POLITECHNIKA GDAŃSKA

WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Mikołaj Jaskulski
Nr albumu: 131521
Studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność/profil: Aplikacje rozproszone i systemy internetowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Graficzny kreator stron www służących do zdalnego monitoringu urządzeń

Tytuł pracy w języku angielskim: Graphic web pages creator for remote devices monitoring

Potwierdzenie przyjęcia pracy	
Opiekun pracy podpis	Kierownik Katedry/Zakładu podpis
dr inż. Łukasz Kuszner	

Data oddania pracy do dziekanatu:

POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

OŚWIADCZENIE

Imię i nazwisko: Mikołaj Jaskulski
Data i miejsce urodzenia: 21.12.1991, Gdynia
Nr albumu: 131521
Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki
Kierunek: informatyka
Poziom studiów: II stopnia
Forma studiów: stacjonarne

Ja, niżej podpisany(a), wyrażam zgodę/nie wyrażam zgody* na korzystanie z mojej pracy dyplomowej zatytułowanej: Graficzny kreator stron www służących do zdalnego monitoringu urządzeń
do celów naukowych lub dydaktycznych.[1]

Gdańsk, dnia ....
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r., nr 90, poz. 631) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),[2] a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ....
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia ....
podpis studenta

*) niepotrzebne skreślić

---

[1] Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

[2] Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym:
Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.
Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.

STRESZCZENIE

The **Streszczenie** should correspond to the **Abstract** and includes the same key elements. It should be prepared according to faculty rules. The **Streszczenie** is a chapter written only by Polish authors who decide to write the diploma thesis in English and it is not applicable to foreign students.

Słowa kluczowe: [keywords] Zażółć gęślą jaźń.

Dziedzina nauki i techniki, zgodnie z wymogami OECD: <dziedzina>, <technika>,. . . [field of science and technology in accordance with OECD requirements: <field>, <technology>,. . .]

SPIS TREŚCI

LIST OF IMPORTANT SYMBOLS AND ABBREVIATIONS

i, j, l, m	—	indeksy
MVC	—	wzorzec projektowy Model View Controller

1. WYBÓR TECHNOLOGII

Rozdział ten zawiera opis wad oraz zalet technologii, których użycie było brane pod wagę do realizacji projektu. Ilość dostępnych technologii oraz narzędzi sprawia, iż wybranie najbardziej odpowiedniej z nich do określonego celu nie jest prostym zadaniem.

1.1. Wybór technologii backendu

Zadanie nieco ułatwił fakt, iż podczas realizacji projektu było dostępne rozwiązanie do komunikacji z serwerem urządzeń. Była nim biblioteka wykonana w technologii C#. Aby bez trudu skorzystać z jej wszystkich możliwości do stworzenia backendowej części aplikacji posłużono się zatem technologią C# oraz frameworkiem pozwalającym na szybkie tworzenie aplikacji internetowych MVC4. Framework ten został wybrany z następujących powodów[1]:

- Jest najnowszym Frameworkiem rozwijanym przez Microsoft
- Pozwala na pełną kontrolę nad dynamicznie generowaną treścią strony
- Jest zgodny z metodyką Test Driven Development
- Łatwo integruje się z JavaScript
- Pozwala na szybkie tworzenie usług typu RESTful

1.2. Wybór technologii Frontendu

W przeciwieństwie do technologii aplikacji po stronie serwera, wybór odpowiedniego narzędzia do stworzenia dynamicznego interfejsu aplikacji internetowej jest trudniejszym zadaniem niż wybór technologii backendowej. Zgodnie z nowoczesnymi trendami oczywiste jest by posłużyć się technologią wspieraną przez każdą współczesną przeglądarkę - JavaScript[2], lecz na tym zadanie wyboru się nie kończy. Wynika to z faktu, iż JavaScript jest technologią w której trudno zachować logiczną strukturę projektu. Dowodem na to jest powstanie dużej ilości frameworków. Istnieje nawet strona internetowa, która pozwala na dobranie odpowiedniego narzędzia do potrzeb projektu z pośród 78[3]. Każdy z nich rozwiązuje jednak problem tworzenia bogatego interfejsu aplikacji internetowej na inny sposób. Często wybieraną formą organizacji projektu narzucaną przez frameworki jest implementacja wzorca projektowego Model View Controller. Jako że nie jest to łatwe zadanie, każdy framework realizuje wzorzec na swój sposób, „rozmywając" zakres obowiązków poszeczgólnych składowych MVC. Z tego powodu powszechne jest stwierdzenie, iż dany framework należy do rodziny MV*, lub MVW - „Model, View, Whatever".

Aby wytypować rozwiązanie które będzie najbardziej pomocne w realizacji projektu należy dokonać analizy posługiwania się nim. Jest to najbardziej skuteczny sposób aby wytypować framework, który najłatwiej zaadaptuje się do określonej sytuacji. W poniższych podrozdziałach przedstawione zostały najbardziej popularne na tę porę rozwiązania służące do tworzenia bogatego interfejsu aplikacji internetowej. Każdemu z nich przypisano również kilka cech, które są równie istotne podczas wybierania narzędzia organizującego projekt. Są nimi:

- dojrzałość rozwiązania
- rozmiar społeczności związanej z rozwiązaniem
- w jakich dużych projektach rozwiązanie zostało wdrożone
- rozmiar rozwiązania

1.3. AngularJS

AngularJS JS jest frameworkiem stworzonym przez firmę Google w 2010 roku. Narzuca on użycie wzorca projektowego MVVM - Model, View, ViewModel, które są głównymi elementami tworzonymi przez programistę podczas tworzenia aplikacji za pomocą frameworku.

1.3.1. Widoki i dyrektywy

AngularJS wyróżnia się z pośród innych frameworków tym, iż umożliwia korzystanie dodatkowych elementów rozszerzających HTML zwanych dyrektywami. Można o nich myśleć jak o dodatkowych atrybutach węzłów HTML, które zaczynają się od znaków. Dyrektywy mają różne zastosowanie. Można wyróżnić między innymi dyrektywy służące do:

- powiązań (binding) między widokiem a modelem i kontrolerem - np. ng-model, ng-binding, ng-controler
- tworzenia prostej logiki generowania dynamicznych elementów strony - np. ng-repeat
- reagowania na zdarzenia - np. ng-click

Oprócz dyrektyw istnieje jeszcze jedno dodatkowe wyrażenie służące do powiązania zmiennej kontrolera z odpowiadającą jej zmienną widoku. Służy do tego podwójny nawias klamrowy, tzw. znacznik. Gdy odpowiadająca mu zmienna kontrolera zmieni swoją wartość, węzeł HTML zawierający znacznik zostanie odświeżony.

Poniżej znajduje się przykład wykorzystania dyrektyw wraz ze znacznikiem:

```
<body ng-app='calculatorApp'>
  <div ng-controller='calculatorController as calc'>
   <div>
       Liczba 1: <input type="number" ng-model="number1">
     </div>
     <div>
       Liczba 2: <input type="number" ng-model="number2">
     </div>
     <div>
       Wynik: {{number1 + number2}}
     </div>
     <button ng-click='calc.showResult()'>Pokaz wynik</button>
  </div>
</body>
```

Pliki zawierające te noszą nazwę szablonu (template) oraz muszą być skompilowane przez kompilator AngularJS przed umieszczeniem ich na serwerze.

1.3.2. Kontroler

Kontrolery są obiektami JavaScript które, zawierają logikę interfejsu. Ich zadanie nie różni się od klasycznego kontrolera z wzorca MVC. Są tworzone po to, by przygotować obiekty Modelu i następnie przekazać je do Widoku. Kontrolery tworzy się za pomocą metody module obiektu angular. Przykładowy kontroler reagujący na zmianę pól tekstowych input oraz wyświetlający winik dodawania wprowadzonych liczb w ostatnim węźle div znajduje się poniżej:

```
angular.module('calculator', [])
.controller('CalculatorController', function() {
  this.number1 = 1;
  this.number2 = 2;

  this.showResult = function pay() {
   result = this.number1 + this.number2;
    window.alert(result);
```

```
  };
});
```

1.3.3. Usługa

Usługi są obiektami, które powinny zawierać logikę biznesową - wszelkie mechanizmy nie powiązane z widokiem. Do tworzenia usługi służy metoda factory obiektu angular. Należy nadać jej nazwę analogicznie jak w przypadku kontrolera. Przykładowe stworzenie usługi może wyglądać następująco:

```
angular.module('calculatorServiceModula', [])
.factory('calculatorService', function() {
   this.showResult = /* cialo metody */
});
```

Aby powiązać kontroler kalkulatora z usługą, należy przekazać jej nazwę do metody controller obiektu angular w taki sposób:

```
angular.module('calculator', ['calculatorServiceModule'])
.controller('CalculatorController', ['calculatorService', function(calculatorService) {
   this.showResult = function(){
      calculatorService.showResult();
   }
}]);
```

1.3.4. Cechy rozwiązania

Tablica 1.1. Cechy jakości frameworku AngularJS

Nazwa cechy	Ocena
dojrzałość rozwiązania	rozwijany od 6 lat
rozmiar społeczności	151954 tematów na StackOverflow
zastosowanie w projektach	bardzo duże, m. in. YouTube, VEVO, Netflix
rozmiar rozwiązania	50KB

1.4. BackboneJS

BackboneJS jest frameworkiem który organizuje strukturę projektu JavaScript wykorzystując pojęcia Widoków oraz Modeli z wzorca MVC [4]. Backbone umożliwa również deklarowanie oraz obsługę zdarzeń. Kolejną zaletą jest łatwa integracja rozwiązania z usługami sieciowymi typu RESTful. Korzystanie z frameworku w większej mierze opiera się na tworzeniu obiektów za pomocą klasy Backbone.

1.4.1. Modele

Modele tworzy się używając metody za pomocą klasy „Model”, która w konstruktorze przyujmuje strukturę typu klucz-wartość. Utworzony obiekt będzie posiadał pola o nazwach odpowiadającym kluczom, i przypisane do nich wartości. Wartością może być też funkcja. Model posiada również metody pozwalające na modyfikację oraz odczyt każdego z pól. Obiekty modelu można

rozszerzać o obsługę zdarzeń. Dla przykładu, możliwe jest zdefiniowanie metody, która zostanie wywołana gdy wartość któregoś z pól obiektu modelu zmieni się. Poniżej przedstawiony został przykład tworzenia modelu wraz z przypisaniem do niego powyższego zdarzenia oraz utworzenia kolekcji modeli z jednym elementem.

```
var Osoba = Backbone.Model.extend({
  wiek,
  waga,
  id,
  url: '/osoba/',
  wypiszBmi: function() {
   alert(waga / (wzrost * wzrost));
  },

});
var osoba = new Osoba({wiek: 20, waga: 80, id: 1});
osoba.on('change', function(model){
   alert('Ktores z pol zostalo zmienione!');
});

var osoby = Backbone.Collection.extend({
   model: Osoba,
   url: '/osoba/
});
osoby.add(osoba);
```

Backbone pozwala na dodanie do każdego modelu ścieżki URI oraz identyfikatora obiektu. Pozwala to na prostą interakcję z usługami sieciowymi wykonanymi w architekturze REST poprzez odpowiednie metody. Identyfikator dodaje się za pomocą pola o nazwie id, jak widać w powyższym przykładzie. Trzeba równiez określić URI usługi która będzie związana z obiektem. Następnie wywołując odpowiednie metody modelu framework wyśle odpowiednie rządania HTTP. Poniżej znajduje się lista metod wraz z odpowiadającymi im adresami rządań HTTP, które zostałyby wywołane dla powyższego przykładu:

Tablica 1.2. Wywoływanie zapytań HTTP za pomocą metod modelu w BackboneJS

Nazwa metody	metoda i URI rządania
model.fetch()	GET /osoba/1
model.save()	PUT /osoba/1
model.destroy()	DEL /osoba/1
kolekcja.fetch()	GET /osoba/
kolekcja.create()	POST /osoba/

1.4.2. Widoki

W odróżnieniu do innych frameworków, w Backbone Widok pełni również funkcje kontrolera. Łączy on logikę interfejsu wraz oraz definiuje wygląd elementu węzłu DOM z którym jest związany. Widoki tak jak i Modele również są obiektami tworzonymi za pomocą klasy Backbone. Każdy z nich posiada pole o nazwie „$el", które jest obiektem odpowiadającym elementowi DOM związanym z widokiem. Pole to musi zostać dodane manualnie do określonego węzła na stronie. Dobrą praktyką jest, aby dany widok operował tylko na „swoim" obiekcie $el. Operacje te

powinny mieć miejsce w metodzie „render” posiadanej przez każdy widok. Pozwala to na zachowanie przejrzystej struktury projektu. Przykład utworzenia widoku i dodania go do konkretnego elementu strony znajduje się poniżej:

```
var Zegar = Backbone.View.extend({
    render: function () {
     this.$el.empty().append(new Date());
    }
});

var zegar = new Zegar()
zegar.$el.appendTo('body')
```

Podczas tworzenia widoku można powiązać je z modealmi. Częstą praktyką jest dodawanie zdarzeń, które wywołują metodę render widoku w momencie gdy model się zmieni.

1.4.3. Cechy rozwiązania

Tablica 1.3. Cechy jakości frameworku BackboneJS

Nazwa cechy	Ocena
dojrzałość rozwiązania	rozwijany od 5 lat
rozmiar społeczności	19391 tematów na StackOverflow
zastosowanie w projektach	bardzo duże, m. in. Reddit, Live Wallpaper For Android, cloud9trader.com
rozmiar rozwiązania	36KB

1.5. KnockoutJS

Nazwa kolejnego frameworku do tworzenia bogatego interfejsu użytkownika w technologii JavaScript jest KnockoutJS. Narzędzie to organizuje tworzoną w oparciu i nie aplikację wykorzystując wzorzec projektowy o nazwie Model View ViewModel (MVVM)[**?**]. Knockout tak jak wcześniej opisane narzędzia pozwala na definiowanie powiązań między obiektami warstwy modelu i widokami, pozwalając skupić się programiście na realizacji aplikacji.

1.5.1. ViewModel

ViewModel w Knockout jest dokładną implementacją swojego odpowiednika z wzorca MVVM i w taki sposób należy go traktować. Obiekty ViewModel związane są bezpośrednio z określonym widokiem i zawierają pola oraz metody, które potrzebne są do prezentacji treści przez ten widok. ViewModel może na przykład zawierać obiekty eksponowanych przez widok modeli oraz dodatkowe pola uzupełniające. Dzięki takiej architekturze pozbywamy się kompletnie nawet najprostszej obróbki danych z widoku, która zostaje przeniesiona właśnie do ViewModel.

Aby utworzyć ViewModel w Knockout należy posłużyć się obiektem „ko” dostarczanym przez bibliotekę. Po zadeklarowaniu klasy, która będzie pełniła funkcję VM należy posłużyć się metodą observable obiektu ko w celu utworzenia pól. Utworzone w ten sposób pola posiadać będą następujące cechy:

- utworzone zostaną dla nich metody funkcje gettera i settera
- możliwe będzie powiązania pola z wybranym węzłem DOM widoku
- framework automatycznie zaktualizuje element węzła DOM powiązany z polem podczas zmiany jego wartości

W przypadku gdy wartość jedno z pól zależy od pozostałych lub istnieje potrzeba powiązania elementu DOM z wynikiem działania na wielu polach należy posłużyć się metodą „subscribe". Spowoduje to odświeżenie pola w przypadku gdy któreś z jego zależności zmieni swoją wartość. W momencie gdy pole zależne jest od dużej ilości pól, można posłużyć się metodą „computed", która przypisuje pole do metody i wywołuje ją za każdym razem gdy któreś z pól się zmieni.

Po stworzeniu obiektu ViewModel należy przypisać go do węzła DOM w widoku, z którym będzie powiązany. Realizuje się to wywołując metodę „applyBindings" obiektu ko. Szczegóły opisane zostały w kolejnym podrozdziale o Widokach.

Przykładowy kod zawierających opis klasy ViewModelu znajduje się poniżej. Jest w nim tworzony obiekt ViewModel który posiada 3 pola. Pole „bmi" jest zależne od dwóch pozostałych.

```
var BmiViewModel = function() {
   this.waga = ko.observable(0);
   this.wzrost = ko.observable(0);
   this.bmi = ko.observable(0);

    this.waga.subscribe(updateBMI);
    this.wzrost.subscribe(updateBMI);

   function updateBMI() {
      this.bmi(this.waga() / (this.wzrost() * this.wzrost()));
   }
}

var bmiViewModel = new BmiViewModel();
ko.applyBindings(bmiViewModel);
```

1.5.2. Widoki

Widokami w KnockoutJS są po prostu strony z kodem HTML. Powiązania między obiektami ViewModel a widokiem definiuje się w widokach za pomocą atrybutu „data-bind". Powinien zawierać on pary klucz-wartość oddzielone od siebie przecinkami. Wartością najczęściej jest obiekt który zostanie powiązany z elementem DOM, a kluczem metoda powiązania. Poniższy przykład zawiera prosty sposób powiązania listy rozwijanej z obiektem ViewModelu.

```
<div id="budowaCialaForm">
   <select data-bind="options: rodzajeBudowy,
                optionsCaption: 'Wybierz...',
                optionsText: 'nazwa',
                value: wybranyRodzaj
                ">
   </select>
</div>
```

```
var BudowaCialaViewModel = function() {
```

```
    this.rodzajeBudowy = [
      { nazwa : 'Ektomorfik', wspolczynnikSpalania: 1.5 },
      { nazwa : 'Endomorfik', wspolczynnikSpalania: 1.25 },
      { nazwa : 'Mezomorfik', wspolczynnikSpalania: 1.0 }
    ];
     this.wybranyRodzaj = ko.observable();
}
ko.applyBindings(new BudowaCialaViewModel(), document.getElementById('budowaCialaForm'));
```

Poniżej znajduje się lista najpopularniejszych opcji powiązań obniektów ViewModel z elementami DOM widoków oraz krótki opis ich działania:

- visible - wiąże zmienną określającą, czy element będzie widoczny na stronie
- text - wartośc tekstowa określonej zmiennej zostanie wyświetlona w powiązanym elemencie
- html - wartość zmiennej zostanie wstrzyknięta jako HTML w powiązanym elemencie
- attr - przyjmuje parę klucz-wartość, wartość zostanie dodana do atrybutu powiązanego elementu o nazwie określonej w kluczu
- if, foreach - zawartość węzła zostanie wygenerowana dynamicznie w oparciu o powiązany obiekt lub listę obiektów
- event - pozwala powiązać zdarzenie elementu z funkcją określonego obiektu; przykładowymi zdarzeniami są: onclick, mouseover, keypress itd.

Widoki można organizować w struktury za pomocą szablonów. Za ich pomocą możliwe jest umieszczenie treści jednego widoku w drugim. Oprócz szablonu strony, można w ten sposób budować małe komponenty gotowe do powtarzalnego użycia na różnych stronach.

1.5.3. Cechy rozwiązania

Tablica 1.4. Cechy jakości frameworku BackboneJS

Nazwa cechy	Ocena
dojrzałość rozwiązania	rozwijany od 6 lat
rozmiar społeczności	15,598 tematów na StackOverflow
zastosowanie w projektach	bardzo duże, m. in. Azure, eventim, jsFiddle
rozmiar rozwiązania	35KB

1.6. CoffeeScript i TypeScript

W tym rozdziale zostaną omówione rozwiązania inne niż poprzednie. Kod napisany w TypeScript lub CoffeeScript, kompilowany jest w całości do języka JavaScript[5**?**]. Języki te ograniczają ilość tworzonego kodu oraz udogadniają korzystanie z podstawowych mechanizmów takich jak dziedziczenie. Aby z nich skorzystać potrzebny jest jedynie kompilator języka z którego chcemy skorzystać.

TypeScript oraz CoffeeScript różnią się składnią. To, która z nich jest bardziej użyteczna jest kwestią gustu. Dodatkowym atutem posiadanym jedynie przez język TypeScript jest wprowadzenie interfejsów oraz kontrola typów na poziomie kompilacji. Kompilator podczas swojej pracy niejako wymusza na programiście zachowanie zdefiniowanych wcześniej typów danych. Jest to ograniczenie podstawowej funkcji języka skryptowego którym jest JavaScript jak i zarazem dodatkowy środek zapobiegający błędom wynikającym z nieuwagi programisty. Właśnie ta funkcja powoduje, iż TypeScript zyskuje popularność wśród programistów. Dowodem na to jest promocja integracji popularnego frameworka Angular 2 z TypeScript przez firmę Google[6].

Poniżej znajdują się przykłady kodu CoffeeScript jak i TypeScript, które realizują te same funkcje oraz rezultat ich kompilacji w postaci kodu JavaScript.

Kod TypeScript

```
class Zwierze {
    constructor(private nazwa: string) {}

    public poruszSie(metry: number): void {
        alert(this.nazwa + " przebyto: " +
            meters);
    }
}

class Pies extends Zwierze {
    public biegnij() : void {
        alert("Bieg");
        super.poruszSie(5);
    }
}

var pies = new Pies("Burek");
pies.biegnij();
```

Kod CoffeeScript

```
class Zwierze
  constructor: (@nazwa) ->

  poruszSie: (metry) ->
    alert @nazwa + " przebyto:
        #{metry}m."

class Pies extends Zwierze
  biegnij : ->
    alert "Bieg"
    super.poruszSie 5

pies = new Pies "Burek"
pies.biegnij()
```

Kod JavaScript

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
var Zwierze = (function () {
    function Zwierze(name) {
        this.name = name;
    }
    Zwierze.prototype.poruszSie = function (meters) {
        alert(this.name + " przebyto: " + meters);
    };
    return Zwierze;
})();
var Pies = (function (_super) {
    __extends(Pies, _super);
    function Pies() {
        _super.apply(this, arguments);
    }
    Pies.prototype.biegnij = function () {
        alert("Bieg");
        _super.prototype.poruszSie.call(this, 5);
    };
    return Pies;
```

```
}(Zwierze));
var pies = new Pies("Burek");
pies.biegnij();
```

Jak widać na powyższym przykładzie, omawiane języki pozwalają zaoszczędzić pisania dużej ilości powtarzalnego kodu.

1.7. Podsumowanie

Wszystkie zaprezentowane technologie służą do ograniczenia powstawania zbędnego kodu. Wybierając technologię należy przedewszystkim wziąć pod uwagę charakter tworzonego projektu. W przypadku ninejszego projektu tworzenie zaawansowanego interfejsu umieszczonego na jednej stronie nie jest potrzebne. Przydatne jest jednak powiązanie elementów DOM z obiektami JavaScript. Należy również zwrócić uwagę na to iż przedmiot ninejszej pracy magisterskiej nie jest standardową aplikacją internetową. Zaletą będzie więc duża elastyczność rozwiązania. Jako framework do realizacji projektu został wybrany KnockoutJS. Poniżej znajdują się cechy które zadecydowały o jego wyborze:

- Bardzo elastyczny mechanizm powiązania obiektów JavaScript z węzłami DOM
- Możliwośc reagowania na zdarzenia
- Nie narzuca struktury projektu
- Brak nadmiarowych funkcji, które nie zostaną wykorzystane w projekcie

Drugą technologią do stworzenia interfejsu użytkownika został wybrany TypeScript. Zastosowanie go zmniejszy ilość powtarzalnego kodu w aplikacji oraz pomoże w jej modularyzacji. Dodatkowymi atutami TypeScript są:

- JavaScript jest w pełni zgodny ze składnią TypeScript, można więc z nim integrować każdą bibliotekę JavaScript
- Kontrola typów oraz interfejsy zmniejszają ryzyko błędów
- TypeScript posiada podobną składnię do języka C#, w której zostanie napisana część aplikacji serwera

2. ASP.NET MVC4

MVC w chwili obecnej jest najnowszym frameworkiem rozwijanym przez Microsoft służącym do szybkiego tworzenia aplikacji internetowych [1]. Nazwa MVC jest akronimem wzorca projektowego „Model View Controller", który jest wykorzystywany do tworzenia aplikacji posiadających interfejs graficzny. Framework niejako wymusza zastosowanie go w wytwarzanej aplikacji. Wiedza na temat powyższego wzorca znacznie zwiększa jakość architektury wytwarzanej aplikacji poprzez lepszą organizację kodu.

2.1. Wzorzec projektowy „Model View Controller"

MVC porządkuje składowe aplikacji w trzy warstwy - warstwę modeli, widoków i kontrolerów. Każda z nich posiada swoje wyjątkowe funkcjonalności, dzięki czemu warstwy mogą być rozwijane niezależnie od siebie. Posługując się MVC sprawiamy, że nasza aplikacja staje się zgodna z jedną z głównych zasad programowania obiektowego - „Separation of concerns".

2.1.1. Model

Model reprezentuje warstwę danych aplikacji. Najczęściej są to klasy, które odzwierciedlają pojęcia istotne z punktu rozwiązywanych przez program problemów. Model powinien spełniać następujące założenia:

- Zapewniać jednolity interfejs dostępu i zapisu danych. Dzięki temu reszta składowych aplikacji nie musi być świadoma tego w jaki sposób są one pozyskiwane.
- Nie posiadać logiki.
- Zmiana sposobu dostępu do danych wymagać będzie zmian jedynie w warstwie modelu.

2.1.2. Widok

Do warstwy widoku aplikacji należy każdy jej fragment, który jest widoczny dla użytkownika - składowe interfejsu graficznego. Rozwijając warstwę widoku należy pamiętać o poniższych zasadach:

- Widok prezentuje użytkownikowi dane w odpowiedniej oprawie graficznej.
- Warstwa nie zna sposobu dostarczania mu udanych.
- Widok zna strukturę dostarczanych mu danych i wie jak je interpretować.
- Logika w widoku powinna zostać ograniczona do minimum.
- Gdyby w przyszłości zaszła potrzeba zmiany interfejsu graficznego, zmiany dotyczyć będą jedynie warstwy widoku.

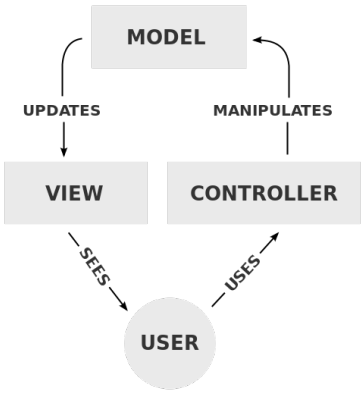
2.1.3. Kontroler

Warstwa kontrolerów zawiera logikę biznesową aplikacji. Obowiązkami Kontrolera są:

- Pobranie odpowiednich danych z warstwy Modelu w zależności od potrzeb użytkownika aplikacji.
- Przygotowanie i ewentualnie wykonanie dodatkowych czynności z wykorzystaniem danych w zależności od potrzeb.
- Spreparowanie i przekazanie danych do wybranego widoku, który zostanie wyświetlony użytkownikowi.

2.1.4. Interakcje

Interakcje między warstwami przedstawia poniższy diagram:



Rys. 2.1. Schemat wzorca projektowego Model View Controller

2.2. Działanie frameworku

ASP.NET MVC pozwala w łatwy oraz intuicyjny sposób obsługiwać rządania HTTP. W dużym skrócie działanie zostaje obsłużone w następujących krokach:

- Framework kieruje rządanie do odpowiedniego kontrolera na podstawie reguł routingu.
- Kontroler generuje odpowiedź posługując się wybranym widokiem, do którego dostarcza spreparowane dane.
- Dane pozyskiwane są przez klasy z warstwy modelu.
- Silnik wypełnia widok danymi oraz zwraca odpowiedź w postaci kodu HTML.

MVC dostarcza również rozwiązań pozwalających na prosty dostęp do parametrów rządaniam, sesji, cookies oraz wiele innych. Mimo bogatej biblioteki oraz wymuszonej struktury projektu, programista posiada dużą swobodę tworząc aplikację.

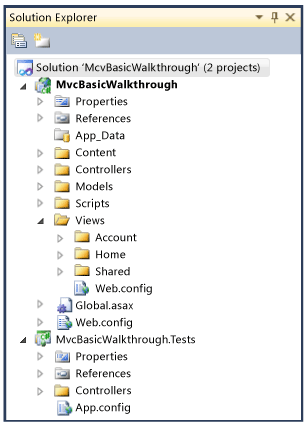
2.2.1. Struktura projektu

Podczas tworzenia projektu MVC zalecane jest korzystane z domyślnej struktury projektu. W takim przypadku oprócz dobrej organizacji kodu, programista zyskuje wiele dodatkowych udogodnień takich jak uproszczone mechanizmy routingu, czy proste powiązanie kontrolera z widokiem. Nazwy folderów w intuicyjny sposób wskazują na to, co powinny zawierać.

2.2.2. Routing

Najprostszym mechanizmem zarządzania routingiem w aplikacji jest trzymanie się konwencji nazewnictwa oraz umieszczania plików w odpowiednich folderach. Przykładowo url o postaci „http://localhost:8080/DeviceSchemes/EditScheme/20” zostanie przekazany do kontrolera o nazwie „DeviceSchemesController” i jego metody „EditScheme”, która powinna posiadać jeden parametr. Parametry mogą zostać rzutowane na typy proste jak i na obiekty, co znacznie upraszcza obsługę złożonych obiektów JavaScript skonwertowanych do postaci JSON.

Drugą częścią konfigurowania routingu jest korzystanie z metod obiektu RouteCollection. Reguły definiuje się określając wzór URLa, oraz kontroler i akcję, do których rządanie zostanie skierowane. Opcjonalnie można zdefiniować też parametry rządania, które zostaną powiązane z parametrami akcji kontrolera. Aby stworzyć powiązanie z poprzedniego przykładu, należy zdefi-



Rys. 2.2. Struktura projektu ASP.NET MVC

niować trasę w następujący sposób:

```
routes.MapRoute("DeviceSchemes", "DeviceSchemes/EditScheme",
   new { controller = "DeviceScheme", action = "EditScheme", schemeId})
);
```

Obiekt RouteCollection umożliwia także tworzenie bardziej uniwersalnych tras dzięki zastosowaniu nawiasów oraz słów kluczowych w URL. Przykładowo dodanie takiego routingu:

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Main", action = "Index", id = UrlParameter.Optional }
);
```

spowoduje powiązanie każdego URL do kontrolera i akcji na podstawie nazewnictwa. W przypadku gdy kontroler i akcja o odpowiednich nazwach nie zostaną znalezione, rządanie zostanie przekierowane do domyślnego kontrolera „MainController” oraz akcji „Index”, której parametry są opcjonalne. Podczas definiowania tras należy pamiętać, iż URL dopasowywany jest do reguł w kolejności ich definicji. Reguły należy zatem deklarować ze szczególną uwagą. Przykładowo, nie ma sensu dodawać reguł do tej samej akcji z różną liczbą parametrów w momencie gdy powyżej znajduje się deklaracja z parametrami opcjonalnymi. Należy również zwrócić uwagę na sytuację, w której powiązanie parametrów rządania z parametrami akcji jest niemożliwe. Dzieje się to, gdy typy parametrów są niezgodne i próba rzutowania kończy się wyrzuceniem wyjątku. Framework nie będzie kontynuował dopasowywania URL do kolejnych reguł.

2.2.3. Warstwa kontrolerów

Klasy będące kontrolerami powinny spełniać następujące założenia:

- Dziedziczyć po klasie Controller.
- Posiadać nazwę która kończy się słowem Controller (np. DeviceSchemesController).
- Znajdować się w katalogu Controllers.

Wszystkie publiczne metody w kontrolerze nazywane są akcjami. W praktyce to właśnie one zawierają logikę biznesową, która obsługuje rządanie HTTP. Typ rządania z jakim powiązana zostanie akcja konfiguruje się za pomocą atrybutów (np. [HttPost] lub [HttpGet]). Rezultat działania

akcji może znacznie różnić się w zależności od zwracanej przez niego wartości. Poniżej znajdują się niektóre z nich:

- View() - zwraca widok o nazwie odpowiadającej nazwie akcji, który musi znajdować się w katalogu o nazwie kontrolera. W przypadku kontrolera „DeviceSchemesController" i akcji „DevicesList" widok powinien nazywać się „DevicesList" i znajdować się w katalogu o ścieżce „Views/DeviceSchemes".
- View(„nazwa widoku") - działa jak wyżej z tą różnicą, iż zamiast nazwy domyślnej można podać jako parametr.
- View(obiekt modelu) - tworzy tak zwany „typowany widok". Szczegóły na jego temat znajdują się w następnym podrozdziale.
- Redirect(„url") - przekierowuje na podany w parametrze adres.
- RedirectToAction(„nazwa akcji", „nazwa kontrolera", [opcjonalne parametry]) - pozwala przekierować rządanie do kolejnej akcji.
- Json(obiekt) - konwertuje dany obiekt do postaci Json. Pozwala to na wygodną obsługę rządań asynchronicznych.

Wszystkie powyższe metody pomocnicze znajdują się w klasie Controller. Dzięki tej klasie możliwe jest również łatwe przesyłanie dowlonych obiektów do zwracanego z kontrolera widoku. Służą do tego obiekty ViewBag i ViewData. ViewData jest mapą, z której można korzystać w następujący sposób:

```
ViewData["lastProjectId"] = lastProjectId;
int lastId = (int) ViewData["lastProjectId"];
```

Minusem mapy ViewData jest to, iż wszystkie wpisywane do niej obiekty są typu Object. Inaczej jest w przypadku ViewBag, który jest dynamicznie tworzonym obiektem. Obiekt ten utrzymuje typy przechowywanych danych, a korzystanie z niego wygląda następująco:

```
ViewData.lastProjectId = lastProjectId;
int lastId = ViewData.lastProjectId;
```

2.2.4. Przykładowy kontroler

Poniżej znajduje się fragment przykładowego kontrolera.

```
namespace Elements.Controllers
{
    public class DesignerController : Controller {

            [HttpPost]
            public ActionResult ProjectsList()
            {
                List<Project> projectsList;
                List<DeviceScheme> schemesList;
                using (var qm = new QueryManager())
                {
                    projectsList = qm.getAllProjects();
                    schemesList = qm.GetAllDevicesSchemes();
                }
```

```
            ProjectsListViewModel model = new ProjectsListViewModel();
            model.DeviceSchemes = schemesList;
            model.Projects = projectsList;
        ViewBag.model = model;

            return View(model);
        }

        ...
    }
}
```

2.2.5. Warstwa widoków

Widokami nazywamy pliki z kodem HTML oraz wstawkami ASPX lub Razor. Służą one do prezentowania użytkownikowi danych spreparowanych przez kontroler. Widok powinien znajdować się w katalogu Views, podkatalogu o nazwie kontrolera, a sam nosić nazwę akcji. Dzieje się tak, gdyż z reguły każdy widok związany jest z określonym kontrolerem oraz akcją. Wstawki ASPX oraz Razor służą do dodawania dynamicznie wygenerowanej treści do widoków. Dzięki nim możliwe jest uzyskanie dostępu do klas pomocniczych oraz obiektów modeli (ViewBag, ViewData, Model) i sesji (Session). W powyższych wstawkach możliwe jest również korzystanie z kodu C#, co umożliwia widokom posiadać logikę. Należy jednak pamiętać, że kod w widokach powinien służyć jedynie do prezentacji danych, a nie zastępować logiki biznesowej kontrolerów. Warto zauważyć, że kod zawarty we wstawkach wykonywany jest po stronie serwera w trakcie renderowania widoku. Do wykonywania działań w aplikacji po stronie klienta używana jest technologia JavaScript. Przykładowy fragment widoku z wykorzystaniem wstawek Razor wygląda następująco:

```
<div id="projectsContainer" class="boxContainer">
    <div class="header">
        <h1>Lista wizualizacji</h1>
    </div>
    <div class="content gradiendBackgroundGray">
        <div id="addProject" class="box addBox"><span>+</span></div>

        @foreach (var project in Model.Projects)
        {
            <div class="box projectBox boxGradientBackground"
                 data-projectname="@project.Name" data-projectid="@project.Id">
                <button class="btn btn-danger actionButton removeButton"><span
                    class="glyphicon glyphicon-remove"></span></button>
                <button class="btn btn-success actionButton updateButton"><span
                    class="glyphicon glyphicon-pencil"></span></button>
                <h4 class="name">@project.Name</h4>
                <p><span class="bold">Grupa: </span><span
                    class="deviceSchemeName">@project.DeviceScheme.Name</span></p>
                <p><span class="bold">Opis: </span><span
                    class="description">@project.Description</span></p>
```

```
            <button class="btn btn-primary actionButtonBig viewProjectButton"><span
                class=" glyphicon glyphicon-search"></span></button>
        </div>
    }
</div>
</div>
```

2.2.6. Cykl życia aplikacji

Rozdział ten zawiera krótki opis życia aplikacji ASP.NET MVC oraz jej składowych. Z punktu widzenia programisty są to bardzo istotne informacje, gdyż tworzenie obiektów klas definiowanych przez programistę często wykonywane jest przez framework. Aby mieć pełną kontrolę nad towrzoną aplikacją warto jest znać kolejne kroki podejmowane przez MVC podczas obsługi rządania HTTP.

W momencie gdy aplikacja jest uruchamiana na serwerze wywoływane są metody klasy MvcApplication, których zadaniem jest konfiguracja aplikacji. Klasa ta jest umieszczona w pliku Global.asax.cs. Pierwszą wywoływaną metodą klasy jest Application_Start. Znajduje się w niej między innymi wywołanie wcześniej wspomnianej metody RegisterRoutes, w której definiuje się trasy routingu. W klasie MvcApplication można dodatkowo zdefiniować „listenery" - metody, które zostają wywołane przez framework w momencie różnych zdarzeń podczas życia aplikacji. Mogą to być na przykład zdarzenia tworzenia oraz niszczenia sesji. Od momentu odebrania rządania HTTP do zwrócenia użytkownikowi odpowiedzi framework wykonuje następujące czynności:

- URL zapytania zostaje porównane z definicjami tras.
- W momencie, gdy zapytanie zostanie dopasowane do kontrolera i akcji framewrok tworzy obiekt odpowiedniej klasy kontrolera.
- Wywoływana jest odpowiednia akcja kontrolera. Gdy posiada ona parametry, są one przekazywane do akcji. Parametry są rzutowane na odpowiednie typy jeśli zachodzi taka potrzeba. W przypadku niepowodzenia rzutowania zostanie wyrzucony wyjątek.
- W zależności od typu zwracanej wartości przez akcję, użytkownikowi użytkownikowi zwracana jest odpowiedź. Może to byc kod HTML, obiekt JSON lub pojedyncza wartość.

Mechanizm obsługi rządania przedstawia poniższy diagram:

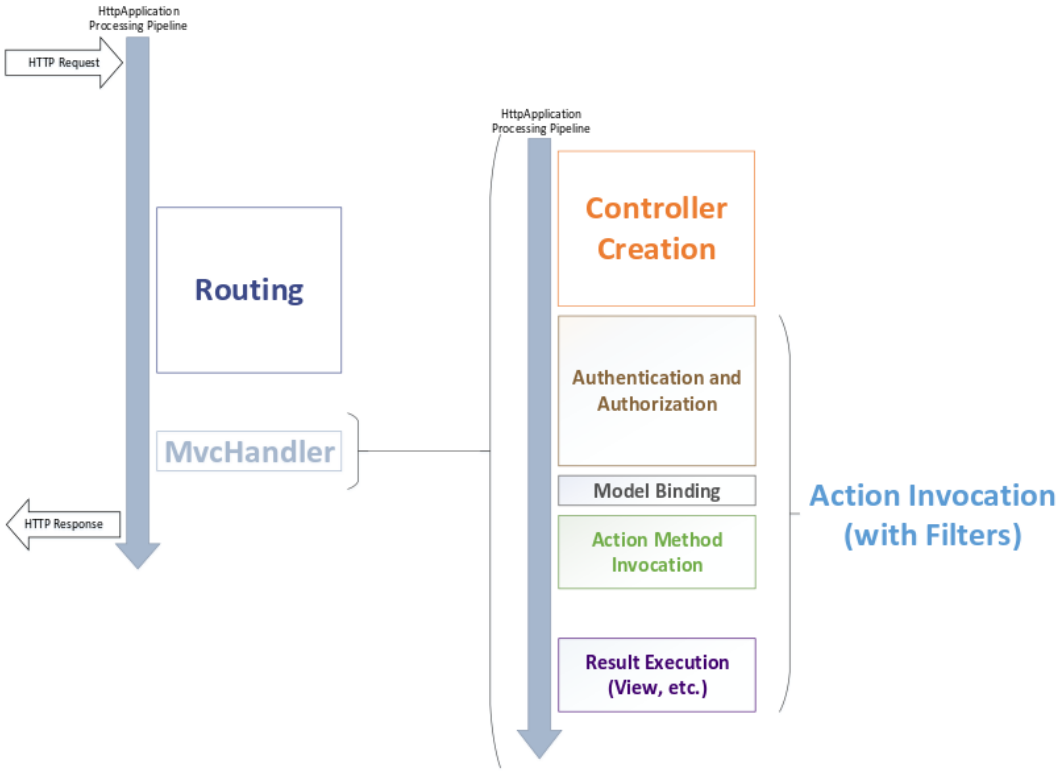
2.2.7. Warstwa modeli

Warstwa modeli zawiera wszystkie klasy, które opisują elementy składowe rozwiązywanego problemu. Warstwa ta powinna zawierać również mechanizmy pobierania oraz zapisu danych modeli. Framework pozostawia w tym miejscu programiście dużą swobodę działania. Nie ma narzuconej metody implementacji warstwy modeli. Jest to logiczne, gdyż aplikacja może pobierać i zapisywać dane na wiele sposobów, na przykład poprzez bazę danych, kolejki, pliki, itp. Przykładowa klasa należąca do warstwy może wyglądać następująco:

```
public class DeviceScheme
{
        public int Id { get; set; }
        public string Name { get; set; }
        public string DevicesNames { get; set; }
        public IList<Project> Projects { get; set; }
```

```
}
```

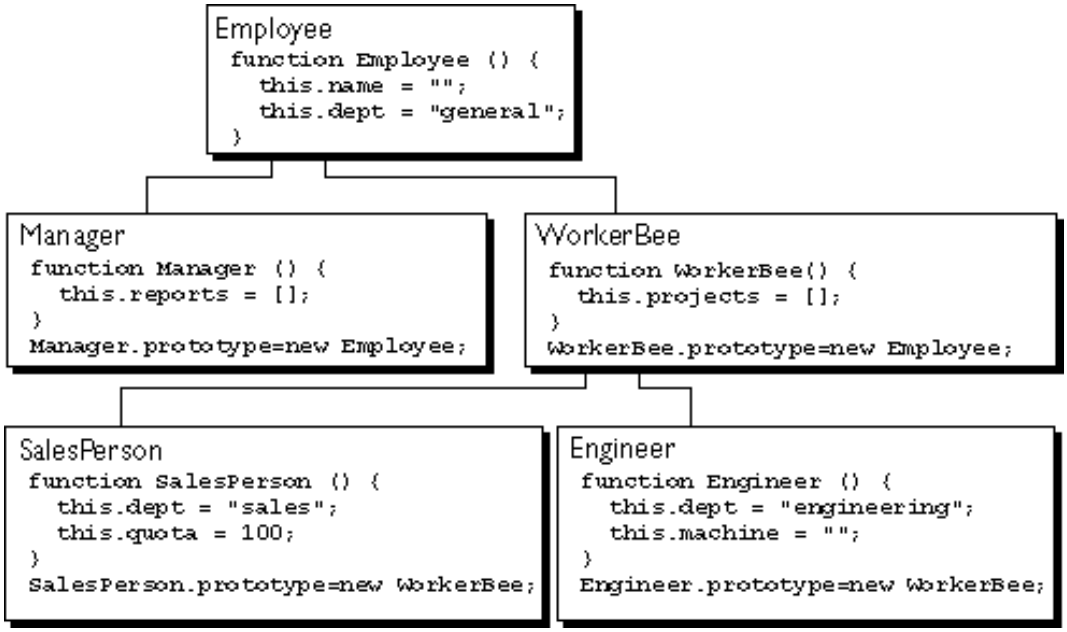
Łatwo zauważyć, iż nie posiada ona żadnej logiki, jedynie istotne atrybuty, które ją opisują.



Rys. 2.3. Obsługa rządania HTTP

3. TYPESCRIPT

JavaScript (JS) jest technologią, która od lat służy programistom do wzbogacania interfejsów stron internetowych o dynamikę. Dzisiaj JavaScript postrzegany jest jako technologia, która sprawia programistom wiele problemów. Wynika to przede wszystkim ze składni języka, braku mechanizmu typowania oraz niespotykanej nigdzie indziej uproszczonej metodzie tworzenia hierarchii klas. Efekt ten jest spowodowany faktem, iż w momencie powstawania języka JavaScript, panowało przeświadczenie, iż nie będzie służył on do tworzenia dużych, kompletnych aplikacji. Postawiono zatem na prostotę rozwiązania. Dzięki niej programista jest w stanie szybko osiągnąć zamierzony cel niewielką ilością kodu.



Rys. 3.1. Mechanizm dDzisiaj rziedziczenia z wykorzystaniem obiektu prototype

Rozwój aplikacji internetowych oraz przeglądarek pokazał, iż tworzenie bogatego interfejsu aplikacji daje wiele korzyści. JavaScript z biegiem lat nabierał znaczenia, gdyż coraz więcej przeglądarek wspierało tę technologię. W roku 1996 w oparciu o JS stworozny został standard o nazwie ECMAScript, który obsługiwany jest dziś przez wszystkie najpopularniejsze przeglądarki.

Programowanie interfejsów aplikacji internetowych doszło zatem do punktu, w którym programiści zmuszeni są używać JavaScript do tworzenia dużych aplikacji. Stało się to pomimo faktu, iż technologia ta została zaprojektowana do innego celu. Odpowiedzią na ten problem jest język TypeScript autorstwa Microsoftu.

3.1. Cechy technologii

Język TypeScript (TS) składa się z dwóch elementów - jęztenyka oraz kompilatora. Oba z nich są projektami na licencji OpenSource dostępnymi dla każdego. Aby lepiej zrozumieć możliwości technologii należy zapoznać się z następującymi faktami na jej temat:

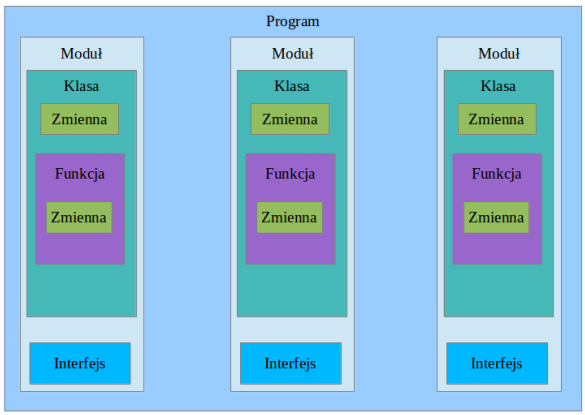
- Język TypeScript jest kompilowany do języka JavaScript. Oznacza to, że poza kilkoma dodatkowymi słowami kluczowymi wystarczy znajomość JavaScript'u do posługiwania się TypeScript'em.
- Główną cechą TS rozszerzającą JS jest możliwość definiowania typów zmiennych. Poprawność typu sprawdzana jest przed kompilacją kodu TS do JS.
- Każdy fragment kodu napisany w JavaScript może zostać użyty do rozwoju klas w

języku TypeScript. Dzięki temu możliwe jest wykorzystywanie popularnych bibliotek takich jak jQuery, MooTools, itp. Społeczność regularnie dostacza pliki TypeScript, które zawierają definicje typów obiektów używanych przez różne biblioteki. Pozwala to na wygodniejsze stosowanie ich zgodnie z zasadami języka TypeScript.

- Kompilator TypeScript'u zawarty jest w jednym pliku JavaScript i może zostać dołączony do każdego środowiska. Technologia nie posiada maszyny wirtualnej i nie ma planów na jej rozwój.

3.2. Elementy składowe TypeScript

Język programowania TypeScript został stworzony na podobieństwo języka C#. Nazewnictwo oraz przeznaczenie elementów składowych języka są niemal identyczne. Schemat programu wykonanego w technologii TypeScript wraz z jego elementami składowymi zobrazowany jest na poniższym rysunku:



Rys. 3.2. Schemat wzorca projektowego Model View Controller

Przez pojęcie „program" w technologii TypeScript rozumiemy zbiór plików źródłowych z rozszerzeniem „.ts". Pliki te mogą zawierać poniższe elementy:

- Moduł - może zawierać wiele klas oraz interfejsów. Można o nich myśleć jak o namesace'ach w języku C#.
- Intefejs - wymusza na klasie posiadanie określonych metod. W TypeScript'cie klasy mogą implementować wiele interfejsów. Interfejsy nie powodują generowania dodatkowego kodu JavaScript. Służą one do wymuszania przez kompilator na programiście zachowania zdefiniowanych kontraktów między klasami.
- Klasa - posiada zmienne oraz funkcje. Dostępne są dwa modyfikatory dostępności funkcji - public oraz private. Klasa może dziedziczyć po jednej, innej klasie.
- Funkcja - zawiera fragment logiki programu. Może przyjmować argumenty oraz zwracać wartość określonego typu lub void. Przykładowa deklaracja funkcji wygląda następująco:

```
function getNumberString(parameter: number): string {
   return 'Podana liczba to: ' + parameter.toString();
}
```

- Zmienna - może być określonego lub dowolnego typu. Dowolny typ należy oznaczać słowem kluczowym „any". Kontrola typów odbywa się na poziomie kompilatora, gdyż

nie jest ona wymuszana przez JavaScript w trakcie działania programu. Sprawdzenie typu przez kompilator odbywa się w przypadku przypisywania wartości do zmiennej, wywoływania funkcji. TypeScript posiada kilka wbudowanych typów. Są nimi:

number - 64-bitowa zmienna zmiennoprzecinkowa

boolean - przyjmuje wartości true lub false

string - sekwencja znaków UTF-16

tablice - definiuje się je za pomocą nawiasów kwadratowych „[]” lub za pomocą klasy Array

3.3. Integracja TypeScript z istniejącym kodem JavaScript

TypeScript umożliwia integrację z istniejącymi fragmentami kodu JS. Jest to przydatne w sytuacjach, gdy w TypeScript chcemy skorzystać ze zmiennych, które zdefiniowane się bezpośrednio w kodzie strony. Przykładowo:

```
...
<script type="text/javascript">
   var zewnetrznaZmienna = 10;
</script>
<script type="text/javascript" src="naszKod.js" />
...
```

Aby w pliku „naszKod” skorzystać ze zmiennej zewnetrznaZmienna należy użyć słowa kluczowego „declare”.

```
declare zewnetrznaZmienna : Number;
```

3.3.1. TypeScript i jQuery

Warto wspomnieć, iż w JavaScript możliwe jest przypisywanie do wskaźnika obiektu this dowolnej wartości. Niektóre biblioteki takie jak jQuery wykorzystują to, by ułatwić użytkownikowi manipulację elementami drzewa strony DOM. Dzieje się to na przykład podczas korzystania z metody each:

```
$('.klasa').each(function(){
   $(this).attr('id', 'noweId');
});
```

Powyższy kod iteruje po wszystkich węzłach o klasie „klasa” i nadaje im nowe id. Dzięki temu, iż funkcja each przypisuje do obiektu this wartość kolejnego iterowanego węzła, możliwe jest proste operowanie na jego parametrach. Jest to jednak kłopotliwe w przypadku języka TypeScript.

Wyobraźmy sobie, iż programista chciałby napisać metodę, która przypisze do pola obiektu wartość tekstową elementu o zadanym id. W poniższym przykładzie nie możliwe jest odniesienie się wewnątrz funkcji each do pól / metod obiektu.

```
class Finder{
   var nameToFind: String;
```

```
    public void findName(){
        $('.klasa').each(function(){
            if ((this).attr('id') == 'nazwa') {
                this.nameToFind = $(this).val(); //blad!
            }
        });
    }
}
```

Twórcy technologii TypeScript dodali odpowiednią składnię do języka, która pomaga obejść ten problem. Wystarczy posłużyć się tak zwaną „arrow function", która zwyczajnie zapisze wskaźnik this do zmiennej pomocniczej _this przed wywołaniem metody w której jest on podmieniany, a następnie przywróci jego wartość po jej wykonaniu. Aby naprawić powyższy przykład należy dokonać takiej modyfikacji:

```
public void findName(){ () => {
    $('.klasa').each(function() {
        if ((this).attr('id') == 'nazwa') {
            this.nameToFind = $(this).val(); //blad!
        }
    }
});
}
```

Skompilowany kod będzie miał następującą postać:

```
Finder.findName = function(){
    var _this = this;
    $('.klasa').each(function() {
        if ((this).attr('id') == 'nazwa') {
            _this.nameToFind = $(this).val();
        }
    })
    this = _this;
    };
```

4. OPIS ZASTOSOWANEGO ROZWIąZANIA

Rozdział ten zawiera opis zastosowanego w ramach pracy rozwiązania służącego do zdalnego nadzorowania pracy urządzeń. Rozwiązanie to zostało zaprojektowane dla dwóch typów użytkowników. Pierwszym z nich jest projektant widoków – osoba odpowiedzialna za ich tworzenie, która posiada podstawową wiedzę na temat urządzeń. Drugim typem użytkownika jest osoba upoważniona do oglądania widoków. W żargonie branży kolejowej termin „widok" można zastąpić słowem „wizualizacja".

Rozwiązanie zostało zaimplementowane jako aplikacja internetowa składająca się z dwóch modułów. Każdy z nich przeznaczony jest dla odpowiedniego typu użytkownika. Zadaniem pierwszego modułu aplikacji - „Elements Designer" jest projektowanie widoków. Są to strony złożone z elementów graficznych, które pozwalają na wizualizację stanu urządzenia. Drugi moduł o nazwie „Elements Viewer" służy do wyświetlania tych widoków wraz z nawiązaniem połączenia z urządzeniami i dostarczeniem danych w odpowiednie miejsca wizualizacji. Kolejne podrozdziały zawierają szczegółowy opis funkcjonalności modułów.

4.1. Elements Designer - Moduł projektowania widoków

Moduł Designer pozwala na projektowanie widoków urządzeń. Jego interfejs składa się z dwóch głównych elementów – menu bocznego oraz głównego obszaru roboczeka, który prezentuje obecny stan projektu. Obszar ten nosi nazwę „sceny". Dzięki temu rozwiązaniu projektant widzi efekt swojej pracy natychmiast po dokonaniu zmian. Projektowanie widoku polega na wybieraniu tzw. komponentów z menu, umieszczaniu ich w odpowiednim miejscu na scenie i konfigurowaniu wedle potrzeb.

4.1.1. Tworzenie projektu

Po wybraniu modułu z odpowiedniej sekcji menu aplikacji użytkownikowi ukazuje się strona wyboru projektu. Każdy projekt widoczny jest jako kafelek z nazwą i opisem projektu. Użytkownik może z tego miejsca przejść do edycji istniejącego projektu lub stworzyć nowy. W takiej sytuacji należy uzupełnić nazwę, opis oraz grupę urządzeń, dla której projektowany będzie widok. Po zatwierdzeniu tych danych użytkownikowi ukazuje się Scena.

4.1.2. Scena

Scena jest obszarem roboczym, który gromadzi wszystkie komponenty. Po zapisaniu projektu, możliwe będzie wyświetlenie jej przez moduł Viewer wraz ze wszystkimi komponentami naniesionymi na nią w trakcie projektowania. Jedynym atrybutem sceny, który można konfigurować jest kolor jej tła.

4.1.3. Komponenty

Komponenty to podstawowe elementy składowe projektu, które pełnią różne funkcje. W praktyce jest to każdy element dodany przez użytkownika do Sceny w trakcie projektowania. W aplikacji Elements istnieją trzy rodzaje komponentów, które można rozróżnić na podstawie ich funkcji:

- Kontener – służy do grupowania innych komponentów. Można w nim umieścić dowolny komponent, ale sam również może być umieszczony w innym kontenerze. Aby umieścić element w kontenerze należy chwycić go myszą i „upuścić" nad wybranym komponentem typu kontener. Czynność tę można odwrócić wybierając opcję „Przenieś wyżej" z menu kontekstowego komponentu.

- TextBox – jego funkcja to przechowywanie tekstu, który można edytować. Dzięki temu możliwe jest tworzenie opisów innych elementów projektu. Tekst wpisywany jest podczas tworzenia komponentu. TextBox można edytować w każdej chwili wybierając odpowiednią opcję z jego menu kontekstowego.
- RefreshedVariable – element ten powiązany jest z konkretną zmienną urządzenia, którą należy wybrać podczas tworzenia komponentu. Jego główną funkcją jest regularne odświeżanie wartości zmiennej urządzenia w przypadku jej zmiany. Dodatkowo można skonfigurować zmianę tekstu i koloru tła komponentu w chwili gdy wartość zmiennej wyniesie lub przekroczy zdefiniowaną wartość. Pozwala to na zwrócenie uwagi użytkownika na zmienne w wyjątkowych sytuacjach. Można też ustawić komponentowi wartość domyślną. Dzięki temu komponent nie musi wcale zawierać liczby, a jedynie tekst informujący o stanie w jakim znajduje się urządzenie. Kolejną opcją jest nałożenie maski bitowej na zmienną. Przed wyświetleniem jej wartości element przemnoży ją przez liczbę ustawioną wcześniej jako maska. Jest to przydatne w przypadku gdy jedną zmienną należy interpretować jako więcej niż jedną liczbę. Ostatnią możliwością jest ustawienie wartości mnożnika. Komponent po prostu przemnoży wartość zmiennej przez mnożnik przed jej wyświetlaniem. W przypadku wybrania przez użytkownika opcji maski bitowej i mnożnika jednocześnie, mnożenie wykonywane jest jako ostatnia operacja.

Wszystkie komponenty posiadają dodatkowo kilka wspólnych właściwości, które można konfigurować wedle upodobań. Są nimi kolor tła, kolor obramowania, współrzędne oraz rozmiar. Wszystkie wyżej wspomniane właściwości można zmienić w dowolnej chwili posługując się menu kontekstowym dostępnym pod prawym przyciskiem myszy. Dla wygody użytkownika rozmieszczenie komponentu określa się przeciągając go po scenie z miejsca na miejsce. Rozmiar natomiast reguluje się chwytając za wybrany narożnik komponentu i rozciągając go.

W momencie gdy użytkownik uzna iż projekt jest już gotowy musi go zapisać, aby zmiany nie zostały utracone. Służy do tego guzik z ikoną dyskietki znajdujący się w menu bocznym. Do edycji można wrócić w każdej chwili wybierając projekt ponownie z listy projektów.

4.2. Elements Viewer – moduł wyświetlania widoków

Moduł Viewer służy do prezentowania widoków użytkownikom końcowym – głównie kolejarzom. Viewer jest w stanie wczytać projekt widoku i wyświetlić go użytkownikowi utrzymując przy tym połączenie z urządzeniami i wyświetlając ich zmienne w skonfigurowany przez projektanta sposób.

4.2.1. Lista urządzeń

Pierwszą stroną, która ukazuje się użytkownikowi po uruchomieniu modułu jest lista wszystkich urządzeń podłączonych do aplikacji Elements. Konfiguruje się ją edytując odpowiedni plik xml w folderze aplikacji. Każde urządzenie na liście reprezentowane jest przez pojedynczy kafelek, który wyświetla jego nazwę. Po kliknięciu na guzik z ikoną lupy użytkownik zostaje przeniesiony do listy projektów dostępnych dla urządzenia.

4.2.2. Lista projektów

Interfejs listy projektów prezentuje się analogicznie do listy urządzeń. Każdy projekt reprezentowany jest przez kafelek z jego nazwą i opisem. Aplikacja przyporządkowuje widoki do urządzeń na podstawie przynależności urządzenia do grupy. Oznacza to, że wiele urządzeń może zostać przeniesionych do tego samego widoku, który wyświetli w nim zmienne wybranego urzą-

dzenia.

4.2.3. Wizualizacje pracy urządzeń

Głównym elementem widocznym na stronie projektu jest Scena. Wygląda ona prawie identycznie jak w module Designer. Różnica polega na tym, że menu projektowania jest ukryte. Użytkownik nie może też przesuwać komponentów ani wywołać menu kontekstowego. Widok jest przeznaczony jedynie do oglądania.

Po wybraniu projeku użytkownikowi ukazuje się układ komponentów wraz z nadanymi im atrybutami przez projektana. Aplikacja nawiązuje połączenie z serwerem wymiany danych z urządzeniami w celu uzyskania wartości zmiennych dla komponentów typu RefreshedVariable. Połączenie z serwerem utrzymywane jest przez cały czas, gdy użytkownik znajduje się na stronie projektu. Dzieje się tak, ponieważ aplikacja regularnie kontroluje, czy wartości zmiennych nie uległy zmianie i w takim przypadku aktualizuje je w komponentach. Aktualizacja zmiennej polega na wyświetleniu jej w komponencie RefreshedVariable, który jej z nią związany. Przed wyświetleniem zmiennej aplikacja wykonuje dodatkowe czynności, uwzględniając konfigurację danego komponentu przez projektanta (zmiana tekstu, zmiana koloru, maska bitowa, mnożnik).

5. IMPLEMENTACJA

Aplikacja Elements została zaprojektowana w taki sposób, by z łatwością można było ją dopasować do innych składowych systemu ogrzewania rozjazdów kolejowych. W związku z tym, aplikacja przygotowana jest do instalacji na serwerze xsp4 zintegrowanym z apache, przy użyciu maszyny wirtualnej Mono. Wyżej wymienione technologie są kompatybilne z systememami z rodziny Unix, na których działa reszta aplikacji systemu.

5.1. Wykorzystane biblioteki

5.1.1. Razor

Razor jest silnikiem generowania widoków autorstwa Microsoft, który w swojej składni jest znacznie prostszy niż jego poprzednik – aspx. Dzięki temu tworzenie stron, które wymagają użycia logiki staje się jeszcze łatwiejsze. Uproszczenie składni polega głównie na zastąpieniu znaczników aspx jednym znakiem - „@”. Służy on do zadeklarowania w widoku intencji użycia zmiennych dostarczanych do niego przez kontroler. Wydawać by się mogło, że nie jest to duża różnica, lecz w przypadku skomplikowanych konstrukcjach warunkowych łatwo dostrzec przewagę silnika Razor nad aspx. Dla przykładu następującą składnię aspx:

```
<%if (Model.Length == 0)
{%>
   <p>Brak</p>
 <%}
       else
 {%>
   <p><%=Model.Item%></p>
<%} %>
```

można zastąpić w taki sposób:

```
@if (Model.Length == 0)
{
        <p>Brak</p>
}
 else
{
        <p>@Model.Item</p>
}
```

W widoku można umieścić sekcję zawierającą logikę. W tym celu należy umieścić linie kodu w konstrukcji @{...}. Warto jednak nie zapominać o podstawowym założeniu MVC – widok powinien wyświetlać dane, a nie służyć do ich generowania [7]. Razor zawiera również prosty mechanizm tworzenia szablonów stron. Dzięki temu ilość kodu w plikach widoków znacznie się zmniejsza, gdyż powtarzalne części strony takie jak menu, czy stopka można wyeksportować do odrębnych plików.

5.1.2. NHibernate

Nhibernate jest biblioteką, która służy do powiązania tabel baz danych na obiekty (Object Related Mapping) [8]. Dzięki niej możliwe jest posługiwanie się prostymi obiektami, które swoją strukturą odpowiadają odpowiednim tabelom. Dużą zaletą NHibernate w stosunku do innych bibliotek spełniających tę funkcję jest możliwosć konfiguracji w kodzie. Skonfigurować w kodzie można zarówno sposób wiązania obiektów z tableami, jak i samo połączenie z bazą. Najprostszą metodą konfiguracji jest wskazanie, w którym pakiecie znajdują się obiekty, które powinny zostać powiązane z tabelami za pomocą klasy Configuration. Należy jedynie pamiętać, by nazwy obiektów oraz ich pól były identyczne jak odpowiednie nazwy tabel oraz ich kolumn. Dzięki klasie Configuration uzyskujemy dostęp do obiektu typu SessionFactory, kóry pozwala na otwieranie i zamykanie sesji z bazą danych.

Kolejną zaletą NHibernate jest możliwość wykorzystania technologii LINQ do wykonywania zapytań. Dzięki temu kod C# zawierający zapytanie jest bardzo podobny do odpowiadającego mu zapytania SQL i naturalnie opisuje intencje programisty. Na przykład zapytanie SQL o postaci:

```
SELECT
    Name
FROM
    Device
WHERE
    Id = 1;
```

można zapisać w następujący sposób:

```
string name = session.QueryOver<Device>()
    .Where(d => d.Id == 1)
    .Select(d => d.Name)
    .SingleOrDefault<string>();
```

Używanie biblioteki opiera się głównie na wykorzystaniu obiektu SessionFactory. Służy on do otwierania połączenia z bazą danych, wykonywania operacji oraz zamykania połączenia. Wszystkimi wymienionymi czynnościami zajmuje się biblioteka, pozwalając programiście skupić się na logice biznesowej programu.

5.1.3. jQuery

jQuery to biblioteka, która pozwala na osiąganie atrakcyjnych efektów wizualnych na stronie WWW niewielkim nakładem pracy programisty. Jak wiadomo z dokumentacji [9], całość wykonana jest w oparciu o technologię JavaScript. Biblioteka jest niezależna od przeglądarki więc programista nie musi dostosowywać kodu do wielu różnych przeglądarek WWW. Korzystanie z biblioteki polega głównie na korzystaniu z obiektu jQuery lub znaku "$", który jest aliasem do tego obiektu. jQuery umożliwia wybieranie tablicy węzłów DOM strony internetowej, a następnie wykonywanie na nich różnych działań za pomocą języka JavaScript. Wybieranie węzłów wykonuje się wykorzystując selektory języka CSS3. Przykładowe działania to:

- dodawanie, usuwanie węzłów
- odczytywanie i modyfikowanie atrybutów i zawartości węzłów
- modyfikowanie stylu węzłów
- animacje

- rozbudowana obsługa zdarzeń takich jak kliknięcie lub przesunięcie kursora nad element

Wszystko to można osiągnąc za pomocą kilku linii kodu, na przykład jeśli programista chciałby zmienić kolor tekstu wszystkich węzłów o klasie „red”, wystarczy posłużyć się następującym kodem:

```
$('.red').css('color': 'red')
```

Kolejną istotną funkcją jQuery z punktu widzenia mojej aplikacji jest możliwość wykonywania zapytań synchronicznych jak i asynchronicznych AJAX. Znacznie przyspiesza to tworzenie aplikacji, gdyż różne przeglądarki wymagają obsługi wyżej wymienionych zapytań na różne sposoby.

W internecie dostępnych jest wiele wtyczek do biblioteki, które rozszerzają funkcjonalność jQuery. Dzięki nim można na przykład w szybki sposób tworzyć interaktywne tabele z danymi (DataTables), przybornik do wyboru koloru (Evol), czy efektywny pokaz slajdów (Fotorama).

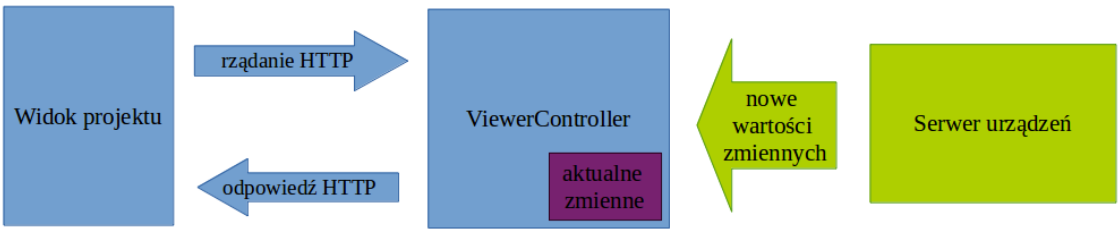
5.2. Architektura aplikacji

Zastosowane przeze mnie rozwiązane zostało zaimplementowane jako aplikacja internetowa. Moduł Viewer ma za zadanie wyświetlanie stanu urządzeń w przeglądarce internetowej, jest to zatem jedyny słuszny wybór. Drugi z modułów służy do projektowania tych widoków. Aby maksymalnie ułatwić użytkownikowi tę czynność, moduł Designer został zrealizowany w ten sam sposób. Dzięki temu projektant widoków ma natychmiastową możliwość oceny rezultatów swojej interakcji z aplikacją. Aplikacja została nazwana „Elements”. Nawiązuje to do małych elementów, z których budowane są widoki.

Architektura projektu Elements składa się z trzech warstw zgodnie z ideą wzorca projektowego MVC. Jest to wymuszone przez framework ASP.NET MVC4.

5.2.1. Warstwa prezentacji - Widoki

Warstwą prezentacji (View) jest interfejs graficzny modułów Designer oraz Viewer. Został on wykonany w dwóch technologiach. Strony internetowe budowane są za pomocą silnika tworzenia widoków Razor. Widoki te zawierają również logikę napisaną w języku TypeScript, który kompiluje się do kodu Javascript. Dzięki temu połączeniu użytkownik jest w stanie poruszać się po interfejsie aplikacji w płynny sposób. Nie ma konieczności przeładowywania formularza na stronach z każdą jego akcją dzięki zastosowaniu zapytań asynchronicznych AJAX. Właśnie na nich oparty w większości jest moduł Viewer. Moduł regularnie wysyła zapytania asynchroniczne do serwera aplikacji w celu pozyskania aktualnych zmiennych. W ten sposób osiągniety został efekt monitorowania urządzeń w czasie rzeczywistym.



Rys. 5.1. Mechanizm odświeżania wartości zmiennych urządzeń

Komponenty zostały zaimplementowane za pomocą klas Typescript. Klasy reprezentu-

jące komponent dziedziczą po klasie Component oraz implementują któryś z interfejsów - Container, TextBox, RefreshedVariable, w zależności od ich przeznaczenia. Dzięki temu aplikacja może za pomocą jednego mechanizmu wyświetlić każdy komponent, zachowując przy tym jego specyficzne zachowanie. Scena jest wyjątkowym komponentem typu kontener. W module Designer pozwala to na przechowywanie komponentów, organizując je w strukturę drzewa. W module Viewer Scena regularnie odświeża zmienne w posiadanych komponentach typu RefreshedVariable. Dzieje się to w następujący sposób:

- Scena iteruje po drzewie komponentów które zawiera i gromadzi listę zmiennych związanych z elementami typu RefreshedVariable
- wysyłane jest zapytanie asynchroniczne z listą zmiennych w celu pozyskania ich
- po odebraniu odpowiedzi z serwera Scena iteruje po swoich komponentach i powiadamia je o zdarzeniu przybycia zmiennej
- Każdy komponent reaguje na przyjście nowej zmiennej w sposób wcześniej skonfigurowany przez projektanta widoków. Wykorzystany został tutaj wzorzec projektowy Observer-Observable.

Komponenty są zdolne do reagowania na zdarzenie przyjścia nowej zmiennej dzięki posiadanych przez nie list tzw. działań (rodzina klas dziedzicząca po klasie Behavior). W momencie przyjścia odpowiedzi z serwera zawierającej zmienne, Scena przekazuje ich wartości do odpowiednich komponentów. Komponenty te uruchamiają metody posiadanych działań, które sprawdzają czy wartość zmiennej spełnia warunek wykonania akcji np. czy wartość zmiennej jest większa od nadanej przez projektanta wartości. Jeśli tak jest, działanie modyfikuje odpowiednie pola komponentu zmieniając np. jego kolor lub tekst. Taki model klas nazywany jest wzorcem projektowym Dekorator. W Elements zostały zaimplementowane następujące zachowania:

- ColorChangeBehavior - zmienia kolor komponentu
- TextChangeBehavior - zmienia tekst komponentu
- ValueChangeBehavior - zmienia wartość numeryczną komponentu - każdy komponent domyślnie posiada to zachowanie

Do komponentu można dodać wiele zachowań jednocześnie. W takim przypadku kolejność zmian atrybutów komponentów jest zależna od kolejności dodania zachowania.

5.2.2. Warstwa logiki biznesowej - Kontrolery

Do warstwy logiki biznesowej w Elements należy część aplikacji wykonana w języku C#, która wykonywana jest przez serwer aplikacyjny. Logika warstwy zawarta jest w tzw. kontrolerach - klasach, których zadaniem jest odbiór rządania HTTP od warsty prezentacji i odesłania jej odpowiedzi. Odpowiedzią może być odpowiednio spreparowany widok lub pojedyncza wartość. Jako że aplikacja Elements jest oparta w dużej mierze na zapytaniach asynchronicznych, zadaniem każdego kontrolera jest dostarczenie odpowiedniego widoku oraz obsługa jego zapytań AJAX.

Głównym zadaniem kontrolera modułu Designer (klasa DesignerControler) jest dostarczanie widoków do tworzenia, usuwania i edytowania projektów. Zapis projektu odbywa się porzez serializację klasy JSONProject dostarczanej z widoku, która zawiera drzewo komponentów, nazwę oraz opis projektu. Kontroler jest w stanie zapisać tę kalsę do pliku jak i do bazy danych. Przesłanie projektu do widoku wykonane jest analogicznie, poprzez pobranie danych o projekcie z bazy i przesłanie ich do widoku w postaci obiektu klasy JSONProject.

Kontroler modułu Viewer (ViewerController) odpowiada za dostarczenie widoków prezentujących listę projektów oraz wyświetlających ich zawartość. Projekty przesyłane są do widoku w taki sam sposób jak opisano powyżej. Głównym zadaniem kontrolera jest dostarczanie zmien-

nych do widoku projektu. Utrzymuje on stałe połączenie z serwerem urządzeń. Dzieki protokołowi wymiany danych P5, serwer urządzeń powiadamia kontroler o nadejściu nowych zmiennych oraz umożliwia natychmiastowe pobranie ich. Wartości zmiennych przechowywane są w słownikowej strukturze danych, która dodatkowo zawiera informacje o tym, czy określona zmienna zmieniła się od ostatniej aktualizacji widoku. Aby maksymalnie zwiększyć szybkość odświeżania wszystkich zmiennych widoku, kontroler przesyła do widoku jedynie te zmienne, które zmieniły swoje wartości od ostatniej aktualizacji.

Pozostałe kontrolery (klasy DeviceSchemesControler, MainControler) zajmują się tworzeniem i edycją grup urządzeń zapisywanych do bazy danych oraz nawigacją po aplikacji.

5.2.3. Warstwa obsługi danych - Modele

Aplikacja Elements zawiera dwa rodzaje modeli:

- Pasywne - wszystkie klasy po stronie serwera zaimplementowane w tehcnologii C#. Służą one głównie do przesyłu danych z widoku do kontrolera oraz do komunikacji z bazą danych. Potrzebne jedynie do reprezentacji fragmentów realizowanego projektu. Nie zmieniają same swojego stanu, gdyż nie posiadają one żadnej logiki.
- Aktywne - zaimplementowane w technologii Typescript. Służą do reprezentacji graficznych elementów interfejsu użytkownika np. scena, komponenty lub urządzenia. Są w stanie zmieniać swój stan, odświeżając przy tym interfejs.

Do przechowywania danych została użyta baza PostgreSQL. Wynika to z faktu, iż jest ona już wykorzystywana w istniejącym systemie ogrzewania i oświetlania rozjazdów. Dodatkowymi atutami bazy PostgreSQL są: [10]

- konkurencyjna wydajność
- bogata w typy danych
- dojrzałość projektu, duża społeczność

6. PODSUMOWANIE

Every diploma thesis must include a chapter entitled **Summary**. It should appear before the **Bibliography** and include a review of the main points of the thesis and/or obtained results. The chapter should also state what should be realized if research into the subject of the thesis is continued.

BIBLIOGRAFIA

[1] Adam Freeman. *Pro ASP.NET MVC 4 4th Edition*. Springer Science+Business Media, 2012.

[2] Douglas Crockford. *JavaScript: The Good Parts*. O'Rilley Media Inc., 2008.

[3] P. Hartig S. Sawchuk C. Eberhardt A. Verschaeve S. Saccone A.Osmani, S. Sorhus. *http://todomvc.com*. 1 luty 2016.

[4] J. Ashkenas. *http://backbonejs.org*. 1 luty 2016.

[5] Steve Fenton. *TypeScript Succinctly*. Syncfusion Inc., 2013.

[6] Google Inc. *https://angular.io/*. 1 luty 2016.

[7] Helm R. Vlissides J. Gamma E., Johnson R. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] NHibernate Community. *NHibernate documentation*. 1 wrzesień 2015.

[9] jQuery Foundation. *jQuery API Documentation*. 1 wrzesień 2015.

[10] Leo S. Hsu Regina O. Obe. *PostgreSQL: Up and Running*. O'Reilly Media, 2012.

SPIS RYSUNKÓW

SPIS TABLIC