

# Воведни напомени

TypeScript (како **над**множество на JavaScript) **прави разлика** помеѓу мали и големи букви во идентификатори (исто како и C#).

Праксата покажа дека при откривање на JavaScript грешки при извршување (run-time) повеќе корисни информации (во вид на пораки за грешки) се добиваат ако за тестирање се користи Google Chrome отколку Microsoft Internet Explorer (и во двата интернет прегледувачи се користи F12 Developer Tools и грешките се гледаат во Console табот). СЕ разбира, можи да се користат и други интернет прегледувачи како Microsoft Edge, Opera, Mozilla или Mozilla Firefox.

CRUD е кратенка која обично ги означува основните операции со објекти во една апликација (**C**reate/**R**etrieve/**U**pdate/**D**elete)

# Dependency Injection

*Dependency Injection (DI)* е техника која се користи при развој на софтвер, во која еден објект (клиент) користи услуги на друг објект (сервис). При тоа, објектот-сервис (кој уште се нарекува **dependency** бидејќи другиот објект зависи од него) се инјектира (оттаму и терминот **injection**) во објектот-клиент (најчесто при создавање на објектот со повикување на неговиот конструктор).

Обично самата развојна платформа и/или програмски јазик му овозможуваат на објектот-клиент да го најди објектот-сервис. За таа цел, обично постои **глобална листа** во која се дефинирани / регистрирани објектите кои можат да се инјектираат, заедно со информација (мета-податоци) како да се создаде нов објект.

Во AngularJS интензивно се користи DI, и тоа на следниот начин:

1. Класите кои треба да бидат инјектирани како зависности во други класи се дефинираат (регистрираат), така што се определува идентификатор (стринг) според кој сите DI клиенти ќе можат да го најдат DI сервисот.
2. При дефинирањето, заедно со идентификаторот се задава и класа од која се објектите кои се инјектираат (за да можи да се создаде нов објект доколку е тоа потребно).

Пример 1: AngularJS сервисите (кои можат да се јават во улога на DI сервис, но и самите можат да бидат DI клиенти) се дефинираат за DI така што после дефиницијата на класата KlasaServis се додава наредба во облик:

angular

.module("ime.na.modul")

.service(“**string identifikator**”, KlasaZaServis);

Пример 2: AngularJS контролерите кои треба да се создадат во време на извршување на апликацијата и кои се придружени на некоја web страна, се дефинираат со слична наредба, после дефиницијата на класата за контролерот:

angular.module("ime.na.modul")

.controller(“**string identifikator**”, KlasaZaController);

На овој начин на AngularJS платформата му се кажува кој DI идентификатор одговара на која класа.

Следен чекор е дефинирањето на врската помеѓу DI клиентот и DI сервисите од кои тој зависи. За оваа цел во секоја AngularJS класа постои поддршка за статичката низа $inject во која се наведуваат **идентификаторите** на DI сервисите од кои зависи класата (како DI клиент). На пример, ако некој AngularJS контролер зависи од два AngularJS сервиси, неговиот $inject ќе гласи:

static $inject = [

“identifikator na service 1”,

"Identifikator na service 2"

];

А неговиот конструктор ќе ја има следната декларација:

constructor(private service1: IService1, private service2: IService2) {

...

}

**Важно е бројот и редоследот на стринг идентификатори во $inject да одговара на бројот и редоследот на параметри во конструкторот. Исто така е важно сервисите кои се дефинирани со дадените идентификатор да ги имплементираат интерфејсите кои се наведени како типови на параметри на конструкторот.**

Напомена: Пракса е за секој AngularJS сервис (не DI сервис) прво да се декларира **интерфејс** кој ќе ги содржи сите јавни методи на тој сервис, а потоа интерфејсот да се користи секаде каде што се работи со сервисот (како во горниот пример со IService1 и IService2). Со ова се постигнува независност од имплементацијата: наместо првичната имплементација на сервисот подоцна можи да се дефинира и регистрира друга имплементација која ќе го почитува истиот интерфејс, а DI клиентите на AngularJS сервисот и понатаму да продолжат да работат со новата имплементација без никакви измени во кодот.

И самиот AngularJS како платформа дефинира свои сервиси кои кодот на апликациите можи да ги користи за разни намени, на пример: за пристап до HTTP протоколот и испраќање на AJAX повици кон web серверот се користи HTTP сервис кој го имплементира ng.IHttpService интерфејсот и кој е дефиниран со стринг идентификатор “$http”, за работа со URL стрингови постои “$location” сервис, а за работа со прозорецот од интернет прегледувачот постои “$window” сервис итн. Сите овие идентификатори и интерфејси се добро познати и се опишани во официјалната документација и упатства за AngularJS.

Бидејќи DI идентификаторот е стринг кој се користи барем два пати (еднаш при дефинирање на DI сервисот и барем еднаш при негово инјектирање во DI клиентот), во LearnByPractice (LBP) се користи помошна класа која ги **централизира** сите стринг идентификатори за сервисите дефинирани во апликацијата. Класата се наоѓа во /wwwroot/app/injectionids.ts. DI идентификаторите се стрингови кои обично имаат внатрешна структура во облик на namespace-и, но можи да бидат и во било кој друг облик со или без внатрешна хиеархија и/или точки (се’ додека вредностите се единствени). Пример за вакви идентификатори се AngularJS сервисите, чии имиња почнуваат со знакот за долар ($) и содржат едноставно име на сервисот, без точки и хиерархија.

Со сето погоре кажано, вообичаено е во LBP при дефинирање на AngularJS сервис или AngularJS контролер **прво** да се определи / измисли неговиот DI идентификатор, со додавање на ново read-only property во класата InjectionIds, на пример:

public static get organizacijaService(): string {

return "app.services.organizacijaService";

}

Или

public static get admin\_organizacii\_indexController(): string {

return "app.admin.organizacii.indexController";

}

Потоа, идентификаторот се користи при дефинирање / регистрирање на AngularJS сервисот или AngularJS контролерот (после дефинирањето на соодветната класа):

angular

.module("app.services")

.service(app.InjectionIds.organizacijaService, OrganizacijaService);

Или

angular.module("app.admin")

.controller(app.InjectionIds.admin\_organizacii\_indexController, IndexController);

На крајот, идентификаторот се користи при дефинирање на зависностите кои треба да се инјектираат во DI клиентите. На пример, ако app.admin.organizacii.IndexController-от користи услуги од app.services.organizational.OrganizacijaService, врската се воспоставува во $inject полето на **контролерот**:

static $inject = [

app.InjectionIds.organizacijaService,

"$scope"

];

(да забележиме дека тука се користи и AngularJS сервисот “$scope”).

Соодветниот конструктор го има истиот број, редослед и тип на параметри:

constructor(

private organizacijaService: app.services.organizational.IOrganizacijaService,

private $scope: ng.IScope) {

super();

this.vchitajOrganizacii();

}

Аналогно на ова, при дефинирањето на навигацијата низ апликацијата преку ткн. *состојби* (ui-state) треба да се воспостави **врска** помеѓу HTML template и соодветниот на него контролер. Врската се воспоставува во /wwwroot/app/app.ts за AngularJS платформата да знае како да **инјектира** контролер за web страната:

let organizaciiConfig: ng.ui.IState = {

name: "admin.organizacii",

url: "/organizacii",

views: {

"@": {

templateUrl: "/wwwroot/app/admin/organizacii/index.html",

controller: app.InjectionIds.admin\_organizacii\_indexController,

controllerAs: "vm"

}

}

};

$stateProvider.state("admin.organizacii", organizaciiConfig);

Напомена: во LBP сите AngularJS сервиси се наследени од основна класа app.services.base.ServiceBase. Бидејќи AngularJS сервисите обично ќе се користат за комуникација со web серверот преку HTTP, сите тие внатрешно ќе го користат AngularJS $http сервисот. За стандардизирање на имплементацијата препорачливо е референца кон овој сервис да постои во основната класа за AngularJS сервиси во LBP. Доколку оваа референца се постави во $inject и во конструкторот како DI зависност, ќе треба сите наследени класи за AngularJS сервиси во LBP да го направат истото (да ја додадат зависноста во $inject и да ја очекуваат инстанцата на сервисот во конструкторот). Моментално, ограничувањата на TypeScript како јазик не даваат можност од изведената класа да се повика конструкторот на основната класа, при што сите зависности би се инјектирале автоматски. Поради ова, ако $http AngularJS сервисот би се додал како DI зависност во основната класа за сервиси во LBP и во сите наследени класи, практично кодот би се зголемил и неговото одржување би станало посложено наместо поедноставно. За да се избегни ова, $http се инјектира **преку код** (со користење на AngularJS методи на пониско ниво) во конструкторот на основната AngularJS сервис класа во LBP и е достапен за сите наследени класи како protected променлива.

protected $http: ng.IHttpService;

constructor() {

let $injector: ng.auto.IInjectorService = angular.injector(["ng"]);

this.$http = $injector.get<ng.IHttpService>("**$http**");

}

Со користење на ваков пристап, конструкторите на наследените LBP AngularJS сервиси се поедноставни, но сите тие го повикуваат наследениот конструктор (со super(); како прва наредба), со што се постигнува правилна иницијализација и инјектирање на потребниот $http сервис.

# Bundles and Bundle Configuration

*Bundle* е група / пакет на CSS, JavaScript и слични датотеки, кои се користат од страна на web страните, a кои интернет прегледувачот можи да ги побара од серверот во само еден HTTP request. Целта на воведувањето на bundle-вите во ASP.NET 4.5 беше да се намали HTTP сообраќајот кој се врши меѓу интернет прегледувачот и web серверот. Ако пребарувачот треба да преземи повеќе датотеки, треба да испрати исто толку одделни HTTP барања и да прими исто толку HTTP одговори. Со користењето на bundle истото се прави во едно барање / одговор (под претпоставка дека апликацијата е објавена на web серверот во ткн. release режим)..

Групирањето на датотеки во bundle (ткн. bundling) оди заедно со намалувањето (minification) на нивната големина. Ова обично се користи во ткн. release режим (кога се отстранети сите логички грешки и се подготвува пакет за испорака на web апликацијата на целниот сервер). Идејата е, со скратување на имињата на променливите и функциите во JavaScript датотеките (најчесто на само една буква) да се намали нивната физичката големина.

Напомена: моментално во LBP не се користи minification.

Поилустративен вовед во bundling & minification и предностите кои ги носи нивната употреба на официјалната страна:

<https://docs.microsoft.com/en-us/aspnet/mvc/overview/performance/bundling-and-minification>

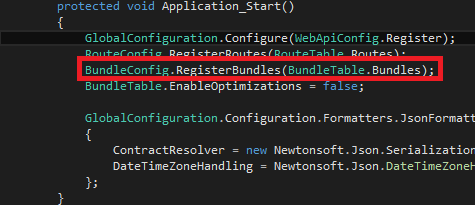
Bundle-овие обично се **дефинираат** во App\_Start\BundleConfig.cs, во методот RegisterBundles. LBP дефинира неколку bundle-ови, чии имиња би требало да ја одразуваат нивната намена:

* ~/bundles/css - CSS дадотеки со стилови кои се користат од страна на сите други bundle-ови. **Напомена**: редоследот на датотеките е важен, затоа Site.css е наведен последен (за да има предност во однос на другите .css датотеки кога се дефинираат стилови за ист CSS селектор).
* ~/bundles/modernizr - JavaScript библиотека за утврдување дали интернет прегледувачот има поддршка за одредени можности и својства (обично HTML5 и CSS3) и прилагодување на однесувањето за постари верзии на интернет прегледувачи.
* ~/bundles/scripts - сите готови (надворешни) JavaScript библиотеки кои се користат од страна на LBP web апликацијата. Редоследот е и тоа како важен, особено ако се има предвид дека некои библиотеки користат други библиотеки. На пример jQuery е наведена прва бидејќи се користи од страна на bootstrap.js, а таа, пак, се користи од страна на angular.js.
* ~/bundles/app - ова се скриптите за LBP web апликацијата. Поради нивниот значително поголем број, поделени се во групи. Редоследот на групите е важен, како што е важен и редоследот на скриптите во рамки на секоја од групите. Постојат неколку правила кои треба да се почитуваат:
  + Во Visual Studio се работи со TypeScript (.**t**s) датотеки, но во bundle-от се наведуваат JavaScript (.**j**s) датотеките со исто **основно** име (кои се генерираат динамички).
  + Секоја група започнува со скрипта која го дефинира модулот во кој ќе се наоѓаат останатите скрипти во таа група.
  + На почетокот се наведуваат скрипти за основниот app модул.
  + Понезависните скрипти се наведуваат порано, а позависните подоцна, за да се постигни логичка поврзаност. На пример, ако некој интерфејс или сервис во својата декларација и дефиниција содржи некоја класа, логично е дека таа класа треба претходно да е позната / дефинирана. Логично, скриптата за класата се наведува **пред** од скриптата за сервисот.

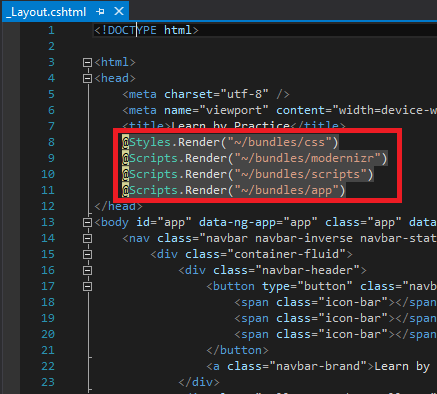
Одделните датотеки се додаваат во bundle-овите како стрингови. Очигледно е дека патеките до имениците во кои има повеќе датотеки ќе се повторуваат при дефиницијата bundle-от (бидејќи обично датотеки од еден вид се сместени во еден именик). За олеснување на конфигурирањето на bundle-овите и намалување на кодот кој се повторува, дефинирани се неколку помошни методи за пресметување на патеки:

* LibFile - пресметува пат до датотека од надворешна библиотека.
* AppFile - пресметува пат до датотека од LBP апликацијата.
* ModelFile - пресметува пат до датотека од TypeScript моделот на податоци.
* ServiceFile - пресметува пат до датотека со TypeScript сервис.

Конфигурацијата на bundle-овите се повикува при стартување на апликацијата, во Global.asax.cs:



Bundle-овите се вклучуваат во HTML markup-от во Views\Shared\\_Layout.cshtml со повикување на методот Render:



Додека редоследот на **дефинирање** на одделните bundle-ови не беше важен, редоследот во кој тие го **испраќаат својот излез** во генерираниот HTML (ткн. **rendering**) е важен. Бидејќи bootstrap користи посебни CSS-и, логично е прво да се наведи ~/bundles/css budnle-от. Понатаму, бидејќи скриптите за LBP апликацијата зависат од скриптите за надворешните библиотеки, логично е ~/bundles/app bundle-от да го испрати својот излез пред ~/bundles/scripts bundle-от. Како краен резултат, листата со bundle-ови и кодот внатре во секој од нив треба да се вчита во интернет прегледувачот без да предизвика JavaScript грешки поради неточен редослед на bundle-овите или скриптите.

# AngularJS модел

Овој модел е **еквивалентен** на Domain Object Model-от имплементиран во Domain.csproj проектот. AngularJS моделот е имплементиран како TypeScript класи во именикот /wwwroot/app/model. Структурата на подименици треба да одговара на соодветната структура во C# проектот.

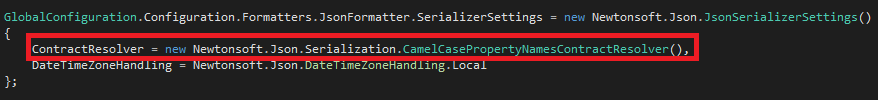
|  |  |
| --- | --- |
|  |  |

На сликата AngularJS моделот не е потполн (допрва треба да се додаваат именици и класи). Колекции **засега** не се потребни, а за пренос на групи од објекти од ист тип се користат полиња (низи).

**Напомена**: Во Java (не JavaScript) светот под терминот domain model се подразбира она што во LBP е имплементирано во BLL проектот (правила за деловна логика). Објектен модел за пренос на податоци помеѓу слоеви во Java светот обично се нарекува DTO (Data Transfer Objects).

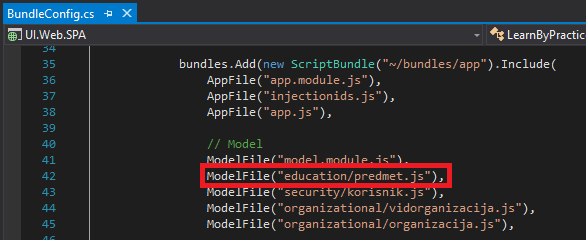
При имплементирање на AngularJS модел класа која одговара на соодветната C# класа од Domain.csproj проектот треба да се води сметка за неколку работи:

1. Хиерархиската структура на именици да одговара на структурата во C# проектот (ова не е задолжително, туку едноставно е договор за полесно следење и наоѓање на соодветните класи)
2. За најлесно (автоматско) серијализирање и десеријализирање на објекти при пренос меѓу JavaScript кодот (кој се извршува во интернет прегледувач) и WebAPI контролерот (на страната на web серверот), потребно е **соодветните својства да имаат исти имиња, со тоа што кај TypeScript класата првата буква треба да е мала**. Овој автоматизам е поставен во кодот во Global.asax.cs кој се извршува при стартување на web апликацијата:

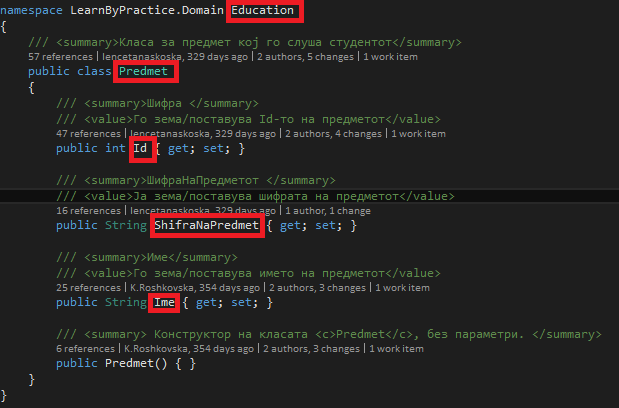


(стручниот назив кој се користи за стандардот за мали и големи букви во JSON својства е camel-case затоа што **i**dentifikatorot**I**menuvan**S**pored**T**oj**S**tandard личи на камила гледана во профил)

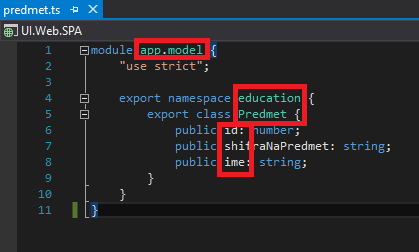
1. Модел класите припаѓаат на app.model модулот, а внатрешната хиерархија на namespace-и одговара на онаа во C# проектот (со тоа што имињата на namespace-ите се со мала почетна буква). И овде се работи за договор за поедноставно следење на кодот, а не за правило кое мора да се почитува.
2. Името на класата во TypeScript не мора да соодвествува на името на класата во Domain.csproj, но за полесно следење се користи истото име, со почетна голема буква).
3. На секое **својство** (property во C# класите) му одговара јавна (public) **променлива** во TypeScript модел класата.
4. Ако TypeScript класата треба да постави почетни вредности за некои од променливите, тоа се прави или во конструкторот или во истиот ред каде што се декларира променливата.
5. Во TypeScript засега постои само еден тип на податок number кој се користи за сместување на сите типови на бројни вредности (целобројни и децимални). Тука се и другите најчесто користени типови како boolean, Date (со големо D) и string.
6. Ако се додава **нова** .ts датотека за нова AngularJS модел класа, истата треба да се додаде во bundle-от на точното место (во групата за Model, но **после** model.module.js, бидејќи за додавање на било каква класа во некој модул, истиот мора претходно да е дефиниран):



На пример, ако класата за предмет во C# изгледа вака:



соодветната AngularJS модел класа во TypeSsript би изгледала вака:

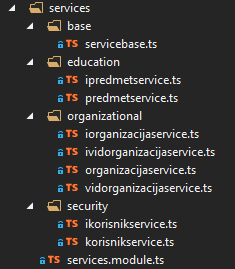


# AngularJS сервиси

Сервисите во AngularJS платформата се начин на делење на заеднички код (а понекогаш и податоци) помеѓу web страните на една AngularJS апликација.

Важно е да се знае дека AngularJS сервисите се ткн. **singleton** објекти, т.е. во даден момент постои најмногу еден објект од тој тип (класа) во апликацијата која се извршува во интернет прегледувачот (browser-от). Тоа значи дека ако повеќе web страни (преку своите контролери) користат еден ист сервис (референциран со неговиот DI идентификатор), AngularJS платформата ќе создаде само една инстанца (само еден објект) од тој сервис (при првото користење) а потоа истата инстанца ќе ја употребува во сите следни повикувања на сервисот.

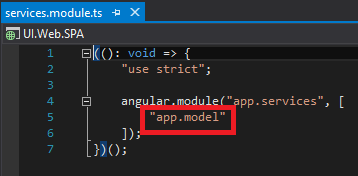
Сервисите се имплементирани како TypeScript класи во именикот /wwwroot/app/services.



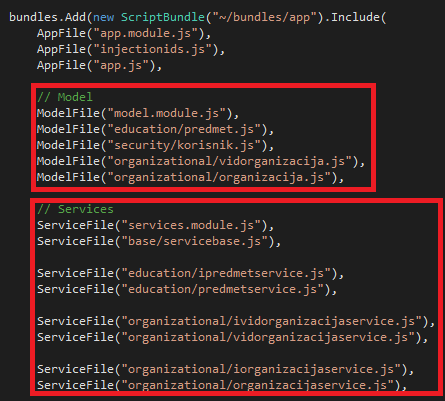
На сликата е прикажана развојна верзија на кодот, кој не ги содржи сите потребни сервиси.

Се забележува следната структура:

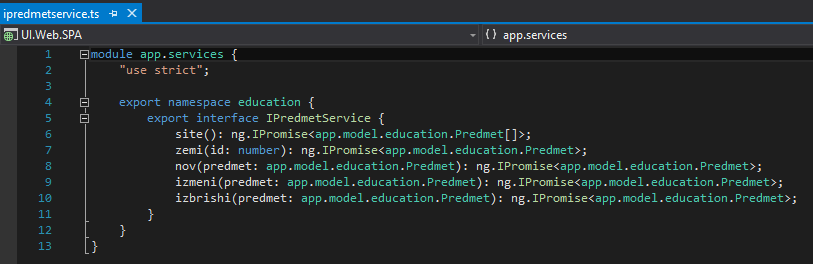
* Датотеката services.module.ts го дефинира задолжителниот модул app.services во кој ќе „живеат“ сервисите. Од содржината на датотеката се гледа дека модулот app.services **зависи** од модулот app.model (што е и логично, бидејќи сервисите испраќаат и примаат податоци преку објекти од моделот).:



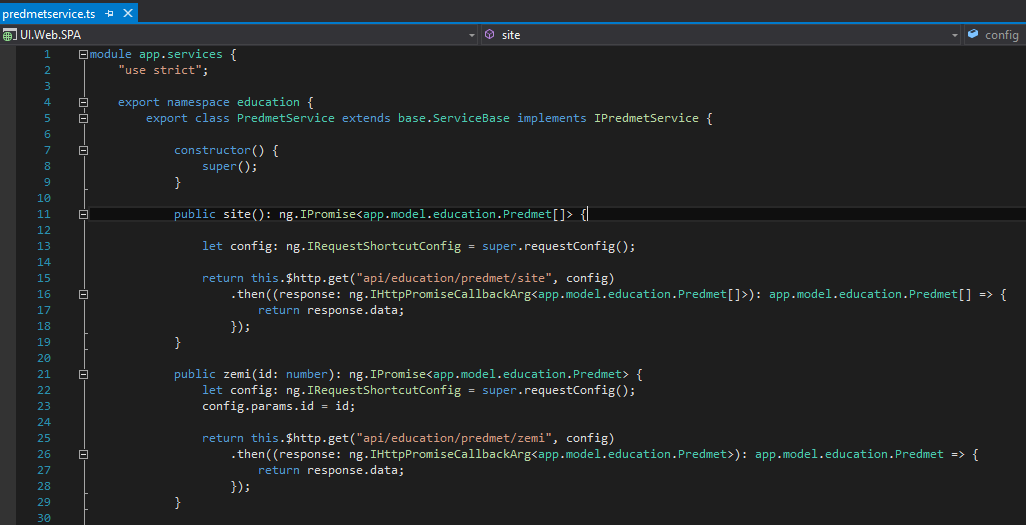
Оваа логичка поврзаност и меѓузависност се отсликува и во конфигурацијата на bundle-от ~/bundles/app (датотеките за моделот се наведени **пред** датотеките за сервисите):



* Датотеката servicebase.ts ја содржи основната класа од која се наследени сите останати сервиси во LBP апликацијата. Функционалности и помошни функции кои се заеднички за сите или барем за повеќето сервиси логично е да се имплементираат во основната класа.
* И сервисите се поделени во подименици (и соодветни namespace-и) според функционалните области на Domain, DAL и BLL проектите.
* За секој сервис постои соодветен интерфејс (кој ги **декларира** методите на сервисот)



додека самиот сервис (како класа) ги **дефинира** (имплементира) методите на интерфејсот:



* Дефиницијата на класата за сервисот е стандардна - сервисот се наследува од основната класа app.services.base.ServiceBase и го имплементира соодветниот интерфејс. На овој начин и формално при преведување на кодот се обезбедува дека класата за сервисот ќе ги имплементира сите методи декларирани во интерфејсот.
* Конструкторот на класата за сервисот го повикува наследениот конструктор со користење на резервираниот збор super();.

Во LBP сервисите најчесто ќе ги користиме за вршење на AJAX повици преку HTTP протокол, за размена на податоци со web серверот. **Најчесто** на секој метод во AngularJS сервис ќе му одговара соодветен метод во некоја WebAPI контролер класа (во подименикот Controllers).

Самата природа на AJAX подразбира дека повикот е **асинхрон**, т.е. откако ќе се испратат податоците преку HTTP, извршувањето на кодот продолжува без да се чека враќањето на одговор / резултатот. Но, сервисот најчесто се користи за да ни врати некакви податоци кои потоа ги обработуваме или прикажуваме на страната на клиентот (интернет прегледувачот). Поради ова, за асинхроните повици во AngularJS се користи механизмот на ткн. *ветувања* (promises). Даден метод не враќа веднаш резултат од одреден тип, туку **враќа ветување** дека ќе врати резултат од одреден тип. Одговорност на повикувачот е при повик на методот да наведе и **парче код** (најчесто во облик на inline функција напишана во истата наредба со повикот) кое ќе се изврши асинхроно од страна на повиканиот метод, **откако** методот ќе го има резултатот.

Секој метод кој врши HTTP AJAX повик кон web серверот има стандардна структура:

public imeNaMetodot(parametar: tip, …): ng.IPromise<TipNaRezultat> {

let config: ng.IRequestShortcutConfig = super.requestConfig();

// поставување на параметри на повикот (ако ги има)...

**return** this.$http.**glagol**("**url**", config)

.then((response: ng.IHttpPromiseCallbackArg<TipNaRezultat>): TipNaRezultat => {

return response.data;

});

}

Значи, методот на сервисот **враќа ветување** дека ќе врати одреден резултат, бидејќи и $http сервисот кој се користи внатре во самиот метод исто така **враќа ветување** дека ќе врати нешто. Ако еден метод во синџирот враќа ветување наместо резултат, тогаш и сите методи кои го повикуваат исто така ќе мора да вратат ветување.

TipNaRezultat го означува типот на резултат кој треба да го очекува повикувачот. Можи да биди поединечен објект (на пример, app.model.education.Predmet) или низа (поле) од објекти (на пример, app.model.education.Predmet[]).

Наредбата:

let config: ng.IRequestShortcutConfig = super.requestConfig();

е стандардизиран начин на создавање на објект преку кој се конфигурира HTTP повикот. Објектот config има подобјект params во кој се додаваат параметрите кои ќе се пренесат преку URL (имињата може да бидат произволни, но треба да одговараат на имињата на параметрите на соодветниот метод на WebAPI контролерот на страната на web серверот). Една од работите кои се дел од конфигурацијата се и параметрите кои се пренесуваат преку URL и кои се користат најчесто при HTTP GET повици. На пример, кај методите со име zemi чија работа е да земат поединечен објект според неговиот интерен идентификатор, преку конфигурација се пренесува токму вредноста на интерниот идентификатор:

config.params.id = id;

Со this.$https се пристапува до претходно подготвената инстанца на AngularJS $http сервисот, кој е инјектиран преку код во основната класа (подетално за ова е кажано во делот за Dependency Injection).

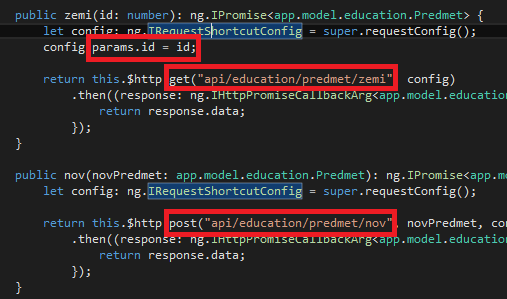
**glagol**-от е метод на $http сервисот, кој одговара на некој од HTTP **глаголите (методите)**. Најчесто за AJAX повици во LBP кои не пренесуваат параметри или пренесуваат едноставни вредности како id, се користи методот get (кој одговара на HTTP глаголот GET). За повици кои кон web серверот испраќаат цели објекти (како на пример, при додавање, измена и бришење на податоци) во LBP ќе се користи методот post, кој одговара на HTTP глаголот POST. Во таков случај објектите се пренесуваат **во самиот повик**, а не преку config.params:

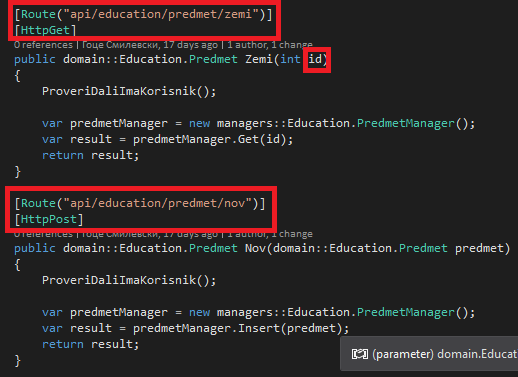
return this.$http.post("url", **objekt1, objekt2**, config)...

url одговара на URL-то зададено во Route атрибутот на соодветниот метод во соодветниот WebAPI контролер.

**Напомена**: за да можи HTTP AJAX повикот да го најди соодветниот WebAPI контролер и да го повика соодветниот метод, потребно е:

1. url параметарот наведен при повикот на $http.get / $http.post методот да одговара на вредноста на Route атрибутот на C# методот во WebAPI контролер класата. По договор, обликот на URL-то при CRUD операции со објекти е api/*oblast*/*objekt*/*operacija*.
2. името на $http методот (**get** или **post**) да одговара на Http**Get** или Http**Post** атрибутот на C# методот во WebAPI контролерот.
3. кај get повици името на параметарот на C# методот треба да одговара на името на својството во config.params





$http.get/post методот **враќа ветување** дека ќе вратиTipNaRezultat (преку ng.IHttpPromise<T> што е посебен вид на ng.IPromise кој се користи при HTTP повици). Како и со секој ng.IPromise, повикувачот на $http.get/post методот (т.е. нашиот AngularJS сервис) треба да наведи кој код ќе се изврши откако ќе заврши HTTP повикот. IHttpPromiseCallbackArg<T> е само **обвивка** околу очекуваниот резултат, која исто така дава дополнителни информации за статусот на HTTP повикот и за евентуалните грешки вратени од серверот. Својството data го содржи самиот резултат од тип T, кое (засега) треба едноставно да се врати на повикувачот на методот од сервисот.

return response.data;

Во овој случај, reponse.data содржи објект (или низа од објекти) од некоја од класите во AngularJS моделот кој одговара на соодветната класа од Domain проектот, и кој преку JSON е пренесен од серверот на клиентот.

**Забелешка**: правилната имплементација на методот во AngularJS сервисот би требало да содржи и проверка на статусот и некаква обработка на грешки, но тоа треба да биди поддржано со соодветна обработка на грешки на страната на web серверот.

Врз основа на досега кажаното, постапката за имплементирање на нов AngularJS сервис се состои од следните чекори:

1. Се определува во која област / namespace ќе се додава сервисот.
2. Се декларира соодветен интерфејс (во датотека со име i*nekojservice*.ts) и имплементација (во датотека со име *nekojservice*.ts). Класата за сервисот се наследува од основната класа за сервиси и го имплементира новиот интерефејс.
3. Новите датотеки се додаваат на соодветното место во ~/bundles/app bundle-от.
4. Се декларираат методите на сервисот (ако некоја класа од моделот недостасува и треба да се додаде, истото се прави според претходно опишаната постапка во делот за AngularJS модел).
5. Се имплементираат методите на сервисот со соодветните $http повици и со стандардната структура опишана погоре (најчесто се имплементираат методи со стандардни имиња:
   1. site() - за земање на листа од сите објекти:
   2. zemi(id) - за земање на поединечен објект по интерен идентифкатор;
   3. nov(T) - за додавање на нов објект од тип T
   4. izmeni(T) - за измена на податоци за постоечки објект од тип T
   5. izbrishi(T) - за бришење на податоци за постоечки објект од тип T.
6. Се определува DI идентификатор за сервисот и се додава соодветно read-only property во injectionids.ts според кое ќе се референцира сервисот за DI.
7. После дефиницијата на класата за сервисот, во истата датотеката се врши и регистрирање на сервисот, за да можи истиот да се инјектира во класите кои ќе го користат, на пример:

angular

.module("app.services")

.service(app.InjectionIds.**predmetService**, **PredmetService**);

За тестирање на функционалноста на AngularJS сервис неопходно е да е имплементиран WebAPI контролер на страната на web серверот.

# WebAPI контролери

Преку WebAPI контролерите деловната логика имплементирана во BLL проектот се прави достапна до презентацискиот слој (web страните имплементирани во HTML5 / CSS кои користат AngularJS и Bootstrap). Практично контролерите играат улога на **web сервиси** кои AngularJS кодот во интернет прегледувачот ги повикува за да чита и запишува податоци. Јадрото на деловната логика останува имплементирано во BLL слојот, па така WebAPI контролерите претставуваат само **танок** слој околу BLL.

Контролерите се имплементирани како класи во подименикот Controllers. Се забележува истата структура на подименици за функционалните области како и кај неколку други проекти.

Класа ControllerBase служи како основа од која обично се изведуваат останатите WebAPI контролери. Таа содржи и некои помошни методи, кои останатите контролери ги користат како готови фукнционалности, на пример при проверка дали некој корисник е претходно најавен. Ваквата проверка е неопходна за да се спречи анонимен пристап до функционалностите на апликацијата (на пример, од страна на нечиј туѓ код преку интернет, на ист начин на кој тоа го прави AngularJS web апликацијата).

**Забелешка**: Класата HomeController се разликува од останатите класи за контролери, и таа служи само за да ја вчита **единствената** MVC страна /Views/Home/Index.cshtml. Таа страна како свој layout ја користи страната /Views/Shared/\_Layout.cshtml, која го дефинира севкупниот стандарден изглед на single-page апликацијата.

Обично, еден WebAPI контролер одговара на еден AngularJS сервис во LBP апликацијата. Секој метод на контролерот одговара на еден метод на AngularJS сервисот. За да се направи методот достапен за повикување преку WebAPI, тој мора да е јавен (public) и да е „украсен“ со два атрибути;

* Route[(“url”)] - ја дефинира патеката (URL-от) на кој ќе „слуша“ методот.
* HttpGet или HttpPost - го определува HTTP **глаголот** (методот) кој ќе се користи при повикување на методот на контролерот.

Како што е претходно опишано во делот за AngularJS сервиси, атрибутите и параметрите на методот на WebAPI контролер мора да се усогласени со кодот на AngularJS сервисот кој ќе го повикува тој метод.

Најчесто, првата наредба на методот во контролерот врши проверка дали повикот се извршува од страна на претходно најавен корисник:

ProveriDaliImaKorisnik();

Автентикацијата во LBP моментално се врши преку механизмот на ASP.NET наречен Forms Authentication. После успешното пријавување, web серверот испраќа колаче (cookie) кон интернет прегледувачот. Тоа колаче се испраќа назад до серверот при секој HTTP повик. Со проверка на постоењето и валидноста на ова колаче се проверува дали методите на WebAPI контролерот се вршат од страна на претходно најавен корисник.

Понатаму, **обично** методот на WebAPI контролерот повикува некоја од manager класите од BLL проектот за да ја изврши бараната операција. Резултатите во облик на објекти од domain object model-от се враќаат преку HTTP во JSON формат и на страната на интернет прегледувачот се реконструираат како објекти од AngularJS моделот (како шо е опишано во претходните делови).

Ако од методот на контролерот треба да се вратат повеќе објекти од ист тип, како тип на резултат на тој метод се користи IEnumerable<T>, на пример:

[Route("api/education/predmet/site")]

[HttpGet]

public **IEnumerable**<domain::Education.Predmet> Site()

{

ProveriDaliImaKorisnik();

var predmetiManager = new managers::Education.PredmetManager();

var result = predmetiManager.GetAll();

return result;

}

Внатре во методот се работи со стандардните класи за колекции од објекти од Domain проектот, кои го имплементираат IEnumerable<T> интерфејсот. Како последица на ова, не се потребни никакви претворања на типови, едноставно се враќа колекцијата од доменски објекти. На страната на интернет прегледувачот, во рамки на AngularJS сервисот, резултатот се враќа како низа (поле) од објекти од AngularJS моделот.

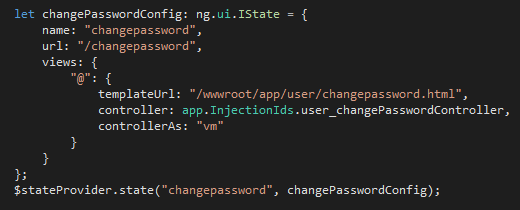
Како основна функционалност, секој WebAPI контролер кој работи со некој вид на доменски објект најчесто и имплементира основните функционалности преку следните методи:

* Site() - враќа колекција од сите познати објекти од тој вид.
* Zemi(int id) - враќа поединечен објект со дадениот интерен идентификатор.
* Nov(T objekt) или Nova(T objekt) - создава нов објект од дадените податоци.
* Izmeni(T objekt) - ги менува податоците за дадениот објект
* Izbrishi(T objekt) ги бриши податоците за наведениот објект.

Се разбира, зависно од потребите на AngularJS апликацијата можни се и други методи и функционалности кои ќе се разликуваат од основите CRUD операции.

# AngularJS web страни и контролери

Преку web страните корисниците пристапуваат до функционалностите на web апликацијата. Од гледна точка на навигацијата (движењето) низ апликацијата, секоја web страна одговара на една состојба (UI state). Состојбата се дефинира во делот од кодот кој ја конфигурира апликацијата при нејзиното стартување, а се наоѓа во датотеката /wwwroot/app/app.ts:

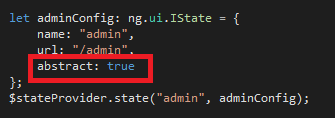


Значењето на својствата за конфигурација е следно:

* name - името на состојбата.
* url - адреса која се прикажува во интернет прегледувачот.
* views - дефинира еден или повеќе „погледи“ (начини на гледање) на состојбата. Секој поглед се идентификува со име (со “@” се означува погледот кој се подразбира) и ги има следните својства:
  + templateUrl - адреса до .html template-от кој го содржи markup-от за погледот.
  + controller - DI идентификатор на контролерот кој ќе се поврзи со погледот (дефиниран како и сите слични DI идентификатори, во /wwwroot/app/injectionids.ts).
  + controllerAs - алиас (друго име) кое се користи во рамки на .html template-от кога треба во HTML markup-от да се вградат вредности од контролерот (на пример, вредности на некои својства). Најчесто како алиас се користи “vm”.

Состојбата се регистрира со повикување на медотот .state(), при што првиот параметар е името (исто така дадено погоре) и самиот конфигурациски објект.

Постојат и ткн. *апстрактни* состојби, кои немаат видливо претставување и служат само за логичка поделба на состојбите по области:



Состојбите можи и да се надоврзуваат една на друга, за што се користи име со хиерархиска структура (со точки, како namespace).



Овде состојбата “admin.predmeti” се надоврзува после состојбата “admin” и соодветно на тоа целиот URL на оваа состојба ќе биде /admin/predmeti (но поради хиерархиската структура се наведува само “predmeti”.

Слично, состојбата “admin.predmeti.vnesuvanje” се надоврзува после состојбата “admin.predmeti” и очекува параметар во URL-от (на пример, URL во облик: “/admin/predmeti/vnesuvanje/5” се користи за отворање на страната за внесување на предмети и се вчитува предметот со интерен идентификатор 5).

Повеќе детали за логиката и синтаксата која се користи при дефинирање на состојбите во AngularJS UI Router на следните адреси:

<https://scotch.io/tutorials/angular-routing-using-ui-router>

<https://www.sitepoint.com/write-modular-code-angular-ui-router-named-views/>

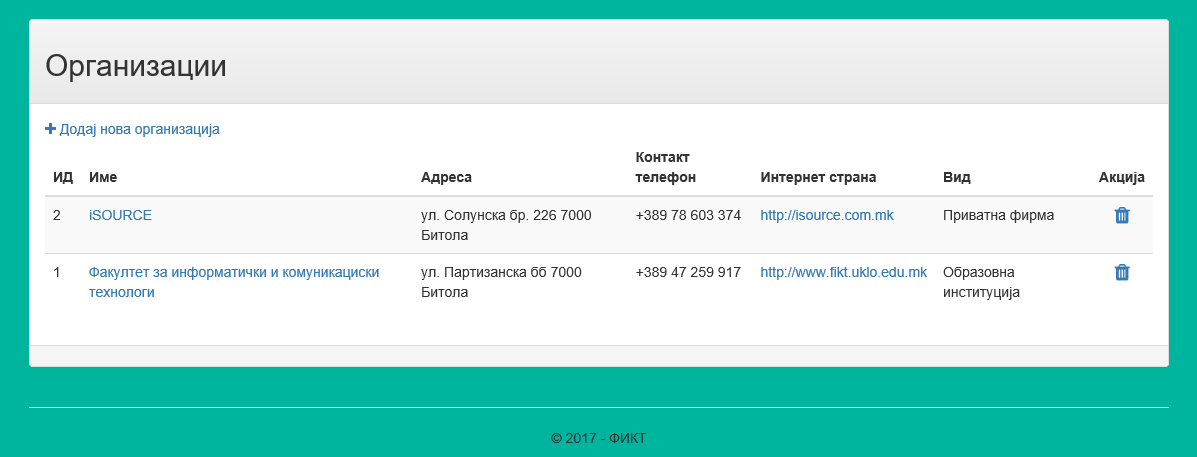
<https://github.com/angular-ui/ui-router/wiki/Multiple-Named-Views>

Од гледна точка на имплементација, секоја web страна се состои од *web template* и *контролер*. Web template-от е .html датотека која содржи HTML 5 markup и во која се прикажуваат вредности на својства од контролерот со користење на AngularJS синтаксата {{ vm.nekoeSvojstvo }}. Контролерот е AngularJS класа која имплементира својства и методи кои се користат во web template-от (својствата се користат во AngularJS изрази ли се вградуваат во HTML-от, а методите се повикуваат од страна на JavaScript event handler-и (како на пример, при кликнување на некое копче од web template-от).

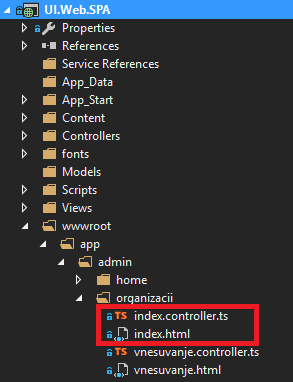
Во продолжение ќе бидат опишани двата најчести видови на web страни во LBP, а по истата логика или со комбинирање на работи од овие два вида можи да се имплементира и некој друг вид на страна.

## Web страна за прикажување на листа од објекти

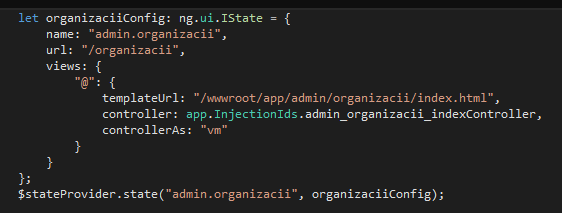
Овој вид на web страна се користи за листање на сите објекти од одреден вид, најчесто во администраторскиот дел. Вообичаено изгледот на ваквите страни е следниот:



Имплементацијата на страната се состои од .html template и соодветен AngularJS контролер.

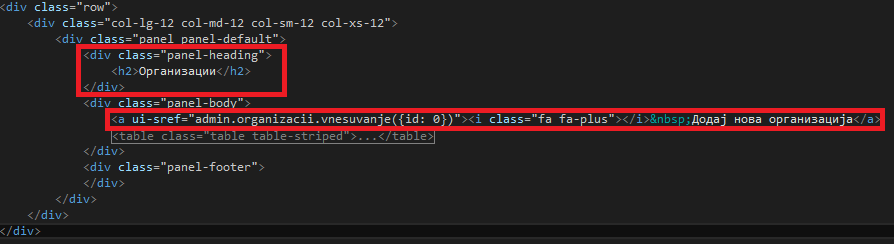


Врската помеѓу .html template-от и AngularJS контролерот се воспоставува преку ткн. состојба, која се конфигурира во app.ts:

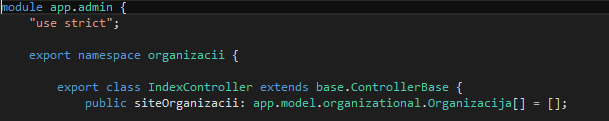


HTML template-от (обично именуван index.html) ги содржи визуелните елементи на **централниот** дел на web страната:

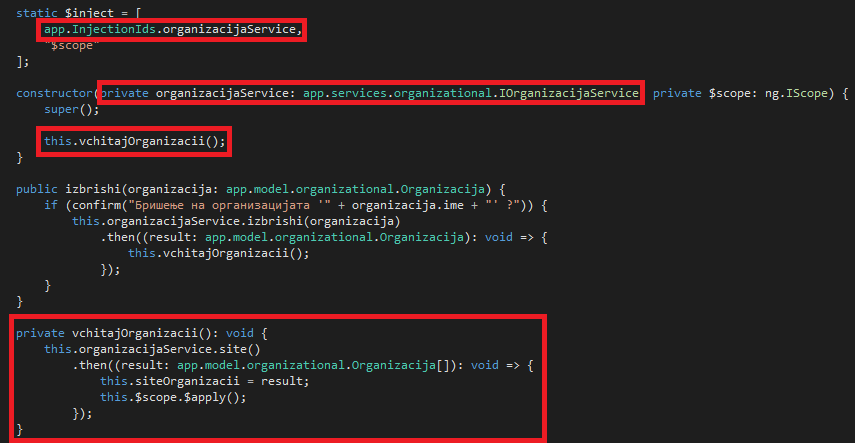
* Целата содржина обично е содржана во div со CSS класи panel и panel-default.
* Во насловот на панелот се содржи насловот на страната (во примерот тоа е текстот *Организации*).
* Под насловот обично се наоѓа хиперлинк со знак “+” и текст кој означува додавање на нов објект од соодветниот вид. Со кликање на хиперлинкот се преминува на нова web страна (опишана во следниот дел од упатството) преку која се внесуваат податоци за нов објект (во примерот, за нова организација). Функционалноста на овој хиперлинк се постигнува со атрибутот ui-sref кој ја содржи вредноста на состојбата кон која треба да се премини (во случајот тоа е admin.organizacii.vnesuvanje), при што се пренесува 0 (нула) како параметар, со што се означува дека се работи за внесување на нов запис.



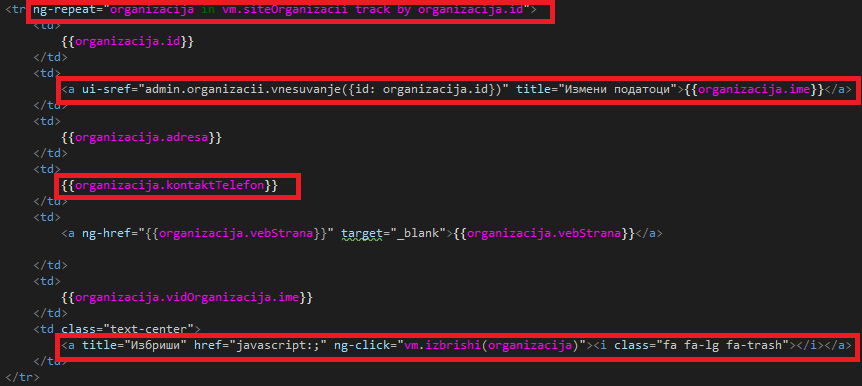
* Понатаму следи табела во која се прикажани податоците за сите постоечки објекти од овој вид (по една редица за секој објект). AngularJS контролерот придружен на web страната обично има својство кое ги содржи сите објекти од еден вид, вчитани во низа (поле, array) од објекти од моделот.



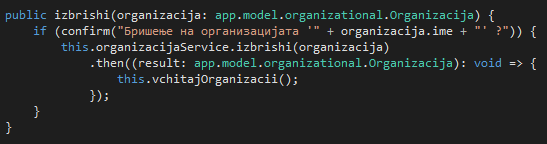
* Овие објекти се вчитуваат во AngularJS контролерот преку соодветен сервис, кој се повикува во конструкторот на контролерот:



* Со користење на AngularJS директивата ng-repeat за секој објект во полето се генерира по еден tr елемент (кој претставува редица во HTML табела) со вгнездени td елементи.
* Секоја колона во табелата прикажува еден атрибут за објектите (во примерот, име, телефон, Интернет страна итн.). Прикажувањето на вредностите на атрибутите се постигнува со користење на AngularJS синтаксата {{ objekt.svojstvo }} во секоја келија од табелата (претставена со по еден td елемент).
* Обично секој објект има име кое еднозначно го идентификува во листата со објекти. Името е прикажан како хиперлинк, со чие кликање се преминува на нова web страна, преку која корисникот ги менува податоците за избраниот објект. Функционалноста на овој хиперлинк се постигнува со атрибутот ui-sref кој ја содржи вредноста на состојбата кон која треба да се премини (во случајот тоа е admin.organizacii.vnesuvanje), при што како аргумент се пренесува идентификаторот на објектот чии податоци се уредуваат (за контролерот на web страната за внесување да можи да ги вчита податоците за тој објект врз основа на неговиот идентификатор).
* На крајот на секоја редица има икона со канта за отпадоци (имплементирана како хиперлинк), со чие кликање се бриши објектот прикажан во таа редица (потребно е корисникот да го потврди бришењето).



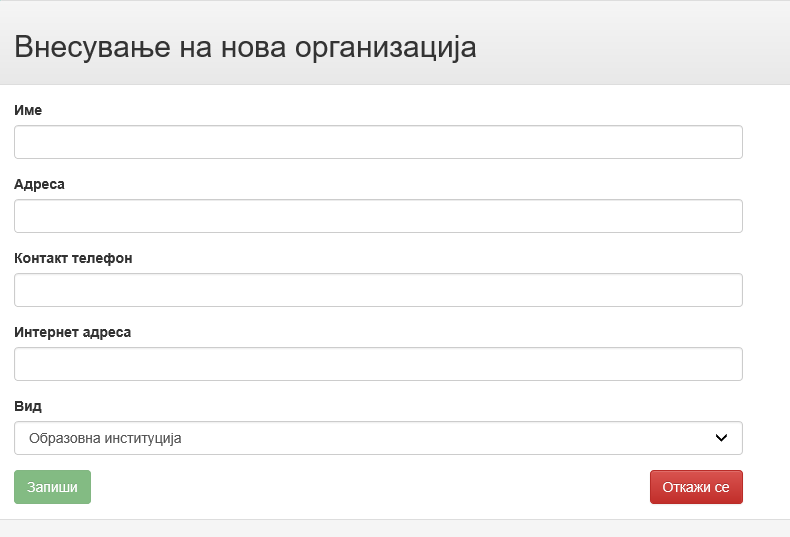
* Како event handler на click настанот на иконата за бришење е поставен метод од AngularJS контролерот, на кој му се проследува објектот кој ќе се бриши (во примерот тоа е повикот на методот во облик vm.izbrishi(organizacija)).
* Во кодот на методот се бара конечна потврда од корисникот, објектот се бриши со повик на соодветниот медот на AngularJS сервисот, а потоа податоците повторно се вчитуваат, за да се освежи приказот:



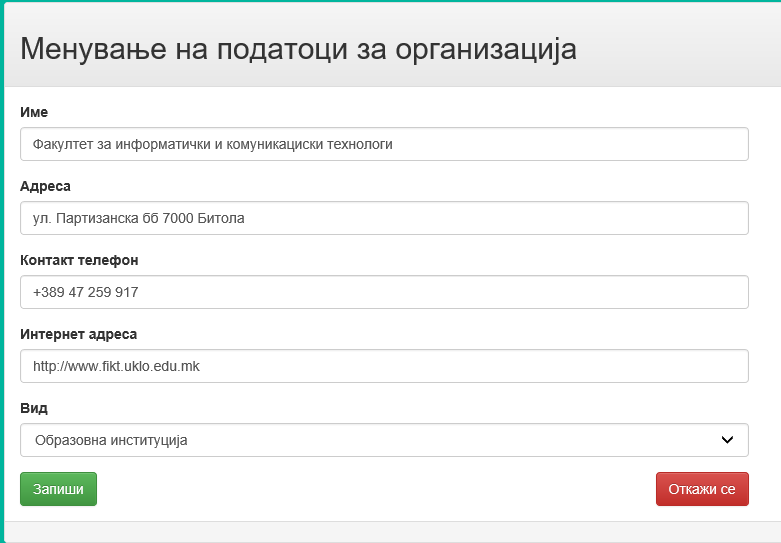
## Web страна за внесување на вредности за еден објект

Овој вид на web страна се користи за внесување на еден објект од одреден вид. Бидејќи податоците кои се внесуваат и при додавање на нов објект и при измена на постоечки објект се најчесто исти, истата web страна се користи за двете намени.

Една таква страна во режимот за додавање на нов објект обично го има следниот изглед (примерот е земен од страната за внесување на организација):

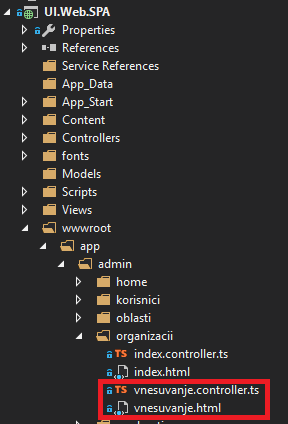


Во зависност од начинот на кој се користи страната се прикажува и соодветен наслов. При внесување на нови податоци најчесто полињата се празни или пополнети со нивните почетни вредности. При измена на постоечки податоци се прикажува соодветен наслов, а во полињата се вчитани постоечките вредности. Така, кога истата страна се користи за измена на податоци за постоечка организација, го има следниот изглед:

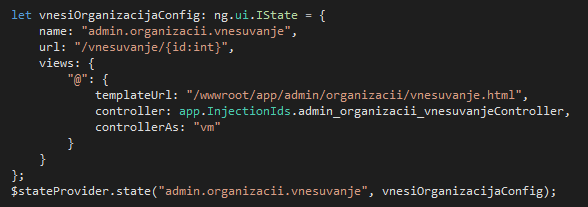


До оваа страна се стигнува преку страната која ги прикажува сите дефинирани организации. Во режим за внесување на нова организација се влегува кога ќе се кликни хиперлинкот со знакот “+“ под насловот на оваа страна. Во режим за измена на податоци за организација се влегува кога ќе се кликни на хиперлинкот со името на некоја организација во табелата.

Имплементацијата на страната се состои од .html template и соодветен AngularJS контролер:



Врската помеѓу .html template-от и AngularJS контролерот се воспоставува преку ткн. состојба, која се конфигурира во app.ts:



Бидејќи web страната треба да знае кој објект се внесува, воведен е параметар во URL-от (во случајот, id од целоброен тип) преку кој се пренесува интерниот идентификатор на објектот кој се внесува. При внесување на нов објект договорот е да се пренеси нула. AngularJS синтаксата која се користи е:

url: “/vnesuvanje/{id:int}”

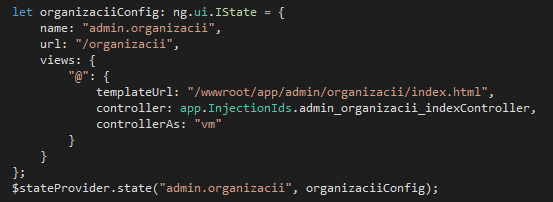
што значи дека во адресното поле на Интернет прегледувачот при внесување на нова организација ќе се прикажи:

<http://learnbypractice.fikt.edu.mk.local/#!/admin/organizacii/vnesuvanje/0>

додека при измена на некоја постоечка организација со интерен идентификатор 1 ќе се прикажи:

http://learnbypractice.fikt.edu.mk.local/#!/admin/organizacii/vnesuvanje/1

URL-от е **релативен** во однос на URL-от на состојбата-родител “admin.organizacii”, поради што се поставува само делот од URL-от (/vnesuvanje/{id:int}) кој доаѓа **после** URL-от за состојбата “admin.organizacii”.



URL-от за оваа состојба, пак, се конфигурира релативно во однос на нејзината родителска состојба “admin”:



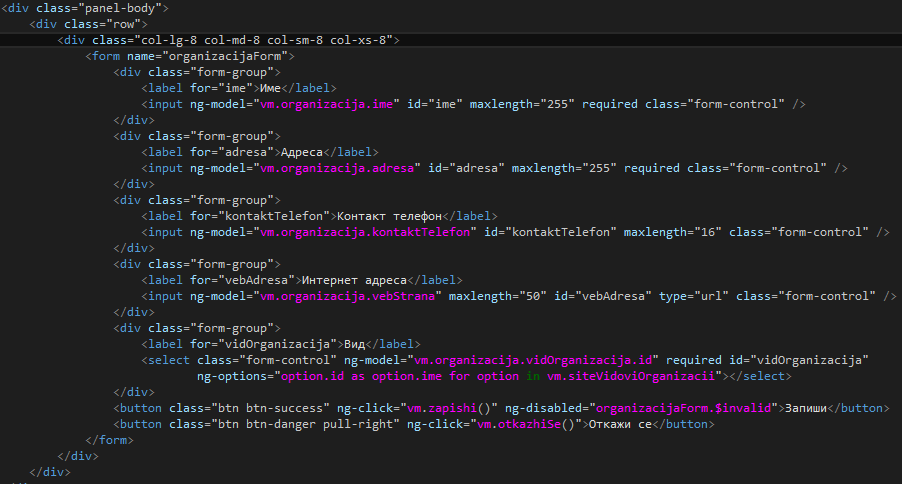
Со поврзување на овие три **релативни** URL-а се добива конечниот URL за состојбата за внесување на организација кој се прикажува во адресната линија.

HTML template-от (обично именуван vnesuvanje.html) ги содржи визуелните елементи на **централниот** дел на web страната:

* Целата содржина обично е содржана во div со CSS класи panel и panel-default.
* Во насловот на панелот се содржи насловот на страната, кој се прикажува во зависност од режимот во кој се користи страната (вредноста се пресметува со користење на ткн. *тернарен оператор* во облик **логички израз ? вредност за true : вредност за false**):



* Телото на панелот содржи форма која ги прикажува одделните полиња (по едно во секој ред), при што за се води сметка HTML контролата (input, check box, radio group, drop down итн.) да одговара на типот на податок за соодветното поле. На дното на формата се прикажани две копчиња со кои корисникот ги потврдува или поништува измените:



* Објектот за кој се внесуваат податоци е претставен во кодот на AngularJS контролерот со јавна променлива:

public organizacija: app.model.organizational.Organizacija = new app.model.organizational.Organizacija();

* Режимот во кој се користи web страната е определен преку две логички променливи:

public novZapis: boolean;

public izmena: boolean;

* Бидејќи секоја организација е поврзана со некој вид на организација, потребно е web страната да понуди листа со видови на организации од кои корисникот избира една. Таа листа се вчитува во поле од соодветниот тип на објекти:

public siteVidoviOrganizacii: app.model.organizational.VidOrganizacija[] = [];

* Потребните AngularJS сервиси се инјектираат на стандардниот начин:

static $inject = [

app.InjectionIds.organizacijaService,

app.InjectionIds.vidOrganizacijaService,

"$scope",

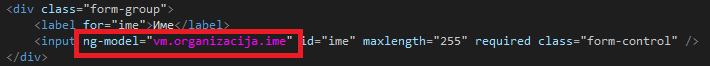
"$state",

"$stateParams"

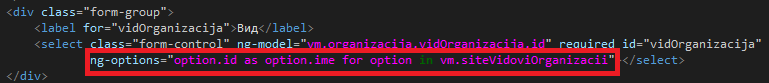
];

Сервисот кој работи со организации се користи при вчитување и запишување на податоците. Сервисот кој работи со видови на организации се користи за вчитување на листата со сите видови на организации дефинирани во системот. Ако објектот кој се внесува е поврзан со други видови на објекти, соодветните листи за избор ќе бидат имплементирани на истиот начин (како низи од објекти од моделот), а вчитувањето ќе се врши преку соодветните AngularJS сервиси кои ќе се инјектираат повторно на истиот начин преку статичкото поле $inject.

* Секое поле за внесување преку ng-bind директивата е поврзано со едно својство од објектот кој се внесува:



* HTML атрибутот **required** се користи како наједноставно средство за валидација на внесените податоци (корисникот не ќе можи да ги запиши податоците без да ги внеси сите задолжителни вредности)
* Внесувањето на видот на организација (како и секоја вредност која треба да се избере од select листа) бара малку повеќе конфигурација, бидејќи треба да се декларира кое поле од листата за избор треба да се земи како вредност, а кое како описен текст во select тагот. За оваа намена се користи AngularJS директивата ng-options



* На дното од формата се поставени копчињата *Запиши* и *Откажи се*, кои преку AngularJS директивата ng-click се поврзани со соодветните методи од AngularJS контролерот кој се извршува за оваа web страна:



* На горната слика се гледа дека со користење на AngularJS директивата ng-disabled копчето *Запиши* се **оневозможува** кога формата што ги опфаќа контролите за внесување е невалидна. На овој начин се оневозможува корисникот да се обиде да запиши непотполни податоци.
* Повеќе за валидацијата со користење на AngularJS и HTML5 на следните линкови:

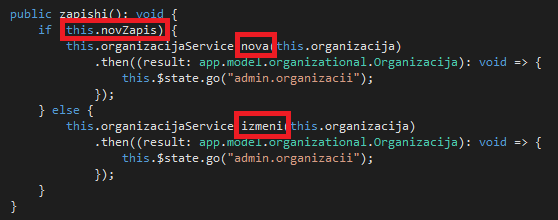
<https://docs.angularjs.org/guide/forms>

<https://www.w3schools.com/angular/angular_validation.asp>

* Конструкторот на контролерот го содржи кодот за иницијализација на web страната:

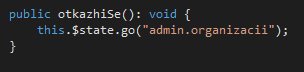


* Откако ќе се вчита листата со сите познати видови на организации се определува дали е проследен параметарот id преку URL-от за состојбата (во режим за менување на податоци се вчитува постоечкиот објект преку AngularJS сервисот).
* Во случај на нов запис се поставува почетна вредност за видот на организација.
* На крајот се поставуваат индикаторите кои се користат за определување на режимот во кој се користи web страната.
* Методот zapishi ги испраќа внесените податоци до web серверот, кој ги запишува во базата на податоци (во зависност од режимот се повикува соодветниот метод):



По запишувањето на податоците контролата се враќа на страната за листање на сите познати организации (со користење на $state сервисот за работа со AngularJS состојби)..

* Методот *otkazhiSe* го прави истиот премин без претходно запишување на податоците:



* Контролере регистриран за DI на вообичаениот начин:

