UNIVERZA LJUBLJANA, FAKULTETA ZA
RAČUNALNIŠTVO IN INFORMATIKO

Peter Stegnar

Razvoj spletnih aplikacij z vzorci MVC na strežniku in odjemalcu

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Janez Demšar

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00558 / 2013
Datum: 15.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **PETER STEGNAR**

Naslov: **RAZVOJ SPLETNIH APLIKACIJ Z VZORCI MVC NA STREŽNIKU IN ODJEMALCU**
DEVELOPMENT OF WEB APPLICATIONS USING THE MVC PATTERNS ON SERVER AND CLIENT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Moderne spletne aplikacije temeljijo na številnih tehnologijah, ki jih moramo znati pravilno povezati, če želimo aplikacijo, ki bo sodobna, odzivna, varna in jo bo zmožno vzdrževati in nadgrajevati. Trenutno popularni pristopi temeljijo na vzorcu model - pogled - kontroler (model - view - controller, MVC). Relativna novost je selitev tega vzorca s strežnika (tudi) na odjemalca, kar vodi v aplikacije na eni (spletni) strani (single page application, SPA). S tem lahko dosežemo večjo odzivnost aplikacije in, v splošnem, boljšo uporabniško izkušnjo. V praksi se trenutno najbolj obnesejo hibridne aplikacije, na katerih se procesiranje primerno razdeli na strežniško in odjemalsko stran. V diplomskem delu čimbolj celovito preglejte aktualne tehnologije s tega področja. Način njihove uporabe predstavite na primeru preproste spletne aplikacije, zgrajene v skladu z najboljšimi praksami s področja.

Mentor:
izr. prof. dr. Janez Demšar

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani **Peter Stegnar**, z vpisno številko **63020292**, sem avtor diplomskega dela z naslovom:

Razvoj spletnih aplikacij z vzorci MVC na strežniku in odjemalcu.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom **izr. prof. dr. Janeza Demšarja**
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne **16.12.2013** Podpis avtorja: _____

Hvala mentorju za vso pomoč, podporo in nasvete ter predvsem za obilo potrpežljivosti.

Rad bi se zahvalil staršema za vso podporo pri šolanju in prav tako ženi Nuši za vso podporo.

To diplomsko delo posvečam ljubima hčerkama Emi in Izabeli, in noseči ženi.

Kazalo

1. Uvod 1
2. Tehnološka zgodovina spletnih aplikacij 3
2.1 Začetki spleta 3
2.2 Zgodovina programabilnih spletnih strani – spletnih aplikacij 3
2.3 Pregled Microsoftovega razvoja spletnih platform 5
3. Teoretično ozadje 7
3.1 Kaj je dinamična spletna stran 7
3.2 Kaj je spletna aplikacija 7
3.3 Spletno ogrodje 8
3.4 Razvijalski vzorec in MVC, MVP, MVVM 8
3.4.1 MVC 8
3.4.2 MVP 9
3.4.3 MVVM 10
3.4.4 Zaključek razvijalskih vzorcev 11
3.5 REST in kaj je RESTful pristop 11
3.6 SPA – Aplikacija ene strani 12
4. Prikaz modernega razvoja spletne aplikacije na konkretnem primeru 15
4.1 Vsebinski opis aplikacije 15
4.2 Tehnični opis aplikacije 16
4.2.1 Zasnova aplikacije 16
4.2.2 Arhitektura aplikacije 18
4.2.2.1 Arhitekturni sloji DMV Portala 18
4.2.2.2 Testiranje Delov 18
4.2.2.3 SOLID pristop 19
4.2.3 Podatkovni sloj 20
4.2.3.1 Delo z bazo 21
4.2.3.2 Vzorec repozitorija 23
4.2.1 Srednji sloj 23

4.2.1.1 Domensko orientirano načrtovanje (DDD) 24
4.2.1.2 Testiranje delov na srednjem sloju 27
4.2.1.3 Preverjanje veljavnosti 29
4.2.1.4 Asinhronost 32
4.2.1.5 Skupna knjižnica 33
4.2.1.6 Vzorec »procesor« 34
4.2.1.7 Obrnjena odvisnost 34
4.2.2 Prezentacijski sloj - strežniški vzorec MVC 35
4.2.2.1 ASP.NET MVC 35
4.2.2.2 WebAPI 41
4.2.3 Prezentacijski sloj - odjemalski vzorec MVC 42
4.2.4 Oblikovanje 50
4.2.5 Priprava aplikacije za produkcijsko okolje 52
4.2.6 Oblačno okolje in namestitev aplikacije 54
5. Zaključek 57
Literatura 59

Kazalo slik

Slika 1 Primer AJAX klica s pomočjo jQuery knjižnice 4
Slika 2 Prikaz modela MVC 9
Slika 3 Prikaz modela MVP 10
Slika 4 Prikaz modela MVVM 11
Slika 5 Primer HTTP akcije - zahtevek GET 12
Slika 6 Začetna ideja aplikacije 16
Slika 7 Okolje Visual Studio 2013 17
Slika 8 Organizacija celotne aplikacije po projektih 17
Slika 9 Arhitektura DMV Portala 18
Slika 10 Primer testne funkcije 19
Slika 11 Primer registracije odvisnosti 20
Slika 12 Primer "reševanja" odvinosti preko vstavljene odvisnosti 20
Slika 13 Podatkovni model aplikacije 21
Slika 14 Primer lambda stavka za izbiro podatkov 21
Slika 15 Primer organizacije baznih skript 22
Slika 16 Primer bazne skripte za kreacijo 22
Slika 17 Primer repozitorijske funkcije "Get" 23
Slika 18 Primer dela domenske entitete 25
Slika 19 Primer dela aplikacijskega upravitelja 26
Slika 20 Primer vmesnika za dostop do baze, ki uporablja domensko entiteto 27
Slika 21 Vmesnik za uvoz avtomobila preko portala mobile.de 28
Slika 22 Funkcionalnost uvoza podatkov avtomobila 28
Slika 23 Definicija testnega razreda 29
Slika 24 Prikaz testne funkcije, ki testira uvoz enega podatka o avtomobilu 29
Slika 25 Primer asinhrone kode 33
Slika 26 Primer razporeditve kode v skupni knjižnici 33
Slika 27 Primer procesorja 34
Slika 28 Primer registracije odvisnosti 35
Slika 29 Primer zahtevka po instanci na podlagi danega vmesnika 35
Slika 30 Primer enostavnega modela z atributi 36
Slika 31 Primer kontrolerja za uvoz podatkov avtomobila 37
Slika 32 Prikaz pogleda 38
Slika 33 Prikaz prilagojenega HTML pomočnika 39
Slika 34 Opozorilo brskalnika pred ponovno akcijo POST 41
Slika 35 Primer REST api končne točke v WebAPI-ju 42
Slika 36 Predlagana ureditev JavaScript kode v projektu 43
Slika 37 Primer AngularJS kontrolerja 45

Slika 38 Datumski zbirnik na podlagi direktive 48
Slika 39 Prikaz uporabe Bootstrap ogrodja 51
Slika 40 Izgled DMV Portala na mobilni napravi 52
Slika 41 Krožna barvna shema za določitev barv strani 52
Slika 42 Primer orodja Glimpse v akciji 54
Slika 43 Dialog za namestitev aplikacije na Azure 55
Slika 44 Uporaba virov, ko je šla aplikacija v produkcijo 55

Terminologija

- SPA (ang. Single Page Application) - Aplikacija ene strani
- MVC (ang. Model View Controller) - Model-Pogled-Kontroler
- Obrnjena odvisnost/Vrinjena odvisnost - ang. Dependency Injection
- Podatkovna vez - ang. Databinding
- Preverjanje veljavnosti - ang. Validation

Povzetek

Namen tega diplomskega dela je umestiti in predstaviti razvoj moderne spletne aplikacije. Na voljo je mnogo virov o posamezni tehnologiji. Viri o tem, kako sestaviti skupek tehnologij in pristopov in razvijalskih vzorcev skupaj v eno celoto – v spletno aplikacijo – pa so redki. To diplomsko delo skuša rešiti ravno omenjeni problem. Delo ponuja celovit pregled nad modernim razvojem spletne aplikacije s poudarkom na prezentacijskem delu grajenem v razvijalskem vzorcu MVC. Pri razvoju aplikacije sta najpomembnejša konsistentnost in uporaba pravilnih spletnih ogrodij na pravi način. Hkrati je potrebno upoštevati okoliščine aplikacije in njene zahteve, temu se mora prilagoditi arhitektura in zasnova same aplikacije. Ključni del te diplomske naloge je praktični primer moderne aplikacije, ki lahko služi kot osnova za razvoj novih spletnih aplikacij. Zato je veliko poudarka v tem delu namenjenega ravno razlagi celotnega praktičnega primera aplikacije s poudarki na ključnih točkah, tehnikah in praksah, ki jih danes srečamo pri razvoju programske opreme.

Ključne besede: spletna aplikacija, MVC, MVVM, SPA, arhitektura, ASP.NET MVC, AngularJS

Summary

The purpose of this final thesis is to properly place and represent development of the modern web application. There are many sources for every given technology out there, but it is a rare breed to find a bundle of technologies assembled together in one application and get a whole picture of them how they fit together. This final thesis tries to fix exactly this problem. This work offers a complete overview of the modern web application development with a emphasis on the presentation layer which is build in the MVC pattern. The most important things in building of web applications are consistency and usage of the proper web tools in a proper way. Besides that we have to take in into an account an application environment and requirements; application architecture and application design have to adapt to this. The key part of this final thesis is a practical example of a modern web application, which can serve as a role model for new web applications. A big emphasis of the following chapters is on the explanation of this practical example, which focuses on key parts, techniques and best practises that are common in today's software development.

Keywords: web application, MVC, MVVM, SPA, architecture, ASP.NET MVC, AngularJS

1. Uvod

Tehnološki napredek na področju spletnih tehnologij in s tem tudi pristopov, tehnik in razvijalskih vzorcev je bil v zadnjih dvajsetih letih izjemen. Edina stalnica na tem področju so nenehne spremembe. Po drugi strani pa osnovni principi ostajajo enaki. To velja tudi za infrastrukturo spletnih aplikacij, kot je npr. tehnološki standard HTTP in označevalni jezik HTML. To velja tudi za nekatere razvijalske vzorce. Dober primer je razvijalski vzorec »Model-View-Controller (v nadaljevanju MVC)«, prevedeno pomeni »Model-Pogled-Kontroler«.

V nadaljevanju si bomo podrobno pogledali moderno gradnjo spletne aplikacije s pomočjo vzorca MVC. Danes je razvijalski vzorec MVC (in njegove različice) prisoten vsepovsod, vzorec pa se je najprej začel pojavljati na strežniški strani, medtem ko se danes pospešeno seli na odjemalsko stran.

Tako se postavi vprašanje, kaj je najboljše za moderno spletno aplikacijo? Kaj uporabiti kdaj, ter kako? Kakšna je najboljša kombinacija tehnologij? Kako zastaviti spletno aplikacijo, da bo zadostila ostrim kriterijem, ki veljajo danes, ko začnemo razvijati spletne aplikacije?

To delo bo podrobno razložilo razvijalski vzorec MVC, njegov smisel ter kako in zakaj ga moramo uporabljati pri razvoju kakršne koli spletne aplikacije na konkretnem primeru. Prav tako bo tudi pokazalo, kako ustrezno umestiti razvijalski vzorec MVC v celotno spletno aplikacijo, kar da samemu delu širši pomen in vrednost.

2. Tehnološka zgodovina spletnih aplikacij

2.1 Začetki spleta

Tehnološka zgodovina spletnih aplikacij je zelo pestra. Njen začetek sega v 80. leta prejšnjega stoletja v CERN [1], kjer je pogodbenik Tim Berners-Lee delal na osebni podatkovni zbirki ljudi in programskih modelov. Zahteva te aplikacije je bila, da je vsaka nova stran povezana z neko obstoječo. »Aplikacija« je morala biti platformno neodvisna, saj so imeli takrat v Cernu veliko različnih računalniških sistemov. Tako se je rodila ideja spleta, kot ga poznamo danes.

Kasneje, leta 1989 se je ta ideja razvila v HTTP 0.9 [2] in HTML [3]. Uradno pa sta bila oba standarda opisana še kasneje, leta 1991. Na tem nivoju lahko govorimo o spletu kot spletnih straneh, ki podajajo statične podatke, statičen tekst. Ta standarda sta se razvijala naprej, do HTTP 1.1 [4] in HTML 5 [5], ki sicer še ni uradno potrjen standard, a je že kar nekaj časa »de iure« standard za spletne aplikacije in ga bomo kot takega privzeli tudi mi, kot da gre za zadnji veljaven standard.

Uporabljamo termina »spletna stran« in »spletna aplikacija«. Kakšna je razlika? Spletna stran je vsaka stran, ki vsebuje neprogramabilne elemente, torej v osnovi tehnologijo, s katero se je splet začel – HTML. V začetnih letih spleta so bile na voljo samo statične, torej ne programabilne spletne strani. Kasneje se je pojavil CGI [6], leta 1995 pa se je zgodil preboj na področju dinamičnih spletnih strani. Spletna aplikacija je opisana v nadaljevanju.

2.2 Zgodovina programabilnih spletnih strani – spletnih aplikacij

Leto 1995 je pomembna prelomnica v svetu WWW, kar se tiče programabilnih spletnih aplikacij. Zgodile so se velike spremembe, tako na strani odjemalca kot na strežniški strani.

Na odjemalski strani se pojavi JavaScript [7], ki ga je razvil Brendan Eich, ko je bil zaposlen pri Netscape-u. JavaScript je interpretski programski jezik z dinamičnim tipskim sistemom, ki bazira na sintaksi programskega jezika C. Funkcije so osnovni gradniki jezika, zato marsikdo razume JavaScript kot funkcijski jezik [8]. JavaScript je bil razvit s ciljem, da bo enostaven interpretski programski jezik, ki bo seveda tekel v brskalniku. JavaScript je bil nato dolga leta koncipiran s strani velike večine razvijalske skupnosti, kot nek nepomemben enostaven programski jezik in tako se večina razvijalcev ni nikoli dovolj poglobila vanj. Morda je bil sam programski jezik tak na začetku, toda čez čas se je razvil v polnovreden programski jezik, ki postaja vedno bolj priljubljen. JavaScript je celo prisoten na strežniški strani v obliki Node.js platforme [9]. Prava moč jezika JavaScript se dandanes predvsem kaže v priljubljenosti t.i. SPA (Single Page Application – aplikacija ene strani) platform, ki počasi postaja prevladujoč model spletnih aplikacij. Torej, če pogledamo v prihodnost, lahko vidimo potencialno popolno prevlado JavaScripta tako na strežniški kot odjemalski strani. V resnici

je že od zmeraj edini programski jezik na strani odjemalca; vsi višji jeziki se prevedejo v JavaScript, tako da lahko govorimo o JavaScriptu kot o spletnem asemblerskem jeziku [10]. JavaScript je prisoten tudi na strežniški strani, predvsem v obliki podatkovnega servisa, kot REST API. A do tam je še vseeno daleč in bo vlogo, ki jo ima tu JavaScript morda celo prevzel kakšen drug jezik. Zanimivo je dejstvo, da JavaScript ni že prej postal tako priljubljen, uporaben in razširjen. Zakaj? Razlogov za to je veliko, od tega, da še ni bil dovolj razvit, do tega, ker je interpretske narave in je bil vedno performančno podhranjen v primerjavi s prevedenimi jeziki.

Statične spletne strani so pridobile dinamičnost, ko so postale programabilne. Kot rečeno, je bilo pri tem odločilno leto 1995 oz. sredina 90-ih let. Pojavili so se programski jeziki, kot PHP, ASP, Ruby, Python ter ostali. S temi programskimi jeziki so se pojavili tudi t.i. razvojni paketi (Solution Stack) [11], kot so:

- **LYME**: Linux, Yaws, Mnesia/CouchDB, Erlang
- **GLASS**: GemStone, Linux, Apache, Seaside, Smalltalk
- **LEAP**: Linux, Eucalyptus, AppScale, Python
- **WISA**: Windows Server, Internet Information Services, Microsoft SQL Server, ASP.NET
- **MEAN**: MongoDB, Express.js, AngularJS, Node.js

in najbolj znani med njimi:

- **LAMP**: Linux, Apache, MySQL, Perl/PHP/Python

Ključna tehnologija, ki je nato sledila, je leta 1999 predstavljen t.i. AJAX (XMLHttpRequest) [12]. Že nekaj let si spletne aplikacije brez tehnologije AJAX ne moremo več predstavljati; to da lahko osvežimo samo del spletne strani, je neprecenljivo. V kombinaciji s kakšno napredno JavaScript knjižnico, kot je npr. jQuery, je uporaba AJAX-a enostavna in hitra (Slika 1).

```
$.ajax({
  type: "POST",
  url: "some.php",
  data: { name: "John", location: "Boston" }
})
  .done(function( msg ) {
    alert( "Data Saved: " + msg );
  });
```

Slika 1 Primer AJAX klica s pomočjo jQuery knjižnice

Naslednja tehnologija, ki jo je potrebno omeniti v kontekstu razvoja spletnih aplikacij, je t.i. protokol WebSocket [13]. WebSocket omogoča polno dvosmerno komunikacijo čez povezavo TCP. Edina komunikacija, ki teče pri WebSocket protokolu čez http, je inicializacija.

Potrebno je še omeniti t.i. »slepo črevo« spletnih aplikacij, in sicer aplikacije, ki potrebujejo dodatno izvajalno okolje znotraj internetnega brskalnika. Najbolj znani primeri take tehnologije so Flash, Java Appleti in Silverlight. Vse te tehnologije so močno v zatonu, nekatere celo izgubljajo podporo v najnovejših različicah spletnih brskalnikov, zmagovalna tehnologija za spletne aplikacije je namreč HTML5. Glavna težava teh tehnologij je dostopnost, platformska zaprtost in varnost. Slednja je tudi razlog, zakaj te tehnologije proizvajalci spletnih brskalnikov počasi ukinjajo.

Zadnja tehnologija oz. pristop, ki postaja vedno bolj popularen, je pristop gradnje spletnih strani SPA. Sam pristop (kot je to značilno za razvijalske pristope) ne uvaja nobene nove tehnologije, temveč le ustrezno poveže vse obstoječe tehnologije, ki smo jih že spoznali. Vsi razvijalski pristopi SPA temeljijo na razvijalskem vzorcu MVC. Hkrati je lastnost večine teh ogrodij, da vsebujejo podatkovno vez (ang. databinding). Za ta razvijalski pristop obstaja cela množica programskih ogrodij, ki so vsa zgrajena okoli programskega jezika JavaScript. Primeri SPA programskih ogrodij so [14]: Ember.js, Backbone.js, Durandal in AngularJS. Slednjega smo uporabili tudi za spletno aplikacijo, ki jo bomo predstavili v nadaljnjih poglavjih.

2.3 Pregled Microsoftovega razvoja spletnih platform

V nadaljevanju si bomo podrobneje pogledali razvoj Microsoftovega spletnega razvojnega ogrodja (WISA). Tudi primer razvite aplikacije je zgrajen na podlagi razvojnega paketa WISA.

- **Klasični ASP** [15]

  Konec leta 1996 je v dinamične spletne strani vstopil Microsoftov klasični ASP. Uporabljen programski jezik je VBScript. IIS 3.0 je prvi, ki je podprl to tehnologijo. Klasični ASP je tudi močno zaostajal za konkurenti, saj so drugi ponudili, kot smo videli, programiranje spletnih strani – grajenje spletnih aplikacij – že nekaj let prej.

- **ASP.NET Web Forms**

  Leta 2002 je luč sveta ugledala velika novost za gradnjo spletnih aplikacij – ASP.NET. Šlo je za revolucionarni pristop gradnje spletnih aplikacij tistega časa. Glavna prednost ASP.NET je bila gradnja na podlagi že znanih razvijalskih pristopov

z namiznega okolja. Tako se razvijalcu ni bilo potrebno ukvarjati s podrobnostmi protokola HTTP. Naslednja velika prednost ASP.NET ogrodja je kontrolni model gradnje. Praktično vsako funkcionalnost se lahko pokrije s kontrolniki. Primer kontrolnikov so gridi, izbirniki, grafi in drugi. Zadnja leta je ASP.NET v velikem zatonu. Koncept je sicer res revolucionaren, a na koncu prinese v praksi preveč težav, sploh v času, ko morajo biti tudi spletne aplikacije visoko interaktivne. To ogrodje je lep primer »puščajoče abstrakcije« [16], danes uporabno samo za posebne primere. Imamo namreč posebne zahteve glede kontrolnikov, ki obstajajo samo za ASP.NET Web Forms.

- **Silverlight**

  Silverlight spada med tako imenovane spletne tehnologije, ki potrebujejo dodatno izvajalno okolje v brskalniku. Kot smo videli v prejšnjem poglavju, se ta pristop na splošno ni prijel in je v zatonu. Tako je v zatonu tudi Silverlight, ki je v resnici uvedel kar nekaj revolucionarnih pristopov in razvijalskih vzorcev, kot na primer servisi RIA. Na podoben koncept danes delujejo aplikacije SPA. Dober primer uporabe Silverlighta so bile LOB (ang. Line of Business) aplikacije; gre za poslovne aplikacije, ki se tipično uporabljajo znotraj organizacije na intranetu. Danes se tudi te pospešeno selijo v HTML5. Kakor koli, Silverlight je danes uporaben samo za posebne primere (npr. 3D grafika, video dekodiranje na odjemalcu, …).

- **ASP.NET MVC**

  ASP.NET MVC gradi na ogrodju ASP.NET, saj ima to ogrodje celo kopico uporabnih mehanizmov, programskih vmesnikov (API) in zgrajenih knjižnic (primer .NET System.Web knjižnica). ASP.NET MVC pa popolnoma spremeni tok spletnega zahtevka in upravljanje z gradnjo dinamične spletne strani. Tukaj ogrodje ne maskira spletnih mehanizmov (HTTP, HTML) ampak jih, nasprotno, spodbuja. Na ta način razvijalec dobi popoln nadzor nad generiranim izhodnim tokom strani, ki gre do odjemalca. Hkrati pa sam koncept MVC spodbuja dinamičnega klienta, tako da na ta način dobimo visoko interaktivno stran, kar je osnovna zahteva spletne aplikacije dandanes. Bistvena sprememba napram Web Forms je tudi mnogo tanjša kopica izvajalnih nivojev – kar pomeni hitrejše izvajanje in manj zapleteno prilagoditev lastnim potrebam.

3. Teoretično ozadje

Namen tega poglavja je predstavitev osnovnih pristopov moderne gradnje spletne aplikacije.

3.1 Kaj je dinamična spletna stran

Dinamična spletna stran [17] je za razliko od statične spletne strani generirana (oz. "renderirana") programsko. Statična stran vsebuje statičen HTML in spletni strežnik samo odgovori spletni zahtevi z ustreznim dokumentom. Pri dinamičnih spletnih straneh pa mora spletni strežnik, ko dobi spletno zahtevo, zagnati ustrezen program. Spletni strežnik nato vrne izhod tega programa. Zato na tej točki lahko govorimo o dinamičnih spletnih straneh tudi kot aplikacijah, spletnih aplikacijah.

Dinamiko spletne strani pa se lahko doseže tudi zgolj na odjemalski strani. Torej, spletni strežnik še vedno vrača statične strani, v tem primeru jih imenujemo predloge (ang. templates), odjemalska aplikacija pa ima dostop do podatkovnega vmesnika za izmenjavo podatkov. Tukaj v resnici govorimo že o aplikaciji tipa SPA [18].

Ko govorimo o generiranju strani, tu mislimo predvsem na generiranje vsebine strani in s tem povezanega HTMLja. Generiranje ostalih vsebin (JavaScript, CSS, …) je seveda možno, a manj pogosto, saj so te vsebine namenjene temu, da jih izvede odjemalec. Tipično se lahko na strežniški strani generira CSS, a predvsem za namene lažjega vzdrževanja in bolj strukturiranega pristopa. Tak primer je tehnologija za generiranje CSSja – LESS [19].

3.2 Kaj je spletna aplikacija

Spletna aplikacija [20] je vsaka spletna stran, ki je dinamično generirana in ima točno določen namen. Širša definicija pravi, da je spletna aplikacija vsaka aplikacija, ki uporablja spletni brskalnik kot odjemalca aplikacije. [21].

Prednosti spletnih aplikacij so:

- enostavnejša namestitev in nadgradnja aplikacije in izjemno enostavna uporaba (uporabnik potrebuje le spletni brskalnik),
- lažja razširljivost same aplikacije in integracija,
- boljša skalabilnost in robustnost aplikacije,
- platformna neodvisnost in dosegljivost in odprtost (odjemalski del kode je na voljo vsem v pregled).

Slabosti spletnih aplikacij so:

- težavno upravljanje z lokalnimi viri,
- nedostopnost aplikacije ob izpadu internetne povezave ali strežnika,

- težave zaradi nezdružljivosti spletnih brskalnikov (to je bil problem predvsem pred leti, danes je situacija mnogo boljša),
- izzivi glede varnosti in zasebnosti.

3.3 Spletno ogrodje

Osnova za razvoj spletnih aplikacij so spletna ogrodja [22]. Razvoj brez njih praktično ni mogoč, saj izjemno pohitrijo in olajšajo implementacijo spletne aplikacije. Za praktično vsak programski jezik, ki je pokrit s spletnimi strežniki, obstaja ogrodje. Spletna ogrodja se razlikujejo glede na nivo abstrakcije in dodatne knjižnice. Lahko podamo primera obeh skrajnosti. Že omenjeni ASP.NET je primer spletnega ogrodja visoke abstrakcije. Uporabniku tega ogrodja ni potrebno, niti v teoriji, poznati modela HTTP. Na drugi strani najnižje možne abstrakcije pa se nahaja Node.js [23], to je strežniška platforma (spletno ogrodje in še dodatno spletni strežnik) v jeziku JavaScript, ki teče v okviru izvajalnega okolja »V8« (spletni brskalnik Chrome).

Tipično spletno ogrodje vsebuje gradnike za dostop do baze, omogoča gradnjo predlog, upravljanje s sejo in ponuja API programsko knjižnico.

3.4 Razvijalski vzorec in MVC, MVP, MVVM

Obstajajo trije uveljavljeni in najbolj znani prezentacijski razvijalski vzorci, in sicer MVC, MVP in MVVM [24]. Na podlagi njih pa obstaja cela množica podvariacij. Razvijalski vzorci standardizirajo razvoj spletnih aplikacij. Osnova vseh prezentacijskih razvojnih vzorcev je MVC, torej ang. Model View Controller - Model-Pogled-Kontroler. Osnovna ideja je jasna separacija zadolžitev posameznega sloja glede tega, kaj (lahko) počne kateri. Potrebno je vedeti, da je tudi osnova praktično vseh spletnih ogrodij in tudi aplikacij razvijalski vzorec MVC.

3.4.1 MVC

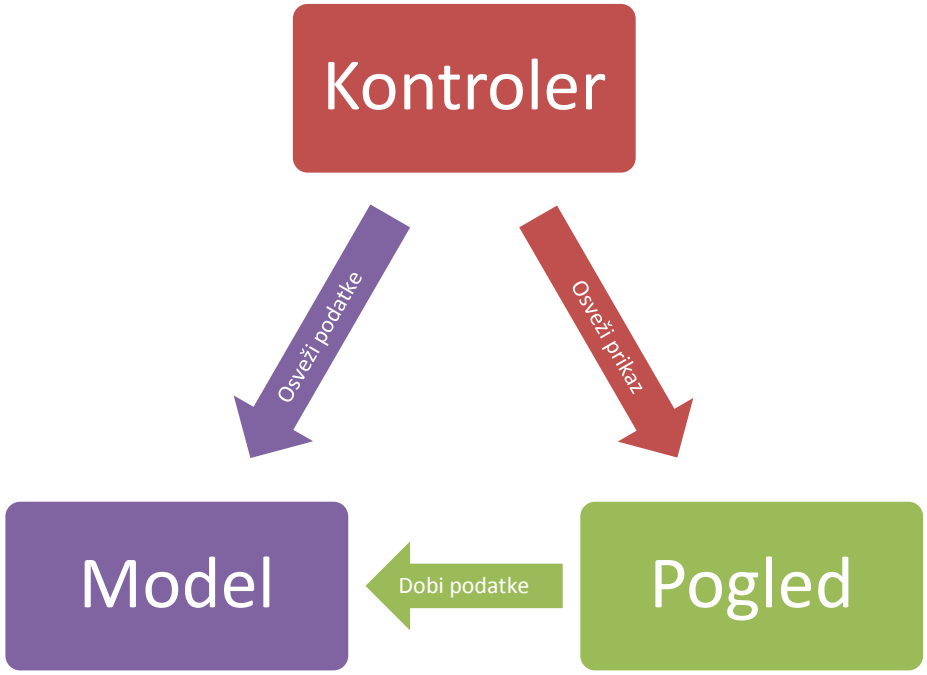
Razvijalski vzorec MVC (ang. Model View Controller) [25] - Model-Pogled-Kontroler je osnovni razvijalski vzorec spletnih aplikacij. Razvijalski vzorec MVC je bil izumljen v 70. letih prejšnjega stoletja v Xerox Parc-u [26].

Posamezni nivoji predstavljajo:

- **Model (ang. Model)**
  Model je podatkovni vir za pogled. Tipično model nastopa kot razred programskega jezika (ang. class), kot nosilec podatkov. Model lahko vsebuje podatkovno logiko. Navadno je večina poslovne logike na višjih nivojih.

- **Pogled (ang. View)**
  Pogled nastopa v spletnih aplikacijah kot HTML predloga. Pogled »dobi« tako ali drugače (na tej točki se tudi ločijo različni razvijalski vzorci) model, ki ga potem na podlagi tega, kako je sam pogled definiran, tudi prikaže. Pogled je strogo funkcijsko omejen na prikazovanje (in zajemanje) podatkov.

- **Kontroler (ang. Controller)**
  Kontroler kontrolira tok dogodkov. Koordinira model s pogledom. Kontroler ne sme vsebovati nobene poslovne logike razen kontroliranja samega toka dogodkov. Kontroler mora tako biti čim bolj osredotočen na samo orkestracijo danega procesa za dano akcijo.



Slika 2 Prikaz modela MVC

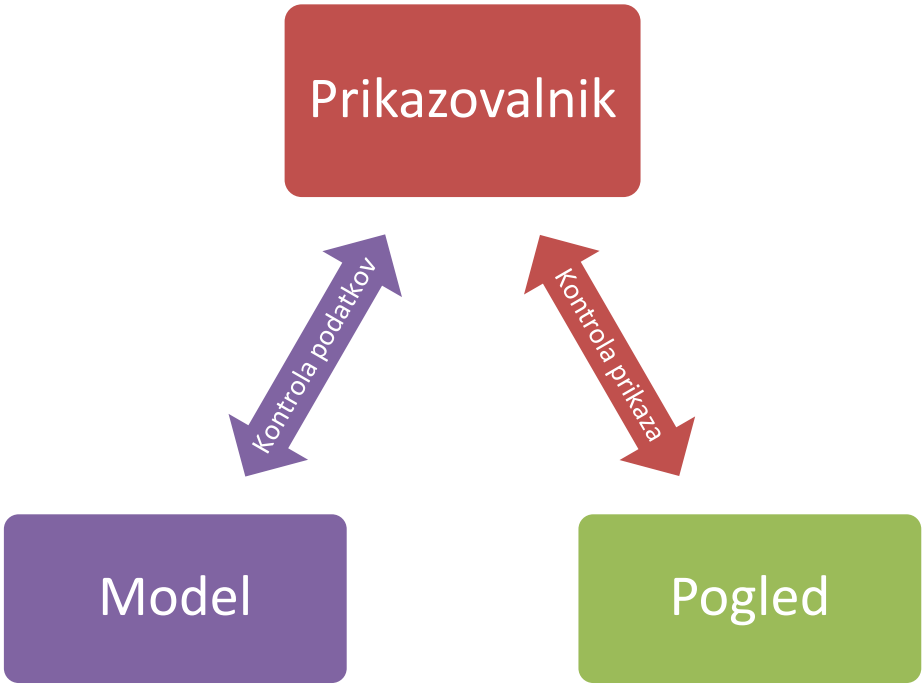
Kontroler dobi spletni zahtevek. Nato poskrbi za ustrezen model, ki ga preda pogledu. Pogled prikaže model in ustrezno preda uporabniške akcije oz. podatke naprej kontrolerju (Slika 2).

3.4.2 MVP

Razvijalski vzorec MVP (ang. Model View Presenter) [27] – Model-Pogled-Prikazovalnik, je variacija MVC razvijalskega vzorca, namesto kontrolerja v tem vzorcu nastopa prikazovalnik, ki ima vse naloge kontrolerja v MVC-ju in hkrati nanj pade še vsa prezentacijska logika, ki je v MVC vzorcu na pogledu (Slika 3).

Ta razvijalski vzorec je zastarel, saj ga je nasledil vzorec MVVM, ki si ga bomo še pogledali. Vzorec ima veliko pomanjkljivost, in sicer to, da ima prikazovalnik preveč pristojnosti.

Primer tehnologije s tem vzorcem je WinForms. Ta vzorec je enostavno implementirati in je performančno tipično boljši kot novejši MVVM model, a ima velike težave glede same kasnejše razširljivosti aplikacije in vzdrževanja. Do tega pride prav zato, ker ima prikazovalnik preveč pristojnosti.



Slika 3 Prikaz modela MVP

3.4.3 MVVM

Razvijalski vzorec MVVM (ang. Model View ViewModel) [28] – Model-Pogled-PogledModel je izboljšan vzorec MVP in seveda s tem tudi variacija vzorca MVC. Pri tem vzorcu nastopa v vlogi prikazovalnika oz. kontrolerja koncept imenovan pogledModel. Ta skrbi za interakcijo z modelom. Povezava s pogledom je implicitna preko mehanizma imenovanega »podatkovna vez« (ang. databinding). PogledModel se tako ukvarja z modelom in orkestracijo (velika podobnost s kontrolerjem), a hkrati je tu bistvena razlika v tem, kako se te spremembe odražajo na pogledu. Pogled se namreč osvežuje avtomatsko preko že omenjenega mehanizma podatkovne vezi (Slika 4). Pristojnosti pogleda v tem razvijalskem vzorcu so podobne kot pri pogledu vzorca MVC, a prilagojene na bistveno razliko podatkovne vezi.

S tem razvijalskim vzorcem smo omejeni tehnološko. Mehanizem podatkovne vezi mora podpreti spletno ogrodje oz. platforma. Spletni model tega v osnovi ne omogoča, zato tam tudi nastopa model MVC, ki lepo podpira nestatično (ang. stateless) naravo spleta. S prihodom novih razvijalskih tehnik in tehnologij ter spletnih ogrodij se tudi to spreminja, saj se spletne aplikacije selijo na odjemalsko stran, kjer pa lahko v jeziku JavaScript

implementiramo mehanizem podatkovne vezi, kar pomeni, da lahko nastopajo na odjemalski strani tudi razvijalska ogrodja, ki podpirajo MVVM.

Slika 4 Prikaz modela MVVM

3.4.4 Zaključek razvijalskih vzorcev

Kot lahko vidimo, se pri prezentacijskih razvijalskih vzorcih vse vrti okoli vzorca MVC kot osnovnega vzorca za interakcijo z uporabnikom. Kaj je glavna lastnost teh vzorcev? Glavna lastnost je v separaciji zadolžitev, kot lahko vidimo pri vzorcu MVP, kjer separacija ni tako dobra in je zato označen kot slabši oz. zastarel vzorec. Zato se zadnje čase pojavlja termin MV* kar koli (ang. MV* whatever), saj na koncu koncev ni bistveno, kateri vzorec točno uporabljamo; še več, navadno se v praksi implementira aplikacija kot mešanica vzorca MVC in MVVM (primer razvijalskega ogrodja AngularJS) ali kako drugače. Bistveno je, da skrbimo za jasno ločitev, kaj spada kam in da na noben način ne mešamo danih zadolžitev (npr. tako, da iz pogleda kličemo podatkovno bazo).

3.5 REST in kaj je RESTful pristop

REST (ang. Representational state transfer) [29] je arhitekturni stil, ki je vgrajen v sam protokol HTTP. Osnova sloga REST je uniformni identifikator vira (ang. URI - Uniform Resource Identifier). Ideja REST-a je, da nad te vire dodamo akcije HTTP, med katerimi so najpomembnejše:

- **GET**
  Zahtevek za vsebino vira. Strežnik vrne vsebino vira v spletnem odgovoru (Slika 5).
- **POST**
  Ustvarimo novo vsebino, strežnik vrne nov URI v spletnem odgovoru.
- **PUT**
  Spremenimo dani vir, akcija je idempotentna.
- **DELETE**
  Izbrišemo dani vir.

Vsaki akciji HTTP sledi spletni odgovor z določenim statusom HTTP in morebitno vsebino. Ta status nam sporoči uspešnost akcije. REST torej ni nič drugega kot dani viri in akcije, ki

jih lahko izvajamo nad tem virom. Posledično to pomeni, da ima tipičen vmesnik REST celo množico različnih virov, ki jih z lahkoto uporabljamo, saj so akcije nad njimi uniformne. To je glavna lepota pristopa REST. Vmesniku, ki zadosti tem enostavnim pogojem, pravimo tudi RESTful.

Slika 5 Primer HTTP akcije - zahtevek GET

3.6 SPA – Aplikacija ene strani

SPA (ang. Single Page Application) [30] Aplikacije ene strani smo že omenili. Gre za zadnji trend razvoja spletnih aplikacij, in sicer gre za premik celotne spletne aplikacije na odjemalca, i.e. spletni brskalnik. Na strežniku ostane samo REST API za podatkovno komunikacijo z odjemalsko aplikacijo in spletni strežnik, ki servira statične predloge za sloj pogleda.

Tipičen razvijalski vzorec uporabljen v ogrodjih za SPA je MVVM. Glavna tehnična razlika napram klasični spletni aplikaciji je v tem, da se v čistokrvni SPA aplikaciji stran nikoli ne osveži v celoti (spletni brskalnik nikoli ne naloži cele strani, razen na začetku), temveč se vse dogaja na odjemalcu v okviru posebnega mehanizma spletnega brskalnika imenovanega DOM (ang. Document Object Model) in programskega jezika JavaScript.

Zakaj postaja SPA pristop vedno bolj priljubljen? Popularnost pristopa SPA raste s priljubljenostjo JavaScripta. Priljubljenost tega programskega jezika pa vztrajno narašča. To se tudi lepo kaže s pohodom JavaScripta na strežniški strani. Še več, obstajajo tudi podatkovne baze vrste NoSQL, ki shranjujejo podatke v formatu JSON, ki je naraven podatkovni format v JavaScriptu. Torej prvi razlog je razvijalsko-tehnološki.

Po drugi strani je arhitekturno gledano pristop, v katerem se sama aplikacija izvaja na odjemalcu performančno izjemno ugoden. Skalabilnost takega pristopa je tudi mnogo večja

kot pri klasičnem pristopu gradnje spletnih aplikacij, ker se večina aplikacije izvaja na samem odjemalcu.

Tipično se zaenkrat v praksi najbolj izkaže hibridni model spletne aplikacije, torej prava kombinacija odjemalskega dela gradnje aplikacije in klasične spletne aplikacije na spletnem strežniku. Kombinacija je v današnjem času idealna predvsem za večje spletne aplikacije.

Primeri ogrodij SPA so: Durandal, Backbone.js, AngularJS (to ogrodje je uporabljeno za praktični primer te naloge. Podprt s strani Googla.), Ember.js in KnockoutJS.

4. Prikaz modernega razvoja spletne aplikacije na konkretnem primeru

V tem poglavju si bomo pogledali, kako razviti, razumeti in implementirati moderno spletno aplikacijo. Aplikacija temelji na razvijalskem vzorcu MVC in vsebuje elemente SPA. Aplikacija tudi rešuje konkreten problem, in sicer izračunavanje davka na motorna vozila (DMV) [31].

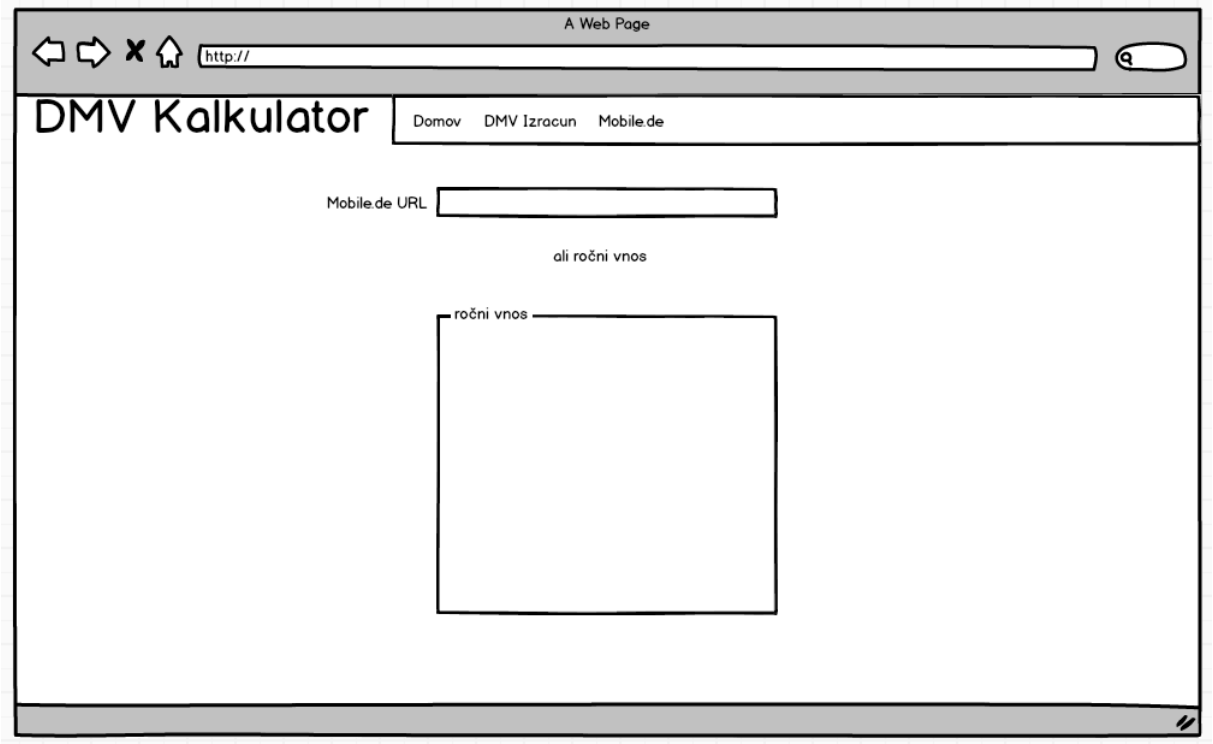
4.1 Vsebinski opis aplikacije

Uvoz avtomobilov iz tujine, še posebej iz EU, je zaradi našega majhnega avtomobilskega trga rabljenih avtomobilov izjemno priljubljen. Poleg ostalih preprek je najpomembnejša informacija pri uvozu danega avtomobila, kakšne davke bo potrebno plačati domovini. Z letom 2010 se je izračun izjemno zapletel in vezal predvsem na izpust CO_2. Tukaj priskoči na pomoč aplikacija, ki je bila razvita za namene tega diplomskega dela.

Ko iskalec avtomobila kupuje avtomobil v tujini, je ena glavnih izvorov Nemčija. V Nemčiji je najbolj priljubljen portal mobile.de [32]. Aplikacija, ki smo jo razvili, omogoča direkten uvoz avtomobila (oz. njegovih podatkov) iz portala mobile.de preko naslova URL.

Z uporabo aplikacije lahko tako izračunamo DMV na podlagi številnih parametrov preko ročnega vnosa ali pa preko uvoza iz portala mobile.de. DMV portal, kakor smo poimenovali aplikacijo, omogoča tudi prijavo in nekaj enostavnih statistik.

Aplikacijo smo začeli snovati preko mrežnih modelov, kar kaže Slika 6.



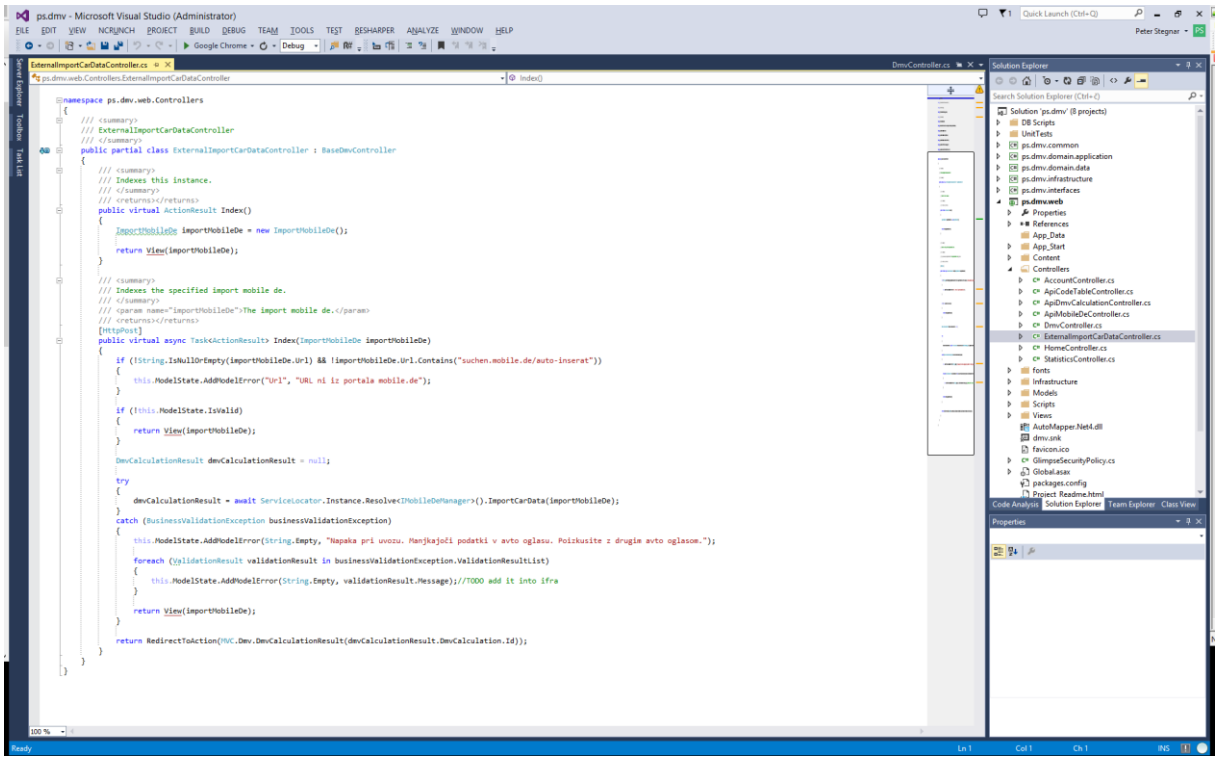
Slika 6 Začetna ideja aplikacije

4.2 Tehnični opis aplikacije

Aplikacija je zgrajena več nivojsko. Srednji sloj je zgrajen na podlagi razvijalskega pristopa DDD (ang. Domain Driven Design), prezentacijski sloj pa na podlagi razvijalskega vzorca MVC. Aplikacija je zasnovana tako, da jo je možno uporabiti kot osnovo za razvoj kakšne druge aplikacije. Izvorna koda je na voljo na GitHub-u [33].

4.2.1 Zasnova aplikacije

Aplikacija je razvita v okolju Visual Studio 2013 (Slika 7) in je zgrajena večinoma v programskih ogrodij proizvajalca Microsoft. Uporabljene so bile vse zadnje verzije teh ogrodij. Tudi sami pristopi in uporabljeni vzorci so izbrani tako, da so moderni in predvsem tako, da so tudi uporabni in se tudi dejansko uporabljajo v praksi.

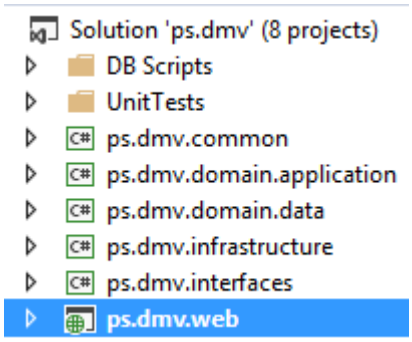


Slika 7 Okolje Visual Studio 2013

Velik poudarek pri gradnji aplikacije je bil namenjen temu, kako lahko bo aplikacijo vzdrževati, nadgrajevati in izboljševati. Ključen poudarek je bil tudi na tem, da je aplikacijo možno testirati. To je namreč pomembno tudi za aplikacije v »realnem svetu«.

Osnova za vsako aplikacijo, ki vsebuje interakcijo z uporabnikom, so zaslonske maske. Razvijalec lahko tako lažje in pravilno umesti grafične elemente na obrazec (Slika 6).

V nadaljevanju si bomo pogledali celotno aplikacijo po aplikacijskih slojih od podatkovne zbirke do prezentacijskega sloja (Slika 8).



Slika 8 Organizacija celotne aplikacije po projektih

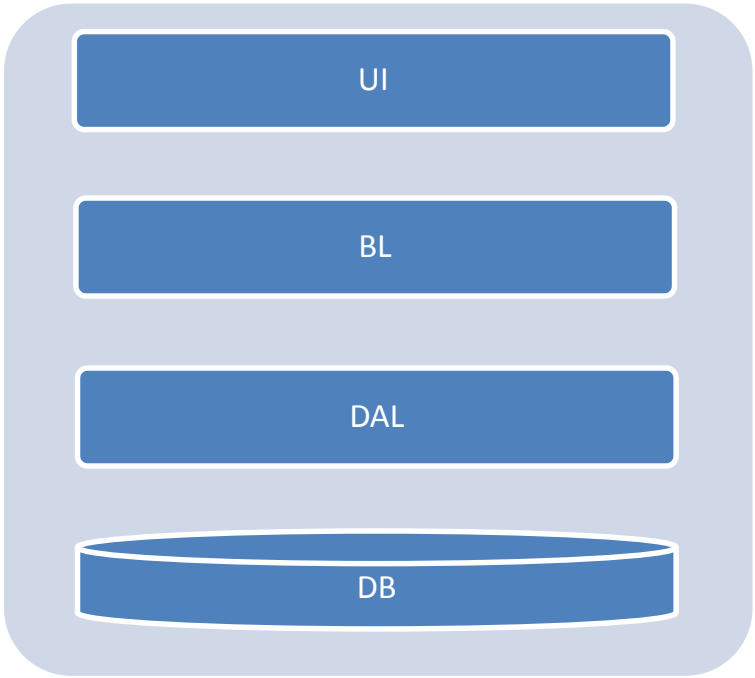
4.2.2 Arhitektura aplikacije

Arhitekturna zasnova aplikacije je klasično troslojna, in sicer vključuje podatkovni nivo (ang. DAL – Data Access Layer), srednji sloj (ang. BL – Business Logic) in prezentacijski sloj (ang. UI – User Interface). Znotraj slojev mora biti funkcionalnost prav tako strogo razdeljena po posameznih enotah.

Sloji in večinoma tudi znotraj posameznih enot morajo biti ločeni preko vmesnika (ang. interface), saj le na ta način lahko dosežemo, da je dani sloj oz. enoto možno testirati. Hkrati pa s tem dosežemo nizko stopnjo sklopljenosti, kar je izjemno pomembno, če želimo ohraniti aplikacijo razširljivo in jo ohraniti enostavno za vzdrževanje.

4.2.2.1 Arhitekturni sloji DMV Portala

Kot omenjeno ima DMV Portal klasično 3-slojno arhitekturo (Slika 9). Ta arhitekturni pristop je pomemben, podobno kot pri razvijalskem vzorcu MVC, zaradi ločitve odgovornosti posameznega sloja. Naprej, posamezni sloj ima lahko naprej ločene odgovornosti, dober primer je »sloj UI«, ki seveda vsebuje razvijalski vzorec MVC.



Slika 9 Arhitektura DMV Portala

4.2.2.2 Testiranje Delov

Testiranje delov (ang. unit testing) je izjemno pomembna postavka pri razvoju programske opreme. Testiranje delov je postal standard pri razvoju programske opreme. Še več, nobena profesionalna aplikacija ne bi smela obstajati brez testiranja delov.

Testiranje delov je ključno že med samim razvojem aplikacije, saj nam zagotavlja pravilnost delovanja posameznih delov čez cel cikel razvoja in enostavno omogoča konstantno predrugačenje (ang. refactoring) izvorne kode, ki je stalnica pri razvoju programske opreme.

Slika 10 prikazuje primer testne funkcije, ki testira en majhen del aplikacije.

```
[Test]
public async void ImportCarFromMobileDeTestkW()
{
    // arrange
    IMobileDeProcessor mobileDeProcessor = ServiceLocator.Instance.Resolve<IMobileDeProcessor>();
    ImportMobileDe importMobileDe = new ImportMobileDe() { Url = _testCarUrlOK1, VehicleTypeId = VehicleTypeEnum.Car };

    // act
    MobileDeCar mobileDeCar = await mobileDeProcessor.ImportCarFromMobileDe(importMobileDe);

    //assert
    Assert.IsTrue(mobileDeCar.DmvCalculation.EnginePowerKw > 0);
}
```

Slika 10 Primer testne funkcije

4.2.2.3 SOLID pristop

Za razvoj obširne programske opreme obstaja več principov in pristopov. Eden od njih je tudi SOLID [34], ki je okrajšava za ang. Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion, skovan s strani Roberta C. Martina.

- **S - Enojna odgovornost**

  Razred (ang. class) naj ima le eno samo odgovornost. Odgovornost lahko gledamo tudi kot »razlog za spremembo«. Torej razred naj bi imel le en razlog za spremembo.

- **O - Odprto-zaprto**

  Entitete naj bodo odprte za razširitev, toda zaprte za modifikacijo. Tipično to lahko dosežemo s polimorfizmom, npr. abstraktnim osnovnim razredom.

- **L - Liskov princip zamenjave**

  Objekti naj bodo potencialno zamenljivi z instancami njihovih podtipov, ne da se ob tem spremeni pravilnost programa. Ta princip v resnici govori o dedovanju (ang. inheritance).

- **I - Segregacija vmesnika**

Več specifičnih vmesnikov je boljše kot en generalni vmesnik. Vmesniki naj vsebujejo dobro granulirano funkcionalnost, tako da uporabniku tega vmesnika ni potrebno implementirati ničesar, česar ne potrebuje.

- **D – Obrnjena odvisnost**

Odvisnost mora bazirati na abstrakciji, ne na konkretni implementaciji. Princip govori o nizki sklopljenosti programske kode. Konkreten primer oz. arhitekturni pristop, ki pomaga pri razklopljeni kodi je vstavljanje odvisnosti (ang. Dependency Injection). Primer vstavljanja odvisnosti se nahaja tudi v praktičnem delu te diplomske naloge.

Obrnjeno odvisnost je vedno potrebo najprej registrirati, kar prikazuje Slika 11.

```
// Repositories
container.RegisterType<IDmvCalculationRepository, DmvCalculationRepository>();
container.RegisterType<IVehicleTypeRepository, VehicleTypeRepository>();
container.RegisterType<IEngineTypeRepository, EngineTypeRepository>();
container.RegisterType<IEuroExhaustTypeRepository, EuroExhaustTypeRepository>();
container.RegisterType<IFuelTypeRepository, FuelTypeRepository>();
container.RegisterType<IMobileDeRepository, MobileDeRepository>();
container.RegisterType<IStatisticsRepository, StatisticsRepository>();
```

Slika 11 Primer registracije odvisnosti

Primer uporabe obrnjene odvisnosti prikazuje Slika 12, kjer samo deklariramo za kateri vmesnik želimo instanco.

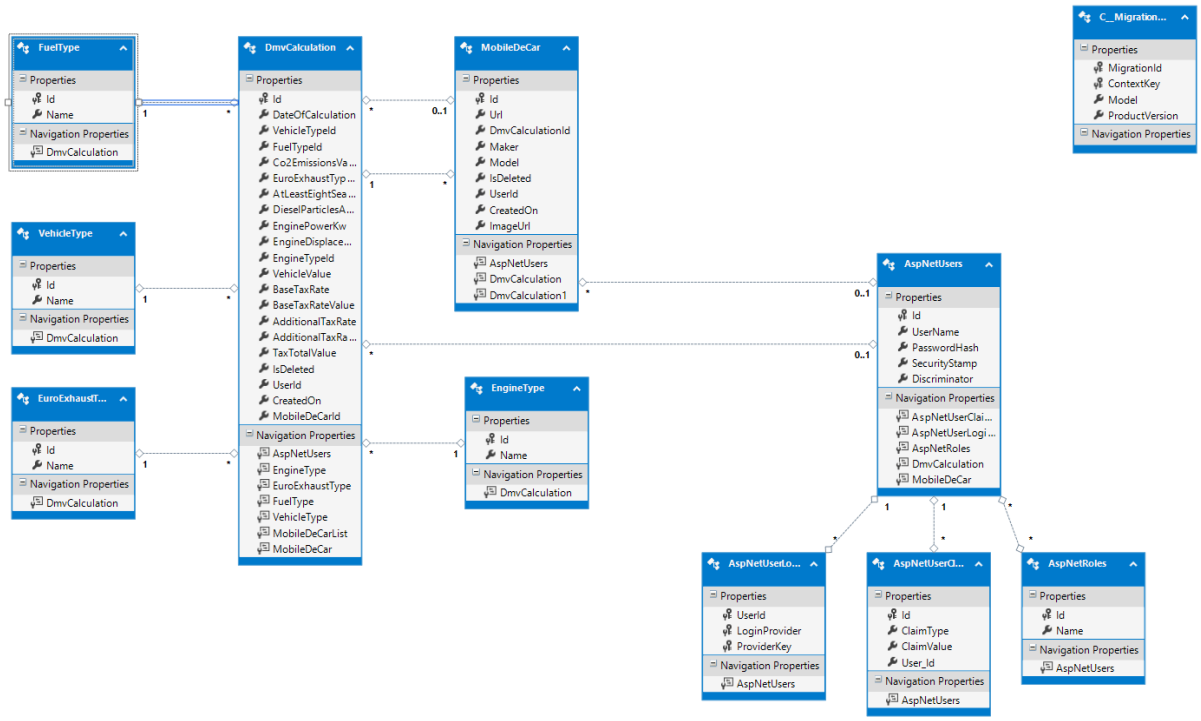
```
_dmvCalculationRepository = ServiceLocator.Instance.Resolve<IDmvCalculationRepository>();
```

Slika 12 Primer "reševanja" odvinosti preko vstavljene odvisnosti

4.2.3 Podatkovni sloj

Osnova vsake aplikacije so podatki. Za podatkovni strežnik aplikacija uporablja MS SQL 2014 CTP 1. Na aplikacijskem delu za dostop do baze pa se uporablja Entity Framework 6.02 (EF), za objektno relacijsko mapiranje (ang. ORM – Object Relational Mapping). Osnova EF-ja je urejevalnik entitet in njihovih relacij.

Podatkovni model aplikacije DMV portal, ki se nahaja v urejevalniku entitet ogrodja EF prikazuje Slika 13.



Slika 13 Podatkovni model aplikacije

ORM je eden ključnih elementov moderne aplikacije. ORM abstrahira dostop do baze, ki tako ne poteka neposredno preko poizvedb SQL, temveč se navadno pišejo poizvedbe neposredno v programskem jeziku, v katerem je aplikacija narejena. V primeru EF se pišejo podatkovne poizvedbe v jeziku C#, najpogosteje v obliki lambda stavkov (Slika 14).

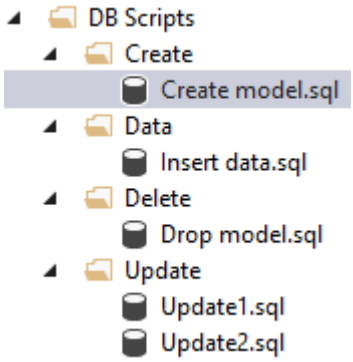
```
dmvCalculationDb = db.DmvCalculation.Include("MobileDeCar").Where(c => c.IsDeleted == false && c.Id == id).FirstOrDefault();
```

Slika 14 Primer lambda stavka za izbiro podatkov

4.2.3.1 Delo z bazo

Pri delu z bazo je zelo pomembna prava organizacija baznih skript. Dober princip organizacije baznih skript je, da so razdeljene kot skripte za (Slika 15):

- ustvarjanje baze,
- dostop do podatkov,
- izbris,
- osvežitev.



Slika 15 Primer organizacije baznih skript

Pomembno je, da imamo celotno bazo v skriptah (Slika 16). Predvsem osvežitve (ang. update) baze morajo biti v idempotentno narejenih skriptah. Baza mora imeti omejilnike (ang. constraints), predvsem te, ki skrbijo za konsistentnost baze in referenčno integriteto. Poslovna logika se ne sme nahajati na baznem, ampak v aplikacijskem nivoju.

```
-- [DmvCalculation]
CREATE TABLE [dbo].[DmvCalculation](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [DateOfCalculation] [datetime] NOT NULL,
    [VehicleTypeId] [int] NOT NULL,
    [FuelTypeId] [int] NOT NULL,
    [Co2EmissionsValue] [smallint] NOT NULL,
    [EuroExhaustTypeId] [int] NOT NULL,
    [AtLeastEightSeatsVehicle] [bit] NOT NULL,
    [DieselParticlesAbove005Limit] [bit] NOT NULL,
    [EnginePowerKw] [int] NOT NULL,
    [EngineDisplacementCcm] [int] NOT NULL,
    [EngineTypeId] [int] NOT NULL,
    [VehicleValue] [float] NOT NULL,
    [BaseTaxRate] [float] NOT NULL,
    [BaseTaxRateValue] [float] NOT NULL,
    [AdditionalTaxRate] [float] NOT NULL,
    [AdditionalTaxRateValue] [float] NOT NULL,
    [TaxTotalValue] [float] NOT NULL,
    [IsDeleted] [bit] NOT NULL,
    [UserId] [nvarchar](128) NULL,
    [CreatedOn] [datetime] DEFAULT GETDATE() NOT NULL,
    [MobileDeCarId] [int] NULL,
    PRIMARY KEY([Id])
)
GO

ALTER TABLE [dbo].[DmvCalculation]  WITH CHECK ADD CONSTRAINT FK_DmvCalculation_VehicleType FOREIGN KEY([Veh
GO

ALTER TABLE [dbo].[DmvCalculation]  WITH CHECK ADD CONSTRAINT FK_DmvCalculation_FuelType FOREIGN KEY([FuelTy
GO

ALTER TABLE [dbo].[DmvCalculation]  WITH CHECK ADD CONSTRAINT FK_DmvCalculation_EuroExhaustType FOREIGN KEY(
GO

ALTER TABLE [dbo].[DmvCalculation]  WITH CHECK ADD CONSTRAINT FK_DmvCalculation_EngineType FOREIGN KEY([Engi
GO

ALTER TABLE [dbo].[DmvCalculation]  WITH CHECK ADD CONSTRAINT FK_DmvCalculation_AspNetUsers FOREIGN KEY([Use
GO
```

Slika 16 Primer bazne skripte za kreacijo

4.2.3.2 Vzorec repozitorija

Vzorec repozitorija je razvijalski vzorec [35] [36], ki narekuje urejenost dostopa do baze, predvsem, kako pravilno abstrahirati višjim slojem dostop do baze. Najbolj pomembna vloga tega vzorca je, da zameji dostop poslovne logike neposredno do baze oz. da bi poslovna logika »preveč vedela« o baznem sloju. S tem vzorcem tudi dosežemo večjo uporabnost kode, ki dostopa do baze.

Tipično repozitorij vsebuje CRUD funkcije. CRUD (ang. Create Read Update Delete) funkcije so kreacijske, bralne, osvežitvene in izbrisne. Te funkcije so tudi izpostavljene preko vmesnika naprej višjemu nivoju (srednjemu sloju).

Primer ene repozitorijske funkcije prikazuje Slika 17.

```
73      /// <summary>
74      /// Gets the specified identifier.
75      /// </summary>
76      /// <param name="id">The identifier.</param>
77      /// <returns></returns>
78      public Domain.DmvCalculation Get(int id)
79      {
80          DmvCalculation dmvCalculationDb = null;
81
82          using (DmvEntities db = new DmvEntities())
83          {
84              dmvCalculationDb = db.DmvCalculation.Include("MobileDeCar")
85                  .Where(c => c.IsDeleted == false && c.Id == id).FirstOrDefault();
86          }
87
88          Domain.DmvCalculation dmvCalculationEntity = Mapper.Map<Domain.DmvCalculation>(dmvCalculationDb);
89
90          return dmvCalculationEntity;
91      }
```

Slika 17 Primer repozitorijske funkcije "Get"

Opis kode s slike: Najprej moramo inicializirati kontekst podatkovnega modela (vrstica 82). Nato z lambda izrazom ustvarimo posredno poizvedbo nad podatkovno zbirko (vrstica 84,85).

4.2.1 Srednji sloj

Srednji sloj je srce aplikacije. Tu se tudi skriva največja poslovna vrednost same rešitve. Malo je razvijalskih vzorcev, ki celostno pokrijejo srednji sloj. Eden izmed njih se imenuje DDD (ang. Domain Driven Design) Domensko orientirano načrtovanje [37] [38].

DDD je zastavljen zelo široko in ne ostaja samo v tehničnih okvirih, temveč predlaga tudi, kako se lotiti poslovnih zahtev, torej kako poslovni problem pretvoriti v programsko kodo, in najpomembnejše, kako to programsko kodo urediti, umestiti v celostno arhitekturo aplikacije.

V nadaljevanju si bomo pogledali prilagojen model DDD-ja, ki ustreza večini aplikacij, tako po velikosti, kot po zahtevnosti in vsebuje vse nujno potrebne elemente, ki so ideja DDD-ja.

4.2.1.1 Domensko orientirano načrtovanje (DDD)

Osnova DDD-ja je domenska entiteta, torej nositelj podatkov danega problema. Ta domenska entiteta je zamejena z domenskim kontekstom. Domenski kontekst je pristop, kako razbiti velik domenski problem, torej klasičen pristop razbitja večjega problema na manjše dele. Razvoj nove funkcionalnosti se po DDD-ju tudi začne z domensko entiteto nato pa sledi razvoj na ostalih slojih.

DDD se močno fokusira na samo problemsko domeno, a hkrati da močan poudarek arhitekturni zgradbi. Ena od idej DDD-ja je, da mora biti sam domenski model zgrajen tako, da ga je možno vzeti iz sistema in ga postaviti v drug sistem. To pomeni, da mora biti v osnovi domenski model zgrajen tako, da je nizko sklopljen z ostalim sistemom: domenski model je dostopen oz. dostopa samo preko vmesnikov in vstavljene odvisnosti. Ta ideja o možnosti postavitve domenskega sistema v drug sistem predvsem pomaga pri načrtovanju in pri odločevanju pri konkretnih arhitekturnih problemih.

Osnovni gradniki DDD:

- **Domenska entiteta**

  Vse se vrti okoli domenske entitete (Slika 18). Tukaj se začne načrtovanje nove funkcionalnosti. Domenska entiteta je preprosto razred, ki vsebuje lastnosti dane entitete in ostalo funkcionalnost, kot npr. preverjanje veljavnosti. Iz domenskih entitet izhaja tako podatkovni model, kot tudi entitete višjih slojev (npr. ViewModel).

```
/// <summary>
/// DmvCalculation
/// </summary>
[HasSelfValidation]
public class DmvCalculation
{
    #region Properties

    /// <summary>
    /// Gets or sets the identifier.
    /// </summary>
    /// <value>
    /// The identifier.
    /// </value>
    public int Id { get; set; }

    /// <summary>
    /// Gets or sets the date of calculation.
    /// </summary>
    /// <value>
    /// The date of calculation.
    /// </value>
    [DisplayName("Datum izračuna")]
    public DateTime DateOfCalculation { get; set; }

    /// <summary>
    /// Gets or sets the vehicle type identifier.
    /// </summary>
    /// <value>
    /// The vehicle type identifier.
    /// </value>
    [DisplayName("Vozilo")]
    [Required(ErrorMessage = "{0} je obvezno polje.")]
    public VehicleTypeEnum VehicleTypeId { get; set; }

    /// <summary>
    /// Gets or sets the fuel type identifier.
    /// </summary>
    /// <value>
    /// The fuel type identifier.
    /// </value>
    [DisplayName("Gorivo")]
    [Required(ErrorMessage = "{0} je obvezno polje.")]
    public FuelTypeEnum FuelTypeId { get; set; }
```

Slika 18 Primer dela domenske entitete

- **Aplikacijski upravitelj**

Aplikacijski upravitelj (Slika 19) je orkestrator danega domenskega konteksta oz. funkcionalnosti. Aplikacijski upravitelj skrbi za programski tok na ravni poslovne logike. Aplikacijski upravitelj neposredno ne vsebuje nobene poslovne logike, ampak le skrbi, da je ustrezno klicana. Aplikacijski upravitelj tudi skrbi oz. upravlja podatkovni sloj.

```
/// <summary>
/// DmvCalculationManager
/// </summary>
public class DmvCalculationManager : ManagerBase<DmvCalculation>, IDmvCalculationManager
{
    private IDmvCalculationRepository _dmvCalculationRepository = null;

    /// <summary>
    /// Initializes a new instance of the <see cref="DmvCalculationManager"/> class.
    /// </summary>
    public DmvCalculationManager()
    {
        _dmvCalculationRepository = ServiceLocator.Instance.Resolve<IDmvCalculationRepository>();
    }

    /// <summary>
    /// Gets all.
    /// </summary>
    /// <param name="pageIndex">Index of the page.</param>
    /// <param name="pageSize">Size of the page.</param>
    /// <param name="includeImportedCalculation"></param>
    /// <returns></returns>
    public PagedList<DmvCalculation> GetAll(int pageIndex, int pageSize, bool includeImportedCalculation)
    {
        return _dmvCalculationRepository.GetAll(pageIndex, pageSize, includeImportedCalculation);
    }

    /// <summary>
    /// Gets the DMV tax value result.
    /// </summary>
    /// <param name="dmvCalculation">The DMV calculation.</param>
    /// <returns></returns>
    public async Task<DmvCalculationResult> ProcessDmvTaxValueResult(DmvCalculation dmvCalculation)
    {
        DmvCalculation dmvCalculationCalculated = ServiceLocator.Instance.Resolve<IDmvCalculationProcessor>().CalculateDmvAll(dmvCalculation);

        dmvCalculationCalculated.CreatedOn = DateTime.UtcNow;

        base.Validate(dmvCalculationCalculated);

        dmvCalculationCalculated = await _dmvCalculationRepository.Save(dmvCalculationCalculated);

        DmvCalculationResult dmvCalculationResult = new DmvCalculationResult(dmvCalculationCalculated);

        return dmvCalculationResult;
    }

    /// <summary>
    /// Gets the last DMV calculation result.
    /// </summary>
    /// <param name="numberOfLastResponses">The number of last responses.</param>
    /// <param name="includeImportedCalculation"></param>
    /// <returns></returns>
    public List<Task<DmvCalculationResult>> GetLastDmvCalculationResult(int numberOfLastResponses, bool includeImportedCalculation)
    {
        return _dmvCalculationRepository.GetAll(DmvConstants.InitialPageIndex, numberOfLastResponses, includeImportedCalculation).Select(async i =>
    }
```

Slika 19 Primer dela aplikacijskega upravitelja

- **Domenski upravitelj**

  Domenski upravitelj vsebuje ključno poslovno logiko. Tipično nima dostopa do podatkovne baze razen za potrebe izvajanja same poslovne logike. Na primer, iz baze domenski upravitelj lahko dobi razne parametre, ki so potrebni za izvajanje poslovne logike. Primer poslovne logike so razni izračuni, zapleteno preverjanje veljavnosti, … Tipično je domenski upravitelj klican s strani aplikacijskega upravitelja.

- **Infrastruktura**

  Infrastruktura v DDD-ju je koncept, ki se dotika zunanjih okvirov domenskega modela. Tipično se v infrastrukturi nahaja podatkovni sloj, torej funkcionalnost dostopa do baze. Naslednji tipični predstavnik infrastrukture je dostop, integracija z zunanjimi storitvami.

Kot rečeno, je pomembno, da je domenski model nizko sklopljen. To pomeni, da so vse odvisnosti ločene preko vmesnikov (Slika 20) in hkrati, da vmesnik vsebuje tipe, ki so znani domenskemu modelu.

```
/// <summary>
/// Saves the specified DMV calculation.
/// </summary>
/// <param name="dmvCalculation">The DMV calculation.</param>
/// <returns></returns>
Task<DmvCalculation> Save(DmvCalculation dmvCalculation);
```

Slika 20 Primer vmesnika za dostop do baze, ki uporablja domensko entiteto

Na ravni slojev navadno nastopajo tudi fizično druge entitete in tako je potrebno njihovo medsebojno mapiranje. Ta postopek je potrebno avtomatizirati, pogoj za to pa je, da so imena lastnosti entitet enaka. Zelo uporabno orodje v svetu .NET je AutoMapper [39].

Najprej preprosto definiramo mapiranje med entitetama:

```
Mapper.CreateMap<MobileDeCar, Domain.MobileDeCar>();
Mapper.CreateMap<Domain.MobileDeCar, MobileDeCar>();

Mapper.CreateMap<DmvCalculation, Domain.DmvCalculation>();
Mapper.CreateMap<Domain.DmvCalculation, DmvCalculation>();
```

Nato pa uporabimo definicijo nad konkretno instanco entitete:

```
List<Domain.MobileDeCar> mobileDeCarEntityList = Mapper.Map<List<MobileDeCar>, List<Domain.MobileDeCar>>(mobileDeCarDbList);
```

Tukaj se na primer v repozitorijski funkciji mapira bazni objekt v domenskega in tako funkcija lahko vrne domenski tip, tipično aplikacijskemu upravitelju. Na višjih nivojih pa je princip podoben; preden se kliče aplikacijski upravitelj, je potrebno mapirati entitete tega sloja v domenske.

4.2.1.2 Testiranje delov na srednjem sloju

Konkreten prikaz enega testa, ki testira uvoz podatkov avtomobila s portala mobile.de. Najprej, razred, ki ga želimo testirati mora imeti ustrezen vmesnik (Slika 21).

```
/// <summary>
/// IMobileDeProcessor
/// </summary>
public interface IMobileDeProcessor
{
    /// <summary>
    /// Imports the car from mobile de.
    /// </summary>
    /// <param name="importMobileDe">The import mobile de.</param>
    /// <returns></returns>
    Task<MobileDeCar> ImportCarFromMobileDe(ImportMobileDe importMobileDe);
}
```

Slika 21 Vmesnik za uvoz avtomobila preko portala mobile.de

Nato mora ta vmesnik implementirati razred, ki vsebuje konkretno implementacijo, to je v našem primeru razred »MobileDeProcessor« (Slika 22).

```
/// <summary>
/// MobileDeProcessor
/// </summary>
public class MobileDeProcessor : IMobileDeProcessor
{
    /// <summary>
    /// Imports the car from mobile de.
    /// </summary>
    /// <param name="importMobileDe">The import mobile de.</param>
    /// <returns></returns>
    public async Task<MobileDeCar> ImportCarFromMobileDe(ImportMobileDe importMobileDe)
    {
        WebPageParser webPageParser = new WebPageParser();

        await webPageParser.ParseWebPage(importMobileDe.Url);

        MobileDeCar mobileDeCar = new MobileDeCar();

        // Default setting for car

        mobileDeCar.Url = importMobileDe.Url;
        mobileDeCar.DmvCalculation.VehicleTypeId = importMobileDe.VehicleTypeId;
        mobileDeCar.DmvCalculation.DateOfCalculation = DateTime.UtcNow;
        mobileDeCar.DmvCalculation.EngineTypeId = EngineTypeEnum.FourTactsRest;
        mobileDeCar.UserId = ServiceLocator.Instance.Resolve<IUserProvider>().GetCurrentUserId();

        //

        string resultNode = null;
        string webPageNode = null;

        webPageNode = webPageParser.GetWebPageNode("co2EmissionValue");
        if (webPageNode != null)
        {
            resultNode = webPageParser.GetWebPageNodeNumericContent(webPageNode);
            mobileDeCar.DmvCalculation.Co2EmissionsValue = Convert.ToInt16(resultNode);
        }
```

Slika 22 Funkcionalnost uvoza podatkov avtomobila

Nato nastopi testiranje tega dela aplikacije. V ta namen potrebujemo testno orodje; v okolju .NET je najpopularnejše okolje NUnit [40]. Zgraditi moramo testni razred (Slika 23).

```
[TestFixture]
public class MobileDeProcessorTest : BaseTest
```

Slika 23 Definicija testnega razreda

In nato v samem razredu implementiramo testno funkcijo, ki nam bo testirala dani del. V našem primeru je to uvoz določenega podatka avtomobila (Slika 24).

```
42      [Test]
43      public async void ImportCarFromMobileDeTestkW()
44      {
45          // arrange
46          IMobileDeProcessor mobileDeProcessor = ServiceLocator.Instance.Resolve<IMobileDeProcessor>();
47          ImportMobileDe importMobileDe = new ImportMobileDe()
48          {
49              Url = _testCarUrlOK1,
50              VehicleTypeId = VehicleTypeEnum.Car
51          };
52
53          // act
54          MobileDeCar mobileDeCar = await mobileDeProcessor.ImportCarFromMobileDe(importMobileDe);
55
56          //assert
57          Assert.IsTrue(mobileDeCar.DmvCalculation.EnginePowerKw > 0);
58      }
```

Slika 24 Prikaz testne funkcije, ki testira uvoz enega podatka o avtomobilu

Opis kode s slike: Dobra praksa testne funkcije je, da je sestavljena v treh delih. Pri del je priprava (vrstica 45), nato sledi dejanje (vrstica 53) in na koncu še preverjanje veljavnosti rezultata (vrstica 56).

4.2.1.3 Preverjanje veljavnosti

Preverjanje veljavnosti na srednjem sloju je izjemno pomembna. Gre namreč za zadnji korak pred zapisom podatkov v bazo. Tipično se tudi velika večina poslovne logike skriva prav v obliki manj ali bolj zahtevnega preverjanja veljavnosti.

Za namene preverjanja veljavnosti srednjega sloja priporočamo uporabo aplikacijskega bloka za preverjanje veljavnosti (ang. Validation Application Block) [41], skupine »patterns & practices«, ki je del Microsofta, a odprtokodna knjižnica.

V srednji sloj je bilo potrebno ustrezno umestiti omenjeno knjižnico, in sicer preko preprostega vmesnika. Preko tega vmesnika se dogaja vse preverjanje veljavnosti. Vmesniku podamo objekt kateremu želimo preveriti veljavnost:

```
/// <summary>
/// IValidationProvider
/// </summary>
public interface IValidationProvider
{
    /// <summary>
    /// Validates the specified object to validate.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="objectToValidate">The object to validate.</param>
    /// <returns></returns>
    ValidationResults Validate<T>(T objectToValidate);
}
```

Implementacija vmesnika:

```
/// <summary>
/// ValidationProvider
/// </summary>
public class ValidationProvider : IValidationProvider
{
    /// <summary>
    /// Validates the specified object to validate.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="objectToValidate">The object to validate.</param>
    /// <returns></returns>
    public ValidationResults Validate<T>(T objectToValidate)
    {
        ValidationFactory.SetDefaultConfigurationValidatorFactory(new SystemConfigurationSource(false));

        Validator<T> validator = ValidationFactory.CreateValidator<T>();

        ValidationResults results = new ValidationResults();

        if (objectToValidate != null)
        {
            results = validator.Validate(objectToValidate);
        }
        else
        {
            results = new ValidationResults();
            results.AddResult(new ValidationResult("Input data object is null.", null, String.Empty, String.Empty, null));
        }

        return results;
    }
}
```

Zadnja točka infrastrukture za preverjanje veljavnosti je skupni osnovni razred aplikacijskega upravitelja, kjer izpostavi funkcionalnost preverjanja veljavnosti aplikacijskim upraviteljem:

```
/// <summary>
/// Validates the specified entity.
/// </summary>
/// <param name="entity">The entity.</param>
/// <exception cref="ps.dmv.common.Exceptions.BusinessValidationException"></exception>
protected void Validate(T entity)
{
    IValidationProvider validator = ServiceLocator.Instance.Resolve<IValidationProvider>();

    ValidationResults validationResults = new ValidationResults();

    ValidationResults validationResultsEntity = validator.Validate(entity);
    validationResults.AddAllResults(validationResultsEntity);

    ValidationResults validationResultsSpecial = ValidateSpecial(entity);
    validationResults.AddAllResults(validationResultsSpecial);

    if (!validationResults.IsValid)
    {
        throw new BusinessValidationException(validationResults);
    }
}
```

Primer dejanske uporabe preverjanja veljavnosti:

```
/// <summary>
/// Updates the specified DMV calculation result.
/// </summary>
/// <param name="dmvCalculation">The DMV calculation result.</param>
/// <returns></returns>
/// <exception cref="System.NotImplementedException"></exception>
public async Task<DmvCalculationResult> Update(DmvCalculation dmvCalculation)
{
    base.Validate(dmvCalculation);

    DmvCalculation dmvCalculationFromDb = await _dmvCalculationRepository.Update(dmvCalculation);

    return new DmvCalculationResult(dmvCalculationFromDb);
}
```

Definirana pravila za preverjanje veljavnosti se nahajajo na domenski entiteti, najpogosteje kot atributi lastnostim:

```
/// <summary>
/// Gets or sets the vehicle type identifier.
/// </summary>
/// <value>
/// The vehicle type identifier.
/// </value>
[DisplayName("Vozilo")]
[Required(ErrorMessage = "{0} je obvezno polje.")]
public VehicleTypeEnum VehicleTypeId { get; set; }

/// <summary>
/// Gets or sets the fuel type identifier.
/// </summary>
/// <value>
/// The fuel type identifier.
/// </value>
[DisplayName("Gorivo")]
[Required(ErrorMessage = "{0} je obvezno polje.")]
public FuelTypeEnum FuelTypeId { get; set; }

/// <summary>
/// Gets or sets the co2 emissions value.
/// </summary>
/// <value>
/// The co2 emissions value.
/// </value>
[DisplayName("Co2 izpusti")]
[Required(ErrorMessage = "{0} je obvezno polje.")]
public short Co2EmissionsValue { get; set; }
```

Ko entiteta ni veljavna, se sproži izjema (ang. Exception). Te izjeme je potrebno ustrezno loviti; dobra praksa je, da jih lovimo na najvišjem možnem sloju. V primeru te aplikacije je to kontroler MVC:

```
try
{
    dmvCalculationResult = await ServiceLocator.Instance.Resolve<IMobileDeManager>().ImportCarData(importMobileDe);
}
catch (BusinessValidationException businessValidationException)
{
    this.ModelState.AddModelError(String.Empty, "Napaka pri uvozu. Manjkajoči podatki v avto oglasu. Poizkusite z drugim avto oglasom.");

    foreach (ValidationResult validationResult in businessValidationException.ValidationResultList)
    {
        this.ModelState.AddModelError(String.Empty, validationResult.Message);
    }

    return View(importMobileDe);
}
```

4.2.1.4 Asinhronost

Do nedavnega je bil aspekt asinhronosti v tipični .NET aplikaciji dokaj zanemarjen in v resnici še vedno je. A asinhronost je ključna za učinkovito delovanje aplikacije. S prihodom ogrodja .NET 4.5 je prišel tudi poenostavljen model asinhronosti v jeziku C#. Zdaj lahko namreč dosežemo asinhronost s preprostima ključnima besedama `async` in `await`. Prevajalnik nato naknadno doda vso potrebno asinhrono funkcionalnost v prevedeno kodo.

Primer asinhrone kode v C# prikazuje Slika 25.

```
/// <summary>
/// Updates the specified DMV calculation result.
/// </summary>
/// <param name="dmvCalculation">The DMV calculation result.</param>
/// <returns></returns>
/// <exception cref="System.NotImplementedException"></exception>
public async Task<DmvCalculationResult> Update(DmvCalculation dmvCalculation)
{
    base.Validate(dmvCalculation);

    DmvCalculation dmvCalculationFromDb = await _dmvCalculationRepository.Update(dmvCalculation);

    return new DmvCalculationResult(dmvCalculationFromDb);
}
```

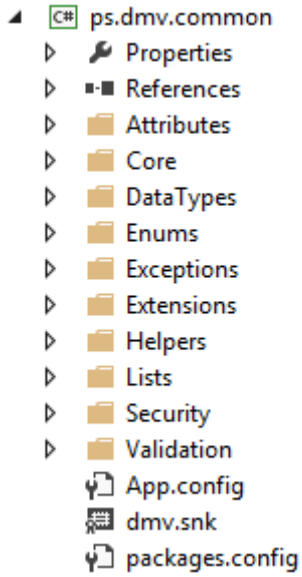
Slika 25 Primer asinhrone kode

Asinhronost je predvsem pomembna zaradi upravljanja z viri. Veliko bolje je, če je proces spletnega strežnika sproščen, kot pa da čaka, da se izvede neka operacija, s čimer morda blokira ostale spletne zahtevke.

4.2.1.5 Skupna knjižnica

Skupna knjižnica (Slika 26) je pomemben koncept in dobra praksa vsake izvorne kode. Tu se namreč nahaja vsa skupna koda, ki nima neposredne zveze z domenskim modelom oz. ni domensko specifična.

Tako lahko tu najdemo razne pomožne funkcije, kot so operacije nad podatkovnim tipom niz, potem ključna koda za preverjanje veljavnosti, varnost in ostalo.



Slika 26 Primer razporeditve kode v skupni knjižnici

4.2.1.6 Vzorec »procesor«

Vzorec procesor je primer ločitve odvisnosti v eno zaključeno enoto. Kot že ime pove, vzorec procesor procesira, obdeluje in izračunava podatke. Namesto, da imamo tako logiko razpršeno ali da kohabitira v domenskem upravitelju z ostalo kodo, jo imamo raje na enem mestu v zaključeni enoti. Razlog za tako separacijo je še en, in sicer v duhu nizke sklopljenosti lahko take procesorje »menjavamo«, ker ima seveda tudi procesor svoj vmesnik in še posebej preko vstavljene odvisnosti je to najmanjša težava, le če imamo seveda ustrezno implementacijo.

Primer implementacije procesorja iz DMV Portala, prikazuje Slika 27.

```
20      public DmvCalculation CalculateDmvAll(DmvCalculation dmvCalculation)
21      {
22          decimal co2ExhaustTaxRate = this.GetTaxRateForTheGivenCo2Exhaust(
23              dmvCalculation.Co2EmissionsValue, dmvCalculation.EnginePowerKw, dmvCalculation.FuelTypeId,
24              dmvCalculation.VehicleTypeId, dmvCalculation.AtLeastEightSeatsVehicle, dmvCalculation.EuroExhaustTypeId);
25
26          decimal euroExhaustAdditionalTaxRate = this.GetAdditionalTaxRateBasedonEuroExhaust(
27              dmvCalculation.EuroExhaustTypeId, dmvCalculation.VehicleTypeId);
28
29          decimal displacementAdditionalTaxRate = this.GetTaxRateForAdditionalDisplacementSize(
30              dmvCalculation.EngineDisplacementCcm, dmvCalculation.VehicleTypeId);
31
32          decimal specialAdditionalTaxRate = this.GetSpecialTaxRateAdditionally(
33              dmvCalculation.DieselParticlesAbove005Limit, dmvCalculation.FuelTypeId,
34              dmvCalculation.VehicleTypeId, dmvCalculation.EngineTypeId);
35
36          // Tax calculationss
37          dmvCalculation.BaseTaxRate = (double)(co2ExhaustTaxRate + euroExhaustAdditionalTaxRate + specialAdditionalTaxRate)/100;
38
39          dmvCalculation.BaseTaxRateValue = dmvCalculation.BaseTaxRate * dmvCalculation.VehicleValue;
40
41          dmvCalculation.AdditionalTaxRate = (double)displacementAdditionalTaxRate/100;
42
43          dmvCalculation.AdditionalTaxRateValue = dmvCalculation.AdditionalTaxRate * dmvCalculation.VehicleValue;
44
45          dmvCalculation.TaxTotalValue = dmvCalculation.BaseTaxRateValue + dmvCalculation.AdditionalTaxRateValue;
46
47          // Other
48          dmvCalculation.UserId = ServiceLocator.Instance.Resolve<IUserProvider>().GetCurrentUserId();
49
50          return dmvCalculation;
51      }
```

Slika 27 Primer procesorja

Opis kode s slike:Procesor dobi kot vhodni podatek objekt »DmvCalculation«, ki vsebuje vse podatke za izračun. Nato procesor uporabi te podatke in na podlagi njih računa davčne stopnje (vrstica 22-34). Same izračunane vrednosti se dodajo nazaj na osnovni objekt (vrstica 37-45). Vrstica 48 prikazuje vrinjeno odvisnost, kjer nam ogrodje vrne instanco objekta, ki zna vrniti identifikator trenutnega uporabnika.

4.2.1.7 Obrnjena odvisnost

Obrnjena odvisnost je eden od ključnih elementov moderne arhitekture, saj omogoča nizko sklopljenost, enostavno razširljivost, vzdrževanje in testiranje delov aplikacije.

Za obrnjeno odvisnost obstaja več pristopov, kot je npr. vstavljanje odvisnosti. Vstavljanje odvisnosti pa lahko poteka preko dveh glavnih mehanizmov:

- vstavljanje odvisnosti preko konstruktorja,
- vstavljanje odvisnosti preko lastnosti.

Za vstavljanje odvisnosti obstaja več ogrodij za okolje .NET. Najbolje se izkaže ogrodje Unity [42], ki prihaja, prav tako kot ogrodje za preverjanje veljavnosti, iz skupine »patterns & practices«.

V splošnem je princip v vseh ogrodjih enak. Najprej moramo registrirati odvisnost, tipično jo registriramo tako, da za dani vmesnik podamo konkretno implementacijo (Slika 28).

```
// Repositories
container.RegisterType<IDmvCalculationRepository, DmvCalculationRepository>();
container.RegisterType<IVehicleTypeRepository, VehicleTypeRepository>();
container.RegisterType<IEngineTypeRepository, EngineTypeRepository>();
container.RegisterType<IEuroExhaustTypeRepository, EuroExhaustTypeRepository>();
container.RegisterType<IFuelTypeRepository, FuelTypeRepository>();
container.RegisterType<IMobileDeRepository, MobileDeRepository>();
container.RegisterType<IStatisticsRepository, StatisticsRepository>();
```

Slika 28 Primer registracije odvisnosti

Vse, kar je še potrebno narediti, je povedati ogrodju, katero instanco vmesnika želimo (Slika 29).

```
_dmvCalculationRepository = ServiceLocator.Instance.Resolve<IDmvCalculationRepository>();
```

Slika 29 Primer zahtevka po instanci na podlagi danega vmesnika

4.2.2 Prezentacijski sloj - strežniški vzorec MVC

Prezentacijski sloj na strežniški strani je zgrajen v okviru ogrodja ASP.NET MVC. To ogrodje gradi na klasičnem ASP.NET v smislu uporabe skupnih knjižnic, podpornih mehanizmov, kot so varnost, beleženje, konfiguracija, …

4.2.2.1 ASP.NET MVC

ASP.NET MVC je bil uradno izdan leta 2009 [43]. Od takrat je postal izjemno priljubljen v razvijalski skupnosti, ki uporablja Microsoftove tehnologije. Velika prednost ASP.NET MVC-ja je v tem, da je možno praktično kateri koli del razširiti in prilagoditi svojim potrebam. Uporablja tudi princip vrinjene odvisnosti.

- **Model**

  Model v ASP.NET MVC-ju nastopa kot CRL objekt z danimi lastnostmi, ki so ustrezno dekorirani z atributi (Slika 30).

Ti atributi so eden ključnih gradnikov, saj narekujejo preverjanje veljavnosti, lokalizacijo in prezenco. Možno je napisati tudi svoje atribute in tako samo funkcionalnost še dodatno prilagoditi danemu problemu.

```
12      /// <summary>
13      /// ImportMobileDe
14      /// </summary>
15      public class ImportMobileDe
16      {
17          /// <summary>
18          /// Gets or sets the URL.
19          /// </summary>
20          /// <value>
21          /// The URL.
22          /// </value>
23          [Url(ErrorMessage = "URL je napačnega formata.")]
24          [Required(ErrorMessage = "{0} je obvezno polje")]
25          [DisplayName("URL")]
26          public string Url { get; set; }
27
28          /// <summary>
29          /// Gets or sets the vehicle type identifier.
30          /// </summary>
31          /// <value>
32          /// The vehicle type identifier.
33          /// </value>
34          [Required]
35          [DisplayName("Vrsta vozila")]
36          public VehicleTypeEnum VehicleTypeId { get; set; }
37      }
38  }
```

Slika 30 Primer enostavnega modela z atributi

Opis kode s slike: vrstica 23 prikazuje preverjanje pravilnosti spletnega naslova, vrstica 24 pa označuje, da je dana lastnost obvezna s prilagojenim sporočilom.

- **Kontroler**

Na kontroler lahko gledamo kot na orkestratorja (po funkciji podobno kot aplikacijski upravitelj). Dober kontroler tako ne vsebuje nobene podatkovne logike niti nobene logike za prikaz podatkov. Pravimo, da mora biti kontroler »tanek« [44].

Osnovna naloga kontrolerja je, da dobi oz. kreira model in ga ustrezno preda pogledu. Naslednja naloga kontrolerja je pravilno upravljanje toka za preverjanje veljavnosti in upravljanje modela (shranjevanje, osveževanje, izbris). Akcija je osnovna enota kontrolerja. Primer takega kontrolerja prikazuje Slika 31.

Vsa ostala logika, npr. poslovna logika, mora iti v sam domenski model.

```
19  public partial class ExternalImportCarDataController : BaseDmvController
20  {
21      /// <summary>
22      /// Indexes this instance.
23      /// </summary>
24      /// <returns></returns>
25      public virtual ActionResult Index()
26      {
27          ImportMobileDe importMobileDe = new ImportMobileDe();
28
29          return View(importMobileDe);
30      }
31
32      /// <summary>
33      /// Indexes the specified import mobile de.
34      /// </summary>
35      /// <param name="importMobileDe">The import mobile de.</param>
36      /// <returns></returns>
37      [HttpPost]
38      public virtual async Task<ActionResult> Index(ImportMobileDe importMobileDe)
39      {
40          if (!String.IsNullOrEmpty(importMobileDe.Url) && !importMobileDe.Url.Contains("suchen.mobile.de/auto-inserat"))
41          {
42              this.ModelState.AddModelError("Url", "URL ni iz portala mobile.de");
43          }
44
45          if (!this.ModelState.IsValid)
46          {
47              return View(importMobileDe);
48          }
49
50          DmvCalculationResult dmvCalculationResult = null;
51
52          try
53          {
54              dmvCalculationResult = await ServiceLocator.Instance.Resolve<IMobileDeManager>().ImportCarData(importMobileDe);
55          }
56          catch (BusinessValidationException businessValidationException)
57          {
58              this.ModelState.AddModelError(String.Empty, "Napaka pri uvozu. Manjkajoči podatki v avto oglasu. Poizkusite z drugim avto oglas
59
60              foreach (ValidationResult validationResult in businessValidationException.ValidationResultList)
61              {
62                  this.ModelState.AddModelError(String.Empty, validationResult.Message);//TODO add it into ifra
63              }
64
65              return View(importMobileDe);
66          }
67
68          return RedirectToAction(MVC.Dmv.DmvCalculationResult(dmvCalculationResult.DmvCalculation.Id));
69      }
70  }
```

Slika 31 Primer kontrolerja za uvoz podatkov avtomobila

Opis kode s slike: vrstica 25 prikazuje akcijo, ki se odzove na HTTP GET zahtevo in vrne pogled s praznim modelom. Vrstica 38 prikazuje akcijo, ki se odzove na HTTP POST zahtevo in preda nižjemu sloju, da izvede uvoz podatkov avtomobila (vrstica 54).

- **Pogled**

  Kontroler preda pogledu model in obratno. Naloga pogleda je ustrezno prikazati in zajeti podatke. Vsa logika, ki določa obliko in način prikaza ter interakcija z uporabnikom, sodita na pogled (Slika 32).

```
2   @using ps.dmv.common.Helpers
3   @model ImportMobileDe
4
5   @{
6       ViewBag.Title = "Uvoz vozila iz portala Mobile.de";
7   }
8
9   <h2>Uvoz vozila iz portala Mobile.de</h2>
10
11  @using (Html.BeginForm())
12  {
13      @Html.AntiForgeryToken()
14
15      <div class="form-horizontal content-background">
16          <h4><a href="http://www.mobile.de">mobile.de</a></h4>
17          <hr />
18          @Html.ValidationSummary(true)
19
20          <div class="form-group">
21              @Html.LabelFor(model => model.Url, new { @class = "control-label col-md-2" })
22              <div class="col-md-10">
23                  @Html.EditorFor(model => model.Url)
24                  @Html.ValidationMessageFor(model => model.Url)
25              </div>
26          </div>
27
28          <div class="form-group">
29              @Html.LabelFor(model => model.VehicleTypeId, new { @class = "control-label col-md-2" })
30              <div class="col-md-10">
31                  @Html.EnumDropDownListFor(model => model.VehicleTypeId)
32                  @Html.ValidationMessageFor(model => model.VehicleTypeId)
33              </div>
34          </div>
35
36          <div class="form-group">
37              <div class="col-md-offset-2 col-md-10">
38                  <input type="submit" value="Izračunaj" class="btn btn-primary btn-large" />
39              </div>
40          </div>
41      </div>
42  }
```

Slika 32 Prikaz pogleda

Opis kode s slike: vrstica 3 določuje model pogleda. Nanj se v pogledu sklicujemo z »this.Model«. Vrstica 23 prikazuje HTML-pomočnika, vrstico pod njim pa je koda, ki prikaže sporočilo, če je model za to lastnost nepravilen.

- **Razor**

  Razor je jezik za ustvarjanje predlog pogledov. Sintaksa jezika je v resnici C#. Sam jezik je zasnovan tako, da čim manj ovira mejo med programsko kodo in označevalnim jezikom HTML. Hkrati se da v njem zelo enostavno pisati.

- **T4 MVC**

  V osnovi je ASP.NET MVC zasnovan tako, da uporablja t.i. zapečene nize (ang. hardcoded) za ukaze. Takole izgleda zapečen niz v tem ogrodju:

```
return RedirectToAction("MojPogled", dmvCalculationResult.DmvCalculation.Id);
```

Takole pa izgleda s pomočjo T4 MVC pomočnika:

```
return RedirectToAction(MVC.Dmv.DmvCalculationResult(dmvCalculationResult.DmvCalculation.Id));
```

- **HTML-pomočniki**

HTML-pomočniki so izjemno močno področje ASP.NET MVC-ja. HTML-pomočniki so ključni del predloge pogleda. Namreč, ravno preko HTML-pomočnikov pogled zgenerira dejansko stran HTML.

Primer HTML-pomočnika za element »input«:

```
@Html.EditorFor(model => model.Url)
```

Tako imamo na enem mestu »logiko«, kako bi radi, da se generira HTML »input«. Seveda je možno privzete HTML-pomočnike prepisati in od njih samo dedovati in tako prilagoditi generiranje HTML-ja našim potrebam. Bistveno pri vsem tem je, da je vse skupaj na enem mestu. To je izjemno pomemben princip [45].

Primer prilagojenega HTML-pomočnika za generiranje padajočega izbirnika prikazuje Slika 33.

```
/// <summary>
/// Enums the drop down list for.
/// </summary>
/// <typeparam name="TModel">The type of the model.</typeparam>
/// <typeparam name="TEnum">The type of the enum.</typeparam>
/// <param name="htmlHelper">The HTML helper.</param>
/// <param name="expression">The expression.</param>
/// <param name="htmlAttributes">The HTML attributes.</param>
/// <returns></returns>
public static MvcHtmlString EnumDropDownListFor<TModel, TEnum>(this HtmlHelper<TModel> htmlHelper, Expression<Func<TModel, TEnum>> expression, object htmlAttributes)
{
    ModelMetadata metadata = ModelMetadata.FromLambdaExpression(expression, htmlHelper.ViewData);
    Type enumType = GetNonNullableModelType(metadata);
    IEnumerable<TEnum> values = Enum.GetValues(enumType).Cast<TEnum>();
    SelectListItem[] singleEmptyItem = new[] { new SelectListItem { Text = "", Value = "" } };

    IEnumerable<SelectListItem> items = from value in values
                                        select new SelectListItem
                                        {
                                            Text = GetEnumDescription(value),
                                            Value = value.ToString(),
                                            Selected = value.Equals(metadata.Model)
                                        };

    // If the enum is nullable, add an 'empty' item to the collection
    if (metadata.IsNullableValueType)
    {
        items = singleEmptyItem.Concat(items);
    }

    return htmlHelper.DropDownListFor(expression, items, htmlAttributes);
}
```

Slika 33 Prikaz prilagojenega HTML pomočnika

- **Filtri akcij**

Filtri akcij so naslednji pomemben koncept v ASP.NET MVC ogrodju. Akcije v kontrolerju je mogoče dekorirati s filtri akcij (ang. Action filter) z atributi. To je eleganten pristop, če želimo dodati ali pa odvzeti funkcionalnost osnovni akciji.

Primer filtrov akcij so filtri akcij za avtentifikacijo, za filtriranje akcij HTTP (npr. dovolimo samo akcije POST HTTP na akciji:

```
[HttpPost]
public virtual async Task<ActionResult> Index(ImportMobileDe importMobileDe)
{
```

- **Potek preverjanja veljavnosti vhodov**

Preverjanje veljavnosti je pomemben element interakcije z uporabnikom in seveda nujen element za zagotavljanje konsistentnih podatkov. V ASP.NET MVC-ju preverjanje veljavnosti poteka preko že omenjenih atributov za preverjanje veljavnosti na modelu, osnova v ogrodju pa je objekt »ModelState«. ModelState vsebuje vse morebitne napake preverjanja veljavnosti, saj vsak objekt avtomatsko preveri njegovo veljavnost, preden ga ogrodje preda sami akciji.

Lastnost modela s preverjanjem veljavnosti:

```
[Url(ErrorMessage = "URL je napačnega formata.")]
[Required(ErrorMessage = "{0} je obvezno polje")]
[DisplayName("URL")]
public string Url { get; set; }
```

Upravljanje toka za preverjanje veljavnosti v akciji kontrolerja:

```
if (!this.ModelState.IsValid)
{
    return View(importMobileDe);
}
```

Če je preverjanje veljavnosti uspešno, gre programski tok naprej; tipično se dani objekt shrani. V primeru, da je preverjanje veljavnosti neuspešno, pa se status preko že omenjenega objekta ModelState prenese tudi na pogled. Na pogledu se mora nahajati tudi označevalnik za prikaz preverjanja veljavnosti:

```
@Html.ValidationMessageFor(model => model.VehicleTypeId)
```

- **Post/Redirect/Get (PRG) vzorec**

Pomemben koncept za spletne aplikacije je t.i. PRG vzorec [46]. Ta preprosto narekuje, da je potrebno po akciji POST obvezno sprožiti ukaz »redirect«, sicer se lahko zgodi, da lahko odjemalec ponovno pošlje akcijo POST na strežnik, če uporabnik osveži stran (F5). Za ta primer imajo varovalko tudi spletni brskalniki, kar prikazuje Slika 34.

Slika 34 Opozorilo brskalnika pred ponovno akcijo POST

Da se temu izognemo, je potrebno v akciji POST sprožiti ukaz za preusmeritev:

```
return RedirectToAction(MVC.Dmv.DmvCalculationResult(dmvCalculationResult.DmvCalculation.Id));
```

4.2.2.2 WebAPI

WebAPI je Microsoftova tehnologija REST, ki bazira na filozofiji ASP.NET MVC. Kot knjižnica pa je zgrajena na novo. REST API je danes ključnega pomena pri gradnji modernih aplikacij, še posebej, če uporabljamo aplikacijo na odjemalski strani ali če želimo omogočiti povezljivost z drugimi sistemi. Na REST API je potrebno gledati kot na prezentacijski sloj za ostale sisteme, zato je zelo pomembno, kako je zasnovan. Predvsem je potrebno striktno slediti zapovedim gradnje REST vmesnikov.

Tipično REST API vrača podatkovni format JSON, ki je tudi najbolj razširjen na odjemalski strani spleta. JSON so serializirani JavaScript objekti.

Primer WebAPI kontrolerja prikazuje Slika 35. Posamezne funkcije kontrolerja pripadajo posameznim HTTP akcijam. Spoznamo jih preprosto po imenu.

```
/// <summary>
/// ApiDmvCalculationController
/// </summary>
public class ApiDmvCalculationController : BaseApiDmvController
{
    private IDmvCalculationManager _dmvCalculationManager = null;

    /// <summary>
    /// Initializes a new instance of the <see cref="ApiDmvCalculationController"/> class.
    /// </summary>
    public ApiDmvCalculationController()
    {
        _dmvCalculationManager = ServiceLocator.Instance.Resolve<IDmvCalculationManager>();
    }

    /// <summary>
    /// Gets the specified page size.
    /// </summary>
    /// <param name="pageSize">Size of the page.</param>
    /// <param name="pageIndex">Index of the page.</param>
    /// <param name="includeImportedCalculation"></param>
    /// <returns></returns>
    public PagedList<DmvCalculation> Get([FromUri]int pageSize, [FromUri]int pageIndex, [FromUri]bool includeImportedCalculation)
    {
        PagedList<DmvCalculation> dmvCalculations = _dmvCalculationManager.GetAll(pageSize, pageIndex, includeImportedCalculation);

        return dmvCalculations;
    }

    /// <summary>
    /// Gets the specified identifier.
    /// </summary>
    /// <param name="id">The identifier.</param>
    /// <returns></returns>
    public DmvCalculationResult Get(int id)
    {
        return _dmvCalculationManager.Get(id);
    }

    /// <summary>
    /// Posts the specified DMV calculation.
    /// </summary>
    /// <param name="dmvCalculation">The DMV calculation.</param>
    /// <returns></returns>
    public async Task<HttpResponseMessage> Post(DmvCalculation dmvCalculation)
    {
        DmvCalculationResult dmvCalculationResult = await _dmvCalculationManager.ProcessDmvTaxValueResult(dmvCalculation);

        HttpResponseMessage httpResponseMessage = base.GetHttpResponseMessage(dmvCalculationResult);

        return httpResponseMessage;
    }
```

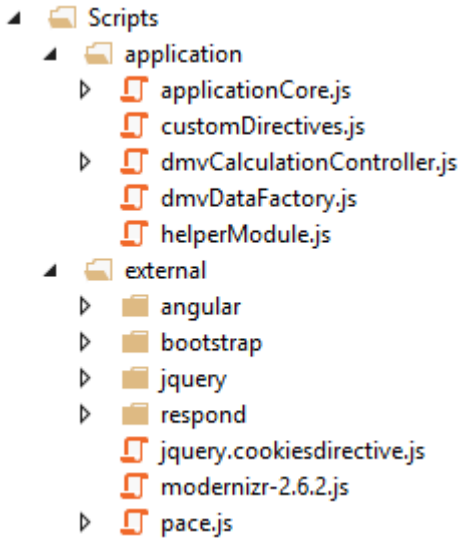
Slika 35 Primer REST api končne točke v WebAPI-ju

4.2.3 Prezentacijski sloj - odjemalski vzorec MVC

Prezentacijski sloj na odjemalski strani je zgrajen okoli ogrodja AngularJS [47]. To ogrodje je namenjeno gradnji spletnih aplikacij na eni strani (SPA), izbrano pa je bilo na podlagi celovitosti ogrodja in dobre medsebojne integracije. Tako ogrodje vsebuje: vezenje podatkov, MVC vzorec, preusmerjanje, testiranje delov, jqLite, vzorce, zgodovino, tovarne, kontrolerje, poglede, direktive, storitve, vrinjeno odvisnost in preverjanje veljavnosti [48].

Zadnji razlog za naš izbor je, da je SPA ogrodje trenutno tudi najbolj tržno zanimivo. Največ služb je na voljo, kjer je zahtevano kakšno od celovitih SPA ogrodij [49].

Pomembno je, da je JavaScript koda ustrezno urejena. Predlog ureditve prikazuje Slika 36.



Slika 36 Predlagana ureditev JavaScript kode v projektu

AngularJS ogrodje se trudi biti kar se da deklarativno, saj je tak pristop bolj primeren za gradnjo uporabniškega vmesnika.

- **Model**

  Model je determiniran, z modelom strežniške strani, če se želimo ogniti dodatnemu mapiranju. Ker gre za model v jeziku JavaScript, ne obstaja deklaracija objekta, ampak se le-ta definira dinamično.

  Zato tudi dinamično deklariran primer modela:

```
$scope.dmvCalculation = {};
$scope.dataLoaded = false;
$scope.onlyNumbers = /^\d+$/;

function init() {

    $scope.dmvCalculation.init = logger.log('DMV form initialized.');

    $scope.dmvCalculation.DateOfCalculation = date.getDateNow();
}

// Data fetch
dataFetch();

function dataFetch() {

    dmvDataFactory.getVehicleTypeCodeTable().then(function (result) {
        init();

        $scope.dmvCalculation.vehicleTypeCodeTable = result;

        $scope.dmvCalculation.VehicleTypeId = $scope.dmvCalculation.vehicleTypeCodeTable[0];

        $scope.dataLoaded = true; //TODO: hack make it for all calls
    });

    dmvDataFactory.getEngineTypeCodeTable().then(function (result) {
        $scope.dmvCalculation.engineTypeCodeTable = result;

        $scope.dmvCalculation.EngineTypeId = $scope.dmvCalculation.engineTypeCodeTable[0];
    });

    dmvDataFactory.getFuelTypeCodeTable().then(function (result) {
        $scope.dmvCalculation.fuelTypeCodeTable = result;
    });

    dmvDataFactory.getEuroExhaustTypeCodeTable().then(function (result) {
        $scope.dmvCalculation.euroExhaustTypeCodeTable = result;
    });
}
```

Model je povezan s pogledom preko AngularJS objekta »$scope«.

- **Kontroler**

Kontroler je v AngularJS tudi osrednji orkestrator. Orkestrira podatke, inicializacijo »$scope« objekta. Tudi za ta SPA kontroler velja pravilo, da naj bo čim bolj tanek. Kot pomoč temu obstajata dva mehanizma, to sta princip tovarne in vstavljene odvisnosti v tem ogrodju.

Primer AngularJS kontrolerja prikazuje Slika 37.

```
3   controllers.dmvCalculationController = ['$scope', '$http', 'logger', 'dmvDataFactory', 'date', '$location',
4           function ($scope, $http, logger, dmvDataFactory, date, $location) {
5
6               // Init
7               $scope.dmvCalculation = {};
8               $scope.dataLoaded = false;
9               $scope.onlyNumbers = /^\d+$/;
10
11              function init() {
12
13                  $scope.dmvCalculation.init = logger.log('DMV form initialized.');
14
15                  $scope.dmvCalculation.DateOfCalculation = date.getDateNow();
16              }
17
18              // Data fetch
19              dataFetch();
20
21              function dataFetch() {
22
23                  dmvDataFactory.getVehicleTypeCodeTable().then(function (result) {
24                      init();
25
26                      $scope.dmvCalculation.vehicleTypeCodeTable = result;
27
28                      $scope.dmvCalculation.VehicleTypeId = $scope.dmvCalculation.vehicleTypeCodeTable[0];
29
30                      $scope.dataLoaded = true; //TODO: hack make it for all calls
31                  });
32
33                  dmvDataFactory.getEngineTypeCodeTable().then(function (result) {
34                      $scope.dmvCalculation.engineTypeCodeTable = result;
35
36                      $scope.dmvCalculation.EngineTypeId = $scope.dmvCalculation.engineTypeCodeTable[0];
37                  });
38
39                  dmvDataFactory.getFuelTypeCodeTable().then(function (result) {
40                      $scope.dmvCalculation.fuelTypeCodeTable = result;
41                  });
42
43                  dmvDataFactory.getEuroExhaustTypeCodeTable().then(function (result) {
44                      $scope.dmvCalculation.euroExhaustTypeCodeTable = result;
45                  });
46              }
47
48              // Actions
49              actions();
50
51              function actions() {
52                  $scope.processDmvCalculation = function (dmvCalculationData) {
53
54                      if (dmvCalculationData.$valid) {
55
56                          //Data Process
57                          dmvCalculationData.VehicleTypeId = dmvCalculationData.VehicleTypeId.id;
58                          dmvCalculationData.EuroExhaustTypeId = dmvCalculationData.EuroExhaustTypeId.id;
59                          dmvCalculationData.EngineTypeId = dmvCalculationData.EngineTypeId.id;
60                          dmvCalculationData.FuelTypeId = dmvCalculationData.FuelTypeId.id;
61
62                          //POST data
63                          dmvDataFactory.postDmvData(dmvCalculationData).then(function (result) {
64
65                              if (result.status === 200)
```

Slika 37 Primer AngularJS kontrolerja

Opis kode s slike: vrstica 3 prikazuje inicializacijo kontrolerja, skupaj z objekti, ki so obrnjeno odvisni. Vrstica 9 prikazuje definicijo regularnega izraza, ki se potem nanj sklicujemo na pogledu v okviru AngularJS direktive. Vrstica 28 prikazuje, kako lahko v AngularJS inicializiramo podatek modela. Vrstica 52 prikazuje funkcijo na katero se

tudi sklicujemo s pogleda. Povezana je na dogodek potrditve forme. Vrstica 54 je nujna, saj je vedno potrebno preveriti pravilnost modela. Vrstica 63 prikazuje asinhroni klic na REST API – HTTP POST zahtevek.

- **Pogled**

Ključen objekt v trojčku Model-Kontroler-Pogled je »$scope. Ta je nosilec modela in hkrati vez med kontrolerjem in pogledom. Pogled je, podobno kot na strežniškem pogledu, predloga. Pogled v AngularJS se implementira tako, da se uporablja razširjen HTML z AngularJS atributi in direktivami.

Najprej moramo ustvariti kontekst pogleda, to storimo takole:

```
<div ng-controller="dmvCalculationController">
```

Pri tem je atribut »ng-controller« enak imenu dejanskega deklariranega kontrolerja v JavaScriptu:

```
controllers.dmvCalculationController = ['$scc
```

Primer avtomatskega vezenja, najprej na strani kontrolerja:

```
$scope.dmvCalculation.DateOfCalculation = date.getDateNow();
```

Potem še na strani pogleda:

```
ng-model="dmvCalculation.DateOfCalculation"
```

Lahko vidimo, da se objekt »$scope« uporablja v kontrolerju eksplicitno, prav tako je prisoten v pogledu, le da implicitno, saj ogrodje poskrbi za sam podatkovni kontekst.

Druga možnost avtomatskega vezenja pa je uporaba naslednje notacije:

```
{{ dmvCalculation.DateOfCalculation }}
```

Naslednji primer lepo pokaže ekspresivno moč AngularJS. Takole se kreira padajoč izbirnik HTML »select«:

```
<select name="vehicleTypeId"
        ng-model="dmvCalculation.VehicleTypeId"
        required class="form-control"
        ng-options="vehicle.name for vehicle in dmvCalculation.vehicleTypeCodeTable"></select>
```

Zanimiv je atribut »ng-options«, čigar vsebina generira HTML elemente »option«.

- **Potek preverjanja veljavnosti vhodov**

  Preverjanje veljavnosti poteka v AngularJS predvsem implicitno in deklarativno, vse preko atributov HTML (zanimivo podobnost lahko najdemo z atributi na strani strežniškega MVC-ja; čeprav gre za tehnično povsem drugačni rešitvi, sta si koncepta precej blizu).

  Preverjanje veljavnosti na ravni posameznega kontrolnika najprej deklariramo z direktivo:

```
required
```

  Rezultat preverjanja veljavnosti prikažemo s pomočjo kode:

```
<span class="error" ng-show="dmvCalculation.co2EmissionsValue.$error.required">*</span>
```

  Celota:

```
<input type="text" name="co2EmissionsValue" ng-pattern="onlyNumbers" ng-model="dmvCalculation.Co2EmissionsValue" required />
<span class="error" ng-show="dmvCalculation.co2EmissionsValue.$error.required">*</span>
<span class="error" ng-show="dmvCalculation.co2EmissionsValue.$error.pattern">**</span>
```

- **Direktive**

  Direktive [50] smo že omenili. So izjemno močen koncept v AngularJS. Gre namreč za koncept za razširitev nabora elementov, ki jih v osnovi podpira HTML. Brati jih seveda zna samo AngularJS. Obstaja cel nabor teh direktiv, možno je pisati tudi svoje, prilagojene direktive. Direktive bi lahko primerjali s HTML-pomočniki v ASP.NET MVC.

  Primer prilagojene direktive za datumski zbirnik:

```
// Date picker
app.directive('jqdatepicker', function () {
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function (scope, element, attrs, ngModelCtrl) {
            element.datepicker({
                dateFormat: 'm-d-yy',
                onSelect: function (date) {
                    scope.dmvCalculation.DateOfCalculation = date;
                    scope.$apply();
                }
            });
        }
    };
});
```

Opis kode s slike: vrstica 4 predstavlja inicializacijo direktive, kjer tudi definiramo njeno ime. V vrstici 6 definiramo tip direktive (A – atribut, E – element, C – CSS razred in M – komentar). Vrstica 7 predstavlja odvisnost na ostale direktive. V vrstici 8-16 se nahaja definicija povezave z DOM-om in »$scope« objektom. V vrstici 9 se nahaja definicija dane funkcionalnosti, ki jo želimo, da jo direktiva podpira. V našem primeru uporabljamo jQuery razširitev »datepicker«.

Uporaba prilagojene direktive:

```
<input name="dateOfCalculation" type="text" ng-model="dmvCalculation.DateOfCalculation" required class="form-control" jqdatepicker />
```

Pri tej direktivi lahko vidimo, da je popolnoma klasičen HTML element »input« pretvorila v datumski zbirnik (Slika 38).

Slika 38 Datumski zbirnik na podlagi direktive

- **Tovarna**

Razvijalski vzorec tovarne je zelo učinkovit pristop razdelitve pristojnosti dela aplikacije na različne module oz. sloje. Naloga tovarne je produkcija objektov. V AngularJS se ta koncept uporablja prav v ta namen, torej da se kontroler razbremeni s kreacijo modelov oz. pridobivanja podatkov za model.

Tovarna:

```
app.factory('dmvDataFactory', ['$http', '$q', 'logger',
    function ($http, $q, logger) {

    // Init
    var dataFactory = {};

    // Data definition

    // GET
    get();

    function get() {
        dataFactory.getVehicleTypeCodeTable = function () {
            var deferred = $q.defer();
            $http.get("api/ApiCodeTable?codeTableType=VehicleType").success(function (data) {
                deferred.resolve(data);
            });
            return deferred.promise;
        };

        dataFactory.getEngineTypeCodeTable = function () {
            var deferred = $q.defer();
            $http.get("api/ApiCodeTable?codeTableType=EngineType").success(function (data) {
                deferred.resolve(data);
            });
            return deferred.promise;
        };

        dataFactory.getFuelTypeCodeTable = function () {
            var deferred = $q.defer();
            $http.get("api/ApiCodeTable?codeTableType=FuelType").success(function (data) {
                deferred.resolve(data);
            });
            return deferred.promise;
        };

        dataFactory.getEuroExhaustTypeCodeTable = function () {
            var deferred = $q.defer();
            $http.get("api/ApiCodeTable?codeTableType=EuroExhaustType").success(function (data) {
                deferred.resolve(data);
            });
            return deferred.promise;
        };
    }
```

Ker v tem primeru tovarna kliče REST API na strežniku za pridobitev podatkov, je potrebno to asinhronost ustrezno upravljati.

Primer uporabe tovarne:

```
dmvDataFactory.getEngineTypeCodeTable().then(function (result) {
    $scope.dmvCalculation.engineTypeCodeTable = result;

    $scope.dmvCalculation.EngineTypeId = $scope.dmvCalculation.engineTypeCodeTable[0];
});
```

- **Vzorec obljube**

Vzorec obljube [51] prestavlja v JavaScriptu objekt, ki predstavlja čakajoč rezultat asinhrone operacije. Obstaja veliko implementacij vzorca obljube, vzorec je

implementiran tudi v AngularJS-u, zato je priporočljivo, da se ob uporabi tega ogrodja uporabi tudi ta implementacija obljube.

Kako se uporabi ta vzorec? Najprej inicializiramo objekt obljube:

```
var deferred = $q.defer();
```

Nato izvedemo samo asinhrono operacijo:

```
$http.get("api/ApiCodeTable?codeTableType=VehicleType").success(function (data) {
    deferred.resolve(data);
});
```

Na klicu, ki se izvede ob končanju operacije, napolnimo obljubo z dejanskim rezultatom. Še prej se vrne prazen inicializiran objekt obljube:

```
return deferred.promise;
```

Funkcijo, ki implementira vzorec obljube, kličemo na sledeč način:

```
dmvDataFactory.getEngineTypeCodeTable().then(function (result) {
    $scope.dmvCalculation.engineTypeCodeTable = result;

    $scope.dmvCalculation.EngineTypeId = $scope.dmvCalculation.engineTypeCodeTable[0];
});
```

Funkcija »then« se izvede, ko se kliče na objektu obljube funkcija »resolve()«.

- **Vstavljena odvisnost**

AngularJS ogrodje je zgrajeno z mislijo na vstavljeno odvisnost. Vsaka inicializacija modula poteka preko vstavljene odvisnosti. Dober primer tega je že sam kontroler:

```
controllers.dmvCalculationController = ['$scope', '$http', 'logger', 'dmvDataFactory', 'date', '$location',
```

Vsi ti objekti so potem v samem kontrolerju inicializirani in pripravljeni za uporabo. Na primer, tudi objekt »$scope«, ki ga že poznamo, se inicializira na tak način.

- **Testiranje delov**

Testiranje delov je tudi ključnega pomena v SPA aplikacijah. Še toliko bolj je to pomembno tukaj, saj zaradi narave jezika JavaScript nimamo zagotovljene varnosti tipov (ang. Type Safety), saj se jezik ne prevaja. Zaradi same zasnove AngularJS glede vstavljene odvisnosti, je tako testiranje enostavno izvesti v okviru AngularJS-a.

4.2.4 Oblikovanje

Na oblikovni strani vsebuje aplikacija CSS ogrodje Bootstrap [52]. Bootstrap je klasično CSS ogrodje, v katerem oblikujemo stran s pomočjo nekaj standardnih gradnikov. Kot primer, za

urejanje delov strani (angl. panel) ima ogrodje Bootstrap sistem, v katerem mora biti seštevek posamezne vertikale razredov CSS enak 12. Če uporabimo razred `col-md-4`, lahko uporabimo tri take razrede:

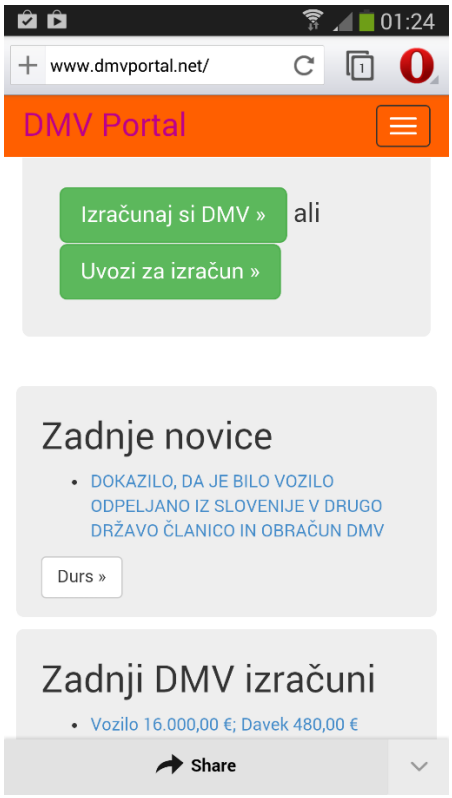
```
<div class="col-md-4 content-background fixed-md-4">

    <h2>Zadnje novice</h2>
    <p>
        <ul>
            <li><a href="http://www.durs.gov.si/si/dav
        </ul>
    </p>
    <p><a href="http://www.durs.gov.si/si/davki_predpi
</div>
<div class="col-md-4 content-background fixed-md-4">
    @Html.Action(MVC.Home.LatestDmvCalculations(DmvCon
</div>
<div class="col-md-4 content-background fixed-md-4">
    @Html.Action(MVC.Home.LatestMobileDeCalculations(D
</div>
```

Nato dobimo rezultat, ki ga prikazuje Slika 39.

Slika 39 Prikaz uporabe Bootstrap ogrodja

Ogrodje Bootstrap podpira odzivno spletno oblikovanje (ang. responsive web design), tako, lahko gledamo portal tudi na manjših mobilnih napravah (Slika 40).



Slika 40 Izgled DMV Portala na mobilni napravi

Za barve portala smo uporabili krožno barvno shemo (Slika 41).

Slika 41 Krožna barvna shema za določitev barv strani

4.2.5 Priprava aplikacije za produkcijsko okolje

Ko je spletna aplikacija implementirana, še ni vse končano. Če jo želimo spraviti v produkcijsko okolje, je potrebno opraviti še nekaj nujnih posegov:

- **Google analitika**

  Ko je aplikacija v produkciji, je pomembno vedeti, koliko se uporablja. Zato je potrebno v aplikacijo vključiti sledilno kodo, ki omogoča detajlno spremljanje dogajanja na strani preko Google Analytics portala [53].

- **Vklop minifikacije in kompresije skript**

  To omogoča ASP.NET MVC ogrodje. Vse skripte je potrebno programsko registrirati in potem optimizacija deluje avtomatsko:

```
public static void RegisterBundles(BundleCollection bundles)
{
    // JS
    /////

    // External
    bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                "~/Scripts/external/jquery/jquery-{version}.js",
                "~/Scripts/external/jquery/jquery-ui-{version}.js"
                ));

    bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                "~/Scripts/external/jquery/jquery.validate*"
                ));

    bundles.Add(new ScriptBundle("~/bundles/bootstrap").Include(
                "~/Scripts/external/bootstrap/bootstrap.js"
                ));

    // Use the development version of Modernizr to develop with and learn
    // ready for production, use the build tool at http://modernizr.com tc
    bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
                "~/Scripts/external/modernizr-*"
                ));

    bundles.Add(new ScriptBundle("~/bundles/external").Include(
                "~/Scripts/external/pace.js",
                "~/Scripts/external/jquery.cookiesdirective.js"
                ));

    bundles.Add(new ScriptBundle("~/bundles/angular").Include(
              "~/Scripts/external/angular/angular.js",
              "~/Scripts/external/angular/angular-route.js",
              "~/Scripts/external/angular/i18n/angular-locale_sl-si.js"
              ));
```

- **Uporaba diagnostičnih orodij**

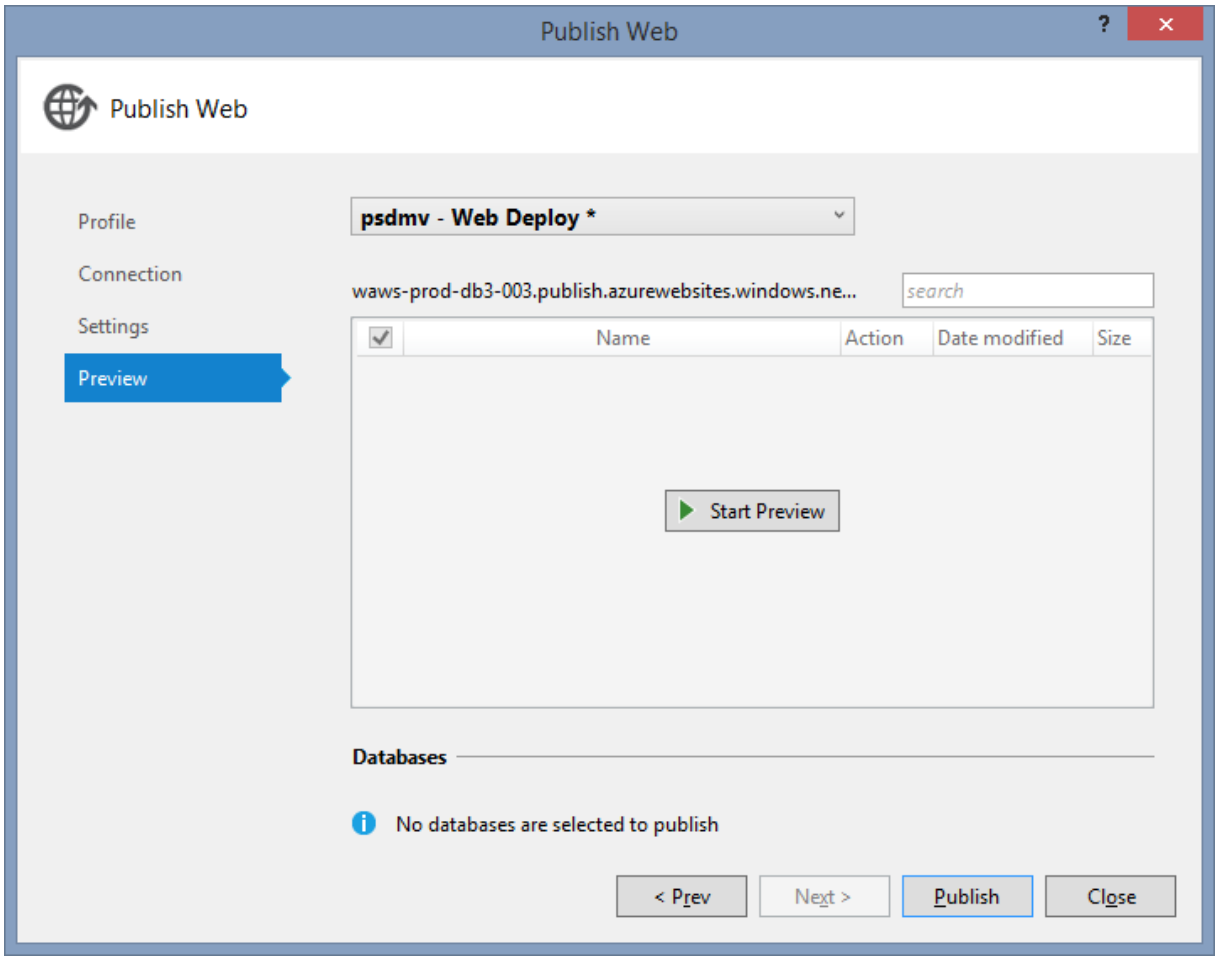
  Za ASP.NET MVC obstajata dve nepogrešljivi orodji, in sicer Elmah [54] za beleženje napak v strežniku in Glimpse [55] za analizo izvajanja zahtevka spletne aplikacije (Slika 42).

Slika 42 Primer orodja Glimpse v akciji

4.2.6 Oblačno okolje in namestitev aplikacije

Zadnje dejanje objave spletne aplikacije širnemu svetu je sama namestitev aplikacije na »produkcijski« strežnik. Dandanes postaja uporaba oblaka vedno bolj popularna in tudi dostopna, tako da si bomo na kratko pogledali kako poteka namestitev aplikacije v oblak.

Pogledali si bomo, kako to poteka z uporabo Visual Studia 2013 in Azura. Predpogoj za tako namestitev je račun na oblačnem sistemu Azure. Nato je sama namestitev aplikacije mogoča kar preko Visual Studia. Najprej uvozimo potrebne podatke za namestitev iz portala Azure, nato pa preko spletnega projekta le zaženemo namestitev, kar prikazuje Slika 43.



Slika 43 Dialog za namestitev aplikacije na Azure

Ko kliknemo »Publish«, je aplikacija v nekaj trenutkih nameščena in dostopna uporabnikom.

Azure nam tudi omogoča spremljanje ključnih virov, ki jih uporablja aplikacija (Slika 44).

Slika 44 Uporaba virov, ko je šla aplikacija v produkcijo

5. Zaključek

Pri razvoju spletnih aplikacij se srečujemo z mnogo možnostmi, začenši že z množico programskih jezikov, ki se potem nadaljuje v izbiro pravega ogrodja, pravih orodij in pripomočkov. Izbira je skoraj neskončna.

Hkrati nastajajo nove dileme o tem, kakšno arhitekturno pot ubrati, kako posamezne module, sloje pravilno umestiti in kako jih povezati med seboj.

Cilj tega diplomskega dela je pomagati reševati te dileme in pokazati eno od možnih poti. Ta nikakor ni edina pravilna in mogoča, a je dokazljivo uspešna ne samo zaradi zgrajenega in delujočega primera v produkciji [56], temveč zaradi razširjenosti uporabe. Potrebno se je zavedati in upoštevati vse dejavnike pri razvoju take aplikacije, to bi bila morda zanimiva tema za nadaljnje raziskovanje.

Najpomembnejša lastnost aplikacije je konsistentnost: tudi če izbrana rešitev ni najbolj optimalna, je veliko pomembnejše, da ostanemo v okviru aplikacije, arhitekture čim bolj konsistentni. Naslednja po pomembnosti je nizka sklopljenost aplikacije, striktna uporaba vmesnikov in uporaba vrinjene odvisnosti.

Izjemno pomembno je slediti razdeljenim odgovornostim, kar razvijalski vzorec MVC tako lepo nakaže in ob enem zahteva. Nikdar ne smemo mešati odgovornosti med različnimi sloji ali deli aplikacije.

V resnici ni splošno pravilnega odgovora, kateri pristop je boljši – odjemalski ali strežniški MVC –, saj sta oba pristopa ustrezna in moderna. Odločitev za pristop oz. za razmerje med njima je popolnoma odvisno od okolja aplikacije in njenih zahtev. Temu se mora aplikacija tudi prilagoditi. Recimo, da imamo skupino .NET razvijalcev, ki ima šibko znanje JavaScripta; že samo na podlagi tega bi se lahko odločili, da večino aplikacije naredimo v ASP.NET MVC. Po drugi strani pa si lahko zamislimo primer, kjer je sama aplikacija dokaj majhna (recimo malo obrazcev), hkrati pa imamo zahtevo, naj bo aplikacija visoko performančna in skalabilna; tukaj bi bilo potrebno resno razmisliti o polnem SPA pristopu.

Kot smo ugotovili tekom dela, so vsi prezentacijski vzorci derivat vzorca MVC, tako da bi bilo smiselno začeti uporabljati termin MV* (* kot kar koli), saj je bistvo vseh teh vzorcev enako – potrebno je ločiti odgovornosti.

Literatura

[1] History of the World Wide Web. Dostopno na: http://en.wikipedia.org/wiki/History_of_the_World_Wide_Web.

[2] (1991) The Original HTTP as defined in 1991. Dostopno na: http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[3] HTML Tags. Dostopno na: http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html.

[4] (1999) Hypertext Transfer Protocol -- HTTP/1.1. Dostopno na: http://tools.ietf.org/html/rfc2616.

[5] HTML: The Markup Language. Dostopno na: http://dev.w3.org/html5/markup/references.html.

[6] Common Gateway Interface. Dostopno na: http://en.wikipedia.org/wiki/Common_Gateway_Interface.

[7] JavaScript. Dostopno na: http://en.wikipedia.org/wiki/JavaScript.

[8] Is Javascript a Functional Programming Language? Dostopno na: http://stackoverflow.com/questions/3962604/is-javascript-a-functional-programming-language.

[9] Node.js. Dostopno na: http://en.wikipedia.org/wiki/Nodejs.

[10] JavaScript is Web Assembly Language and that's OK. Dostopno na: http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx.

[11] Solution stack. Dostopno na: http://en.wikipedia.org/wiki/Solution_stack.

[12] (2012) XMLHttpRequest. Dostopno na: http://www.w3.org/TR/XMLHttpRequest/.

[13] (2011) The WebSocket Protocol. Dostopno na: http://tools.ietf.org/html/rfc6455.

[14] TodoMVC. Dostopno na: http://todomvc.com/.

[15] Active Server Pages. Dostopno na: http://en.wikipedia.org/wiki/Active_Server_Pages.

[16] (2002) The Law of Leaky Abstractions. Dostopno na: http://www.joelonsoftware.com/articles/leakyabstractions.html.

[17] Dynamic web page. Dostopno na: http://en.wikipedia.org/wiki/Dynamic_web_page.

[18] (2013) SPA and the Single Page Myth. Dostopno na: http://www.johnpapa.net/pageinspa.

[19] The dynamic stylesheet language. Dostopno na: http://lesscss.org/.

[20] Web application. Dostopno na: http://en.wikipedia.org/wiki/Web_applications.

[21] Web Applications. Dostopno na: http://webtrends.about.com/od/webapplications/a/web_application.htm.

[22] Web application framework. Dostopno na: http://en.wikipedia.org/wiki/Web_application_framework.

[23] Nodejs.org. Dostopno na: http://nodejs.org/.

[24] (2011) Fundamental MVVM. Dostopno na: http://visualstudiomagazine.com/articles/2011/08/15/fundamental-mvvm.aspx.

[25] Model–view–controller. Dostopno na: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.

[26] Model View Controller History. Dostopno na: http://c2.com/cgi/wiki?ModelViewControllerHistory.

[27] Model–view–presenter. Dostopno na: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter.

[28] (2012) Implementing the MVVM Pattern. Dostopno na: http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx.

[29] (2004) Web Services Architecture. Dostopno na: http://www.w3.org/TR/ws-arch/#relwwwrest.

[30] Single-page application. Dostopno na: http://en.wikipedia.org/wiki/Single-page_application.

[31] (2010) ZDMV-C. Dostopno na: http://www.uradni-

list.si/1/objava.jsp?urlid=20109&stevilka=314.

[32] Mobile.de. Dostopno na: http://www.mobile.de/.

[33] DmvCalculator GitHub. Dostopno na: https://github.com/PeterStegnar/dmvcalculator.

[34] SOLID (object-oriented design). Dostopno na: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

[35] Repository. Dostopno na: http://martinfowler.com/eaaCatalog/repository.html.

[36] The Repository Pattern. Dostopno na: http://msdn.microsoft.com/en-us/library/ff649690.aspx.

[37] Domain-driven design. Dostopno na: http://en.wikipedia.org/wiki/Domain-driven_design.

[38] (2009) An Introduction To Domain-Driven Design. Dostopno na: http://msdn.microsoft.com/en-us/magazine/dd419654.aspx.

[39] AutoMapper. Dostopno na: http://automapper.codeplex.com/.

[40] NUnit. Dostopno na: http://www.nunit.org/.

[41] (2007) Validation Application Block. Dostopno na: http://msdn.microsoft.com/en-us/library/ff648831.aspx.

[42] patterns & practices - Unity. Dostopno na: https://unity.codeplex.com/.

[43] ASP.NET MVC Framework. Dostopno na: http://en.wikipedia.org/wiki/ASP.NET_MVC_Framework.

[44] (2008) MVC: How to write controllers. Dostopno na: http://andrzejonsoftware.blogspot.com/2008/07/mvc-how-to-write-controllers.html.

[45] Don't repeat yourself. Dostopno na: http://en.wikipedia.org/wiki/Don't_repeat_yourself.

[46] Post/Redirect/Get. Dostopno na: http://en.wikipedia.org/wiki/Post/Redirect/Get.

[47] AngularJS. Dostopno na: http://angularjs.org/.

[48] (2013) AngularJS in 60-ish Minutes – The eBook. Dostopno na: http://weblogs.asp.net/dwahlin/archive/2013/07/30/angularjs-in-60-ish-minutes-the-

ebook.aspx.

[49] Careers 2.0 - jobs for “angularjs”. Dostopno na: http://careers.stackoverflow.com/jobs?searchTerm=angularjs.

[50] Creating Custom Directives. Dostopno na: http://docs.angularjs.org/guide/directive.

[51] (2013) JavascriptPromise. Dostopno na: http://martinfowler.com/bliki/JavascriptPromise.html.

[52] Bootstrap. Dostopno na: http://getbootstrap.com/.

[53] Google Analytics. Dostopno na: https://www.google.com/analytics/.

[54] elmah - Error Logging Modules and Handlers for ASP.NET. Dostopno na: https://code.google.com/p/elmah/.

[55] Glimpse. Dostopno na: http://getglimpse.com/.

[56] Portal za izračun DMV. Dostopno na: http://www.dmvportal.net/.