LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ANGULARJS LIETOTNES MIGRĀCIJA UZ ANGULAR

BAKALAURA DARBS

Autors: **Edgars Andrucis**
Studenta apliecības Nr.: ea13036
Darba vadītājs: Dr.dat. Uldis Straujums

RĪGA 2017

ANOTĀCIJA

Darbā tiek aprakstītas un apkopotas dažādas stratēģijas un pieejas “AngularJS” projektu migrācijai uz jauno “Angular” ietvaru. Tai skaitā dažādi migrāciju priekšdarbi un stratēģijas, kā lielā sprādziena migrācijas stratēģija un inkrementālā migrācijas stratēģija. Šīs stratēģijas tiek novērtētas aprakstot to trūkumus un priekšrocības. Balstoties uz tā, darba autors veica tīmekļa pieteikumu formu “AngularJS” lietotnes migrāciju uz jauno “Angular” ietvaru. Pēc kā tas novērtēja apkopoto informāciju un papildināja to ar savām atziņām un novērojumiem.

Rezultāta tika nomigrēta tīmekļa pieteikumu formas lietotne un veikti secinājumi par projektu lietotnes migrēšanu, izvēlēto migrēšanas stratēģiju un priekšdarbu ietekmi uz migrācijas procesa atvieglošanu.

Atslēgvārdi: AngularJS, Angular, Migrācija, Inkrementālā stratēģija, Lielā sprādziena stratēģija, Pieteikumu formas.

ABSTRACT

AngularJS application migration to Angular

In this paper are described and summarized different strategies and approaches for migration of “AngularJS” projects to the new “Angular” framework. This information includes different migration preparation works and strategies like big bang migration strategy and incremental migration strategy. These strategies are evaluated by listing their pros and cons. Based on that the work author migrates “AngularJS” web claim form application to the new “Angular” framework and evaluates gathered information and complements it with his own observations and insights.

As the result web claim form has been migrated and conclusions were made about application migration, chosen migration strategy and the impact of migration preparation works on process facilitation.

Keywords: AngularJS, Angular, Migration, Incremental strategy, Big bang strategy, Claim forms.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

Angular – ir uz TypeScript bāzēts, atvērtā pirmkoda, klienta puses tīmekļa lietojumprogrammu ietvars.

AngularJS – ir uz JavaScript bāzēts, atvērtā pirmkoda, klienta puses tīmekļa lietojumprogrammu ietvars.

API – jeb lietojumu saskarnes programmatūra ir definīciju, protokolu un rīku kopums, kas paredzēts lietojumprogrammu izstrādē.

ASP.NET– ir atvērtā pirmkoda servera puses tīmekļa lietojumprogrammu sistēma, kas paredzēta dinamisku tīmekļu lapu izstrādei.

CommonJS – ir specifikācija, kas nosaka, kā "JavaScript" moduļi jāorganizē un jāizmanto.

CSS – ir lapas stila valoda, ko izmanto, lai aprakstītu dokumenta izskatu un prezentāciju rakstveida iezīmēšanas valodā.

DOM – jeb dokumenta objekta modulis ir tīmekļa specifikācija, kas apraksta dinamisko HTML dokumentu struktūru, tādā veidā, kas ļauj tos manipulēt iekš tīmekļa pārlūkprogrammās.

ES – jeb "ECMAScript" ir skriptu valodas specifikācija.

Git – ir versiju kontroles sistēma, kas uzskaita izmaiņas datora failos un tiek izmantota lai koordinētu darbu ar šiem failiem starp vairākiem cilvēkiem.

HTML – ir iezīmēšanas valoda standarts priekš tīmekļa lapām un lietotnēm.

Jasmine – ir atvērta pirmkodam testēšanas ietvars priekš "JavaScript" programmām.

JavaScript – ir augsta līmeņa, dinamiska, bez tipu, un interpretējama izpildes laika programmēšana valoda.

jqLite – ir niecīga, "API" saderīga "jQuery" apakškopa, kas ļauj "AngularJS" ietvaram manipulēt "DOM" ar pārrobežu pārlūku saderīgā veidā.

jQuery – ir starp platformu "JavaScript" bibliotēka, kas izveidota, lai vienkāršotu klienta puses skriptu veidošanu, lai manipulētu HTML.

Karma – ir "JavaScript" testu izpildīšanas rīks, kas tika izveidots priekš "AngularJS" ietvara.

MVC – ir programmatūras arhitektūras modelis ar ko tiek īstenotas lietotāja saskarnes lietotnes.

OOP – jeb objektorientētā programmēšana ir programmēšanas paradigma, kurā programma tiek veidota izmantojot objektus un klases.

Protractor – ir lietotnes saskarnes testa ietvars priekš "AngularJS" ietvara lietotnēm.

SystemJS – ir moduļu ielādētājs, kas var importēt moduļus palaišanas laikā kādā no populārākajiem formātiem.

TypeScript – ir atvērtā pirmkoda programmēšana valoda, kura ir stingra “JavaScript” valodas kopa.

Typings – ir “JavaScript” bibliotēka, kas kalpo kā “TypeScript” definīciju pārvaldnieks

TSD – ir “JavaScript” bibliotēka, kas kalpo kā TypeScript definīciju pārvaldnieks izmantojot “Git” repozitoriju ar nosaukumu “DefinitelyTyped”. Šī bibliotēka ir novecojusi un netiek rekomendēta lietošanai, tā vietā tiek rekomendēts izmantot “Typings” bibliotēku.

Webpack – ir moduļu ielādētājs un pakotājs.

IEVADS

Darba autors patreizējā darba vietā izstrādā apdrošināšanas tīmekļa pieteikumu formu lietotni, kura tiek izstrādāta ar "AngularJS" ietvara palīdzību. Projekts tiek izstrādāts spējā un atrodas pastāvīgā izmaiņu stadijā, sakarā ar kompānijas filozofijas maiņu būt klientu centrētiem un uzlabot lietotāju pieredzi ar kompānijas produktiem. Tādēļ projekta lietotnei tiek bieži manītas prasības un izstrādātāji kopā ar biznesu ir patstāvīgi izaicināti radīt klientam draudzīgāku un inovatīvāku lietotni nekā konkurentiem.

Projekts tika uzsākts 2016. gada sākumā un kopš tā brīža ir piedzīvojis vairākas iterācijas, kā arī ir ticis palaist produkcijā. Taču neskatoties uz to, projekts ir tikai sākuma stadijā un ir paredzēts vēl izstrādāt daudzas pieteikumu formas.

Kad 2016. gada beigās tika oficiāli izlaists jaunais "Angular" ietvars, izstrādātāju komanda uzstādīja mērķi izpētīt un novērtēt migrācijas iespējas uz šo jauno ietvaru. Ņemot vērā, ka lietotne tiks uzturēta un izstrādāta vēl ilgu laiku, laicīga migrācija uz jaunāku ietvaru šķita kā labs un tālredzīgs solis lietotnes panākumu veicināšanai nākotnē[1].

"Angular" ietvars nav vienkāršs versijas jauninājums esošajam "AngularJS" ietvaram, bet gan no pamatiem izveidots jauns ietvars. Vienīgais risinājums kā to sākt izmantot esošajās lietotnes projektos ir migrēt lietotnes pirmkodu uz jauno ietvaru, kas ir nepatīkams, laikietilpīgs un riskants process, kura laikā var sabojāt vai zaudēt jau esošās lietotnes funkcionalitātes.

Darba mērķis un uzdevums ir izpētīt un novērtēt dažādas migrācijas stratēģijas un riskus, kas ar tām ir saistītas. Kā arī apkopot citu izstrādātāju pieredzi, ieteikumus, pamācības, labās prakses un biežāk sastopamās problēmas un kā no tām izvairīties, ai balsoties uz tām veiktu nepieciešamos sagatavošanas darbus un veiktu tīmekļa pieteikumu formu lietotnes migrāciju. Pēc kā arī tos novērtēt, papildināt un izdarīt secinājumus par lietotnes migrēšanu kopumā.

Darbs pamatā tiek balstīts uz "Angular" izstrādātāju dokumentācijas un citu izstrādātāju rakstiem, kuros tie dalās ar savām lietotņu migrācijas pieredzēm, izsakot padomus, ieteikumus un pamācības, kas jāņem vērā pirms migrēt un kā pareizi sagatavoties un veikt lietotnes migrēšanu.

Darbs sastāv no četrām pamata nodaļām, kur pirmajā nodaļā tiek aprakstītas jaunā ietvara priekšrocības un migrācijas šķērsļi, otrajā nodaļa tiek apkopoti dažādi priekšdarbi, kurus veicot ir iespējams atvieglot lietotnes migrāciju, trešajā tiek aprakstītas iespējamās migrācijas stratēģijas kā lielā sprādziena un inkrementāla stratēģija un ceturtajā tiek aprakstīta darba autora praktiski veiktā lietotnes migrācija. Darba beigās autors apkopo darba rezultātus un secinājumus.

1. PĒTĀMĀS PROBLĒMAS APRAKSTS

2014.gada septembrī "ng-Europe" konferencē Parīzē tā brīža "AngularJS" izstrādātāji pasludināja, ka tiks izdota jauna ietvara iterācija ar nosaukumu "Angular 2.0"[1]. Šis paziņojums izraisīja daudzas diskusijas un satraukumu izstrādātāju vidū, jo pēc "AngularJS" veidotāju vārdiem, šī iterācija, tika vērienīgi izmanīta salīdzinot ar esošo ietvaru, ne tikai ar jaunām funkcionalitātēm, bet arī ar lielām sintakses izmaiņām, izstrādes pieeju un filozofiju, kā rezultātā izstrādātāju priekšā stājās grūta izvēle, vai migrēt uz šo jauno ietvara iterāciju[2].

2016.gada septembrī beidzot tika oficiāli izlaista "Angular 2.0" versija[3]. Neilgi pēc tam šī paša gada decembrī tika paziņots par jau nākamo lielo versijas atjauninājumu. Kā ar tika pieņemts, ka veco "Angular 1.x" versijas ietvaru sauks par "AngularJS" un jauno ietvara versiju un visas tās turpmākās versijas sauks par "Angular"[4].

"Angular" nav parasts versijas uzlabojums jau esošajam "AngularJS" ietvaram, bet gan pilnīgi jauns projekts, kurš tika veidots ar labākajām jau esošā ietvara funkcionalitātēm prātā un atmetot visus tā trūkumus.

Ņemot vērā, ka "Angular" ir pilnīgi jauns ietvars, tas nozīmē, ka eksistējošos "AngularJS" projektus produkcijā vai vēl izstrādēs stadijā nevar vienkārši pārlikt uz jauno ietvaru, bet gan ir nepieciešams migrēt, kas sevī ietver pilnīgu pirmkoda pārrakstīšanu, gan no sintakses, gan no loģikas puses.

Projekta pirmkoda pārrakstīšana ir viena no nepatīkamākajām izstrādātāju nodarbēm, jo tā ir ne tikai bieži vien dārga un laikietilpīgā, bet arī ietver sevī risku sastapties ar jaunām problēmām un sabojāt jau strādājošu risinājumu, tādēļ izstrādātāji izvairās no radikālām izmaiņām pirmkodā īpaši jau produkcijā izmantojamiem projektiem. Kā arī reālā darba vidē izstrādātājiem ir grūti pārliecināt biznesa cilvēkus par projekta migrēšanas vajadzību, kad pastāv tik lieli riski un tēriņi tā realizācijai.

Tomēr "Angular" ietvars sevī iekļauj daudz labus jauninājumus un vērtīgas funkcionalitātes, kas ilgtermiņā spēj nest projekta izstrādei nozīmīgu pievienotu vērtību un nodrošināt, ka arī nākotnē ar to varēs realizēt visas jaunākās prasības un lietotne pareizi funkcionēs uz jaunākajiem pārlūkiem.

Ievērojamākās "Angular" ietvara jaunās iezīmes ir sekojošas:

- Uzsvars uz mobilo tehnoloģiju – ietvars pats rūpējas par mobilo ierīču veiktspēju.
- Modularitāte – lielākā kodola funkcionalitāte tika pārvietota uz moduļiem, kas samazināja tās smagumu un palielināja tās ātrumu.

- Mūsdienīgas pārlūkprogrammas – samazināja nepieciešamību, pielāgoties atšķirīgu pārlūkprogrammu prasībām.
- Uzlabota atkarību inžekcija.
- Dinamiskā lejupielāde.
- Vienkāršāka maršrutēšana.
- Vienkāršāka sintakse.
- Kontrolieru un tvēruma aizvietošana ar komponentēm un direktīvām.
- Asinhrona šablonu kompilācija.
- Iebūvēta "Diary,js" logošana, kas palīdz noteikt atsevišķu funkciju darbības laikus un atrast sastrēgumus programmā.
- "TypeScript" – skriptu valoda, kas sevī iekļauj tādas funkcionalitātes, kā iterāciju, refleksiju, "for/of" ciklēšanu, "OOP", vispārīgumu un lambdas.[5]

Daudzas no šīm "Angular" ietvara funkcionalitātēm var būtiski uzlabot, projekta izstrādi un tālāko uzturēšanu, gan projektiem, kas ir tikai plānošanas stadijā, gan jau eksistējošām vai izstrādes stadijā esošām "AngularJS" lietotnēm. Tomēr pastāv zināma neskaidrība par "AngularJS" migrēšanas procesu uz "Angular" ietvaru. Tādēļ ir nepieciešams izpētīt doto procesu, atrodot labākās migrācijas pieejas konkrētajiem projekta veidiem. Aptverot vidējo migrācijas laikietilpīgumu un nepieciešamo resursu un zināšanu daudzumu, lai veiksmīgu spētu nomigrēt dažādas "AngularJS" lietotnes un laicīgi izvairītos no tipiskākajām kļūdām un problēmām.

Kaut arī oficiālajā "AngularJS" dokumentācijā ir virspusīgi aprakstīts ieteicamais migrācijas process ir nepieciešams saprast, ko tieši tas nozīmē un kā tas izpaužas un tiek implementēts reālos tīmekļu lietotņu projektos, kuri ir nepārtrauktā izstrādē un jau atrodas produkcijā. Ir nepieciešams atrast un apkopot vērtīgākās atziņas no izstrādātāju rakstiem un pamācībām, kuros tie dalās ar pieredzi un ieteikumiem, un uz to pamata veikt tīmekļa formu migrēšanu,

2. PIRMS MIGRĀCIJAS PRIEKŠDARBI

“AngularJS” lietotņu migrēšanas sarežģītība un laikietilpīgums ir atkarīgs no konkrētās lietotnes un tās pirmkoda. To ir iespējams samazināt sekojot “Angular” izstrādātāju ieteikumam pirms migrācijas uzsākšanas veltīt laiku eksistējošā pirmkoda uzlabošanai, kas atvieglos lietotnes migrēšanu, it īpaši, ja tā tiks veikta inkrementāli. Papildus tam, šie migrēšanas priekšdarbi var uzlabot esošo lietotnes pirmkodu padarot to vieglāk uzturamu, neatkarīgāku un saderīgāku ar jaunākajām izstrādes bibliotēkām[6].

Šajos migrēšanos priekšdarbos pamatā tiek aprakstītas lietotņu pirmkoda un struktūras uzlabošanas metodes, kas balstītas uz “AngularJS” izstrādātāju stila vadlīnijām, kurās tiek apskatītas labās un sliktās lietotnes pirmkoda izstrādes prakses[6].

Bez pašu “AngularJS” izstrādātāju migrēšanas ieteikumiem ir arī minēti vairāki citu izstrādātāju ieteikumi un brīdinājumi, kas balstīti uz konkrētiem piemēriem, migrējot reālas lietotnes un projektus.

Darba autors lielāko daļu šo priekšdarbu izmantoja savas tīmekļa pieteikumu formu lietotnes migrēšanā, kā arī daži no tiem jau bija implementēti iepriekš. Tādēļ šie pirms migrācijas priekšdarbi ir arī papildināti ar autora konkrētajām lietotnes migrēšanā gūtām atziņām.

2.1. Versijas jauninājums

Viens no pirmajiem migrācijas priekšdarbiem ko būtu vērts darīt, ir esošās “AngularJS” bibliotēkas atjaunināšana. Svarīgi, lai “AngularJS” versija būtu ne vecāka par “AngularJS 1.5” versiju, jo šī versija iekļauj sevī daudzas palīgfunkcijas un jauninājumus, kas tika implementēti ar domu atvieglot lietotņu migrēšanu un loģiski pietuvinātu pirmkoda struktūru pie jaunā “Angular” ietvara[16].

Šī darba rakstīšanas laikā jaunākā “AngularJS” versija ir “AngularJS 1.6”, bet darba autors neuzskata par nepieciešamu atjaunināt lietotnes līdz šai versijai, lai varētu veiksmīgi nomigrēt “AngularJS” lietotnes uz “Angular” ietvaru. Sakarā ar to, ka šis jauninājums pamatā izlaboja iepriekšējās versijas kļūdas un pievienoja jaunas funkcionalitātes, nepienesot nekādu pievienoto vērtību pie migrēšanas uz jauno “Angular” ietvaru, kā to dara “AngularJS 1.5” versija[16].

Atjauninot “AngularJS” versiju, papildus ir jāņem vērā vai projektā tiek izmantota “TypeScript” programmēšanas valoda un kāds no “TypeScript” definīciju pārvaldniekiem, kā “TSD” vai “Typings”. “TypeScript” programmēšanas valodas definīcijas būs nepieciešams

atjaunot uz attiecīgo versiju, kā rezultāta var rasties nepieciešamība papildināt lietotnes pirmkodu, lai tas spētu kompilēties bez kļūdām.

Protams šo migrēšanas priekšdarbu ir iespējams izlaist, ja netiek lietota inkrementālā migrācijas stratēģija, bet gan tiek migrēta visa lietotne uzreiz. Tomēr darba autors uzskata, ka pat šajā gadījumā versijas atjaunināšana ir vērtīga, lai laicīgi un nesāpīgi varētu atklāt jau esošo slikto pirmkodu un funkcionāli loģiskos migrācijas šķēršļus. Kā arī novērtēt migrācijas sarežģītību un optimālo stratēģiju.

2.2. Struktūras izmaiņas

Pirmā lielā atšķirība starp "AngularJS" un "Angular" ietvariem ir veids kā to lietotņu projektos tiek strukturēti faili un organizēts pirmkods.

"AngularJS" lietotni veido direktīvas, kontrolieri un servisi, kas var tikt organizēti dažādos veidos. Piemēram, var pieturēties struktūrai, kas ir līdzīga "MVC" ietvaru struktūrai, kā tas ir redzams *2.1.* attēlā, kur faili tiek sadalīti mapītēs pēc to veida kā skati, direktīvas, servisi u.t.t..

```
app
| - controllers
    | - mainController.js
    | - otherController.js
| - directives
    | - mainDirective.js
    | - otherDirective.js
| - services
    | - someService.js
    | - otherService.js
| - js
    | - externalLibrary.js
    | - otherExternalLibrary.js
app.js
views
    | - mainView.html
    | - otherView.html
    | - index.html
```

2.1. att. **Projekta struktūras kas līdzīga "MVC" ietvara struktūrai**

Autora migrētajā tīmekļa pieteikumu formu lietotnē faili tika organizēti, kā redzams *2.2.* attēlā, kur direktīvas failos atradās arī attiecīgās direktīvas kontrolieri un moduļi.

```
app
| - services
    | - someService.js
    | - otherService.js
| - components
    | - someComponent
        | - someComponent.html
        | - someComponentDirective.js
    | - otherComponent
        | - someComponent.html
        | - someComponentDirective.js
app.js
| - dist
    | - externalLibrary.js
    | - otherExternalLibrary
```

2.2. att. **Autora migrētās tīmekļa pieteikumu formas lietotnes projekta struktūra**

Taču abas šīs projektu struktūras neseko “AngularJS” stila vadlīnijām, kā rezultātā tas var apgrūtināt projekta migrēšanu uz “Angular” ietvaru. Šādām un vēl citām projektu struktūrām, kas neseko ieteiktajām vadlīnijām pastāv risks, ka kādi svarīgi pirmkoda gabali tiks pazaudēti pie migrēšanas. Tādēļ ir nepieciešams pārveidot projekta struktūru, kas līdzinātos “Angular” labās prakses projektu struktūrai, kura ir redzama *2.3.* attēlā un ir aprakstīta oficiālajā “Angular” stila vadlīniju dokumentācijā[17]. Visi komponentes saistītie faili tie grupēti zem vienas mapes.

2.3. att. **“Angular” projekta struktūras sadalījums pēc labās prakses**

Svarīgākās lietas pie kā jāpieturas strukturējot “Angular” vai “AngularJS” projektu ir definētas “AngularJS” sila vadlīniju dokumentācijā. Tās ir, ka katram failam ir jāatbild tikai par vienu lietu, kas varbūt serviss, komponente, modulis, palīgfunkcija u.t.t.. Pēc faila nosaukuma ir jāsaprot, kas tajā atrodas, tādēļ jāizvairās no vairāku komponenšu vai palīgfunkciju likšanu vienā failā. Papildus tam, faila nosaukumus jāveido konsekventi. Failus jāgrupē pēc komponentēm, kur visi faili kas saistīti ar komponenti atrastos vien vietā tai skaitā testi, stila un skata faili. Šādas struktūras mērķis ir panākt katras komponentes neatkarību un vienkāršu uzturēšanu, kur veicot kādas komponentes pirmkoda izmaiņas uzreiz ir pārskatāmi kādas citas komponentes tas ietekmē un kurus failus vēl ir nepieciešams papildināt[17].

Šī struktūra ir svarīga lielām lietotnēm ar daudzām komponentēm, kuras nepārtraukti tiek papildinātas un uzlabotas gan izstrādes laikā, gan jau pēc lietotnes izlaišanas produkcijā.

Šāda struktūra ir īpaši ērta darba autora tīmekļa pieteikumu formu lietotnes gadījumā, kur ir liels daudzums dažādu formu komponenšu. Tās pastāvīgi tiek mainītas un uzlabotas, un ir arī nepieciešamība pēc to versiju kontroles, gadījumos kad atsevišķās formās tiek testētas kādu komponenšu jaunās versijas saglabājot paralēli vecās komponenšu versijas.

2.3. Moduļu ielādētāju izmantošana

Pēc lietotnes projekta struktūras izmaiņām pa vairākām komponentēm ir jāiegūst projekta struktūra, kurā ir liels daudzums mazu failu. Tādā veidā izveidojot struktūru, kurā ir viegli atrast konkrētās komponentes, klases un funkcijas, jo tās visas atrodas atsevišķos failos. Tomēr šādas pieejas mīnuss ir kad tā apgrūtina skriptu ielādi "HTML" lapā, jo tad ir jāuzskaita lielais failu daudzums un ir jākontrolē to ielādes kārtība. Tādēļ ir svarīgi, lai projektā tiek izmantots moduļu ielādētājs.

Modeļu ielādētāji nodrošina, kad viss lietotnes pirmkods ir ne tikai iekļauts un ielādēts "HTML" lapās kur tas nepieciešams, bet arī kad tas tiek darīts pareizajā secībā. Kā rezultātā atvieglojot darbu lietotnes izstrādātāju komandai. Papildus, moduļu ielādētāji vēl nodrošina lietotņu skriptu pakošanu priekš lietotņu produkcijas vidēm[18].

Ir liela izvēle starp moduļu ielādētājiem, kā "SystemJS", "Webpack" un "CommonJS". Katram no šiem moduļu ielādētajiem ir savas priekšrocības un trūkumi, un katrs ir piemērotāks vienam vai citam projektam.

Piemēram, "SystemJS" ir ļoti vienkārši konfigurējams un tas ielādē konkrētos "JavaScript" failus, tikai tad, kad tas ir nepieciešams. Rezultātā lietotnes lapa ielādējas daudz ātrāk. Savukārt "Webpack" ir mazliet sarežģītāks moduļu ielādētājs, kurš minimizē un sapako visus lietotnes failus vienā kopīgā failā. Kā rezultātā lietotne darbojas zibenīgi ātri, pēc mazliet ilgākas sākotnējās ielādes. Un "CommonJS" ir arī ļoti vienkāršs moduļu ielādētājs, kurš ir piemērots servera puses lietotnēm, jo skriptu ielāde notiek sinhroni strikti noteiktā kartībā[18].

Moduļu ielādētāju ir īpaši vērts izmantot, ja projektā tiks izmantota "TypeScript" programmēšanas valoda. Tie nodrošina to, lai pirmkodā varētu tikt importētas un eksportētas dažādas klases. Kā arī, lai tās varētu tikt izsekotas, ejot dziļumā un uzreiz nokļūstot pie tām un to funkciju implementācijām. Papildus, kompilators uzreiz spēs atpazīt konkrētās klases atribūtus un metodes, kuras tiek izmantotas svešos failos un laicīgi pamanīt drukas kļūdas funkciju un atribūtu nosaukumos, un nepareizi padotos vai saņemtos parametrus.

Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnē jau tika izmantota "TypeScript" valoda kopā ar "SystemJS" moduļu ielādētāju. Šis ielādētājs ir vispiemērotākais, jo tas ielādē nepieciešamos skriptus tikai pēc vajadzības. Tas ir ļoti ērti dinamiskās pieteikumu formās, kur visi lauki un sekcijas uzreiz nav nepieciešamas, tādēļ forma var tikt ielādēta bez aizkavēšanās un tās paslēptās sekcijas tiek ielādētas tikai, kad lietotājs līdz tām nokļūst.

2.4. Migrācija uz “TypeScript”

Atšķirībā no “AngularJS” ietvara kurš tika veidotas uz “JavaScript” programmēšanas valodas bāzes, “Angular” ietvars tika veidots uz “TypeScript” programmēšanas valodas bāzes, tādēļ arī tā izstrādātāji iesaka rakstīt “Angular” lietotņu kodu “TypeScript” programmēšanas valodā. Zīmīgi ir arī tas, kad visa oficiālā “Angular” dokumentācija ir rakstīta tieši ar “TypeScript” piemēriem, lai pēc iespējas vairāk atvieglotu izstrādātājiem tā lietošanu un apgūšanu savos projektos[6].

“TypeScript” programmēšanas valodas izmantošanas priekšrocība ir, ka ar to var ērti dalīt kodu moduļos. Kā arī pati pirmkoda uzbūve ir daudz lasāmāka un tuvāka “OOP” nekā “JavaScript” programmēšanas valodā veidotais pirmkods. Izmantojot “TypeScript” programmēšanas valodu var dalīt lietotnes servisus, komponentes un kontrolierus klasēs, kā arī piešķirt mainīgiem dažādus tipus un pievienot failiem anotācijas. Kā rezultātā tiek palielināts, ne tikai lietotnes izstrādes ātrums, bet arī tiek atvieglota pirmkoda atkļūdošanu. Attēlos *2.4.* un *2.5.* ir redzams vienkāršs salīdzinājums starp “TypeScript” programmēšanas valodā veidoto pirmkodu, kurā tiek definēta klase un ekvivalentu “JavaScript” programmēšanas valodā veidoto pirmkodu, kurā tiek definēta tieši tā pati klase.

```typescript
//TypeScript
class HelloWorld {
    sayHello(text = "World") {
        console.log("Hi $(text)!!!!");
    }
}

var helloWorld = new HelloWorld();
helloWorld.sayHello();
```

2.4. att. **“TypeScript” pirmkoda piemērs helloWorld.ts**

```javascript
//JavaScript
"use strict"
var HelloWorld = function HelloWorld() {};
($traceurRuntime.createClass)(HelloWorld,(sayHello: function() {
    var text = arguments[0] !== (void 0) ? arguments[0] : 'World';
    console.log(("Hello"+ text + "!!!!"));
}}, {});

var helloWorld = new HelloWorld();
helloWorld.sayHello();
return {};
```

2.5. att. **“TypeScript” pirmkoda “JavaScript” ekvivalents pirmkoda piemērs helloWorld.js**

Protams “TypeScript” programmēšanas valodas izmantošana nav obligāta, lai veidotu “Angular” projektus, ir iespējams tos izstrādāt arī ar “JavaScript” programmēšanas valodu. “Angular” izstrādātāji ir parūpējušies par dokumentāciju, gan “TypeScript”, gan “JavaScript”

programmēšanas valodām, tādēļ abas šīs opcijas ir vienlīdz labas. Tādēļ ātrākai migrācijai, ja esošais "AngularJS" projekts jau netiek rakstīts "TypeScript" valodā, var šo priekšdarbu izlaist un jau pēc migrācijas domāt par "TypeScript" valodas izmantošanu projektā[6].

Tomēr pēc darba autora domām migrēšana uz "TypeScript" valodu ir vērtīgs ieguldījums, kas nav pārāk laikietilpīgs un sarežģīts process, jo "TypeScript" valodas pamatā ir tā pati "JavaScript" valodas kopa. Tādēļ nekāda īpaši lielā pirmkoda migrēšana nav jāveic, bet gan pamatā jāveic pirmkoda sintakses izmaiņas. Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnes projekts, jau bija veidots "TypeScript" valodā, tādēļ arī vēlākā migrēšana uz "Angular" ietvaru bija atvieglota, jo projekta pirmkods bija daudz īsāks, vienkāršāks un lasāmāks par tā ekvivalento "JavaScript" valodas kodu, kas ļāva daudz raitāk migrēt atsevišķas klases, jo sintaksiskās kļūdas tika identificētas jau rakstīšanas brīdī, kas nebūtu iespējams izmantojot "JavaScript" valodu. Kā arī projekta atkļūdošana bija atvieglota, jo bija iespējams uzreiz atsekot problēmas cēloņus un kļūdainās vietas.

"AngularJS" projekti, kuri ir veidoti "TypeScript" programmēšanas valodā, darba autors iesaka vērst uzmanību tam kāds definīciju pārvaldnieks tiek izmantos. Vecākiem "AngularJS" projektiem, kuros tiek izmantots "TSD" definīciju pārvaldnieks būs nepieciešams atbrīvoties no tā, jo tas ir novecojis. Tā vietā ir jāizmanto "Typings" definīciju pārvaldnieks, kurš ir daudz ērtāks nekā "TSD". Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnes projektam arī bija jāveic pāreja no "TSD" uz "Typings" definīciju pārvaldnieku. Kā rezultātā bija jāveic papildinājumi lietotnes pirmkodā. Darba autors vēlas vēl pievērst uzmanību tam, kad "Typings" definīciju pārvaldniekā nav atrodamas vecākas bibliotēku definīciju versijas, tādēļ iespējams pastāv risks kad projektos arī vajadzēs atjaunot ārējo bibliotēku versijas, lai būtu iespējams izmantot attiecīgās "TypeScript" valodas definīcijas.

2.5. Pārveidot kontrolierus kā klases nevis kā funkcijas

Šis priekšdarbs ir cieši saistīts, gan ar "TypeScript" valodu, gan ar to, kad "Angular" ietvarā vairs netiek izmantoti kontrolieri, tādēļ darba autors iesaka pārveidot projektu kontrolierus par klasēm. Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnē visi kontrolieri tika veidoti, kā strikti definētas klases, kas atviegloja to migrēšanu uz "Angular" ietvaru.

2.6. Komponenšu direktīvas

"AngularJS" lietotnes elementus veido no direktīvām un kontrolieriem, taču jaunajā "Angular" ietvarā viena no būtiskajām izmaiņām ir atteikšanās no kontrolieriem un uzsvars uz komponentēm[6].

Esošajā "AngularJS" ietvarā ir iespējams imitēt šīs "Angular" ietvara komponentes ar tā sauktajām komponenšu direktīvām. Kurās tiek definētas datu savienojumi, šabloni un kontrolieri. Šo komponenšu direktīvu migrēšana ir daudz vienkāršāka nekā migrēt direktīvas, kuras izmanto zema līmeņa funkcionalitāti kā "ng-controller", "ng-include" un "$scope" jeb tvērumu[6].

Lai komponenšu direktīvas būtu saderīgas ar jauno "Angular" ietvaru tām jāseko striktai atribūtu uzbūves struktūrai, kas ir detalizēti aprakstīta oficiālajā "Angular" migrācijas ieteikumu dokumentācijā:

- restritc: 'E' - norāda ka komponente ir elements
- scope: {} - izolēts tvērums. "Angular" visi tvērumi ir izolēti no pārējiem elementiem.
- bindToController - komponentes izvade un ievadam jābūt saistītam ar kontrolieri nevis ar tvērumu
- controller un controllerAs - komponentei ir savs kontrolieris
- template vai templateUrl - komponentei ir savs šablons

Papildus komponenšu direktīvas var saturēt:

- transclude: true - ja komponenti ir jāiekļauj kaut kur citviet
- require - ja komponentei ir jāsazinās ar vecāku komponentes kontrolieri

Ir jāizvairās no sekojošiem atribūtiem kā:

- compile - netiek atbalstīts "Angular" ietvarā
- priority - netiek atbalstīts "Angular" ietvarā
- terminal - netiek atbalstīts "Angular" ietvarā
- replace: true - "Angular" ietvarā netiek aizvietoti komponentes elementi ar komponentes šabloniem, kā arī "AngularJS" ietvarā šis atribūts ir novecojis

Izmantojot komponenšu direktīvas mēs aizvietojam risinājumu kā redzams *2.6.* attēlā ar komponenšu direktīvu risinājumu, kurš ir redzams *2.7.* attēlā.

```
<!-- index.html-->
<div ng-controller="FooBarController">
    {{text}}
</div>

<!--fooBarController.js-->
demoApp.controller("FooBarController", function($scope) {
    $scope.text = "FooBar"
});
```

2.6. att. **"AngularJS" kontroliera piemērs "FooBarController"**

```
<!--index.html-->
<foo-bar>

<!--fooBarTemplate.html-->
{{text}}

<!--fooBarDirective.js-->
demoApp.directive("fooBar", function() {
    return {
        restrict: "E",
        templateUrl: "fooBarTemplate.html",
        link: function(scope, element, attrs) {
            scope.text = "FooBar";
        }
    };
});
```

2.7. att. **Pārveidotais "AngularJS" kontrolieris "FooBarController"**

Šīs pieejas priekšrocības ir tādas, kad ir iespējams pielāgot komponentes ar "HTML" atribūtiem. Ir vienkāršāk atkārtoti izmantot komponentes dažādās lietotnes vietās, papildus tam netiek zaudēta nekāda kontrolieru funkcionalitāte un tiek uzspiesta labās prakses izmantošana. Komponentēm uzreiz tiek piešķirtas vērtības pēc noklusējuma, tādiem atribūtiem, kā "scope" un "restritc"[6].

Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnes komponentes tika veidotas pēc šiem direktīvas labās prakses ieteikumiem, pateicoties "TypeScript" valodas izmantošanai, kas daudz stingrāk uzspieda augstāk minēto atribūtu izmantošanu, tādēļ darba autoram nebija jāveic papildus direktīvu pārveidošana un uzlabošana. Kā rezultātā migrēšana uz "Angular" ietvaru notika daudz raitāk un vienkāršāk, jo nebija daudz sliktā pirmkoda ko migrēt. Tas apliecināja darba autoram, kad "TypeScript" izmantošana un konsekventa pieturēšanās pie labās prakses projektā ne tikai uzlabo pirmkoda lasāmību, bet gan papildus nodrošina to gatavību nākotnes izmaiņām.

2.7. Atteikšanās no tvērumiem

Tvērums jeb "$scope" ir "AngularJS" objekts, kurš savienojas ar "DOM" elementu, kurā tiek pielietos kontrolieris. Visi tā bērnu elementi var piekļūt, vērot un modificēt tvērumā datus, izņemot, ja tie ir izolēti.

Jaunajā "Angular" ietvarā šie tvērumi vairs neeksistē un nevarēs tikt izmantoti, to vietā būs jāizmanto "this" atslēgas vārds. Kā ir redzams *2.8.* un *2.9.* attēlos, kur kontrolierī nevis tiek padots tvērums jeb "$scope", kurš jau tālāk tiek manipulēts, bet gan kontrolieris būs jau savienots ar attiecīgiem atribūtiem un metodēm ārējā līmenī[10].

```
1   // Piemērs ar tvērumu jeb $scope
2   app. controller("SampleCtrl", function($scope) {
3       $scope.header = "Sample Header";
4   });
5
6   // Alternatīva ar "this" tvēruma vietā
7   app.contoller("SampleCtrl", function() {
8       this.header = "Sample Header"
9   })
```

2.8. att. **"AngularJS" tvēruma alternatīvas piemērs sampleCtrl.js**

```
1   <!--Izmantojot tvērumu HTML elements-->
2   <div ng-contoller="SampleCtrl">
3       <input type="text" ng-model="header" />
4   </div>
5
6   <!--Izmantojot tvēruma alternatīvas "this" HTML elements-->
7   <div ng-controller="SampleCtrl as sample">
8       <input type="text" ng-model="sample.header" />
9   </div>
```

2.9. att. **"AngularJS" tvēruma alternatīvas piemērs index.html**

Darba autora tīmekļa pieteikumu formu "AngularJS" lietotnē ir liela atkarība no tvērumiem. Visas lietotnes komponentes tika veidotas ar tvērumiem un to palīdzību tiek padoti, manipulēti un savienoti dati. Tādēļ tvērumi ir neatņemama lietotnes sastāvdaļa tīmekļu pieteikumu formu veidošanā.

Par laimi ir iespējams pārrakstīt pirmkodu tā, lai gala rezultātā tas darbotos, tāpat kā ar tvērumiem, bet tos neizmantojot. Lai to realizētu ir nepieciešams izpildīt iepriekšējo pirms migrācijas priekšdarbu par "AngularJS" versijas atjaunošanu, jo tvērumiem alternatīvs aizvietotājs paradījās tikai sākot ar "AngularJS 1.4" versiju. Kas arī atvieglos tālāko lietotnes migrēšanu uz "Angular" ietvaru, jo nekas papildus nebūs jāveic sakarā ar tvērumiem.

Attēlos *2.10.* un *2.11.* ir redzams piemērs, kā darba autors pārveidoja vienu tīmekļa pieteikumu formu "AngularJS" lietotnes direktīvu, kas izmantoja tvērumu, uz ekvivalentu

direktīvu, kas no tā izvairās. Tas tiek panākts pārvietojot visus tvēruma jeb "scope" atribūtus un metodes uz "bindToController" objektu. Kā rezultāta padarot visus šos atribūtus un metodes pieejamus iekšā kontrolierī. Kas jau tālāk ļauj tos manipulēt un padod tālāk citām direktīvām ar kontroliera palīdzību. Tādā veidā atvieglojot šīs direktīvas migrēšanu uz "Angular" ietvaru.

```
angular.module('app.shared.core')
 .directive('textInput', function() {
  return {
   restrict: 'E',
   require = "^form";
   scope: {
     header: "@?",
     id: "@",
     isDisabled: "=?",
     isRequired: "=?",
     maxLength: "=?",
     minLength: "=?",
     model: "=",
     onFocus: "&",
     patternErrorText: "@?",
     placeholder: "@?",
     requiredErrorText: "@?",
     subHeader: "@?",
     tip: "@?",
   },
   template: '../app/shared/core/textInput/textInput.html',
   }
  };
 });
```

2.10. att. **"AngularJS" direktīvas - "textInput" piemērs ar "scope" pieeju**

```
angular.module('app.shared.core')
 .directive('textInput', function() {
  return {
   restrict: 'E',
   require = "^form";
   scope: {},
   controller: textInputCtrl,
   controllerAs: vm,
   bindToController: {
     header: "@?",
     id: "@",
     isDisabled: "=?",
     isRequired: "=?",
     maxLength: "=?",
     minLength: "=?",
     model: "=",
     onFocus: "&",
     patternErrorText: "@?",
     placeholder: "@?",
     requiredErrorText: "@?",
     subHeader: "@?",
     tip: "@?",
   }
   template: '../app/shared/core/textInput/textInput.html',
  };

  function textInputCtrl() {};
  }
 });
```

2.11. att. **"AngularJS" direktīvas - "textInput" piemērs ar "bindToController" pieeju**

2.8. Izmantot .service() nevis .factory()

“AngularJS” lietotnes sastāv no vairākām objektiem, kā direktīvas, kontrolieri, servisi, filtriem u.c.. Servisi var tikt implementēti divos veidos kā “.service()” un “.factory()”, dažādi izstrādātāji, dažādos projektos izvēlas izmantot vienu vai otru, kur biežāk tiek izmantots “.factory()”.

Kaut arī abu šo funkciju pamatā tās abas darbojas gandrīz identiski, to vienīgā atšķirība ir kad “.service()” funkcija tiek pasaukta kā konstruktora funkcija. Tas ir nozīmīgi migrēšanai uz “Angular” ietvaru, īpaši ja projektā tiek plānots nākotnē izmantot “ES6” skriptu specifikācijas standartu, jo “.factory()” funkciju nav iespējams izmantot pie šīs specifikācijas, jo tā tiek izsaukta kā vienkārša funkcija[19].

Pārveidot “.factory()” objektus uz “.service()” ir pavisam nesāpīgi, īpaši ja tie jau iepriekš tika veidoti kā klases “TypeScript” programmēšanas valodā. Ir nepieciešamas minimālas sintaksiskas izmaiņas projekta pirmkodā, kas neietekmē nekādu funkcionalitāti un tām nepastāv nekāds kļūdu rašanās risks.

Darba autora tīmekļa pieteikumu formu “AngularJS” lietotnē pamatā tika izmantotas “.factory()” funkcijas veidotas klases “TypeScript” valodā, kuras bija vienkārši pārveidot uz “.service()” funkcijas klasēm, pēc kā arī tās tika migrētas uz “Angular” ietvaru iegūstot nākotnes drošu kodu, priekš “ES6” specifikācijas standarta implementēšanas.

2.9. Komponenšu maršrutētājs

“AngularJS” lietotņu projektiem, kuri izmanto klienta puses maršrutēšanu ir jāņem vērā, kad jaunajā “Angular” ietvarā tiek izmantots jauni izveidotais komponenšu maršrutētājs, kurš slēpj sevī dažādus veiktspējas uzlabojumus. Pozitīvi ir tas, kad sākot ar “AngularJS 1.4” versiju šis maršrutētājs ir iekļauts arī vecajā ietvarā, kas tika darīts ar domu, lai atvieglotu migrēšanu uz “Angular” ietvaru. Tādēļ projektiem, kas izmanto vecākas versijas par ‘AngularJS 1.4” ir ieteicams izpildīt versijas atjauninājuma migrācijas priekšdarbu[10].

Darba autora tīmekļa pieteikumu formu “AngularJS” lietotne neizmanot klienta puses maršrutēšanu, bet gan “ASP.NET” projekta servera puses maršrutēšanu, tādēļ šis un arī citi projekti kas neizmanto “AngularJS” klienta puses maršrutēšanu var izlaist šo migrācijas priekšdarbu.

2.10. Izvairīties no “jqLite”

“AngularJS” ietvars iekļāva sevī iebūvētu “jqLite” bibliotēku, kas nodrošina lietotnes “DOM” manipulācijas. “jqLite” ir “jQuery” bibliotēkas maza apakškopa, kurā ir iekļautas tikai visbiežāk izmantotās funkcijas. Tā kā “Angular” ietvarā vairs “jqLite” nav iekļauts, ir jārēķinās, ka veiksmīgai migrēšanai, funkcijas, kuras lieto šo bibliotēku ir jāpārraksta vai arī projektā ir jāiekļauj pilna “jQuery” bibliotēka[16].

Izstrādātājiem pašiem ir jānovērtē vai konkrētās lietotnes projektā ir iespējams attiekties no “jqLite” bibliotēkas izmantošanas, aizvietojot to ar jauni izveidotām palīgfunkcijām.

Kopumā izstrādātāji iesaka izvairīties no “jQueary” bibliotēkas un to spraudņu izmantošanas projektā, ja tas ir iespējams, jo visas to piedāvātas funkcionalitātes ir iespējams implementēt pašā ietvarā. Kā arī ne visiem “jQuery” spraudņiem ir izveidotas attiecīgas “TypeScript” valodas definīcijas, pieņemot, ka projekts tiek izstrādāts “TypeScript” valodā. Bieži vien izstrādātāji arī nepareizi iekapsulē šos spraudņus, kas rezultē kļūdās un konfliktos manipulējot “DOM” vienlaicīgi ar pašu “Angular” ietvaru[20].

Taču tehniski pašā “Angular” ietvarā nav nekādu šķēršļu, lai izmantotu “jQuery” bibliotēku un uz tās veidotos spraudņus. Arī darba autora tīmekļa pieteikumu formu lietotnē tiek izmantots datuma izvēlnes “jQuery” spraudnis, kurš arī pēc migrācijas uz “Angular” ietvaru darbojās korekti un šādu spraudņu izmantošana ietaupa lielu izstrādes laiku. Kaut arī “Angular” ietvars eksistē jau labu laiku, tajā vēl nav izveidoti visi mūsdienu tīmekļu lapas izstrādei nepieciešamie spraudņi un utilītprogrammas, lai nebūtu nepieciešams izmantot “jQuery” bibliotēku un uz tās veidotos spraudņus un logrīkus.

2.11. Dekoratori

“AngularJS” lietotnēm, kuras tika veidotas uz “AngularJS 1.3” un “AngularJS 1.4” versijas ietvara pastāv iespēja izmantot dekoratora bibliotēkas, kuru mērķis ir ļaut izstrādātājiem rakstīt “Angular” ietvara sintaksē paliekot tajā paša “AngularJS” ietvar. Tādā veidā ļaujot izstrādātajiem palikt sev ierastajā ietvarā, ko tie labi pārzin. Tajā pašā laikā paralēli pierodot un apgūstot jaunā ietvara sintaksi, neriskējot izveidot kļūdas un bojājumus jau esošā lietotnes risinājumā.

Šādas dekoratoru bibliotēkas ir vairākas kā “angular2-now”, “angular-decorators”, “a1script”, “ng-classy” un “ngForward”. Šīs dekoratoru bibliotēkas, tika galvenokārt izveidotas laikā, kad “Angular” ietvars vēl atradās alfa un beta izstrādes stadijās. Tie tika veidoti ar domu, lai motivētu “AngularJS” izstrādātājus izmēģināt un izvērtēt jaunā ietvara sintaksi un loģiku[5].

Taču darba autors neiesaka izmantot šīs dekoratoru bibliotēkas reālos lietotnes projektos. Šo dekoratoru trūkums ir tas, ka tie tikai imitē "Angular" sintaksi un beigās migrējot "AngularJS" lietotni uz "Angular" ietvaru tāpat būs jāpapildina pirmkods, tādā veidā veicot pirmkoda parakstīšanu divreiz. Nākamais šo dekoratoru trūkums ir kad neviens no šiem dekoratoriem nav pilnīgs, kas palielina risku radīt liekas problēmas un nesaprašanās tos izmantojot un vēlāk migrējot uz "Angular" ietvaru. Papildus tam lielākā daļa šo dekoratoru vairs netiek uzturēti un atjaunoti atbilstoši jaunajām "Angular" un "AngularJS" versijas izmaiņām.

Darba autors izmēģināja izmantot šos dekorators dažās savās tīmekļa pieteikumu formu "AngularJS" lietotnes komponentēs, un neatrada pietiekami daudz pozitīvu iemeslu par labu šo dekoratoru izmantošanā. Mazos "AngularJS" lietotņu projektos, kā pieteikumu formās vai vienkāršās tīmekļa lapās dekoratoru izmantošana ir lieka, jo daudz ātrāk un vienkāršāk ir migrēt visu lietotni uz jauno ietvaru, jo sintaktiski pirmkoda pārrakstīšana ir ļoti līdzīga. Savukārt lielos projektos daudz drošāk ir inkrementāli migrēt uz jauno ietvaru izmantojot pašu "Angular" izstrādātāju veidoto starp bibliotēku "ngUpgrade", kas ir tieši paredzēta abu ietvaru komponenšu starp lietošanā. Kā arī šī bibliotēka ir pastāvīgi atjaunota un uzturēta, kurā visas tālākās "AngularJS" un "Angular" ietvara izmaiņas tiek ņemtas vērā. Tādēļ arī stabilitātes ziņā tā ir labāka alternatīva par dekoratoriem.

Šo dekoratoru pamata uzdevums bija tikt izmantotiem, kad vēl valdīja neskaidrība un izstrādātājiem nebija pārliecības vai tev vēlas migrēt uz jauno ietvaru, tādēļ šie dekoratori piedāvāja sava veida kompromisu. Daudzi izstrādātāji implementēja šos dekoratorus savos projektos, iedzenot sevi nepatīkamā situācijā kopš tos beidza uzturēt, jo to mērķis nebija aizvietot "Angular" ietvaru, bet gan kalpot kā īslaicīgam migrēšanas tiltam un jaunā ietvara mācību avotam[5].

3. MIGRĀCIJAS STRATĒĢIJAS

Veicot dažādus migrācijas priekšdarbus, tādā veidā uzlabojot "AngularJS" lietotnes projektu, nākamais solis pirms sākt migrēt ir saprast, kura migrācijas stratēģija ir vispiemērotākā.

Izvēle pastāv starp divām projektu migrācijas stratēģijām – lielā sprādziena migrācijas stratēģija un inkrementālā migrācijas stratēģija. Abām šīm stratēģijām ir savas priekšrocības, tādēļ ir nepieciešams rūpīgi izvērtēt, kura ir piemērotākā stratēģija attiecīgiem projektiem pamatojoties uz dažādiem faktoriem, kā projekta lielums, pieejamais laiks migrācijai, ārējo atkarību daudzums u.c. [6][7][11][12].

Darba autors, savu tīmekļa pieteikumu formu "AngularJS" lietotni migrēja izmantojot lielā sprādziena migrācijas stratēģiju. Šī izvēlē tiks pamatota zemāk redzamās nodaļas apakšnodaļās, kur tiks apskatīta katra stratēģija, kā arī tās priekšrocības un trūkumi, konkrētajā darba autora tīmekļu pieteikumu formas projekta gadījumā.

3.1. Lielā sprādziena migrācijas stratēģija

Lielā sprādziena migrācijas stratēģijas būtība ir migrēt visu uzriezi jeb vienā lielā sprādzienā. Pilnībā nomigrējot visu projekta pirmkodu, atjaunojot visu bibliotēku versijas un atmetot veco "AngularJS" ietvara bibliotēku[8].

Izmantojot šo stratēģiju jāņem vērā, ka ir nepieciešams apturēt lietotnes izstrādi, kamēr tā visa netiks nomigrēta uz jauno ietvaru. Rekomendācija apturēt lietotnes izstrādi uz migrācijas laiku ir pamato ar to, lai netiktu implementētas jaunas funkcionalitātes vai pārveidotas jau esošās, lai migrācija nebūtu jāveic divreiz un nerastos papildus problēmas un šķēršļi. Kā arī gadījumos, lai netiek veidotas jaunas funkcionalitātes, kuras ir atkarīgas no kādām funkcijām un klasēm kuras jaunajā ietvarā tiktu veidotas no jauna[8].

Taču darba autors uzskata, kad gala lēmums vai apturēt lietotnes izstrādi uz migrācijas laiku, tomēr ir jāpēta atsevišķi katram projektam, jo šī prasība nav obligāta. Darba autors uzskata, kad nelielas izmaiņas un papildinājumi var tikt pievienoti migrācijas beigās, kas parasti ir raksturīgi mazām lietotnēm. Tādēļ projekta lielums ir ļoti svarīgs šīs stratēģijas izmantošanai, jo ir nepieciešams novērtēt cik ilgs laiks ir vajadzīgs pilnīgai migrācijai, tāpēc arī šo stratēģiju ir rekomendēts izmantot salīdzinošiem maziem lietotnes projektiem, lai to migrācija neievilktos uz nenoteiktu laiku, jo šīs stratēģija galvenā priekšrocība ir pēc iespējas ātrāka migrācija uz jauno ietvaru un atbrīvošanās no vecā ietvara.

Papildus projekta lielumam arī projekta stadija spēlē lielu lomu, jo ne vienmēr ir iespēja pilnībā apturēt tālāko lietotnes izstrādi, lai veltītu laiku migrācijai. Piemēram, ja lietotnes tuvojas izlaišanas termiņam vai arī ir steidzami nepieciešams ieviest jaunu funkcionālīti vai izmaiņas, ja lietotne jau ir produkcijā. Tāpēc ir nepieciešams koordinēt ar biznesu, gan lēmums par migrāciju, gan stratēģiju, kurā tā tiks realizēta un skaidri jāieplāno termiņi no kuriem līdz kuriem, nebūs iespējams biznesam pieprasīt lietotnes izmaiņas. Tādēļ šī stratēģija ir arī iesakāma projektiem, kuri ir ne tikai mazi, bet arī atrodas vēl izstrādes stadijā, kur šādas izmaiņas nav tik riskantas un iespējams pat bija ieplānotas lietotnes izstrādes termiņos.

Bez lietotnes projekta lieluma un izstrādes stadijas posma ir svarīgi ņemt vērā atkarību no ārējām bibliotēkām, kuras tiek izmantotas projektā un pārliecināties vai tām eksistē atjauninājumi, kuri ir saderīgi ar jauno ietvaru. Pretējā gadījumā būs nepieciešams meklēt alternatīvas ārējās bibliotēkas, kuras atbalsta jauno ietvaru vai arī veidot savu risinājumu konkrētai funkcionalitātei. Tādēļ projektiem kuri ir ļoti atkarīgi no šādām bibliotēkām ir kārtīgi jāizpēta visi potenciālie riski. Pozitīvi ir tas, kad jaunais "Angular" ietvars tika palaist pietiekamu ilgu laiku atpakaļ, tādēļ daudzām bibliotēkām ir bijis laiks migrēt un tikt atjaunotām, lai tās būtu saderīgas ar jauno ietvaru. Tāpēc ir iespējams, kad atsevišķos gadījumos var nerasties nekādu problēmu.

Šīs stratēģijas priekšrocības, ir tādas, kad pēc iespējas ātrāk var atbrīvoties no "AngularJS" ietvara bibliotēkas un sākt izmantot jauno "Angular" ietvaru ar tā priekšrocībām un funkcionalitāti.

Darba autors savu tīmekļa pieteikumu formu "AngularJS" lietotni izvēlējās migrēt tieši ar lielā sprādziena stratēģiju. Darba autors balstīja savu izvēli uz augstāk minētiem faktoriem. Projekta lielums migrēšanas brīdi bija neliels, jo to sastādīja trīs pieteikumu formas ar dažām vienkāršām komponentēm un servisiem. Klāt nelielajam projekta lielumam arī tika veikti vairāki migrācijas priekšdarbi, kas papildus atviegloja komponenšu migrēšanu. Tīmekļa pieteikuma formu lietotne jau atradās produkcijā un paradījās laiks projekta tehniskajiem uzlabojumiem, kamēr bizness sagatavo prasības nākamajām pieteikumu formām ko pievienot lietotnē, tāpēc bija iespēja veikt projekta migrāciju bez izstrādes apturēšanas. Projektam bija minimāla atkarība uz ārējām bibliotēkām, kurām visām bija pieejami versiju atjauninājumi, kas atbalstīja jauno "Angular" ietvaru. Papildus šiem faktoriem bija vēlme no izstrādātāju puses migrēt visu uzriezi, lai pēc iespējas ātrāk visu nomigrētu un pilnībā atbrīvotos no "AngularJS" ietvara, kur pretējā gadījumā izmantojot inkrementālo migrācijas stratēģiju samazinātos pirmkoda lasāmība un pastāvētu risks par migrācijas procesa ievilkšanos, jo tās pilnīgā migrācija nebūtu tik kritiska, kā lielā sprādziena migrācijas gadījumā.

3.2. Inkrementālā migrācijas stratēģija

Inkrementālās migrācijas stratēģijas būtība ir pakāpeniski migrēt pirmkoda komponentes uz jauno "Angular" ietvaru iekš jau esošajā "AngularJS" lietotnes kodā. Tas tiek panākts izmantojot abus ietvarus paralēli un nodrošinot komunikāciju starp tiem ļaujot abu ietvaru komponentēm un servisiem mijiedarboties. Kā rezultātā, gan "AngularJS" ietvara komponentes var tikt izmantotas iekš "Angular" komponentēs un otrādi, "Angular" komponentēs var tik izmantotas iekš "AngularJS" komponentēs[8].

Šīs stratēģija priekšrocības ir ļaut izstrādātāju komandai būt elastīgai lietotnes migrācijā. Ļaujot izstrādātājiem pakāpeniski un tiem ērtā veidā migrēt lietotnes pirmkodu uz jauno ietvaru, neapturot lietotnes izstrādi un nepārrakstot visu pirmkodu uzriezi.

Piemēram, pielietojot šo stratēģiju izstrādātāju komanda var atstāt visu iepriekšējo "AngularJS" ietvara pirmkodu un veidot tikai jaunās komponentes izmantojot jauno "Angular" ietvaru. Vai arī, izstrādēs laikā pamazām migrēt vienkāršākos pirmkoda gabalus uz jauno ietvaru atstājot sarežģītākos servisus un komponentes uz vēlāku laiku.

Papildus, izmantojot šo stratēģiju ir iespējams izmantot visas jaunās "Angular" ietvara funkcionalitātes un priekšrocības it kā lietotne būtu veidota tikai ar "Angular" ietvaru tādā veidā neko nezaudējot[8].

Kopumā šīs stratēģijas elastīgums, atver izstrādātājiem daudz iespējas, kā organizēt un plānot lietotnes migrāciju uz jauno ietvaru, samazinot atkarību no projekta biznesa prasībām un dodot iespēju neaiztikt bīstamos pirmkoda gabalus, kuru migrēšanai pastāv paaugstināts risks.

Inkrementālo stratēģiju ir vērts izmantot lieliem projektiem, kur nav pietiekami daudz resursu, lai būtu iespējams nomigrēt visu projektu uz jauno ietvaru. Kā arī projektiem, kuru izstrādi nevar apturēt priekš lielā sprādziena migrācijas, tādā veidā ļaujot izstrādātājiem paralēli, gan veidot jaunu funkcionalitāti, gan migrēt pakāpeniski projekta kodu uz jauno ietvaru. Ja lietotnes projektam ir daudzas kritiskas atkarības no daudzām ārējām bibliotēkām, kuras nav iespējams vai ir riskanti migrēt uz jauno "Angular" ietvaru, tad arī šajā gadījumā pakāpeniskā stratēģija ir labs risinājums, jo tā ļauj migrēt saglabājot visas vecās arējās bibliotēkas tik ilgi, kamēr tām parādīsies attiecīgie jauninājumi, kas ļaus tās izmantot jaunajā ietvarā[8].

Šīs stratēģijas galvenie trūkumi ir abu ietvara bibliotēku izmantošana, tādēļ arī lietotnes ielāde pārlūkprogrammā būs ilgāka, jo būs nepieciešams lejupielādēt un parsēt, gan veco "AngularJS", gan jauno "Angular" ietvaru. Tādā veidā samazinot lietotnes ātrdarbību un veiktspēju, kas noteiktām lietotnēm varbūt ļoti kritisks faktors[8].

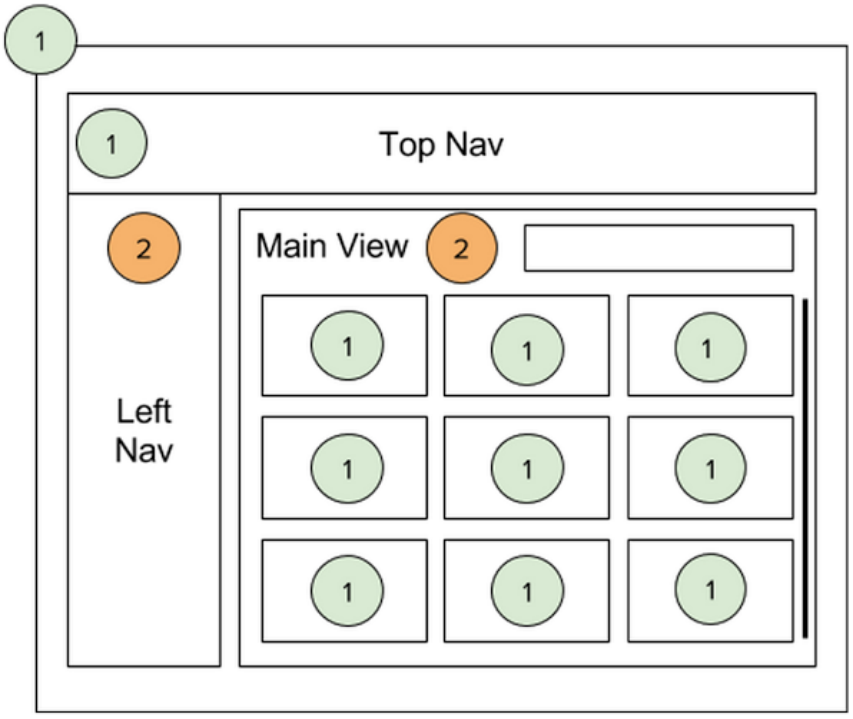
Nākamais šīs stratēģijas trūkums ir pirmkoda lasāmība un uzturēšana, izmantojot abus šos ietvarus vienā projektā radīsies nedabisks pirmkods. Abi šie ietvari kaut arī nedaudz līdzīgi, tomēr atšķiras pēc to filozofijas un loģikas, tādēļ arī tie ir divi dažādi ietvari. Šāda pirmkoda himera radīs daudz grūtāk uzturamu kodu, kur lietotnes izstrādātājiem būs jāpārzina abu ietvaru nianses, kā arī to mijiedarbības specifiku. Kas mūsdienu darba tirgū, var sagādāt grūtības piesaistot jaunus izstrādātājus projektiem, paildzinot laiku kas tiem ir nepieciešams lietotnes projekta pirmkoda apguvei. Papildus tam pastāv risks, ka galīga lietotnes migrācija tā arī nekad netiks veikta atstājot lietotnes pirmkoda migrāciju pusratā, tādēļ ir nepieciešams saplānot skaidru migrācijas plānu ar termiņiem, lai lietotnes migrācija tiktu novesta līdz galam[8].

Darba autors neizvēlējās šo migrācijas stratēģiju izmantot savas tīmekļa pieteikumu formu "AngularJS" lietotnes migrācijā, balstoties uz sekojošiem faktoriem. Pirmkārt lietotnes projekts nebija pietiekami liels, lai nebūtu iespējams to nomigrēt visu uzreiz ar lielā sprādziena stratēģiju. Otrkārt nepastāvēja šķērslis ārējo bibliotēku migrēšanā uz jauno "Angular" ietvaru. Treškārt izstrādātāju komanda izvērtējot inkrementālās stratēģijas riskus, nevēlējās apgrūtināt pirmkoda lasāmību un uzturēšanu izmantojot abus ietvarus paralēli. Sakarā ar to, kad ne visi komandas izstrādātāji pārzin abus šos ietvarus pietiekami augstā līmenī, kas prasītu papildus laiku un atbalstu no citiem izstrādātājiem abu ietvaru un to savstarpējās mijiedarbības apguvei. Ceturtkārt takā šīs lietotnes tīmekļa pieteikumu formas jau bija produkcijā, pastāvēja risks, ka izmantojot abus ietvarus palielināsies lietotnes lejupielādes laiks salīdzinot ar jau esošo, kas ir īpaši svarīgi mobilo ierīču lietotājiem.

Protams, lai darba autors spētu novērtēt inkrementālās migrēšanas stratēģijas pielietošanas sarežģītību un tās potenciālo izmantošanu savas tīmekļa pieteikumu formu "AngularJS" lietotnes migrēšanā bija nepieciešams izpētīt tās pielietojumu ar piemēriem. Ko vēlāk noprezentēt pārējiem izstrādātājiem, lai spētu veikt migrācijas stratēģijas izvēlēs veikšanu.

3.3. Inkrementālā migrācijas stratēģijas implementēšana

Inkrementālās migrēšanas stratēģiju padara iespējamu bibliotēka "ng-upgrade", kura ir "Angular" izstrādātāju veidota. Iekļaujot šo bibliotēku esošajā "AngularJS" projekta kodā kopā ar "Angular" ietvara bibliotēku ir iespējams izmantot abus šos ietvarus vienlaicīgi. Bibliotēka "ng-upgrade" kalpo kā tilts starp abiem šiem ietvariem radot ilūziju it kā tiktu izmantots tikai viens ietvars, gan lietotnes darbībā, gan projekta pirmkoda veidošanā. Kā uzskatāms piemērs kalpo *3.1.* attēls, kur ir parādīts piemērs kā eksistējošā "AngularJS" lietotnē ir iespējams izmantot jaunā "Angular" ietvara komponentes un izmantot tās iekš vienu otrā. Attēlā ar cipariem viens tiek apzīmētas "AngularJS" ietvarā veidotas komponentes un ar ciparu divi tiek apzīmētas komponentes kas tika veidotas ar jauno "Angular" ietvaru. Attēlā ir redzams, kā lietotne sastāv no trim galvenajām komponentēm, kā augšējais navigācijas panelis – "Top Nav", kreisais navigācijas panelis "Left Nav" un galvenais skats – "Main View". No kuriem kreisais un galvenais skats ir jaunās "Angular" komponents. Papildus tam galvenā skata komponente izmanto iekš sevī citas vecās "AngularJS" komponentes. Tāpat līdzīgā veidā ir arī iespējams izmantot, ne tikai komponentes, bet arī servisus un filtrus savā starpā, kā rezultāta nav nekādu ierobežojumu, pirmkoda migrācijas kārtībai un secībai kādā tas tiks darīts[11].



3.1. att. **"AngularJS" lietotnes diagramma, kurā tiek pielietota inkrementālā migrēšanas stratēģija[11]**

Kaut arī komponenšu un servisu migrāciju kārtībām nav nekādu ierobežojumu, lieliem projektiem, pie kuriem iespējams strādā vairākas izstrādātāju komandas būtu ieteicams strukturēt lietotnes pakāpenisko migrāciju, lai tā nenotiktu haotiski un nerastos nesaprašana komandu un izstrādātāju starpā.

Viena no šādām inkrementālās migrēšanas stratēģijas struktūrām ir vertikālā griezuma pieeja, kur migrācija notiek tādā kartībā, kad tiek migrēts katrs lietotnes skats, tādā veidā viss, kas tiek paradīts darbojas vai nu vienā vai otrā ietvarā. Šīs pieejas trūkums ir, kad lietotnēm kurām ir daudz kopīgo komponenšu vai servisu, būs nepieciešams veidot un uzturēt daudz dublikātus jeb abu ietvaru komponenšu versijas. Kā arī ļoti dinamiskām lietotnēm, kur netiek lietota vairāku skatu pieeja, bet gan tiek lietota vienas lapa pieeja vertikālā griezuma migrācijas pieeja nav derīga[22].

Tomēr lietotnēm, kurām tiek izmantota vairāku skatu pieeja vertikālā griezuma migrācijas priekšrocības ir kad to ir vieglāk saprast un atkļūdot, jo netiek jaukti haotiski dažādu ietvaru pirmkodi. Un ir iespējams ērti sadalīt darbu un atbildību starp izstrādātajiem, katram piešķirot savu skatu tādā veidā samazinot atkarību un izmaiņu konfliktus to starpā. Papildus, tā kā katrs skats izmanto tikai vienu ietvaru, šie skati ielādējas un darbojas daudz ātrāk optimizējot lietotnes veiktspēju[22].

Otra iespējamā inkrementālās migrēšanas stratēģijas struktūra ir horizontāla griezuma pieeja. Tās sākumā tiek migrētas lietotnes zemākā līmeņa kopīgās komponentes, piemēram kā dažādu ievada lauku komponentes. Pēc kā pāriet uz dažādām virs komponentēm līdz visa lietotne ir nomigrēta. Horizontālā griezuma pieejas priekšrocība ir, kad šāda pieeja ir daudz brīvāka par vertikālo pieeju, jo nav obligāti jānomigrē viss skats, lai lietotne darbotos[22].

Pilnīgu abu ietvaru paralēlu darbību, “ng-upgrade” bibliotēka nodrošina atrisinot savstarpējo mijiedarbību starp trim galvenajiem ietvara moduļiem, kā atkarību inžekcija, komponenšu ligzdošana un savstarpējā iekļaušana, un izmaiņu fiksēšana[8][14][15].

Ietvara atkarību inžekcijas modulis nodrošina funkciju un klašu izsaukšanu dažādās lietotnes vietās. “AngularJS” ietvarā atkarību inžekcija modulis tiek realizēts kā vienotās saknes inžekcija, bet “Angular” ietvarā tiek izmantota hierarhiskā inžekcija. To lai abu ietvaru inžekcijas instances būtu pieejamas no jebkuras ietvara komponentes, tiek nodrošināts automātiski ar “ng-upgrade” bibliotēku. Tādā veidā ļaujot “Angular” ietvara komponentēm izsaukt visas “AngularJS” servisus un funkcijas. Savukārt, lai “AngularJS” direktīvas un servisi spētu izsaukt visus “Angular” ietvara servisus un komponentes, papildus ir nepieciešama kartēšanas konfigurācija. Kā rezultāta ir iespējams migrēt lietotnes servisus pa vienam izmantojot tos dažādos ietvaros[14][15].

Abos ietvaros lietotnes komponente sastāv no direktīvas un šablona ko tā izmanto. Lai būtu iespējams iekļaut komponentes vienu otrā neatkarīgi no ietvara kurā tā tika veidota, "ng-upgrade" bibliotēka automātiski apliek šīs komponentes ar nepieciešamajiem dekoratoriem. Tādā veidā nav nepieciešams rakstīt komponenti nekādā īpašā sintaksē, jo bibliotēka apliks komponenti ar dekoratoriem automātiski. Papildus tam katra komponente izpildīsies attiecīgajā ietvarā, kurā tā tika veidota. Rezultātā izmantojot sava ietvara priekšrocības un funkcionalitāti neskatoties uz to kad virs komponente vai pati lietotne tiek izpildīta uz vecā "AngularJS" ietvara[14][15] .

Tā kā abiem ietvariem ir atšķirīgi izmaiņu fiksācijas moduļi, kur "AngularJS" seko līdzi izmaiņām "scope" objektā un jaunajā "Angular" ietvarā šī objekta vairs nav. Tad "ng-upgrade" bibliotēkā apvieno abu ietvaru izmaiņu fiksācijas vienā kopīgā ciklā, kā rezultāti abu ietvaru izmaiņas fiksācija moduļi darbojas starplikām[14][15].

Vēl viens modulis, kuru iespējami ir vēlams pārmigrēt ir lietotnes maršrutēšana, bet tā nav kritiska, jo pati par sevi var darboties ar jebkādu jau pieejamo maršrutēšanu, bez "ng-upgrade" bibliotēkas starpniecību, tādēļ var tikt nomainīta tikai pašā pēdējā brīdī, kad visa lietotnes migrācija tiks pabeigta[11].

Padarot šos trīs galvenos abu ietvaru moduļus savietojamus ir iespējams izmantot inkrementālo migrācijas stratēģiju lietotnes migrācijā uz jauno "Angular" ietvaru, soli pa solim pārnesot katru komponenti uz jauno ietvaru.

Praksē inkrementālā migrācijas stratēģija tiktu implementēta sekojoši. Sākumā eksistējošajam "AngularJS" projektam pievieno "Angular" ietvaru kopā ar "ng-upgrade" bibliotēku. Pēc kā var izvēlēties pirmo komponenti ko migrēt uz jauno "Angular" ietvaru, pārveidojot direktīvas šablonu uz "Angular" komponentes sintaksi, kā arī konvertēt šīs direktīvas kontrolieri un savienojuma funkciju uz "Angular" sintaksi un semantiku. Tālāk eksportēt šo jauni izveidoto komponenti ar "ng-upgrade" bibliotēku kā "AngularJS" direktīvu, lai to varētu izmantot vecajā lietotnes kodā. Pēc vēlamo direktīvu migrēšanas var izvelēties servisu, kuru nomigrēt uz jauno ietvaru. Šīs migrācijas pirmkoda izmaiņām parasti ir jābūt ļoti minimālām īpaši, ja serviss tika veidots izmantojot kā klase "TypeScript" programmēšanas valodā. Pēc kā eksportē šo servisu uz vietām vecajā ietvara pirmkodā kur tas tiek izmantots ar "ng-upgrade" bibliotēku. Tālāk atkārtot direktīvas un servisu migrācijas soļus sev vēlamajā secībā, kamēr vien lietotnē pastāv pirmkods ko migrēt. Beigās kad tiek nomigrētas visas "AngularJS" direktīvas un servisi aizvieto "AngularJS" sākum palaidēju ar jauno "Angular" ietvara sākum palaidēju, tādā veidā pabeidzot lietotnes migrāciju[8].

Ir jāņem vērā kad inkrementālā migrācijas stratēģija neatbalsta ne komponenšu direktīvas, kuras jaunajā "Angular" ietvarā skaitās kā slikts stils. Kā arī kopumā nevajadzētu būt gadījumiem pēc to nepieciešamības lietotnes projektā[11].

Tālāk darba autors izklāstīs kā šis implementācijas apraksts tika pielietots pa solim vienkārša piemēra izveidošanā, lai gūtu priekštatu par inkrementālās migrācijas stratēģijas pielietojumu praksē.

Inkrementāli migrējot "AngularJS" lietotnes projektu ar "ng-upgrade" bibliotēku, tā saknes līmenī visu laiku tiks lietots "AngularJS" ietvara sākum palaidēj modulis. Tas nozīmē, ka migrēšanas procesā jaunā "Angular" ietvara komponentes vienmēr tiek palaista no vecā "AngularJS" ietvara.

Darba autors kā piemēru izveido izdomātas lietotnes "sampleApp" moduli, kura ir redzama *3.2.* attēlā. Šādu moduli parastā "AngularJS" ietvara lietotnē sākumielādē izmantojot "ng-app" atribūtu novietojot to galvenajā "HTML" šablonā, bet inkrementālās migrācijas gadījumā ar "ng-upgrade" bibliotēku ir jāpielieto "ngUpgrade" modulis.

```
1   var app = angular.module("sampleApp", []);
```

3.2. att. **"AngularJS" lietotnes moduļa piemērs**

Kā redzams *3.3.* attēlā, lai pielietotu "ngUpgrade" moduli sākuma tas ir jāieimportē no ārējās bibliotēkas un jāizveido tam jauna instance. Pēc kā tam tiek izsaukta "bootstrap" funkcija, padodot "sampleApp" kā atkarības moduli.

```
1   import { UpgradeAdapter } from '@angular/upgrade'
2
3   var adapter = new UpgradeAdapter();
4   var app = angular.module("sampleApp", []);
5
6   adapter.bootstrap(document.body, ["sampleApp"])
```

3.3. att. **"AngularJS" lietotnes "ngUpgrade" moduļa izmantošanas piemērs**

Papildus šim solim ir svarīgi noņemt "ng-app" atribūtu no "HTML" šablona. Pēc kā arī lietotne ir veiksmīgi sasāknēta ar "ngUpgrade" moduli un ir iespējams sākt izmantot abu ietvaru komponentes "sampleApp" lietotnes projektā. Vienīga piepilda ir par "UpgradeAdapater" objekta instanci izveidošanu ir kad atšķirībā no piemēra, reālajā lietotnes projektā ir vēlams to atdalīt atsevišķā modulī un tālāk jau importēt šo instanci vietās kur tas ir nepieciešams, tādā veidā izvairoties no vairāku instanču veidošanas un padarot kodu daudz lasāmāku un vieglāk uzturamu inkrementālās migrācijas laikā.

Tālāk darba autors parādīs piemēru, kā jaunās “Angular” komponentes var izmantot un izsaukt iekš vecajās “AngularJS” komponentēs. Kā piemērs tiek izmantota vienkārša “AngularJS” komponente - “simplePerson”, kuras funkcija ir attēlot padotās personas vārdu, uzvārdu un vecumu, kā redzams *3.4.* attēlā.

```
app.component("simplePerson", () => {
    bindings: {
        person: "="
    },
    controller: "SimplePersonController",
    template:
        `<p>{{$ctrl.simplePerson.name}}</p>
        <p>{{$ctrl.simplePerson.surname}}</p>
        <p>{{$ctrl.simplePerson.age}}</p>
        `
});
```

3.4. att. **“AngularJS” komponentes piemērs**

Pēc *3.4.* attēlā redzamās “AngularJS” komponentes migrācijas uz “Angular” ietvaru šai komponentei būtu jāizskatās kā *3.5.* attēlā redzamā komponente. Vienīgā piepilda, ka attiecīgi labajam stilam, objekta klasi “SimplePerson” būtu jāatdala atsevišķā failā, gadījumiem, ja to atkārtoti izmanto arī citās komponentēs.

```
@Component({
    selector: "simple-person",
    template:
        `<p>{{simplePerson.name}}</p>
        <p>{{simplePerson.surname}}</p>
        <p>{{simplePerson.age}}</p>
        `
})

class SimplePerson {
    @Input() simplePerson: SimplePerson;
}
```

3.5. att. **“Angular” piemēra komponente**

Pēc komponentes izveidošanas viss kas atliek, lai to spētu izmantot vecās “AngularJS” ietvara komponentes ir jāpievieno pirmkoda rinda, kura ir redzama *3.6.* attēlā. Viss ko šī pirmkoda rinda dara ir izmanto “UpgradeAdapater” funkciju “downgradeComponent”, kurai padot “Angular” ietvara komponenti un tai tiek izveidota ekvivalenta “AngularJS” direktīva, kuru jau šis ietvars var izmantot.

```
app.directive("simplePerson",
    adapter.downgradeComponent(SimplePerson));
```

3.6. att. **“AngularJS” lietotnes “downgradeComponent” funkcijas piemērs**

Kaut arī šādā veidā jaunā komponente tiek pārveidota par "AngularJS" direktīvu un tiek izsaukta ar šo pašu ietvaru, tomēr šīs komponentes skats tiek kontrolēts ar jauno "Angular" ietvaru tādā veidā izmantojot šī jaunā ietvara funkcionalitāti un veiktspēju.

Nākamajā piemērā darba autors aprakstīs kā ir iespējams izmantot "AngularJS" ietvara direktīvas iekš nomigrētajās "Angular" ietvara komponentēs. Attēlā *3.7.* ir redzama "AngularJS" direktīva "simplePerson". Šai direktīvai obligāti jābūt veidotai pēc nodaļā *2.4.* aprakstītās struktūras, pretēji "ng-upgrade" bibliotēka nespēs pārveidot direktīvu tā, lai to varētu izmantot jaunās "Angular" ietvara komponentes.

```
export function simplePerson() {
    return {
        restrict: "E",
        scope: {},
        bindToController: {},
        controller: simplePersonController,
        controllerAs: "simplePerson",
        template: ` <p>{{simplePerson.name}}</p>
                    <p>{{simplePerson.surname}}</p>
                    <p>{{simplePerson.age}}</p>
                    `
    };
}
```

3.7. att. **"AngularJS" lietotnes komponentes direktīvas piemērs**

Atšķirībā no "Angular" ietvara komponentes, kur pietika pievienot vienu pirmkoda rindu, lai to varētu izmantot visās "AngularJS" direktīvās, "AngularJS" direktīvai ir jāveic mazliet vairāk darba. Kā redzams *3.8.* attēlā no sākuma ir jāieimportē "Angular" ietvara un "ng-upgrade" bibliotēkas moduļi, pēc kā jāizveido jauna "Angular" ietvara direktīva, kas ir "UpgradeComponent" moduļa paplašinājums. Šai jauni izveidotajai direktīvas konstruktorā ir jāveic "super" funkcijas izsaukums, kā rezultātā *3.7.* attēlā redzamā direktīva būs izmantojama "Angular" ietvarā.

```
import { Directive, ElementRef, Injector, Input, Output, EventEmitter } from '@angular/core';
import { UpgradeComponent } from '@angular/upgrade/static';
import { SimplePerson } from '../simplePerson';
@Directive({
  selector: 'simple-person'
})
export class SimplePersonDirective extends UpgradeComponent {
  @Input() simplePerson: SimplePerson;
  constructor(elementRef: ElementRef, injector: Injector) {
    super('simplePerson', elementRef, injector);
  }
}
```

3.8. att. **"Angular" lietotnes piemēra direktīvas komponente**

Viss ko atliek vēl izdarīt ir papildināt “Angular” lietotnes ietvara moduļa deklarāciju ar jauni izveidoto direktīvu un ieimportēt “ng-upgrade” bibliotēkas moduli, lai ietvars var atrast un atpazīt šīs direktīvas, kā tas ir redzams *3.9.* attēlā .

```
@NgModule({
    imports: [
        BrowserModule,
        UpgradeModule
    ],
    declarations: [
        SimplePersonDirective
    ]
})
```

3.9. att. **“AngularJS” direktīvas reģistrēšana**

Nomigrētās komponentes kuras izmanto dažādus lietotnes servisus nevarēs strādāt ar vecā ietvara servisiem. Un vecā ietvara direktīvas nevar izmantot pārmigrētos servisus uz jauno ietvaru. Tādēļ ir nepieciešams tās pielāgot ar “ng-upgrade” bibliotēkas palīdzību, lai tos spētu izmantot, gan vecā ietvara direktīvas, gan jaunā ietvara komponentes.

Piemēram, lai izmantotu pārmigrēto servisu “simplePersonService”, kurš ir redzams *3.10.* attēlā, ir nepieciešams to pievienot pie apgādātāju saraksta “app.module.ts” faila, kā redzams *3.11.* attēlā. Pēc kā šo servisu varēs ietīt ar “ng-upgrade” bibliotēkas “downgradeInjectable()” funkciju kā redzams *3.12.* attēlā, kā rezultātā ir iespējams izveidot šī servisa “AngularJS” ietvara versiju piešķirot tam jaunu nosaukumu. Tālāk šo servisu ir iespējams lietot kā parastu “AngularJS” ietvaru visās ietvara direktīvās.

```
import { Injectable } from '@angular/core';
import { SimplePerson } from '../simplePerson';

@Injectable()
export class SimplePersonService {
  getPerson() {
    return simplePerson;
  }
}
```

3.10. att. **“Angular” servisa piemērs**

```
import { SimplePersonService } from './simplePersonService';

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule
  ],
  providers: [ SimplePersonService ]
})
```

3.11. att. **“Angular” servisa reģistrēšana**

```
import { downgradeInjectable } from '@angular/upgrade/static';

angular.module('sampleApp', [])
  .factory('simplePersonService', downgradeInjectable(SimplePersonService))
  .component('simplePersonServiceJS', SimplePersonServiceJS);

```

3.12. att. **"Angular" servisa pārveidošana uz "AngularJS" servisu ar downgradeInjectable() funkcijas palīdzību**

Savukārt, lai izmantotu "AngularJS" ietvara servisu "simplePersonService", kurš redzams *3.13.* attēlā, iekš jaunajās "Angular" komponentēs ir ieteicams veidot atsevišķu failu, kurā glabās visus vecā "AngularJS" ietvara servisu pārveidojumus, kā tas ir redzams *3.14.* attēlā. Tādā veidā atvieglojot tā uzturēšanu un atsaukšanos uz to.

```
import { SimplePerson } from '../simplePerson';

export class SimplePersonService {
  getPerson() {
    return simplePerson;
  }
}
```

3.13. att. **"AngularJS" servisa piemērs**

```
import { SimplePersonService } from './simplePersonService';

export function simplePersonServiceFactory(i: any) {
  return i.get('simplePersonService');
}

export const simplePersonServiceProvider = {
  provide: SimplePersonService,
  useFactory: simplePersonServiceFactory,
  deps: ['$injector']
};
```

3.14. att. **"AngularJS" servisa pārveidošana uz "Angular" servisu**

Šajā failā tiek importēti visi vecie "AngularJS" ietvara servisi un tiem tiek veidotas konstantas funkcijas, kuras ir atkarīgas no "AngularJS" ietvara inžekcijas. Pēc kā visas šīs jauni izveidotās servisu atsauksmes ir jāpievieno galvenajā faila "app.module.ts" atsauksmju sarakstā, kā tas ir redzams *3.15*. attēlā.

```
import { simplePersonServiceProvider } from './ajs-upgraded-providers';

@NgModule({
  imports: [
    BrowserModule,
    UpgradeModule
  ],
  providers: [
    simplePersonServiceProvider
  ],
})
```

3.15. att. **"AngularJS" servisa reģistrēšana**

4. TĪMEKĻA PIETEIKUMU FORMU KOMPONENŠU MIGRĒŠANA

Darba autors ikdienā strādā pie apdrošināšanas tīmekļa pieteikumu formu izstrādes Skandināvijas valstu klientiem. Tās tiek veidotas ar "AngularJS" ietvara lietotnes palīdzību, kā dinamiskas vienas lapas lietotnes, kurām jāspēj darboties un labi izskatīties uz visām mūsdienu mobilām ierīcēm. Formas izskats un lauki tiek dinamiski manīti atkarība no klienta ievadītiem datiem un dotajām atbildēm.

Formas tiek izstrādātas ar fokusu būt klientam draudzīgām un pēc iespējas vienkāršākām, lai vecinātu klientu motivāciju tās aizpildīt, nevis nākt uz filiāli vai zvanīt pieteikumu operātoriem. Tādēļ projekta komanda patstāvīgi tiek izaicināta meklēt labākus risinājumus un veikt izmaiņas, gan jau izveidotajās formās, gan formās kuras tiek vēl izstrādātas.

Tīmekļa pieteikumu formu lietotnes projekts tika aizsākts pagājušā gada sākumā. Projekta mērķis bija aizvietot visas novecojušās kompānijas pieteikuma formas uz jaunām, ieskaitot arhitektūru un procesus, kas aiz tām slēpjas. Uz doto brīdi produkcijā jau ir tikušas izlaistas vairākas jaunās pieteikumu formas, bet to skaits ir salīdzinoši mazs ar kopējo ieplānoto formu skaitu, ko paredzēts izveidot, tādēļ projekts nav ne tuvu tā gala stadijai un pat tagad daudzas pamata komponentes un servisi tiek patstāvīgi mainīti un uzlaboti ar mērķi atrast vispiemērotāko un labāko risinājumu.

Vēloties būt mūsdienīgam un nākotnes drošam projektam izstrādātāju komanda uzzinot par "Angular" ietvara izlaišanu, kas notika pagājušā gada rudenī, izlēma novērtēt un ieplānot iespējamo migrāciju uz šo jauno ietvaru pēc iespējas ātrāk, kamēr vēl lietotnes projekta lielums ir salīdzinoši neliels.

Izpētot un novērtējot situāciju tika nolemta stratēģija, kā uzsākt projekta migrāciju neietekmējot nepārtraukto pieteikuma formu izstrādi. Tika nolemts atdalīt no projekta kopējās direktīvas un servisus, kas tiek lietoti visās formās un sastāda galveno formas funkcionālīti un apkopot tās atsevišķā pakotnē. Kopā ar atdalīšanu tika izlemts šīs komponentes migrēt uz jauno ietvaru. Kā rezultātā izveidojot šo pakotni būtu iespējams veidot jaunas pieteikuma formas izmantojot jauno ietvaru, kā arī vienkārši pārmigrēt jau eksistējošās pieteikumu formas, jo sarežģītākās komponentes un servisi būtu pārmigrēti jau pakotnē.

Pirms uzsākt komponenšu migrēšanu darba autors izveidoja vienkāršu testa pieteikumu formas lietotni, kurā testēt pārmigrētās komponentes un servisus un vizuāli novērtēt, vai šīs komponentes darbojas korekti un nav zaudējušas nekādu funkcionalitāti.

4.1. Priekšdarbi

Pirms sākt veikt direktīvu un servisu migrēšanu, darba autors apkopoja un implementēja dažādus ieteiktos pirms migrācijas priekšdarbus, kas tika aprakstīti šī darba nodaļā "Pirms migrācijas priekšdarbi".

Kā pirmais ieteikums bija atjaunot esošā "AngularJS" ietvara versiju uz jaunāko pieejamo versiju vai vismaz "AngularJS 1.5" versiju. Tā kā darba autora tīmekļa pieteikumu formas lietotnes izmantoja "AngularJS 1.5" versiju darba autors šo priekšdarbu izlaida, kaut arī uz to brīdi jaunākā ietvara versija bija "AngularJS 1.6".

Nākamā lieta kas tika ieteikta bija pārveidot projekta struktūru attiecīgi pa komponentēm un servisiem, kur ar noteikto servisu un komponenti saistītie faili atrodas blakus vienā direktorijā. Šāda tīmekļu pieteikumu formu projekta struktūra jau tika uzturēta ar izņēmumu, kad "CSS" faili netika turēti, jo projekts izmantoja kompānijas veidoto kopējo stilu "CSS" bibliotēku, tādēļ tie nebija nepieciešami. Servisu un komponenšu vienību testi tika turēti atsevišķi, tādēļ vienīgais, kas tika darīts bija šo testu pārvietošana uz attiecīgām direktīvu direktorijām.

Pēc versijas un struktūras izmaiņām, nākamais ieteikums bija izmantot moduļu ielādētājus, veidot projektu "TypeScript" programmēšanas valodā un veidot kontrolierus kā klases. Tā ka darba autora tīmekļa pieteikumu formu lietotne, tika izstrādāta "TypeScript" programmēšanas valodā šo priekšdarbu nebija nepieciešams implementēt. Taču bija nepieciešams atjaunot "TypeScript" uz jaunāku versiju, kā rezultātā tika jāpapildina direktīvu un servisu pirmkods, jo atjauninājuma tika pievienotas stingrākas deklarācijas, kas prasīja no darba autora papildināt deklarācijas ar datu tipiem. Papildus tam bija jāaizvieto "TSD" deklarāciju pārvaldnieks ar jauno "Typings" deklarāciju pārvaldnieku izdzēšot nevajadzīgos konfigurācijas parametrus un failus un pievienojot jaunus atbilstoši "Typings" deklarācijas pārvaldnieka bibliotēkas prasībām.

Tālāk darba autors pārliecinājās ka tīmekļa pieteikuma formu lietotnes direktīvas sekoja komponenšu direktīvas vadlīnijām. Pateicoties tam ka projekts tika veidots "TypeScript" programmēšanas valodā arī šis priekšdarbs nebija nepieciešams, jo tas jau atbilda labajai praksei un saturēja visus obligātos atribūtus un netika izmantoti novecojušie atribūti.

Atteikšanās no tvērumiem bija laikietilpīgākais priekšdarbs, kas tika veikts. Tā kā pieteikumu formu pamatā ir daudz dažādu ievades lauki, darba autora projektā bija liela atkarība no tvērumiem jeb "scope" direktīvas atribūta. Šie tvērumi, gan atsevišķās vietās tika vēroti, gan izmantoti kontrolieros un manipulēti. Kaut arī atteikšanās no tvērumiem loģiski neietekmēja nekādi direktīvu un servisu darbību un mijiedarbību savā starpā tas tāpat prasīja lielu koda pārraksti.

Pēc tvērumu pārveidošanas darba autoram bija jāpārveido lietotnes servisi, jo tie visi tika veidoti ar ".factory()" funkciju nevis ar rekomendēto ".service()" funkciju šīs izmaiņas arī bija triviālas, jo viss, kas mainījās bija pirmkoda sintakse, kurā tika veidoti šo servisu klases, kuru skaits nav liels tīmekļa pieteikumu formu lietotnē.

Darba autora tīmekļa pieteikumu formu "AngularJS" lietotne neizmanto klienta puses maršrutēšanu, bet gan "ASP.NET" projekta servera puses maršrutēšanu, tādēļ šis priekšdarbs nebija jāimplementē. Kā arī tā kā tika migrēta un veidota tikai pakotne nevis visa lietotne, tad maršrutēšana bija nepieciešama tikai testa formai.

Pēdējais ieteikums ko darba autors izskatīja bija izvārīšanās no "jqLite" izmantošanas, ko darba autors ignorēja. Kaut arī vairāki avoti neiesaka tā izmantošanu "Angular" ietvaros tehniskā līmenī nav nekādu šķēršļu tā izmantošanā un tīmekļu pieteikuma formu lietotnē dažas komponentes kā datuma izvēlne ir atkarīgas no "jQuery" bibliotēkas, tādēļ tika pieņemts lēmums to atstāt.

4.2. Servisu migrēšana

Pēc pirms migrāciju priekšdarbu veikšanas darba autors uzsāka tīmekļa pieteikumu formu komponenšu pakotnes migrāciju uz jauno "Angular" ietvaru. Kā pirmos darba autors migrēja lietotnes servisus. Tie tika migrēti pirmie, jo lietotnes direktīvas un kontrolieri ir no tiem atkarīgi, tādēļ loģiskais lēmums bija sākt ar servisu migrēšanu sākot ar tiem, kuri ir neatkarīgi no citiem servisiem.

Lietotnē pavisam tiek izmantoti ap divdesmit servisiem, kā kļūdu apstrādes, teksta tulkošanas, formu validācijas, sarakstu veidošanas u.c. servisi. Tā kā lietotnes servisi tika veidoti ar "TypeScript" valodas klasēm to migrēšana bija pavisam triviāla. Mazliet izmainot pirmkoda sintaksi un aizvietojot vecās "AngularJS" ietvara klases ar to ekvivalentām jaunajā "Angular" ietvara klasēm.

Papildus, lai sekotu labā stila vadlīnijām, darba autors atdalīja šajos servisu failos definētās un izmantotās objektu klases un to prototipus iznesot tos atsevišķos failos. Pēc kā visi nomigrētie servisi tika reģistrēti lietotnes galvenajā moduļa atsauksmju sarakstā.

Kopumā darba autors konstatēja, kad servisu migrācija ir diezgan vienkāršs un ātrs process pateicoties tam kad servisu pirmkods tika veidots "TypeScript" programmēšanas valodā un servisi tika pārveidoti no ".factory()" uz ".service()" funkciju pirms migrācijas sākuma, kā rezultātā darba autoram neradās nekādi šķēršļi un grūtības pie servisu migrācija.

4.3. Direktīvu un kontrolieru migrēšana

Pēc visu servisu migrācijas darba autors veica direktīvu un to kontrolieru migrāciju uz "Angular" ietvaru pārveidojot tās par komponentēm. Tīmekļu pieteikumu formu lietotnē pavisam uz migrēšanas brīdī bija ap trīsdesmit komponentēm, kā dažādi ievad lauki – teksta, datuma, telefona u.c., radio pogas izvēlne, rūtiņu izvēlne, izvēlnes saraksts un daudzas citas ar formām saistītās komponentes.

Sākumā tika atdalīti visas direktīvās izmantotās objektu klases un to prototipi atdalot tos atsevišķos failos, lai sekotu izstrādātāju noteiktajām stila vadlīnijām. "AngularJS" ietvarā direktīvas un kontrolieri tika veidotas kā divas atsevišķas klases, taču "Angular" ietvarā abas šo klašu funkcijas pilda komponente. Tādēļ darba autors apvienoja abas šīs klases vienā komponentes klasē. Tā rezultātā varēja atbrīvoties no liekajiem direktīvas atribūtiem un direktīvas savienojuma funkcijas, atstājot visu komponentes loģiku vienā vietā. Un beigās darba autors papildināja "HTML" šablona failu ar nepieciešamo jaunā ietvara sintaksi, kur atšķirībā no "AngularJS" ietvara, specifiskie ietvara atribūti tiek aplikti ar kvadrāta un apaļajām iekavām. Pēc kā visas nomigrētās komponentes tika reģistrētas lietotnes galvenajā moduļa deklarāciju sarakstā.

Kopumā darba autors konstatēja, kad direktīvu un kontrolieru pārveidošana uz komponentēm bija sarežģītāks process par servisu migrāciju, bet tas tāpat bija salīdzinoši vienkāršs un veikls. Atkal pateicoties "TypeScript" valodas izmantošanai un laicīgai atbrīvošanās no tvērumiem, tika atvieglota pirmkoda migrēšanu atstājot lielāko tā daļu, tādu kāds tas bija sākuma un aizvācot liekas vecā ietvara specifikas. Darba autora rezultējošo komponenšu pirmkods kļuva daudz tīrāks un īsāks atvieglojot ārējo servisu importēšanu un aizvācot liekos atribūtus, kas uzlaboja tā lasāmību.

4.4. Testu migrēšana

Darba autora tīmekļa pieteikuma formu lietotnei ir divu veidu testi – lietotāju saskarnes testi un komponenšu vienību testi.

Lietotāju saskarnes testiem tiek izmantos "Protractor" testēšanas ietvars, kurš atbalsta gan veco "AngularJS", gan jauno "Angular" ietvaru. Viss kas nepieciešams, lai testēšanas ietvars varētu darboties ar "Angular" lietotni ir atjaunot tā versiju uz "Protractor 5". Tīmekļu pieteikumu formu lietotne izmantoja "Protractor 4" versiju un to atjaunojot nebija jāveic nekādas papildus izmaiņas. Lietotāju testi tiek izpildīti identiski abos ietvaros un tos nav nepieciešams modificēt viss, kas nepieciešams, lai tie varētu korekti strādāt ir saglabāt esošās lauku identifikatora atribūta vērtības.

Vienību testi lietotnes projektā tiek veidoti katrai tīmekļa pieteikuma komponentei ar “Jasmine” testēšanas ietvara palīdzību un tiek izpildīti ar “Karma” testa veicēja palīdzību, kas atbalsta gan “AngularJS” , gan “Angular” ietvara lietotnes. Taču atšķirībā no lietotāj saskarnes testiem, nepietiek tikai ar ietvara versijas atjaunošanu. Tā kā direktīvu vienību testi izmanto “AngularJS” specifiskos servisus, kam nav ekvivalenti servisi jaunajā “Angular” ietvarā kā piemēram “RootScopeService”, kurš atbild par lietotnes tvērumiem, kuri vairs netiek izmantoti “Angular” ietvarā. Tādēļ ir jāveic pilnīga visu direktīvu vienību testu pārrakstīšana, izstrādes laika gaitā, pakāpeniski tam atvēlot laiku un atgūstot visu iepriekšējo testu daudzumu.

Tīmekļu pieteikuma formu lietotnes servisa vienību testi atšķirībā, no direktīvu vienību testiem tika veidoti citādāk. Tie neizmantoja nekādus vecajam “AngularJS” ietvaram specifiskos servisus, bet gan tika veidoti, kā parastas “OOP” klases vienību testi, kas testē klases metodes un vērtības ko tās atgriež. Tāpēc tos bija vienkārši papildināt, lai tie sintaktiski atbilst jaunajam “Angular” ietvaram.

REZULTĀTI

Darba izstrādes laikā autors nomigrēja tīmekļa pieteikumu formu "AngularJS" lietotni uz "Angular" ietvaru. Autors pārrakstīja lietotnes komponenšu un servisu vienību testus, lai tie būtu saderīgi ar jauno ietvaru. Lietotnes migrācijai tika pielietota lielā sprādziena migrācija un veikti vairāki pirms migrācijas priekšdarbi - struktūras izmaiņas, servisu un komponenšu pārveidošana un ārējo atkarību versiju atjaunināšana.

Lietotnes migrācija tika veikta, balstoties uz oficiālo "Angular" izstrādātāju veidotu dokumentāciju un darba autora savāktajiem dažādu izstrādātāju rakstiem, kuri bija veidoti gan kā migrācijas pamācības, gan kā ieteikumu un pieredzes dalīšanās raksti. No šiem rakstiem tika izvilkti un apkopoti dažādi ieteiktie pirms migrācijas priekšdarbi un aprakstīti to implementēšanas ieguvumi.

Tika arī apkopotas, aprakstītas un izvērtētas iespējamās lietotņu migrācijas stratēģijas, tādas kā lielā sprādziena stratēģija un inkrementālā stratēģija, kur inkrementālajai stratēģijai tika piedāvātas divas pieejas - vertikālā un horizontālā griezuma projekta migrēšana. Tika noskaidrots, ka lielā sprādziena stratēģiju ir iesakāms izmantot maziem lietotņu projektiem, kuri nav atkarīgi no daudzām arējām bibliotēkām un izstrādātājiem ir atvēlēts pietiekams laiks pilnīgai lietotnes migrācijai. Inkrementālā stratēģija ir paredzēta lielākiem projektiem vai projektiem ar daudzām atkarībām no ārējām bibliotēkām, kas vēl neatbalsta jauno ietvaru un kur nav iespējama pilnīga lietotnes migrācija, tādēļ tā ir jāveic pakāpeniski. Inkrementālajai stratēģijai tika izveidots apraksts ar piemēriem un to pielietojumiem.

Balstoties uz darba autora iegūto pieredzi tīmekļa pieteikumu formu lietotnes migrācijā, tika papildināta apkopotā migrācijas procesa informācija ar darba autora atziņām un novērojumiem. Projektiem, kas ir veidoti "TypeScript" programmēšanas valodā, ir nepieciešama tā versiju atjaunināšana un definīciju pārvaldnieka maiņa no "TSD" uz "Typings", kā arī ir ieteicams pārveidot lietotnes kontrolierus no funkcijām par klasēm, ja tas tā jau netiek darīts. Ir jāpārbauda ārējo bibliotēku saderība ar jauno ietvaru. Nav nepieciešams aizvietot vai izvairīties no "jqLite" izmantošanas, jo var droši lietot "jQuery" bibliotēku "Angular" ietvarā. Jāizvairās no dekoratoru izmantošanas. Bez lietotnes pirmkoda ir nepieciešams migrēt arī vienību testus.

Autora ieteikumi ir izmantojami tīmekļa pieteikumu formu lietotņu migrēšanā no AngularJS uz Angular ietvaru.

SECINĀJUMI

Darba autors secināja, kad "AngularJS" lietotnes migrācija ir laikietilpīgs un pietiekami sarežģīts process, kurš slēpj sevī dažādus šķēršļus. Tas ir ne tikai atkarīgs no lietotnes projekta lieluma, funkcionalitātes un sarežģītības, bet vēl papildus no pirmkoda kvalitātes, kādā tas tika veidots, jeb cik stingri projekts pieturējās pie ietvara izstrādātāju ieteiktajām stila vadlīnijām. Kā arī tas ir atkarīgs no ietvara versijas, projekta vecuma, programmēšanas valodas, kurā tas tiek izstrādāts, ārējo bibliotēku atkarību daudzuma un to nozīmes lietotnes projektā. Tādēļ ir kārtīgi jāizpēta esošie lietotnes projekti un, balstoties uz šiem kritērijiem ir iespējams novērtēt migrācijas laikietilpīgumu un sarežģītību. Ir jāizvērtē nepieciešamība pēc migrācijas uz jauno ietvaru kopumā, lai laiks un resursi, kas tiks tam veltīti būtu pamatoti ar ieguvumiem, kas tiks gūti lietojot jauno ietvaru.

Pat tad, ja projekts neplāno migrāciju uz jauno ietvaru, autors uzskata, ka nodaļas "Pirms migrācijas priekšdarbi" minētie esošā projekta pirmkoda uzlabojumi ir vērtīgi jebkuram projektam, neskatoties uz to, vai tas tiks vai netiks nākotnē migrēts uz jauno ietvaru. Šie priekšdarbi uzlaboja autora tīmekļa pieteikuma formas pirmkoda kvalitāti un lasāmību. Piedevām, tie atviegloja ietvara migrāciju, jo tie kļuva līdzīgāki dotiem piemēriem oficiālajā "Angular" ietvara migrācijas dokumentācijā un citu izstrādātāju izveidotajām migrācijas pamācībām. Tādēļ lietotnes projektus, kas tiek izstrādāti vai plānoti ilglaicīgi tikt uzturēti, būtu vērts izpētīt un ievest savos lietotnes projektos.

Autors, izpētot citu izstrādātāju rakstus un dokumentāciju, nekonstatēja lielu uzskatu un ieteikumu dažādību, pamatinformācija visos avotos bija līdzīga un tika piedāvātas visur tās pašas migrācijas stratēģijas un ieteikumi ar minimālām papildinājumiem no konkrēto rakstu autoriem. Darba autors citu izstrādātāju rakstos saskatīja atziņu un ieteikumu trūkumu, kas būtu balstīti uz konkrētiem piemēriem, nevis tiktu pasniegti kā vispārinājumi, ko iespējams varētu skaidrot ar salīdzinoši neilgo ietvara pastāvēšanas laiku vai projektu migrāciju retumu reālās biznesa lietotnēs.

Bez tā visa, autors, izpētot dekoratora bibliotēkas, kuru pamatdoma bija motivēt izstrādātājus migrēt uz jauno ietvaru, secināja, ka no tiem ir jāizvairās to zemās pievienotās vērtības dēļ un fakta, ka tie netiek vairs uzturēti līdz jaunā ietvara izmaiņām, kas rada risku papildus sarežģīt migrācijas procesu.

IZIMANTOTĀ LITERATŪRA UN AVOTI

1. Coman Hamilton, *A sneak peek at the radically new Angular 2.0.* [tiešsaiste]. [atsauce 08.05.2017]. Pieejams internetā: https://jaxenter.com/angular-2-0-112094.html
2. Coman Hamilton, *Angular 2.0 announcement backfires.* [tiešsaiste]. [atsauce 08.05.2017]. Pieejams internetā: https://jaxenter.com/angular-2-0-announcement-backfires-112127.html
3. Frederic Lardinois, *Google launches final release version of Angular 2.0.* [tiešsaiste]. [atsauces 08.05.2017]. Pieejams internetā: https://techcrunch.com/2016/09/14/google-launches-final-release-version-of-angular-2-0/
4. Igor Minar, *Ok… let me explain: it's going to be Angular 4.0, or just Angular.* [tiešsaiste]. [atsauce 08.05.2017]. Pieejams internetā: https://techcrunch.com/2016/09/14/google-launches-final-release-version-of-angular-2-0/
5. Chris Nwambe, *Seamless Ways to Upgrade Angular1.x to Angular 2.* [tiešsaiste][atsauces 08.05.2017]. Pieejams internetā: https://scotch.io/tutorials/seamless-ways-to-upgrade-angular-1-x-to-angular-2
6. *Upgrading from AngularJS*. [tiešsaiste]. [atsauce 15.05.2017]. Pieejams internetā: https://angular.io/docs/ts/latest/guide/upgrade.html
7. Todd Motto, *Walkthrough to upgrade an Angular 1.x component to Angular 2*. [tiešsaiste]. [atsauce 15.05.2017]. Pieejams internetā: https://toddmotto.com/walkthrough-to-migrate-an-angular-1-component-to-angular-2/
8. Maarten Hus, *The road to Angular 2.0 part 6: Migration*. [tiešsaiste]. [atsauce 15.05.2017]. Pieejams internetā: http://dontpanic.42.nl/2015/08/the-road-to-angular-20-part-6-migration.html
9. *Quick start*, [tiešsaiste]. [atsauce 15.05.2017]. Pieejams internetā: https://www.typescriptlang.org/docs/tutorial.html
10. Alexandra Atzl, *What I learned at ng-conf: Write AngularJS with migration in mind*. [tiešsaiste]. [atsauce 15.05.2017]. Pieejams internetā: http://www.effectiveui.com/blog/2015/04/20/learned-ng-conf-write-angularjs-migration-mind/
11. Brad Green, *Angular 1 and Angular 2 integration: the path to seamless upgrade.* [tiešsaiste]. [atsauce 16.05.2017]. Pieejams internetā: https://angularjs.blogspot.com/2015/08/angular-1-and-angular-2-coexistence.html

12. K. Aishwarya, *Migration from Angular 1.x to Angular 2.0*. [tiešsaiste]. [atsauce 16.05.2017]. Pieejams internetā: https://aishwaryakumthekar.wordpress.com/author/aishwarya275/
13. Pascal Precht, *Upgrading Angular apps using ngUpgrade.* [tiešsaiste]. [atsauce 16.05.2017]: Pieejams internetā: https://blog.thoughtram.io/angular/2015/10/24/upgrading-apps-to-angular-2-using-ngupgrade.html
14. Daniel Figueiredi Caetano, *Upgrading your application to Angular 2 with ng-upgrade.* [tiešsaiste]. [atsauce 16.05.2017]. Pieejams internetā: http://blog.rangle.io/upgrade-your-application-to-angular-2-with-ng-upgrade/
15. Jesus Rodriguez, *AngularJS, Angular, Angular v4*. [tiešsaiste]. [atsauce 16.05.2017]. Pieejams internetā: http://angular-tips.com/blog/2017/03/angularjs-angular-and-angular-v4/
16. *Migrating an App to a newer version*. [tiešsaiste]. [atsauce 20.05.2017]. Pieejams internetā: https://docs.angularjs.org/guide/migration
17. *Style guide*. [tiešsaiste]. [atsauce 21.05.2017]. Pieejams internetā: https://angular.io/docs/ts/latest/guide/style-guide.html
18. Elias Carlston, *History and Background of JavaScript Module Loaders.* [tiešsaiste]. [atsauce 20.05.2017]. Pieejams internetā: https://appendto.com/2016/06/the-short-history-of-javascript-module-loaders/
19. Pascal Precht, *Service vs Factory – Once and For All.* [tiešsaiste]. [atsauce 21.05.2017]: Pieejams internetā: https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html
20. Dave Baskin, *Yes, I Used JQuery In My Angular 2 Application.* [tiešsaiste]. [atsauce 21.05.2017]: Pieejams internetā: http://www.wintellect.com/devcenter/dbaskin/yes-used-jquery-angular2-application
21. Rachit Agarwal, *How to upgrade AngularJS apps to AngularJS 2.0.* [tiešsaiste]. [atsauce 21.05.2017]: Pieejams internetā: http://www.algoworks.com/blog/upgrading-angularjs-apps-to-angularjs2-0/
22. Victor Savkin, *Two Approaches to Upgrading Angular Apps.* [tiešsaiste]. [atsauce 21.05.2017]: Pieejams internetā: https://blog.nrwl.io/two-approaches-to-upgrading-angular-apps-6350b33384e3

Bakalaura darbs „AngularJS lietotnes migrācija uz Angular” izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____ Edgars Andrucis

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Darba vadītājs: Dr.dat. Uldis Straujums _____ ___.05.2017.

Recenzents: Dr.dat. Aivars Niedrītis

Darbs iesniegts Datorikas fakultātē 29.05.2017.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe_____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

___.06.2017. prot. Nr. ___

Komisija sekretār___: _____