



**Софийски университет „Св. Климент Охридски“**

Факултет по математика и информатика

*Катедра „Изчислителни системи“*

# **ДИПЛОМНА РАБОТА**

на тема

**„Уеб-базирано приложение за създаване,  
визуализиране, валидиране и редактиране на  
електронни структурирани документи за целите на  
заявяване и предоставяне на електронни  
административни услуги“**

**Дипломант:** Росица Стоянова Георгиева

**Специалност:** Технологично предприемачество и иновации в  
информационните технологии

**Факултетен номер:** M22411

**Научен ръководител:**

доц. д-р Камен Спасов

София, 2014 г.



# Съдържание

Съдържание.....	3
Списък на фигурите .....	5
1    Увод .....	6
1.1    Актуалност на проблема и мотивация.....	6
1.2    Цел и задачи на дипломната работа .....	6
1.3    Анотация на решението .....	6
1.4    Структура на дипломната работа .....	7
2    Предметна област .....	9
2.1    Основна терминология .....	9
2.1.1    Електронно управление и Електронно правителство.....	9
2.1.2    Електронна административна услуга .....	9
2.1.3    Електронен документ .....	9
2.1.4    Оперативна съвместимост .....	10
2.1.5    Информационни обекти .....	11
2.2    Съществуващи решения.....	12
2.3    Предимства на новото решение .....	12
3    Използвани технологии .....	14
3.1    Структура и представяне на обекти.....	14
3.1.1    Стратегия за разработка на клиентски приложения – SPA.....	14
3.1.2    Платформа за разработка на клиентски приложения – Durandal.....	16
3.1.3    Библиотека за разработка на клиентски приложения – Knockout .....	18
3.1.4    Библиотека асинхронно зареждане на ресурси – RequireJS .....	19
3.1.5    Добра практика Revealing Prototype Pattern .....	21
3.2    Управление на данни .....	22
3.3    Работа с XML съдържание .....	23
4    Представяне на решението.....	25
4.1    Структура на приложението .....	25
4.1.1    Информационни обекти .....	25
4.1.2    Шаблони .....	26
4.1.3    Конфигурация .....	27

4.1.4	Валидация.....	29
4.1.5	Извличане на данни .....	29
4.2	Реализиране функционалността на приложението .....	29
4.2.1	Представяне на ИО от тип Термин.....	30
4.2.2	Представяне на ИО от тип Номенклатура .....	30
4.2.3	Представяне на ИО от тип Стойност .....	31
4.2.4	Представяне на ИО от тип Сегмент.....	32
4.2.5	Представяне на ИО от тип Документ.....	36
4.2.6	Работа с XML съдържание и JavaScript обекти.....	38
4.2.7	Създаване, визуализиране и редактиране на електронен структуриран документ 41	
4.2.8	Валидиране на електронен структуриран документ .....	48
4.2.9	Управление на данните .....	51
5	Тестване.....	54
6	Внедряване .....	56
6.1	Подход за интеграция .....	56
6.2	Входни данни .....	56
6.2.1	Вид документ за редакция.....	57
6.2.2	XML съдържание .....	57
6.3	Определяне режима на работа на приложението.....	57
6.4	Изходни данни .....	58
7	Практическо приложение.....	59
8	Заключение .....	61
9	Референции .....	62
10	Приложения .....	63
10.1	Пример за XSD дефиниция за структуриран документ .....	63
10.2	Пример за XML съдържание на документ.....	64
10.3	Пример за генериран документ с грешки, възникнали при валидация .....	66
10.4	Описание на съдържанието на приложения компакт диск.....	66

## Списък на фигурите

Фиг. 1 Структура на SPA .....	15
Фиг. 2 Взаимодействие между обектите в SPA .....	16
Фиг. 3 Структура на Durandal.....	17
Фиг. 4 Структура на обектите на приложението .....	26
Фиг. 5 Структура на изгледите в приложението .....	27
Фиг. 6 Визуално представя на изброени стойности .....	32
Фиг. 7 Йерархия от обекти описваща електронния документ .....	33
Фиг. 8 Екран за режим на редакция на електронен структуриран документ .....	43
Фиг. 9 Екран за режим на визуализация на електронен структуриран документ .....	46
Фиг. 10 Екран на изпълнени тестове за приложението .....	55

# **1 Увод**

## **1.1 Актуалност на проблема и мотивация**

Една от задачите с най-голямо значение в контекста на стратегическото развитие на България е създаването и въвеждане в експлоатация на електронно правителство. Гарантирането на оперативна съвместимост и еднозначна комуникация както между граждани/бизнес и администрация, така и между административните единици в рамките на държавната администрация е едно от предизвикателствата за осъществяването на тази идея. Решението на този казус е реализирано чрез дефиниране на структурирани изисквания, на които трябва да отговаря всяка страна вземаща участие в споменатата комуникация. Така описаните изисквания са организирани в публично достъпни регистри, наречени регистри за оперативна съвместимост (РОС). Един от тях е Регистърът на информационните обекти, който ясно и конкретно дефинира структурата на всеки обект използван за обмен на данни (информационен обект), правила за неговата обработка, визуализация и валидация. По този начин се гарантира семантична и синтактична еднозначност на обектите и данните, обменяни в рамките на електронното правителство.

За да бъдат достъпни и удобни за обработка от ползвателите на услугите на електронното правителство, е необходимо така дефинираните информационни обекти да бъдат представени в разбираем и удобен за използване вид. Затова може да се говори за необходимост от унифициран подход при редактиране и визуализиране на електронни структурирани документи. Той трябва да предоставя възможност съответните информационни обекти да бъдат разпознаваеми от заинтересованите страни и да бъдат модифицирани и обработвани по предсказуем начин.

## **1.2 Цел и задачи на дипломната работа**

Целта на дипломната работа е да опише подход за структуриране и разработка на продукт, който базирайки се на налагащите се тенденции и технологии в уеб-базираните приложения, предлага решение за работа с електронни структурирани документи. Така разработения продукт предоставя богат и интерактивен интерфейс за работа с електронни структурирани документи за потребителите, но също така и унифициран метод за разширяване и поддръжка на приложението от разработчиците.

## **1.3 Анотация на решението**

В рамките на тази дипломна работа ще бъде представено софтуерно приложение, което предоставя имплементация на такъв унифициран подход. Ще бъдат разгледани структурата и методологията за разработка на това приложение. Ще бъде изложено описание на използваната SPA (Single Page Application) методология за създаване на уеб-базирани приложения. Тази технология позволява структурата на приложението да се организира в логически единици, които

улесняват неговата поддръжката и разширяемост. Благодарение на асинхронния характер на зареждане на необходимите (и само на необходимите) програмни единици (JavaScript файлове, HTML файлове, допълнителни данни), усещането за време на изчакване за потребителя е значително подобро. Характерът на тази методология (презентационната логика и работата с дефинираните обекти се случват на клиента/browser) предоставят възможност за интерактивност и богат интерфейс за съответното уеб-базирано приложение.

Някои от основните характеристики на описаното приложение са, както следва:

- то е уеб-базирано;
- предоставя достъпен, интуитивен и интерактивен интерфейс;
- предоставя визия следваща съвременните тенденции за изглед в уеб-формат;
- работи с електронни структурирани документи – изпълнява изискванията за визуализиране, валидиране и обработка за реферираните информационни обекти, дефинирани в РОС;
- лесно разширяемо - има унифициран и стандартизиран метод за добавяне на имплементация за нови документи, отговарящи на предварително зададени условия;

Така описаният продукт ще бъде тясно интегриран с приложението предоставящо за гражданите публичен достъп за заявяване на електронна административна услуга. Също така той ще бъде достъпен и от системата използвана от служителите в администрацията, чрез която се осъществява самото изпълнение на съответната услуга, тъй като това изпълнение също изисква възможност за работа с електронни структурирани документи.

## **1.4 Структура на дипломната работа**

В глава 2 – „Предметна област” е направен кратък обзор на предметната област. Обяснени са основни понятия свързани с електронното управление, които са необходими за изясняване приноса на разглежданото решение. Направен е анализ на съществуващи решения на поставения проблем и са изброени предимствата на тук представеното такова.

В глава 3 – „Използвани технологии” са изброени и описани използваните технологии и методологии за реализиране на решението. За всяка от тях е направен кратък обзор и са представени аргументи за нейния избор.

В глава 4 – „Представяне на решението” е изложено описание на самото решение, неговата имплементация и организация. В първата част (глава 4.1 „Структура на приложението”) е описана архитектурата на приложението и значението на всяка отделна логическа единица. Във втората част (глава 4.2 „Реализиране функционалността на приложението”) е описана в детайли самата реализация на разглежданото приложение. Разгледани са методите за работа с отделните обекти, начините на прилагане на използваните технологии и добри практики. С цел по-голяма яснота е разгледана конкретна реализация на примерен електронен структуриран документ.

В глава 5 – „Тестване” е описана методологията приложена за тестване на приложението. Представен е метод за автоматизиране на дефинирани тестове и е представен пример за реализирането му.

В глава 6 – „Внедряване“ е направено кратко представяне на подхода за интеграция на приложението с други системи. Дефинирани са необходимите входни данни (глава 6.1 – „Входни данни“) – какви видове са и какъв е метода за тяхното прочитане. Също така е обяснено как се задава режима на работа на приложението в зависимост от подадените входни данни. В последната част от тази глава са представени изходните данни на приложението (в какъв вид се генерира крайния резултат).

В глава 7 – „Практическо приложение“ е представен проекта, в който е внедрено представеното решение. Описани са изискванията, на които той трябва да отговаря и са дефинирани функционалностите, които той предоставя.

Глава 8 – „Заклучение“ представлява кратко обобщение на ползите и предимствата на продукта, както и са споменати някои предложения за по-нататъшно доразвиване и подобрене.

В глава 9 – „Референции“ са изброени всички използвани литературни източници в текущата теза.

В глава 10 – „Приложения“ са представени примерни резултатни документи, генерирани чрез използване на приложението. Също така е цитирана XML дефиницията на примерния документ (XSD), разглеждан при представяне на решението. Добавено е и кратко описание на съдържанието на компакт диска, на който е записано самото приложение.



## 2 Предметна област

За да бъде изяснено значението и приноса на предмета на изложената дипломна работа е необходимо предварително да бъде описан контекстът на проблема, който представения продукт решава. В тази точка ще бъдат разгледани и обяснени основни термини и понятия, необходими за представянето на проблемите на предметната област и текущото състояние на проблема. Ще бъдат разгледани някои съществуващи решения, ще бъде направен анализ на техните недостатъци и ще се направи сравнение с предложеното тук решение.

### 2.1 Основна терминология

#### 2.1.1 Електронно управление и Електронно правителство

Електронно правителство като част от електронното управление е заложено като основен приоритет за развитие на страната [1]. Докато електронното управление включва в себе си всички аспекти на комуникация между граждани/бизнес и доставчици на електронни услуги във всички сфери на обществения живот (включително здравеопазване, образование, бизнес и др.), то електронното правителство се отнася до пряката комуникация между гражданите/бизнеса и административните органи, предоставящи електронни административни услуги. Един от основните принципи на електронното управление е акцентирането върху улесняване на комуникацията между заинтересованите страни в контекста на предоставяне и ползване на електронни услуги [2]. Участниците в тази комуникация – граждани/бизнес и служители в държавни, общински и местни администрации се явяват потребителни на електронни услуги.

#### 2.1.2 Електронна административна услуга

Дефинирането и подробното описание на електронните административни услуги, както и методологията на тяхното предоставяне е изложено в ЗЕУ (Закон за електронното управление). Под електронна административна услуга в рамките на текущата теза се взема предвид следната дефиниция: *„Електронни административни услуги са административните услуги, предоставяни на гражданите и организациите от административните органи, услугите, предоставяни от лицата, на които е възложено осъществяването на публични функции, както и обществените услуги, които могат да се заявяват и/или предоставят от разстояние чрез използването на електронни средства.“* [3]

В своята същност процесът по предоставяне на такава услуга е тясно свързан с възможността за работа с електронни документи.

#### 2.1.3 Електронен документ

Електронният документ е основното средство за заявяване и предоставяне на резултат от дадена услуга. Също така, той е и необходим инструмент за изпълнението на отделните стъпки в рамките на самия процес за обработка на заявление за електронна услуга и изготвяне на изисквания резултат. Поради тази причина работата с електронни документи и изобщо изясняването на това понятие по еднозначен начин е абсолютно необходимо за осъществяването на предоставяне на електронни административни услуги. ЗЕДЕП (Закон за електронния документ и електронния подпис) е нормативният документ, който урежда дефинирането на този термин,

както и други понятия свързани с него и необходими за неговото поясняване. Според него „Чл. 3. (1) *Електронен документ е електронно изявление, записано върху магнитен, оптичен или друг носител, който дава възможност да бъде възпроизведено.*”, където за електронно изявление се има предвид: „Чл. 2. (1) *Електронно изявление е словесно изявление, представено в цифрова форма чрез общоприет стандарт за преобразуване, разчитане и визуално представяне на информацията.*” [4].

#### **2.1.4 Оперативна съвместимост**

Поради организацията на административните структури в страната, всяка администрация или административен орган разполага с определена степен на автономност, регламентирана нормативно чрез вътрешни правила за съответната администрация. Това позволява на отделните администрации да разполагат със свободата да предоставят дефинирани услуги спрямо собствени правила и изисквания, стига те да отговарят на нормите наложени за цялостното административно обслужване за страната. Именно това ниво на автономност позволява в отделните административни единици предоставянето на дадена електронна административна услуга да се извършва от различни информационни системи. Тези системи представляват АИС (административни информационни системи), където *„Административна информационна система е система, която осигурява поддържането и обработката на данните за оборота на електронни документи и документи на хартиен носител при предоставяне на административни услуги и изпълнение на административни процедури.”* [3]. Това поражда проблем относно комуникацията и синхронизацията на тези системи. Налага се нужда от формализиране на комуникацията между всички заинтересовани страни, т.е. всички участници в процеса по заявяване и предоставяне на електронна административна услуга. Именно тази синхронизация на метода на комуникация между отделни информационни системи се нарича оперативна съвместимост [6].

Точната дефиниция на понятието оперативна съвместимост спрямо ЗЕУ – *„Оперативна съвместимост е способността на информационните системи да обработват, съхраняват и обменят електронни документи и данни помежду си, използвайки единни технологични стандарти и процеси”* [3].

Във фокуса на този документ в повече детайли ще бъде разгледана частта от ОС (оперативна съвместимост) наречена семантична ОС. Тя също е нормативно дефинирана и разгледана в повече конкретика в ЗЕУ. Спрямо този закон *„Семантична оперативна съвместимост е елемент на оперативната съвместимост, означаващ способността за еднаква интерпретация на едни и същи данни от различни информационни системи.”* [3].

С цел гарантиране, именно на семантичната ОС са създадени набор от структурирани изисквания, на които трябва да отговаря всеки участник в процеса на обмен на електронни документи. Тези изисквания са организирани в публично достъпни регистри – РОС. Създадени са следните регистри:

- Регистърът на регистрите и данните съдържа данни за всички регистри, списъци и други набори от данни, водени от административните органи в Република България,

осигурява уникален индекс за всеки набор от данни или раздел от набор от данни, поддържа определения на неунифицирани и унифицирани данни и унифицирани определения на етапи на административни услуги и процедури.

- Регистърът на информационните обекти съдържа формализирани технологични описания на информационните обекти, събирани, създавани, съхранявани и обработвани от административните органи в рамките на тяхната компетентност.
- Регистърът на електронните услуги съдържа формализирани технологични описания на електронните административни услуги и вътрешните електронни административни услуги [7] .

#### **2.1.5 Информационни обекти**

В регистъра на информационните обекти са организирани формализираните дефиниции на данни обменяни по електронен път в рамките на електронното правителство. Именно те са прякото средство за комуникация между всички административни единици. Затова изискванията, на които трябва да отговарят техните дефиниции и формат, тяхното категоризиране и изобщо организацията на тези регистри са нормативно регламентирани в наредба – НРИОЕУ (Наредба за регистри на информационните обекти и на електронните услуги). Като формална дефиниция на информационен обект е заложен следният текст: „Чл. 4. (1) Информационен обект са единични или съставни данни, събирани, създавани, съхранявани или обработвани от административните органи в рамките на тяхната компетентност.” [8]. В РИО (регистър на информационните обекти), информационните обекти са организирани в следните категории:

- „термин“ – понятие, което се интерпретира еднозначно от всички участници в административния процес;
- „номенклатура“ – краен списък от тематично свързани термини, вписани в регистъра;
- „стойност“ – представя величина и се описва с краен брой значения, зададени с формални ограничения;
- „сегмент“ – структура, съставена от вече вписани в регистъра термини, номенклатури, стойности и/или други сегменти;
- „документ“ – сегмент, за който е осигурено програмно приложение, даващо възможност за пълна, точна и вярна визуализация на съдържащите се данни.

В по-горе споменатата наредба обстойно и в конкретика са описани точната структура и формат за всяка от тези категории, както и самият процес по тяхното вписване в РИО. Изброени са основните характеристики, които описват всеки вид ИО (информационен обект) в зависимост от неговото значение и специфика. Такива характеристики са например наименование, уникален регистров идентификатор (УРИ), статус, описание на указания за обработка, визуализация и валидация, описание на XML дефиниция (където има смисъл). Дефиниран е също и точният стандарт, използван за описание на XML структурата на ИО – XSD 1.0 (Чл.10(1)). Нормативното регламентиране в такова ниво на детайлност, на технологичната специфика за работа с тези обекти е едно от необходимите условия за гарантиране на уеднаквяване и стандартизиране на представянето на данните, обект на комуникация в процеса на предоставяне на ЕАУ (електронна административна услуга).

## 2.2 Съществуващи решения

В предходните секции бяха изяснени основните термини и положения, описващи контекста, в който ще бъде поставен проблемът, чието решение е обект на настоящата дипломна работа. Беше поставен проблемът, налагащ изисквания за семантична оперативна съвместимост и беше описан метод за неговото решение, изпълняващ тези изисквания – нормативното регламентиране и реализиране на РОС и по-специално РИО. По този начин е намерено технологично решение за уеднаквяване на изискванията за обмена на данни. Конкретната имплементация на тези обекти и тяхното представяне във вид позволяващ те да бъдат достъпни и удобни за обработка от ползвателите на услугите на електронното правителство остава задача, която всеки доставчик на ЕАУ има свободата да реши по начин, удачен в контекста неговите вътрешни работни процеси. Независимо от автономния подход за техническа реализация е нужно дефинираните в РОС информационни обекти да бъдат представени в разбираем и удобен за използване вид за крайния потребител.

Към момента има няколко съществуващи варианта за предоставяне на решение, удовлетворяващо тези изисквания.

Един от тях е предоставянето на офлайн приложения, които да предоставят тази функционалност за работа с електронни структурирани документи. Тези приложения имат възможности за предоставяне на богат интерфейс и голям набор от функционалности [9]. Недостатък на този тип приложения е необходимостта всеки потребител да инсталира допълнително съответния софтуер.

Друг вариант, който се предоставя като решение за предоставяне на унифициран интерфейс за работа с електронни структурирани документи са уеб-базирани приложения, които съдържат уеб-страници за редактиране и визуализиране на конкретни документи. Този подход решава проблема с нуждата от допълнителна инсталация, тъй като средата за работа с тези продукти е уеб-браузър. По този начин се постига и ефект на платформена независимост. От технологична гледна точка, проблемът на този тип решение е, че поради своя характер на конкретика (за всеки един документ се създава отделна страница) преизползваемостта, респективно и поддръжката му не са достатъчно добре оптимизирани [10]. От потребителска гледна точка, поради ограниченията наложени от характера на протокола, използван за уеб-приложения повечето от тях не предоставят достатъчно интерактивен и удобен интерфейс – при възникване на събитие породено от потребителското поведение е нужно да се направи заявка към сървъра, където това събитие да бъде обработено.

## 2.3 Предимства на новото решение

Предложеното тук решение преодолява вече споменатите проблеми, съобразявайки се с вече доста по-високите изисквания на потребителите относно възможностите и удобството, които информационните системи предлагат в наши дни. То е уеб-базирано, което автоматично премахва ограниченията свързани с допълнителна инсталация и в същото време осигурява платформена независимост. Базирайки се на съвременни технологии и подходи (библиотеки като Knockout,

RequirJS и подход за SPA приложение), то осигурява удобен интерфейс, богата функционалност и бърза реакция на приложението. Използвайки наложени тенденции за визуално представяне и организиране на информация в уеб-формат (Twitter Bootstrap) е постигната унифицираност и предвидимост на изобразяваните данни.

От технологична гледна точка, дизайнът на приложението и архитектурата съобразена с предметната област, съчетани с предоставените възможности от използваните платформи (Durandal) предоставят възможност за преизползваемост, лесна разширяемост и оптимизирана поддръжка на приложението като софтуер. Фактът, че то е изградено основно на базата на технологии, които са базирани на функционалност изпълнявана директно на клиента (браузъра) и са платформено независимост (в крайна сметка, изпълнимият код представлява файлове, които се зареждат на клиента и там се изпълняват), прави интегрирането на приложението в други проекти лесно и са оптимизирани ограниченията, свързани с тази интеграция. Поради асинхронния характер на зареждане както на файловете предоставящи самите обекти от данни и тяхната функционалност, така и на файловете представящи тяхното изобразяване, се постига усещане за намалено изчакване при работа със системата за крайния потребител. По този начин се постига предоставяне на богата и интерактивна функционалност характерна за приложения, които са офлайн базирани и в същото време се избягва всякакъв вид платформена зависимост и необходимост от допълнителна инсталация.

## 3 Използвани технологии

### 3.1 Структура и представяне на обекти

#### 3.1.1 Стратегия за разработка на клиентски приложения – SPA

Стратегията, използвана за реализиране на решението, описано в настоящата теза е подход, който се налага все повече при уеб-базираните приложения – SPA (Single Page Application). Затова най-напред ще бъде направен кратък обзор на тази идеология.

С развитието на ИТ и налагащото се разнообразие от мобилни устройства на пазара, с подобряването на свързаността и достъпа до интернет в глобален аспект се налага тенденция за използване все повече на уеб-базирани приложения. Изискванията и очакванията на потребителите също все повече се завишават, именно поради голямата конкуренция и разнообразие от решения достъпни в интернет пространството. Интерактивност, бърза реакция, разнообразие от предоставяни функционалности и то по възможно по-бърз и елегантен начин, максимална достъпност и платформена независимост са някои от очакванията, които налагат радикална промяна при подхода за предоставяне на решения в уеб-пространството. Така се заражда и се популяризира стратегията за SPA и по-конкретно реализирането ѝ базирано на JavaScript (тъй като Java applets, Flash/Flex също могат да се разглеждат като имплементации на същата тази идеология).

*„SPA е приложение, което се зарежда в браузъра и което не презарежда страницата по време на работа” [12].* – е една дефиниция за SPA, която е формулирана в *„Single Page Web Applications: JavaScript end-to-end”*. Там са представени предимствата на този подход и неговата JavaScript реализация – именно посрещане на изброените по-горе нужди на потребителите. Описани са също в детайли ролите, които изпълняват браузър и сървър в процеса на комуникация, както и самото осъществяване на тази комуникация в контекста на изграждането на приложения базирани на SPA стратегията.

Друга формулировка и анализ на самата дефиниция на SPA като термин е представена от John Papa в *„SPA and the Single Page Myth” [11]*. В тази статия се изяснява какво стои зад наименованието на подхода за „Приложение от Една Страница”, а именно идеята, че в браузъра се зарежда тази Една Страница, така нареченият *„shell”*, която представлява общата опаковка на приложението. Всички останали допълнителни елементи на приложението се зареждат асинхронно и то при възникване на нужда за тяхното използване. В тази статия също отново се описват ролите на клиента и сървъра в неговия контекст и се прави друга препратка към цитат на Ward Bell за описанието на тези роли:

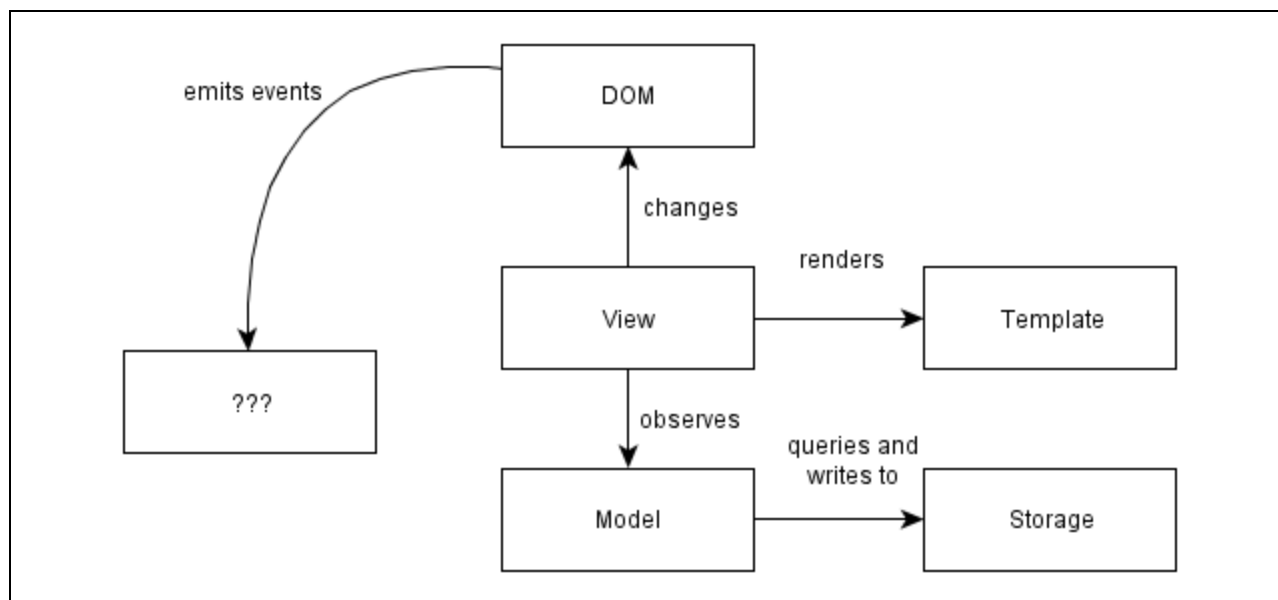
*„Сървърът не играе никаква роля в определянето на това какво приложението прави със SPA. Това е работа на клиента. Клиентът е този, който композира/съставя страници на приложението от HTML шаблони (HTML фрагменти) и данни, като и двете той сваля асинхронно при нужда”.*

По същество тази стратегия представя идеята да се минимизира трансфера между клиент и сървър и по този начин едно уеб-базирано приложение да бъде оптимално удобно за работа и усещането за забавяне при интеракция с него да бъде подобро. Или иначе казано, да се

постигне функционалност и удобство подобно на това при използване на офлайн (*desktop*) приложения, но комбинирано с платформената независимост и всеобща достъпност, които предоставят уеб-базираните приложения.

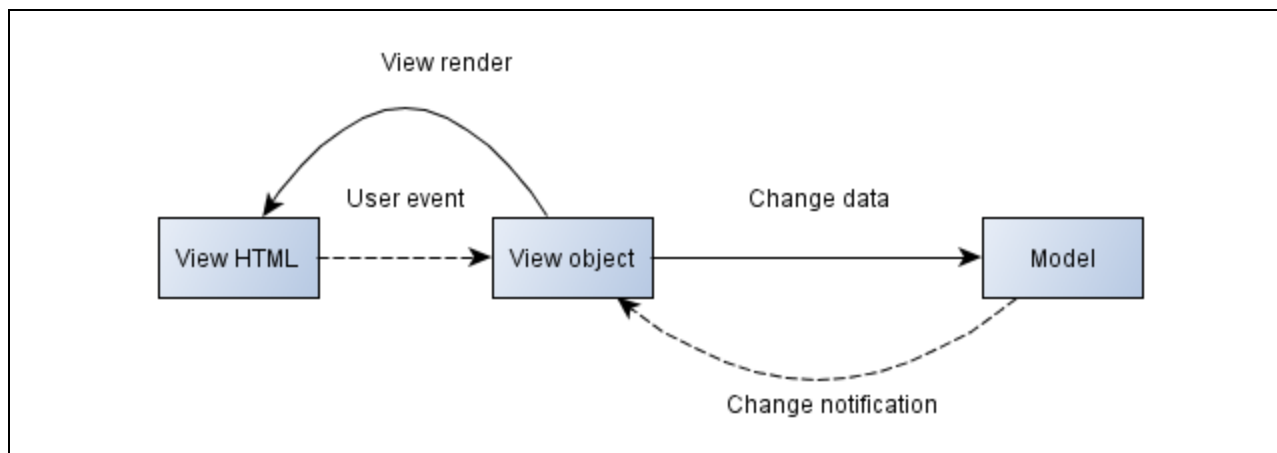
Тези цели се постигат чрез няколко основни принципа, които се спазват при прилагането на тази стратегия. На първо място е фактът, че имплементацията е базирана именно на JavaScript като средство за разработка на логиката на приложението. Това гарантира платформената независимост и премахва нуждата крайният потребител да трябва да инсталира допълнителен софтуер.

Основната идея, която е причината този подход да постига поставените цели е разделението на презентационната логика от представянето на данните. Именно тази идея е описана в детайли в „*Single page apps in depth*“, Mikito Takada [13]. Това разделение се постига чрез използването на шаблони (*templates*) и обекти (*models*) и чрез гарантирането на еднозначната им свързаност.



Фиг. 1 Структура на SPA

На Фиг. 1 е представена примерна схема на това взаимодействие. Моделът е обектът, който се грижи за представянето на логиката и функционалността за конкретен елемент от приложението. Моделът представлява JavaScript обект, който спазвайки добра практика за имплементация гарантира предоставяне на нужната функционалност, постигане на необходимото ниво на капсулация и модулност. По този начин, разпределяйки логиката в капсулирани обекти, се постига възможност за поддръжка и тестване. Тези обекти, се зареждат асинхронно при необходимост и се изпълняват на клиента. Шаблонът (*View*) отговаря за визуалното представяне на съответния модел. Двустранната връзка между *View* и *Model* се осъществява чрез правило на свързване (*binding*), което се предоставя от някоя от библиотеките имплементиращи такава функционалност точно с цел използването ѝ за изграждане на SPA приложения (например Knockout; такъв предоставя и платформата Angularjs). Тази връзка се грижи за непрекъснатия синхрон между обекта и неговото представяне, т.е. при възникване на промени в обекта автоматично те да се отразят визуално. Именно тази синхронизация предоставя възможност за добра интеракция на потребителя с приложението. Тя е изобразена на Фиг. 2.



**Фиг. 2 Взаимодействие между обектите в SPA**

В заключение на краткото описание на подхода за SPA трябва да се спомене, че това е стратегия, чиято конкретна имплементация може да бъде адаптирана към нуждите и изискванията за конкретните приложения. Този подход става все по-популярен и предпочитан за използване при реализация на уеб-базирани приложения особено, когато те предоставят сложна функционалност и се цели добро потребителско усещане (*User experience*). В този контекст за разработени и разпространени платформи (*frameworks*), които предоставят готова реализация на принципите описани в тази точка и чието прилагане дава добра основа за изграждане на приложения със сложна логика и функционалност и в същото време с оптимизирано бързодействие и интерактивност. Такива платформи са например Durandal, Angularjs и други. Поради характера си (реализиране на клиента и следователно самата имплементация е в контекста на файловата система), този вид приложения нямат ограничения относно използваната технологията за реализиране на логиката от страна на сървъра. В случая на използване на ASP.NET (MVC/Web Forms), с цел улеснение са интегрирани готови шаблони с включени заготовки за SPA приложения със средата за разработка (Visual Studio 2012). Тази методология може да се нарече млада, но се развива с бързи темпове и интересът към нея нараства все повече.

В разглежданото тук решение като платформа за SPA приложение е използвана Durandal.

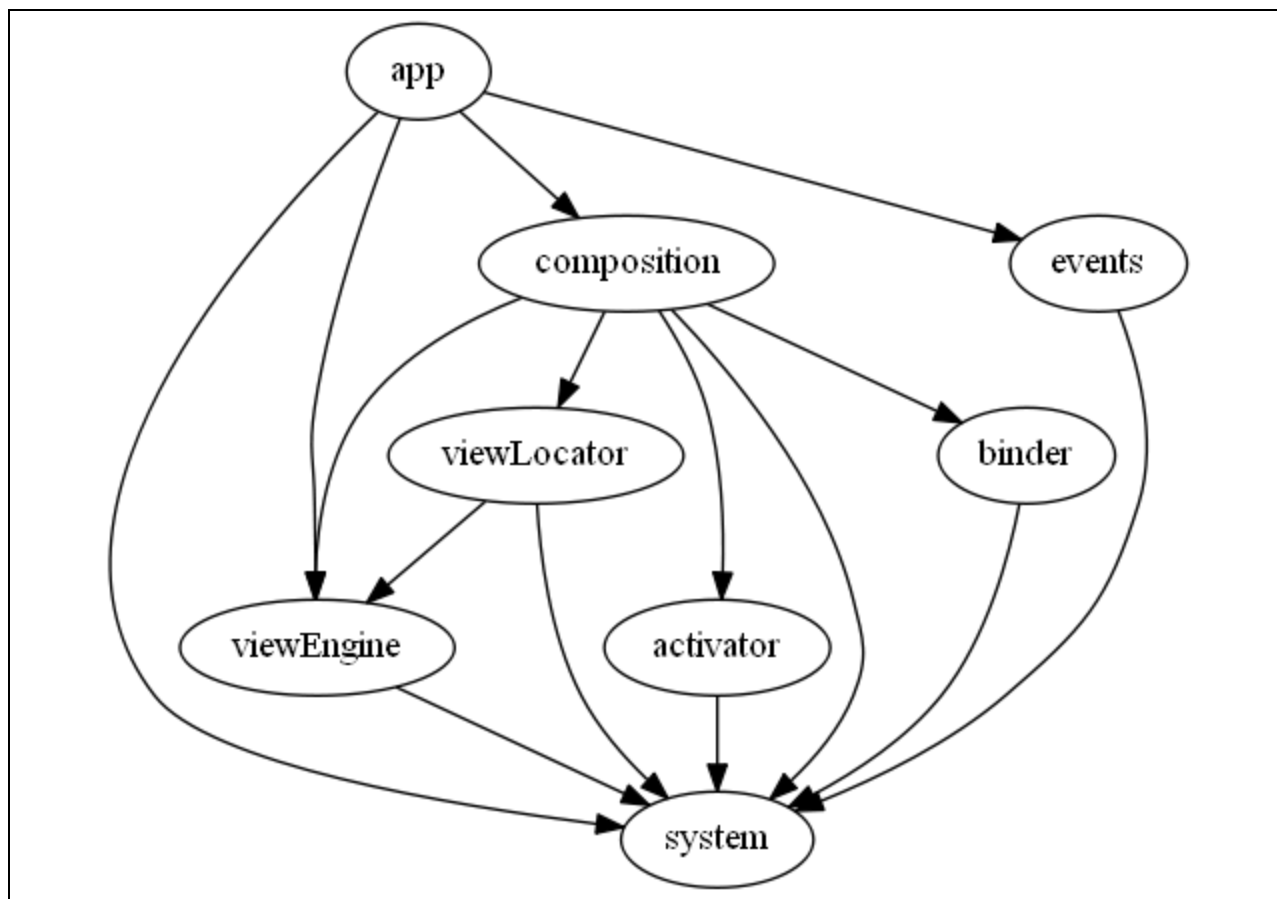
### **3.1.2 Платформа за разработка на клиентски приложения – Durandal**

Тя е малка JavaScript платформа, която е създадена с цел изграждане на SPA приложения по лесен и елегантен начин. Тя е базирана на библиотеки като Knockout, RequireJS и JQuery.

Благодарение на възможностите за композиране (*composition*) на обекти и техните изгледи (*view*), асинхронния характер (базирана на RequireJS) тя поставя основа за изграждане на приложение, което отговаря на изискванията на SPA. Самата Durandal също представлява набор от асинхронно дефинирани модули. Връзките между основните такива са изобразени на Фиг. 3.

[19]





Фиг. 3 Структура на Durandal

Използването на тази платформа при изграждане на SPA приложение освен, че не налага ограничения за избор на технологии за реализиране на сървърната част на приложението, тя не ограничава и избора на методологията за представяне на обектите в частта на клиента. Предимствата произлизащи от използването на библиотеки като Knockout и RequireJS ще бъдат разгледани в повече детайли в следващите точки. Дефинирането на конкретни конвенции и предоставянето на програмен интерфейс (API), който дава достъп и възможност за специфична имплементация в различни етапи от процеса на изпълнение и композиране на отделните части на едно приложение са основни предимства на използването на Durandal. Тук ще бъде обърнато повече внимание на възможностите за композиране на елементите на приложението, които са обособени като отделни модули. След като приложението е структурирано като набор от модули представящи конкретни обекти и логически единици е нужно те да бъдат свързани и да бъде описано взаимодействието помежду им. Тази необходимост се отнася както за самите логически обекти, така и за елементите отговарящи за тяхната визуализация.

Композирането на обектите, спрямо техните зависимости един от друг се осъществява чрез функционалността предоставена от RequireJS за управление на асинхронно зареждане на модули.

Композирането на визуалните елементи се осъществява чрез правилото за свързване (*binding*) „*compose*“, което като параметри приема наименованието на обекта, който ще бъде изобразяван (*model*) и наименованието на визуализиращия елемент (*view*). Отговорник за локализирането и съпоставянето на тези елементи е *viewLocator*. Предоставя се възможност за динамично задаване на тези зависимости (когато за *model* се зададе проследяема променлива – *observable*).

Друга функционалност, която се предоставя е поддръжката на области (*areas*) при организирането визуализиращите елементи. Така за конкретен модул може да се предоставят повече от един визуализиращи елементи в зависимост от режима, в който трябва да бъде достъпен съответния модул. Например могат да се дефинират области за Визуализация и за Редакция и конкретен модул да се изобразява в единия или другия режим в зависимост от логиката на приложението.

Други предимства и функционалности, които предоставя използването на тази платформа като например управление на навигацията, SEO, локализация и други няма да бъдат разглеждани в детайли.

### 3.1.3 Библиотека за разработка на клиентски приложения – Knockout

За да бъде изобщо възможно осъществяването на всичко описано в предходните точки от ключово значение е по лесен начин да се дефинира и синхронизира взаимодействието между модула представящ логическия обект и неговата визуализация. Трябва да бъде поддържана по лесен начин двустранната връзка между JavaScript обекта и неговото визуално изобразяване. Именно тази основополагаща роля изпълнява Knockout.

Тя се основава на три основни принципа.

- 1) Проследяеми променливи (*observables*) и проследяване на зависимостите.
- 2) Декларативно указване на свързаност (*declarative bindings*).
- 3) Шаблони (*templates*).

Чрез проследяемите променливи се постига синхронизация на модела съдържащ данните за представяне (*ViewModel*) с потребителския интерфейс (*UI*), т.е. при възникване промяна модела тя автоматично се отразява визуално в интерфейса. Дефинирането на такъв обект става аналогично на дефиницията на обикновен JavaScript обект, с малки специфики, като например:

```
var myViewModel = {  
    personName: ko.observable('Bob'),  
    personAge: ko.observable(123)  
};
```

Указването на това как визуално трябва да бъде представен моделът, т.е. на връзката между отделната променлива и нейното визуално представяне става декларативно чрез специалния „*data-bind*” атрибут. В контекста на предходния пример:

```
<span data-bind="text: personName" ></span>
```

Библиотеката предоставя набор от дефинирани правила за свързаност (*bindingHandlers*), т.е. конкретен начин на обработка на стойността при визуализиране или прочитане от *UI*. Те могат да бъдат използвани в по-голямата част от стандартни сценарии, които се срещат при повечето уеб приложения. Друго голямо предимство е възможността за разширяемост и предефиниране на вече предоставена функционалност. Всеки, който използва този продукт, има опцията да дефинира собствени правила или да предефинира стандартните, които се предоставят. След като са дефинирани всички зависимости и правила за дадения обект (*ViewModel*), те се прилагат чрез извикване на конкретна функция:

```
ko.applyBindings(myViewModel);
```

Третият принцип е в основата на възможността за постигане на модулно разделение на визуализиращите елементи, тяхното обособяване като отделни градивни единици на интерфейса и преизползването им. Неговото голямо значение произлиза от предоставянето на възможност да се реализира логика на представянето (*presentation logic*). Тази възможност е и в основата на предоставената от Durandal функционалност за композиране на визуализиращи елементи. Тя се постига благодарение на използването на елементи за управление на представянето (*control-flow elements*) като „*foreach*“, „*if*“, „*ifnot*“, „*with*“. Тези елементи предоставят възможност за влагане на логика на изобразяване на елементи, но сами по себе си нямат визуална стойност, т.е. не се изобразяват визуално като отделни елементи. Те могат да бъдат използвани със следния синтаксис:

```
<!-- ko control-flow element: condition -->  
<!-- /ko -->
```

По този начин в крайно генерирания HTML не се появяват излишни елементи (*div*, *span* и др.), чиято единствена функция е да дефинират определено правило за свързване (*binding*).

Благодарение на тези основни функционалности се постига ясно разделение на логиката за обработка на данните и тяхното визуализиране. Приложенията, използващи тези средства имат изчистена структура, която може да бъде лесно поддържана и има опции за преизползване на програмни елементи (както JavaScript обекти, така и HTML фрагменти). В същото време поради дефинираната тясна свързаност се постига много добро ниво на интерактивност и бърза реакция на приложението при работа с него. Библиотеката е със свободен достъп и предоставя изчерпателна и добре организирана документация и много примери за използване. Също голямо предимство е, че е широко разпространена и поради тази причина източниците на информация и възможностите за споделяне на опит за работа с нея са много и лесно достъпни.

### 3.1.4 Библиотека асинхронно зареждане на ресурси – RequireJS

До момента бяха представени принципите на SPA базирания приложения, беше разгледана основата предоставена от Durandal и предимствата, които тя предоставя като динамичното композиране на отделните модули – логически и визуализиращи и беше представено как се постига динамичната синхронизация между тях. И така става ясно, че основното предимство на този подход е именно разбиването на логиката на приложението на модули и възможността да се поддържат динамични връзки между тези модули. Тъй като става въпрос за функционалност реализирана на клиента, трябва да се обърне внимание на количеството ресурси, които се зареждат за изпълнение и самия процес на зареждане. Важно е да се зареждат само необходимите ресурси и това да се случва асинхронно с цел подобряване на производителността на приложението. RequireJS е библиотека, която предоставя такива възможности. Тя отговаря за дефинирането на връзки между отделни обособени единици, техните зависимости и за тяхното зареждане на клиента само в случай, че те са реферирани. Този подход се нарича AMD (Asynchronous Module Definition).

За да бъде възможно създаването и поддържането на такава структура от зависимости между отделните модули е необходимо всеки модул да бъде дефиниран чрез уникален идентификатор (*moduleID*) и да му бъде указано от кои други модули зависи той, т.е. да бъдат дефинирани неговите зависимости. Синтаксисът, който библиотеката предоставя за дефиниране на тези дефиниции е следният:

```
define('myModule', ['dependency1', ' dependency2'], function (dep1, dep2) {  
    //Define the module value by returning a value.  
    return function () {  
        ...  
        dep1();  
        dep2();  
        ...  
    };  
});
```

По този начин новият модул е дефиниран с идентификатор „*myModule*” и за него е указано, че зависи от модули с идентификатори „*dependency1*” и „*dependency2*”. Следователно при зареждане на „*myModule*”, автоматично се зареждат и тези, от които зависи той.

Тези модули са реализирани като JavaScript обекти, които се зареждат асинхронно от файлова структура на сървъра. Поради тази причина организацията им трябва да следва някои конвенции или да се дефинират допълнителни конфигурационни правила, за това къде физически се намира обектът, който библиотеката трябва да извлече. RequireJS зарежда ресурсите чрез относителен път спрямо местоположението дефинирано със стойност за „*baseUrl*”. Спрямо конвенцията по подразбиране, това е пътят до скрипта, който стартира асинхронното зареждане на ресурси. Стартирането на процеса по анализ на връзките между ресурсите и тяхното извличане се започва при зареждане на скрипта „*require.js*”. Чрез специален атрибут „*data-main*” се указва пътят до файла, който отговаря за допълнителни конкретни конфигурации на библиотеката в случай, че не се използват стойностите по подразбиране (ако структурата на приложението се различава от тази, описана като изисквана за спазване на конвенциите).

```
<script src="libs/require.js" data-main="js/app.js"></script>
```

Там могат да се зададат стойности за път „*baseUrl*” и да се дефинират допълнителни относителни спрямо него пътища (*paths*). Например в „*app.js*”, рефериран в горния пример, би имало следните дефиниции:

```
requirejs.config({  
    //By default load any module IDs from js/lib  
    baseUrl: 'js/lib',  
    //except, if the module ID starts with "app",  
    //load it from the js/app directory. paths  
    //config is relative to the baseUrl, and  
    //never includes a ".js" extension since  
    //the paths config could be for a directory.  
    paths: {  
        app: '../app'    }  
});
```

```
});  
}
```

Има няколко изключения при анализ на дефинираните модули. В някои случаи не се вземат под внимание конфигурациите (по подразбиране или изрично зададени) за „*baseURL*” и относителните пътища спрямо него. Такива са случаите, в които даден модул е дефиниран с идентификатор, което отговаря на някое от следните условия:

- Завършва с „*.js*”
- Започва с „*/*”
- Съдържа *URL* протокол като например „*http:*” или „*https:*”

Тогава се приема, че съответният модул е дефиниран с абсолютния път до него и RequireJS го търси спрямо тази дефиниция без да взема предвид други конфигурации.

Разбира се, както вече беше споменато, тази идеология е възможна само при условието, че кода предоставящ логическата имплементация на приложението е организиран по структуриран начин и логическите единици в него са отделени една от друга, т.е. е модулно ориентиран. Говорейки в контекста на JavaScript реализация това означава да бъде постигнати разделение на кода в обособени единици, които не са просто част от глобалния обхват (*global scope*), а предоставят някакво ниво на капсулация. То се осъществява чрез спазването на добри практики (*patterns*), които подхождат по различен начин в зависимост от нуждите на приложението и предоставят прилагане на принципите на обектно ориентираното програмиране в JavaScript. В рамките на текущото решение е избрана добрата практика Revealing Prototype Pattern за дефиниране на обектите, като най-подходящ подход за целите на проекта.

### 3.1.5 Добра практика Revealing Prototype Pattern

Този подход е начин за представяне на обектите, чрез прототипично наследяване. Неговите основни предимства са, че той предоставя възможност за капсулация, модулно разделение на обектите и преизползване на дефинициите на функции отнасящи се за даден тип обекти. Образно казано, той предоставя възможност за създаване на обекти по образец [14]. Примерна реализация на този шаблон изглежда по следния начин:

```
function () {  
    var ObjectName = function () {  
        this.property1 = {};  
        this.property2 = {};  
    }  
    ObjectName.prototype = function () {  
        var publicFunctionDefinition = function () {  
            ...  
        },  
        privateFunctionDefinition = function () {  
            ...  
        };  
        return {  
            publicFunction: publicFunctionDefinition  
        }  
    }();  
    return ObjectName;  
}
```

```
}
```

Накратко представянето на един обект, спрямо този подход се състои от три основни части.

Първата част е дефинирането на конструктор за обекта. Конструкторът е функция, в която са изброени полетата (*properties*), които съдържа създавания обект. Обикновено наименованието ѝ започва с главна буква (общоприета конвенция). Инстанцирането на такъв обект се осъществява чрез извикване на тази функция (конструктор) със запазената дума „*new*”. Полетата дефинирани в конструктора са представени като собствени полета – дефинирани чрез запазената дума „*this*”. По този начин при създаване на нова инстанция на обект от този тип, се създават нов обект, дефиниращ неговите полета, т.е. всеки обект съдържа собствени стойности за дефинираните му полета.

Втората част характерна за този подход е частта, описваща методите за този тип обект. Техните дефиниции са заложили в прототип (*prototype*) частта на конструктора на обекта, описан по-горе. Прототипът е характерна функционалност за всеки обект/функция в JavaScript, която може да се използва за разширяване на дефиницията му. Добрата практика се възползва от тази вградена функционалност на езика. Така организирани дефинициите на методите за типа обект се гарантира, че те се създават само веднъж за този тип независимо от броя на инстанциите, които са инициализирани. Всяко извикване на метод за конкретна инстанция се обръща към обща дефиниция на функцията, но със собствени стойности като параметри. Това е много удобен начин за оптимизиране на ресурси и намаляване на използваната памет при изпълнение на приложението, особено в случаи, когато се налага инстанцирането на голям брой обекти.

Прототипът на обект е публично достъпна функционалност. Важен момент в по-горе изложения пример е начинът, по който е дефиниран прототипа. Той представлява самоизпълняваща се функция (*immediate function*), в която са дефинирани методите. Чрез опаковането на дефинициите в обхвата на функцията се постига капсулация, която позволява скриване на функционалност, която не трябва да бъде видима от външни елементи. Методите дефинирани в рамките на тази функция не са видими отвън. Тези, които трябва да бъдат публично достъпни са изброени накрая в „*return*” секцията.

## 3.2 Управление на данни

Макар по-голямата част от логиката на приложението да е имплементирана за изпълнение в клиентската част, неизбежно се стига до момент, в който е нужна комуникация със сървърната част – за извличане или изпращане на данни. Освен самия процес на комуникацията, много важен елемент от приложението е и начинът на обработка на въпросните данни, предмет на обмен от двете страни.

За управление на данните в разглежданото приложение е използвана библиотеката Breeze. За нея е характерно, че е насочена за подпомагане процеса на обработка на данни за приложения ориентирани към богата функционалност изпълнявана на клиента.

Основни предимства на тази технология, които са използвани в приложението са:

- Възможност за работа с обекти на клиента, копие на обектите дефинирани в сървърната част. Те са обвързани с потребителския интерфейс и по този начин се поддържа синхронизация.
- Възможност за дефиниране на заявки на клиента (JavaScript), които имат опции за филтриране, сортиране и др. Тъй като се поддържа Open Data Protocol (OData) [16] стандарта, автоматично са предоставени и възможности за извличане на свързани обекти (*expand related entities*) и за проекция – избиране само на някои от полетата от извлечените обекти. [17]
- Възможност за управление на кеша. Преди да се изпрати заявка за извличане на данни от сървъра може да се провери дали тези данни вече са заредени в паметта на клиента (кеширани) и ако да – да се използват те. Така се намалява броят на изпратените заявки и се подобрява производителността на приложението.

Управлението на данните и заявките за работа с тях се осъществява чрез обекта `EntityManager`. Той се използва както за осъществяване на връзката със сървърната част за обмен на данни, така и за изпълнение на заявки към заредените вече локално на клиента данни (кеша на `EntityManager`). Изпълнението на дефинираните заявки, се извършва асинхронно.

Друга важна характеристика на тази технология са метаданните (*metadata*). Те съдържат описание на обектите, представящи данните на клиента и техните полета и връзките помежду им. По този начин веднъж инициализирани тези метаданни, чрез тях приложението разполага с дефиниран модел на данните без да се налага той да бъде описван ръчно. Тази инициализация се извършва чрез заявка към сървъра.

### 3.3 Работа с XML съдържание

До момента бяха описани набора от библиотеки и други методологии използвани за реализирането на възможността за работа с техническото представяне на електронни структурирани документи. Друга използвана технология е тази, отговорна за двустранното конвертиране на това представяне (JavaScript обекти) и искания XML формат, представляващ структурирания документ.

За тази функционалност отговаря библиотеката `x2js`. Тя е малка (10KB) и няма зависимости от други библиотеки. Тя предоставя лесен за използване програмен интерфейс (API) за работа с JavaScript обекти и с XML документи. Функциите, които са използвани от това приложение са:

- `<instance>.xml_str2json` – Конвертира XML съдържание представено в текстов формат в JSON обект;
- `<instance>.json2xml_str` – Конвертира JSON обект в XML съдържание представено в текстов формат.

Използват се точно този вид конвертиране, тъй като двата вида обекти всъщност са представени в текстов формат в момента на тяхното съпоставяне. Повече за използването на библиотеката и нейното интегриране в контекста на приложението ще бъде разгледано в следващата точка в хода на описанието на самата имплементация на софтуерния продукт.





## 4 Представяне на решението

В тази глава ще бъде представено решението, предоставящо унифициран метод за обработка на електронни структурирани документи. Ще бъде разгледано в детайли реализирането на функционалностите за работа с един конкретен електронен структуриран документ.

Като примерна реализация ще бъде разгледан документът „Заявление за издаване на удостоверение за постоянен адрес”.

Следвайки подхода за SPA приложение, както беше описано в точката за *„Използвани технологии”*, приложението предоставя имплементация на решението базирайки се основно на JavaScript обекти, представящи обектите от реалния свят (ИО) и HTML файлове, представящи тяхната визуализация. Поради характера на този подход, решението е базирано на набор от библиотеки, платформи и добри практики, за да бъде достатъчно надеждно, предоставящо добра архитектура и възможност за поддръжка, както и да предоставя оптимална възможност за достъпност (да бъде използваемо от възможно по-голям набор от браузъри). С оглед на тези изисквания и характеристики, основната функционалност е разделена в три логически (и физически) части – функционалността на самото приложение (app), набор от библиотеки, подпомагащи имплементацията (libs) и тестове (tests). Тези разделения, разбира се, са част от уеб проект – в конкретния случай ASP.NET MVC Web Application, поради което тази организация се намира в рамките на стандартната папка Scripts на приложението.

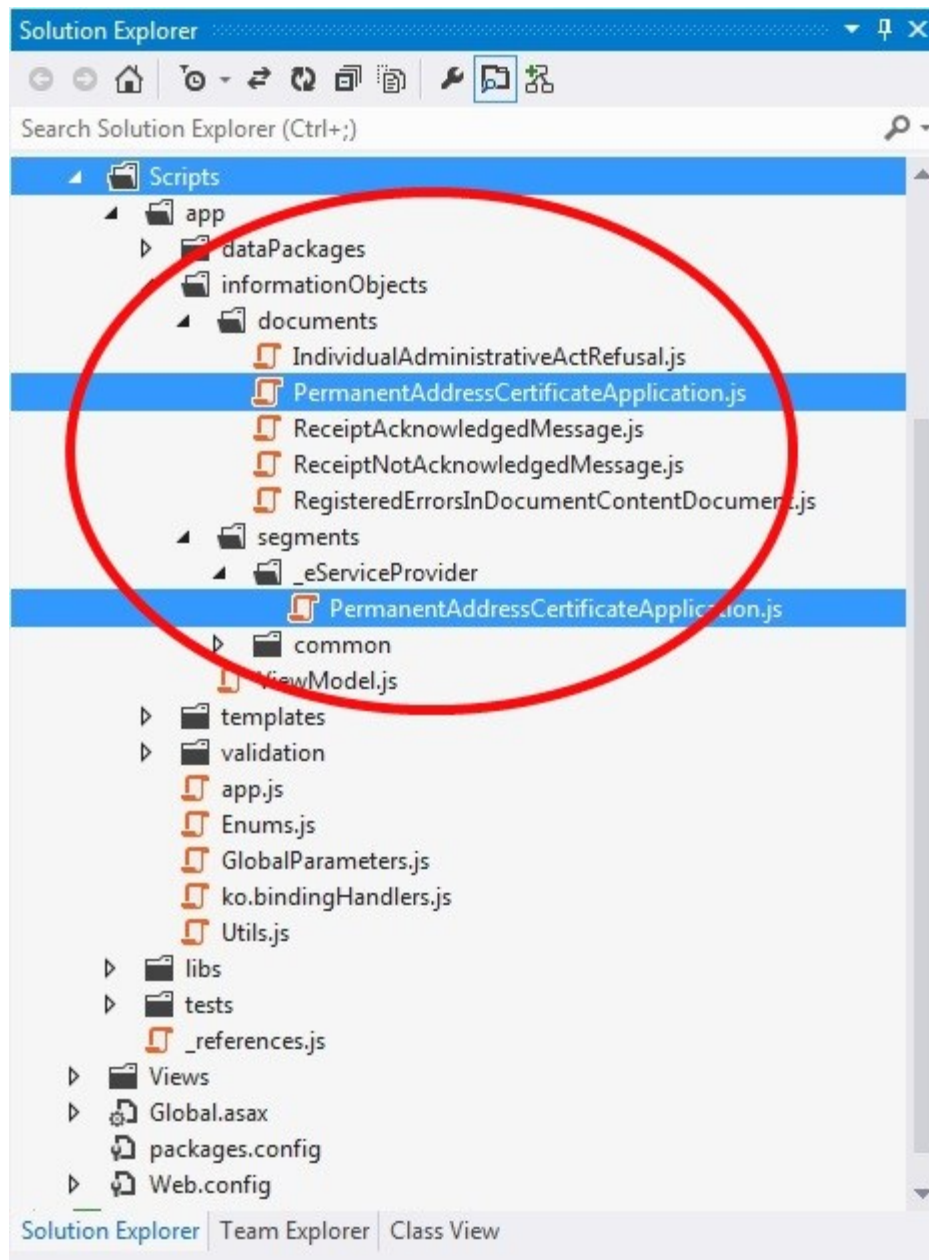
Както вече беше споменато, основната функционалност за имплементиране на JavaScript обектите, представящи ИО и тяхната визуализация, са базирани на изискванията наложени от използването на платформата Durandal. Следвайки конвенциите по подразбиране, които са заложили при използването на тази технология е необходимо архитектурата на приложението да следва някои принципи на организация.

### 4.1 Структура на приложението

#### 4.1.1 Информационни обекти

Информационните обекти вписани в РОС са представени като JavaScript обекти, които са организирани в папката informationObjects.

- Документи – за всеки документ, дефиниран в РОС е дефиниран по един обект с метаданни, необходима за генерирането на XML документ, отговарящ на изискванията описани в регистъра. В този обект също се пази референцията към основния сегмент, описващ неговата функционалност (също в съответствие с вписания в РОС сегмент).
- Сегменти
  - В отделни папки са разделени съответно сегментите, които са специфични за конкретния доставчик на електронни услуги;
  - В отделна папка са отделени общите сегменти и стойности, които се използват от други сегменти;
  - На по-горно ниво е обособен главният обект (ViewModel), отговорен за основната функционалност свързана с работа с електронен документ;



Фиг. 4 Структура на обектите на приложението

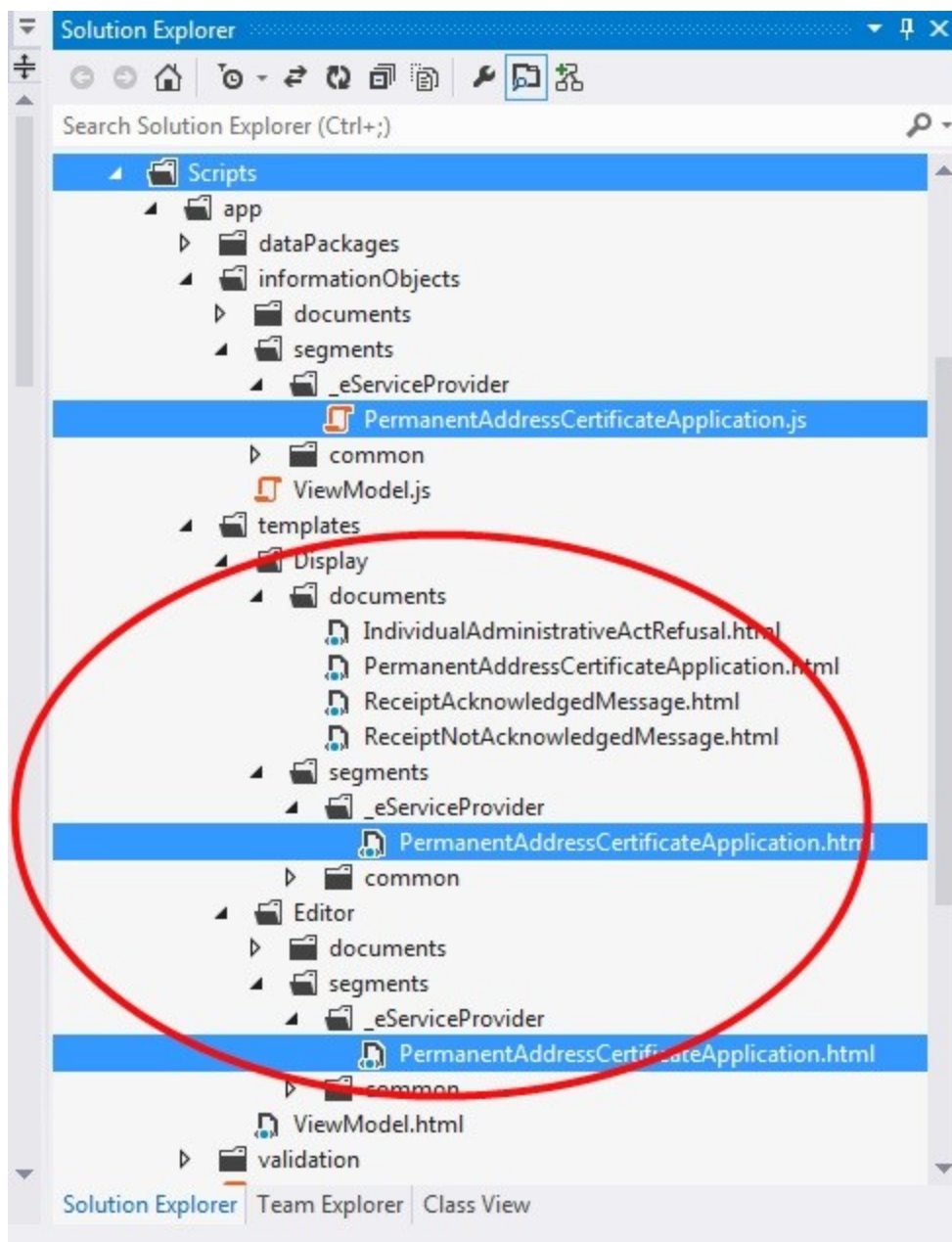
#### 4.1.2 Шаблони

В отделна папка (*templates*) са организирани файловете за визуално представяне на дефинираните обекти.

- За визуализиране (*Display*) – в тази секция се намират шаблоните, отговарящи за визуализирането на обектите (сегменти и стойности) спрямо изискванията за визуализация на съответните ИО описани в РОС;
- За редактиране (*Editor*) – в тази секция се намират шаблоните отговарящи за обработка на обектите (сегменти и стойности) спрямо изискванията за обработка на съответните ИО описани в РОС;

- Основен шаблон за ViewModel обекта, в който е опакована основната функционалност.

Забележка: Всяко от подразделенията за шаблони (за визуализиране и за редактиране) съдържа в себе си структура, отговаряща на структурата дефинирана за самите обекти.



Фиг. 5 Структура на изгледите в приложението

### 4.1.3 Конфигурация

#### 4.1.3.1 Настройки на библиотеки

Организирането в огледална структура на йерархията на JavaScript обектите и тази на шаблоните за визуализиране и редактиране е породено от използването на възможността за

съпоставяне на обект и неговия шаблон за визуализация (Durandal Composition), предоставена от Durandal като стандартна функционалност. Разбира се, поради спецификата на предметната област и конкретната реализация е необходима и допълнителна конфигурация. Конфигурирането на това съответствие е дефинирано в „app.js” файла, където се извършват всички инициализиращи и конфигурируеми параметри, необходими за функционирането на приложението:

```
var viewmodels = 'informationObjects',
    views = 'templates';

app.start().then(function () {
    viewLocator.useConvention(viewmodels, views);
    app.setRoot(v, 'entrance');
    ...
});
```

Тук се задават относителни пътища за зареждане на библиотеки и файлове с RequireJS, декларира се уникални псевдоними за вече заредени библиотеки, задават се настройки за библиотеката, отговаряща за валидацията на обектите knockout.validation. Тези настройки се извършват при инициализация. Това е първият изпълним файл, който стартира създаването на главния елемент (ViewModel) и оттам на цялата функционалност за работа с даден електронен структуриран документ. В следващите глави на изложението ще бъде разгледан процесът на изпълнение в повече детайли.

#### **4.1.3.2 Глобални изброени стойности**

Във файла „Enums.js” се намират всички изброени стойности, достъпни глобално от приложението. Тук са организирани всички стойности необходими за дефинираните електронни структурирани документи. Това включва всички термини, които участват като допустими стойности в състава на сегменти, които от своя страна участват в състава на електронния документ, обект на обработка. Те са представени като публично достъпни полета за обекта Enums, които от своя страна представляват масиви от обекти, съдържащи полета за ключ и наименование. За стойност на полето ключ е въведена стойността УРИ за съответния термин вписан в РОС, а за наименование – наименованието на термина отново от същия регистър.

Освен дефинираните термини в публичния регистър, тук се намират и някои изброени стойности, които имат помощни функции и са общи за цялото приложение. Те имат същата структура, като за стойност за ключ вместо строго дефинирано УРИ, се използва достатъчно говорещо и уникално служебно наименование.

#### **4.1.3.3 Глобални системни настройки**

Подобно предназначение има и файлът „GlobalParameters.js”. В него отново се съхраняват стойности и функционалности глобални за приложението, но те са по-скоро със системно значение. Тук, например, се прави проверка дали е нужно допълнително инсталиране на flash (в случаите, когато браузъра не поддържа FileAPI). Също тук се инстанцира обект отговарящ за

обръщенията към сървъра за извличане на данни (за стойностите от пакети от данни) – EntityManager.

#### **4.1.3.4 Помощни функционалности**

Друг файл (и в него дефиниран съответният JavaScript обект) е „*Utils.js*”. В него са дефинирани функциите отговарящи за конвертирането на JavaScript обект в резултатния XML документ, както и обратния сценарий. На него ще бъде отделено повече внимание в точката „*Работа с XML съдържание и JavaScript обекту*”.

#### **4.1.4 Валидация**

Поради характера на приложението, говорейки за валидация, се има предвид валидация извършвана на клиента (браузъра). За реализиране на валидация, като основа се ползва библиотеката knockout.validation. Тя предоставя тясна интеграция с библиотеката Knockout, която е основната градивна единица на идеологията, върху която е изградено приложението. Освен това, тя е и много лесно разширяема. Поради спецификата на изискванията за валидация на обектите, с които се борави в приложението (трябва да отговарят на изискванията описани в РОС), възможността за допълнителна имплементация на правила за валидация е от особена важност. Във файла ko.validation.custom.js се намират дефинициите на правилата за валидация, на които трябва да отговарят информационните обекти.

#### **4.1.5 Извличане на данни**

Въпреки че основната функционалност на приложението е изнесена за изпълнение на клиента, неизбежна е нуждата от комуникация и извличане на данни от сървъра. Пример за такова взаимодействие е извличане на стойности от Пакети от данни, като изброени стойности, дефинирани в „Регистъра на регистрите и данните”. Обръщенията за извличане на данни са отделени в отделен файл datacontext.js. Методът на извличане на данни и обръщения към сървъра ще бъде изяснен в повече детайли в главата „*Управление на данните*”.

### **4.2 Реализиране функционалността на приложението**

След като беше представена обобщена информация за структурата на приложението и ориентиловъчна информация за предназначението на всяка от секциите, следва да бъдат разгледани в повече детайли самата имплементация на описаната идея. Както вече беше споменато, за целите на представяне на идеологията, ще бъде разгледана една конкретна имплементация на електронен структуриран документ, неговото създаване, визуализиране на вече съществуващо съдържание, редактиране на съществуващо съдържание, както и валидиране на попълнените стойности (при попълване на стойности по време на работа с приложението и проверка за валидност на генерирания резултатен XML документ спрямо XML дефинициите, описани в РОС за всеки от съставните елементи на електронния документ).

Ще бъде описан подходът на представяне на всеки от описаните по-горе ИО във вид, позволяващ обработка в контекста на софтуерното приложение. Ще бъде описано прилагането на модела Prototype Revealing Pattern и организация на зависимостите между отделните модули чрез

RequireJS за целите на постигане на капсулация, организация на програмния код и оптимизация на производителността на приложението.

Ще бъде проследен процесът на двустранното взаимодействие между XML документ и JavaScript обект. Ще бъдат разгледани и методите за реализиране на други основни функционалности като валидация, извличане на данни, динамично определяне на вида документ.

#### 4.2.1 Представяне на ИО от тип Термин

Термините, които са реферирани от имплементацията на електронния структуриран документ са представени като JavaScript обект – литерал. В контекста на приложението термини се използват в два основни случая.

- В първия случай те представят изброените стойности, съдържащи се в ИО от тип номенклатура. Тъй като тези стойности са конкретни, вписани в РОС със съответния УРИ, те са представени като точно дефиниран и конкретен обект в приложението. Този обект е имплементиран като JavaScript обект – литерал, съдържащ в себе си две полета: *key* – съдържащо стойността на УРИ полето вписано в РОС, *name* – съдържащо стойността на Наименование полето вписано в РОС. Пример:

```
{  
  key: "0006-000083",  
  name: "Обикновена"  
}
```

- Във втория случай те представят обект, дефиниращ конкретен тип грешка, възникнала при изпълнение на валидацията на даден ИО спрямо изискванията описани за него. Поради тясната връзка с функционалността за осъществяване на валидация, този тип термини са дефинирани в частта на приложението, отговаряща за нея. Там те са представени чрез дефинирани конкретни съобщения за грешка, следващи конвенция на форматиране. По-конкретно, всяко дефинирано съобщение за грешка има следния формат: „<УРИ на термин>: <Наименование на термин>”, като за форматиране на полето за наименование са приложени „Указанията за обработка”, където са специфицирани такива в РОС. Така дефинираното съобщение, се трансформира в JavaScript обект, когато съответната грешка възникне и е необходимо да се създаде такъв обект, представящ съответния термин. Поради изискванията за прилагане на валидационни правила и записването на възникналите грешки, резултат от направената валидация, във формализиран формат се създава документ, съдържащ списък от ИО от тип Стойност, описващи тези грешки. Повече за този процес ще бъде разгледано в „*Валидиране на електронен структуриран документ*”.

#### 4.2.2 Представяне на ИО от тип Номенклатура

Номенклатурите като вид ИО са представени като поредица от термини. В приложението, тяхното представяне е аналогично. По-точно казано, номенклатурите представляват масиви от обекти. Тези обекти са именно литерали, представящи термини по начина, описан в предната

точка. Дефиницията на номенклатурите е реализирана чрез глобален обект – Enums, достъпен в контекста на цялото приложение. Самите стойности в него са изброени като конкретни и са съобразени с дефинираните в РОС обекти. Една примерна дефиниция е тази на номенклатурата „Номенклатура на видовете услуги, спрямо срока за предоставянето им“:

```
serviceTermTypes =
[
  {
    key: "0006-000083",
    name: "Обикновена"
  },
  {
    key: "0006-000084",
    name: "Бърза"
  },
  {
    key: "0006-000085",
    name: "Експресна"
  }
]
```

, където всеки литерал съдържащ се в дефиницията на този масив, всъщност дефинира термин по правилата описани в предходната точка.

#### 4.2.3 Представяне на ИО от тип Стойност

ИО от тип стойност са прости обекти. Те са представени като полета на съставни обекти (сегменти), които се инициализират за всяка инстанция на съответния обект представящ сегмента (в контекста на прилаганата добра практика Revealing Prototype Pattern). За целите на двупосочна трансформация на обекта, с който се работи в програмната част на приложението (JavaScript обектът) и резултатния XML документ (съобразени с функционалността предоставена от библиотеката x2js), са наложени някои конвенции, които улесняват процеса на разработка. Конкретно за обектите от тип Стойност общата конвенция е, че всяко поле представлящо дадения ИО, носи наименованието му такова, каквото е дефинирано в РОС, но с водеща малка буква (Например: ServiceTermType - serviceTermType). По този начин в общия случай генерираният (или съответно прочетеният) XML елемент ще съответства на очакваното наименование спрямо XML дефиницията (XSD) за тази стойност. Разбира се, има и изключения от общото правило, за които е необходимо изрично да се конфигурира съответствието между наименованията на елемента от XML документа и това на съответното поле, представлящо този елемент в контекста на програмния обект. Тази конфигурация се запазва в служебното поле „\_settings“, което се обработва по различен начин от останалите полета. Пример:

```
...
this._settings.options = {
  ...
  propertiesTitles: {
    egn: 'EGN',
    lnch: 'LNCh'
  }
};
```

```
this.egn = ko.observable();  
...
```

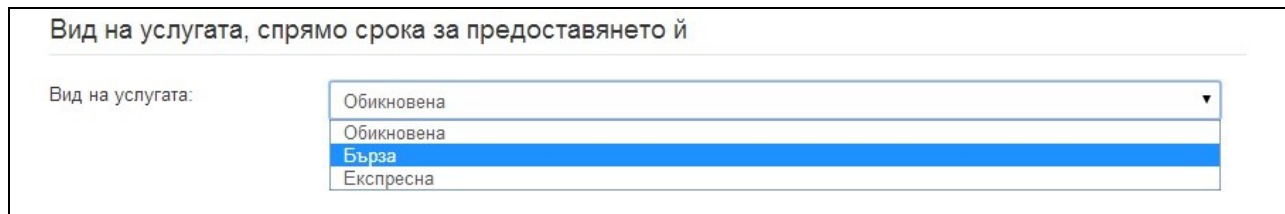
Особености за този тип обект възникват при изпълнението на условията за валидация и имплементирането на ограниченията, които са заложили в неговата XML дефиниция. Такова ограничение е например, когато за дадена стойност в XML дефиницията е заложило тя да бъде една от изброените стойности от дефинирана вече номенклатура. В този случай, изброените стойности за дадената номенклатура се представят като поле на полето, представляващо ИО от тип Стойност. Самото поле за Стойност съхранява избрания елемент от изброените стойности.

Пример: Реализирането на тази функционалност в конкретен случай – например за ИО ServiceTermType от тип стойност с УРИ: 0008-000143 е представено по следния начин:

```
this.serviceTermType = ko.observable();  
this.serviceTermType.serviceTermTypes = ko.observableArray(Enums.serviceTermTypes);  
this.serviceTermType.title = 'Вид на услугата, спрямо срока за предоставянето ѝ';  
this.serviceTermType.extend({  
    fieldIsFromEnum: {  
        field: this.serviceTermType,  
        nomenclatureTitle: 'Номенклатура на видовете услуги, спрямо срока за  
предоставянето им',  
        nomenclatureValues: Enums.serviceTermTypes  
    }  
});
```

, където Enums.serviceTermTypes съответства на дефиницията на номенклатура дадена като пример в предната точка (описание на „Представяне на ИО от тип Номенклатура“).

Визуално тази функционалност е представена като падащ списък с изброени стойности:



Вид на услугата, спрямо срока за предоставянето ѝ

Вид на услугата:

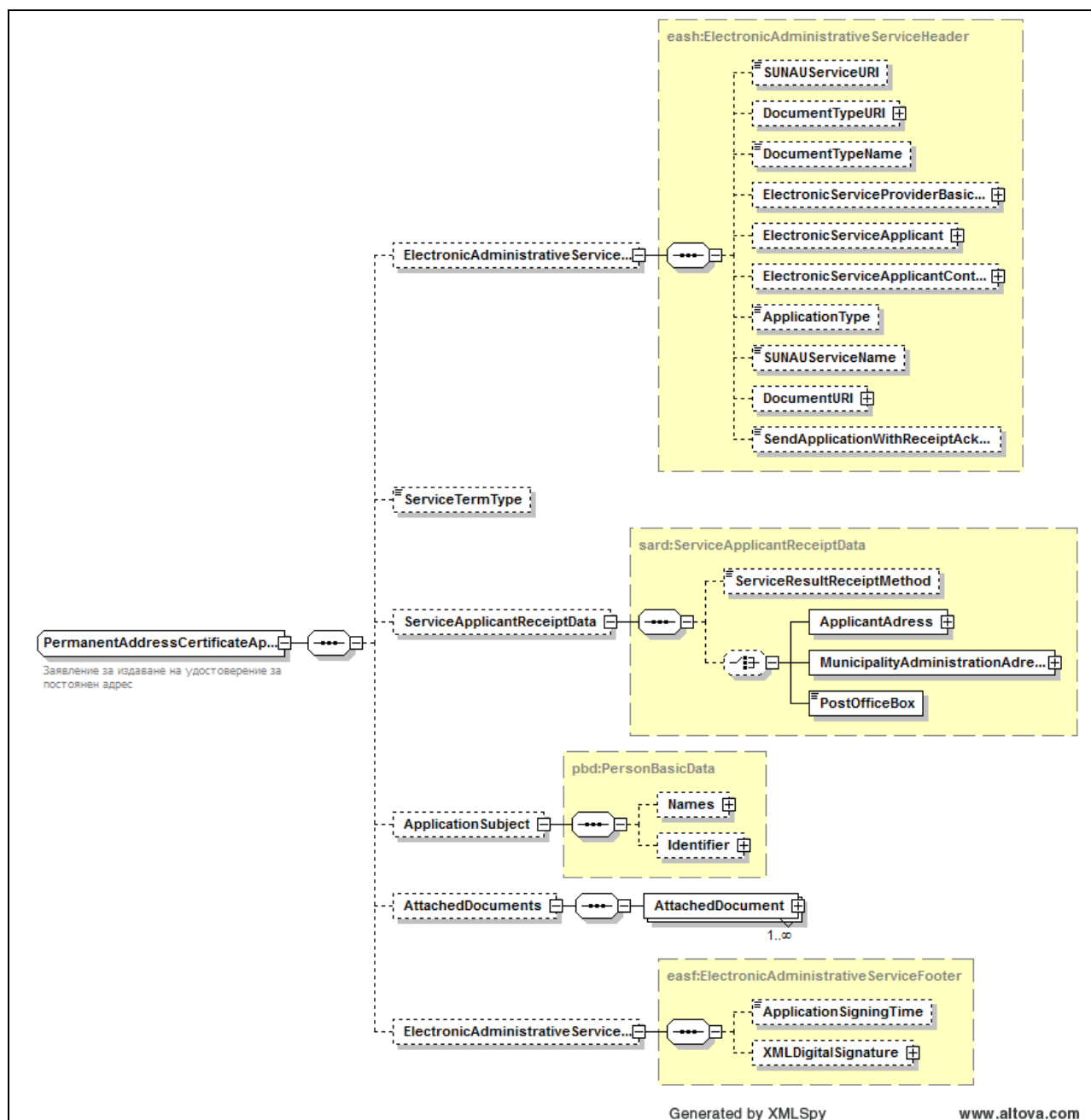
- Обикновена
- Обикновена
- Бърза**
- Експресна

Фиг. 6 Визуално представя на изброени стойности

#### 4.2.4 Представяне на ИО от тип Сегмент

ИО от тип Сегмент е представен като JavaScript обект и е основната програмна единица, представляваща имплементацията на нужната функционалност за работа с ИО и оттам с електронни структурирани документи. Сегментите в контекста на ИО са сложни обекти, които съдържат в себе си стойности или други сегменти. Именно този подход на представяне на информацията чрез обекти с определено ниво на влагане, предоставя възможност за преизползване на вече дефинирани обекти и следователно тяхното еднозначно представяне и разпознаване, независимо от контекста, в който те са реферирани. В конкретния случай на разглежданата имплементация сегментът „Заявление за издаване на удостоверение за постоянен адрес“ с УРИ: „0009-000146“ има следната структура (до второ ниво на влагане):





Фиг. 7 Йерархия от обекти описваща електронния документ

В приложение са добавени ресурси описващи изчерпателно цялостната йерархия от обекти. С цел по-голяма яснота тук тя е представена до второ ниво на вглъбване.

Подобна йерархия от обекти (модули) е изградена в рамките на софтуерното приложение, предмет на текущата теза.

Примерна реализация на обекта, съответстващ на този ИО е:

```
define('_eServiceProvider/PermanentAddressCertificateApplication',
  ['ko', 'jquery', 'GlobalParameters', 'Enums', 'Utils',
```

```

        'common/ElectronicServiceProviderBasicData',
        'common/EntityBasicData',
        'common/PersonBasicData',
        'common/ServiceApplicantReceiptData',
        'common/ElectronicAdministrativeServiceHeader',
        'common/RegisterObjectURI',
        'common/ElectronicAdministrativeServiceFooter',
        'common/AttachedDocuments'],
function (ko, $, gp, Enums, Utils,
    ElectronicServiceProviderBasicData,
    EntityBasicData,
    PersonBasicData,
    ServiceApplicantReceiptData,
    ElectronicAdministrativeServiceHeader,
    RegisterObjectURI,
    ElectronicAdministrativeServiceFooter,
    AttachedDocuments) {
    var PermanentAddressCertificateApplication = function () {
        this._settings = {};
        this._settings.sectionTitle = 'Заявление за издаване на удостоверение за
постоянен адрес';

        ...
        var localEHeader = new ElectronicAdministrativeServiceHeader();
        localEHeader.initElectronicServiceHeader(...);
        this.electronicAdministrativeServiceHeader = ko.observable(localEHeader);
        this.electronicAdministrativeServiceHeader.title = 'Водеща част на заявление за
предоставяне на електронна административна услуга';

        ...
        this.serviceTermType = ko.observable();
        this.serviceTermType.serviceTermTypes =
ko.observableArray(Enums.serviceTermTypes);
        this.serviceTermType.displayValue = ko.observable();
        this.serviceTermType.subscribe(this.initServiceTermType, this);
        this.serviceTermType.title = 'Вид на услугата, спрямо срока за предоставянето
й';

        ...
        this.serviceApplicantReceiptData = ko.observable(new
ServiceApplicantReceiptData());
        this.serviceApplicantReceiptData.title = 'Данни за получаване на резултат от
услуга от заявителя';

        ...
        this.applicationSubject = ko.observable(new PersonBasicData());
        this.attachedDocuments = ko.observable(new AttachedDocuments());
        this.electronicAdministrativeServiceFooter = ko.observable(new
ElectronicAdministrativeServiceFooter());
    };

    PermanentAddressCertificateApplication.prototype = function () {
        var initServiceTermType = function () {
            if (gp.isLoadingDocument === true) {
                var serviceTypeCode = this.serviceTermType();
                if (serviceTypeCode) {
                    this.serviceTermType.displayValue(ko.utils.arrayFirst(this.serviceTermType.serviceTermTypes(),
function (item) {
                        if (item.key === serviceTypeCode) {
                            return item;
                        }
                    }));
                }
            }
        };
    };
}

```

```

    },
    toJSON = function () {
        if (this.attachedDocuments &&
            this.attachedDocuments.attachedDocument &&
            this.attachedDocuments.attachedXmlDocument &&
            this.attachedDocuments.attachedDocument.length === 0 &&
            this.attachedDocuments.attachedXmlDocument.length === 0)
        {
            this.attachedDocuments = undefined;
        }
        return Utils.toJSONForXML(this);
    };
    return {
        toJSON: toJSON,
        initServiceTermType: initServiceTermType
    }
}();
return PermanentAddressCertificateApplication;
}
);

```

Идентификаторът на дефинирания модул спрямо конвенциите на RequireJS е „\_eServiceProvider/PermanentAddressCertificateApplication“, а масивът с изброен списък от идентификатори представя зависимостите на този сегмент от други обекти.

За представянето на самия обект, представящ ИО от тип сегмент е приложен методът Prototype Revealing Pattern. В конкретика, за всеки ИО от тип Сегмент е дефиниран конструктор с наименование съответстващо на наименованието на главния елемент (*root*), указан в XML дефиницията в РОС. В *prototype* частта му са дефинирани неговите функции и накрая са определени кои от тях да бъдат публично достъпни. Наложени са някои конвенции, които всеки такъв обект трябва да следва, за да бъде възможна универсалната им обработка в по-нататъшните стъпки от разработката на приложението.

Конвенцията за съпоставяне на наименованията на полетата на обекта и очакваните наименования на генерираните XML елементи, описана при „Представяне на ИО от тип Стойност“ важи и в случая, когато полето е съставен обект т.е. Сегмент. Конфигурирането изрично на допълнително съответствие също става аналогично.

Полетата за този сегмент са дефинирани спрямо структурата на XML дефиницията в РОС. Те могат да бъдат или Стойности или други Сегменти. Във втория случай полето се инициализира като се извиква конструктора на съответния Сегмент, който от своя страна е дефиниран по аналогичен начин на тук описвания обект. Именно по този начин е реализирана йерархията от обекти, които от своя страна са дефинирани аналогично, следвайки приетите конвенции.

Относно дефинираните методи за обектите също се използват конвенции. Всеки от тях трябва да предоставя имплементация на собствена публична функция *toJSON*. В общия случай тя от своя страна извиква функцията *Utils.toJSONForXML()* с подадени като параметри текущия обект (задължително) и служебните данни за указване на специфики при генерирането на XML съдържание, съответстващо на обекта (namespace, съответствия на наименования между полета и генерирани XML елементи) – *this.\_settings.options*, ако такива са дефинирани. Тук е мястото и за специфични обработки, които биха се случили непосредствено преди конвертирането на обекта

във вид, подходящ за генериране на XML съдържание. В цитирания пример се прави проверка дали при попълване на данни за този обект са зададени някакви данни за Приложени документи (обекти дефинирани по аналогичен начин). В случай, че такива не са зададени полето, съдържащо списък с Приложени документи се занулява. По този начин се избягва генерирането на празен елемент в резултатния XML документ.

Другата функция дефинирана в примера е *initServiceTermType()*. Този тип функции също следват конвенция, макар и нейната цел е по-скоро яснота на кода и по-лесна поддръжка (наименованието е „*init*” + наименование на поле с водеща главна буква). Тук се извършват допълнителни обработки, когато обектът се инициализира в режим на визуализиране на подадено съдържание. Те се отнасят за по-специфичната обработка на конкретно поле и се извикват при промяна на неговата стойност - *.serviceTermType.subscribe(this.initServiceTermType, this)*, но функционалността се изпълнява само в режим на визуализация - *if (gp.isLoadingDocument === true)* {...}. В конкретния случай *initServiceTermType* отговаря за намирането на наименованието (*name*) за термина, чийто ключ (*key*) отговаря на подадената стойност за полето за „Вид на услугата, спрямо срока за предоставянето ѝ” (*serviceTermType*) от номенклатурата „Номенклатура на видовете услуги, спрямо срока за предоставянето им” (*Enums.serviceTermTypes*).

#### 4.2.5 Представяне на ИО от тип Документ

Представянето на ИО от тип документ е аналогично на това на ИО от тип Сегмент.

Разглежданият документ е дефиниран с идентификатор „documents/PermanentAddressCertificateApplication”. Указани са също неговите зависимости от други файлове (отново дефинирани чрез идентификатори) – в случая библиотеката Knockout, обектът за работа с XML – Utils и обектът, представящ сегмента PermanentAddressCertificateApplication.

```
define('documents/PermanentAddressCertificateApplication',
  ['ko', 'Utils', '_eServiceProvider/PermanentAddressCertificateApplication'],
  function (ko, Utils, PermanentAddressCertificateApplication) {
    var PermanentAddressCertificateApplicationDocument = function () {
      this.segment = ko.observable(new PermanentAddressCertificateApplication());
      this._settings = {};
      this._settings.fromXML = {};
      this._settings.xmlns = 'http://ereg.egov.bg/segment/0009-000146';
      this._settings.options = {
        xmlns: this._settings.xmlns,
        xmlAttributes: {
          '_xmlns:xsd': 'http://www.w3.org/2001/XMLSchema',
          '_xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance'
        },
        propertiesTitles: {
          segment: 'PermanentAddressCertificateApplication'
        }
      };
      this._settings.isEditable = ko.observable(true);
    };
    PermanentAddressCertificateApplicationDocument.prototype = function () {
      var toJSON = function () {
        return Utils.toJSONForXML(this, this._settings.options);
      };
    };
  });
```

```

    },
    fromJSON = function () {
        return Utils.fromJSONToJS(this, this._settings.fromXML);
    };
    return {
        toJSON: toJSON,
        fromJSON: fromJSON
    }
}());
return PermanentAddressCertificateApplicationDocument;
}
);

```

Като собствени полета, обектът за документ съдържа два елемента.

Единият елемент е системното поле „*\_settings*”, което съдържа служебна информация за самия обект, за характеристики, описващи начина му на конвертиране в желан XML формат. В случая на обект, представящ ИО от тип документ, съхраняваната в това поле информация има някои специфики, които не са характерни за другите типове обекти (тези, описващи ИО от тип Сегмент).

Така например тук се пази информацията дали този документ подлежи на редакция или не по дефиниция (в полето *this.\_settings.isEditable*).

Също характерно е полето *this.\_settings.fromXML*, в което се съхранява JavaScript обекта генериран от x2js при прочитането на XML съдържанието на документ в случай на сценарий за визуализация.

Последната специфика се отнася за структурата на полето *this.\_settings.options*, което се подава като параметър при извикване на функцията за генериране на XML документ. За този тип ИО освен данни за *targetNamespace* и съответствия на елементи и полета се дефинира още едно поле *xmlAttributes*, което се използва за задаване на стойности на атрибути за главния елемент (*root*) на XML съдържанието.

Другото собствено поле за обекта е семантично значимо и е дефинирано чрез създаване на инстанция от тип основния сегмент за този документ. Обработката на това поле става по стандартен начин. Като конвенция е прието, това поле да бъде дефинирано със програмно наименование *this.segment*, а наименованието на елемента, на който ще отговаря да бъде зададено изрично като съответствие в *propertiesTitles* полето на *this.\_settings.options*.

Следвайки същия подход, както при представянето на обекта за Сегмент (Revealing Prototype Pattern), дефинициите на функциите са описани в *prototype* частта. В случая на описанието на документ, дефинираните функции са две – функция за генериране на XML съдържание (*toJSON*) и функция, отнасяща се за прочитането на XML съдържание (*fromJSON*). Първата функция е стандартната за имплементацията на всички обекти и се извиква аналогично. Втората, обаче, се дефинира само тук. Вътрешно тя извиква функция от помощния обект *Utils*, която прави съответствието между генерирания JavaScript обект при прочитането на XML съдържание и обекта, отговарящ на текущия тип документ. Двата обекта се подават като параметри на тази функция: *Utils.fromJSONToJS(this, this.\_settings.fromXML)*.

#### 4.2.6 Работа с XML съдържание и JavaScript обекти

След като вече беше изяснено как всеки тип ИО вписан в РОС се представя като програмен обект в контекста на софтуерното приложение, следващата стъпка е да бъде изяснен подходът, чрез който от така представените обекти се генерира резултатен XML документ, отговарящ на всички изисквания за електронен структуриран документ. Ще бъде разгледан и подходът за реализиране на обратния сценарий, т.е. от вече съществуващ електронен документ да се създаде и инициализира програмен обект с подходящата структура, който да предоставя възможност за обработка в контекста на приложението. Така създаденият обект да може да визуализира стойностите от съдържанието на XML документа и да предоставя възможност за редакция в случаите, когато такова действие е удачно за съответния тип документ.

##### 4.2.6.1 Генериране на XML структуриран документ от JavaScript обект

При описанието на представянето на сложните типове ИО Сегменти и Документи, беше споменато, че всеки такъв обект спазва конвенция за дефиниране на функция *toJSON*. Тази функция има специално поведение в контекста на Knockout и *ko.toJSON*. Всъщност това се отнася и изобщо за конвертирането на JavaScript обект в JSON чрез *JSON.stringify* – за по-новите браузъри или *json2.js* – библиотека изпълняваща същата функционалност за по-стари браузъри, защото *ko.toJSON* в изпълнението си също прави обръщение към *JSON.stringify*.

*„Ако обект, който се конвертира към JSON има поле наречено *toJSON*, чиято стойност е функция, тогава методът *toJSON* променя поведението на процеса по конвертиране към JSON: вместо да бъде конвертиран самият обект, стойността върната от метода *toJSON* се конвертира.”*<sup>[15]</sup>.

Спецификата на тази функция е използвана като за всеки обект имплементирането ѝ следва конвенцията – да извиква *Utils.toJSONForXML*, която от своя страна обработва отделните полета на обекта а за съставните се обръща към тяхната *toJSON* функция. По този начин се постига индиректна (косвена) рекурсия за обработка на елементите на йерархията от обекти, която представя един документ. Затова при извикване на метода за главния елемент – документ, всъщност се извършва обхождане на цялата йерархия.

```
...  
var tempJSON = ko.toJSON(self.doc);  
...
```

В *tempJSON* се съдържа JSON представяне на конструирания обект след извършени необходимите корекции с цел постигане на желания формат, който трябва да бъде конвертиран в XML.

В повече конкретика процесът по конвертиране на обекта може да бъде изяснен като се разгледа в детайли изпълнението на функцията *Utils.toJSONForXML*.

При нейното извикването се обхождат всички полета на подадения като параметър JavaScript обект (*io*) и за всяко от тях се изпълняват следните стъпки:

- вземат се подадените като параметър опции (*o*) описващи специфики за атрибутите XML елемента, който трябва да се генерира като резултат от тази функция;

```

...
var options = $.extend({ xmlns: undefined, xmlAttributes: undefined }, o)
...

```

- не се взема под внимание служебното поле „*\_settings*“;

```

for (var i in io) {
    if (i === '_settings') {
        continue;
    }
    ...
}

```

- проверява се дали е подадено изрично заглавие за съответното поле: ако да, се взема то като наименование на съответното поле в новия обект (респективно на генерирания XML елемент), ако не – за наименование се взема наименованието на полето започващо с главна буква;

```

for (var i in io) {
    ...
    if (options.propertiesTitles !== undefined && options.propertiesTitles[i] !== undefined) {
        capitalI = options.propertiesTitles[i];
    } else {
        capitalI = i.charAt(0).toUpperCase() + i.slice(1);
    }
    ...
}

```

- опциите подадени като параметър (стойности за namespace и други атрибути), както и самата стойност за съответното поле се присвояват на точно определени полета на новия обект (*\_xmlns*, *\_\_text*)

```

...
result[capitalI] = $.extend({}, {
    _xmlns: options.xmlns,
    __text: io[i]
}, options.xmlAttributes);
...

```

- изрично се обработват по-специални типове обекти (напр. масиви, дати) използва се функционалността на *toJSON* и описаната индиректна рекурсия.

```

...
result[capitalI] = $.extend({}, io[i].toJSON(), { _xmlns: options.xmlns },
options.xmlAttributes);
...

```

След изпълнението на описаните стъпки, стартирани с извикването на *toJSON* на главния елемент-документ, в резултат е налице JSON обект във формат, който гарантира генерирането на желаното XML съдържание след обръщение към функционалността на *x2js*. Тъй като за целта е

необходим JavaScript обект, то се прави нова конверсия и вече за новия обект се извиква генерирането на XML.

```
...
var tmpJSONJS = ko.utils.parseJson(tempJSON);
var xmlDirective = "<?xml version='1.0' encoding='utf-8' ?>";
var tempXML = xmlDirective + x2js.json2xml_str(tmpJSONJS);
...
```

Резултатният XML документ за примерната реализация на електронен документ е предоставена в приложение „Пример за XML съдържание на документ”.

#### 4.2.6.2 Създаване и инициализиране на JavaScript обект от XML съдържание

Обратният процес по прочитане и инициализиране (парсване) на съдържанието отново се реализира чрез прилагане на рекурсия при обхождане на йерархията от обекти. За разлика от предния сценарий, в този става въпрос за директна рекурсия. Тази логика е реализирана в тялото на функцията *Utils.fromJSONToJS*. Тя приема като параметри двата обекта – инициализиран нов обект от съответния тип (io) и този генериран от x2js (data). Още една съществена разлика с подхода при имплементацията на предния сценарий, тук тази функция се извиква само от едно място – главния обект в йерархията, т.е. документа. Като параметри се подават самият обект, дефиниращ документа (*this*) и неговото поле *this.\_settings.fromXML*, в което се съхранява генерирания обект от прочетеното съдържание:

```
...
var tempJSON = x2js.xml_str2json(xmlContent);
...
self.doc()._settings.fromXML = tempJSON;
self.doc().fromJSON();
...
```

За всяко поле или обект от новосъздадения обект се извършват следните действия:

- от полето „*\_settings*” се извлича ако за съответното поле има дефинирано заглавие на полето от обекта, съдържащо съдържанието от XML документа, ако няма дефинирано такова, тогава съответствието се прави спрямо конвенцията;

```
var options = {};
for (var j in io['_settings']) {
    if (j === 'options') {
        options = io['_settings'][j];
        break;
    }
}
for (var i in io) {
    if (i === '_settings') {
        continue;
    }
}
var capitalI;
if (options !== undefined && options.propertiesTitles !== undefined &&
options.propertiesTitles[i] !== undefined) {
    capitalI = options.propertiesTitles[i];
```



```

} else {
    capitalI = i.charAt(0).toUpperCase() + i.slice(1);
}
...

```

- изрично се обработват по-специалните случаи (масиви – извиква се функцията за всеки елемент; масив, но само с един елемент; *boolean* тип). За съставните обекти (т.е. всички, които имат дефинирано поле от тип функция с наименование „toJSON“) се прави рекурсивно обръщение към функцията:

```

...
if (data[capitalI] && io[i]() !== undefined && io[i]()["toJSON"] !== undefined && typeof
io[i]()["toJSON"] === 'function') {
    fromJSONToJS(io[i](), data[capitalI]);
}
...

```

- за стойност на поле на новия обект се взема стойността, съхранена в полето `__text` от обекта с данни или стойността съхранена в поле със съответстващото наименование;

```

if (io[i] && data[capitalI] && data[capitalI].__text && i !== "toJSON") {
    ...
    if (io[i] && data[capitalI] && i !== "toJSON") {
        ...
        io[i](data[capitalI].__text);
    } else if (io[i] && data[capitalI] && i !== "toJSON") {
        ...
        io[i](data[capitalI]);
    }
}

```

В резултат от изпълнението на така описания процес се инициализира обектът, подаден като параметър (*io*) при първоначалното извикване, т.е. йерархията от обекти, представлява съответния електронен документ.

#### 4.2.7 Създаване, визуализиране и редактиране на електронен структуриран документ

За да бъде приложението практически приложимо и оптимално удобно за работа е необходимо да предоставя възможност за работа с тези документи във всеки възможен режим за работа. Три са възможните сценарий за работа с тези документи:

- създаване на нов документ;
- визуализиране на вече създаден документ;
- редактиране на вече създаден документ.

Първият сценарий за създаване на нов документ предполага нуждата от задаване на конкретния вид документ, който потребителят би искал да създаде. За да се избегне конкретно изброяване и да се улесни разширяването на функционалността на приложението, реализацията на текущото решение предлага динамично установяване на вида документ за създаване. Чрез установена конвенция, при подаване на наименованието на елемента от XML-документа (съответстващ и на наименованието на JavaScript обекта, дефиниращ документ) и чрез

използване на функционалността на RequireJS динамично се определя вида на документа и се извличат ресурсите отнасящи се само за него. Представена е примерна имплементация:

```
var createNewDocument = function () {  
    if (this._settings.documentType()) {  
        var self = this,  
            //намираме наименованието на модула за зареждане  
            docName = getDocumentName(this._settings.documentType());  
        //зареждане на необходимите js файлове  
        requirejs([docName], function (document) {  
            //инициализиране на искания документ  
            self.doc(new document());  
        })  
    }  
}
```

Наименованието на документа, отговарящо на наименованието на главния елемент (*root*) в XML документа се съхранява в служебните данни за документа в полето *this.\_settings.documentType()*. На базата на това наименование и на структурата на приложението (обектите дефиниращи документите са отделени в отделна папка) се извлича идентификатора на модула за конкретния подаден вид документ.

```
//намираме наименованието на модула за зареждане  
getDocumentName = function (documentType) {  
    //модула се локализира в папката за документи/име на документ  
    var docName = 'documents/' + documentType;  
    return docName;  
}
```

Така намереният идентификатор се подава като параметър на RequireJS като текстово поле (спрямо изискванията на библиотеката), която в резултат зарежда модула отговарящ на този идентификатор, както и всички останали модули, които той реферира. Със запазената дума „*new*” се извиква конструктора на вече заредения обект и така се инстанцира нужният на приложението съставен обект (йерархия от обекти) за работа със структуриран документ. Визуализирането на тази структура се осъществява чрез „елемент без контейнер” и функционалността за композиране на визуализиращи елементи (*compose*), предоставена от Durandal:

```
<!-- ko compose: { model: doc, area: 'Editor' } -->  
<!-- /ko -->
```

, където за модел за визуализиране е току-що създаденият празен документ и той се визуализира в режим на редактиране. В резултат за примерната имплементация пред крайния потребител се отваря следния екран:

## Заявление за издаване на удостоверение за постоянен адрес

### Данни за услуга

Административна услуга:	Издаване на удостоверение за постоянен адрес
Наименование на тип документ:	Заявление за издаване на удостоверение за постоянен адрес
Вид на заявлението:	Първоначално заявление за предоставяне на електронна административна услуга <input checked="" type="checkbox"/>

### Доставчик на електронни административни услуги

Административен орган - Доставчик на електронна административна услуга, ЕИК/БУЛСТАТ XXXXXXXXX

### Заявител на електронна административна услуга

Електронна поща:	<input type="text"/>		
<input checked="" type="checkbox"/> Авторът съпада с получателя			
Изберете тип на автор:	Физическо лице <input checked="" type="checkbox"/>		
Трите имена:	Собствено име <input type="text"/>	Бащино име <input type="text"/>	Фамилно име <input type="text"/>
Псевдоним:	Псевдоним <input type="text"/>		
Идентификатор:	ЕГН <input checked="" type="checkbox"/>	Единен граждански номер <input type="text"/>	

### Данни за контакт

Област:	<input type="text"/>	Община:	<input type="text"/>
Населено място:	<input type="text"/>	Пощенски код:	<input type="text"/>
Адрес:	<input type="text"/>	Пощенска кутия:	<input type="text"/>
Нов телефонен номер:	<input type="text"/> <input type="button" value="Добави"/>	Нов факс номер:	<input type="text"/> <input type="button" value="Добави"/>
Списък с добавени телефонни номера:		Списък с добавени факс номера:	

### Изпращане на „Потвърждаване за получаване“

☐ Изпращане на електронния документ и приложените към него документи заедно с документа „Потвърждаване за получаване“

### Вид на услугата, спрямо срока за предоставянето ѝ

Вид на услугата:	Обикновена <input checked="" type="checkbox"/>
------------------	--

### Начин на получаване на резултат от услуга

Начин на получаване:	Чрез електронна поща/уеб базирано приложение <input checked="" type="checkbox"/>
----------------------	--

### Данни за подател

Трите имена:	Собствено име <input type="text"/>	Бащино име <input type="text"/>	Фамилно име <input type="text"/>
Псевдоним:	Псевдоним <input type="text"/>		
Идентификатор:	ЕГН <input checked="" type="checkbox"/>	Единен граждански номер <input type="text"/>	

### Приложени документи

<input type="button" value="Генерирай"/>	<input type="button" value="Валидирай"/>
--	--

Фиг. 8 Екран за режим на редакция на електронен структуриран документ

Другият основен сценарий за работа с документ е визуализирането на вече създадено съдържание. Примерна реализация:

```
var loadDocumentContent = function (...) {
    var self = this;
    var xmlContent = this._settings.xmlContent();
    var tempJSON = x2js.xml_str2json(xmlContent);
    var documentType;
    for (var i in tempJSON) {
        documentType = i;
        break;
    }
    gp.isLoadingDocument = true;
    var docName = getDocumentName(documentType);
    requirejs([docName], function (document) {
        self.doc(new document());
        ...
        self.doc()._settings.fromXML = tempJSON;
        self.doc().fromJSON();
        viewLocator.translateViewIdToArea(system.getModuleId(self.doc()), 'Display');
        ...
        generateErrXML(self);
        self._settings.showErrors(self.doc.errors() && self.doc.errors().length > 0);
        self._settings.isEditable(self.doc()._settings.isEditable());
        gp.isLoadingDocument = false;
    })
}
```

Подаденото съдържание (XML документ) се запазва в служебните данни на главния елемент (ViewModel) и по-конкретно в полето `this._settings.xmlContent()`. За така прочетената стойност се изпълняват поредица от стъпки, които гарантират възможността за визуализиране и по-нататъшна работа с документа. Най-напред се прочита съдържанието и се създава JavaScript обект генериран от JavaScript библиотеката за работа XML – x2js. След това се определя вида на документа, за който се отнася съдържанието като се взема първото поле на генерирания обект и се прави съответствие аналогично на процеса за създаване на нов документ, отговарящ на термините на приложението. Новосъздаденият документ се популира със стойностите прочетени от подадения документ като се извиква дефинираната за всеки обект от тип документ функция - `self.doc().fromJSON()` и за съдържание на този документ се подава генерираният от x2js обект - `self.doc()._settings.fromXML = tempJSON`. Така се извиква функционалността за прочитане на XML в Utils.js, която беше описана в предходните точки. След като вече е създаден и инициализиран обектът, с който работи приложението се използва функционалността на Durandal за определяне режима на работа – визуализация (Display). Визуализирането на така създадения обект се извършва аналогично на новосъздадения обект от предния сценарий, но вече в друг режим на работа:

```
<!-- ko compose: { model: doc, area: 'Display' } -->
<!-- /ko -->
```

Следват извикване на функция, отговаряща за валидацията на обекта (или по-точно цялата йерархия от обекти), проверка дали за този обект е позволен режим на редакция - `self._settings.isEditable(self.doc()._settings.isEditable())` и обозначаване, че създаденият обект е генериран от вече съществуващо съдържание - `gp.isLoadingDocument = false` (от значение е при инициализиране на някои от обектите). В резултат, в случая на примерната реализация, крайният потребител вижда съдържанието на документа визуализирано по следния начин:

#### Грешки

- 0006-000024: Невалиден "ЕИК/БУЛСТАТ".
- 0006-000069: Невалидна структура на обекта съгласно XML дефиницията му, вписана в регистъра на информационните обекти.

## Заявление за издаване на удостоверение за постоянен адрес

### Данни за услуга

Административна услуга:	Издаване на удостоверение за постоянен адрес
Наименование на тип документ:	Заявление за издаване на удостоверение за постоянен адрес
Вид на заявлението:	Първоначално заявление за предоставяне на електронна административна услуга

### Доставчик на електронни административни услуги

Административен орган - Доставчик на електронна административна услуга, ЕИК/БУЛСТАТ XXXXXXXXX

### Заявител на електронна административна услуга

Електронна поща:	ivan.ivanov@email.bg
------------------	----------------------

### Данни за автор - Физическо лице

Трите имена:	Иван	Иванов	Иванов
Псевдоним:			
Идентификатор:	ЕГН	6101047500	

### Данни за получател - Физическо лице

Трите имена:	Иван	Иванов	Иванов
Псевдоним:			
Идентификатор:	ЕГН	6101047500	
В качеството на:			

### Данни за контакт

Област:	BGS Бургас	Община:	BGS04 Бургас
Населено място:	07079 Бургас	Пощенски код:	
Адрес:		Пощенска кутия:	

### Изпращане на „Потвърждаване за получаване“

☐ Изпращане на електронния документ и приложените към него документи заедно с документа „Потвърждаване за получаване“

### Вид на услугата

Вид на услугата:	Обикновена
------------------	------------

### Начин на получаване на резултат от услуга

Начин на получаване:	Чрез електронна почта/уеб базирано приложение
----------------------	---

### Данни за подател

Трите имена:	Иван	Иванов	Иванов
Псевдоним:			
Идентификатор:	ЕГН	6101047500	

### Приложени документи

Редактирай Генерирай Валидирай

Фиг. 9 Екран за режим на визуализация на електронен структуриран документ

Последният сценарий за работа с документ е за вече създадено съдържание да се предостави начин за редактирането му. При зареждането на съдържанието в режим на преглед интерфейсът на приложението не позволява редакция на полетата на документа. В зависимост от вида документ се определя дали неговото съдържание подлежи да редактиране. За някои документи, например автоматично генерирани от АИС („Потвърждаване за получаване”, „Съобщение, че получаването не се потвърждава”), процесът на работа с тях не предполага тяхното съдържание да подлежи на редактиране от крайния потребите, т.е. те трябва да са достъпни само за преглед. За други, обаче, стандартният метод на работа изисква на потребителя да бъде предоставена възможност след преглед на съществуващо съдържание да може да направи корекции по него. Информацията за това дали даден документ подлежи на редактиране или не се пази в служебните полета на обекта за документ: `this._settings.isEditable()`. В зависимост от стойността на това поле в интерфейса на приложението се визуализира или се скрива бутон за редакция на документа:

```
<input type="button" class="btn" name="editDocument" value="Редактиране" data-bind="visible:
_settings.isEditable, click: editDocument" />
```

В случая, когато редактирането е позволено като режим на работа при натискане на бутона за „Редактирай” се преминава от режим на преглед в режим на редактиране. На предходната стъпка за визуализиране на подаденото съдържание (описаният по-горе сценарий) вече е създаден и инициализиран обектът, конструиран спрямо термините на приложението. Затова в този сценарий остава задачата за този обект да се извърши прехода от единия в другия режим на работа. Изпълнението на тази задача става относително тривиално.

```
editDocument = function () {
    ...
    this._settings.showErrors(false);
    viewLocator.translateViewIdToArea(system.getModuleId(this.doc()), 'Editor');
    this._settings.isEditable(false);
    return true;
}
```

Стъпките за изпълнението на задачата са две. Първо чрез програмния интерфейс (API), на Durandal се намира идентификаторът за модула представящ документа, чието съдържание е визуализирано – `system.getModuleId(this.doc())`. На втората стъпка се използва функционалността за разделението на визуализиращите шаблони на области и преминаването от една в друга област. Така с извикването на `viewLocator.translateViewIdToArea(moduleId, area)` се преминава от режим на преглед в режим на редактиране. Самият обект за работа остава непроменен и съответно неговите полета са инициализирани спрямо подаденото съдържание. Бутонът за „Редактирай” се скрива. Оттук работата с документа е аналогична на първия сценарий – за създаване на нов документ. За крайния потребител процесът на работа протича в две стъпки:

- Отваряне на екран аналогичен на този от втория описан сценарий – за визуализиране на съдържание.

- След натискане на бутона „Редактирай“ (при условие, че го има) отваряне на екран за редактиране аналогичен на този от първия описан сценарий – за създаване на нов документ, но с попълнени стойностите от първата стъпка.

#### 4.2.8 Валидиране на електронен структуриран документ

Много важен момент при работата с електронен документ е гарантирането, че въведените стойности за попълване на данни отговарят на изискванията, дефинирани за съответните полета. Поради естеството на предметната област в контекста, на която е разработено текущото решение, изискванията на които трябва да отговарят въведените данни са конкретно дефинирани. За всеки ИО вписан в РОС е въведено и описание за „Указания за проверка на валидност“. Реализацията на тази проверка в текущото софтуерно приложение е строго съобразена с тези указания.

Вземайки предвид подхода за представяне на ИО в програмен аспект – чрез JavaScript обекти, които използват Knockout за свързване с графичното им представяне (*bindings*), логично и естествено идва решението валидацията на тези обекти да бъде реализирана по начин, използващ това представяне. Така се стига до използването на библиотеката Knockout.Validation. Тя лесно се интегрира в изградената структура на приложението. Нейната конфигурация се извършва във файла, където се конфигурират и относителните пътища за RequireJS, както и други инициализиращи дейности – app.js.

```
...
ko.validation.configure({
  insertMessages: false, // automatically inserts validation messages as
  <span></span>
  parseInputAttributes: true, // parses the HTML5 validation attribute from a form
  element and adds that to the object
  writeInputAttributes: true, // adds HTML5 input validation attributes to form
  elements that ko observable's are bound to
  decorateElement: true, // false to keep backward compatibility
  errorElementClass: 'koError', // class to decorate error element
  errorMessageClass: 'text-error', // class to decorate error message
  grouping: {
    deep: true, //by default grouping is shallow
    observable: true //and using observables
  }
});
ko.validation.registerExtenders();
...
```

Там се задават някои настройки на това, как да се изобразяват възникнали грешки, къде да се позиционира съобщението за грешка, дали валидацията да се изпълнява в дълбочина по йерархията или само за първото ниво за подадения елемент.

Самото прилагане на правилата за валидация става отново в контекста на Knockout и проследяеми променливи. То се осъществява чрез разширяване на конкретните полета, дефинирани като проследяеми (*observable*). Там се изброяват кои са правилата за валидация, които трябва да се приложат за конкретно поле и съответно се подават необходимите параметри, които очаква функцията изпълняваща самата валидация.



```

this.identifier = ko.observable(new PersonIdentifier());
this.identifier.title = 'Идентификатор на физическо лице';
this.identifier.extend({
    fieldIsRequired: {
        field: this.identifier,
        sectionTitle: this._settings.sectionTitle
    }
});

```

По този начин се гарантира, че при всяка промяна на стойността за даденото поле, ще се извикат функциите за валидация и те ще проверят дали новата стойност изпълнява дефинираните изисквания. В случай на грешка се връща съобщение за грешка, което подлежи на предефиниране, както и самите правила за валидация.

Основно предимство на тази библиотека, освен тясната интеграция с Knockoutjs е възможността за разширяемост, която тя предлага. Така по лесен начин се дефинират правила за валидация, които следват изискванията дефинираните в РОС правила за валидация и съответните термини, които се генерират в случай на грешка.

```

ko.validation.rules['fieldIsRequired'] = {
    validator: function (val, params) {
        return !isEmptyVal(val);
    },
    message: function (params) {
        return '0006-000015: Полето "{0}" от секцията "{1}" трябва да е
попълнено.'.replace(/\{0\}/gi, params.field.title).replace(/\{1\}/gi, params.sectionTitle);
    }
};

```

В резултат на извършената проверка, в случай на възникване на грешка се генерира съобщение за грешка, което следва конвенция с цел по-нататъшна обработка. В текста на съобщението се заместват стойностите подадени като параметри при извикването на конкретната валидация. Съобщението има формат: „УРИ на термин, дефиниращ възникналата грешка”: „Наименование на термин, дефиниращ възникналата грешка”.

Тази конвенция е наложена поради изискването при валидация да бъде генериран електронен структуриран документ, съдържащ списък с възникналите грешки. За възникнала грешка също е дефиниран сложен ИО от тип Сегмент с УРИ: 0009-000024. Процесът по конвертиране на генерираното съобщение за възникнала грешка в обект, представящ сегмента е обособен в следната функция:

```

...
parseErrorMessage = function (msg) {
    var regIndex,
        batchNumber,
        errMessage;

    regIndex = msg.match(/^0*(\d*)-/)[1];
    batchNumber = msg.match(/-0*(\d*)/)[1];
    errMessage = msg.match(/:\s(.*)$/)[1];

    var err = new RegisteredErrorInDocumentContent();

```

```

    var termURI = new RegisterObjectURI();
    termURI.registerIndex(regIndex);
    termURI.batchNumber(batchNumber);
    err.termURI(termURI);
    err.errorDescription(errMessage);

    return err;
}
...

```

Това конвертиране се случва при извикване на валидация за цялата йерархия от обекти, дефинираща конкретен структуриран документ - *ko.validation.group(self.doc)*. В този случай натрупаните грешки, в резултат от валидацията се записват в ново поле за главния елемент, което представлява масив с грешки – *self.doc.errors()*. Всяка грешка от този масив се обработва и се генерира съответният обект:

```

...
err = parseErrorMessage(self.doc.errors()[i]());
self.errDoc.segment.registeredErrors().errors.push(err);
...

```

В този фрагмент *self.errDoc* е новосъздаден обект, представящ документа съдържащ списък с грешките. Този документ се третира от приложението по аналогичен начин в сценария за неговото генериране. В „Пример за генериран документ с грешки, възникнали при валидация” е добавен примерен вариант на такъв документ.

На един вид грешка е нужно да бъде обърнато по-специално внимание, а именно „0006-000069: Невалидна структура на обекта съгласно XML дефиницията му, вписана в регистъра на информационните обекти” – съобщение за грешка, възникнала при извършена XSD валидация.

Процесът по извършване на този вид валидация, за разлика от всички други проверки (както и изобщо по-голямата част от функционалността) не се извършва на клиента. Генерираният XML документ (по описания в предни точки метод), се изпраща на сървъра чрез асинхронна заявка.

```

validateXML = function (docContent, errDoc, errors) {
    var xmlDocument = ko.toJSON({
        "documentContent": docContent
    });
    return $.ajax({
        type: "POST",
        url: gp.applicationRootURL() + "odata/Documents/ValidateXML",
        processData: false,
        contentType: 'application/json; charset=utf-8',
        data: xmlDocument
    }).done(function (errData) {
        if (errData) {
            var err = parseErrorMessage(errData["odata.error"].message.value);
            errDoc.segment.registeredErrors().errors.push(err);
            errors.push(errData["odata.error"].message.value);
        }
    });
}

```

Там се извършва съответната валидация и при възникване на грешка от съответния тип (*XmlException* или *XmlSchemaValidationException*) към клиента се връща обект следващ структурата на другите видове грешки. Полученият обект се обработва също по аналогичен начин на другите грешки – от съобщението се генерира нужният обект и той се добавя към документа със списък от грешки.

Тази комуникация между клиента и сървъра се осъществява, следвайки OData протокол. За него и за други случаи на комуникация – за правене на заявки за извличане на данни от пакети от данни, ще бъде изложена по-детайлна информация в следващата точка.

#### 4.2.9 Управление на данните

Функционалността отговаряща за обмен на данни между клиентската и сървърната част е изнесена в отделен файл `datacontext.js`. В него са дефинирани всички заявки, които се изпращат към сървъра, използвайки средствата предоставени от Breeze. Всяка функция отговаряща за извличането на данни от сървърната част приема поне един параметър – променлива, в която ще се съхрани полученият резултат. Във функцията е дефинирана заявка и тя се изпълнява чрез глобално дефинирания `EntityManager`. При получаване на отговор от сървъра се извиква функция – `success`, отговаряща за обработването на резултата (в конкретния случай запазването му в променливата подадена като параметър).

В примерния код е демонстриран този подход чрез дефиниране на заявка за извличане на всички общини в рамките на една област, които се съхраняват в пакет от данни дефиниран в `POC`. В този случай функцията приема още един параметър (освен стандартния, за съхраняване на резултата), който подава стойността използвана за филтриране на данните – кода на областта, за която се отнасят общините.

```
...
var manager = gp.configureBreezeManager();
...
getMunicipalitiesByDistricts = function (districtCode, nomMunicipalities) {
    var query = new breeze.EntityQuery()
        .from("Municipalities")
        .where("districtCode", "==", districtCode);
    ...
    manager.executeQuery(query)
        .then(success)
        .fail(function (e) {
            ...
        });

    function success(data) {
        data.results.unshift({});
        nomMunicipalities(data.results);
    }
}
...
```

В рамките на приложението е дефиниран общ/споделен обект за `EntityManager`, който отговаря за изпълнението на всяка дефинирана заявка. По този начин извлечените обекти и данни

могат да бъдат преизползвани многократно и от различни извикващи ги обекти. Конфигурирането и инстанцирането на този споделен обект се извършва в GlobalParameters.js. За него изрично е зададено, че за комуникация със сървърната част се използва OData протокол.

```
...
configureBreezeManager = function () {
    breeze.config.initializeAdapterInstances({
        dataService: "OData"
    });
    breeze.NamingConvention.camelCase.setAsDefault();
    var manager = new breeze.EntityManager(applicationRootURL() + 'odata');
    return manager;
};
...
```

При изпълнение на заявка чрез дефинирания по този начин обект извлеченият резултат се съхранява в локалния му кеш. Използвайки подхода на споделянето му дава достъп на цялото приложение до този набор от данни. Така при последваща нужда от използване на същите данни, те могат да бъдат преизползвани без да се налага да се прави нова заявка към сървъра. Спестяването на тези обръщания по мрежата оптимизира производителността и подобрява потребителското усещане при работа с приложението. В следващия фрагмент програмен код е представена проверката дали данните са вече извлечени и тяхното използване, в случай на положителен резултат.

```
...
var lQuery = undefined;
try {
    lQuery = manager.executeQueryLocally(query);
} catch (e) {
    lQuery = undefined;
}
if (lQuery != undefined && lQuery.length > 0) {
    var query = query.using(breeze.FetchStrategy.FromLocalCache);
}
...
```

Тук *query* е вече дефинираната заявка от предходния пример. Ако данните все пак не са заредени локално, то заявката се изпълнява по стандартния метод описан по-горе. Целият този сценарий остава скрит за извикващите обекти. Те просто се обръщат към функцията с необходимите параметри и извършват необходимите обработки при получаване на резултата.

```
...
datacontext.getMunicipalitiesByDistrict(newValue.code(),
this.municipalityCode.nomMunicipalities)
    .then(function () {
        ...
    });
...
```

По този начин се извършват обръщенията от клиента към сървъра за извличане на данни. Този сценарий, както беше споменато, се изпълнява в случаите на избор на стойност, която има ограничение да е измежду стойностите включени в пакетите от данни дефинирани в РОС. Те от своя страна, аналогично на електронните структурирани документи, са представени чрез XML съдържание. Поради тази причина в процеса на обработка на данните от сървърна страна, всъщност става въпрос за извличане на данни от XML документи. В контекста на Breeze и OData протокола се очаква мета-описание на модела от данни, с който се борава. Тази дефиниция се осъществява чрез *OdataModelBuilder*. Чрез него се дефинират обектите, които се използват за представяне на прочетените от XML съдържанието данни.

```
ODataModelBuilder builder = new ODataConventionModelBuilder();  
var municipalityConfig = builder.EntitySet<Municipality>("Municipalities");  
municipalityConfig.EntityType.HasKey<string>(e => e.Code);
```

Подобен сценарий и детайлно описание на подхода за неговото реализиране е представен в [18].

Преобразуването на данните от XML към C# обекти се извършва чрез LINQ to XML и резултатът се връща под формата на *IQueryable<Municipality>*.

Веднъж прочетени данните се съхраняват в паметта на сървъра за преизползване при следващо извикване. Изрично е зададено тези данни да се инвалидират при промяна на файла, съдържащ съответния пакет от данни.

```
string fileLocation = "";  
municipalities = LoadDataPackages.GetMunicipalities(out fileLocation);  
HttpContext.Current.Cache.Insert("Municipalities", municipalities, new  
CacheDependency(fileLocation), DateTime.Now.AddMinutes(10), Cache.NoSlidingExpiration);
```

## 5 Тестване

Тестове за приложението са отделени в отделна структурна единица. В отделна папка са разпределени дефинициите на самите тестове, както и тяхната конфигурация.

За тестване функционалността на приложението се използват QUnitjs. Тестовите, които са дефинирани са модулни (*Unit tests*). Те са отделени в логически единици – модули, които съдържат в себе си тестове отнасящи се до логически свързана функционалност. Тяхното изпълнение се извършва асинхронно като зависимостите на тестовите и обектите, които са необходими за тяхното изпълнение са дефинирани чрез RequireJS – аналогично на подхода използван за реализацията на функционалността на самото приложение.

```
define('PhoneNumbersTest',
    ['common/PhoneNumbers'],
    function (PhoneNumbers) {
        QUnit.start();
        // Define the QUnit module and lifecycle.
        module("PhoneNumbers tests");

        asyncTest("Adding new phoneNumber", function () {
            var phoneNumbers = new PhoneNumbers();
            phoneNumbers.phoneNumber.itemToAdd('123');
            phoneNumbers.addPhoneNumber();
            strictEqual(phoneNumbers.phoneNumber().length, 1);
            start();
        });
        ...
    });
```

За автоматизираното изпълнение на така дефинираните тестове се използва допълнително добавен *plugin* – qunit-composite. Чрез неговото използване са предоставени възможности за визуализация, автоматизация и организация на тестови случаи. В резултат на тяхното изпълнение се зарежда екран с отчет за получените резултати, подобен на изображения на Фиг. 10. Фигурата изобразява примерен набор от тестове, подбран с цел описание на подхода за дефиниране на тестови сценарии.

# Test Suite

☐ Hide passed tests
 ☐ Check for Globals
 ☐ No try-catch

Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E; InfoPath.2; rv:11.0) like Gecko

Tests completed in 1397 milliseconds.  
 14 assertions of 14 passed, 0 failed.

1. Composition #1: qUnitTests/AttachedDocumentsTest.cshtml (0, 2, 2) Rerun 257 ms

1. Add attachedDocuments tests: Adding new attachedDocument: okay  
 2. Add attachedDocuments tests: Unsuccessful adding new attachedDocument: okay

2. Composition #1: qUnitTests/PhoneNumbersTest.cshtml (0, 4, 4) Rerun 267 ms

1. PhoneNumbers tests: Adding new phoneNumber: okay  
 2. PhoneNumbers tests: Adding existing phoneNumber: okay  
 3. PhoneNumbers tests: Adding empty phoneNumber: okay  
 4. Remove phoneNumbers tests: Removing a phoneNumber: okay

3. Composition #1: qUnitTests/FaxNumbersTest.cshtml (0, 4, 4) Rerun 270 ms

1. Add faxNumbers tests: Adding new faxNumber: okay  
 2. Add faxNumbers tests: Adding existing faxNumber: okay  
 3. Add faxNumbers tests: Adding empty faxNumber: okay  
 4. Remove faxNumbers tests: Removing a faxNumber: okay

4. Composition #1: qUnitTests/PersonBasicDataTest.cshtml (0, 2, 2) Rerun 236 ms

1. Changing person identifier: Changing person identifier type to 'egn': okay  
 2. Changing person identifier: Changing person identifier type to 'Inch': okay

5. Composition #1: qUnitTests/ElectronicAdministrativeServiceHeaderTest.cshtml (0, 2, 2) Rerun 333 ms

1. ElectronicAdministrativeServiceHeader tests: Changing application type to '0006-000121': okay  
 2. ElectronicAdministrativeServiceHeader tests: Changing application type to not '0006-000121': okay

Фиг. 10 Екран на изпълнени тестове за приложението

## 6 Внедряване

### 6.1 Подход за интеграция

Поради естеството на описаното решение интегрирането му в други приложения е достатъчно лесно и неограничаващо. В качеството си на едно клиентски базирано уеб-приложение, неговата функционалност е организирана в ресурси, които се изпълняват в средата на извикващата го страна – браузъра. Поради тази причина, за използването му в рамките на друго приложение е достатъчно да бъдат добавени файловете, дефиниращи неговата функционалност. Тъй като структурата му също е строго дефинирана, то този процес е относително ясен и недвусмислен. Относно стартирането му е необходимо на страницата, в която се очаква да се инсталира редактора да се добави елемент следващ конвенциите описани в документацията на платформата Durandal (с *id*= „applicationHost”):

```
<div id="applicationHost">  
  @RenderPage("_splash.cshtml")  
</div>
```

, където „\_splash.cshtml” е страница визуализираща изображение за изчакване, докато се заредят необходимите ресурси за първоначалното стартиране на приложението. И последното изискване за интегрирането на приложението е да му бъдат предоставени необходимите входни данни за неговото стартиране.

### 6.2 Входни данни

Приложението дава възможност за работа в два режима – на редакция (за създаване на нов документ) или на преглед (за визуализиране на създадено съдържание). В зависимост от избора режим на работа то очаква да получи като входни данни съответно вида на документа за създаване или създаденото съдържание (виж т. 4.2.7).

За целите на прочитане на входните данни е дефинирано правило на свързване „hiddenInputValue”, което реализира логика на поведение точно противоположна на тази на стандартно дефинираното правило за свързване „value”:

```
ko.bindingHandlers.hiddenInputValue = {  
  update: function (element, valueAccessor, allBindingsAccessor) {  
    var hiddenVal = $(element).val(),  
    bindings = allBindingsAccessor();  
    ...  
    bindings.hiddenInputValue(hiddenVal);  
  }  
};
```

, т.е. прочита стойността, която е зададена в атрибута „value” и я запазва в проследяемата променлива подадена като параметър на правилото за свързване.



### 6.2.1 Вид документ за редакция

Когато приложението се отваря за създаване на документ, в скрито поле се задава стойността, дефинираща типа на документа, който трябва да се инициализира и съответно да се отвори форма за попълване на стойности за неговото генериране. Тази стойност спазва конвенция – наименованието на типа документ отговаря на главния елемент (*root*), дефиниран в основния сегмент, описващ документа в XSD вписан в РОС. Същата тази стойност отговаря и на наименованието на обекта реализиращ типа Документ в контекста на приложението. Тази стойност е подадена на атрибута „value” на HTML елемента и чрез правило за свързване „*hiddenInputValue*” тя се присвоява на полето „documentType” на главния обект (ViewModel), отговарящ за функционалността на приложението.

```
...
<input type="hidden" name="documentTypeName" value="@ViewBag.DocumentTypeName" data-
bind="hiddenInputValue: _settings.documentType" />
...
```

### 6.2.2 XML съдържание

В другия сценарий на работа, когато приложението се отваря в режим на преглед метода на подаване на входните данни е аналогичен, но в този случай в скритото поле се съхранява самото съдържание на XML документа. Отново то е подадено на атрибута „value” на HTML елемента и чрез същото правило за свързване „*hiddenInputValue*” то се присвоява на полето „xmlContent” на главния обект (ViewModel), отговарящ за функционалността на приложението.

```
...
<input type="hidden" name="xmlContent" value="@ViewBag.XMLContent" data-
bind="hiddenInputValue: _settings.xmlContent, applyValue: _settings.xmlContent.applyValue" />
...
```

## 6.3 Определяне режима на работа на приложението

В зависимост от това кой от двата параметъра е подаден като входен се определя и в какъв режим на работа се отваря приложението. С приоритет се проверява дали е подадено съдържание и едва, когато тази стойност е празна се търси стойност за типа документ. Ако не е подаден нито един от параметрите, се визуализира съобщение за грешка.

```
loadDocument = function () {
    ...
    //ако е подадено xml съдържание на документ
    if (this._settings.xmlContent()) {
        this.loadDocumentContent();
        this._settings.documentNotFoundMessage(undefined);
        return true;
    }
    //ако е зададено наименованието на типа документ (root елемента/име на documents модул)
    } else if (this._settings.documentType()) {
        this.createNewDocument();
        this._settings.isEditable(false);
        this._settings.documentNotFoundMessage(undefined);
        return true;
    }
}
```

```
        else {
            this._settings.documentNotFoundMessage("Не е подаден тип на заявление за  
попълване!");
        }
    };
};
```

## 6.4 Изходни данни

Като резултат от използването на функционалността на този продукт се явява един електронен структуриран документ. Той се създава при извикване на функция дефинирана в главния обект на приложението (ViewModel) и се съхранява в неговото поле „*\_settings.resultXML*”. В последствие тази стойност може да бъде прочетена и съответно обработвана по специфичен начин в зависимост от бизнес логиката на извикващото приложение.

```
generateXML = function (data) {
    var self = data;
    self._settings.resultXML('');
    var tempJSON = ko.toJSON(self.doc);
    var tmpJSONJS = ko.utils.parseJson(tempJSON);
    var xmlDirective = "<?xml version='1.0' encoding='utf-8' ?>";
    var tempXML = xmlDirective + x2js.json2xml_str(tmpJSONJS);
    self._settings.resultXML(tempXML);
    return true;
},
```

## 7 Практическо приложение

Софтуерният инструмент, предмет на разглежданата тема е внедрен успешно в АИСКАО (административна информационна система за комплексно административно обслужване), реализирана в проект „Въвеждане на комплексно административно обслужване за предоставяне на качествени услуги на гражданите и бизнеса“ [20]. Проектът е осъществен с финансовата подкрепа на Оперативна програма „Административен капацитет“, съфинансирана от Европейския съюз чрез Европейския социален фонд и от държавния бюджет на Република България. Той цели постигането на КАО. Това понятие е въведено в базисния модел на КАО, документ изготвен в същия проект. Спрямо него *„Комплексно е административното обслужване, при което административната услуга се предоставя от компетентните административни органи, лицата, осъществяващи публични функции, и организациите, предоставящи обществени услуги, без да е необходимо заявителят да предоставя информация или доказателствени средства, за които са налице данни, събирани или създавани от административни органи, първични администратори на данни, независимо дали тези данни се поддържат в електронна форма или на хартиен носител.“* [5].

Решението, резултат от изпълнението на този проект, е внедрено в три пилотни администрации:

- ИАМА (Изпълнителна агенция „Морска администрация“);
- БАБХ (Българска агенция по безопасност на храните);
- Общинска администрация Велико Търново.

За целите на заявяване и предоставяне на конкретни административни услуги за изброените администрации са изготвени дефиниции на ИО и съответната им имплементация за обработка чрез уеб-базираното приложение за работа с електронни структурирани документи. Изготвените документи са, както следва:

Документи дефинирани за услуги внедрени в Общинска администрация Велико Търново:

- Заявление за издаване на удостоверение за административен адрес на поземлени имоти;
- Удостоверение за административен адрес на поземлени имоти.

Документи дефинирани за услуги внедрени в ИАМА:

- Заявление за издаване/преиздаване на свидетелство за завършени курсове за специална и допълнителна подготовка, изисквани съгласно международни конвенции;
- Заявление за издаване/преиздаване на свидетелство за правоспособност „Водач на кораб до 40 БТ по море“ и „Водач на малък кораб“;
- Заявление за издаване на удостоверение за плавателен стаж;
- Заявление за издаване/преиздаване на свидетелство за правоспособност за плаване по ВВПЕ;
- Заявление за издаване/преиздаване на свидетелство за правоспособност за работа по море;
- Заявление за удължаване на срока на валидност на свидетелство за правоспособност за работа по море;
- Удостоверение за плавателен стаж на български език;

- Удостоверение за плавателен стаж на английски език.

Документи дефинирани за услуги внедрени в БАБХ:

- Заявление за регистрация на зоопаркове, аквариуми, терариуми, циркове, ферми, волиери и вивариуми;
- Удостоверение за регистрация на животновъден обект;
- Заявление за регистрация на обект за производство и търговия с храни;
- Удостоверение за регистрация на обект за производство и търговия с храни;
- Заявление за регистрация или промяна на обстоятелства по регистрация на обекти и оператори във фуражния сектор;
- Удостоверение за регистрация на обекти и оператори, които не задържат фуражи на склад;
- Заявление за регистрация на превозвач, транспортиращ фуражи;
- Удостоверение за регистрация на превозвач, транспортиращ фуражи;
- Заявление за издаване на разрешение за търговия на едро, търговия на дребно и преупаковане на продукти за растителна защита по чл. 23 от Закона за растителна защита;
- Разрешение за търговия на едро, търговия на дребно и преупаковане на продукти за растителна защита по чл. 23 от Закона за растителна защита.

## 8 Заключение

В настоящата теза беше представен подход, с който чрез използване на съвременни технологии и стратегии за програмиране е решен проблем, който е ключов за предоставянето на електронни административни услуги и осигуряването на достъпност до електронно обслужване. Чрез него се постига удовлетворяване на изискванията на крайния потребител за приложение с оптимизирана производителност, богат и интерактивен интерфейс и в същото време предсказуемост и унифициран подход при работа с електронни структурирани документи.

Чрез прилагането на добри практики и платформи от техническа гледна точка, продуктът предоставя лесен метод за поддръжка, разширяемост и преизползване на програмни единици. Реализацията, макар и клиентски ориентирана, предлага модулна организация, капсулация и обектно ориентиран подход за постигане на поставените изисквания.

Описаното приложение е внедрено практически и към текущия момент се използва от три администрации в страната.

Като възможности за развитие на продукта биха могли да бъдат реализирани някои функционалности свързани със спецификата на изискванията на отделните администрации, които да предоставят по-добро ниво на параметризация и абстракция. Такива са например по-добър подход за интеграция с приложения, предоставящи конкретни данни за доставчика на електронни услуги. Друг такъв случай е имплементирането на филтри или специфични изисквания за обработка на някои ИО за някои документи и услуги (например ограничаване възможностите за избор на изброени стойности от номенклатура за ИО от тип стойност). Тази нужда възниква поради нормативно дефинирани вътрешни правила в рамките на конкретна администрация.

## 9 Референции

- 1) Обща стратегия на страната – [http://www.mtitc.government.bg/upload/docs/Obshta\\_Strategia\\_eGovernment\\_2011\\_2015.pdf](http://www.mtitc.government.bg/upload/docs/Obshta_Strategia_eGovernment_2011_2015.pdf) -
- 2) Стратегия за Е-управление – <http://www.mtitc.government.bg/upload/docs/eGovStrategy.pdf>
- 3) ЗЕУ – <http://lex.bg/bg/laws/ldoc/2135555445> -
- 4) ЗЕДЕП – <http://lex.bg/laws/ldoc/2135180800> -
- 5) Базисен модел за КАО – <http://www.strategy.bg/Publications/View.aspx?lang=bg-BG&categoryId=155&y&m&d> –
- 6) Оперативна съвместимост – <http://www.lex.bg/bg/laws/ldoc/2135606737>
- 7) РОС – <http://egov.bg/ereg-public/home.rg>
- 8) НРИОЕУ – <http://www.lex.bg/bg/laws/ldoc/2135588749>
- 9) Редактор (desktop) - <http://eimoti.egov.bg/>
- 10) Редактор (онлайн)-СЗ - <http://egateway.starazagora.bg/service.aspx?FormID={AD4B0803-AE9C-4CEC-8AE2-62382BB126FC}>
- 11) SPA – <http://www.johnpapa.net/pageinspa/>
- 12) SPA – Michael S. Mikowski and Josh C. Powell: „Single Page Web Applications: JavaScript end-to-end“
- 13) SPA – <http://singlepageappbook.com/goal.html>
- 14) Revealing Prototype Pattern – <http://addyosmani.com/resources/essentialjsdesignpatterns/book/>
- 15) JSON.stringify– [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/JSON/stringify](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify)
- 16) OData – <http://www.odata.org/>
- 17) Breeze – <http://www.breezejs.com/documentation/introduction>
- 18) Julie Lerman – <http://msdn.microsoft.com/en-us/magazine/dn201742.aspx>
- 19) Durandal – <http://durandaljs.com/>
- 20) За проекта – <http://www.government.bg/cgi-bin/e-cms/vis/vis.pl?s=001&p=0235&n=236&q>

## 10 Приложения

### 10.1 Пример за XSD дефиниция за структуриран документ

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema targetNamespace="http://ereg.egov.bg/segment/0009-000146"
  xmlns="http://ereg.egov.bg/segment/0009-000146"
  xmlns:eash="http://ereg.egov.bg/segment/0009-000152"
  xmlns:stbt="http://ereg.egov.bg/value/0008-000143"
  xmlns:sard="http://ereg.egov.bg/segment/0009-000141"
  xmlns:easf="http://ereg.egov.bg/segment/0009-000153"
  xmlns:ad="http://ereg.egov.bg/segment/0009-000139"
  xmlns:pbd="http://ereg.egov.bg/segment/0009-000008"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

  <xsd:import namespace="http://ereg.egov.bg/segment/0009-000152" schemaLocation="0009-000152_ElectronicAdministrativeServiceHeader.xsd"/>
  <xsd:import namespace="http://ereg.egov.bg/value/0008-000143" schemaLocation="0008-000143_ServiceTermType.xsd"/>
  <xsd:import namespace="http://ereg.egov.bg/segment/0009-000141" schemaLocation="0009-000141_ServiceApplicantReceiptData.xsd"/>
  <xsd:import namespace="http://ereg.egov.bg/segment/0009-000153" schemaLocation="0009-000153_ElectronicAdministrativeServiceFooter.xsd"/>
  <xsd:import namespace="http://ereg.egov.bg/segment/0009-000139" schemaLocation="0009-000139_AttachedDocument.xsd"/>
  <xsd:import namespace="http://ereg.egov.bg/segment/0009-000008" schemaLocation="0009-000008_PersonBasicData.xsd"/>

  <xsd:element name="PermanentAddressCertificateApplication"
    type="PermanentAddressCertificateApplication"/>
  <xsd:complexType name="PermanentAddressCertificateApplication">
    <xsd:annotation>
      <xsd:documentation xml:lang="bg">Заявление за издаване на удостоверение за постоянен адрес</xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="ElectronicAdministrativeServiceHeader"
        type="eash:ElectronicAdministrativeServiceHeader" minOccurs="0"/>
      <xsd:element name="ServiceTermType" type="stbt:ServiceTermType" minOccurs="0"/>
      <xsd:element name="ServiceApplicantReceiptData"
        type="sard:ServiceApplicantReceiptData" minOccurs="0"/>
      <xsd:element name="ApplicationSubject" type="pbd:PersonBasicData" minOccurs="0"/>
      <xsd:element name="AttachedDocuments" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="AttachedDocument" type="ad:AttachedDocument"
              maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="ElectronicAdministrativeServiceFooter"
        type="easf:ElectronicAdministrativeServiceFooter" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

## 10.2 Пример за XML съдържание на документ

```
<?xml version='1.0' encoding='utf-8' ?>
<PermanentAddressCertificateApplication xmlns='http://ereg.egov.bg/segment/0009-000146'
xmlns:xsd='http://www.w3.org/2001/XMLSchema' xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance'>
  <ElectronicAdministrativeServiceHeader>
    <SUNAUServiceURI xmlns='http://ereg.egov.bg/segment/0009-000152'>0011-
000010</SUNAUServiceURI>
    <DocumentTypeURI xmlns='http://ereg.egov.bg/segment/0009-000152'>
    <RegisterIndex xmlns='http://ereg.egov.bg/segment/0009-
000022'>10</RegisterIndex>
    <BatchNumber xmlns='http://ereg.egov.bg/segment/0009-
000022'>7</BatchNumber>
    </DocumentTypeURI>
    <DocumentTypeName xmlns='http://ereg.egov.bg/segment/0009-000152'>Заявление за
издаване на удостоверение за постоянен адрес</DocumentTypeName>
    <ElectronicServiceProviderBasicData xmlns='http://ereg.egov.bg/segment/0009-
000152'>
      <EntityBasicData xmlns='http://ereg.egov.bg/segment/0009-000002'>
        <Name xmlns='http://ereg.egov.bg/segment/0009-000013'>Доставчик
на електронна административна услуга</Name>
        <Identifier xmlns='http://ereg.egov.bg/segment/0009-
000013'>XXXXXXXXXX</Identifier>
      </EntityBasicData>
      <ElectronicServiceProviderType xmlns='http://ereg.egov.bg/segment/0009-
000002'>0006-000031</ElectronicServiceProviderType>
      </ElectronicServiceProviderBasicData>
      <ElectronicServiceApplicant xmlns='http://ereg.egov.bg/segment/0009-000152'>
        <RecipientGroup xmlns='http://ereg.egov.bg/segment/0009-000016'>
          <Author>
            <Person xmlns='http://ereg.egov.bg/segment/0009-000012'>
              <Names xmlns='http://ereg.egov.bg/segment/0009-
000008'>
                <First
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иван</First>
                <Middle
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Middle>
                <Last
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Last>
              </Names>
              <Identifier
xmlns='http://ereg.egov.bg/segment/0009-000008'>
                <EGN
xmlns='http://ereg.egov.bg/segment/0009-000006'>6101047500</EGN>
              </Identifier>
            </Person>
          </Author>
          <Recipient>
            <Person xmlns='http://ereg.egov.bg/segment/0009-000015'>
              <Names xmlns='http://ereg.egov.bg/segment/0009-
000008'>
                <First
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иван</First>
                <Middle
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Middle>
                <Last
xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Last>
              </Names>
```



```

<Identifier
xmlns='http://ereg.egov.bg/segment/0009-000008'>
    <EGN
xmlns='http://ereg.egov.bg/segment/0009-000006'>6101047500</EGN>
    </Identifier>
</Person>
</Recipient>
</RecipientGroup>
<EmailAddress xmlns='http://ereg.egov.bg/segment/0009-
000016'>ivan.ivanov@email.bg</EmailAddress>
</ElectronicServiceApplicant>
<ElectronicServiceApplicantContactData xmlns='http://ereg.egov.bg/segment/0009-
000152'>
    <DistrictCode xmlns='http://ereg.egov.bg/segment/0009-
000137'>BGS</DistrictCode>
    <DistrictName xmlns='http://ereg.egov.bg/segment/0009-
000137'>България</DistrictName>
    <MunicipalityCode xmlns='http://ereg.egov.bg/segment/0009-
000137'>BGS04</MunicipalityCode>
    <MunicipalityName xmlns='http://ereg.egov.bg/segment/0009-
000137'>България</MunicipalityName>
    <SettlementCode xmlns='http://ereg.egov.bg/segment/0009-
000137'>07079</SettlementCode>
    <SettlementName xmlns='http://ereg.egov.bg/segment/0009-
000137'>България</SettlementName>
</ElectronicServiceApplicantContactData>
<ApplicationType xmlns='http://ereg.egov.bg/segment/0009-000152'>0006-
000121</ApplicationType>
<SUNAUServiceName xmlns='http://ereg.egov.bg/segment/0009-000152'>Издаване на
удостоверение за постоянен адрес</SUNAUServiceName>
<SendApplicationWithReceiptAcknowledgedMessage
xmlns='http://ereg.egov.bg/segment/0009-
000152'>>false</SendApplicationWithReceiptAcknowledgedMessage>
</ElectronicAdministrativeServiceHeader>
<ServiceTermType>0006-000083</ServiceTermType>
<ServiceApplicantReceiptData>
    <ServiceResultReceiptMethod xmlns='http://ereg.egov.bg/segment/0009-
000141'>0006-000076</ServiceResultReceiptMethod>
</ServiceApplicantReceiptData>
<ApplicationSubject>
    <Names xmlns='http://ereg.egov.bg/segment/0009-000008'>
        <First xmlns='http://ereg.egov.bg/segment/0009-000005'>Иван</First>
        <Middle xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Middle>
        <Last xmlns='http://ereg.egov.bg/segment/0009-000005'>Иванов</Last>
    </Names>
    <Identifier xmlns='http://ereg.egov.bg/segment/0009-000008'>
        <EGN xmlns='http://ereg.egov.bg/segment/0009-000006'>6101047500</EGN>
    </Identifier>
</ApplicationSubject>
<ElectronicAdministrativeServiceFooter>
    <ApplicationSigningTime xmlns='http://ereg.egov.bg/segment/0009-000153'>2014-03-
06T21:23:18.005Z</ApplicationSigningTime>
    <XMLDigitalSignature xmlns='http://ereg.egov.bg/segment/0009-000153'>/>
</ElectronicAdministrativeServiceFooter>
</PermanentAddressCertificateApplication>

```

### 10.3 Пример за генериран документ с грешки, възникнали при валидация

```
<?xml version='1.0' encoding='utf-8' ?>
<RegisteredErrorsInDocumentContent xmlns='http://ereg.egov.bg/segment/0009-000146'
xmlns:xsd='http://www.w3.org/2001/XMLSchema' xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance'>
  <DocumentTypeURI xmlns='http://ereg.egov.bg/segment/0009-000025'>
    <RegisterIndex xmlns='http://ereg.egov.bg/segment/0009-
000022'>10</RegisterIndex>
    <BatchNumber xmlns='http://ereg.egov.bg/segment/0009-000022'>7</BatchNumber>
  </DocumentTypeURI>
  <DocumentTypeName xmlns='http://ereg.egov.bg/segment/0009-000025'>Заявление за издаване
на удостоверение за постоянен адрес</DocumentTypeName>
  <RegisteredErrors xmlns='http://ereg.egov.bg/segment/0009-000025'>
    <Error>
      <TermURI xmlns='http://ereg.egov.bg/segment/0009-000024'>
        <RegisterIndex xmlns='http://ereg.egov.bg/segment/0009-
000022'>6</RegisterIndex>
        <BatchNumber xmlns='http://ereg.egov.bg/segment/0009-
000022'>24</BatchNumber>
      </TermURI>
      <ErrorDescription xmlns='http://ereg.egov.bg/segment/0009-
000024'>Невалиден "ЕИК/БУЛСТАТ".</ErrorDescription>
    </Error>
    <Error>
      <TermURI xmlns='http://ereg.egov.bg/segment/0009-000024'>
        <RegisterIndex xmlns='http://ereg.egov.bg/segment/0009-
000022'>6</RegisterIndex>
        <BatchNumber xmlns='http://ereg.egov.bg/segment/0009-
000022'>69</BatchNumber>
      </TermURI>
      <ErrorDescription xmlns='http://ereg.egov.bg/segment/0009-
000024'>Невалидна структура на обекта съгласно XML дефиницията му, вписана в регистъра на
информационните обекти.</ErrorDescription>
    </Error>
  </RegisteredErrors>
</RegisteredErrorsInDocumentContent>
```

### 10.4 Описание на съдържанието на приложения компакт диск

В приложения компакт диск е записан цялостната имплементация на описаното решение. В него е реализирана функционалността за работа с примерния електронен структуриран документ, както и още някои публично достъпни документи („Потвърждаване за получаване” и „Съобщение, че получаването не се потвърждава”). След стартиране на приложението се зарежда страница с изброени вариантите за избор на документ и режим, в който да бъде стартирано приложението. Тази страница представлява входната точка за стартиране на редактора (за цел демонстриране функционалността на приложението). В реално прилагане тя е заменена от конкретна имплементация в зависимост от бизнес логиката на извикващата система.