JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 7 (2018) 178-182
Wysłane: 2017-12-07
Przyjęte: 2017-12-11

Analiza użytkowa frameworka AngularJS w kontekście prostej aplikacji internetowej

Krzysztof Pawelec*

Politechnika Lubelska, Instytut Informatyki, Nadbystrzycka 36B, 20-618 Lublin, Polska

Streszczenie. Celem artykułu było przeprowadzenie analizy możliwości użytkowych frameworka frontendowego AngularJS w porównaniu do języka programowania JavaScript, na którym bazuje. Wybrano kilka funkcjonalności frameworka i zestawiono je z samodzielnie zaimplementowanymi rozwiązaniami w JavaScript. Porównania dokonano według określonych kryteriów: prostoty użycia, możliwości wykorzystania i zdatności do wielokrotnego zastosowania. Utworzone skrypty potwierdzają przyjęte założenia, że jest możliwe napisanie w JavaScript uproszczonych i zdolnych do ponownego użytku implementacji przydatnych mechanizmów znajdujących się w AngularJS.

Słowa kluczowe: framework; użyteczność; AngularJS; JavaScript

*Autor do korespondencji.

Adres e-mail: krzysztof.pawelec1@pollub.edu.pl

Usability analysis of AngularJS framework in the context of simple internet application

Krzysztof Pawelec*

Institute of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The goal of this article was to perform analysis of usability possibilities of frontend framework AngularJS compared to native programming language JavaScript, on which it is based on. Several framework funcionalities were chosen and set together with self implemented solutions in JavaScript. Comparison was made according to specified criteria: usage simplicity, possibilities of utilization and reusability. Created scripts confirm accepted assumptions, that in JavaScript it is possible to write simplified and reusable implementations of useful mechanisms, which are present in AngularJS.

Keywords: framework; usability; AngularJS; JavaScript

*Corresponding author.

E-mail address: krzysztof.pawelec1@pollub.edu.pl

1. Wstęp

W ostatnich latach istnieje coraz większe zapotrzebowanie na aplikacje internetowe, gdzie duża część logiki umiejscowiona jest po stronie frontendu [1]. Aby przyspieszyć proces ich tworzenia powstały frameworki, które potrafią wykonać coraz więcej często realizowanych przez programistę czynności związanych głównie z budową struktury aplikacji i zarządzaniu danymi pomiędzy jej warstwami [2]. Warto zastanowić się nad tym, czy do każdej aplikacji – bez względu na skomplikowanie i wielkość - potrzeba dołączać framework. Z uwagi na udostępnianie wielu funkcjonalności, nierzadko wymuszenie zastosowania określonej struktury aplikacji i posiadania zdolności do kontrolowania zmiany danych w jej obrębie, pliki źródłowe frameworków zawierają bardzo dużą ilość kodu, a zatem zajmują więcej miejsca na dysku.

Z racji zwiększonego nacisku na wydajność, skalowalność i responsywność aplikacji na popularności zyskują metody dynamicznego tworzenia treści i doładowywania do niej danych z serwera w razie potrzeby, co znacząco zmniejsza konieczność odświeżania strony [3]. Przykładem jest model *Single Page Application*. Stosowanie go sprawia, że przy przełączaniu podstron ich zawartość ładuje się szybciej, przez co wydawać by się mogło jakby aplikacja była zbudowana z pojedynczej strony. Jednym z powodów jest przesyłanie z hosta mniejszej ilości odpowiednio opisanych danych, które są przygotowane do wyświetlenia i umieszczane w widoku w warstwie logiki aplikacji klienckiej [4]. Proces ten, w zależności od potrzeb, wymaga implementacji mniej lub bardziej złożonych funkcji.

Celem artykułu było porównanie wybranych funkcjonalności AngularJS w odniesieniu do własnych implementacji tych mechanizmów utworzonych w języku JavaScript.

Wykonane skrypty pomogły zweryfikować hipotezy postawione w artykule:

- własne implementacje mechanizmów dostępnych we frameworku mogą być stosowane w prostych przypadkach podczas budowy aplikacji,
- własne implementacje porównywanych funkcjonalności mogą być reużywalne – czyli zdatne do wielokrotnego użycia dla danych wejściowych dostarczonych w akceptowanej przez mechanizm formie.

1.2. Obszary badawcze

AngularJS oferuje wiele przydatnych narzędzi do tworzenia aplikacji typu *Single Page*, m.in.: router, serwis *$http* odpowiedzialny za łączność z zewnętrznymi hostami, zautomatyzowany *cache* przechowujący raz utworzone dane

w celu szybszego dostępu do nich, walidatory formularzy, serwis *$sce* zapewniający podstawową ochronę przed atakami *XSS* [6].

Z racji braku dostatecznego miejsca w niniejszym artykule, do porównania wybrano następujące funkcjonalności służące do dynamicznego budowania warstwy treści w aplikacji:

1) interpolacja wyrażeń,
2) powielanie elementów HTML.
3) monitorowanie zmian danych w warstwach treści i logiki.

Poszczególne funkcjonalności aplikacji będą najpierw zademonstrowane z użyciem frameworka AngularJS, a potem przedstawiona zostanie implementacja w języku JavaScript. Na koniec każdego badania zestawione będą ze sobą, w formie tabeli, subiektywne oceny sprawdzonych funkcjonalności na podstawie kryteriów:

- prostota użycia,
- możliwości zastosowania,
- reużywalność.

Punkty przyznawane będą w skali od 1 – 5, gdzie 1 oznacza najgorszą a 5 najlepszą ocenę.

Warto wspomnieć, że AngularJS posiada wbudowany mechanizm kompilacji kodu HTML [5]. Jego rolą jest rekursywne iterowanie po wskazanych węzłach DOM i sprawdzanie, czy dodane na nim atrybuty mają nazwę zapisaną w odpowiedniej konwencji. W pozytywnym przypadku wartość atrybutu zostaje połączona z odpowiadającym jej polem lub metodą w obiekcie *$scope*. Ten z kolei stanowi łącznik pomiędzy modelem – w nim zlokalizowane są skrypty z logiką aplikacji – a widokiem – prezentuje on przetworzone przez framework wartości do wyjściowej formy, którą widzi użytkownik. Atrybuty HTML zawierające w nazwie przedrostek *ng-*, po kompilacji, stanowią w AngularJS dyrektywy. Ich rolą jest informowanie frameworka o sposobie i ewentualnym przebiegu procesu dynamicznej modyfikacji określonego elementu HTML [6].

Na potrzeby analizy zaimplementowano w JavaScript uproszczony proces przeszukiwania drzewa DOM i odnajdywania elementów z atrybutami o specyficznej nazwie.

Interpolacja jest przydatna podczas umieszczania dopasowanych do konkretnego fragmentu tekstu wartości z modelu danych.

Składają się na nią z reguły dwa znaczniki – otwierający i zamykający. Obejmują one wyrażenia języka JavaScript oraz ewentualne fragmenty tekstu. Dzięki temu parser HTML potrafi rozpoznać miejsce, w którym powinien ewaluować występujące tam dane w określonym kontekście, którym w jest obiekt *$scope* [6].

Przykład 1. Kod HTML dla interpolacji wyrażeń w AngularJS.

```
<ul><li>[{{technology}}] Jest to {{text + " " +
getIncrementedNumber()}} przykładowego tekstu.</li>
<li>[{{technology}}] Jest to {{text + " " +
getIncrementedNumber()}} przykładowego tekstu.</li>
<li>[{{technology}}] Jest to {{text + " " +
getIncrementedNumber()}} przykładowego tekstu.</li>
</ul>
```

W elementach <li>, z przykładu nr 1, umieszczono tekst ze znacznikami interpolacji:„{{” oraz „}}”. Pomiędzy nimi znajdują się nazwy pól, które w modelu mają przypisane odpowiednie wartości.

Skrypt w modelu zawierał surowe użycie $interpolate. Podczas zwyczajnego korzystania z frameworka nie ma potrzeby odwoływania się do tego serwisu, ponieważ jest on wywoływany automatycznie.

Przykład 2. Kod skryptu dla interpolacji wyrażeń w AngularJS.

```
var context, interpolate;
var elementWithInterpolation = document.querySelector('ul');
angular.injector(['ng']).invoke(function($interpolate,
$rootScope) {
interpolate = $interpolate;
context = $rootScope;
context.technology = 'AngularJS';
context.text = 'linijka numer';
context.iterator = 0;
context.getIncrementedNumber = function() {
return ++context.iterator;
}; });
for (var i = 1; i <= 3; i++) {
angular.element(elementWithInterpolation).html(interpolate(e
lementWithInterpolation.innerHTML)(context));
}
```

W przykładzie 2 zadeklarowano zmienne pomocnicze **context** oraz **interpolate**. Służą one za aliasy, do których przypisywane są odpowiednio serwisy: *$interpolate* i *$rootScope*. Użycie interpolacji sprowadziło się do wywołania serwisu *$interpolate*, gdzie przekazano kod HTML z listy *<ul>*. Przetworzoną zawartość umieszczono z powrotem w liście.

Do implementacji algorytmu interpolacji w JavaScript (przykład nr 3) zastosowano wyrażenia regularne, dzięki którym można wyszukać i, w razie potrzeby, zamienić wystąpienia określonych znaków w tekście [7]. Kod HTML nie uległ zmianom.

Przykład 3. Kod skryptu dla interpolacji w JavaScript.

```
var context = {
text: 'linijka numer',
technology: 'JavaScript',
iterator: 0, getIncrementedNumber: function() {
return ++this.iterator;
} };
var elementWithInterpolation = document.querySelector('ul');
function customInterpolate(text, expression) {
var matchValueBetweenBrackets = /\{{(.*?)\}}/g;
var matchSeparatorsNotSurroundedByQuotes =
/\s*\+\s*"([^"]*)"\s*\+\s*/;
var matchValueAndFunctionInvocation = /\w+(\(\))?/g;
return text.replace(matchValueBetweenBrackets,
function(match, foundSubstring) {
var expressionsWithoutSeparators = foundSubstring.
replace(matchSeparatorsNotSurroundedByQuotes, '$1');
return evaluatedValues =
expressionsWithoutSeparators.replace(matchValueAndFunctio
nInvocation, function(match) {
return match.indexOf('(') !== -1 ? expression[match.slice(0, -
2)]() : expression[match];
}); } }
function appendToDOM(content, parent) {
parent.innerHTML = content;
}
for (var i = 1; i <= 3; i++) {
```

```
appendToDOM(customInterpolate(elementWithInterpolation.i
nnerHTML, context), elementWithInterpolation);
}
```

Własna implementacja interpolacji nie jest łatwa. Trudność polega na odpowiednim dobraniu wyrażeń regularnych, a od ich liczby i skomplikowania zapisu zależy wydajność działania mechanizmu. Mimo, że rozróżnia on pola kontekstu od metod, to obsługuje tylko separatory w postaci spacji.

Natywna implementacja nie jest też w stanie obsłużyć sytuacji, gdy w wyrażeniu będzie ulokowane działanie matematyczne. Serwis $interpolate waliduje typy danych wejściowych - wartości innego typu niż string są na niego rzutowane [8].

Interpolacja w AngularJS jest znacznie bardziej rozbudowana, gdyż wyrażenia w niej umieszczane są obserwowane przez pętlę $digest. Akceptowane są ścieżki do wyświetlania pól i wywoływania metod z zagnieżdżonych struktur danych. Framework posiada wbudowane zabezpieczenia przed atakami XSS w formie serwisów $sceProvider i $SanitizeProvider.

Tabela 1. Oceny kryteriów dla badania interpolacji.

Kryterium:	**AngularJS:**	**JavaScript:**
Prostota:	5	2.5
Możliwości:	4	2
Reużywalność:	4.5	3
Średnia:	**4.5**	**2.5**

2.2. Powielanie elementów HTML

Na stronach internetowych często spotkać można listy. Przeznaczone są do grupowania powiązanych ze sobą informacji, aby powiązać i przedstawić je w czytelny sposób. Język HTML oferuje specjalne znaczniki, które wyróżniają ten rodzaj treści. Są nimi: *<ul>*, *<ol>* i *<dl>*. Stosuje się je w zależności od formy jaką ma pełnić lista: nieuporządkowaną, uporządkowaną lub opisową [9].

Listy są dobrym przykładem elementów HTML, na których można pokazać proces powielania. AngularJS posiada przeznaczoną do tego dyrektywę *ng-repeat*. Framework udostępnia za jej pośrednictwem pomocne zmienne, m.in.: *$index*, *$first*, *$last*, które informują o aktualnym indeksie elementu oraz czy zajmuje on pierwsze, czy ostatnie miejsce w kolekcji źródłowej [10].

Przykład 4. Kod HTML dla powielania elementów w AngularJS.

```
<div ng-app="test" ng-controller="ctrl">
<ul>
<li ng-repeat="line in twoLines">[{{$index + 1}}]Pierwsza
lista - na górze.</li> </ul>
<ul>
<li ng-repeat="line in fourLines">[{{$index + 1}}]Druga
lista - na dole.</li>
</ul> </div>
```

Klonowanie elementów HTML w AngularJS realizowane jest za pośrednictwem dyrektywy *ng-repeat*. W kodzie z przykładu 4 zamieszczono w nich referencje do tablic **twoLines** i **fourLines** (utworzonych w przykładzie nr 5), po których następowała iteracja. Udostępniona z *ng-repeat* zmienna *$index* oznacza indeks aktualnie iterowanego elementu w tablicy.

Przykład 5. Kod skryptu dla powielania elementów w AngularJS.

```
angular.module('test', []).controller('ctrl', function($scope) {
$scope.twoLines = [1, 2];
$scope.fourLines = [1, 2, 3, 4];
});
```

W przypadku natywnej implementacji, kod HTML jest mniejszy, gdyż nie zawiera znacznika * * z dodatkowymi dyrektywami. W elementach *<ul>* znajdują się pojedyncze znaczniki *<li>* z atrybutami **data-lines**, w których wpisano liczbę linijek do wyświetlenia. Wewnątrz elementów wstawione zostały wyrażenia interpolacyjne oraz zwykły tekst.

W skrypcie z przykładu 6 posłużono się funkcjami **appendToDom** i **customInterpolate** użytymi w poprzednim przykładzie. Pierwszą z nich przebudowano, aby usuwała pierwszy, nieprzetworzony element listy, gdy całość będzie już przeformowana.

Przykład 6. Kod skryptu dla powielania elementów w JavaScript.

```
function appendToDOM(content, parent, clearParentContent)
{
if (clearParentContent)
parent.removeChild(parent.children[0]);
parent.appendChild(content);
}
var listRefs = document.querySelectorAll('li');
var context = {
lineNumber: 0,
getLineNumber: function() {
return ++this.lineNumber;
} };
for (var i = 0; i < listRefs.length; i++) {
var listRefsItem = listRefs[i];
for (var j = 0, len = listRefsItem.getAttribute('data-lines'); j <
len; j++) {
var clonedListRefsItem = listRefsItem.cloneNode(true);
clonedListRefsItem.innerHTML =
customInterpolate(listRefsItem.innerHTML, context)
appendToDOM(clonedListRefsItem, listRefsItem.parentNode, j
=== len - 1);
} }
```

Liczba docelowych kopii odczytywana jest z atrybutu **data-lines**. Zawartość elementu przekazywana jest do interpolacji. Po przetworzeniu tekst wstawiany jest do sklonowanego elementu *<li>*, po czym dodawany do rodzica, czyli listy *<ul>*.

Różnica względem AngularJS polega na innej numeracji w drugiej liście. Dyrektywa *ng-repeat* tworzy nowy *$scope*, więc każda z list ma liczby skupione w zasięgu lokalnym [10]. Natywna implementacja funkcjonalności w JavaScript nadaje liczby inkrementując je za każdym razem dla następnej listy. Aby temu zaradzić należało by np. rozszerzyć funkcję **customInterpolate** o resetowanie licznika dla każdej listy podczas ewaluacji interpolowanych wyrażeń.

Tabela 2. Oceny kryteriów dla badania powielania elementów HTML.

Kryterium:	**AngularJS:**	**JavaScript:**
Prostota:	4	3
Możliwości:	4.5	2.5
Reużywalność:	4.5	2
Średnia:	**4.33**	**2.5**

2.3. Monitorowanie zmiany danych w warstwach treści i logiki

Nieodłączną częścią aplikacji, w której użytkownik może wprowadzać dane i wchodzić w interakcje z jej widokiem, jest mechanizm monitorujący zmiany w obrębie danych. Framework posiada mechanizm, który umożliwia sprawdzanie, a w razie konieczności aktualizację, danych pod kątem spójności pomiędzy modelem a widokiem. Obserwacji poddać można wartości typu prymitywnego i obiekty, zarówno o płaskiej strukturze, jak i z licznymi zagnieżdżeniami. Oczywiście, wraz ze wzrostem skomplikowania rozkładu danych i ich liczby, czas tego procesu wydłuża się. Pętla *Digest Loop* jest najbardziej istotnym mechanizmem w AngularJS. Odpowiada za synchronizację danych w modelu i widoku, zarówno automatycznie gdy wartości podłączone są do obiektu *$scope*, jak i ręcznie przez dostępne metody: *$watch*, *$watchCollection* i *$watchGroup* [11][12].

Przykład 7. Kod HTML dla monitorowania zmian w danych w AngularJS.

```
<div ng-app="test" ng-controller="ctrl">
<label for="field">Określ długość listy:
<input id="field" ng-model="amount" type="number">
<ul>
<li ng-repeat="item in list; track by $index">{{$index + 1}}
element listy.
<button type="button" ng-
click="removeItem($index)">X</button>
</li>
</ul>
</div>
```

W przykładzie 7 użyto dyrektywy *ng-model*, która ustawia wartość z pola *$scope.amount* na taką, jaka znajduje się w atrybucie **value** elementu <*input*> [13]. Listę *$scope.list* można dynamicznie wydłużać i skracać. Jej długość kontrolowana jest przez liczbę wpisaną do wspomnianego pola. Dodatkowo kliknięcie w przycisk <*button*> usuwa jeden element z listy.

Przykład 8. Kod skryptu dla monitorowania zmian w danych w AngularJS.

```
angular.module('test', []).controller('ctrl', function($scope) {
$scope.amount = '';
$scope.list = [];
$scope.removeItem = function( index ) {
$scope.list.splice(index, 1);
$scope.amount--;
}
$scope.$watch('amount', function() {
$scope.list = new
Array(+$scope.amount).fill($scope.amount);
});
});
```

W modelu (przykład 8) zastosowano metodę *$scope.$watch*, gdzie wykryta zmiana w obserwowanym polu **amount** umożliwia zmianę aktualnie przypisanej tablicy do *$scope.list* na nową, o długości wpisanej przez użytkownika w znaczniku <*input*>. Metoda *$scope.removeItem* usuwa wybrany za pośrednictwem indeksu element z tablicy i dekrementuje wartość pola *$scope.amount*, które połączone jest z <*input*>..

HTML dla natywnej implementacji (przykład 9) ma znacznie mniej kodu niż w AngularJS. Element <*ul*> jest początkowo pusty, ponieważ później jest on uzupełniany dynamicznie.

Przykład 9. Kod HTML dla monitorowania zmian w JavaScript.

```
<label for="field">Określ długość listy:
<input id="field" type="number">
<ul></ul>
```

Skrypt z przykładu nr 10 korzysta z wcześniej przygotowanej funkcji **customInterpolate**.

Przykład 10. Kod skryptu dla monitorowania zmian w JavaScript

```
function removeFromDom() {
listRef.removeChild(listRef.children[listRef.children.length -
1]);
context.index--;
}
var context = {
index: 0,
getIndex: function() {
return ++this.index;
}
};
var list = [];
var element = document.createElement('li');
element.innerHTML = '{{getIndex()}} element listy.<button
type="button">X</button>';
var listRef = document.querySelector('ul');
var input = document.querySelector('input');
input.addEventListener('input', function(evt) {
var val = evt.target.value;
if (val < context.index) {
removeFromDom();
} else {
context.index = 0;
while(listRef.children.length) {
listRef.removeChild(listRef.children[listRef.children.length - 1]
);
}
for (var i = 0; i < val; i++) {
var clonedElement = element.cloneNode(true);
clonedElement.innerHTML =
customInterpolate(element.innerHTML, context);
listRef.appendChild(clonedElement);
}
}
});
document.querySelector('ul').addEventListener('click',
function(evt) {
if (evt.target.tagName === 'BUTTON') {
removeFromDom();
input.value = context.index;
}
})
```

Użyte zostały dwie metody *addEventListener* na elementach DOM. Służą one do reagowania na określone w parametrze zdarzenia, gdzie przy każdym wystąpieniu wywołują podaną funkcję zwrotną – tam przekazywany jest obiekt zdarzenia [14]. W aplikacji, metody nasłuchujące reagują na zdarzenia **input** oraz **click**. Ich obsługa w funkcjach zwrotnych dzieli się na trzy części. Podczas zmiany liczby w polu <*input*>, albo usuwany jest ostatni <*li*>, albo jeden jest dodawany na koniec listy. Kliknięcie w przycisk usuwa konkretny element. Wszystkie przypadki aktualizują licznik długości listy **context.index**.

Problemem podczas nasłuchiwania zdarzeń może być np. sytuacja, gdy obiekt DOM, na którym są podłączone zostanie usunięty – ponowne jego utworzenie nie spowoduje

wznowienia nasłuchiwania jeśli nowy element nie będzie tym samym co poprzedni. Można temu zaradzić przechowując usunięty element w cache. Dzięki zjawisku propagacji zdarzeń możliwe jest natomiast ograniczenie liczby metod nasłuchujących daną grupę elementów, gdy podłączy się jedną z nich do wspólnego kontenera [14].

Metody obecne w AngularJS obserwują wartości obiektu *$scope*. nie tylko w płaskiej strukturze danych, lecz także na głębokim poziomie zagnieżdżenia [12].

Tabela 3. Oceny kryteriów dla badania monitorowania zmiany danych.

Kryterium:	**AngularJS:**	**JavaScript:**
Prostota:	4.5	4
Możliwości:	4	3
Reużywalność:	4	3.5
Średnia:	**4.16**	**3.5**

Przedstawione w analizie skrypty udowadniają, że przydatne funkcjonalności, oferowane przez framework AngularJS, można, w uproszczonych wersjach, wprowadzić do aplikacji samemu. Obie hipotezy zostały potwierdzone, z zaznaczeniem, że aspekt reużywalności natywnie zaimplementowanych mechanizmów może być ograniczony.

Średnia punktów z trzech testów (tabele nr 1, 2 i 3) dla frameworka wynosi: 4.33pkt., zaś dla języka JavaScript: 2.83pkt. Nie ulega wątpliwości, że praca z AngularJS jest łatwiejsza, ponieważ istotne procedury wspomagające budowę aplikacji są przez framework wykonywane bez ingerencji programisty. Wyróżnić tu można: skomplikowany proces kompilacji drzewa DOM, synchronizacja modelu z widokiem oraz obsługa DOM – czyli restrukturyzacja i modyfikacja elementów widoku. Natomiast w przypadkach, gdy zachodzi konieczność realizacji niestandardowej funkcjonalności, framework zazwyczaj udostępnia narzędzia przydatne do tego celu.

Implementacja w czystym JavaScript własnych mechanizmów do obsługi tego, co AngularJS automatyzuje jest dosyć pracochłonna. W istocie wykonanie uproszczonych funkcji bazując na natywnym języku ogranicza możliwości przez nie oferowane, jak choćby walidacja wartości wejściowych lub niezależny zakres dostępności dla danych. Nie wykluczone, że może pojawić się potrzeba rozszerzenia konkretnego mechanizmu. Nie mniej, dodanie takiego rodzaju API jest możliwe i warto być tego świadomym.

Rezygnacja lub ograniczenia używania frameworka w dobrze dobranych przypadkach zmniejszy ilość danych, które przeglądarka musi pobrać oraz przyspieszy działanie aplikacji. Bowiem, mimo że metody z API frameworka są lepiej dopracowane pod względem obsługi różnych przypadków użycia i oddają do dyspozycji programisty dużo bardziej rozbudowane mechanizmy niż ma to miejsce przy posługiwaniu się natywnym JavaScript, to są one w wydaniu AngularJS mniej wydajne [15]. Przykładowo, jeśli w modelu danych zostanie ustawionych przesadnie dużo wartości podłączonych do obiektu *$scope*, a także gdy znacząca liczba parametrów do dyrektyw i komponentów będzie przekazywana przez wiązania dwukierunkowe, spowoduje to obciążenie pętli *$digest*, która będzie sprawdzać dużą liczbę danych za każdym razem – nawet gdy zmiana nie będzie się bezpośrednio do nich odnosić [16]. Wpłynie to na opóźnienia w synchronizacji modelu z widokiem i nadmierne obciążenie podzespołów sprzętowych.

Nie należy zatem traktować funkcjonalności frameworka jako przedmiotu do nadużywania z powodu wygody, czy automatyzacji wszystkiego, co jest potencjalnie możliwe. W każdym przypadku – czy to skromnej aplikacji, czy rozbudowanej – warto jest przemyśleć co konkretnie będzie przydatne podczas procesu programowania i w miarę możliwości starać się ograniczać korzystanie z tych funkcjonalności, które pogarszają efektywność działania, albo zastąpić gotowe narzędzia mniejszymi odpowiednikami lub napisać własne skrypty pomocnicze [17]. Albowiem, można zaimplementować mechanizmy w podstawowym stopniu odpowiadające tym dostępnym we frameworku i czerpać z nich korzyści przy zmniejszonym wpływie na szybkość działania aplikacji.

Literatura

[1] K. Nyg˚ard, Single page architecture as basis for web applications, Aalto University: School of Science, Espoo, 2015

[2] https://www.thebalance.com/what-is-a-front-end-framework-and-why-use-one-2071948 [27.11.2017]

[3] W. Chansuwath, T. Senivongse, A Model-Driven Development of Web Applications Using AngularJS Framework, Department of Computer Engineering at Chulalongkorn University, Bangkok, 2016.

[4] M. S. Mikowski, J. C. Powell, Single Page Web Application, Manning Publications Co., Shelter Island in New York, 2014

[5] https://docs.angularjs.org/guide/compiler [27.11.2017]

[6] A. Lerner, ng-book – The complete Book on AngularJS, Fullstack.io, 2013.

[7] J. Friedl, Mastering Regular Expressions, 3rd Edition, O'Reilly Media, 2006.

[8] https://docs.angularjs.org/guide/interpolation#how-the-string-representation-is-computed [27.11.2017]

[9] https://www.w3.org/wiki/HTML_lists [27.11.2017]

[10] https://docs.angularjs.org/api/ng/directive/ngRepeat [27.11.2017]

[11] https://docs.angularjs.org/guide/scope#scope-life-cycle [27.11.2017]

[12] https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$rootScope.Scope-methods [27.11.2017]

[13] https://docs.angularjs.org/api/ng/directive/ngModel [27.11.2017]

[14] S. Alimadadi, S. Sequeira, A. Mesbah, K. Pattabiraman, Understanding JavaScript Event-Based Interactions, Electrical and Computer Engineering at University of British Columbia, Vancouver, 2014.

[15] M. Ramos, M. Tulio Valente, R. Terra, AngularJS Performance: A Survey Study, Computing Research Repository - ArXiv, 2017

[16] M. Ramos, M. Tulio Valente, R. Terra, G. Santos, AngularJS in the Wild: A Survey with 460 Developers, Association for Computing Machinery, 2016

[17] http://blog.scalyr.com/2013/10/angularjs-1200ms-to-35ms/ [27.11.2017]