. Kontekst renderowania jest przypisany do katalogu skryptu bezpośrednio po utworzeniu mechanizmu renderowania w obsłudze żądania. Dzięki temu, jeśli zachodzi taka potrzeba, kontroler może do niego przypisać zmienne w innych metodach.

Teraz, kiedy w aplikacji występuje model i kontroler, widok będzie obsługiwał tylko te zadania, które są specyficzne dla niego samego. Będzie on przekazywał dane za pośrednictwem metod GET i POST oraz za pośrednictwem nagłówków w wywołaniach Ajaksa. Do realizacji tego celu nie jest potrzebna wiedza na temat sposobu interakcji z innymi elementami w pozostałych warstwach. Widok musi jedynie pytać egzemplarz obiektu `Messenger` o komunikaty oraz informacje o błędach, tak by można było podjąć decyzję o tym, co należy wyświetlić w wyniku.

### 7.2.3. Widok

W pokazanej przykładowej architekturze widok składa się z silnika renderowania, szablonów oraz architektury po stronie klienta opisanej w rozdziale 3. Podobnie jak w przypadku warstwy modelu, te klasy i szablony zawierają niewiele kodu związanego z innymi warstwami aplikacji.

Dzięki wyraźnemu odseparowaniu od aplikacji internetowej działającej po stronie serwera aplikacja działająca po stronie klienta może istnieć jako niemal całkowicie odrębna aplikacja. Aplikacja po stronie klienta wykorzystuje ją jedynie jako dostępny interfejs API. Taki podział zapewnia również niezwykłą elastyczność zarówno aplikacji działającej po stronie klienta, jak i serwera, ponieważ jedynym wymaganiem jest utrzymanie spójnych interfejsów obiektów.

### 7.2.3.1. Renderowanie

Programiści mogą użyć PHP jako języka obsługi szablonów — przykład takiego użycia języka PHP zaprezentowano w tym podpunkcie. Większość mechanizmów obsługi szablonów zawiera bogaty zbiór narzędzi do „unieszkodliwiania", przetwarzania w pętli oraz grupowania zbioru znaczników w logiczne fragmenty. W tym przykładzie ograniczono się do niezbędnego minimum w celu pokazania, że nawet niezwykle prosty silnik renderowania, wykorzystujący język PHP w roli swojego języka obsługi szablonów, w dalszym ciągu wymaga abstrakcji niezbędnej do działania warstwy widoku aplikacji.

Klasa `RenderingEngine`, zamieszczona poniżej, implementuje zasadnicze własności mechanizmu renderowania. Można do niej przypisać zmienne, które w momencie wyświetlania zostaną udostępnione szablonom. Klasa może zmienić kontekst tak, aby szablony mogły włączać pliki bez konieczności znajomości ścieżek dostępu do nich. Klasa przesyła dodatkowe nagłówki odpowiedzi, choć domyślnie nie obsługuje żadnych nagłówków, ponieważ jej jedynym przeznaczeniem jest umożliwienie rozszerzania jej przez inne klasy:

```
class RenderingEngine {
    // Katalog bazowy.
    protected $context = '.';
    // Nazwa katalogu zawierającego szablony.
    protected $templates = 'templates';
    // Nazwa pliku szablonu do wyświetlenia.
    protected $template = 'index.php';
    // Zapewnienie możliwości jawnego przesyłania zmiennych do szablonu.
    protected $variables = array();

    /**
     * Ustawienie katalogu bazowego, skąd mają być włączane pliki.
     */
    public function setContext($path) {
        $this->context = $path;
    }

    /**
     * Przesłonięcie domyślnego katalogu szablonów.
     */
    public function setTemplatesDirName($dir) {
        $this->templates = $dir;
    }

    /**
     * Metoda służąca do przekazania zmiennych do szablonu, dzięki czemu
```

```
     * szablon nie musi zajmować się ustawianiem wartości zmiennych
     * — jest to zadanie kontrolera.
     */
    public function setVariable($key, $value) {
        $this->variables[$key] = $value;
    }

    /**
     * Przesłonięcie domyślnej nazwy szablonu.
     */
    public function setTemplate($filename) {
        $this->template = $filename;
    }

    /**
     * Zmiana do przypisanego kontekstu, dzięki czemu wywołania włączania plików
     * wykonane z poziomu szablonu nie wymuszają od szablonu
     * znajomości własnej ścieżki.
     */
    public function display() {
        // Zapisanie informacji o bieżącym katalogu roboczym w zmiennej tymczasowej.
        $cwd = getcwd();
        chdir($this->context);
        $template_path = $this->templates . DIRECTORY_SEPARATOR . $this->template;
        if (file_exists($template_path)) {
            $this->sendHeaders();
            include $template_path;
            chdir($cwd);
        } else {
            chdir($cwd);
            throw new Exception(
                'Szablon "' . $template_path . '" nie istnieje.'
            );
        }
    }
}
```

Powyższa implementacja Widoku tworzy niezwykle prosty interfejs do wykorzystania go z poziomu Kontrolera. Klasa `RegistrationController` z wcześniejszej części niniejszego rozdziału wykorzystywała silnik renderowania poprzez wywołanie następujących czterech metod w różnych punktach przetwarzania:

```
$this->view->setContext(dirname(__FILE__));
$this->view->setTemplate('index.php');
$this->view->sendHeaders();
$this->view->display();
```

Trzecia z metod — `sendHeaders()`— występuje jako oddzielna metoda, ponieważ klasa `RenderingEngine` może zawierać zagnieżdżone egzemplarze do renderowania szablonów, które wyświetlają jeden fragment całego wyniku. Próba wysłania dodatkowych nagłówków podczas tej czynności nie tylko zakończyłaby się porażką, ale także zgłoszeniem błędu przez środowisko PHP — jest to bowiem niedozwolona operacja.

Kiedy silnik renderowania wyświetla szablony, robi to za pomocą instrukcji `include`. Powoduje to uruchomienie szablonów w kontekście samego silnika szablonów z dostępem do tablicy `variables` w obrębie obiektu. Szablony uzyskują także możliwość wywoływania metod obiektu. W ten sposób powstaje bardzo wygodny zasięg deklaracji metod unieszkodliwiania ciągów znaków i formatowania.

### 7.2.3.2. Szablony

Zaprezentowana architektura nie tylko znacznie ułatwia renderowanie wyniku w wielu formatach, ale także w przypadku formatu XHTML strona ma postać interfejsu w formie zakładek i kroków. Warstwy Kontrolera i Modelu nic o tym nie wiedzą i nie muszą wiedzieć, ponieważ wszystkie operacje mogą być obsłużone przez szablony, a architektura aplikacji po stronie klienta zmienia się w całości w interfejs sterowany żądaniami Ajax. Kiedy użytkownik wypełni bieżący zbiór pól, aplikacja po stronie klienta wysyła żądanie Ajax na serwer i ustawia te specyficzne pola. Dzięki temu można obsłużyć wszystkie błędy przed umożliwieniem użytkownikowi przejścia do następnej zakładki.

Same szablony w większości przypadków zawierają bardzo niewiele kodu PHP, ponieważ istnieją one głównie po to, by tworzyły zestaw znaczników obsługujący dane i funkcje aplikacji. Główny szablon strony rejestracyjnej zawiera zaledwie kilka fragmentów kodu PHP. Jeśli przeglądarka nie obsługuje Ajaksa i wymusza wykorzystanie ładowania pełnej strony, wówczas jedna ze stron ma za zadanie wybór zagnieżdżonego szablonu na podstawie bieżącego kroku:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Rejestracja użytkownika</title>
<link rel="stylesheet" type="text/css" href="style.css" />
<script type="text/javascript" src="../includes/main.lib.js"></script>
<script type="text/javascript" src="../includes/ajax.lib.js"></script>
<script type="text/javascript" src="../includes/effects.lib.js"></script>
<script type="text/javascript" src="controllers/default/javascripts/default.js"></
```

```
<script>
</head>
<body>
<h1>Przykład prostej rejestracji <acronym title="Interfejs
użytkownika">UI</acronym></h1>

<div class="demo">
    <ol id="registration_tabs" class="navigation_tabs">
        <li<?php if ($step == 1) { ?> class="selected"<?php } else if ($step > 1)
{ ?> class="completed"<?php } ?>>
            <a href="./?step=1">Konto</a>
            <span class="status">
                (<?php if ($step == 1) {
                    echo 'w trakcie';
                } else if ($step > 1) {
                    echo 'wykonany';
                } ?>)
            </span>
        </li>
        <li<?php if ($step == 2) { ?> class="selected"<?php } else if ($step > 2)
{ ?> class="completed"<?php } ?>>
            <a href="./?step=2">Profil</a>
            <span class="status">
                (<?php if ($step < 2) {
                    echo 'niekompletny';
                } else if ($step == 2) {
                    echo 'w trakcie';
                } else if ($step > 2) {
                    echo 'wykonany';
                } ?>)
            </span>
        </li>
        <li<?php if ($step == 3) { ?> class="selected"<?php } ?>>
            <a href="./?step=3">Potwierdzenie</a>
            <span class="status">
                (<?php if ($step < 3) {
                    echo 'niekompletny';
                } else if ($step == 3) {
                    echo 'w trakcie';
                } else if ($step > 3) {
                    echo 'wykonany';
                } ?>)
            </span>
        </li>
    </ol>
    <?php
    if ($step == 3) {
        include 'step3.php';
    } else if ($step == 2) {
        include 'step2.php';
```

```
    } else {
        include 'step1.php';
    }
    ?>
</div>
</body>
</html>
```

Jeszcze mniej skomplikowane są szablony odpowiedzi Ajaksa, ponieważ w ogóle nie zawierają formatowania i spełniają wyłącznie rolę posłańców dla aplikacji po stronie serwera. W zależności od tego, czy aplikacja korzysta z formatu XML czy JSON, wykorzystuje ona jeden z dwóch poniższych szablonów:

```
<?php
global $messenger;
$messages = $messenger->getQueue();

echo "[\n";
for ($i = 0; $i < count($messages); $i++) {
    if ($i > 0) {
        echo "\n,";
    }
    echo '{"type":"',
        $this->escape($messages[$i]->type),
        '","',
        $this->escape($messages[$i]->content),
        '"}';
}
echo "\n]";
?>
```

Szablon formatu JSON wyświetla całą zawartość szablonu za pomocą instrukcji `echo` — format JSON zawiera tak mało znaków, że próba oddzielenia go od wyjścia PHP pogorszyłaby tylko jego czytelność i utrudniła pielęgnację:

```
<?php
global $messenger;
$messages = $messenger->getQueue();
?><?xml version="1.0"?>
<messages>
    <?php for ($i = 0; $i < count($messages); $i++) { ?>
    <message type="<?php echo $this->escape($messages[$i]->type); ?>">
        <?php echo $this->escape($messages[$i]->content); ?>
    </message>
    <?php } ?>
</messages>
```

W szablonie dotyczącym formatu XML zestaw znaczników jest jednak znacznie bardziej obszerny. W dalszym ciągu zawiera on niewiele znaczników PHP, ale jest czytelny. Wyjście zarówno w formacie JSON, jak i XML renderuje się tylko wtedy, gdy zostanie „skonsumowane" przez kod JavaScript w aplikacji po stronie klienta i w związku z tym nie wymaga żadnej dodatkowej logiki z kontrolera. Pomimo wszystko wprowadzenie danych do tego wyjścia nie wprowadza dodatkowych komplikacji do szablonów bądź mechanizmu obsługi szablonów. Jedynym wymaganiem jest to, aby przed renderowaniem kontroler był zdolny do obsługi logiki niezbędnej do odczytania danych i przypisania ich do mechanizmu obsługi szablonów.

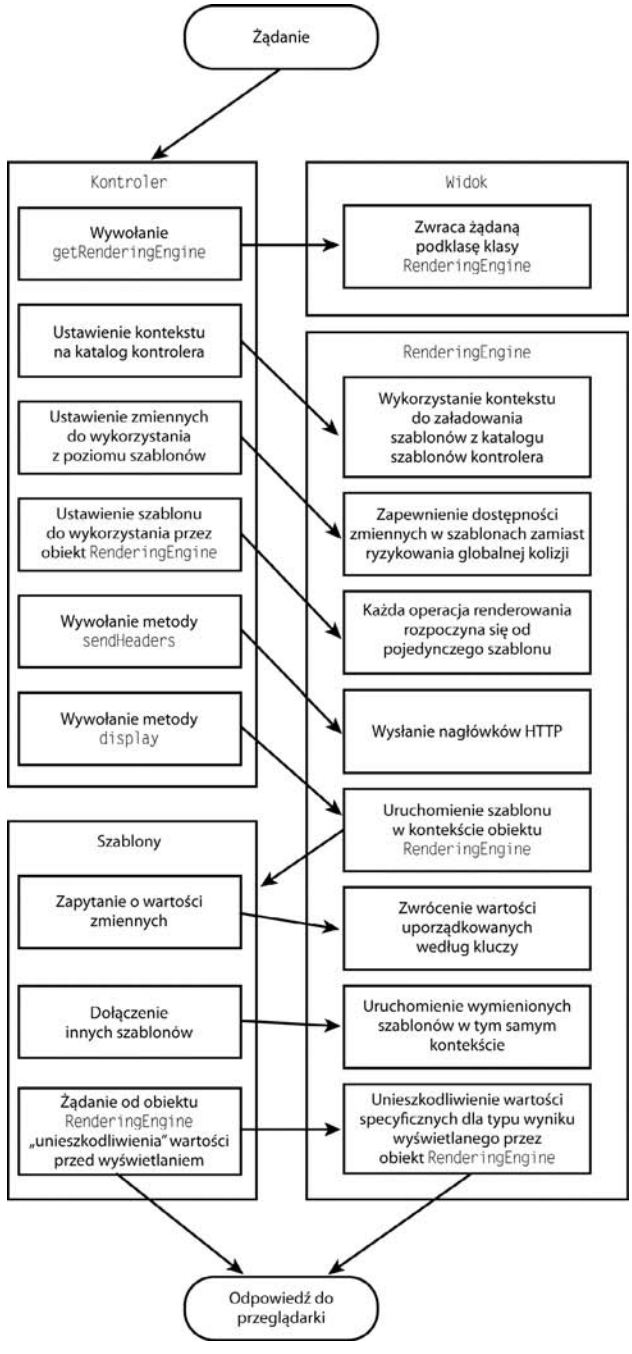
## 7.3. Wykorzystanie wzorca Fabryka wraz z mechanizmem obsługi szablonów

Na podstawie przekazanych parametrów we wzorcu Fabryka, pokazanym na rysunku 7.1, wykorzystano ogólny interfejs w celu utworzenia egzemplarza wybranej klasy podrzędnej obiektu. W kontekście mechanizmu obsługi szablonów Fabryka powinna zwrócić obiekt mechanizmu szablonów, który byłby skonfigurowany dla określonego formatu odpowiedzi — na przykład XHTML. Kod żądający utworzenia egzemplarza tego obiektu nie musi znać typu zwracanego obiektu szablonu. Powinien jedynie wiedzieć, w jaki sposób obsługiwać obiekty szablonów.

Powyższa struktura bazuje na ogólnym interfejsie do każdego z obiektów zarządzających szablonami i renderowaniem dla wybranego formatu wyjścia. Dzięki zaprojektowaniu architektury w ten sposób dowolna część aplikacji internetowej może obsługiwać wyjście do innego formatu bez konieczności modyfikacji logiki poszczególnych komponentów. Klasa `View` implementuje metody abstrakcyjne wykorzystywane w klasie `RegistrationController` zaprezentowanej w poprzednim punkcie:

```
class View {

    public static $METHOD_KEY = 'm
ethod';
    public static $METHOD_JSON = 'json';
    public static $METHOD_XHTML = 'xhtml';
    public static $METHOD_XML = 'xml';
    protected static $loaded = array();
```




RYSUNEK 7.1. Diagram przepływu danych dla wzorca Fabryka przypisanego do mechanizmów obsługi szablonów

```
/**
 * Abstrakcja w celu pobrania klucza mechanizmu renderowania z żądania.
 */
public static function getMethodFromRequest($request) {
    return (isset($request[self::METHOD_KEY]))
        ? $request[self::METHOD_KEY] : self::METHOD_XHTML;
}

/**
 * Zwraca egzemplarz mechanizmu renderowania dla określonego klucza.
 * Domyślnym formatem jest XHTML, jeśli żądany format nie jest obsługiwany.
 */
public static function getRenderingEngine($method) {
    global $views;

    if (self::loadRenderingEngine($method)) {
        return new $views[$method]['class']($request);
    } else if (self::loadRenderingEngine(self::$METHOD_XHTML)) {
        return new $views[self::$METHOD_XHTML]['class'](($request);
    }
    throw new Exception('Nie udało się załadować mechanizmu renderowania');
}

/**
 * Ładuje właściwy mechanizm renderowania dla podanego klucza.
 */
protected static function loadRenderingEngine($key) {
    global $views;

    // Czy próba ładowania nastąpiła wcześniej?
    if (isset(self::$loaded[$key])) {
        return self::$loaded[$key];
    }

    // W innym przypadku sprawdzenie obecności pliku.
    if (isset($views[$key])) {
        $path = 'views' . DIRECTORY_SEPARATOR
            . $views[$key]['filename'] . '.php';
        // Włączenie pliku, jeśli jest dostępny.
        if (file_exists($path)
                && is_readable($path) && include $path) {
            // Ustawienie flagi wskazującej na powodzenie operacji
            // w celu uniknięcia powielania tych samych informacji przy następnym żądaniu
            // dla tego samego widoku.
            return self::$loaded[$key]
                = class_exists($views[$key]['class']);
        }
    }
```

```
        // Nie udało się znaleźć mechanizmu renderowania ani się do niego odwołać — zwrócenie
        // wartości false.
        return false;
    }
}
```

Powyższa implementacja wzorca Fabryka oferuje tylko dwie metody w ramach publicznego interfejsu obiektu, trzecia metoda (chroniona) występuje jedynie w celu uniknięcia dublowania kodu w metodzie `View::getRenderingEngine()`. Każdy kontroler może obsługiwać wiele formatów wyniku poprzez wywołanie `View::getRenderingEngine(View::getMethodFromRequest($this->raw_get));`. Instrukcja ta powoduje utworzenie egzemplarza wymaganego mechanizmu renderowania.

Każdy mechanizm renderowania jest rozszerzeniem klasy `RenderingEngine` zaprezentowanej w poprzednim punkcie. Klasa `XHTMLRenderingEngine`, zamieszczona poniżej, rozszerza klasę `RenderingEngine`, która zaimplementowała ogólne zmienne obiektów oraz metody wykorzystywane przez wszystkie mechanizmy renderowania w tej aplikacji. W mechanizmie renderowania specyficznym dla formatu XHTML pozostało do zaimplementowania tylko to, co jest dla niego specyficzne:

```
class XHTMLRenderingEngine extends RenderingEngine {
    protected $headers = array(
        'Content-Type' => 'application/xml+xhtml'
    );

    /**
     * Metoda służąca do przesyłania formatu text/html do przeglądarek, które nie obsługują XHTML.
     */
    protected function sendHeaders() {
        global $controller;
        $accept = $controller->getHeader('Accept');
        if (!$accept
                || strpos($accept, $this->headers['Content-Type'])) {
            $this->headers['Content-Type'] = 'text/html';
        }
        return parent::sendHeaders();
    }

    /**
     * Krótszy sposób unieszkodliwiania encji XHTML.
     */
    public function escape($string) {
        return Utilities::escapeXMLEntities($string);
    }
}
```

Powyższa klasa mogłaby również implementować metody generowania znaczników `html` i `body`, a także bloku `head` wykorzystywanego przez wszystkie szablony XHTML. Pozwoliłoby to na uniknięcie dublowania znaczników w każdym z nich. Klasa `RenderingEngine` dla formatów RSS lub Atom mogłaby zawierać metody formatowania znaczników czasu w sposób wymagany przez każdą z tych specyfikacji.

Ponieważ klasa-fabryka `RenderingEngine` obsługuje dostępne w tablicy mechanizmy renderowania, aplikacja może z łatwością obsługiwać niestandardowy format dla wybranego kontrolera — wystarczy, aby po utworzeniu egzemplarza tego kontrolera przez obiekt `CentralController` został wpisany na listę obiekt `RenderingEngine` tego kontrolera. Dzięki temu kontroler obsługi raportów może wykorzystywać ten sam mechanizm szablonów co pozostała część aplikacji w celu generowania wyjścia bezpośrednio w formacie Microsoft Excel, PDF, SVG lub innych dowolnych formatach alternatywnych, które nie są wymagane w innych częściach aplikacji.