Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Rožman

Primerjava razvojnih modelov .NET MVC in .NET Web Forms

diplomsko delo

univerzitetni študij računalništva in informatike

Ljubljana, 2016

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Rožman

Primerjava razvojnih modelov .NET MVC in .NET Web Forms

diplomsko delo

univerzitetni študij računalništva in informatike

Mentor: doc. dr. Mojca Ciglarič

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Na primeru aplikacije, napisane s pomočjo .NET Web Forms, identificirajte njene slabosti in pomanjkljivosti. Opišite tehnologije, ki so kandidati za implementacijo nove različice aplikacije. Določite kriterije za primerjavo in izbiro razvojnih modelov in tehnologij, pri čemer upoštevajte tudi zahteve naročnika. Na podlagi izbranih kriterijev primerjajte razvojna modela .NET MVC in .NET Web Forms. Na primeru prenovljene aplikacije konkretno ovrednotite izbiro modela po prej omenjenih kriterijih in komentirajte prednosti in slabosti.

Zahvalajujem se svoji mentorici dr. Mojci Ciglarič za njene nasvete in pomoč pri pisanju diplomske naloge.
Zahvalil bi se tudi podjetju Crea, kjer sem lahko sodeloval na različnih projektih, kjer sem pridobil veliko izkušenj, ki so mi pomagale pri pisanju diplomske naloge.
Posebna zahvala gre tudi direktorju podjetja Crea, dr. Mateju Trampušu, za vse nasvete in pomoč pri pisanju diplomske naloge.
Še posebaj pa bi se zahvalil moji družini, mati Ireni, očetu Poldetu in bratu Mateju, ki so me vseskozi spodbujali.
Zahvaliljujem se tudi stricu Miranu za vso izkazano podporo.
Brez vas diplomskega dela ne bi bilo.

Kazalo

Seznam uporabljenih kratic in simbolov

BPM - Business Process Management (upravljanje s poslovnimi procesi)
ASP - Active Server Page (aktivne strežniške strani)
.NET - ogrodje za spletni strežnik, ki skrbi za obdelavo zahtevkov in pripravo odgovorov HTTP *OSX* - operacijski sistem Apple
HTML - Hypertext Markup Language (jezik za označevanje besedila)
XML - Extensible Markup Language (razširljiv označevalni jezik)
OMS - Order Management System (sistem za upravljanje z naročili)
VS - Visual Stuido, razvojno ogrodje
API - Application programming interface (programski vmesnik)
GUI - Graphical User Interface (grafični uporabniški vmesnik)
SoC - Separation of concerns (ločevanje delov kode)
WSDL - Web Services Description Language (jezik za opis spletnih storitev)
DLL - Dynamic Link Library (dinamična povezovalna knjižnica)
FCL - Framework Class Library (razredna knjižnica ogrodja)
URI - Uniform Resource Identifier (enotni identifikator vira)
JSON - JavaScript Object Notation (zapis objekta JavaScript)
BSD - Berkeley Source Distribution (distribucija programske kode Berkeley
MVP - Minimum Viable Product(minimalno izvedljiv izdelek)
AST - Abstract Syntax Tree (abstrakno sintaktično drevo)

Povzetek

Telekomunikacijska industrija se hitro razvija, zato pogosto pride do problema, ko obstoječa spletna aplikacija OMS ne nudi več zadostne podpore samim poslovnim zahtevam. OMS je spletna aplikacija, ki upravlja z naročili. Omogoča zajem, obdelavo in izvedbo naročil za uporabnike operaterskih storitev. Pri izgradnji nove spletne aplikacije OMS smo imeli v podjetju, ki za enega od slovenskih operaterjev deluje kot podizvajalec, dva možna razvojna modela ASP .NET Spletni obrazci in ASP .NET MVC, z obzirom na trenutno infrastrukturo, obstoječo programsko kodo in podporo, ki jo ima v lasti naročnik operater. Diplomsko delo v prvem delu predstavi oba razvojna modela, opis ključnih značilnosti in delovanje. Delo predstavi predvsem ključne mehanizme razvojnih modelov, ki jih omogočata in po čemer se v veliki večini tudi razlikujeta. V istem delu sledi tudi pregled tehnologij na strani odjemalca, ki jih mora bralec poznati za nadaljnje razumevanje. Poudarek je na programskem jeziku TypeScript, saj med vsemi znanimi programskimi jeziki, ki tečejo na strani odjemalca, odpravlja ključne pomanjkljivosti programskega jezika JavaScript na način, ki zahteva najmanj prilagajanja. Sledi poglavje o primerjavi med obstoječo in novo spletno aplikacijo OMS. V poglavju so predstavljeni kriteriji, ki so bili pri izbiri primernega razvojnega modela najbolj pomembni. Na koncu poglavja sledijo primeri in opisi iz obstoječe in nove spletne aplikacije OMS po ključnih kriterijih. Sledi zaključno poglavje, ki poda končne ugotovitve glede izbire kriterijev in dejanske implementacije spletne aplikacije OMS z vidika pravilne izbire razvojnega modela.

Ključne besede:
.NET, Spletni, obrazci, MVC, kriteriji, TypeScript

Abstract

Telecommunications industry is developing at a fast pace, so companies face numerous problems as a consequence of OMS web applications redundancy. OMS (Order Management Systems) are applications that manage orders. They enable the capture, processing and execution of orders for customers of operator services. When building a new OMS web application for a client (a company that is contracted by one of the largest telecommunications services providers in Slovenia), we had the option of choosing ASP .NET Web Forms or ASP .NET MVC as our development framework, with regard to existing code legacy and support.The work will focus mainly on the supported development models of the frameworks. In the first part of the thesis, both their characteristics and in functionalities will be presented along with a review of required client-side technologies that the reader must be familiar with, in order to fully understand the following chapters. The emphasis is on TypeScript programming language. This language (more efficiently than any other client side programming languages) eliminates the shortcomings of JavaScript with the least amount of required adjustments. Next, a comparison is made between the existing and the new OMS application, along with the key criteria that was used to select the development framework. Towards the end of the thesis use-cases and comparison between the old and the new application are described. In the final chapter I present the key findings about the criteria selection and OMS Web App development and how they correlate to the choice of the development framework.

Key words:
.NET, Web, Forms, MVC, criteria TypeScript

Poglavje 1

Uvod

1.1 Uvod v problematiko

Združeno telekomunikacijsko podjetje se kot telekomunikacijski operater dnevno srečuje z upravljanjem velikega števila naročil, ki jih prejema prek različnih prodajnih kanalov. Naročila se zajemajo in obdelujejo skozi različne IT sisteme. Pri upravljanju obstoječih IT sistemov se podjetje srečuje z raznimi omejitvami, kot so kompleksna ponudba z raznolikimi pravili, pomanjkljiva podpora več prodajnim kanalom, pomanjkljiva arhitekturna zasnova, ki preprečuje določene razširitve, pomanjkljiva nastavljivost sistema ob uvedbi novih ponudb, neenotno zajemanje in arhiviranje naročniške dokumentacije, upravljanje naročil pa je preveč povezano s samo kompleksnostjo IT infrastrukture, ki zagotavlja izvedbo in sledenje izvedbe naročila stranke. Prav tako se danes vedno več kode izvaja na strani odjemalca, kar je tudi ena od možnosti za pohitritev aplikacije. Eden izmed ključnih razlogov za prenovo je tudi zahteva po programskem vmesniku, ki bi bil izpostavljen preko spletnih storitev. Zaradi vseh omejitev obstoječega sistema in nezmožnosti hitrega prilagajanja novim tehnološkim trendom, se je pojavila potreba po novejšem sistemu, ki bo v čim večji meri odpravil pomanjkljivosti starega sistema. V diplomskem delu bom največji del posvetil sami izbiri modela za razvoj spletne aplikacije OMS.

1.2 Namen in cilj

Namen diplomske naloge:

- opredelitev težav obstoječe spletne aplikacije OMS,
- opredelitev zahtev naročnika,
- določitev ključnih kriterijev za primerjavo razvojnih modelov in izbira najbolj primernega,
- izdelava nove spletne aplikacije OMS na podlagi izbranega razvojnega modela in
- ovrednotenje obstoječe in nove spletne aplikacije OMS po izbranih ključnih kriterijih.

Poglavje 2

Opredelitev tehnologij na strani strežnika

V okviru podjetja, v katerem sem zaposlen, smo dobili naročilo za razvoj nove, sodobnejše spletne aplikacije OMS za enega od operaterjev v Sloveniji. Operater ima trenutno spletno aplikacijo OMS na strežniški strani v večini podprto s strani Microsoftovih tehnologij ter tehnologije IBM BPM. Le-ta se v sodobnih aplikacijah uporablja za nudenje podpore procesnemu delu. V podjetju smo se zaradi optimizacije razvojnega procesa in delovnih izkušenj razvojnega oddelka s tehnologijami Microsoft odločili, da bo nova rešitev OMS na strežniški strani prav tako podprta s tehnologijo Microsoft. Trenutno sta na trgu najbolj popularna razvojna modela ASP .NET Spletni obrazci in ASP. NET MVC. V sledečih poglavjih sledi opis razvojnih modelov in drugih tehnologij, ki jih mora bralec diplomske naloge poznati, da lahko razume problemsko domeno znotraj kriterijev. Te smo si kot podjetje zadali za ključne elemente, na podlagi katerih bomo izbrali najbolj ustrezno in perspektivno tehnologijo za izgradnjo nove spletne aplikacije OMS.

2.1 Opis tehnologij

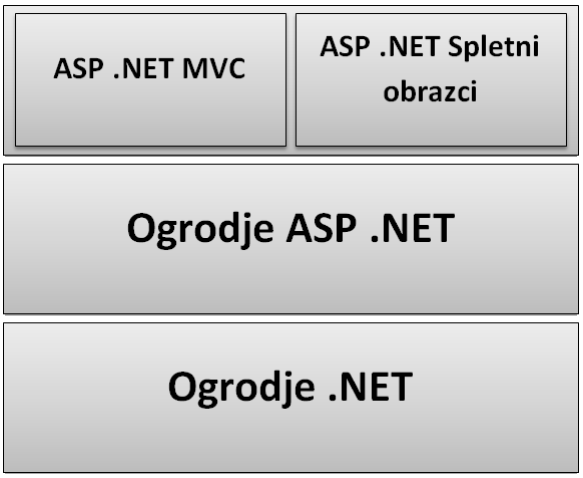
V tem poglavju diplomsko delo predstavi tehnologije, ki so možni kandidati za razvoj nove spletne aplikacije OMS. Pri kandidatih smo morali upoštevati naročnikovo obstoječo infrastrukturo in optimalne razvojne možnosti. Naročnik ima v lasti infrastrukturo, ki temelji na operacijskem sistemu Microsoft Windows, ter vse potrebne licence. Po drugi strani pa smo imeli v podjetju, kjer sem zaposlen, obstoječo programsko kodo razvito prav tako v tehnologiji, razviti s strani Microsoft-a, in bi nam izbira tehnologij, ki niso Microsoftove, povzročila ogromno dodatnega dela, saj bi morali na novo napisati programsko kodo, ki jo sicer lahko ponovno uporabimo pri izdelavi nove spletne aplikacije OMS.

Na strani spletnega strežnika sem se v diplomski nalogi osredotočil na dva Microsoft .NET razvojna modela, ki sta bila najustreznejša kandidata. Oba uporabljata ASP.NET, ki je ogrodje za spletni strežnik, ki skrbi za obdelavo zahtevkov in pripravo odgovorov HTTP. ASP.NET temelji na ogrodju .NET, ki je bilo razvito za uporabo na operacijskem sistemu Microsoft Windows. Slika 2.1 prikazuje sklad tehnologij. Leta 2015 ga je Microsoft delno prenesel tudi na druga okolja, kot so Linux in operacijski sistem podjetja Apple, z imenom OSX [1]. .NET vsebuje veliko knjižnico razredov (FCL) in zagotavlja skladno komunikacijo med več skupnimi jeziki, tako da lahko vsak jezik uporablja kodo napisano v drugem jeziku.

2.2 Razvojni model .NET Spletni obrazci

2.2.1 Zakaj je prišlo do razvoja razvojnega modela .NET Spletni obrazci?

Z razvojnim modelom .NET Spletni obrazci so pri Microsoft-u skušali pred razvijalci skriti uporabo protokola HTTP in programski jezik za označevanje HTML pri razvoju uporabniškega vmesnika, kot se je razvijal pri namiznih aplikacijah.



Slika 2.1: Sklad tehnologij .NET

Protokol HTTP je znan po tem, da v osnovi nima stanja. To pomeni, da strežniku ni potrebno ohranjati informacij ali statusa zahtevkov oz. odgovorov HTTP. Ker omenjenih informacij spletni strežnik ne rabi beležiti oz brati in kamorkoli zapisovati, lahko spletni strežnik deluje bistveno hitreje, kar je tudi glavni namen protokola HTTP brez stanja. V praksi se izkaže, da določene spletne aplikacije potrebujejo sledenje uporabniškim zahtevkom med različnimi spletnimi stranmi, kadar želimo le-te prilagoditi uporabniškim potrebam. Osnovna ideja je, da se na odjemalca prenese majhna količina podatkov, preko katerih spletni strežnik pri nadaljnjih zaporednih zahtevkih identificira sejo, ki pripada uporabniku. Sejo na strežniku torej potrebujemo vedno. Poznamo sledeče mehanizme prenosa podatkov:

- uporaba piškotkov HTTP,
- skrite spremenljivke, kadar spletna stran vsebuje spletni obrazec
- in URL prepisovanje z uporabo URI kodiranih parametrov, npr. $\ldots/index.php?id_seje = 1234$

V času, ko je na trg prispel razvojni model ASP.NET Spletni obrazci, je bilo veliko razvijalcev, ki so slabo poznali programski jezik HTML, saj so bile spletne tehnologije šele v vzponu. Rešitev je bila ravno razvojni model ASP.NET Spletni obrazci, saj vsebuje nabor komponent za uporabniški

vmesnik (gumb, stran, vnosno polje, itd.) ter objektno orientiran razvojni model za grafični uporabniški model (GUI), ki zagotavlja tudi stanje.

2.2.2 Kaj je ASP .NET Spletni obrazec?

Razvojni model predstavi tako imenovan koncept spletnih obrazcev, ki jih uporabnik zahteva z uporabo spletnega brskalnika. Strani so lahko napisane kot kombinacija kode HTML, jezikov na strani odjemalca, strežniških komponent in kode, ki se izvaja na strani spletnega strežnika. Ko uporabnik zahteva določeno stran, se le-ta prevede in izvede na strežniku s pomočjo ogrodja, ki poskrbi za izgradnjo kode HTML, ki jo brskalnik lahko bere in predstavi uporabniku [2].

ASP .NET Spletni obrazeci imajo sledeče ključne lastnosti:

- ločevanje kode HTML in druge kode povezane z uporabniškim vmesnikom od kode aplikacije (poslovne logike),
- bogat niz strežniških komponent za pogoste scenarije, vključno z dostopom do baze podatkov,
- zelo močno povezovanje samih podatkov na uporabniškem vmesniku z odlično podporo že vgrajenih orodij v razvojnem okolju VS,
- podpora skriptnim jezikom na strani odjemalca, ki se izvaja v brskalniku,
- podpora različnim ostalim mehanizmom, kot so usmerjanje, varnost, zmogljivost, mednarodna neodvisnost, testiranje, razhroščevanje, upravljanje z napakami ter upravljanje stanja.

2.2.3 Tipične težave na strani odjemalca

S pomočjo ASP .NET Spletnih obrazcev lahko rešujemo probleme, ki niso najbolj tipični za aplikacije, ki se izvajajo na strani odjemalca:

- Izgradnja bogatega spletnega uporabniškega vmesnika – Z uporabo osnovnega HTML-ja je zelo težko načrtovati in izdelati uporabniški

vmesnik, še posebno ko stran zahteva kompleksno oblikovanje, veliko vsebino dinamičnih vsebin in objektov, ki so namenjeni komunikaciji z uporabnikom in vsebujejo bogat nabor funkcionalnosti.

- Ločevanje odjemalca od spletnega strežnika – Pri spletnih aplikacijah sta pogosto odjemalec (brskalnik) in spletni strežnik različna programa, ki vedno tečeta na različnih računalnikih ter po navadi tudi na različnih operacijskih sistemih. Posledično si oba večja kosa programske rešitve delita zelo majhen nabor informacij; sicer omogočata medsebojno komunikacijo, vendar si tipično izmenjata zelo majhen in enostaven del informacij.
- Izvajanje brez stanja – Ko spletni strežnik dobi zahtevek za spletno stran, jo poišče, obdela, pošlje odjemalcu in potem izbriše vse informacije v zvezi z omenjeno spletno stranjo. Če uporabnik ponovno zahteva enako stran, mora spletni strežnik opraviti celotno zaporedje nalog še enkrat od samega začetka. Povedano drugače, spletni strežnik nima prav nobenega spomina o straneh, ki jih je obdelal, strani so torej brez stanja. V primeru, ko ima aplikacija potrebo po upravljanju informacij o straneh, situacija brez stanja postane izziv. Izjema je predpomnenje, ki omogoča, da kodo HTML cele spletne strani ali njenega dela zgradimo ob prvi zahtevi, rezultat pa si zapomnimo. Odgovore za vse nadaljne zahtevke istega tipa pa poiščemo v spominu.
- Neznane sposobnosti odjemalca – V veliko primerih je spletna aplikacija različnim uporabnikom dostopna v različnem naboru brskalnikov, kot so namizni brskalnik, brskalnik na mobilnem telefonu ali tablici. Brskalniki imajo različne zmogljivosti, zaradi česar je posledično težko izdelati aplikacijo, ki nudi popolnoma enako funkcionalnost na vseh brskalnikih na trgu.
- Težave z dostopom do podatkov – Dostopanje do podatkov, torej branje in pisanje, je v tradicionalnih spletnih aplikacijah zelo oteženo in tipično potratno z viri.
- Težave s razširljivostjo programske rešitve – Pogosto pride do problema

pri doseganju želenih ciljev razširljivosti aplikacije, zaradi načrtovanja z obstoječimi metodami, saj pride do pomanjkanja združljivosti različnih komponent spletne aplikacije. To je pogosto razlog za točko odpovedi spletne aplikacije, ki je pod pritiskom teže rasti cikla.

2.2.4 Reštive težav na strani odjemalca

Soočanje s tovrstnimi izzivi lahko za spletne aplikacije zahteva ogromno časa in truda. ASP .NET Spletni obrazci in ASP .NET ogrodje rešujeta izzive s pomočjo sledečih mehanizmov:

- Intiutiven, skladen objektni model – ASP .NET ogrodje predstavi obrazec kot objektni model, ki ne deluje ločeno na strežniški in odjemalčev del. Omogoča zmožnost dodajanja nastavitev elementom strani in odzivanja na dogodke. ASP .NET strežniške komponente so povzetek fizične vsebine strani HTML in neposredne interakcije med brskalnikom in strežnikom. V splošnem lahko uporabimo strežniške komponente, ne da bi morali razmišljati kako kreirati kodo HTML za predstavitev in obdelavo gradnikov in njihove vsebine.
- Dogodkovno voden programski model – ASP .NET Spletni obrazci je spletnim aplikacijam predstavila znan model pisanja obravnave dogodkov, ki se lahko pojavijo tako na strani odjemalca kot strežnika. ASP .NET ogrodje povzame ta model na način, da so mehanizem zajemanja dogodkov na strani odjemalca, prenos le-teh na stran strežnika, in klic ustrezne metode popolnoma avtomatski in nevidni za razvijalca. Rezultat je enostavno pisanje različnih struktur, ki podpirajo dogodkovno krmiljen razvoj.
- Intiutivno upravljanje stanja – ASP .NET ogrodje avtomatsko upravlja stanje strani in njenih komponent in nam ponudi ekspliciten dostop do upravljanja informacij, ki so pomembne za spletno aplikacijo. To lahko dosežemo brez prevelike porabe virov spletnega strežnika in je lahko podprto z ali brez pošiljanja piškotkov na brskalnik.

- .NET ogrodje poskrbi za možnost razširitve same izvedbe na strani stežnika – ASP .NET ogrodje nam omogoča, da lahko enostavno in brez zapletenih sprememb v aplikacijski logiki razširimo spletno aplikacijo z računalniškega strežnika z enim procesorjem na računalniško spletno kmetijo.

2.2.5 Življenjski cikel strani

Ko se ASP .NET stran izvaja, pride do izvedbe niza korakov, ki ga imenujemo življenski cikel strani(Page Life Cycle) [3]. Ti koraki vključujejo:

- začetno vzpostavitev,
- izgradnjo primerkov kontrol,
- obnavljanje in ohranjevanje stanja,
- izvajanje kode za upravljanje dogodkov in
- izdelavo predstavitvene kode.

Življenjski cikel strani je sestavljen iz različnih stopenj, znotraj katerih se izvajajo dogodki.

Ključne stopnje življenskega cikla ASP .NET spletnih strani

V splošnem gre stran skozi stopnje. Nekateri deli življenjskega cikla strani se izvedejo le v primeru povratnega klica (PostBack). Stopnje so:

- Zahtevek za stran(Page Request) - Zahtevek za stran se pojavi pred začetkom življenjskega cikla strani. Ko se pojavi, ASP .NET določi, ali je stran potrebno razčleniti in pripraviti ali pa v odgovor posreduje stran, ki jo ima shranjeno v svojem začasnem spominu, brez njene obravnave.
- Začetek (Start) - V začetnem stanju, se nastavijo lastnosti strani, kot sta zahtevek in odgovor. V tem stanju stran določi, ali gre za novo zahtevo ali povratni klic, ter nastavi lastnost $isPostBack$. Stran nastavi

tudi $UICulture$ lastnost, ki predstavlja bližnjico do niti, ki je trenutno v izvajanju.

- Začetna vzpostavitev (Initialization) - V fazi začetne vzpostavitve strani se vsem kontrolam na strani določi unikatni ID ključ. Uporabi se tudi $MasterPage$ in teme, če so te primerne. V kolikor je trenutni zahtevek povratni klic, podatki iz le-tega še niso bili naloženi in vrednosti lastnosti kontrol še niso bili obnovljene na vrednosti iz stanja pogleda.
- Nalaganje (Load) - V fazi nalaganja se v primeru povratnega klica vrednosti lastnosti komponent napolnijo iz stanja pogleda in iz stanja komponent.
- Upravljanje z dogodki pri povratnem klicu (Postback event handling) - V primeru povratnega klica, se izvede upravljanje dogodkov na strani komponent. Za tem se kliče preveritvena metoda na vseh komponentah, ki nastavi $isValid$ lastnost posamezne komponente in strani.
- Izris (Rednering) - Pred izrisom se shrani stanje pogleda za stran in vse komponente. V stopnji izrisa stran pokliče metodo za izris za vsako komponento, ki zagotavlja zapisovalnik besedila, ki zapiše rezultat kot $OutputSream$ objekt, ki se nahaja v odgovoru strani.
- Razbremenitev (Unload) - Do stopnje razbremenitve pride po tem, ko je stran v celoti izrisana, poslana odjemalcu in pripravljena na uničenje. V tej točki se lastnosti, kot sta odgovor in zahtevek, razbremeni ter izvede čiščenje.

Dogodki življenjskega cikla strani

Dogodki življenjskega cikla strani so:

- $PreInt$ - Sproži se, ko se zaključi začetna stopnja in preden se začne začetna vzpostavitev. Uporabimo ga, ko želimo preveriti, ali gre za povraten klic ali začetno zahtevo, ko želimo izdelati dinamične komponente, ko želimo nastaviti $MasterPage$ dinamično, ko želimo nastaviti temo dinamično.
- $Init$ - Sproži se, ko so vse komponente vzpostavljene. Dogodeg $Init$ po-

samezne komponente se izvede pred *Init* dogodkom strani. Uporabimo ga, ko želimo brati ali vzpostaviti lastnosti komponent.

- *InitComplete* - Se sproži po koncu stopnje začetne vzpostavitve. Uporabimo ga, ko želimo uveljaviti spremembe stanja pogleda, vendar moramo zagotoviti, da so prisotne tudi po povratnem klicu.
- *PreLoad* - Sproži se, ko stran naloži stanje pogleda zase in za komponente, zatem pa obdela podatke iz povratnega klica, ki se nahajajo v primerku zahteve.
- *Load* - Kliče se *OnLoad* metoda na strani, ki nato rekurzivno pokliče *OnLoad* metodo na komponentah, dokler stran ni naložena v celoti. Load dogodek se pri posamezni komponenti izvede za *Load* dogodkom na strani. *OnLoad* metodo uporabimo, ko želimo nastaviti lastnosti komponent ter vzpostaviti povezavo s podatkovno bazo.
- Dogodki na strani komponent - Uporabimo jih, ko želimo obravnavati različne dogodke na strani komponent, kot na primer *Click* dogodek na komponenti *Button*.
- *LoadComplete* - Sproži se ob koncu stopnje upravljanja z dogodki. Uporabimo ga za opravila, pri katerih potrebujemo naložene vse komponente.
- *PreRender* - Sproži se, ko je stran narejena v celoti z vsemi komponentami, ki so zahtevane za izris celotne strani. Stran sproži *PreRender* dogodek na strani ter za vse komponente rekurzivno. *PreRender* na komponenti se izvede za *PreRender* dogodkom na strani.
- *PreRenderComplete* - Sproži se za vsako komponento, ki ima nastavljeno lastnost *DataSourceID*, ter pokliče metodo *DataBind*.
- *SaveStateComplete* - Sprožise, ko se stanje pogleda in stanje komponente shrani za stran in vse njene komponente. Vse spremembe v tej točki vplivajo na izrisovanje strani, vendar spremembe niso prisotne ob povratnem klicu.
- *Render* - *Render* ni dogodek. Stran na tej stopnji kliče *Render* metode za vse kontrole. Vse ASP .NET strežniške komponente imajo *Render*

metodo, katere rezultat je koda HTML, ki se pošlje odjemalcu. V primeru komponent po meri je tipično potrebno povoziti to metodo na način, da vrne kodo HTML, ki jo potrebuje za izris. V primeru uporabniških komponent pa je izrisovanje vključeno samodejno, zato ni potrebe po dodatni kodi za sam izris.

- *Unload* - Sproži se za vsako komponento in na koncu še za stran. Uporabimoga, ko želimo počistiti podatke specifičnih komponent, kot je npr. zapiranje povezave do baze podatkov na komponenti, ki je z bazo podatkov povezana. V primeru strani ga uporabimo, ko želimo počistiti določene podatke, kot so zapiranje odprtih dokumentov, aktivnih povezav na bazo podatkov ali pa izvesti zaključno beleženje ali druge naloge, ki so vezane na zahtevek.

Slika2.2 prikazuje nekaj najbolj pomembnih metod razreda *Page*, ki jih lahko povozimo z namenom, da dodamo kodo, za katero želimo, da se izvede v točno določenih točkah znotraj življenjskega cikla strani. Slika 2.2 prikazuje tudi povezavo med metodami, dogodki strani ter dogodki komponent.

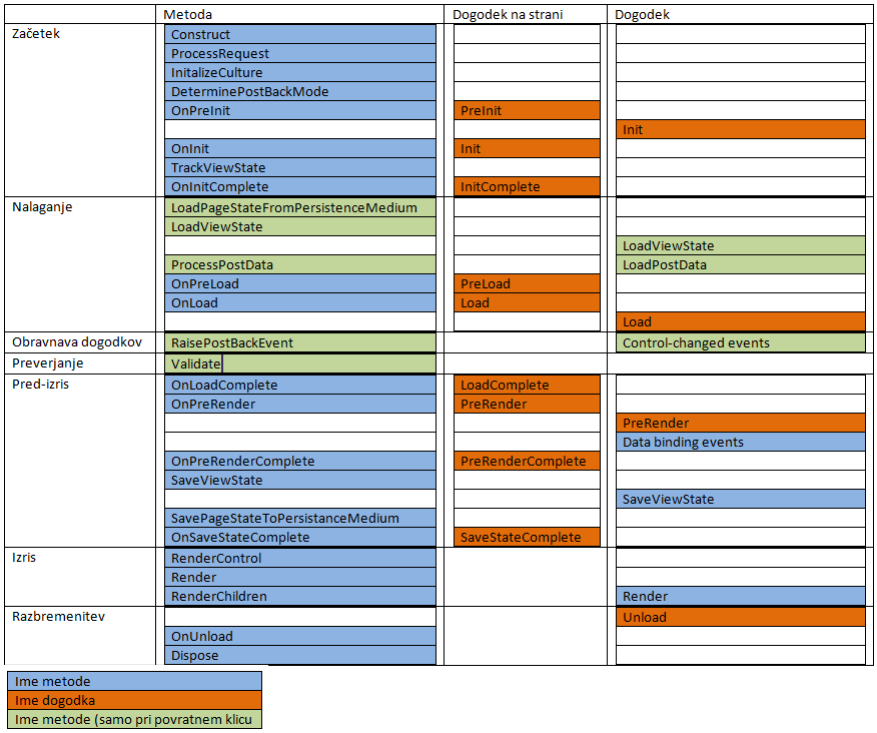
Življenski cikel je kompleksen, mnogim razvijalcem nerazumljiv, zato včasih predstavlja tudi breme.

Podobna težava je tudi generiranje kode HTML, na katerega ima razvijalec le malo vpliva, kar lahko predstavlja težave, če želimo na odjemalcu kodo HTML spreminjati s pomočjo ene od odjemalskih knjižnic (npr. JQuery). Prav z razmahom odjemalskih knjižnic je postala to ena od večjih slabosti ASP.NET Spletnih obrazcev.

Povratni klic in stanje pogleda

Povratni klic je ime postopka predložitve ASP .NET strani strežniku za obdelavo. Vsakič, ko se zgodi povratni klic, se stran HTML pošlje spletnemu strežniku. Spletni strežnik naloži stran, obdela dogodke, ustvari novo kodo HTML in jo pošlje nazaj odjemalcu. Tradicionalen pristop je, da se ob povratnem klicu vedno osveži celotena stran HTML.

Pri velikih in zapletenih spletnih aplikacijah, ki shranjujejo ogromno količino



Slika 2.2: Življenski cikel strani

podatkov stanja pogleda, je lahko to časovno zelo potratno za odjemalca. To postane velik problem za spletni strežnik, ki je omejen z viri, kot so spomin in pasovna širina.

Spletena aplikacija navadno vključi stanje pogleda v stran HTML. Po citiranju Microsoft-a je stanje pogleda tehnika, uporabljena s strani ASP Net spletnih strani, s katero lahko ohranimo spremembe stanja .NET Spletnih obrazcev med povratnimi klici. Stanje pogleda omogoča shranjevanje stanja objektov in se shrani znotraj strani HTML kot skrito polje. Stanje pogleda se tako prenese na odjemalca in nazaj na spletni strežnik, pri čemer se ne shrani na spletnem strežniku ali v katerikoli drugi obliki. Stanje pogleda se uporablja za ohranjanje stanja strežniških objektov med samimi povratnimi klici in postane zelo velik pri velikih in zapletenih aplikacijah.

2.2.6 Podpora komponentam

Razvojni model .NET Spletni obrazci podpira dve vrsti komponent - uporabniške komponente in komponente po meri. Uporabniške komponente so enostaven način za razdelke uporabniškega vmesnika. Temeljijo na enakem principu kot .NET Spletni obraci. Sintaksa za izgradnjo je prav tako podobna sintaksi za izgradnjo .NET spletne strani. Razlikujeta se po tem, da uporabniške komponente ne vsebujejo elementov HTML $< html >$, $< body >$ in $< form >$, saj je uporabniška komponenta uporabljena znotraj .NET spletne strani. Primer enostavne uporabniške komponente, ki je v prvem delu sestavljena iz bloka kode, namenjene izvajanju na strežniku, ter bloka kode, namenjene za izris komponente, kot prikazuje slika programske kode 2.3.

Uporabniško komponento lahko enostavno uporabimo v .NET spletni strani, kot prikazuje slika programske kode 2.4.

Komponente po meri so z razliko od uporabniških komponent že prevedeni kosi kode, ki so razporejeni v ločenih sklopih dinamično povezovalnih knjižnic (DLL). Slika programske kode 2.5 prikazuje primer uporabniške komponente. Komponente po meri lahko znotraj spletne strani uporabimo na enak način, kot uporabniške komponente.

```
<!--koda za izvajanje na spletnem strežniku-->
<script language="C#" runat="server">
   public void button1_Click(object sender, EventArgs e)
   {
      label1.Text = "Hello World!!!";
   }
</script>
<!--koda za izris uporabniške komponente-->
<asp:Label id="label1" runat="server"/>
 <br><br>
<asp:button id="button1" text="Hit"
        OnClick="button1_Click" runat="server" />
```

Slika 2.3: Primer uporabniške komponente

```
<!--registracija komponente-->
<%@ Register TagPrefix="UC" TagName="TestControl" Src="test.ascx" %>
<!--uporaba komponente-->
<html>
    <body>
        <form runat="server">
            <UC:TestControl id="Test1" runat="server"/>
        </form>
    </body>
</html>
```

Slika 2.4: Primer uporabe uporabniške komponente

```
using ...
namespace CustomControls
{
  [DefaultProperty("Text")]
  [ToolboxData("<{0}:ServerControl1 runat=server></{0}:ServerControl1 >")]
  public class ServerControl1 : WebControl
  {
    [Bindable(true)]
    [Category("Appearance")]
    [DefaultValue("")]
    [Localizable(true)]
    public string Text
    {
      get
      {
        String s = (String)ViewState["Text"];
        return ((s == null) ? "[" + this.ID + "]" : s);
      }
      set
      {
        ViewState["Text"] = value;
      }
    }
    protected override void RenderContents(HtmlTextWriter output)
    {
      output.Write(Text);
    }
  }
}
```

Slika 2.5: Primer komponente po meri

Lastnosti uporabniških komponent:

- Izdelane za enkratne funkcionalnosti v aplikaciji.
- Izdelava je preprosta, podobna izdelavi .NET spletni strani.
- Odlična izbira, ko potrebujemo statično vsebino s točno določeno postavitvijo.
- Pisanje ne zahteva veliko oblikovanja, saj se oblikuje v času načrtovanja in večinoma vsebuje statične podatke.

Lastnosti komponent po meri:

- Izdelane za večkratno uporabo.
- Izdelava je zelo naporna, saj ne vsebuje podpore za izgled.
- Primerno za spletne aplikacije s spremenljivo vsebino.
- Izdelava komponente na začetku zahteva veliko znanja in razumevanja življenjskega cikla strani, za kar je že poskrbljeno pri uporabniških komponentah.

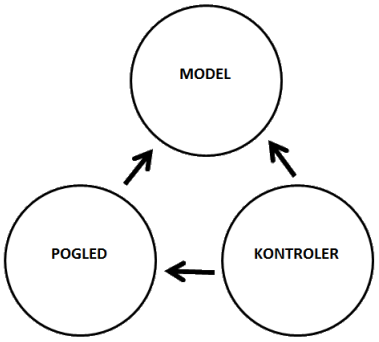
2.3 Razvojni model .NET MVC

Oktobra leta 2007 je Microsoft izdal razvojni model .NET MVC za razvoj spletnih aplikacij, zgrajen na jedru ASP.NET ogrodja, kot direkten odgovor na razvoj tehnologij, kot je Rails, in kot odgovor na kritike .NET Spletnih obrazcev.

2.3.1 MVC

MVC ne velja za nov arhitekturni pristop, saj se je o njem prvič govorilo leta 1978 na Xerox PARC [5]. Šele kasneje je MVC postal popularen tudi pri razvoju spletnih aplikacij. Eden izmed razlogov je, da se uporabniška izkušnja z MVC aplikacijami odraža na zelo naraven način. Uporabnik sproži akcijo, aplikacija pa odgovori tako, da popravi podatke v modelu, ter uporabniku vrne posodobljen pogled, kot prikazuje slika 2.6. Cikel se ponavlja, kar se

zelo lepo ujema z dostavo spletne aplikacije kot niz zahtevkov in odgovorov HTTP.



Slika 2.6: Model-pogled-krmilnik

Ena izmed najbolj pomembnih funkcionalnosti, ki jo mogoča MVC pristop, je ločevanje pristojnosti. V aplikaciji si želimo kar se da neodvisne komponente in toliko odvisnosti, ki jih še lahko obvladujemo. V idealni situaciji bi imeli opravka s komponentami, ki ne bi nič vedele o drugih komponentah, ter bi se vsaka ukvarjala samo s svojo domeno, z drugimi komponentami pa bi komunicirala preko abstraktnih vmesnikov. Tako povezane komponente v računalniškem žargonu imenujemo šibko sklopljene. Šibko sklopljene aplikacije so poglavitne za enostavnejše testiranje in spreminjanje aplikacije [6].

2.3.2 Vključevanje odvisnosti

Šibko sklopljenost v aplikaciji omogoča *DependencyInjection*. To je način povezovanja komponent, za katerega je značilno, da lahko konkretno implementacijo izberemo v času izvajanja, ne pa v času prevajanja. Ne omogoča enostavnega pridobivanja objekta, razen na način, ko ustvarimo nov objekt s privzeto besedo *new*, kar pa omej`C#`uje pomen šibko sklopljenih komponent, saj na ta način enostavna menjava posameznih delov ni mogoča. Rešitev omenjenega problema vsebuje dva koraka. V prvem moramo poskrbeti, da odstranimo vse odvisnosti do razredov znotraj posamične komponente ter

da komuniciramo samo preko vhodnih objektov, ki implementirajo vmesnik. Drugi korak pri vzorcu vključevanju odvisnosti je, da se odvisnosti vključijo s pomočjo mehanizma ob času kreiranja naše komponente (objekta/razreda).

Sedaj smo uspešno izločili odvisnosti komponente od razredov, še vedno pa moramo določiti mesto, kjer se bo objekt nekega razreda ustvaril. Rešitev za to je tako imenovani *dependencyinjectioncontainer*, ali krajše IoC [7].

2.3.3 Stroj za poglede Razor

MVC nudi za namen generiranja kode HTML največjo podporo stroju za poglede Razor. Razor rešuje sledeče probleme:

- definicija in dostopanje do določenega tipa modela z uporabo izraza @*model*,
- zmanjšanje podvojene kode v pogledih z uporabo postavitve strani,
- določevanje privzetege postavitve z porabo "*viewstart* pogleda,
- posredovanje vrednosti podatkov s kontrolerja v pogled s pomočjo modela pogleda ali pa z uporabo *viewbag* mehanizma,
- prilagajnaje vsebine glede na vrednost podatkov s pomočjo Razor pogojnih stavkov,
- sprehajanje skozi zbirko podatkov z uporabo izraza @$foreach$ in
- dodajanje imenskih prostorov znotraj pogleda z uporabo izraza @$using$.

Poglavje 3

Tehnologije za podporo izvajanju kode na strani odjemalca

V tem poglvaju sem za ključni pregled uporabil vir [18].

3.1 Uvod v problematiko

Ker spletne aplikacije postajajo vse bolj vplivne in prefinjene, se od njih tudi pričakuje, da delujejo enako kot namizne aplikacije. Osnovna arhitektura spletnih aplikacij je zgrajena tako, da se večina izvorne kode in knjižnic nahaja in obravnava na strani spletnega strežnika. Osnovna naloga spletnega strežnika je sprejemanje vhodnih zahtev HTTP in vračanje zahtevane podatke v odgovoru HTTP. Za enkrat ni videti, da bi bile spletne strani zgrajene iz drugega programskega jezika kot HTML.

Skripta na strani odjemalca je programska koda, ki obstaja znotraj odjemalčeve strani HTML. Ta koda se izvede na strani odjemalca, zato se ne izvede povratni klic na spletni strežnik. Običajno se skripte na strani odjemalca uporabljajo za navigacijo strani, preverjanje podatkov in pretvorbo podatkov. Najbolj razširjen jezik za tovrstno pisanje skript je JavaScript. Ja-

vaScript je podprt v vseh spletnih brskalnikih, danes pa doživlja intenziven razvoj, po tem, ko je imel skoraj 10 letni premor [8].

Dve ključni prednosti skriptnih jezikov na strani odjemalca sta:

- rezultat uporabniških akcij je takojšen odgovor spletne strani, saj le-ta ne potrebuje potovanja na spletni strežnik, kjer poteka obdelava in pot nazaj, in
- spletni strežnik potrebuje in porabi manj virov.

3.2 Ajax in jQuery

Dve ključni prednosti programiranja na strani klienta sta Ajax in jQuery. Kratica Ajax pomeni Asinhroni JavaScript in XML. XML je razširljiv označevalni jezik, primeren za izmenjavo podatkov med aplikacijami. Danes se XML pogosto zamenja z objektom JSON. Ajax ni orodje ali programski jezik, ampak je koncept. Omogoča klic strežnika direktno s strani odjemalca brez uporabe mehanizma s povratnim klicem. Z drugimi besedami, gre za mehanizem izmenjave podatkov s spletnim strežnikom in posodobitev le dela strani HTML brez ponovnega nalaganja celotne strani HTML. Tipične aplikacije, ki uporabljajo AJAX so Gmail, Google Maps, Youtube, itd. Klasična spletna stran zahteva ponovno nalaganje spletne strani, da se posodobi njena vsebina. V primeru spletne e-pošte bi to pomenilo, da bi moral uporabnik ročno osvežiti nabiralnik e-pošte, da bi preveril, če ima kakšno novo sporočilo. Tak način bi zelo upočasnil aplikacijo, poleg tega pa bi od uporabnika zahteval še vhodno akcijo. Ko bi uporabnik osvežil nabiralnik e-pošte, bi moral strežnik ponovno izgraditi celotno spletno stran, vključno s kodo HTML, CSS, JavaScript in seveda tudi uporabniško e-pošto. To bi bilo popolnoma neučinkovito. Ideja je, da bi strežnik uporabniku poslal le novo sporočilo in ne celotne strani. Do leta 2003 je glavnina spletnih brskalnikov omenjeni problem reševala s sprejetjem $XMLHttpRequest$ objekta, ki je omogočal brskalniku komunikacijo s strežnikom brez nepotrebnega nalaganja celotne strani. Ajax zahtevki so sproženi s strani JavaScript kode. JS koda pošlje zahtevek na URL naslov.

Po prejetju odgovora, se sproži povratni klic, ki poskrbi za obravnavo odgovora. Ker je zahtevek asinhron, se preostanek kode izvaja, medtem ko se obdeluje prejeti zahtevek, zato je nujno, da uporabimo povratni klic za obravnavo odgovora. Na žalost različni brskalniki podpirajo AJAX API na različne načine. Običajno bi to pomenilo, da bi moral razvijalec poskrbeti, da bi AJAX klici na vseh brskalnikih delovali enako. Na srečo pa jQuery knjižnica vsebuje tudi tovrstno podporo, ki nevtralizira razlike med brskalniki. Vsebuje tako funkcionalno bogate metode, kot je $\$.ajax()$, in enostavne udobne metode ko so $\$.get()$, $\$.getScript()$, $\$.getJSON()$, $\$.post()$ and $\$().load()$. Večina jQuery aplikacij pravzaprav ne uporablja XML za izmenjavo podatkov, kot je razvidno iz imena AJAX. Namesto XML se večinoma uporablja kar čisti HTML ali JSON. jQuery je hitra, majhna in s funkcionalnostmi bogata JavaScript knjižnica. Omogoča prečkanje in manipulacijo HTML dokumenta, upravljanje z dogodki, animacijami in poenostavlja uporabo Ajax tehnologije z enostavno uporabo uporabniškega vmesnika(API), ki je podprt v večini spletnih brskalnikov. S kombinacijo vsestranskosti in razširljivosti je jQuery dosegel to, da danes veliko število ljudi piše JavaScript kodo [9].

3.3 TypeScript

3.3.1 Kratek opis jezika

TypeScript je odprtokodni porgramski jezik. Razvija in vzdržuje ga podjetje Microsoft, za razvoj Angualar 2.0 pa ga je prevezl tudi Google [10]. Je stroga podmnožica JavaScript-a, ki mu dodaja opcijske statične tipe in razredno-usmerjeno in objektno-usmerjeno programiranje. Pri razvoju TypeScript-a je sodeloval tudi glavni razvojni arhitekt programskega jezika `C#` in stvaritelj Delphi in TurboPascal programskih jezikov, Anders Hejsberg. TypeScript predstavlja možnost za razvoj aplikacij JavaScript tako na strani odjemalca kot na strani strežnika. Ker je TypeScript podmnožica JavaScripta, je vsaka obstoječa koda JavaScript tudi veljavna koda TypeScript. Ker se koda TypeScript prevede v berljivo kodo JavaScript, je možno tudi obratno prevajanje,

kjer je na koncu rezultat berljiva koda TypeScript. Tako velika skladnost z jezikom JavaScript in izboljšava ključnih pomanjkljivosti JavaScript je ključni uspeh jezika TypeScript [12].

3.3.2 Poljubna uporaba statičnih tipov in sklepanje le teh

Jezik JavaScript ne uporablja statičnih tipov ampak dinamične tipe. Kakšnega tipa je neka spremenljivka, lahko izvemo šele v času izvajanja, kar pa je lahko prepozno. Omenjeno pomankljivost JavaScript-a odpravi TypeScript z opcijsko uporabo statičnih tipov. Napake, povzročene ob upoštevanju napačnih predpostavk nekaterih spremenjivk določene vrste je možno popolnoma izkoreniniti. Omenjeno velja, če seveda tipe uporabljamo na pravi način in konsistentno.

TypeScript omogoča tudi sklepanje/ugotavljanje tipa, kar poenostavi uporabo tipov ter jih naredi manj izrazne. Na primer $varx = "hello"$ v TypeScript jezku pomeni enako kot $varx : string = "hello"$. Tip spremenljivke je tako moč ugotoviti iz same uporabe. Tudi če ne uporabljamo statičnih tipov ter namesto tega uporabimo ključno besedo var, so spremenljivke še vedno tipizirane, kar preprečuje, da razvijalec naredi napako, ki bi se med izvajanjem pokazala kot napaka pri izvajanju programa.

TypeScript privzeto podpira opcijsko uporabo tipov. Na primer $functiondivideByTwo(x)returnx/2$ je veljavna funkcija v jeziku TypeScript, ki je lahko poklicana s poljubno vrednostjo, tudi z vrednostjo niz, kar pa bi v omenjenem primeru povzročilo napako pri izvajanju kode. Ta mehanizem deluje, ker TypeScript avtomatsko doda tip Any, kadar tip ni določen ali pa ne more biti ugotovljen, kot je to v omenjenem primeru. Zgornja funkcija $divideByTwo$ v praksi postane $functiondivideByTwo(x : any) : any$. Na voljo imamo tudi direktivo prevajalniku, s katero lahko onemogočimo tovrstno privzeto nastavljanje. Če uporabimo to zastavico, potem preidemo na višji nivo varnosti kode, posledično pa to pomeni, da moramo bolj dosledno uporabljati tipe.

3.3.3 Izboljšana IDE podpora

Izkušnja razvijalca pri uporabi TypeScript-a je mnogo boljša, kot pri uporabi JavaScript-a. IDE je obveščen v realnem času s strani prevajalnika TypeScript-a, na podlagi bogate informacije o tipu. To nam omogoča kar nekaj prednosti, kot je refakturiranje na področju celotne programske kode ali pa nudenje pomoči in informacij skozi razvoj. Ni več potrebno, da bi si funkcije morali zapomniti ali pa jih preverjati na spletnih referencah. Napake pri prevajanju so posredovane direktno IDE okolju, ko je razvijalec zaposlen s pisanjem kode. Navsezadnje to pomeni, da nam TypeScript z bogato IDE podporo omogoča večjo produktivnost v primerjavi s pisanjem kode v jeziku JavaScript. Tako lahko porabimo več časa za pisanje kode ter manj za razhroščevanje. Na trgu obstaja več okolij IDE, ki imajo odlično podporo za TypeScript, kot npr.: VisualStudio, Atom, Sublime ali IntelliJ/WebStrom.

3.4 Dart

3.4.1 Kratek opis jezika

Programski jezik Dart je splošno namenski jezik, ki ga je v sami zasnovi razvilo podjetje Google, kasneje pa je napredoval v obliki standarda Ecma (ECMA-408). Danes ga pogosto uporabljamo za razvoj spletnih, mobilnih in strežniških aplikacij ter za naprave, ki se povezujejo z internet stvarmi. Je odprtokoden programski jezik izdan pod licenco BSD. Je razredno orientiran, objektno usmerjen programski jezik s sintakso podobno C programskemu jeziku, ki se prav tako prevede v JavaScript kodo. Podpira uporabo interface vmesnikov, abstraktnih razredov, enumov, reified generics in opcijsko uporabo tipov [13].

Dart omogoča razvijalcem gradnjo bolj kompleksnih in bolj zmogljivih aplikacij za sodoben splet. Z uporabo jezika Dart lahko hitro napišemo prototipe aplikacij, ki se hitro razvijajo. Na voljo imamo napredna orodja, zanesljive knjižnice in programske inženirske tehnike.

3.4.2 Podpora Dart-u

Dart ni samo programski jezik, je ogrodje. To pomeni, da vključuje svoje standardne knjižnice in orodja. Kljub temu da jezik Dart lahko prevedemo direktno v JavaScript, imamo tudi možnost predogleda za brskalnika Chrome in Dartium s pomočjo Dart VM (virtual machine). Chrome je eden izmed najbolj popularnih brskalnikov, Dartium pa je posebna verzija Cromium-a z vgrajenim Dart VM. Chromium je projekt, ki je nastal pod okriljem Google Chrome brskalnika in Google Chrome OS (https://www.chromium.org/). Pri uporabi Dartium-a nam ni potrebno predhodno prevajati programske kode Dart-a v programsko kodo JavaScript, ampak lahko izvajamo kar programsko kodo Dart. Prevajanje v kodo JavaScript je potrebno šele v fazi, ko želimo programsko rešitev preizkusiti v kakšnem drugem brskalniku, ki nima Dart podpore. Sintaksa razreda je sorodna programskim jezikom Java in `C#`. Enako kot TypeScript, nudi podporo opcijski uporabi tipov. To pomeni, da je uporaba tipov popolnoma poljubna. Pri posredovanju kode prevajalniku le-ta omogoča opozorila vezana na uporabo tipov. Na ta način rešuje ključna problema sintakse in semantike JavaScript-a. Ker se Dart tako zelo razlikuje od JavaScript-a, nima podpore za uporabo obstoječih JavaScript knjižnic znotraj Dart programske kode. Lahko pa uporabimo posebne skladne knjižnice, ki nudijo ovite verzije poljubnih JavaScript objektov. Na ta način Dart poskrbi, da je koda JavaScript na varni razdalji in da težave, ki jih ima jezik JavaScript, ne zahajajo v programsko kodo jezika Dart. Pri tem pa nastane težava, saj je razvijalec prisiljen v uporabo platforme. Trenutno prihaja do veliko izdaj novih in obstoječih knjižnic JavaScript, ki jih pri Dartu niso podprli do takšne mere, kot bi si želeli sami razvijalci. Uporaba ogrodja Dart bi torej bila kar velika cena, ki bi jo morali plačati, da bi se izognili uporabi JavaScript-a [14].

3.5 Kratek opis ostalih popularnih tehnologij za podporo izvajanju kode na strani odjemalca

3.5.1 ECMAS

ECMAScript ali ES je skriptni programski jezik z blagovno znamko, ki je bil standardiziran s strani Ecma Intenational v ECMA-262 in ISO(IEC 16262. ECMASCript je standard za skriptne programske jezike, JavaScript pa je jezik, ki temelji na ECMAScript standardu. Glavne funkcionalnosti JavaScript-a temeljijo na ECMAScript standardu, toda JavaScript ima še druge dodatne funkcionalnosti, ki pa ne ustrezajo standardu. JavaScript ni edini programski jezik, ki podpira ECMAScript standrd. Med najbolj znanimi so še ActionScript, uporabljen s strani Adobe Flash-a, ter JScript, uporabljen s strani Microsoft-a [15].

3.5.2 ASM.JS

ASM.JS je vmesni programski jezik, ki je zasnovan tako, da podpira uporabo računalniških programov, napisanih v programskem jeziku C, da se izvajajo kot spletne aplikacije, pri čemer bistveno bolje ohranja značilnosti delovanja od standardnega JavaScript-a. Programska koda, napisana v programskem jeziku, ki temelji na statični uporabi tipov z lastnim upravljanjem spomina (npr.: C), se prevede s pomočjo namenskega prevajalnika v točno določeno podmnožico kode JavaScript [16].

3.5.3 Web Assembly

WebAssembly ali krajše wasm je poskusni nizkonivojski programski jezik za uporabo v brskalnikih, ki je trenutno še v razvoju. Njegovo glavno poslanstvo je podpora prevajanju iz programskih jezikov C in C++, kasneje pa bodo podprti še mnogi drugi programski jeziki. Začetna verzija WebAssembly-ja

temelji na asm.js in prenosnem odjemalcu. Takoj po izvodu MVP je v načrtu podpora zbiranju smeti, kar bi WebAssembly naredilo za ciljno zbiranje za programske jezike, kot so Java in `C#`, ki so znani po uporabi zbiranja smeti. Razvojno ekipo sestavljajo inženirji iz podjetij Mozilla, Microsoft, Google in Apple. WebAssembly je definiran kot abstraktno sintaktično drevo AST, ki se shrani v binarnem formatu. To nam omogoča, da aplikacijo zgradimo iz manjših posameznih paketov [17].

Poglavje 4

Izbira primernih tehnologij za nadaljnji razvoj

4.1 Kriteriji primerjave razvojnih modelov .NET MVC in .NET Spletni obrazci

Ker sta razvojna modela zelo široko uporabljena, sem se znotraj diplomskega dela osredotočil na kriterije, ki so bili ključni pri odločanju o izbiri najustreznejšega razvojnega modela. Naročnik je pri naročilu nove spletne aplikacije OMS izrazil željo po novih funkcionalnostih, skladnih s korakom časa. Sledijo ključne zahteve naročnika, ki so vplivale na izbiro ustreznega razvojnega modela:

- Prva od novih zahtev je bila, da moramo razviti tudi spletni vmesnik API za vso funkcionalnost, ki smo jo podprli v okviru spletne aplikacije OMS. Spletni API mora biti izpostavljen preko spletnega vmesnika. V tej točki smo imeli na voljo, da vso to funkcionalnost izdelamo dvakrat, enkrat za uporabo v spletni aplikaciji OMS in eno, ki bi jo izpostavili kot spletni vmesnik API. Seveda pri razvoju tovrstnih spletnih aplikacij, kot tudi sicer na splošno, razvijalec nima neomejeno časa, prav tako ne finančnih in drugačnih virov, zato smo želeli skupno funkcionalnost

razvijati samo enkrat.

- Sledila je zahteva, da mora spletna aplikacija OMS delovati v spletnem brskalniku na vseh vrstah računalniških naprav, kot so osebni računalnik, prenosni računalnik, mobilni telefon in računalnik tablica. Na trgu obstaja več različnih načinov, kako doseči podporo na vseh omenjenih napravah, vendar pa vse temeljijo na dobri kontroli nad kodo HTML. V kolikor je kontrola nad kodo HTML slaba ali pa je sploh ni, potem tovrstnih orodij ne moremo uporabiti, saj je praktično nemogoče dodati atribute in CSS nastavitve na objekte, katerih unikatnih ključev ne moremo izračunati.
- Ena izmed internih zahtev na strani podjetja, v katerem sem zaposlen, in v okviru katerega smo novo programsko rešitev izdelali, je bila tudi boljša podpora testno vodenemu razvoju. Na preteklem projektu, ki je bil izdelan s pomočjo razvojnega modela .NET Spletni obrazci, smo imeli nemalo izkušenj s tem, da je razvijalec odpravil napako v programski kodi, vendar smo včasih šele čez kakšno leto ugotovili, da smo s posegom povzročili napako nekje drugje. Včasih so nas na to opozorili sami naročniki, nekatere napake pa smo odkrili tudi sami v okviru testiranja novih funkcionalnosti. Tovrstni pripetljaji so vedno nezaželeni, saj navadno naročnik oz. naročnikova stranka zahteva pojasnilo o dogodku ali pa, še huje, zahteva odškodnino, kot je navedeno v pogodbenih obveznostih izvajalca projekta do naročnika.
- In zadnja pomembna zahteva je bila, da mora biti aplikacija visoko odzivna in da nobena operacija ne sme presegati zadanih časovnih okvirjev. Če bi tovrstno odzivnost na podlagi novih časovnih okvirov merili pri spletni aplikaciji, razviti po razvojnem modelu .NET Spletni obrazci, bi aplikacija padla na testu. Ogromno podatkov je namreč v samem ViewStat-u ter bistveno premalo prisotnosti JavaScript funkcionalnosti, posledično manjša uporaba Ajax-a in jQuery knjižnic, kar na koncu pripelje do povečanja zakasnitev (latenc) in obremenitve spletnega strežnika. Zahteva mora namreč od odjemalca do strežnika veli-

kokrat preko povezav z veliko zakasnitvijo, kot npr.: mobilno omrežje in nazaj.

Do sedaj smo pregledali zahteve naročnika spletne aplikacije, sedaj pa sledi pregled ključnih kriterjiv, ki imajo vpliv na izpolnitev naročnikovih zahtev. Vsak kriterij je opisan v svojem podpoglavju.

4.2 Ločevanje programske kode (Soc)

SoC v kontekstu razvoja spletnih programskih rešitev pomeni, da želimo kar se da ločiti programsko kodo poslovne logike od programske kode, ki je namenjena generiranju HTML. Pri razvojnem modelu .NET MVC, je organizacija same kode vsebinsko razbita na tri ključne dele - model, pogled in kontroler - ki zagotavljajo čisto in pregledno ločevanje programske kode. Ta kriterij igra ključno vlogo pri sami prožnosti aplikacije, saj omogoča enostavnejšo razširitev funkcionalnosti ter poenostavi vzdrževanje take aplikacije.

Microsoftov poizkus ločevanja poslovne logike od kode HTML ni bil najbolje posrečen pri .NET Spletni obrazci. Sprva so bili vsi navdušeni nad tem, da je bila koda aplikacije prestavljena v ločen zaledni razdred. V realnosti se izkaže, da so razvijalci hitro začeli kodi med sabo mešati. Bodisi tako, da so v kodi namenjeni prezentaciji popravljali drevo komponent, za čigar nastanek skrbi strežnik. Druga skrajnost pa je npr. manipulacija z bazo podatkov znotraj istega code-behind razreda. Ali pa programsko kodo, namenjeno za prikaz, združili s programsko kodo, namenjeno za izvajanje na strežniku v eno datoteko. Končni rezultat je hitro lahko krhka in nerazumljiva koda.

Slika 4.1 prikazuje podporo posameznega razvojnega modela ločevanju programke kode.

Nivo podpore	Nima	Slaba	Zadostna	Absolutna
.NET Spletni obrazci		X		
.NET MVC				X

Tabela 4.1: Podpora ločevanju programske kode

4.3 Integracija s tehnologijami na strani odjemalca

Tehnologije na strani odjemalca so poglavitne pri sami hitrosti izvajanja aplikacije ter sami obremenitvi spletnega strežnika. Velikokrat je možno določene funkcionalnosti podpreti le na strani odjemalca, če le-ta ne zahteva prenos vseh podatkov v ali iz strežnika. V HTML dokumentu vsakemu gradniku pripada tudi ID, ki pa mora biti enoličen znotraj HTML dokumenta. Atribut ID je podprt v vseh najbolj razširjenih brskalnikih, kot so Mozilla, Internet Explorer, Firefox, Safari and Opera. Pri razvojnem modelu .NET MVC lahko za izgradnjo uporabniškega modela uporabimo različna orodja, med katerimi je najbolje integriran in podprt stroj Razor. Pri uporabi stroja Razor imamo ogromno kontrolo nad izgradnjo ID ključev, kar je ključno za enostavnejšo integracijo s tehnologijami na strani odjemalcev. Pri razvojnem modelu .NET Spletni obrazci pa nimamo kontrole nad ključi ID, zato je tudi integracija s tehnologijami na strani odjemalca precej omejena. Pri uporabi ugnezdenih kontrol se namreč bistveno zaplete tudi izgradnja samega ključa ID. Na strani odjemalca je tako težko predvideti, kakšen bo ključ nekega polja. V praksi pogosto pride do situacije, ki razvijalca prisili, da naredi povratno analizo povratnega klica (revers-engieniring postback event mehanizma), da lahko izračuna ID ključ ali pa mora po nepotrebnem vložiti čas, da doseže želen ID ključa. To pa je lahko velika ovira tudi za izkušene spletne razvijalce.

Slika 4.2 prikazuje podporo posameznega razvojnega modela integraciji s tehnologijami na strani odjemalca.

Nivo podpore	Nima	Slaba	Zadostna	Absolutna
.NET Spletni obrazci		X		
.NET MVC				X

Tabela 4.2: Podpora integraciji s tehnologijami, na strani odjemalca

Nivo podpore	Nima	Slaba	Zadostna	Absolutna
.NET Spletni obrazci			X	
.NET MVC		X		

Tabela 4.3: Podpora znanju in stopnji izkušenosti razvojnega oddelka

4.4 Znanje in stopnja izkušenosti razvojnega oddelka

Zelo pomembno pri izbiri razvojnega modela na strani spletnega strežnika je tudi znanje in izkušenost razvojne ekipe. Razvojni model .NET Spletni obrazci nam velikokrat prihrani veliko dela oz. poznavanja HTML-ja, saj se le-ta generira včasih brez naše posebne pozornosti. Pri tem modelu lahko hitro naredimo spletno stran, ki lahko pokaže določeno funkcionalnost, brez da bi razvijalec skrbel za samo kreiranje kode HTML. Stran je na prvi pogled videti enostavna in organizirana, pogled v kodo HTML pa nam razkrije, da je le-ta daleč od tega. Po drugi strani deluje razvojni model .NET Spletni obrazci precej enostavno, seveda za običajne scenarije in primere. V kolikor pa želimo doseči bolj specifično funkcionalnost, pa hitro naletimo na težave, povezane z življenjskim cikelom strani. Majhen odstotek razvijalcev po razvojnem modelu .NET Spletni obrazci res dobro pozna in razume celoten življenjski cikel strani.

Slika 4.3 prikazuje podporo posameznega razvojnega modela znanju in stopnji izkušenosti razvojnega oddelka.

4.5 Testno voden razvoj

Pri razvoju programske opreme so nam lahko v veliko pomoč avtomatski testi. Tukaj so v ospredju predvsem testi posameznih enot sistema (unit testi). Poznamo tudi druge vrste testov, kot so testi uporabniških vmesnikov in pa testi integracij. Pri izbiri razvojnega modela je ta kriterij lahko ključnega pomena, saj je količina programske kode, ki jo je možno enostavno in hitro testirati pomembna pri razvoju novih funkcionalnosti, pri čemer stara funkcionalnost ostane nespremenjena. V splošnem lahko to preverimo s pomočjo enotnih testov ali pa tako, da testiramo vse možne scenarije, ki so potrebni za test vseh funkcionalnih delov kode. Načrtovalci pri razvojnem modelu .NET Spletni obrazci so si težko predstavljali, da bo avtomatsko testiranje postalo ključna komponenta razvoja programske opreme. Razvojni model .NET Spletni obrazci zaradi velike povezanosti poslovne logike z uporabniškim vmesnikom ne omogoča pisanja enostavnih enotnih testov. Nasprotno pa je pri razvojnem modelu .NET MVC logika vsebovana v razredih, do katerih dostopamo samo iz nadzorniške komponente, ki priskrbi potrebne podatke, jih napolni v model ter določi pogled za izris le-teh. Ker je poslovna koda praktično nepovezana z ostalimi deli modela, je tudi pisanje enotnih testov enostavnejše ter lahko z njimi pokrijemo bistveno večji odstotek poslovne logike kot pri Forms modelu. Tak pristop pa omogoča, da poslovno logiko oz. funkcionalnost lahko razvijamo že v času priprave specifikacije, tako da za funkcionalnost napišemo test. Kasneje ob realizaciji gradnje spletne aplikacije pa delčke funkcionalnosti lahko že uporabimo. Za ta namen smo v našem podjetju enotno testiranje nadgradili še z dodatno funkcionalnostjo ponavljajočega testiranja. Ključna prednost takšnega sistema je avtomaski zagon testov po vsaki spremembi kode v sistemu za verzioniranje pri čemer delovna postaja samega razvijalca ni dodatno obremenjena.

Slika 4.4 prikazuje podporo posameznega razvojnega modela testno vodenemu razvoju.

Nivo podpore	Nima	Slaba	Zadostna	Absolutna
.NET Spletni obrazci		X		
.NET MVC				X

Tabela 4.4: Podpora testno vodenemu razvoju

4.6 Enostavnost razvoja in podpora razvojnemu modelu

Pomemben vidik pri izbiri tehnologije je tudi enostavnost razvoja in podpora, ki jo imamo na voljo v spletu. Razvojni model .NET Spletni obrazci je nedvomno enostavnejši in manj zahteven za uporabo, saj omogoča velik nabor že izdelanih komponent, ki jih ponudi v orodjarni komponent. Prav tako vsebuje več čarovnikov, ki poskrbijo za delni izdelek samo na podlagi izbire določenih parametrov, kar je enostavna rešitev za skrivanje kompleksnosti modela. Ker se razvojni model uporablja že od leta 2002, velja za zrelo tehnologijo na tržišču z veliko bazo uporabnikov ter neizčrpno zalogo primerov, vodičev in ne-nazadnje forumov, kjer je možno pridobiti informacijo, ki nas zanima pri samem reševanju problemov. Na drugi strani ima razvojni model .NET MVC bistveno manjši nabor komponent, ki so bolj primitivne in so namenjene za boljšo izkušnjo uporabe tehnologij na strani klienta, kot je to jQuery. Razvojni model .NET MVC je mlajši model, vendar ima prav tako že zelo veliko bazo uporabnikov, primerov, saj je bil zaradi svoje enostavne ločitve kode ter nudenju podpore na klientu zelo hitro priljubljen med spletnimi navdušenci.

Slika 4.5 prikazuje podporo posameznega razvojnega modela enostavnosti razvoja.

Nivo podpore	Nima	Slaba	Zadostna	Absolutna
.NET Spletni obrazci				X
.NET MVC			X	

Tabela 4.5: Podpora enostavnosti razvoja

Kriterij	Prioriteta	Boljši raz. model
Testno voden raz.	1	MVC
Integracija s teh. na strani odjemalca	2	MVC
Ločevanje programske kode	3	MVC
Znanje in stopnja izk. raz. odd.	4	Spletni obrazci
Enostavnost raz. in pod. raz. modelu	5	Spletni obrazci

Tabela 4.6: Primerjava podpore razvojnim modelom

4.7 Razrvrstitev kriterijev glede na pomembnost pri samem projektu

V prejšnjih poglavjih so bili predstavljeni ključni kriteriji za izbiro tehnologije za razvoj nove OMS aplikacije. Ker vsi kriteriji nimajo enake teže, sledi predstavitev kriterijev glede na pomembnost posameznega kriterija, na podlagi katere je potekal izbor razvojnega modela. Rezultati so predstavljeni v tabeli 4.6.

Cilj diplomskega dela ni opis izdelave spletne aplikacije, pač pa izbira primernega razvojnega modela. Sledeče poglavje predstavlja, kako se posamezen razvojni model obnese v praksi glede na izbrane kriterije.

Iz tabele je lepo razvidno, da se v najbolj pomembnih kriterijih bolje izkaže .NET MVC razvojni model. Na podlagi analize smo se v podjetju odločili, da bomo pri izgradnji nove spletne aplikacije OMS uporabili razvojni model .NET MVC.

Poglavje 5

Ovrednotenje razvojnih modelov glede na izbrane kriterije

Poglavje vsebuje ovrednotenje razvojnih modelov glede na izbrane kriterije na dejanskem primeru. Ovrednotenje temelji na obstoječi in novo-razviti spletni aplikaciji OMS. Izpostavljene so ključne prednosti in slabosti, ki so se pokazale pri izgradnji, vzdrževanju in obvladovanju spletne aplikacije OMS, razvite po enem in drugem razvojnem modelu.

5.1 Primeri .NET Spletni obrazci

5.1.1 Ločevanje programske kode

Kljub temu da .NET Spletni obrazci nudijo podporo ločevanju kode namenjene izrisu kode HTML in zaledni razred *.cs*, nas sam koncept ne sili, da bi se tega držali. Zato velikokrat pride do zlorabe, ko uporabnik vstavi kodo, ki se izvaja na spletnem strežniku, v kodo, ki je namenjena generiranju kode HTML. Slika 5.1 prikazuje primer tovrstne programske kode.

Tak način pisanja programske kode na večjih projektih je zelo slab, saj

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs"Inherits="_Default" %>
<script runat="server">

void Button1_Click(Object sender, EventArgs e)
{
 Label1.Text = "Clicked at " + DateTime.Now.ToString();
}

</script>
<html>
 <head>
  <title>Single-File Page Model</title>
 </head>
 <body>
  <form id="Form1" runat="server">
   <div>
    <asp:Label id="Label1" runat="server"></asp:Label>
    <br />
    <asp:Button id="Button1" runat="server"
     onclick="Button1_Click"
     Text="Click Me!">
    </asp:Button>
   </div>
  </form>
 </body>
</html>
```

Slika 5.1: Primer nepravilnega ločevanja kode

je tako kodo težje najti, prav tako pa tovrstne kode ni mogoče uporabiti v testih. Bolje je, da *aspx* datoteka vsebuje le programsko kodo, ki je namenjena generiranju kode HTML, kar je njen osnovni namen. Programsko kodo, ki ni namenjena generiranju kode HTML, pa je bolje prestaviti v zaledno *cs* datoteko. Na ta način dosežemo lepo ločevanje programske kode, kar posledično pomeni, da je tako spletno aplikacijo lažje vzdrževati, ji dodajati nove funkcionalnosti in spreminjati obseg obstoječih funkcionalnosti.

5.1.2 Znanje in stopnja izkušenosti razvojnega oddleka

Prvi problematičen primer s strani Web Forms razvojnega modela je potrebno poznavanje življenjskega cikla strani. V obstoječi OMS aplikaciji v različnih scenarijih želilmo nastaviti nekatere lastnosti kontrol na začetku. Pogosto v programski kodi najdemo spodnji pogojni stavek, na podlagi katerega samo pri prvem zahtevku, ko parameter isPostback ni nastavljen na true, želimo nastaviti vrednosti določenih lastnosti kontrol, kot prikazuje slika programske kode 5.2.

Da bi potrebo po poznavanju življenjskega cikla strani kar najbolje omejili, smo izdelali osnovni razred, ki implementira ključne dogodke življenjskega cikla .NET spletnega obrazca in jih izpostavili kot metode, ki jih mora vsaka spletna stran po potrebi ponovno implementirati. Ključna metoda je izbira modela, ki se uporablja za delovanje .NET Spletnega obrazca in je prikazana na sliki 5.3.

Najprej pokličemo privzeto implementacijo dogodka OnInitComplete, nato pridobimo parametre Modela glede na to, ali gre za povratni klic ali ne pa se implementaciji razlikujeta. V kolikor gre za prvi prihod na stran, je potrebno v sejo dodati nov privzeti model, sicer pa koda vrne podatke iz seje, kamor so bili nazadnje shranjeni. Tako smo poskrbeli, da v pravem trenutku posodabljamo Model spletne strani in v pravem trenutku osvežimo podatke strani HTML. Velikokrat se je namreč zgodilo, da je razvijalec posodobil Model strani v napačnem dogodku spletne strani in kot posledica rezultat sploh ni bil viden. V praksi je potem izgubil veliko časa, da je odkril razlog napake.

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    ShowDefaultView();
  }
}

public void ShowDefaultView()
{
  FillCombos();
  btnDelete.Enabled = Model.creauser.IsActive != false;
  btnActivate.Enabled = !btnDelete.Enabled;
}

private void FillCombos()
{
  // MA's & POS...
  dlMA.Items.Clear();
  List<TypeValue> tvs =
         Utils.AddSelectToSortedDropDown(GlobalDictionary.MasterAgent);
  foreach (TypeValue tv in tvs)
  {
    dlMA.Items.Add(new ListItem(tv.Description, tv.TypeValueId.ToString()));
  }
   dlPOS.Items.Clear();
   dlPOS.Items.Add(new ListItem(tvs[0].Description,
   RequiredDropDownValidator.MissingValue));
   dlPOS.Enabled = dlMA.SelectedValue != RequiredDropDownValidator.MV;
}
```

Slika 5.2: Posebna obravnava pri povratnem klicu

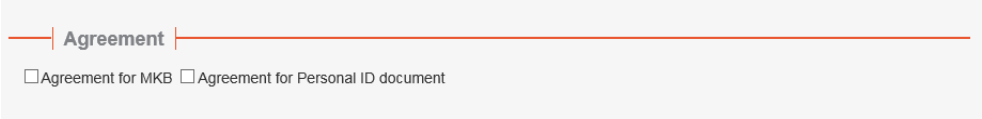
```
/// <summary>
/// Ovverride this to provide default model for testing
/// </summary>
/// <returns></returns>
public abstract object GetDefaultModel();

protected override void OnInitComplete(EventArgs e)
{
  base.OnInitComplete(e);
  pageState = new CreaOM.Common.GUI.Classes.PageState(this);
  GetParameters(out setModel);
  if (!IsPostBack)
  {
    object model=GetDefaultModel();
    setModel(model);
    pageState.SetPageState(model);
  }
  else
  {
    setModel(pageState.GetPageState());
    if (pageState.GetPageState() == null)
    {
      throw new Exception("Session timeout");
    }
  }
}
```

Slika 5.3: Primer uporabe dogodka življenskega cikla strani

5.1.3 Integracija s tehnologijami na strani odjemalca

Kot primer lahko izpostavimo težavo, ki nastane pri generiranju ID ključev, ki so poslani na odjemalčevo stran. Slika 5.4 predstavlja primer želenega uporabniškega vmesnika, izsek programske kode 5.5 pa primer kode .NET strani.



Slika 5.4: Izsek iz zaslonske maske

```
<fieldset id="fsAgreementSection" runat="server" visible="false">
  <legend>
    <asp:Localize ID="Localize1" runat="server" Text="<%$ Resources:
pageCustomerUserDataAgreement %>"></asp:Localize>
  </legend>
  <asp:CheckBox ID="cbMKBAgreement" runat="server"
       Text="<%$ Resources: cbMKBAgreement %>"  AutoPostBack="true"/>
  <asp:CheckBox ID="cbPersonalIdDocumentAgreement" runat="server"
       Text="<%$ Resources: cbPersonalIdDocumentAgreement %>"  AutoPostBack="true"/>
</fieldset>
```

Slika 5.5: Del kode .NET strani

V kolikor bi bila potrebna integracija s tehnologijami na strani odjemalca, bi se razvijalec zelo namučil, da bi našel pravi ključ HTML elementa, kot prikazuje primer programske kode 5.6. V primeru kode, ki je zadolžena za izgradnjo HTML dokumenta, lahko vidimo, da je kot ključ ID določena vrednost cbMKBAgreement, ko pa pogledamo vrednost ID ključa na spletni strani, ki jo dobi v izris spletni odjemalec, pa je vrednost $ctl00_ContentPlaceHolder1_customerEditor_cbPersonalIdDocumentAgreement$. Pozitivna stran takega ključa je, da razvijalec, ki ne pozna arhitekture spletne aplikacije, lahko hitro vidi poreklo elementa HTML, torej od kod izhaja.

5.1.4 Testno voden razvoj

V poglavju za primerjavo kriterijev v povezavi s testno vodenim razvojem smo že ugotovili, da ima razvojni model .NET Spletni obrazec precejšnjo

```
<fieldset id="ctl00_ContentPlaceHolder1_customerEditor_fsAgreementSection">
  <legend>
    Agreement
  </legend>
  <input id="ctl00_ContentPlaceHolder1_customerEditor_cbMKBAgreement"
type="checkbox" name="ctl00$ContentPlaceHolder1$customerEditor$cbMKBAgreement"
onclick="javascript:setTimeout('__doPostBack(\'ctl00$ContentPlaceHolder1$customerEdit
or$cbMKBAgreement\',\'\')', 0)" /><label
for="ctl00_ContentPlaceHolder1_customerEditor_cbMKBAgreement">Agreement for
MKB</label>
  <input id="ctl00_ContentPlaceHolder1_customerEditor_cbPersonalIdDocumentAgreement"
type="checkbox"
name="ctl00$ContentPlaceHolder1$customerEditor$cbPersonalIdDocumentAgreement"
onclick="javascript:setTimeout('__doPostBack(\'ctl00$ContentPlaceHolder1$customerEdit
or$cbPersonalIdDocumentAgreement\',\'\')', 0)" /><label
for="ctl00_ContentPlaceHolder1_customerEditor_cbPersonalIdDocumentAgreement">Agreemen
t for Personal ID document</label>
</fieldset>
```

Slika 5.6: Del kode HTML

pomanjkljivost. Ker pa .NET Spletni obrazci omogočajo boljši prototipni razvoj, smo aplikacijo najprej razvili, nato pa smo začeli pisati enotne in integracijske teste. Programsko kodo je bilo v splošnem zelo težko testirati, zato smo včasih za potrebe testiranja celotno naročilo serializirali v obliki datoteke XML, jo shranili v mapo, namenjeno testnim primerom, in jo uporabili znotraj testov. Glavni problem tega pristopa je, da je razvijalec zelo težko ugotovil, katere testne podatke za testiranje mora pravzaprav osvežiti, da bodo testi še uspešni. Kljub temu da je razvijalec vedel, katere testne podatke mora popraviti, pa mu je to vzelo veliko časa, saj je moral vnesti nov primerek. Skozi leta vzdrževanje se je tako pokazalo, da so testi pokrivali absolutno premajhen kos programske kode. Rezultat tega je bil, da smo marsikdaj popravili kakšno napako, zraven pa neopazno pridelali eno ali več novih, zato je bilo včasih potrebno več ciklov nameščanja popravkov, dokler nismo izločili vseh novonastalih težav.

5.1.5 Enostavnost razvoja in podpora razvojnemu modelu

Sledi opis na dejanskem projektu glede kriterija znanja razvojnega oddelka. Tukaj lahko kot pozitivno izpostavim, da smo se tekom razvoja projekta vsi

razvijalci iz podjetja lahko ne glede na stopnjo znanja razvoja spletnih aplikacij vključevali na projekt in izdelovali dele aplikacij, brez da bi v potankosti poznali HTML, HTTP in CSS. Razvoj .NET Spletnih obrazcev v razvojnem ogrodju VS namreč zelo poenostavi samo izgradnjo. V kolikor želimo na spletno stran dodati določen gradnik, ga lahko enostavno poiščemo v orodjarni komponent in dodamo z načinom »potegni in spusti«(drag and drop). Na ta način se nam samodejno doda koda za izris posameznege komponente. V razvojnem ogrodju nam potem nudi tudi enostavno dodajanje dogodkov. Če želimo dodati logiko, ki se izvede ob kliku na gumb, moramo najprej iz orodjarne potegniti in spustiti gumb na določeno mesto v aspx datoteki, z dvojnim klikom na omenjeni gradnik, pa se avtomatsko doda definicija dogodka v zaledno cs datoteko. Vse kar nam preostane je, da znotraj definiranega dogodka dodamo svojo logiko. Ta pristop izhaja iz načina razvoja namiznih aplikacij Windows, s katerimi se je spoznala večina razvijalcev v svojih prvih fazah razvoja programske opreme. Poglavje bi zaključil še z enim primerom dobre prakse. To je velika baza znanja in primerov, kar lahko z veliko mero prepisujemo dejstvu, da je razvojni model .NET Spletni obrazci zrela tehnologija, saj ima na trgu ogromno privržencev, po drugi strani pa že zaradi samega dejstva, da je razvojni model na trgu že od leta 2002.

5.2 Primeri .NET MVC

5.2.1 Ločevanje programske kode

Ločevanje programske kode na vsebinsko ločene dele programske kode, pogled, model in krmilnik nam je olajšalo samo izgradnjo funkcionalnosti API knjižnice. Ključnega pomena je koncept, ki razvijalca sili, da vso poslovno logiko lahko uporablja le znotraj krmilnika. Tako je bilo ključne dele poslovne logike lažje prestaviti v ločene razrede, ki so bili v začetku izpostavljeni le za uporabo v spletni aplikaciji OMS, kasneje pa je bilo iz teh razredov možno z nekaj truda izdelati tudi spletni vmesnik funkcionalnosti API knjižnice brez nepotrebne ponovne implementacije funkcionalnosti API knjižnice. Pri

tem smo si pomagali s posebnim označevanjem API metod, na podlagi katerega smo potem potrebne dele API knjižnice izpostavili tudi preko spletnega vmesnika. Dodatno pozornost so zahtevali določeni parametri metod, kot so tipi, ki imajo lahko ničelno vrednost, da se pravilno pretvorijo v ustrezne sheme WSDL. WSDL je zelo razširjen jezik za opis spletnih storitev. Kot problematično pa se je izkazalo, da je zaradi potrebe po API-ju in večji razslojenosti aplikacije prišlo do večkratnega preslikovanja, saj se je objekt na baznem nivoju pretvoril v objekt, primeren za uporabo znotraj pogleda. Največja omejitev večkratnega preslikovanja je bila, da smo imeli težave pri razvijanju funkcionalnosti, ki je bila potrebna za objekt, primeren za uporabo v pogledu, kot tudi objekt za uporabo nad samo bazo podatkov. Včasih smo skupno logiko lahko podprli z uporabo direktiv, namenjenih procesorju, znanih po angleškem imenu Ifdef [19].

```
#if CompileForAPI
        OMS.API.Model.Characteristic characteristic
#else
        OMS.Database.Model.Characteristic characteristic
#endif
```

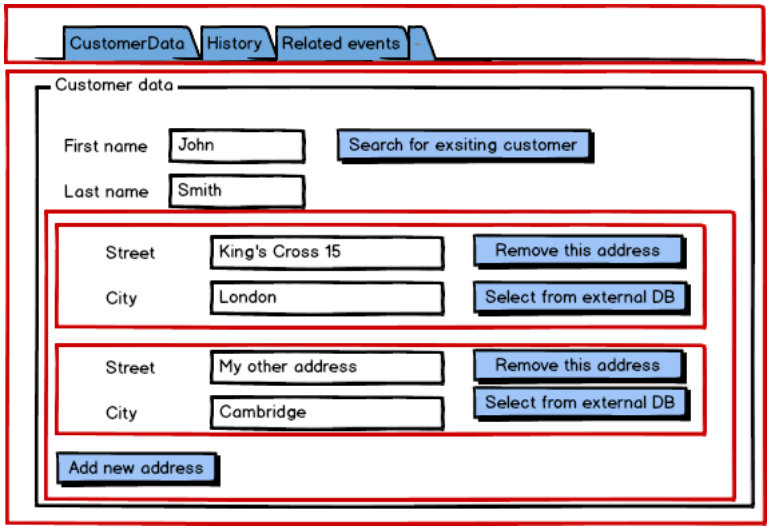
Slika 5.7: Uporaba direktive procesorju

V skrajnem primeru smo morali napisati omenjeno funkcionalnost dvakrat. Da bi se izognili težavam pri vzdrževanju, smo v obeh metodah dodali komentar z unikatno kodo, tako da je razvijalec, ki je zadolžen za dograditev ali popravljanje take funkcionalnosti, obveščen, da mora popravek razviti na dveh mestih.

5.2.2 Znanje in stopnja izkušenosti razvojnega oddelka

Ker je programski jezik .NET MVC bolj nizko nivojski, je seveda potreba po poznavanju razvojnega modela ključnega pomena. V podjetju smo se zato odločili, da probleme rešujemo na skupen način. Če je kak razvijalec naletel na problem, ki nam je že bil poznan, si je lahko rešitev implementacije pogledal na lokalni WIKI strani, kjer je bila tudi razložena. Eden od problemov

ASP.NET MVC je, da nima standardnega mehanizma za pisanje krmilnikov dogodkov za dogodke na različnih delih GUIja, ki je sestavljen iz komponent. "Komponente"pozna na nivoju pogledov, ki jih lahko razdelimo na več CSHTML datotek, ne pa na nivoju obravnave dogodkov. Primer malo bolj "kompleksnega"GUIja z več komponentami [4] prikazuje slika 5.8.



Slika 5.8: Oris zaslonske maske

Primer postavljenih pravila:

- Kontrolerji, ki uporabljajo dogodke, morajo dedovati iz CreaControllerBase.
- Dogodke in event handlerje je treba znotraj controllerja registrirati.
- Spletni obrazec mora vsebovati ustrezna polja, ki jih dodamo s $@Html.RednerPostFormToServerFields()$
- Na klientu sprožimo s povratnim klicem z metodo za pomoč, $Html.PostbackToServer$. primer: $< ahref = onclick = "@Html.PostFormToServer("PackageSelected", p.PackageId) \gg @p.PackageDisplayName < /a >$
- Glej CustomerController za primer.

Če komponenta drži referenco na del HTMLja, ki se zamenja zaradi Ajax povratnega klica, potem je treba te reference po povratnem klicu osvežiti,

drugače se lahko zgodi, da komponenta dela nad delom kode HTML, ki ni več v dokumentu. Posledica je nepravilno delovanje dogodkov. Primer: če komponenta kliče postBackToServer na spletnem obrazcu, ki ga ni več v kodi HTML, se bo zgodil celostranski povratni klic. To pomeni, da bi se po nepotrebnem prenašala vsebina celotne spletne strani, namesto da bi se prenesel samo delček strani.

5.2.3 Integracija s tehnologijami na strani odjemalca

Primer enostavne kode znotraj programskega jezika TypeScript, ki se prevede v enakovredno, berljivo kodo JavaScript, prikazuje slike 5.9.

Slika 5.10 prikazuje primer prevedene kode iz TS v JavaScript.

Prevajalnik poleg omenjene datoteke UserSearchClient.js, generira tudi UserSerarchClient.js.map, ki služi temu, da lahko uporabljamo razhroščevalnike tudi pri kodi TypeSctipt in ne samo pri kodi JavaScript. Orodja, ki omogočajo razhroščevanje te vrste, so npr. VS in pa razvojna orodja v brskalniku Chrome. Enak format zemljevida virov so uporabili tudi nekateri drugi prevajalniki v JavaScript, npr. CoffeeScript.

Seveda je potrebno omenjeno JavaScript funkcionalnost vgraditi na ustrezno mesto pri izgradnji HTML dokumenta. Na tem mestu smo samo vključevanje poenostavili na način, da vse JavaScript datoteke vključimo že v glavno spletno stran, ki poleg omenjenih datotek vključuje tudi CSS stile posameznih HTML elementov. Ker so vse JavaScript knjižnice dosegljive na vsakem koraku, je nato potrebna le še pravilna uporaba znotraj CSHTML datoteke. Najprej moramo na mestu, kjer želimo kodo JavaScript uporabiti, dodati spodnjo registracijo dogodkov, kot prikazuje slika 5.11.

V CSHTML datoteki nato na mestu, kjer želimo uporabiti omenjeno funkcionalnost, vstavimo kodo za izris gumba, čigar ID se mora ujemati z vhodnim parametrom v metodo JavaScript, kot prikazuje slika 5.12.

selectUser pa je metoda, definirana znotraj TypeScript razreda, ki ustvari UserSearchClient-a ter mu posreduje zahtevnae parametre, kot prikazuje slika 5.13.

```
enum UserSearchAction {
    UserSelected
}
interface UserSearchFinishedEvent { (action: UserSearchAction, userId?: string): void
};
class UserSearchClient {
  // Constructor
  dialogFinished: UserSearchFinishedEvent;
  constructor(public clientObject: JQuery) {
    this.attachEvents()
  }
  callDialogFinished(action: UserSearchAction, userId: string) {
    this.clientObject.on('hidden.bs.modal', () => {
      this.dialogFinished(action, userId);
    });
    this.clientObject.modal('hide');
  }
  enterPressed(eventObject) {
    if (eventObject.which == 13) {
      this.clientObject.find("#btnSearchUser").click();
    }
  }
  attachEvents() {
    this.clientObject.on("click", ".userSelected",
      (eventObject: JQueryEventObject) => {
        var targetElement = $(eventObject.currentTarget);
        var userId = targetElement.data("parameter");
        this.callDialogFinished(UserSearchAction.UserSelected, userId);
      });
    this.clientObject.on('hidden.bs.modal', () => {
      this.detachEvents();
    });
    this.clientObject.on("keyup",
      (eventObject: JQueryEventObject) => {
        this.enterPressed(eventObject);
      });
  }
  detachEvents() {
    this.clientObject.off("click", ".userSelected");
    this.clientObject.off('hidden.bs.modal');
    this.clientObject.off("keyup");
  }
  showSearchUser(dialogFinished: UserSearchFinishedEvent) {
    this.dialogFinished = dialogFinished;
    this.clientObject.modal({ 'backdrop': 'static' });
  }
}
```

Slika 5.9: Izsek kode TypeScript

```
var UserSearchAction;
(function (UserSearchAction) {
    UserSearchAction[UserSearchAction["UserSelected"] = 0] = "UserSelected";
})(UserSearchAction || (UserSearchAction = {}));
;
var UserSearchClient = (function () {
    function UserSearchClient(clientObject) {
        this.clientObject = clientObject;
        this.attachEvents();
    }
    UserSearchClient.prototype.callDialogFinished = function (action, userId) {
        var _this = this;
        this.clientObject.on('hidden.bs.modal', function () {
            _this.dialogFinished(action, userId);
        });
        this.clientObject.modal('hide');
    };
    UserSearchClient.prototype.enterPressed = function (eventObject) {
        if (eventObject.which == 13) {
            this.clientObject.find("#btnSearchUser").click();
        }
    };
    UserSearchClient.prototype.attachEvents = function () {
        var _this = this;
        this.clientObject.on("click", ".userSelected", function (eventObject) {
            var targetElement = $(eventObject.currentTarget);
            var userId = targetElement.data("parameter");
            _this.callDialogFinished(UserSearchAction.UserSelected, userId);
        });
        this.clientObject.on('hidden.bs.modal', function () {
            _this.detachEvents();
        });
        this.clientObject.on("keyup", function (eventObject) {
            _this.enterPressed(eventObject);
        });
    };
    ...
```

Slika 5.10: Izsek prevedene kode JavaScript

```
<script type="text/javascript">
$(this.clientObject).find("#btnSelectUser").click(
      () => { this.selectUser(); }
      );
</script>
```

Slika 5.11: Registracija dogodka JavaScript

```
<button type="button" id="btnSelectUser" class="btn btn-primary">
```

Slika 5.12: Koda za izris gumba

```
selectUser()
  {
    new UserSearchClient(
      $("#UserSearchWholeDialog")
      ).showSearchUser((action, id?) => { this.userSelected(action, id) });
  }
```

Slika 5.13: Metoda v jeziku TypeScript

Uporaba kode JavaScript v povezavi s kodo TypeScript je torej zelo enostavna pri razvojnem modelu .NET MVC. AddressSearchViewModel.cshtml primer, kako enostavno lahko uporabimo ajax funkcionalnost, da ni potrebno ponovno nalaganje celotne strani.

5.2.4 Testno voden razvoj

Kot primer testno vodenega razvoja lahko predstavim izgradnjo funkcionalnosti za prenos telefonske številke od drugega operaterja, k operaterju, za katerega smo izdelali spletno aplikacijo. Gre za veliko različnih scenarijev prenosa telefonske številke ter predvsem več različnih faz pri samem prenosu telefonskih številk, ki pa so standardizirane in regulirane s strani Agencije za komunikacijska omrežja in storitve republike Slovenije. Primer enostavnega testa prikazuje slika 5.14. Na spletni strani agencije je zapisano: "Končni uporabnik, ki želi zamenjati operaterja in svojo obstoječo telefonsko številko prenesti k drugemu operaterju, lahko to stori na prodajnem mestu izbranega operaterja, v omrežje katerega bi želel prenesti številko, tako, da izpolni zahtevo za prenos telefonske številke. Pri tem mora navesti operaterja dajalca številke (t.j. tisti operater, katerega storitve je do tistega trenutka uporabljal) in številko računa, ki ni starejši od treh mesecev."

Metoda $NPOrderCreate_CommonAsserts$ pa je zadolžena, da preveri, ali so res vsi vhodni podatki pravilno vneseni, da lahko operater začne postopek prenosa telefonske številke.

Kot prikazuje slika 5.15 smo poskrbeli, da je bila omenjena funkcionalnost sprva podprta kot test, kasneje pa smo jo vključili v dejansko aplikacijo na prava mesta. Poleg testov zgornjega tipa pa smo v spletni aplikaciji OMS

```
[TestMethod]
public void
PortabilityCallback_NPOrderCreate_PhoneInNationalFormat_ShouldCreateNewOrder()
{
  // arrange
  NPOrderCreateRequest request = ArrangeRequest(true, true,
  phoneNumber: "59912258");

  // act
  var calbackResponse = PortabilityCallback.NPOrderCreate(request);

  // assert
  NPOrderCreate_CommonAsserts(request, calbackResponse);
}
```

Slika 5.14: Primer testne metode

```
    private void NPOrderCreate_CommonAsserts(NPOrderCreateRequest request,
    NPOrderCreateResponse response)
    {
      // check response
      Assert.IsNotNull(response);
      Assert.AreEqual((int)NPOrderCreateResponseCodeEnum.OK, response.Code, "Test
failed because: " + response.Description +
(!response.OrderValidationMessages.IsNullOrEmpty() ? "; " + string.Join(", ",
response.OrderValidationMessages) : ""));
      Assert.IsNotNull(response.OrderId);
      Assert.IsTrue(response.OrderId != 0);

      var order = orderAPI.OrderGet(response.OrderId.Value);

      var rootOi = order.OrderItems.First(x => x.SpecCode == PlmTestConst.TopTrio);
      var actionOis = OrderContractExtensions.GetOmsActionsRecursive(rootOi);
      var oiPortOut = actionOis.FirstOrDefault(x =>
OrderContractExtensions.HasOmsActionType(x, PlmConst.OmsActionTypes.PortOut));

      // check portout
      Assert.IsNotNull(oiPortOut);
      var charInvoiceId =
OMS.Rules.API.CharacteristicBL.GetCharacteristicRecursiveFirst(oiPortOut,
PlmConst.CharCodes.CharInvoiceId);

      // check characteristics
      Assert.AreEqual(request.InvoiceId, charInvoiceId.CharValue);
    }
```

Slika 5.15: Primer preverjanje vhodnih podatkov

izdelali veliko število testov, ki testirajo že izdelane in uveljavljene oz. potrjene funkcionalnosti z razlogo, da jih kasneje pri vzdrževanju in nadgradnjah ne bi pokvarili ali spremenili. Primer testa prikazuje slika 5.16:

```
[TestMethod]
public void OrderCheckout_ShouldAddModems()
{
  //arrange
  var o = CreateValidOrderWithTopTrio();
  //act
  var checkoutResult = orderAPI.OrderCheckout(o.OmsOrderId);
  //assert
  Assert.IsTrue(checkoutResult.CheckoutStatusTypeId ==
    (int)CheckoutStatusTypes.CheckedOut);
}
```

Slika 5.16: Primer testne metode OrderCheckout

S testom, ki ga prikazuje slika 5.16, preverjamo, ali se je po zaključku naročila, ko le-tega ni več možno spreminjati, na naročilo dodal pravilen modem glede na specifike naročila, ki jih je zahteval naročnik. Logika izračuna najustreznejši modem in ga doda na naročilo znotraj klica metode $orderAPI.OrderCheckout(longOrderId)$. Znotraj omenjene metode se prav tako izvede preverjanje ustreznosti celotnega naročila. V kolikor preverjanje ni uspešno, med drugim tudi zaradi manjkajočega modema, se status naročila ne spremeni v status $CheckedOut$. Z dobro podporo izdelavi testov smo tako lahko po vsaki novi različici programske opreme precej bolj gotovi, da nismo česa nevede pokvarili, spremenili ali kakor koli drugače vplivali na drugo programsko kodo, ki ni predmet spremembe obdelujoče programske kode.

5.2.5 Enostavnost razvoja in podpora razvojnemu modelu

Tudi pri razvojnem modelu je podpora zelo široka. Na spletu nam je na voljo ogromno že izdelanih primerov, rešenih problemov in vsesplošnih razprav. Prav tako je za razvojni model .NET MVC razvitih veliko pripomočkov, ki omogočajo enostavnejšo uporabo, kot so Twitter Bootstrap in TypeScript. Primer uporabe Bootstrap tehnologije za enostavno delitev strani na stolpce:

Če želimo narediti labelo, ki zasede dva stolpca od dvanajstih, smo to do sedaj naredili, kot kaže slika 5.17 [20].

```
@Html.LabelFor(model =>
model.StreetNumber, new { @class = "control-label col-md-2" })
```

Slika 5.17: Primer uporabe orodja Twitter Bootstrap

Pri izgradnji novih delov kode HTML smo hitro nakopičili ogromno kode, ki je bila razvijalcu, ki ni sodeloval pri izgradnji, lahko zelo nerazumljiva, kar je privedlo do novih težav pri izgradnji kode HTML. Kasneje smo ugotovili, da se omenjena sintaksa uporablja na veliko mestih v spletni aplikaciji, zato smo jo prestavili v novo metodo za pomoč ter poenostavili pisanje tovrstnih primerov in po drugi strani bistveno povečali preglednost CSHTML datotek, kot prikazuje slika 5.18.

```
@Html.LabelForMD2(model => model.StreetNumber)
```

Slika 5.18: Poenostavitev uporabe metod za GUI

Res je, da razvojni model .NET MVC zaradi svoje nizkonivojske zasnove omogoča podporo za boljšo izdelavo gradnikov HTML, vendar v zameno zahteva odlično poznavanje razvojnega modela in soudeleženih pripomočkov. V kolikor le-ta znanja pri razvijalcu niso bila prisotna v dovolj veliki meri, je razvoj trajal bistveno dlje, kot bi bilo to potrebno. Zato je bilo na projektu pri razvoju nove spletne OMS aplikacije bistvenega pomena, da smo se razvijalci hitro učili, se trudili pisati transparentno programsko kodo in učinkovito predajali svoje znanje drug drugemu.

Poglavje 6

Zaključek

Še pred izborom tehnologij smo v podjetju, v katerem sem zaposlen, vedeli, da .NET MVC ni zamenjava za .NET Web Forms razvojni model. Gre torej za vzporedni tehnologiji, ki imata vsaka svoje prednosti in slabosti. Za novo spletno aplikacijo OMS smo se na koncu odločili za razvojni model .NET MVC. Ugotovili smo, da bomo vso zahtevano funkcionalnost lahko pokrili v obeh tehnologijah, vendar smo se na koncu vseeno odločili za .NET MVC. Ključni razlog za izbiro je odlična kontrola nad izgradnjo kode HTML ter odlična podpora jeziku TypeScript v razvojnem ogrodju Visual Studio. Na ta način smo lahko dosegli hitro odzivnost aplikacije, saj smo lahko minimizirali število klicev na spletni strežnik in količino podatkov, ki se prenaša med odjemalcem in spletnim strežnikom. Po drugi strani je pri razvojnem modelu .NET MVC ključno tudi to, da nam je testno voden razvoj omogočil hiter oz. prototipen razvoj določenih funkcionalnosti, ki smo jih kasneje lahko enostavno vključili v projekt OMS. Po drugi strani pa smo razvijalci naleteli na nove težave, povezane z odpravo napak, odkritih med izvajanjem spletne aplikacije. Nemalokrat je bilo znanje razvijalca enostavno premalo popolno, da bi lahko hitro odkril vir napake. Glavni problem za to je razpršenost ene GUI funkcionalnosti preko množice različnih tipov datotek, ki se v sami funkcionalnosti v velikih primerih uporablja implicitno. Tovrstnih težav je bilo pri razvojnem modelu .NET Web Forms bistveno manj. Zaradi omenje-

nih težav smo se morali v podjetju dogovoriti o določenih pravilih, ki smo jih zapisali na skupno lokalno WIKI stran, saj se je pred dogovorom začelo dogajati, da je vsak razvijalec pri reševanju problema uporabil svoj pristop, ki je bil velikokrat napačen in okoren, navadno zaradi pomanjkanja znanja o .NET MVC razvojnem modelu. Problem smo morali zajeziti na začetku projekta, saj bi v nasprotnem primeru naleteli na veliko težavo pri vzdrževanju projekta OMS. Splošno je namreč znano, da je življenjska doba tovrstnih aplikacij v povprečju do 10 let, kar je lahko zelo dolga in draga doba za podjetje, ki vzdržuje projekt. Kljub vsemu lahko trdim, da smo izbrali pravi razvojni model, kljub začetnim težavam. V podjetju smo mlada, samoiniciativna razvojna ekipa, ki se je vedno pripravljena učiti in je pri tem tudi zelo uspešna, zato nam to ni predstavljalo nepremostljivih ovir. V kolikor pa bi bila razvojna ekipa nefleksibilna in imela težave s sledenjem novim tehnologijam, pa bi bilo boje, da bi se poslužili .NET Web Forms tehnologije. Konec koncev tudi .NET Web Forms razvojni model ne tone v pozabo, kot je možno zaznati na raznih spletnih forumih, ampak se njegov razvoj še vedno nadaljuje, doplonjuje.

Slike

Literatura

[1] It is very easy to get started with .net core on your platform of choice. https://www.microsoft.com/net/core#windows.

[2] Introduction to asp.net web forms. http://www.asp.net/web-forms/what-is-web-forms.

[3] Asp.net page life cycle overview. https://msdn.microsoft.com/en-us/library/ms178472.aspx.

[4] How to handle actions from different parts complex viewmodels in asp.net mvc. http://stackoverflow.com/questions/19467748/.

[5] Mvc xerox parc 1978-79. https://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[6] Asp.net tutorial. http://www.w3schools.com/aspnet/default.asp.

[7] Adam Freeman. *Pro ASP.NET MVC 5*. Apress, 1. edition, 2013.

[8] Ecmascript. http://www.codebulb.ch/2015/09/the-javascript-typescript-coffescript-dart-choice-2015-part-1.html.

[9] Ecmascript. https://learn.jquery.com/ajax/, Marec 2015.

[10] Microsoft and google collaborate on angular 2 framework, typescript language. https://techcrunch.com/2015/03/05/, Marec 2015.

[11] Relation to javascript. http://stackoverflow.com/questions/12694530/.

[12] Relation to javascript. https://en.wikipedia.org/wiki/Dart_(programming_language).

[13] Dart (programming language). https://en.wikipedia.org/wiki/Dart_(programming_language).

[14] Jeff Walker. Why dart isn't the answer. http://walkercoderanger.com/blog/2014/03/dart-isnt-the-answer/, Marec 2014.

[15] Ecmascript. https://en.wikipedia.org/wiki/ECMAScript.

[16] John resig. http://ejohn.org/blog/asmjs-javascript-compile-target/.

[17] Eric Elliott. What is webassembly. https://medium.com/javascript-scene/what-is-webassembly-the-dawn-of-a-new-era-61256ec5a8f6#.h59xzgicu.

[18] Eric Elliott. The javascript / typescript / coffescript / dart choice. http://www.codebulb.ch/2015/09/the-javascript-typescript-coffescript-dart-choice-2015-part-1.html, September 2015.

[19] if (csharp reference). https://msdn.microsoft.com/en-us/library/4y6tbswk.aspx.

[20] Bootstrap. http://getbootstrap.com/.