



**УНИВЕРСИТЕТ ПО БИБЛИОТЕКОЗНАНИЕ И ИНФОРМАЦИОННИ
ТЕХНОЛОГИИ**

**КАТЕДРА „ИНФОРМАЦИОННИ СИСТЕМИ И ТЕХНОЛОГИИ”
МАГИСТЪРСКА ПРОГРАМА
„ИНФОРМАЦИОННИ ТЕХНОЛОГИИ”**

МАГИСТЪРСКА ТЕЗА

на тема:

Модерни JavaScript приложения с AngularJS

Дипломант:

Научен ръководител:.....

София
2016

РЕЗЮМЕ

Димчевска, Елеонора. Модерни JavaScript приложения с AngularJS. Научен ръководител проф. Иван Иванов. 2016, София. Катедра „Информационни системи и технологии“, Факултет по Информационни науки, УНИБИТ. 109 с. Брой източници – 23. Брой приложения – 10.

Цел на магистърската теза е да се проектира и разработи JavaScript уеб приложение с AngularJS, следвайки новите тенденции в областта. Да се опишат похвати за оптимизиране и организиране на кода и средата за разработка. Създаване на тестове още на ниво код, предпазващи от потенциални грешки при разрастване на приложението и генериране на документация.

Ключови думи: уеб приложение, JavaScript, AngularJS, SPA, GruntJS, NPM, unit test, Karma, Jasmine.

СЪДЪРЖАНИЕ

РЕЗЮМЕ	2
СЪДЪРЖАНИЕ	3
УВОД.....	6
1. Глава първа Развитие на JavaScript като технология и приложение	8
1.1. Развитие на JavaScript като технология и възможности	8
1.2. Въведение в модерните уеб приложения с JavaScript	11
2. Глава втора Анализ на технологиите	14
2.1. AngularJS - същност и приложение	14
2.2. Структурно разделяне на кода според функционалността при AngularJS.....	16
2.3. Организиране на среда за разработка - средства и похвати	18
2.3.1. Среда за разработка с NodeJS и NPM	19
2.3.2. Оптимизация на кода с GruntJS - същност и значимост	21
2.4. Предимствата на CSS препроцесорите като SASS	23
2.5. Възможност за тестване на JavaScript код с Karma и Jasmine	25
3. Глава трета Проектиране на приложението	29
3.1. Планиране на процеса	29
3.2. Описване на изискванията.....	31
3.3. Планиране и изготвяне на архитектура и дизайн.....	32
4. Глава четвърта Разработка на проекта.....	38
4.1. Настройване на средата за разработка	38
4.1.1. Конфигуриране на проекта с package.json.....	38

4.1.2.	Конфигурация за оптимизиране с GruntJS	40
4.1.2.1.	Оптимизиране на броя JavaScript файлове, обединявайки ги в един чрез <i>grunt-contrib-concat</i>	40
4.1.2.2.	Проверка за коректно написан JavaScript чрез <i>grunt-contrib-jshint</i>	43
4.1.2.3.	Намаляване размера на JavaScript файловете чрез <i>grunt-contrib-uglify</i>	45
4.1.2.4.	Оптимизиране на процесите чрез <i>grunt-contrib-watch</i>	46
4.1.3.	Конфигурация за стартиране на проекта	47
4.2.	Разработка на проекта	49
4.2.1.	Създаване на архитектурата и разделяне на модули	50
4.2.2.	Функционалност за управление на рецептите - <i>cb-recipe</i>	55
4.2.3.	Функционалност на модул за изглед на единична рецепта - <i>cb-singleView</i>	69
4.2.4.	Функционалност за търсене - <i>cb-search</i>	71
4.3.	Създаване на тестове за приложението	75
4.3.1.	Конфигурация	75
4.3.2.	Методика за съхраняване на файловете	78
4.3.3.	Създаване на тестове	80
4.4.	Създаване на документация за приложението	87
	Заклучение	92
	ИЗПОЛЗВАНА ЛИТЕРАТУРА	94
	ПРИЛОЖЕНИЕ 1 – Финален вид на <i>package.json</i>	96

ПРИЛОЖЕНИЕ 2 – Финален вид на Gruntfile.js	97
ПРИЛОЖЕНИЕ 3 – Екран добавяне на рецепта от разработеното приложение	100
ПРИЛОЖЕНИЕ 4 - Екран редактиране на рецепта от разработеното приложение	101
ПРИЛОЖЕНИЕ 5 – Екран изтриване на рецепта от разработеното приложение	102
ПРИЛОЖЕНИЕ 6 – Екран показване на грешка от разработеното приложение	103
ПРИЛОЖЕНИЕ 7 – Екран преглед на рецепта от разработеното приложение	104
ПРИЛОЖЕНИЕ 8 - Екран търсене на рецепта от разработеното приложение	105
ПРИЛОЖЕНИЕ 9 – Финален вид на karma.conf.js	106
ПРИЛОЖЕНИЕ 10 – Екран от генерираната документация от разработеното приложение	108
Списък на използваните съкращения.....	109

УВОД

Програмирането за уеб среда е изминало дълъг път от първия появил се HTML документ през 1991 г. Вече има много по интерактивно и визуално презентирано съдържание, благодарение на развитието на технологиите CSS и JavaScript.

В същото време развитието на хардуера позволява все по-широката употреба на уеб технологиите, които стават неделима част от ежедневието ни. Човек днес може да следи трафика през мобилни приложения и дори да приготви сутрешното си кафе чрез мобилния телефон [18].

Всичко това позволява изработването на приложение да не зависи от една конкретна технология. Вече е възможно само с използването на HTML и JavaScript да се разработи приложение за компютър или мобилен телефон.

Новите тенденции за разработване на приложения в интернет пространството са за използване именно на тези технологии. Те позволяват част от логиката да се изнесе от сървърната страна и да се прехвърли в брауъра на клиентската машина. Базирайки се на тази тенденция, в настоящия проект ще се разгледа как се реализира едно модерно уеб приложение. То ще бъде JavaScript базирано, като ще се използват последните иновации в сферата. Ще се използва едно от най-популярните средства за разработване на подобен тип приложение – AngularJS, което е разработка на софтуерния гигант Google. Както и ще се покажат способности за управление на външни ресурси с NodeJS и Node package manager (NPM), различни средства за спомагане процеса на разработка - от оптимизиране на процесите с GruntJS до изготвяне на тестове за създадения код и документация.

Самото приложение ще представлява интернет версия на книга за рецепти. Ще работи изцяло в клиентска среда, в браузер, без сървърна част. Това означава, че ще може да бъде изградена в последствие връзка със всяка сървърна технология, както и при необходимост да се превърне в мобилно

приложение. И всичко това с използването на вече съществуващият код, без писането на нов за специфична среда или платформа.

Функционално то ще предоставя възможност на потребителя по лесен и достъпен начин да въвежда нови рецепти, да преглежда списък на вече създадени, да вижда пълната информация за избраната от него рецепта, да редактира или трие вече съществуваща рецепта, както и да търси конкретна рецепта, чрез възможност за търсене по съставки. По този начин ще се покаже потенциала, силата и достъпния начин, по който може да се създаде JavaScript уеб базирано приложение, което в същото време ще е интуитивно и практично за употреба.

1. Глава първа

Развитие на JavaScript като технология и приложение

1.1. Развитие на JavaScript като технология и възможности

Програмният език JavaScript е бил създаден през 1995г от Брендан Ейк (Brendan Eich), за разработване на динамично съдържание и работа в браузър [13]. Първоначално е бил създаден за нуждите на браузъра на компанията Netscape, но бързо набира популярност. Постепенно се появява и стандарт на езика известен като ECMAScript. Благодарение на това, езика започва да се развива и подобрява и днес вече имаме версия ECMAScript6, с официално наименование ECMAScript2015, с която се въвеждат доста от правилата познати от традиционните обектно ориентирани езици за програмиране.

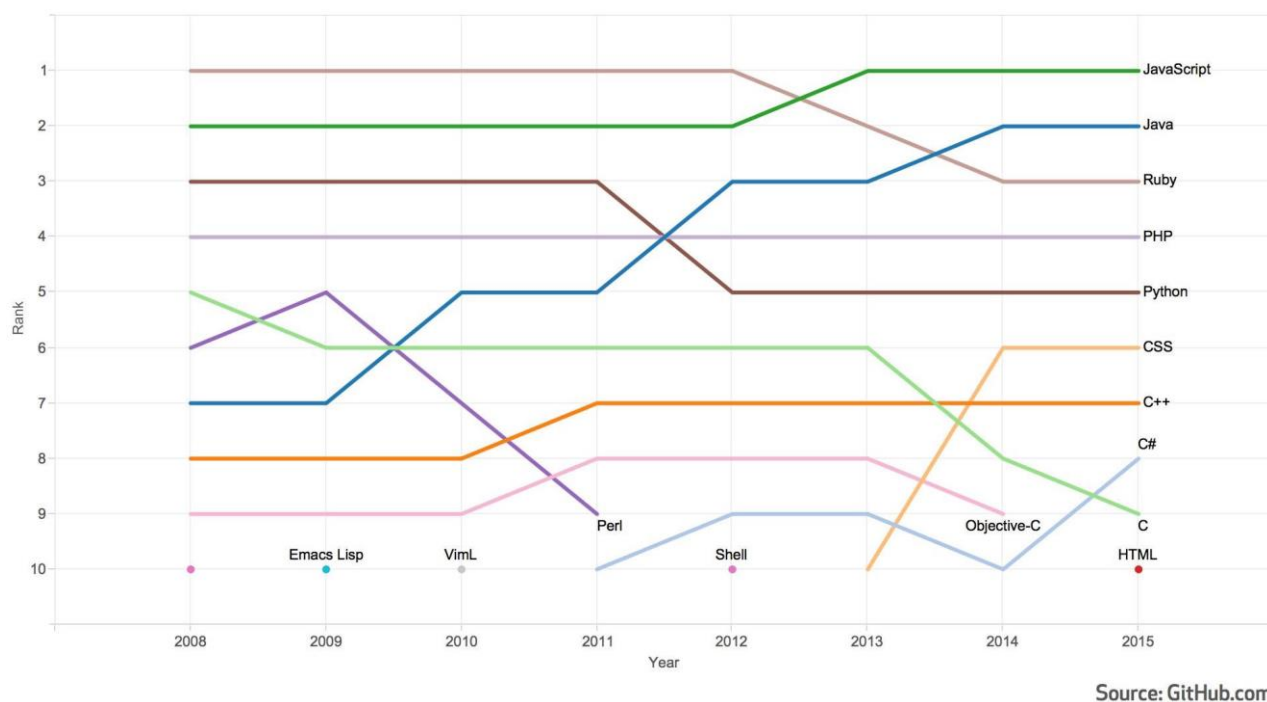
Причина за бързо нарасналата популярност на JavaScript е, че той е доста лесен и начинаещ разработчик може да започне с него с малко усилия, за разлика от традиционните езици за програмиране, като Java или C++. Освен това той се изпълнява в браузър, не са нужни компилатори или нещо друго за да бъде използван. Заедно с това и самата уеб среда започва бързо да се развива, което увеличава и популярността на езика. Постепенно се появяват различни библиотеки написани изцяло на JavaScript, които предоставят по кратки и лесни методи за писане на вече познати операции. Така например появата на библиотеката JQuery през 2006 г., позволи на уеб разработчиците да създават динамично съдържание много по-лесно и то без да се притесняват от различния начин, по който някои браузъри работят с него.

С нарастващата популярност на JavaScript се увеличава и разнообразието и начините му на употреба. Създават се приложения изцяло реализирани само с тази технология. Заедно с това се появява софтуер, така наречения фреймуърк (от английската дума *framework*), който позволява по-лесно реализиране на архитектурата на такъв тип проекти. Един от най-популярните фреймуъркси, AngularJS, се появява през 2012 г. и е разработка на

софтуерния гигант Google [23]. А с последните развития както на AngularJS така и на други подобни на него фреймуъркси, вече става възможно кода, с който е разработено едно стандартно уеб приложение, то да се превърне в мобилно, без да се налага пренаписване на неговия код за специфична мобилна среда.

Паралелно с употребата на JavaScript в браузър се появява възможност за използването му и на сървърна среда. През 2009 г. се появява и първата разработена среда за стартиране на JavaScript на сървър, наречена NodeJS. Благодарение на нея вече има и сървъри работещи изцяло на JavaScript. Първоначално тази среда работи само на Линукс (*Linux*) машини, но от 2011 г. вече може да се използва и на Уиндоус (*Windows*) [21].

С развитието на самия език нараства и неговата популярност, започва да се появяват статистики за неговата употреба. Една от най-големите онлайн услуги за съхраняване и управление на проекти, GitHub, през 2015 г. представи статистика за най-използвания език за програмиране. Резултатите са базирани на всички публични и частни проекти, които се съхраняват в GitHub и анализират езика, на който са разработени. Според резултатите най-много проекти от 2012 г. до момента на създаване на статистиката се създават на JavaScript, като между 2008 г. и 2012 г. той е втория най-популярен език за разработване на проекти [5]. (Фиг. 1)



Фиг. 1

Сферите на употреба и стабилния ръст в популярност на технологията може да се проследят и от статистиките на едни от най-популярните сайтове сред разработчиците - Stack Overflow. Това е сайт предназначен за разработчици от цял свят, които да имат възможност да споделят помежду си информация за проблем, който срещат при използване на дадена технология под формата на задаване и отговаряне на въпроси. В началото на всяка година, се публикува статистика базирана на допитване до неговите потребители и обхваща всички сфери свързани с разработването на софтуерни продукти. В публикуваната през март 2016 г. статистика се наблюдават следните тенденции [20]:

- JavaScript е най-популярния език за разработка, като поддържа челно място от 2014 г. до днешни дни;
- Разработчиците използват средно между 3 до 4 различни програмни езици, технологии и фреймуъркси, като JavaScript е сред най-честото комбинирани технологии;
- Според сферата, в която разработват програмистите, JavaScript се нарежда в челни позиции дори и при програмисти разработващи за сървърна среда (*Back-End* програмисти), както и тези, които

разработват както на клиентска така и на сървърна среда (*Full-Stack* програмисти)

1.2. Въведение в модерните уеб приложения с JavaScript

В днешно време употребата на JavaScript далеч не се изчерпва само до това да се направи динамично съдържание на една страница. Вече има приложения базирани изцяло на него, изпълняващи се не само на уеб среда. Това са така наречените *Single Page Application (SPA)* или едностранцови приложения и представляват стратегия за изграждане на уеб апликации или уеб сайтове.

Стандартната стратегия за изграждане на сайтове разчита на сървър да подава към клиента (браузър) поискана страница и съответните ресурси, т.е. създават страниците като обединяват и сглобяват HTML структурата с данните на сървърно ниво и изпращат до браузъра (клиента) вече сглобената страница. Това са т.нар. многостраницови приложения (*multi page apps*). При тях, със всяко ползване на навигацията на дадена страница, води до това, че цялото съдържание се презарежда. Концепцията за едностранцови приложения изнася тази логика изцяло в клиентската част. Както говори самото име, в такъв тип приложения няма различни страници, а само една. Към браузъра на потребителя се подава само една HTML страница и с нея се зареждат всички останали ресурси, необходими му за стартиране на сайта или приложението. Начина, по който се визуализира информацията на потребителя в едно SPA приложение е посредством:

- HTML фрагменти наречени *Views*;
- концепция за навигиране между HTML фрагменти наречена маршрутизация (*routing*) чрез услуга (способ) позната като *router*;
- механизъм за работа с данни;

В SPA приложенията няма презареждане на страниците при всяко навигиране на потребителя из менюто. Зарежда се само частта, която е поискана. За да се постигне това, всяка нова показана информация

представлява малък фрагмент от основната страница на самото приложение. Те се наричат *View*, в буквален превод изглед. Това е събирателен термин, за това което вижда потребителя. Концепция за изграждането им е спрямо модела на приложението и следи визуалното да се представя актуална информация. Декларативния начин, по-който се следва логиката на модела и осъществява презентацията в *View* е посредством използването на шаблони (*template*) [2]. Основната страница може да бъде изградена от отделни елементи, създадени от различни *Views*. Така например навигацията на едно такова приложение може да представлява *View*, а самото съдържание да е друго *View*.

За да може да се преминава от едно *View* към друго, подобно на стандартните приложения от една страница към друга, се използва способ *router*. Той позволява използването на навигация, познато от стандартните приложения, както от самото меню на приложението, така използвайки и навигацията на самия браузър. Посредством него и при SPA приложение браузъра може да пази история за посетените „страници“, да се навигира между тях чрез навигацията на браузъра (бутоните за напред и назад) или чрез директно въвеждане на адрес в навигационната лента на браузъра. Предимството тук е, че по този начин ще се зареди информация и необходимите ресурси само за поискания фрагмент (*View*), а не цялата страница. Обработката на навигацията се случва още в клиента, а не от сървъра.

В приложение използващо принципите на SPA, основната част от него се изпълнява при клиента, например показване на *Views* и навигиране между тях. Тези механизми сами по себе си обаче не показват информация на потребителя, а представляват структура, в която да се организира информацията. Тук идва ролята на данните. Те се обработват и подават от сървъра, фрагментите ги получават и показват от шаблона, с който разполагат. Данните обаче трябва да имат определен вид, за да може да бъдат показани правилно. Това означава правилно изградена структура между двете части на приложението (сървърна и клиентска). Трябва да се създаде така наречения приложно-програмният интерфейс (на английски: *Application Programming*

Interface, API), който се използва за създаване на дефиниции, протоколи и инструменти за лесно изграждане на приложение [21]. Чрез обръщане към приложно-програмният интерфейс (*API Calls*) се подават или получават нужните данни. Най-често данните се подават от сървъра във JSON формат, но може да бъдат други HTML фрагменти, както и специфични данни за конкретното *View*, JavaScript или CSS.

Тази стратегия за изграждане на сайтове или приложения (SPA) се нуждае и от архитектурен модел за организация на всичките различни файлове и ресурси. Повечето приложения работещи на една страница са със структури от типа *Model-View-Controller* (MVC). Този модел е анонсиран през 1970 г. и е станал популярен в приложенията писани на C++, Java, Objective-C. Това, което се крие зад тази концепция е ясно разделение на данните от самия код (модел от английската дума *model*), логиката на приложението (контролер от английски *controller*) и презентацията на информацията и данните (*view*) [16]. В последствие се развиват и разновидности на тази концепция като *Model-View-Presenter* (MVP) и *Model-View-View-Model* (MVVM). MVP е вариация на MVC с фокус върху презентационна логика, а MVVM е базирано на предните два модела и се опитва да даде по-ясно разграничение на разработването на потребителски интерфейси (*UI*) от бизнес-логика и поведение на дадено приложение.

2. Глава втора

Анализ на технологиите

2.1. AngularJS - същност и приложение

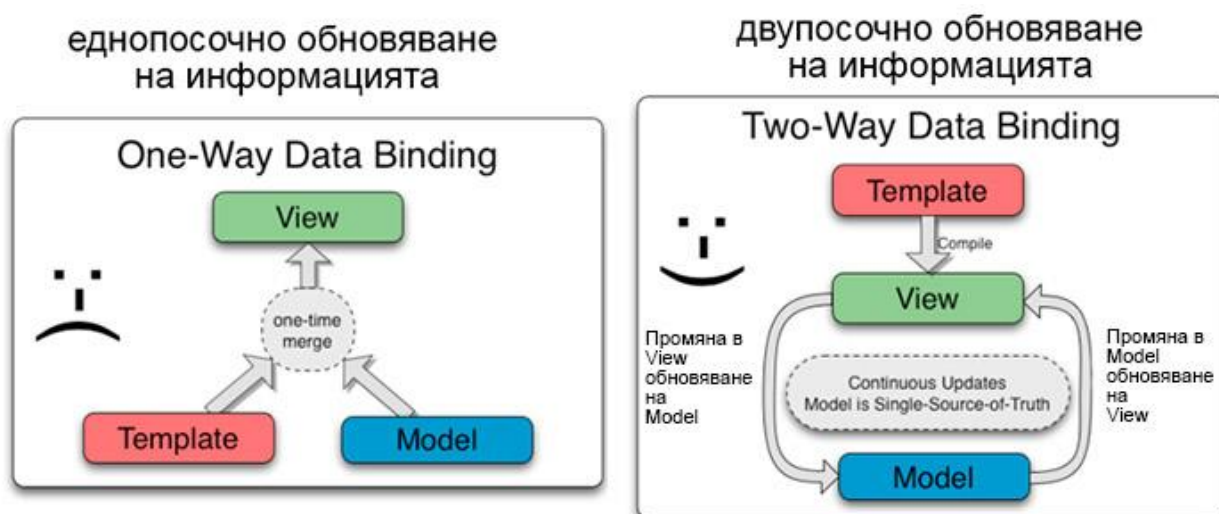
Създаването на SPA приложение, без помощта на библиотеки или фреймуърк, може да е доста трудна задача. В днешно време вече има голямо разнообразие и решението, кое точно да се избере може да варира според типа продукт, типа на нужния архитектурен модел, от личните предпочитания на програмиста и дори нивото на владение на JavaScript.

Един от най-популярните и бързо развиващи се фреймуъркси е *AngularJS*. Той е JavaScript фреймуърк, който позволява лесно създаване на SPA приложения. Използването му позволява да се избере, кой точно от вече разгледаните архитектурни модели, ще е по-подходящ за приложението, което ще се разработва. Затова и неговия модел е кръстен *Model-View-Whatever* (MVW), където последната част *Whatever* идва от английското *Whatever Works for You* (каквото работи за вас) [16]. Често разработчиците го предпочитат и защото не са нужни солидни познания по самия фреймуърк, за да може да почне да се работи с него, за разлика от други подобни, а функционалността, която предлага, е доста обширна и подходяща и за сложни приложения.

Самият AngularJS борави с HTML фрагментите (Views), подобно на стандартен SPA фреймуърк. Презентирането на информацията се осъществява посредством използването на шаблон (темплейт - на английски *template*). Шаблоните и данните (*data*), се пращат на браузъра и там се сглобяват, подход типичен за SPA. Ролята на сървъра е само да изпраща статичните ресурси за шаблоните и необходимите данни за него. Това е една от основите в концепцията на AngularJS, сглобяването на информацията става на ниво клиент (*Client-Side*) [3].

Основно понятие, което стои зад структурната концепция на AngularJS е *Data Binding*, т.е. вграждане на данните в структурата. В една стандартна JavaScript страница, в която трябва да се обновява информация,

трябва в кода да се направи низ със HTML структурата на елемента и в него динамично да се добавя новата информация. В повечето случаи това означава, че структурата на страницата, известна като *DOM* (английската аббревиатура за модела на дървовидна структура на документа в Интернет - *Document Object Model*), непрекъснато ще се изтрива и добавя наново, но с новата информация. Обновяването само на един единствен елемент, за който е дошла нова информация предизвиква обновяване на цялата структура. Посоката на обновяване обикновено е еднопосочен, само при действие от страна на потребителя. Това, което се случва с метода на Data Binding в AngularJS, е да се обновява само информацията, за която е нужно в момента, без да се обновява цялото DOM дърво. Това е така нареченото двупосочно обновяване на информацията, след действие от страна на потребителя (промяна на View) или след промяна на самата информация от страна на модела (Model) (Фиг. 2.) [6]

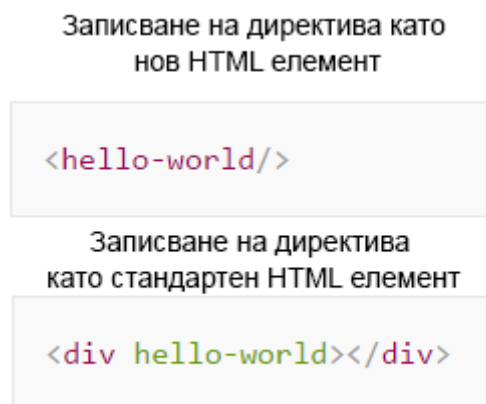


Фиг. 2 [6]

Това, което е ново в AngularJS е използването на готови функции наречено *Dependency Injection* (добавяне на зависимости). Основната идея е, че когато част от кода зависи от друга, тя да може да се извиква лесно и в същото време кода да се запази чист и четим.

Едно от най-често сочените предимства на AngularJS е писането на собствени шаблони наречени директиви (*Directives*). Това са маркери на

елементите в DOM дървото. Те може да са както HTML елемент, атрибут на елемент или клас за стилове (Фиг. 3).



Фиг. 3

AngularJS е създаден за приложения, при които данните са от първостепенно значение. Боравеното с данни и обработката им, независимо дали промяната идва от страна на View или от страна на Model е лесно и почти мигновено. В случай, че приложението, което ще се разработва, ще има повече DOM манипулации, на каквото разчитат повечето стандартни интернет сайтове, не се препоръчва използването на AngularJS. Тогава може би ще бъде по-удачна някоя друга библиотека, като JQuery или подобна.

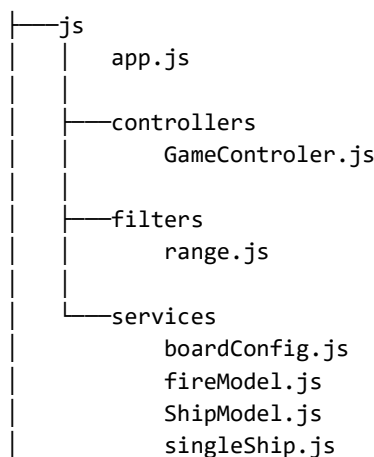
2.2. Структурно разделяне на кода според функционалността при AngularJS

В основата на структурното разделяне седи идеята да може да се пише код, който да е разделен на логически части, за да може да се планира по-детайлно функционалността, както и тестването на кода след това. Структурата на кода трябва да бъде компонентно ориентирано. Това е ключа към писането на качествен продукт (софтуер). Начина, по който се постига това в AngularJS, е чрез писане на компоненти, които може да се преизползват и се наричат модули (*modules*). Две са основните концепции при изграждането на модули [16]:

- разделяне на модулите по слоеве (*modularization by layers*)

- разделяне на модулите по функционалност (modularization by features)

При концепцията за разделяне на модулите по слоеве, идеята е всяка част от функционалността да е в съответната директория. Това може да се види на фигура 4.

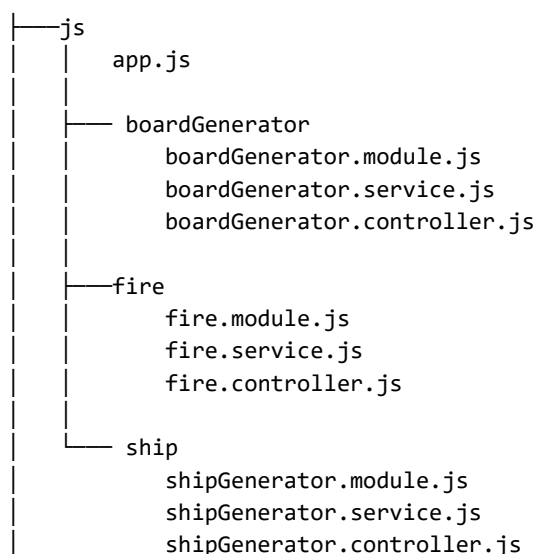


Фиг. 4

Така например, всичко свързано с връзката между данните и шаблоните ще в папка с име *controllers*, функционалностите свързани с обработката на данни ще е в *services* и често повтарящи се операции като например методи за сортиране ще са в *filters*. Всичко това накрая ще бъде обединено в общ модул от файл наречен *app.js*. Този тип на разделяне на модули се ползва често, заради лесното ориентиране в това файловете в дадена папка какво правят. Така ако се появи грешка или проблем в даден тип компонент, например сървис (*services*), се знае къде да се търси. Когато приложението обаче се разрасне и се усложни, тази структура вече не е практична и удобна. Търсенето на отделните части на една функционалност в различните папки и нуждата от непрекъснато преминаване между папките става неудобно, а също може да бъде и объркващо.

Затова има втора концепция за структуриране на проект наречена модули по функционалност. Тук водеща е функционалността. Разделението няма да е както е показано на фигура 3, а всички файлове свързани с една

конкретна функционалност (controller, module, service, filter) ще са в една папка (Фиг. 5).



Фиг. 5

Тази структура е по-четима и лесна за ориентиране, още преди да се стигне до разглеждане на самите файлове. При промяна или дори изтриване на дадена функционалност са видим всички файлове отговарящи за нея само с един поглед. Въвеждането на нов член в екип, работещ по проект с подобна структура, също се улеснява. Именно тази структура ще се използва в настоящия проект.

2.3. Организиране на среда за разработка - средства и похвати

Всяко JavaScript приложение може да се започне бързо и лесно следвайки стандартния подход чрез ръчно добавяне на всеки нов нужен ресурс, всеки нов създаден скриптов файл, следене за версиите на външните библиотеки или фреймуъркси, следене за последователността на зареждане на всички ресурси и други. Всичко това отнема време и изисква особено внимание и познания, защото може да доведе до сериозни грешки в приложението. Освен това използването на различни похвати за оптимизиране на ресурсите, така че да не са прекалено големи, когато приложението се пусне в експлоатация, за да

работи по-бързо и по-добре, ще е доста трудоемко, отнемащо време и ще се налага да се повтаря след всяка промяна в някой от файловете.

Всичко това може да бъде автоматизирано и програмиста да няма ангажимент да се грижи за тези процеси, а те да се случват на заден фон в процеса на разработка. За тази цел трябва да се подготви среда и да се инсталират правилните приложения и други помощни средства и инструменти. Избора и тук, както при фреймуъркси е голям, но ще се разгледат едни от основните, по-популярните и често използвани:

- Среда за разработка с NodeJS и NPM
- Оптимизация на кода с GruntJS

2.3.1. Среда за разработка с NodeJS и NPM

За да може да се изпълнява JavaScript извън браузера, а директно на компютъра, трябва да се подготви средата. За целта трябва да се инсталира NodeJS. Това е JavaScript базиран сървър, но далеч не означава, че сървърната част на проекта трябва също да е написана на JavaScript. С негова помощ ще може да се организира, управлява и оптимизира целия проект, с лесни команди и настройки. NodeJS разполага с интерфейс с команден ред (*Command-line interface (CLI)*).

Важен инструмент на NodeJS е неговия мениджър на пакети (*NPM*). Благодарение на него може да се организира целия проект, да се изтеглят допълнителни пакети за оптимизиране на средата, пакети за други инструменти ползващи и работещи на NodeJS среда, да поддържа документирана среда за всички използвани пакети, външни ресурси на проекта и команди за оптимизиране. Всичко това се осъществява посредством един конфигурационен файл - *package.json*. Както се вижда и от разширението на файла, той също е JavaScript базиран. Разбира се, наличието на *package.json* в проекта не е задължително за неговото функциониране, но е важна част от неговата функционалност.

Конфигурационният файл може да се създаде ръчно, но има и оптимизиран начин, чрез извикване на команда *npm init* в конзолата (стандартната конзола за Windows, MAC OS, Linux или конзола идваща от софтуерната програма за разработване на програмиста - IDE). На екрана ще се появят поредица от въпроси, на които след отговор ще покаже цялата събрана информация за проекта и накрая ще ги запамети в автоматично генериран *package.json* файл. Така генерирания файл е отправна точка за неговата структура и основната информация, която е препоръчително да съдържа. При необходимост той може да бъде редактиран ръчно, както и по преценка на програмиста някои от редовете могат да бъдат изтрети. (Фиг. 6).

```
d:\wamp\www\cooking-book>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (cooking-book) cooking-book
version: (1.0.0)
description: angular cooking book
entry point: (index.js)
test command:
git repository:
keywords:
author: Eleonora Dimchevska
license: (ISC) MIT
About to write to d:\wamp\www\cooking-book\package.json:

{
  "name": "cooking-book",
  "version": "1.0.0",
  "description": "angular cooking book",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Eleonora Dimchevska",
  "license": "MIT"
}

Is this ok? (yes) █
```

Фиг. 6

Също по толкова лесен начин се инсталират и запазват имената на необходимите пакети, т.нар. дипендънсита (от англ. *dependence* - зависимост). С команда от типа *npm install <package_name> --save-dev*, след инсталирането на

пакета, той автоматично се записва в конфигурационния файл с името си и версията си. Всичките файлове на съответния пакет се свалят в директорията на проекта в автоматично генерирана папка с име *node_modules*.

Трябва да се обърне внимание, че може да запазим свалените пакети в две различни графи - *dependencies* и *devDependencies*. В първата графа се пазят неща, без които приложението не може да работи, като допълнителни библиотеки например, а втората - за помощните средства необходими само за средата за разработка (Приложение №1). Голямото предимство в използването на *package.json* е точно в тази част. Запазвайки в един файл имената на пакетите и техните версия, всеки разработчик участващ в писането на проекта знае какво е добавено и коя точно версия е. Така може лесно и да се обнови всеки ресурс или пакет до по-нова версия при необходимост. Друг важен момент е, че всички добавени дивендънсита за нужни само за разработката на приложението, не са нужни при пускането му в експлоатация. Благодарение на конфигурационния файл, всички програми и допълнителни файлове, които се свалят, не е нужно да се пренасят заедно с целия проект. Така се пести място и се избягва прехвърляне на файлове, който не са нужни във всички етапи на разработката. Освен това при използването на система за менажиране на проекти като SVN или GIT, е достатъчно да се съхраняват само файловете на самото приложение, без допълнително свалени пакети и външни ресурси. В момента, в който са нужни с проста команда (*npm install*) могат да се инсталират отново, защото са описани в конфигурационния файл. По този начин се поддържа чиста и оптимизирана среда за разработка.

2.3.2. Оптимизация на кода с GruntJS - същност и значимост

По време на разработката на един проект има много действия, които се повтарят като минимизиране на ресурсите (JavaScript и CSS файловете), проверка на кода за грешки, документиране на кода и много други. За осъществяване на автоматизиране на тези процеси вече съществува голямо разнообразие от инструменти, с които може да се постигне тази цел.

Инструмента, който постави началото и продължава да е от най-ползваните сред разработчиците се нарича *Grunt*. И именно той ще бъде използван в настоящата разработка.

GruntJS е JavaScript базирано приложение наричано *task runner* (стартиране на задачи), което с лесни настройки, създаване на команди с JavaScript, позволява автоматизиране на много от повтарящите се операции в разработката на проекта. Всичко това е изключително полезно и необходимо особено в проекти с AngularJS и подобни на него, където се борави с много на брой JavaScript файлове и оптимизирането е нужно за по-лесното им управление, тестване и проверка на качеството на кода и оптимизиране при пускане в експлоатация.

За да може да извършва всички тези задачи на GruntJS му трябва инсталирана NodeJS среда. Той работи с пакети и тяхното инсталиране става с мениджъра на пакети NPM. С помощта на NPM става и самото инсталиране на GruntJS. Първоначално трябва да се инсталира интерфейса с команден ред на GruntJS (`npm install -g grunt-cli`), като само това се прави глобално за компютъра, на който се разработва проекта. И след това вече се инсталира самия GruntJS (`npm install grunt --save-dev`), като се запазва в конфигурационния файл на проекта, в графата пакети нужни само за разработката му (Приложение №1).

След подготвянето на средата за стартиране на GruntJS, настройването става сравнително лесно. Първо трябва да се изберат пакетите, които са нужни за съответното приложение и да се свалят и инсталират чрез мениджъра на пакети на NodeJS. Следва създаване на конфигурационен файл за самия GruntJS, който по правило се кръщава *Gruntfile.js*. Както си личи по разширението на файла, GruntJS наистина е JavaScript. Следващата стъпка е да се напишат задачите за оптимизациите (Фиг. 7).

```
module.exports = function(grunt) {  
  // Project configuration.  
  grunt.initConfig({
```

```
pkg: grunt.file.readJSON('package.json'),
uglify: {
  my_target: {
    options: {
      sourceMap: true,
      preserveComments: 'some'
    },
    files: {
      'build/js/<%= pkg.name %>.min.js': ['build/js/<%= pkg.name %>.js']
    }
  }
}
});

grunt.loadNpmTasks('grunt-contrib-uglify');

grunt.registerTask('build', ['uglify']);
}
```

Фиг. 7

Всеки пакет има подробна страница с описание, какви настройки може да се добавят за съответната задача. Така всеки програмист може да направи достатъчно точни настройки за конкретния проект.

2.4. Предимствата на CSS препроцесорите като SASS

С развитието на средствата за разработка и архитектурния модел на приложенията се появява и необходимост писането на стилове да следва тези принципи. Появяват се CSS препроцесорите, които позволяват в писането на стиловете да се ползват променливи, използване на функции, блоково разделение и организиране на файловете със стилове според структурата на приложението. Един от най-популярните е SASS (Syntactically Awesome Stylesheets). Той е скриптов език, който се компилира до CSS. Разширението на файловете, които ползват този синтаксис са .scss или .sass [21]. Тези файлове не се ползват директно за зареждане на стиловете, а се компилират до стандартен CSS файл, който да се използва. Има няколко начина за използването му.

Единия вариант е чрез използване на софтуерна програма за разработване (IDE). Голяма част от тях поддържат работа със SASS и не е

нужно друго да се инсталира на машината на разработчика. Друг вариант е да се избере начин на ползване, който не е толкова тясно обвързан с конкретна програма. За целта обаче е необходимо да се инсталират няколко неща последователно, като стъпките и нужните инструменти зависят от операционната система на компютъра на програмиста. За една Уиндоус (Windows) машина първо е необходимо да се инсталира Ruby и след това самия SASS.

Предимствата в този начин на писане на стилове е, че стават структурирани. Използването на променливи в кода позволява после лесно да може да се нанасят промени. Освен това наличието на функции и цикли, познати от програмните езици, позволява по-гъвкаво писане на стилове и преизползване на общи дефиниции (Фиг. 8). Независимо, че крайния резултат винаги е стандартен CSS файл, тази технология помага в процеса на разработка и последващо поддържане и развитие на проекта. По този начин се поддържат чисти стилове и се намалява риска при промяна на някой стил да се допусне грешка или да се изпусне да се промени някой ред.

Използване на променливи в SASS	Крайня резултат в CSS
<pre> \$primary-color: #3bbfce; \$margin: 16px; .content-navigation { border-color: \$primary-color; color: darken(\$primary-color, 10%); } .border { padding: \$margin / 2; margin: \$margin / 2; border-color: \$primary-color; } </pre>	<pre> .content-navigation { border-color: #3bbfce; color: #2b9eab; } .border { padding: 8px; margin: 8px; border-color: #3bbfce; } </pre>

<p>Mixins – блок стилове за преизползване в SASS</p> <pre> @mixin table-base { th { text-align: center; font-weight: bold; } td, th { padding: 2px; } } #data { @include table-base; } </pre>	<pre> #data th { text-align: center; font-weight: bold; } #data td, #data th { padding: 2px; } </pre>
<p>Използване на цикъл в SASS</p> <pre> \$squareCount: 3 @for \$i from 1 through \$squareCount #square-#{ \$i } background-color: red width: 50px * \$i height: 120px / \$i </pre>	<pre> #square-1 { background-color: red; width: 50px; height: 120px; } #square-2 { background-color: red; width: 100px; height: 60px; } #square-3 { background-color: red; width: 150px; height: 40px; } </pre>

Фиг. 8 [21]

2.5. Възможност за тестване на JavaScript код с Karma и Jasmine

Всеки софтуерен продукт трябва да бъде тестван. Това важи и за приложенията и сайтовете написани на JavaScript. Писането на тестове е нужно, за да се намерят съществуващи грешки в приложението. Като по този начин не се гарантира, че грешки липсват, но се повишава качеството и сигурността на готовия продукт. Този вид тестване е познато като тестване на парчета код (*unit test*). Тестовите се пишат от самия програмист и представляват парче код написано специално да тества определена част от приложението, наречена *unit* (единица, парче), от където идва и името. Това далеч не означава, че при наличието на тези тестове няма нужда от тестовите извършвани от инженерите

за проверка на качеството (*quality assurance* - *QA*). Двата вида тестване изпълняват различна роля и се прилагат на различни етапи от разработката на приложението.

Всеки програмист докато пише определена функционалност прави ръчни тестове (*manual tests*) на кода, но те не са достатъчно ефективни, не са структурирани, не могат да се преизползват и обикновено са временно парче код, което после се изтрива. Тук идва ролята на тестовете на отделни парчета. Ползите от този вид тестова са [15]:

- Драстично се намалява броя на грешките в кода
- Подобрява се конструкцията на проекта
- Представяват добра документация на кода
- Намаляват рисковете при промяна в кода да се възпроизведе проблем или грешка
- По време на писането на тестовете, кода може да се рефактурира с цел по-добра изпълнение или производителност
- Съхраняват се заедно с целия код и се изпълняват при всяка нова промяна във функционалността, която тестват

Има два основни подхода при писането на тестовете *Test-driven development (TDD)* и *Behavior-driven development (BDD)*. При първият подход, първо се пишат тестовете, а след това и самия код. Той е по-сложен и отнема повече време, но подобрява архитектурата на кода. Вторият е по-често използван, при него тестовете се правят след като вече конкретната функционалност е написана. Съответно е и по-бърз метод, защото още докато се пише конкретния код се обмисля как да бъде структуриран и кое може да доведе до грешка и после всичко това трябва само да се имплементира (напише) под формата на тестове.

Както за повечето свързани с разработката на JavaScript приложения и тук има голямо разнообразие от среди за пускане на тестове и фреймуъркси за писането им. Една от най-популярните среди за изпълнение е *Karma*. Тя е разработка на екип работещ по AngularJS и е била създадена

първоначално за да тества техния собствен код. Karma е JavaScript базиран инструмент и се изпълнява в конзола [12]. Използва се, за да може да създаде сървър, с който да се стартира проекта и да изпълнят написаните тестове. Работи с NodeJS и затова инсталирането му става през NPM. Отново има глобално инсталиране на интерфейса с команда ред (`npm install -g karma-cli`) и след това на самата Karma (`npm install karma --save-dev`), като тази информация вече се записва в конфигурационния файл на проекта `package.json`.

Това, за което се използва този инструмент, е да може автоматично да се изпълняват тестовете и да се вижда резултата от тях, без да е нужно програмиста да изпълнява всеки един от тях ръчно на всички нужни браузъри. Както повечето инструменти, които автоматизират процеса на разработка и Karma има конфигурационен файл - `karma.conf.js`. Това е JavaScript файл, в който се описват всички нужни конфигурации, за да може тестовете да се изпълнят по желания от програмиста начин. Подобно на създаването на `package.json` и конфигурационния файл на Karma, вместо да се създава ръчно, може да се използва команда `karma init`. В конзолата ще се появят поредица от въпроси и след отговарянето им, файла ще бъде създаден с необходимите настройки. И в този случай генерирания файл е отпавна точна и в последствие всяка конфигурация може да бъде коригирана или да се добави нова (Фиг. 9).

```
d:\wamp\www\cooking-book>karma init
```

```
Which testing framework do you want to use ?
```

```
Press tab to list possible options. Enter to move to the next question.
```

```
> jasmine
```

```
Do you want to use Require.js ?
```

```
This will add Require.js plugin.
```

```
Press tab to list possible options. Enter to move to the next question.
```

```
> no
```

```
Do you want to capture any browsers automatically ?
```

```
Press tab to list possible options. Enter empty string to move to the next question.
```

```
> Chrome
```

```
> Firefox
```

```
> PhantomJS
```

```
>
```

```
What is the location of your source and test files ?
```

```
You can use glob patterns, eg. "js/*.js" or "test/**/*.Spec.js".
```

```
Enter empty string to move to the next question.
```

```
> src/**/*.module.js
> src/**/*.js
> views/**/*.html
> views/*.html
>

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> no

Config file generated at "d:\wamp\www\cooking-book\karma.conf.js".

d:\wamp\www\cooking-book>
```

Фиг. 9

Фреймуърка, който най-често се използва за писане на тестове за AngularJS приложения и работи много добре с Karma е *Jasmine*. Той се използва за писане на BDD тестове на JavaScript. Това, с което Jasmine става по-предпочитан пред останалите, е че съдържа всичко необходимо за писане на тестове, а не работи с различни модули, които трябва да се инсталират допълнително. Неговото инсталиране отново става през NPM. А при инсталацията му също се добавя допълнителния флаг `--save-dev`, за да се запази в конфигурационния файл на самия проект, като ресурс нужен само за разработването.

3. Глава трета

Проектиране на приложението

3.1. Планиране на процеса

Създаването на приложения може да бъде сложна задача, която отнема много време както на програмистите в проекта така и на други членове от екипа. Затова с времето са се обособили различни методики и практики, които улесняват живота на програмистите и в този процес. Общото между всички тях е, че разработката на всеки софтуерен продукт преминава през няколко етапа, а именно [1]:

- Събиране на изискванията за продукта и изготвяне на задание;
- Планиране и изготвяне на архитектура и дизайн;
- Реализация (включва писането на програмен код);
- Преработка и оптимизиране на създаденият код;
- Изпитания на продукта (тестове);
- Внедряване и експлоатация;
- Поддръжка;

Целта на събиране на изискванията и изготвяне на задание е да може да се опише идеята зад проекта. Описва се набор от изисквания, дефиниращи действия от страна на потребителя и компютъра, които в общия случай улесняват извършването на досега съществуващи дейности. Изискванията за продукта обикновено се дефинират под формата на документи. На този етап не се програмира. Изискванията се дефинират от експерти, запознати с проблематиката на конкретната област, които умеят да ги описват в разбираем за програмистите вид. В общия случай тези експерти не са специалисти по програмиране и се наричат бизнес анализатори.

След като изискванията бъдат събрани, идва ред на етапа на планиране. През този етап се съставя технически план за изпълнението на проекта, който описва платформите, технологиите и първоначалната архитектура (дизайн) на програмата. Тази стъпка включва значителна творческа

работа и обикновено се реализира от софтуерни инженери с много голям опит, наричани понякога софтуерни архитекти. Въпреки че, съществуват много правила, спомагащи за правилния анализ и планиране, на този етап се изискват значителна интуиция и усет. Тази стъпка предопределя цялостното по-нататъшно развитие на процеса на разработка.

Етапът, най-тясно свързан с програмирането, е етапът на реализацията (имплементацията). На този етап, съобразно заданието, дизайна и архитектурата на приложението се пристъпва към реализирането му. Етапът "реализация" се изпълнява от програмисти, които пишат програмния код. При малки проекти останалите етапи могат да бъдат много кратки и дори да липсват, но етапът на реализация винаги се извършва.

Важен етап от разработката на софтуер е етапът на изпитания на продукта. Той цели да удостовери, че реализацията следва и покрива изискванията на заданието. Този процес може да се реализира ръчно, но предпочитаният вариант е създаването на автоматизирани тестове, които да реализират проверките. Тестовите са малки програми, които автоматизират, до колкото е възможно, изпитанията. Съществуват парчета функционалност, за които е много трудно да се напишат тестове и поради това процесът на изпитание на продукта включва както автоматизирани, така и ръчни процедури за проверка на функционалността и качеството. Процесът на тестване се реализира от екип инженери по осигуряване на качеството – *quality assurance* (QA) инженери.

Внедряването или инсталирането е процесът на въвеждане на даден софтуерен продукт в експлоатация. Ако продуктът е сложен и обслужва много хора, този процес може да се окаже най-бавният и най-скъпият. За по-малки програми това е относително бърз и безболезнен процес. Най-често се разработва специална програма – инсталатор, която спомага за по-бързата и лесна инсталация на продукта.

В процеса на експлоатация неминуемо се появяват проблеми – заради грешки в самия софтуер или заради неправилното му използване и

конфигурация или най-често заради промени в нуждите на потребителите. Тези проблеми довеждат до невъзможност за решаване на бизнес задачите чрез употреба на продукта и налагат допълнителна намеса от страна на разработчиците и експертите по поддръжката. Процесът по поддръжка обикновено продължава през целия период на експлоатация независимо колко добър е софтуерният продукт. [1]

Така, спазвайки стандартите за процеса на разработка трябва да се изготви план за изработване на приложението, което ще бъде изготвено. Разглеждайки внимателно общите процеси за разработка се вижда, че не всички ще са приложими в този конкретен случай. Така например момента с въвеждане в експлоатация не е много приложим, защото приложението е уеб базирано. Единственото нужно за използване на приложението от страна на потребителя е браузър, а той е неделима част вече от дигиталните устройства (лаптоп, телефон, таблет).

Ще се започне с описване на изисквания и проектиране на архитектурата. Тъй като това е JavaScript базирано приложение и файловете, с който се бори са много, трябва да се предвиди момент, в който да се подготви средата за разработка, за да е оптимизирана от разработчика. След етапа на разработка ще следва етап на писане на програмни тестове (*unit tests*) и създаване на документация на кода.

3.2. Описване на изискванията

За да се създаде какъвто и да е уеб продукт първо трябва да се събере информация за изискванията и да се придобие представа какво ще прави и как ще функционира. На този начален етап на изпълнение, ако трябва да се запишат изискванията за електронния вариант на книга за рецепти, те може да се обобщят по следния начин:

- да може да съхранява въведени рецепти;
- да може да се въвеждат рецепти - да може да се въведе име на рецептата, съставки на рецептата, описание на рецептата;

- да може да се достъпи списъка с рецептите;
- да може да се достъпи цялата информация въведена за дадена рецепта;
- да може да се търси рецепта по вече въведени продукти;
- да може да се редактира или трие вече съществуваща рецепта;

Друг важен момент от изискванията е свързан с технологиите и начина на реализация на проекта. Приложението ще е уеб базирано и реализирано с JavaScript. Това означава, че при разработката трябва да се предвиди:

- приложението да се изпълнява в браузера;
- да има напълно коректно работеща функционалност от страна на кода дори и без да е вързан с база данни и сървърна функционалност;

3.3. Планиране и изготвяне на архитектура и дизайн

След като са налице основните изисквания, които трябва да се реализират, следва етапа на изготвяне на архитектурата и дизайна. Това е един от най-обемните и комплексни етапи при разработката на приложение.

Когато се говори за уеб разработки в днешно време едно от основните неща, които трябва да се направят преди започване на разработка е да се изберат технологии за разработка. Съвременното уеб пространство дава изключителна свобода в избора на технологии и начина на разработване. В същото време позволява и използването на езици за програмиране, който дават възможност за разделяне на логиката и кода на част, която да се изпълнява при клиента (в браузера на компютъра на крайния потребител) и логика, която е изцяло на сървър. Реално може да се създаде напълно функционално приложение, което да се изпълнява при клиента дори и без да има свързана сървърна логика към него. Това позволя и пълна независимост на двете среди една от друга.

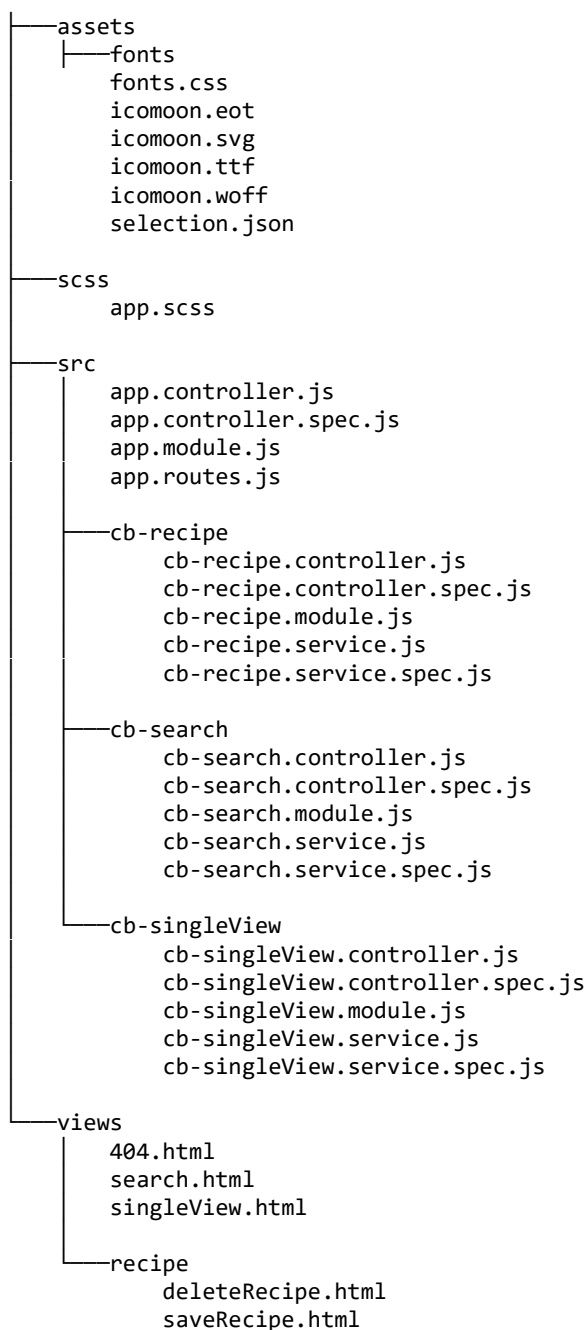
За целта на проекта и конкретното приложение се избира AngularJS. Той ще позволи да се разработи приложението без да има притеснения за сървърната технология, тъй като той е JavaScript базиран фреймуърк и единственото, което ще е нужно, за да се вижда приложението е браузър. В същото време концепцията, която стои зад AngularJS, ще позволи разделянето на кода на логически части, сравнително независими, а това ще позволи и по-детайлно планиране на самото разработване, а в последствие тестване на приложението. С този фреймуърк ще може да се изгради лесно стандартно SPA приложение с MVC архитектурен модел, разделяйки модулите по функционалност (modularization by features).

Както беше разгледано по-рано, AngularJS работи с набор от HTML фрагментите (Views), които се показват в браузъра, където шаблоните ползвани в Views се събират с данните, за да се визуализира крайния резултат. Разглеждайки основните изисквания за приложението се вижда, че трябва да се предвиди създаването на пет html файла, които ще представляват съответните Views:

- за начална страница;
- за добавяне и редактиране на рецепти;
- за изтриване на рецепти
- за показване на цялостната информация за една рецепта;
- за търсене на рецепти;

На същия принцип трябва да се раздели и функционалността. Всяка отделна част (начална страница, търсене на рецепти, добавяне на рецепти и показване на конкретна рецепта) ще бъде разделена на модули и по този начин в структурата на проекта ще са в самостоятелна папка. За структурата на модулите ще се използва принципа на разделяне на функционалност. (Фиг. 10).

```
cooking-book
├── Gruntfile.js
├── index.html
├── karma.conf.js
└── package.json
```



Фиг. 10

Както се вижда от горната фигура структурата, която трябва да придобие проекта, трябва да отговоря и на принципа всяка функционалност да е отделена. Спазен е общоприетия принцип за кръщаване на файлове и папки. Така например, папката, където ще са шаблоните, е кръстена `views`, а всеки шаблон носи името на конкретната функционалност, за която ще се използва.

Впечатление прави наличието на папка *recipe*. В нея ще са шаблоните за модула за манипулиране на рецепта (*cb-recipe* - добавяне, триене, редактиране) Тъй като ще има функционалност освен за добавяне на рецепти, така и за редактиране и триене на текущи, е добра практика шаблоните да може да са максимално опростени. Обмисляйки изпълнението на функцията за добавяне и редактиране, те в голяма степен ще се препокриват. И спокойно могат да ползват един шаблон (*saveRecipe.html*), а функционалността за триене на рецепта да се изнесе в самостоятелен шаблон.

В структурата на проекта се вижда и темплейт с име, което не съвпада с никоя от описаните функционалности - файла с име *404.html*. Необходимо по време на планиране на проекта да се предвиди и как ще се показват съобщенията за грешка на потребителя. Такава ситуация за потребителя може да възникне, ако той се опитва да зареди несъществуваща рецепта или такава, която вече е изтрита. Както всички файлове така и този се наименува според функционалността, която ще изпълнява. Код *404* обикновено се връща от сървър, когато се опитваме да достъпи несъществуващ ресурс или в нашия случай отговаря на липсваща рецепта или *view*.

Единствения шаблон, който не е в общата папка, е този за началната страница (*index.html*). По правило е отделен, защото обикновено той е рамката на цялата страница и в него се зареждат останалите шаблони при поискване. Началната страница съдържа информация, която е статична, а ако има наличие на полета, които си променят информацията, то те са малко и свързани с основната функционалност.

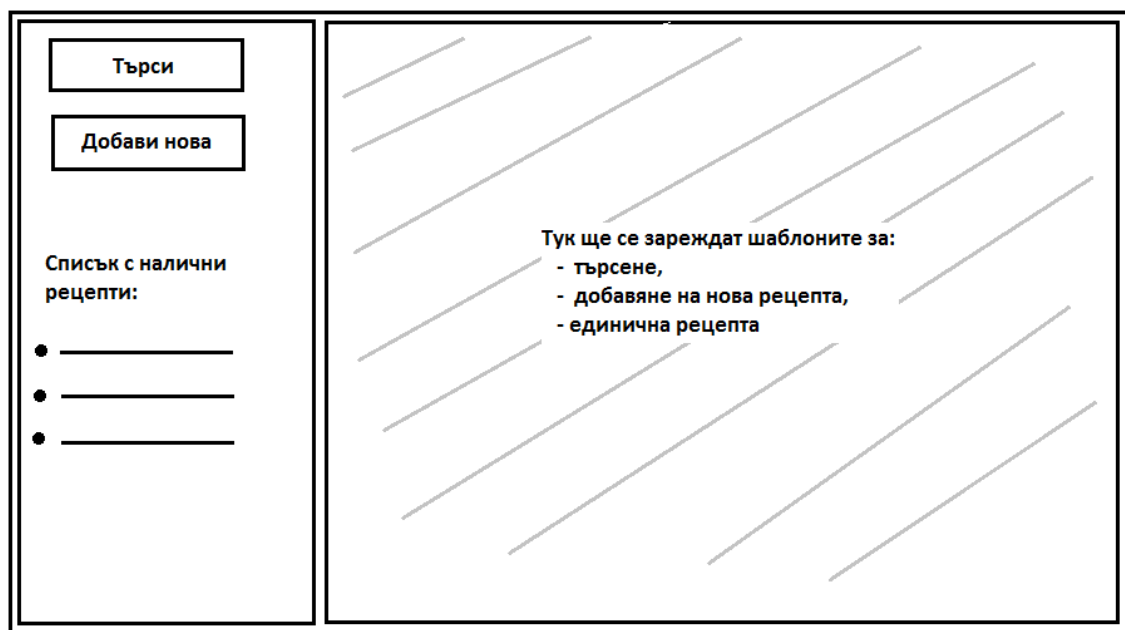
Всички JavaScript файлове, свързани с конкретна функционалност се събират в една папка. По правило тя се кръщава *src* идващо от английската дума за ресурс - *source*. Както се вижда от структурата, всяка функционалност на приложението си има своя собствена папка. Името на всеки файл е комбинация от името на конкретната функционалност, от която е част (*cb-addRecipe* – добавяне на рецепти, *cb-search* – търсене на рецепти, *cb-recipe* – основна функционалност за рецептите) както и специфичната роля, която има -

контролер (*controller*), модул (*module*), сървис (*service*), филтър (*filter*). Това е стандарт за наименоване на файловете в AngularJS проекти, според документацията на фреймуърка.

Както при шаблоните така и файловете, които са свързани с `index.html` не са в подразделение. Другата причина е, че те ще обединят и направят връзката между останалите файлове на различните функционалности. По правило името на тези файлове започва с *app* от английската дума за приложение - *application*.

Останалите две папки *assets* и *scss* са свързани с всички стилизиращи ресурси нужни за приложението. В *assets* обикновено се съхраняват картинки, шрифтове за текстове, файловете на всички библиотеки нужни за приложението, които не се инсталират през NPM. Докато папката *scss* е за файла или файловете със стиловете на приложението. Тъй като това не са стандартни CSS файлове, няма да се съхраняват в папката за статични ресурси *assets*. Причината поради, която е избрана SASS технологията, както беше разгледано в глава втора (2.4), е че освен стандартния начин за писане на стилове тя позволява използването на променливи и по-съкратен и структуриран начин за писане на стилове. Освен това за едни модерен проект, който е SPA и AngularJS базиран може да се използват последните тенденции и в писането на CSS.

Дизайна на архитектурата на едно приложение е пряко свързано с функционалното му поведение. Структурното разделение на проекта трябва да е неделима част и от презентационната му част в браузъра. В тази връзка когато се изгражда структурата, тя трябва да отговаря на това, което трябва да се изпълнява (Фиг. 11).



Фиг. 11

На горното изображение (Фиг. 11) е показано как приложението трябва да функционира. Ако схемата, която е предвидена за разделяне на функционалностите и цялостната архитектура на проекта се приложи, този резултат трябва да е на лице.

4. Глава четвърта

Разработка на проекта

4.1. Настройване на средата за разработка

След планирането и изготвянето на архитектурата на проекта и наличието на описани изисквания, следващата стъпка е да се подготви и средата за разработка на самия проект. Както вече се видя, от описаните технологии трябва да се инсталират NodeJS, за да има среда за изпълнение на JavaScript извън браузъра и GruntJS - за автоматизиране на процесите по време на разработката. След като всичко необходимо е инсталирано, средата трябва да се подготви. Това става чрез:

- Конфигуриране на проекта с `package.json`
- Конфигурация за оптимизиране с GruntJS
- Конфигурация за стартиране на проекта

4.1.1. Конфигуриране на проекта с `package.json`

Както беше дискутирано вече в глава втора (2.3.1), създаването на конфигурационния файл с основните и важни подробности за проекта е доста лесно (Фиг. 6). Това обаче е само стартовата точка. Файла е генериран, но в него липсват допълнителните ресурси за проекта (дипендънсита). Все още в него няма нищо конкретно и за самата среда.

Проекта, който ще се разработва ще ползва AngularJS фреймуърк. За това първото, което трябва да се направи, е да се инсталират всички негови компоненти. За целта ще се използва предимството, което дава NodeJS и неговия мениджър на пакети. Вместо ръчно да се свалят самите файлове с код на фреймуърка, без да може да се запази информацията за конкретната версия, ще се използва по оптимизирания начин. Всички нужни модули на AngularJS ще се инсталират с помощта на NPM, като се запазват в графата за ресурси

необходими за работата на приложението (*dependencies*) чрез добавяне към командата на допълнителната инструкция (флаг) *--save*. Нужните модули са:

- *angular* - основната функционалност на фреймуърка
- *angular-animate* - за по-добро визуално представяне на части от информацията посредством анимация, т.е. когато има промяна в HTML структурата да се създава усещане за движение
- *angular-local-storage* - приложението няма да е свързано към сървърна част, за това ще е необходимо да се симулира поведение подобно на съхраняване и достъпване на информация от база данни. Със стандарта HTML5 това е възможно да се случва в браузъра, като се използва обект наречен *localStorage*. Инсталирането на този модул на AngularJS ще позволи по-лесно боравенето с *localStorage* обекта.
- *angular-ui-route* - за да може да се осъществява превключването между различните Views

Командата, която ще се изпълни в конзолата, за инсталирането на всички тези модули ще има следния вид:

```
npm install angular angular-animate angular-local-storage angular-ui-route  
--save
```

Накрая е необходимо да се инсталира и самия GruntJS преди да може да се премине към неговата конфигурация. Това става с помощта на NPM, като той ще бъде запазен като ресурс необходим само за средата за разработка, чрез добавяне към командата на допълнителната инструкция (флаг) *--save-dev*:

```
npm install grunt --save-dev
```

След изпълнението на командите и инсталирането на пакетите, информацията за тях трябва да се е попълнила автоматично в конфигурационния файл *package.json* (Приложение №1).

4.1.2. Конфигурация за оптимизиране с GruntJS

След като е направен първоначален план на проекта и са свалени основните пакети нужни за самото приложение, трябва да се добави настройката и за оптимизация. При избиране на правилните пакети на GruntJS, които да се свалят, трябва да се прегледа пак структурата на проекта (Фиг. 10). Няколко са основните неща, които могат да се направят:

- Оптимизиране на броя JavaScript файлове, обединявайки ги в един чрез *grunt-contrib-concat*
- Проверка за коректно написан JavaScript чрез *grunt-contrib-jshint*
- Намаляване размера на JavaScript файловете чрез *grunt-contrib-uglify*
- Оптимизиране на процесите чрез *grunt-contrib-watch*

Всички тези пакети ще се свалят с NPM, като те ще са нужни само за разработката на приложението. Командата ще има следния вид:

```
npm install grunt-contrib-concat grunt-contrib-jshint grunt-contrib-uglify  
grunt-contrib-watch --save-dev
```

4.1.2.1. Оптимизиране на броя JavaScript файлове, обединявайки ги в един чрез *grunt-contrib-concat*

Това, което се откроява много ясно е, че в проекта ще има много JavaScript файлове. Най-неприятната част в работа с много ресурси, е че трябва всеки един от тях да се добавя на страницата ръчно. Ако се изпусне дори един, ще започнат да се появяват грешки и функционалностите могат да спрат да работят. Освен това трябва да се внимава за поредността, в която се зареждат, защото между тях има сложна зависимост. От гледна точка за поведението на приложението в браузъра, броя на ресурсите, които трябва да се заредят, също са от съществено значение. Колкото повече файлове трябва да се свалят от клиента преди да тръгне приложението, толкова повече това води до забавяне на стартирането му. Тук идва ролята на оптимизацията, за да може програмиста

да не е ангажиран за такива дребни подробности по оптимизирането на проекта, а то да се случват автоматично на заден фон.

При работа с много единични JavaScript файлове, е най-добре всичките да се обединят в един. С пакета на GruntJS наречен *concat* (*grunt-contrib-concat*), ще може да се осъществи тази задача (от английската дума *concatenation* – събиране, нанизване). Важно е обаче, да се внимава в каква последователност ще се настрои да се събират всички файлове в едно. Един файл зависи от друг и има определена последователност на изпълнение и извикването им. От разгледаната структура на AngularJS, е ясно, че първо трябва да се заредят модулите (*modules*), след това са файловете, които изграждат адресите (*routes*) и чак след това всички останали. По правило, файла, който ще се генерира автоматично и ще е сбор от всички останали файлове, се отделя в самостоятелна папка, в случая на проекта папката ще се казва *build*. Папката не се създава от разработчика, тя се генерира от задачата изпълнявана от GruntJS. Също така, тя не се приема като обект на оптимизация и програмиста не я използва за разработка. Там се съхраняват генерираните и оптимизирани файлове за работа на приложението. Съответно ако в проекта се използва някоя от системите за управление на кода, не е нужно тя да се пази, защото при всяка промяна на кода тя се актуализира. Същевременно при пускането на приложението точно тя се пренася на сървър, вместо *src* папката, защото приложението ползва оптимизираните файлове по време на работа. Процеса на генериране на сборния файл се осъществява, чрез изпълнението на команда в конзолата - *grunt concat*.

В проекта има два типа JavaScript файлове: създадени от програмиста, и идващи от AngularJS фреймуърка. Добра практика, е когато се настройва задачата за събиране на файловете в едно да се раздели на две части. В един файл да се събират всички файлове идващи от фреймуърка и в отделен файл да се събират всички написани от разработчика. Имената на тези генерирани файлове се определя при записването на конкретната настройка. За първия ще се даде име, което да показва че това са външни ресурси (*thirdparty*

). За втория, съдържащ всички файлове написани за приложението, името ще се генерира спрямо това, което е записано като име на проекта в `package.json`, чрез параметър в конфигурацията (`<%= pkg.name %>.js`).

Друга фина настройка, която може да се направи, с този задача в GruntJS, е да се добавят коментари автоматично. Така когато се генерира новият файл, преди да се добави съдържанието на даден файл, ще се запише коментар, който ще описва от кой файл се зарежда. Това е много полезно, когато програмиста трябва да проверява изпълнението на кода по време на работа на приложението. Така винаги може да се провери конкретния код от къде точно идва. Също така може да се добави и функционалност наречена *sourceMap*. Тя също улеснява проверката на изпълнението на кода по време на работа, като може да покаже директно в браузера оригиналния вид на кода на програмиста преди събирането му в един общ файл. Конфигурацията на тази задача ще има следния вид:

```
concat: {
  options: {
    process: function(src, filepath) {
      return '/*! Source: ' + filepath + ' */' + '\n' +
        src.replace(/(^|\n)[ \t]*('use strict'|"use strict");?/s/g, '$1');
    },
    sourceMap: true
  },
  dist: {
    src: [
      //първо се събират файловете modules
      'src/*.module.js', 'src/**/*.module.js',
      //после routes
      'src/*.routes.js', 'src/**/*.routes.js',
      //след това всички останали
      'src/*.js', 'src/**/*.js',
    ],
    dest: 'build/js/<%= pkg.name %>.js'
  },
  thirdparty: {
    src: [
      //външни ресурси
      'node_modules/angular/angular.js',
      'node_modules/angular-ui-router/release/angular-ui-router.js',
      'node_modules/angular-local-storage/dist/angular-local-storage.js',
      'node_modules/angular-animate/angular-animate.js'
    ],
    dest: 'build/js/thirdparty.js'
  },
}
```

4.1.2.2. Проверка за коректно написан JavaScript чрез `grunt-contrib-jshint`

Всеки програмист има различен подход при писане на JavaScript и използва различна софтуерната програма за разработване (IDE). Понеже JavaScript не е стриктен език, някои неща може да се напишат по няколко различни начина, а това понякога може да доведе до грешки. Има различни подходи как да се избегнат тези грешки, а при наличието им разработчика да ги вижда в движение. Един удачен вариант, който да не принуждава програмиста да ползва конкретно IDE и позволява „настройване“ на правилата според нуждите, е използването на модула на GruntJS - `grunt-contrib-jshint`. Така ще може да се въведат правила за писане, валидни за целия проект и за всеки програмист, които работи по проекта. Като най-основни, предпазващи от сериозни грешки и помагачи кода да бъде пазен чист са [11]:

- Да не се позволява изпускането на фигурни скоби при изписването на цикли;
- Да се избере предпочитан подход при използването на логични оператори за сравняване на стойности с двойно равенство или тройно (`==`, `!=` или `===`, `!==`);
- Дали да се показва съобщение за грешка ако се прави проверка за равенство на стойност с `null`;
- Обозначаване на средата за разработка. Настоящото приложение ще се изпълнява в браузъра и така Grunt модула ще знае за предварително дефинираните променливи идващи от средата на браузъра (*document*, *navigator* и т.н.). Съответно няма да показва грешка, че се използва променлива, която не е дефинирана;
- Превенция на презаписване на променливи чрез използване на капсулиране при ползване на цикъл, подобно на капсулирането съществуващо в тялото на стандартна функция (*function scope*);

- Също така съобщава когато има дефинирана променлива глобално (без декларативната дума *var*) или когато има дефинирана променлива, но тя не се използва никъде.

Понеже структурата на проекта е разделена на много файлове, който после се обединяват в един, добра практика е да има две отделни подзадачи за проверка на кода: една за всеки файл преди събирането им (*beforeconcat*) и една за сглобения файл (*afterconcat*). Така ще може да се провери дали всичко е наред на всички етапи. Настройката на тази задача ще изглежда по следния начин:

```
jshint: {
  options: {
    curly: true,
    eqeqeq: false,
    eqnull: true,
    browser: true,
    funcscope: false,
    unused: true
    // globals: {
    //   jQuery: true
    // }
  },
  beforeconcat: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js'],
  afterconcat: ['build/js/<%= pkg.name %>.js']
}
```

След като са определени всички тези правила лесно може да се видят допуснатите нарушения. След извикване на команда в конзолата, за да се извърши проверката (*grunt jshint*), ще се изпише съобщение, ако някое от правилата е нарушено и кое точно (Фиг. 12).

```
d:\wamp\www\cooking-book>grunt build
Running "jshint:beforeconcat" (jshint) task

src/app.controller.js
 60 |         var test = 'jshit test';
    |             ^ 'test' is defined but never used.

>> 1 error in 12 files
Warning: Task "jshint:beforeconcat" failed. Use --force to continue.

Aborted due to warnings.

d:\wamp\www\cooking-book>
```

Фиг. 12

4.1.2.3. Намаляване размера на JavaScript файловете чрез `grunt-contrib-uglify`

Когато се разработва приложение за уеб, от изключителна важност за неговата оптимална и бърза работа е големината на ресурсите. Това може да се постигне чрез оптимизация, която се нарича загрозяване на кода, от английската дума `uglify` (*grunt-contrib-uglify*). Нейната цел е, крайния код да бъде минимизиран, което затруднява четенето му. Така когато приложението се пусне да работи много по-трудно злонамерени хора ще могат да откраднат функционалността или да се опитат да разбият кода. Освен това минимизирането означава и намаляване на размера на файла, което пести пространство, важен фактор за самия проект. Малкия размер на файла е важно и от страна на потребителя, защото когато той зарежда страницата или приложението, големината на ресурсите, които ще сваля е намалено и това води до по-бързо зареждане на приложението.

Този процес на минимизиране се извиква чрез команда в конзолата *grunt uglify*, при което се прочита вече сглобения файл, който имаме и генерира нов. Задачата е настроена, така че новия файл да има същото име като на първоначално сглобения файл (*cooking-book.js*), но с добавка *.min* (*cooking-book.min.js*), като отново се ползва параметър (`<%= pkg.name %>`). Така ще се следва общоприетия начин за наименуване на минимизирани файлове.

Освен това и тук може да се позволи ползването на *sourceMap* както при задачата *concat*. Така в самото приложение ще се зарежда само минимизирания файл, още на ниво разработка, а при нужда разработчика ще може да проверява изпълнението на кода в браузъра в четим за него вид и във вида, в който го е написал. Тази настройка ще изглежда по следния начин:

```
uglify: {
  my_target: {
    options: {
      sourceMap: true,
      preserveComments: 'some'
    },
    files: {
      'build/js/<%= pkg.name %>.min.js': ['build/js/<%= pkg.name %>.js']
    }
  }
}
```

```

    }
  },
},

```

4.1.2.4. Оптимизиране на процесите чрез `grunt-contrib-watch`

За да се изпълни всяка една от показаните до сега настройки се иска извикване на команда в конзола. При всяка промяна в оригиналните файлове, за да се създаде наново автоматично сглобения файл, и останалите оптимизации се извиква съответната командата всеки път. Това е не по-малко разсейващо, от това да се добавя всеки файл самостоятелно. С тази цел ще се добави още една настройка в GruntJS. Тя позволява създаване на така наречената автоматизирана среда за разработка. Добавката, която се ползва се нарича *watch* (*grunt-contrib-watch*) от английската дума гледам, наблюдавам. Това, за което тя е предназначена, е да следи дали, в някой от оригиналните файлове е налична промяна, дали има нов запис в него и автоматично да извика всички останали задачи за оптимизиране настроени до момента. При започване на работа трябва само веднъж да се изпълни командата *grunt watch* в конзолата, за да стартира наблюдаването на файловете, а настройката на задачата за конкретния проект ще има следния вид:

```

watch: {
  js: {
    files: ['src/*.js', 'src/**/*.js'],
    tasks: ['jshint:beforeconcat', 'concat:dist']
  },
  scss: {
    files: ['scss/*.scss'],
    tasks: ['sass']
  }
},

```

За да се оптимизира максимално процеса, ще се приложи и друга възможност на GruntJS. Ще се направи регистър на задачите. Това ще позволи всички описани вече задачи да се групират и да се извикват заедно. По този начин ще може да се разделят в две графи:

- команда за цялостно генериране и оптимизиране на всички файлове само веднъж *grunt build* - така ще може да има готова команда, с която приложението да се подготви с всички нужни ресурси за пускане в експлоатация;
- команда нужна само за целите на разработката *grunt dev* - Тя ще извиква първата команда (*grunt build*), за да е сигурно, че приложението ще се стартира с актуални и работещи файлове. След това ще извика задачата за *watch*, с която ще следи за промяна във файловете, за да може да генерира минимизирания файл отново.

```
grunt.registerTask('build', ['jshint:beforeconcat', 'concat', 'jshint:afterconcat', 'uglify']);  
grunt.registerTask('dev', ['build', 'watch']);
```

Най-важните задачи за оптимизация вече са готови и може да се премине към самото писане на код. Това ще позволи фокусиране върху разработването на функционалността на приложението, без разсейване дали всички нужни ресурси са добавени или дали са в правилната последователност (Приложение №2).

4.1.3. Конфигурация за стартиране на проекта

Подготвена е средата за разработка, има изготвен архитектурен план на приложението и изискванията към функционалността му са ясни. Остава само да се уверим, че е наличен и локален сървър, за да може лесно да се стартира приложението от компютъра на разработчика. Няколко са вариантите, с който може да се постигне това. Първата опция е ако се използва някое IDE, което поддържа тази функция и при стартиране на html документ да го зарежда в браузъра под localhost. Подобни софтуерни програми за разработване са Visual Studio и WebStorm. Вторият вариант - чрез използване на програми като XAMPP или WAMP. Те са лесни за инсталиране и ползване и не обвързват разработката с използване на конкретно IDE. И двата инструмента

представяват среда за разработка за php и Apache проекти, която стартира локален сървър.

За проект като настоящия, който ползва NodeJS може и да се извика прост веб сървър от папката на проекта. За целта трябва да се инсталира модул на NodeJS *http-server*, като това трябва да стане глобално за компютъра на разработване, без да се запазва в конфигурационния файл на проекта (*npm install http-server -g*). Създадения сървър ще се ползва само за стартиране на проекта по време на разработване и няма да е нужно да се пази в конфигурацията на проекта. Освен това всеки програмист може да има различен подход на стартиране на проекта. Сървъра ще се стартира лесно чрез команда в конзолата от локацията на проекта (*http-server -o -c-1*). Не трябва да се забравя, че при този вариант трябва да има и втора конзола, в която да се стартират задачите за оптимизация на проекта от GruntJS.

Тъй като проекта е JavaScript и разполага с JavaScript сървър, може и да се напише такъв за стартиране на приложението. За целта ще е нужно да се инсталира друг модул за по-лесно писане в NodeJS среда - *express*. Както в предния вариант, така и тук ще се инсталира без да се запазва в конфигурационния файл на проекта, само в директорията на проекта (*npm install express*). След това в папката на проекта ще се създаде файл с име *server.js*. Конфигурацията в него ще бъде сравнително проста, ще създава сървър с адреса *localhost* и порт *2112*:

```
var express = require('express');
var app = express();
app.use(express.static('./'));
var server = app.listen(2112, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

Стартирането на сървъра ще става отново в конзолата с команда *node server.js*. В браузъра ще е нужно само да се запише адреса, който е

конфигуриран в `server.js` файла - `http://localhost:2112/`. И при този вариант трябва да има две стартирани конзоли, за да може да вървят и двата процеса - за сървъра и за GruntJS.

Последната опция е да се зареди проекта в браузъра Mozilla Firefox. Той е създаден така, че да може да зарежда страници и техни ресурси директно от компютъра, имитирайки сървърна среда. Единственото, което е нужно да се зареди основната страница `index.html` в браузъра. Това само по себе си е ограничение, защото няма да позволи проекта да се прегледа под други браузъри.

Всеки от вариантите има своите плюсове и минуси. Избора им обикновено е на база на предпочитанията на програмиста или изградена политика за разработка на проект в колектива. Никой от вариантите няма да обвърже проекта с конкретна сървърна технология. Те ще позволят стартиране на локален сървър. Както се вижда от посочените варианти има инструменти създадени за `php` проекти, както и такива за JavaScript среда. За конкретния проект ще се използва вариант с WAMP, тъй като няма да е нужно да се поддържат няколко конзоли с различни команди, което понякога може да доведе и до объркване.

4.2. Разработка на проекта

След като е сигурно че проекта ще може да бъде стартиран и всички останали нужни конфигурации са направени, може да се премине към същинският процес на разработване. Имайки предвид дизайна на архитектурата, трябва да се направят следните неща:

- Създаване на архитектурата и разделяне на модули
- Функционалност за управление на рецептите- модул `cb-recipe`
- Функционалност на модул за изглед на единична рецепта - модул `cb-singleView`
- Функционалност за търсене - модул `cb-search`

4.2.1. Създаване на архитектурата и разделяне на модули

На този етап трябва да се изгради скелета на приложението. Това означава, че трябва да се създадат основните за зареждането на приложението файлове: *index.html*, *app.module.js*, *app.controller.js*, и *app.routes.js*.

Както вече беше описано, за едно стандартно SPA приложение е важна началната страница. Тя реално е и единствена за него и играе ролята на скелет. Нейната структура е стандартна, базираща се на стандарта HTML5 и има всички задължителни елементи като *doctype*, *html*, *head* и *body*. Първото нещо, което трябва да се зареди на тази страница са ресурсите. В началото на файла, в *head* частта, се зареждат стиловете и допълнителните стилизиращи приложението ресурси.

```
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
user-scalable=0">
  <meta name="description" content="Електронна книга за рецепти">
  <meta name="author" content="EleonoraDimchevska">
  <link rel="stylesheet" type="text/css" href="assets/fonts/fonts.css">
  <link rel="stylesheet" type="text/css" href="build/css/app.css">
</head>
```

В края на страницата, точно преди затварящия *body* елемент се поставят JavaScript файловете. Според начина, по който са конфигурирани задачите на GruntJS, това са два файла: *thirdparty.js* - сборен от всички файлове на фреймуърка и всички външни ресурси, *cooking-book.js* - сборен от всички файлове създадени за самото приложение.

```
<body>
  ...
  <script src="build/js/thirdparty.js"></script>
  <script src="build/js/cooking-book.js"></script>
</body>
```

Разглеждайки внимателно примерния дизайн (Фиг. 11) може да се определят основните елементи, които трябва да съдържа началната страница. От ляво ще бъде разположена секция с навигация (бутоните за търсене и добавяне на рецепта).

```
<nav>
  <ul class="nav">
    <li ui-sref-active="active">
      <i class="icon-document-search2"></i>
      <a ui-sref="search">Търси</a>
    </li>
    <li ui-sref-active="active">
      <i class="icon-clipboard-add2"></i>
      <a ui-sref="addRecipe">Добави рецепта</a>
    </li>
  </ul>
</nav>
```

Като може да се види това е съвсем стандартна навигация. В нея ще се приложи едно от предимствата при AngularJS. За да може при ползването ѝ да се знае във всеки един момент, кой точно бутон е избран, ще се ползва директива идваща от модула *ui.router* на фреймуърк - *ui-sref-active*. Това ще позволи динамично добавяне на клас (*active*), щом даден бутон стане активен, т.е. потребителя кликне на него. Така ще може да се напишат стилове, които да показват това действие на потребителя. Под навигацията ще бъде списъка с въведените рецепти, както може да се види и от грубия дизайн показан на фигура 11 от глава трета:

```
<div class="list-wrap">
  <ul>
    <li>
      <i class="icon-clipboard-list2"></i>
      <p>Налични рецепти</p>
    </li>
    <li ng-repeat="recipe in recipeList" ui-sref-active="active">
      <a ui-sref="singleView({recipeID: recipe.id})">
        {{recipe.name}}
      </a>
    </li>
  </ul>
</div>
```

За списъка с рецепти ще се използва шаблон, който ще позволи динамично показване на всички налични рецепти, както и ако се добави нова, тя ще се появи автоматично, без да е необходимо презареждане на цялата страница. За целта ще се ползва директива идваща от AngularJS - *ng-repeat* и за изписването на имената на рецептите двупосочно обновяване на данните с фигурни скоби. Също така, за да може при избирането на някоя рецепта от списъка да се заредят нейните данни, ще се ползва друга директива, която ще посочва, коя е избраната рецепта - *ui-sref*. Параметрите посочени в тази директива ще се обработват от *app.routes.js*, който ще се създаде щом се създадат основния модул и контролер на приложението.

Следвайки дизайна, от глава трета, от дясно ще бъде контейнера, който ще зарежда различните Views. Той е доста прост като структура, тъй като ползва готова AngularJS директива - *ui-view*. Така приложението ще знае къде да разположи останалите темплейти.

```
<div class="col">
  <div ui-view class="view"></div>
</div>
```

Следващата стъпка е да се дефинира главния модул на приложението *app.module.js*. По този начин ще се осъществи връзката между файловете отговарящи за функционалността и темплейтите на приложението. Създавайки модул *cookingBook* в JavaScript файла, после се посочва в *index.html*, коя част ще бъде менажирана от AngularJS с директивата *ng-app*. Най-препоръчвания вариант е директивата да се слага още на самия *html* таг. Така фреймуърка ще знае, че трябва да управлява цялата страница. Важното за модула е да се посочат правилно всички допълнителни ресурси на които ще разчита, като:

➤ *ui.router* и *ngAnimate* - инсталираните през NPM допълнителни пакети на AngularJS

➤ *cookingBook.recipe*, *cookingBook.singleView*, *cookingBook.search* - под модулите за функционалностите, който ще трябва да се създадат

Така в споменатите файлове трябва да е наличен следния код:

Създаване на модул
(*app.module.js*)

```
(function () {
  'use strict';
  angular.module('cookingBook', [
    'ui.router',
    'ngAnimate',
    'cookingBook.recipe',
    'cookingBook.singleView',
    'cookingBook.search'
  ]);
})();
```

Ползване на модул в html
(*index.html*)

```
<html ng-app="cookingBook">
...
</html>
```

Както се вижда от файла за модула, важно при писането на всички JavaScript файлове, е те да бъдат капсулирани или изолирани, т.е. всяка променлива и функция да не е достъпна за останалите. Така ще бъде сигурно, че след изпълнението на задачата на GruntJS concat, няма да се получи презаписване на стойности и функции, ако се има дублиране във файловете.

Следващата стъпка е да се създаде и контролера - *app.controller.js*. Той ще осъществява връзката между темплейта и логиката на приложението. Понеже това е скелета на приложението, тук няма да се създава функционалност. Единственото, което е нужно тук е да се взимат наличните рецепти. Тази информация ще идва от модула *cookingBook.recipe*. Затова в контролера на този етап ще има само една DI (dependency injection), т.е. че ще се ползва сървис от този модул (*cbRecipeService*). От него ще се извиква функция за взимане на рецептите (*cbRecipeService.getRecipe()*), която ще трябва да се имплементира по-късно когато се имплементира модула *cookingBook.recipe*:

```
(function () {
  'use strict';
  var app = angular.module("cookingBook");
  app.controller("CookingBookController",
    [ "$scope", '$rootScope', 'cbRecipeService',
      function($scope, $rootScope, cbRecipeService){
        $rootScope.recipeList = cbRecipeService.getRecipe();
      }
    ]
  );
})();
```

Останалите две DI, които се ползват в контролера са стандартни идващи от фреймуърка и с тях се управлява подаването на информацията към темплейтите - *\$scope*, *\$rootScope*.

Друг важен момент за всеки контролер е начина, по който се създава. Както се вижда и от кода, първо се посочва модула, от който е част (*var app*), след което се създава и самия контролер (*app.controller*).

Остава само да се създаде и файла отговарящ за навигирането между темплейтите - *app.routes.js*. Знаейки структурата на приложението и функционалността, може още на този етап да се разпишат нужните връзки (*state*). По този начин ще може да се определи и как да изглежда самия адрес (*url* адрес), на всяко действие. Това се постига чрез обект, в който се конфигурират:

- името на адреса
- темплейта, който ще ползва
- и контролера, който ще управлява самия темплейт

```
.state('search', {
  url: '/search',
  templateUrl: 'views/search.html',
  controller: 'SearchController'
})
```

Самия навигационен адрес според конфигурацията ще има следния вид:

<http://localhost/cooking-book/index.html#/search>

Прави впечатление, че в адреса, освен стандартното изписване, се съдържа символ (#). Това е начина, който фреймуърка използва, за да отделя визуално и функционално, коя част от адреса се отнася до конкретния шаблон.

По показания начин трябва да се направи конфигурация за страниците за евентуална грешка 404.html, добавяне, показване на рецепта, редактиране и триене на рецепта. Последните три случая ще са обвързани с конкретна рецепта, за да може функционалността да работи. Това може да се постигне, като към свойството `url` на обекта се подаде и така наречения параметър. Той ще представлява идентификатора на конкретната рецепта. Начина, по който се записва, е като след името на адреса който е зададен се добавят разделителна наклонена черта с две точки и име на параметъра (`/:recipeID`). Така ще се знае коя точно рецепта ще бъде изтрита, редактиране или показана.

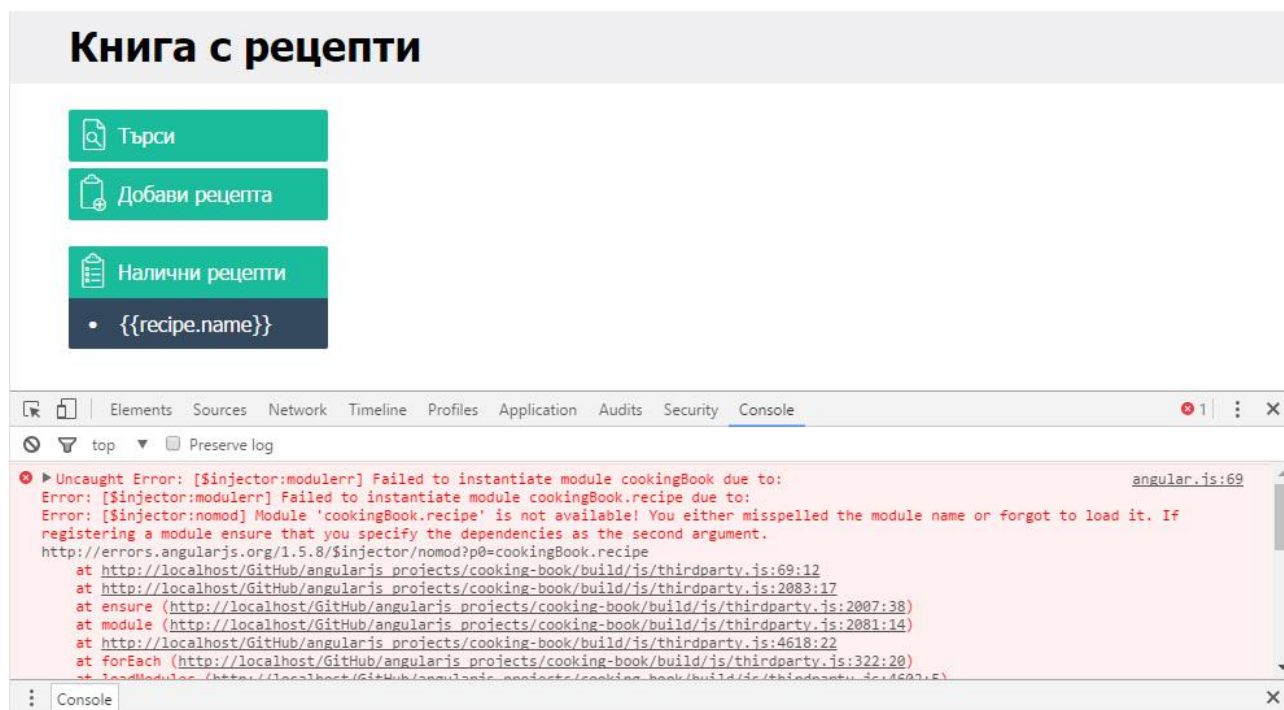
```
.state('deleteRecipe', {  
  url: '/delete/:recipeID',  
  templateUrl: 'views/recipe/deleteRecipe.html',  
  controller: 'RecipeController'  
})
```

Съответно адреса на една подобна конфигурация ще изглежда по следния начин, като цифрата в края е уникалният номер на рецептата идващ от параметъра (`/:recipeID`):

`http://localhost/cooking-book/index.html#/delete/2`

4.2.2. Функционалност за управление на рецептите - `cb-recipe`

В този етап на разработка от приложението няма да се вижда много освен бутоните от навигацията и темплейта за наличните рецепти. Освен това в конзолата на браузъра ще се изписват множество грешки. Това е така, защото в `app.module.js` и `app.routes.js` вече са дефинирани останалите модули, но те все още не съществуват (Фиг. 13).



Фиг. 13

Първият модул, който ще се разработи ще бъде *cookingBook.recipe*. Той е най-големият и най-сложният. От него ще се управляват данните за рецептите, за операциите добавяне, триене и редактиране на рецептите. Също така ще трябва да има функция и за взимане на всички налични рецепти. Това са стандартни операции при работа с база данни, затова и тези функции ще бъдат изградени на подобен принцип.

Както вече беше споменато, приложението няма да работи със сървърна среда, съответно няма да има и връзка с база данни. В конкретния случай ще се използва обекта в браузъра *localStorage*. Предимството в този подход е, че:

- съхраняването на данните в *localStorage* много наподобява нерелационна база данни
- лесно се структурират функционалности, които в последствие може да се използват за връзка с истинска база от данни
- *localStorage* може да се използва заедно с база данни, като по този начин ще позволи на приложението да работи и ако достъпа до базата данни бъде прекъснат (офлайн режим)

Трябва да се внимава колко и какви данни ще се съхраняват по този начин, защото пространството е ограничено за разлика от истинска база данни. Обекта `localStorage` разполага с приблизително ~5MB памет, като има леки вариации според браузъра, който се ползва. За целите на текущото приложение това пространство е напълно достатъчно.

Преди да се започне с писането на функционалността, първо трябва да се създаде модул. Това става като се създаде файл *cb-recipe.module.js*, в който да се дефинираме съответния модул *cookingBook.recipe*. Както и при *app.module.js*, така и тук се описват допълнителните модули, на които разчита конкретна функционалност:

```
(function () {  
  'use strict';  
  angular.module('cookingBook.recipe', [  
    'LocalStorageModule',  
    'cookingBook.singleView'  
  ]);  
})();
```

Това, което е специфично за модулната функционалност, е че всяка има собствен модул, но във самата `html` структура фигурира само едно извикване на главния модул *ng-app*. Затова и името на всеки подмодул съдържа името на главния и с точка се добавя името на подмодула. Като съответно името трябва да описва функционалността, която ще изпълнява.

Вече може да се пристъпи и към конкретната функционалност. Ще се започне със създаването на сървис (*cb-recipe.service.js*), тъй като той съдържа цялата логика. Създаването на сървис наподобява това на контролер:

```
(function () {  
  'use strict';  
  var app = angular.module("cookingBook.recipe");  
  app.service("cbRecipeService", ['localStorageService', function (localStorageService) {  
    ...  
  }]);  
})();
```

И тук първо се посочва модула, към който се създава. Сървисите също могат да имат DI, като записването им става по същия начин както при контролера. Този сървис ще разчита на функционалност идваща от `angular-local-storage`, който се свали още при настройване на проекта, за да може да запазва рецептите в `localStorage` обекта на браузъра.

Първата функционалност, която ще се създаде в сървиса, ще показва наличните рецепти. За да може да се визуализира някакви рецепти още при първоначалното стартиране на приложението ще се дефинира променлива (*initData*) с първоначално генерирани рецепти. Тя ще е масив от обекти, а всеки обект ще бъде една рецепта. В случай, че приложението в последствие бъде свързано да работи с база данни, този променлива ще взема данните от там, а няма да е със статични данни. Обекта за една рецепта трябва да съдържа следните свойства и стойности от типа:

```
{
  id: 1,
  name: "Кекс",
  description: "някакво описание",
  ingredients: [
    { "ingredientName": "масло", "amount": "100", "amountUnits": "rp" },
    { "ingredientName": "захар", "amount": "100", "amountUnits": "rp" }
  ]
},
```

Процеса на взимане на текущите рецепти ще се извършва от функцията *getRecipe*, която се извиква и в `app.controller.js`, за показване на рецептите в лявата колона. Остава да се добави проверка, при зареждане на приложението, която да проверява дали вече има някакви записани данни в `localStorage` обекта, за да знае функцията какви стойности да връща при извикването си - статично дефинираната променлива *initData* или ако не се стартира за пръв път приложението и вече има запазени рецепти в `localStorage` да връща директно записани там рецепти. Така информацията, която потребителя вече е въвел, ще може да се пази и при всяко ново отваряне на браузъра ще е налична. При първо стартиране на приложението е важно информацията от *initData* да се запишат в `localStorage`. По този начин при

следващо зареждане в браузъра, ще може лесно да се разбере, че не е първо зареждане. А при евентуално последващо свързване на приложението с база данни, това би спестило едно обръщане към базата за визуализиране на рецепти. Всичко това в тялото на функцията *getRecipe* ще има следния вид:

```
var localStorage = localStorageService.get( "recipeList");
if(localStorage === null) {
    localStorageService.set( "recipeList", initData);
    return initData;
} else {
    return localStorage;
}
```

Следващата функционалност е за добавяне на рецепти (*addRecipe*). Тя ще трябва да получава три параметъра: името на рецептата (*recipeName*), списъка с продуктите (*ingredientsList*) и описанието на рецептата (*recipeDescription*). За да може да се запише новата рецепта в *localStorage*, обекта за нея, ще се генерира динамично с всички нужни данни за една рецепта и те ще се пазят в променлива *recipeValues*. Данните, които потребителя въвежда за рецептата са известни, те идват с параметрите. Остава само да се създаде уникален номер на рецептата (*id*). Един възможен вариант това да стане е като се вземе списъка с наличните рецепти, да се провери уникалния номер (идентификатор) на последната рецепта и към него да се добави единица. След това променливата заедно с данните за новата рецепта се добавят в списъка с наличните рецепти и той се записва обратно в *localStorage* обекта.

```
$this.addRecipe = function(recipeName, ingredientsList, recipeDescription){
    var recipeList = $this.getRecipe();
    $this.recipeValues = {
        id:            recipeList[recipeList.length - 1].id + 1,
        name:          recipeName,
        ingredients:    ingredientsList,
        description:    recipeDescription
    };
    recipeList.push($this.recipeValues);
    localStorageService.set( "recipeList", recipeList);
    //обновяване на $scope.recipeList
    return recipeList;
};
```

Редактирането на рецепта е много подобно на добавянето. Тук е необходимо да се знае уникалният номер на рецептата, която ще се редактира. Затова функцията ще получава като параметри трите вече посочени за функцията `addRecipe` (*recipeName*, *ingredientsList*, *recipeDescription*) както и променлива съдържащата уникалният номер на рецептата (*recipeID*). Този параметър идва посредством конфигурацията на `app.routes.js`, която вече беше показана.

Разликата при тази функционалност, е че при запазването на рецептата имаме всички нови данни за нея и не трябва да се генерира нищо. За да се запази обаче трябва да се намери къде се намира тя в целия списък, т.е. нейния индекс. Това ще се постигне, като се създаде функция, с която да се търси индекса на рецептата (*returnRecipeIndex*). Тя ще получава като параметри списъка със всички рецепти и уникалният номер на всяка. Ще се провери къде в списъка се намира съответният идентификатор и ще върне неговия индекс. След това е необходимо само да се замени старата рецептата със новата. Това се постига като направо се презаписва цялата рецепта в масива, а самия той после трябва да бъде наново записан в `localStorage` обекта, за да се обнови с последната информация:

```
$this.updateRecipe = function(recipeName, ingredientsList, recipeDescription, recipeID){
    var recipeList = $this.getRecipe();
    var recipeIndex = $this.returnRecipeIndex(recipeList, parseInt(recipeID));
    $this.recipeValues = {
        id:           parseInt(recipeID),
        name:         recipeName,
        ingredients: ingredientsList,
        description: recipeDescription
    };
    recipeList[recipeIndex] = $this.recipeValues;
    localStorageService.set( "recipeList", recipeList);
    //обновяване на $scope.recipeList
    return recipeList;
};
```

И двете функции на добавяне и редактиране на рецептите накрая ще връщат стойност към контролера – *recipeList*. По този начин контролера ще има

обновения списък с рецепти и ще може да обнови списъка в лявата част с наличните рецепти да обнови веднага списъка.

Триенето на рецептите също е сравнително лесна операция. Информацията необходима на функцията *deleteRecipe* е само идентификатора на рецептата. Ще се използва вече създадената функция *returnRecipeIndex*, за да се намери индекса на рецептата. Така лесно ще може да се открие позицията, на която е записана в масива с рецепти и да се премахне от него. Накрая функцията ще връща към контролера обект (*\$this.confirmData*), с потвърдителна информация, че е изтрита рецепта, за да може да се обнови както текущия шаблон и списъка на наличните рецепти в лявата колона:

```
$this.deleteRecipe = function (recipeID){
    var recipeList = $this.getRecipe();
    var recipeIndex = $this.returnRecipeIndex(recipeList, recipeID);
    recipeList.splice(recipeIndex, 1);
    localStorageService.set( "recipeList", recipeList);
    //обновяване на $scope.recipeList
    $this.confirmData = {
        confirmDeleting: 1,
        updateRecipeList: recipeList
    };
    return $this.confirmData;
};
```

След като функционалността е готова трябва да се подготвят шаблоните. Специфичното за тях е, че те се създават във файлове с разширение *html*, но не съдържат никой от задължителните елементи на една страница. При тях не се записват елементите *html*, *head*, *body*, а само тези необходими за визуализацията на конкретна информация. Това е така понеже те реално не са отделни страници, а парчета код, който ще се визуализира в контекста на главната страница (*index.html*).

Както вече беше разгледано в глава трета (Фиг. 10), шаблоните нужни за модула *cookingBook.recipe* ще са два: за добавяне и редактиране на рецепта *saveRecipe.html* и за изтриване на рецепта *deleteRecipe.html*. Първия темплейт трябва да има следните полета за попълване на данните:

➤ името на рецептата;

- за описанието на рецептата;
- за съставките;

Ако се разгледа отново обекта с данни за една рецепта, се забелязва, че продуктите имат три стойности: име на съставка (*ingredientName*), количество (*amount*) и мерна единица (*amountUnits*). Така в шаблона за една съставка на рецепта ще трябва да има три полета. Понеже всяка рецепта съдържа различен брой продукти, ще се добавят и два бутона (Приложение №3):

- За добавяне на нов ред с полета за записване на нов продукт
- За триене на определен продукт

За да може цялата тази информация да се запази ще има още един бутон „Запази“, който да изпълнява тази функция (Фиг. 14).

Фиг. 14

Така направен шаблона ще може да се използва както за добавяне на нова рецепта, така и за редактиране на съществуваща. В темплейта ще се използва метода за двупосочното обновяване на данните специфично за

AngularJS с помощта на *ng-model*, което ще позволи информацията в полетата да се подава към модела, така и модела да попълва информация в тях (Приложение № 4):

```
<input type="text" ng-model="recipeName" class="recipe-info"
      required id="title_field" autocomplete="off"
>
```

На текстовото поле за въвеждане на име на рецептата ще се добави атрибут *required*. Така ще се избегне записването на рецепта без име. Ако потребителя се опита да запази рецепта, без да е въвел име, ще му се покаже съобщение, което да го уведоми, че това поле е задължително.

Действията на бутоните „Изтрий“ и „Добави“ ще бъдат вързани с контролера чрез използване на директивата *ng-click*. По този начин ще се посочва, която точно функция от контролера ще се изпълнява:

```
<button type="button" class="btn" ng-click="addIngredient()">
  <i class="icon-clipboard-add2"></i> Добави
</button>
```

Бутона за запазване на информацията „Запази“ ще бъде обикновен от типа *submit*. По този начин ще може да се използва функционалността на AngularJS за запамятаване на форми. Цялата структура на темплейта е поставен във форма и по този начин прашането на цялата информация към контролера ще се постигне лесно помощта на *ng-submit*:

```
<form name="recipeForm" ng-submit="saveRecipe()">
  ...
  <div class="btn-wrap">
    <button type="submit" class="btn">
      <i class="icon-clipboard-checked2"></i> Запази
    </button>
  </div>
</form>
```

Връзката между темплейта и модела ще се осъществи от контролера *cb-recipe.controller.js*. При неговото създаване трябва да се запишат всички допълнителни функционалности и сървиси на който ще разчита:

```
(function () {  
  'use strict';  
  var app = angular.module("cookingBook.singleView");  
  app.controller("RecipeController",  
    ["$scope", '$rootScope', '$stateParams', '$location', 'cbRecipeService', 'cbSingleViewService',  
     function($scope, $rootScope, $stateParams, $location, cbRecipeService, cbSingleViewService){  
       ...  
     }]);  
})();
```

Този контролер разчита на глобалните обекти и сървис идващ и от самия фреймуърк - *\$scope*, *\$rootScope*, *\$stateParams*, *\$location* и два сървиса от самото приложение - *cbRecipeService*, *cbSingleViewService*. Трябва да се обърне внимание, как се подават DI на самата функция. Има масив, в който всички необходими обекти и сървиси се записват първо като стринг и после се подават на самата функция, която също е част от масива. Това се прави с цел, когато кода бъде оптимизиран и задачата за загрозяване и минимизиране на GruntJS uglify се изпълни, да не наруши функционалността. Когато един JavaScript код се минимизира, използваните параметри и променливи не запазват първоначалния си вид, а се намаляват до еднобуквен символ. Това обаче може да доведе до сериозни проблеми за самата функционалност на AngularJS. Когато всички важни параметри са записани под формата на стринг, те остават непроменени и служат като показател, какво означават параметрите във функцията. Затова се препоръчва този подход на писане.

Първото, което ще се запише в този контролер, е проверка, как е извикан шаблона - дали чрез бутона от навигацията в лявата част на приложението за добавяне на рецепта или с бутона за редактиране. Тази информация идва от глобална променлива в AngularJS *\$stateParams.recipeID*, с която се взимат данните от навигационния адреса (url), с който се зарежда шаблона. Това са правилата за адресите, които са дефинирани в *app.routes.js*. От

този параметър контролера ще прецени дали полетата на шаблона ще се попълнят с информация за конкретна рецептата, за да може потребителя да въведе корекции, както и коя е точно рецептата. Ако параметърът е празен, няма стойност, това ще означава, че шаблона ще бъде зареден с празна информация, за да може да се въведе нова рецепта :

```
var currentRecipe = cbSingleViewService.findRecipe($stateParams.recipeID, $scope.recipeList);
if ($stateParams.recipeID && currentRecipe !== null) {
    $scope.currentID = currentRecipe.id;
    $scope.recipeName = currentRecipe.name;
    $scope.recipeDescriptionField = currentRecipe.description;
    $scope.ingredientsList = currentRecipe.ingredients;
} else {
    //нулиране на стойност на ingredients обект
    $scope.ingredientsList = [{}];
}
```

Бутона за запазване на информацията на рецептата е един и същ и за двете ситуации - запазване и редактиране. Функцията, която ще прави връзката между темплейта и функционалността в сървиса за двете действия в контролера, също ще е обща - *saveRecipe*. Тук също ще се добави проверка, ползваща *\$stateParams.recipeID*, за да прецени, кое действие да се извърши, запазване или редактиране:

```
$scope.saveRecipe = function () {
    var updatedRecipeList;
    if($stateParams.recipeID){
        updatedRecipeList = cbRecipeService.updateRecipe(
            $scope.recipeName, $scope.ingredientsList,
            $scope.recipeDescriptionField, $stateParams.recipeID
        );
        $location.path('/singleView/' + $stateParams.recipeID);
    } else {
        updatedRecipeList = cbRecipeService.addRecipe(
            $scope.recipeName, $scope.ingredientsList,
            $scope.recipeDescriptionField
        );
    }
    $rootScope.recipeList = updatedRecipeList;
    $scope.recipeName = null;
    $scope.recipeDescriptionField = null;
    $scope.ingredientsList = [ {} ];
};
```

В условната конструкция първо се проверява дали шаблона е бил зареден с конкретна рецепта в навигационния адрес. При този случай след нанасяне на корекции по съществуваща рецепта, чрез извикване на съответната функция от сървиса (`cbRecipeService.updateRecipe`) и запамятаване на промените, ще се препраща обратно към темплейта, който ще показва вече обновените данни за рецептата *singleView.html*. Съответно ако темплейта се зареди без конкретна рецепта ще извика функцията от сървиса `cbRecipeService.addRecipe`. Независимо обаче, коя от двете функции е извикана, след запазването на данните ще се обнови и информацията в лявата колона на приложението, със списъка на наличните рецепти (*\$rootScope.recipeList*).

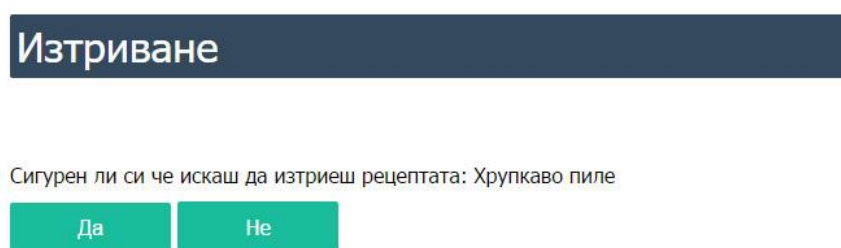
Всяка рецепта има различен брой съставки. Затова ще се създаде функция за добавяне на нов ред с рецепти `addIngredient`. Тя е сравнително лесна. Декларира се масив, в който да се пазят продуктите на рецептата (*\$scope.ingredientsList*), при натискане на бутона „Добави“ от потребителя се добавя още един нов празен обект за съставките на рецептите:

```
$scope.addIngredient = function() {  
    var ingredients = $scope.ingredientsList;  
    ingredients[ingredients.length] = {};  
};
```

По този начин в темплейта автоматично ще се появява празен ред, за добавяне на нова съставка. При всяко добавяне на нов ред за продукти се добавя и бутон за изтриване на съответния ред. Премахването на реда става още по-лесно. При натискане на бутона „Изтрий“ се подава индекса на реда и така лесно може да се намери, коя съставка точно трябва да се изтрие. Съответно след изпълнението на функцията, ще се обнови автоматично и темплейта:

```
$scope.removeIngredient = function(index) {  
    $scope.ingredientsList.splice(index, 1);  
};
```

Когато потребителя поиска да изтрие някоя рецепта трябва да му се зареди темплейта за изтриване, който ще е сравнително прост. Ще има две секции - една, която ще изисква от потребителя да потвърди триенето на рецептата и друга, която ще показва потвърдително съобщение, че рецептата е изтрита. Първата секция ще се показва при зарежда на темплейта. Потребителя ще бъде питан дали е сигурен, че иска да изтрие рецептата или не (Фиг. 15).



Фиг. 15

При изтриване на рецептата ще се показва втората секция с потвърдително съобщение, че изтриването е успешно и ще скрива предната секция (Приложение №5). Управлението на това, коя секция да се показва, ще се постигне с помощта на директивите на фреймуърка *ng-show* и *ng-hide*. Чрез подаване на параметри към тях, в зависимост дали тяхната стойност се приема за вярна или не, ще се покаже определена част от темплейта, в правилния момент. В случай, че потребителя реши да не изтрие рецептата, той ще бъде връщан към изгледа на самата рецепта. Навигацията към конкретната рецепта ще се осъществи отново чрез готовата директива на AngularJS - *ui-sref*.

```
<div ng-hide="confirmDeleting == 1" class="deletedRecipe">
  <p>Сигурен ли си че искаш да изтриеш рецептата: {{recipeName}} </p>
  <div>
    <button type="button" class="btn" ng-click="removeRecipe(currentID)">Да </button>
    <a ui-sref="singleView({recipeID: currentID})" class="btn">Не </a>
  </div>
</div>
<p ng-show="confirmDeleting == 1">Рецептата е изтрита</p>
```

Показването на този шаблон от контролера отново ще се извършва чрез проверка на навигационния адрес, за да се знае дали ще се добавя нова рецепта или ще се трие. Тази проверка ще се добави преди вече наличната, за това дали ще се редактира или добавя рецепта. Условната конструкция в този случай ще проверява друг параметър от навигационния адрес. Няма да ползва параметъра *\$stateParams.recipeID*, а частта преди него – името, което е въведено за съответния адрес в *app.routes.js* (свойството *templateUrl*). Тук може да се добави и проверка дали идентификатора на рецептата съдържащ се в адреса отговаря на налична рецепта и ако не да покаже темплейта за грешка *404.html* (Приложение №6).

```
var viewUrl = $location.path().split('/');
//ако рецептата е вече изтрита
if(currentRecipe === null && viewUrl[1] !== 'addRecipe' ){
    $location.path('/404');
}else if (viewUrl[1] === 'delete') {
    $scope.templateTitle = "Изтриване";
} else {
    $scope.templateTitle = $stateParams.recipeID ? 'Редактирай рецепта' : 'Добави рецепта';
}
```

Самата функция за изтриване на рецептата *removeRecipe* ще извиква съответната функция от сървиса (*cbRecipeService.deleteRecipe*), която вече беше разгледана. Резултата, който тя връща ще се използва като параметър в темплейта *\$scope.confirmDeleting*, за да може директивата *ng-show* да покаже потвърдителното съобщение. Списъка със всички рецепти от лявата страна на приложението също ще се обнови чрез *\$rootScope.recipeList*:

```
$scope.removeRecipe = function(recipeID) {
    var deletingData = cbRecipeService.deleteRecipe(recipeID);
    $scope.confirmDeleting = deletingData.confirmDeleting;
    $rootScope.recipeList = deletingData.updateRecipeList;
};
```

4.2.3. Функционалност на модул за изглед на единична рецепта - `cb-singleView`

При създаване на модула за преглед на конкретна рецепта се започва с файла `cb-singleView.module.js`, където се посочват допълнително ползваните модули, тези на които ще разчита конкретната функционалност ако има такива. Отново ще се спази правилото за наименоуване на подмодулите:

```
(function () {  
  'use strict';  
  
  angular.module('cookingBook.singleView', [  
    'ui.router'  
  ]);  
})();
```

След това може да се премине към самия сървис `cb-singleView.service.js`. Той ще е сравнително малък и ще има само една функция, понеже единственото, което трябва да прави, е да връща информацията за конкретната рецепта. Функцията `findRecipe` ще получава като входни параметри уникалния номер на рецептата `recipeID` и списъка със всички рецепти `allRecipes`. С цикъл ще се обхожда целия списък, за да се намери търсената рецепта и ще връща обект със нейните данни:

```
$this.findRecipe = function(recipeID, allRecipes) {  
  for (var i = 0; i < allRecipes.length; i++) {  
    if (allRecipes[i].id == recipeID){  
      $this.foundRecipe = allRecipes[i];  
      break;  
    }  
    $this.foundRecipe = null;  
  }  
  return $this.foundRecipe;  
};
```

Темплейта, който ще зарежда данните ще трябва да показва името на рецептата, нейните съставки и описанието. Всички тези данни ще се показват с двупосочното обновяване на информацията ползвайки фигурни скоби (`{{ }}`)

```
<h1>{{name}}</h1>
```

Броят на използваните продукти за всяка рецепта са различни. Затова ще се показват динамично. За по-прегледно ще се показват в таблица. Подобно на принципа на показване на списъка с рецепти в лявата секция на приложението и тук темплейта ще съдържа само единия ред от таблица, като ще се добавя нов динамично с директивата *ng-repeat* спрямо броя на продуктите в рецептата (Приложение №7).

```
<tr ng-repeat="ingredient in ingredients" ng-hide="getLength(ingredient) == 1">
  <td>{{ingredient.amount}}</td>
  <td>{{ingredient.amountUnits}}</td>
  <td>{{ingredient.ingredientName}}</td>
</tr>
```

Бутоните за редактиране и изтриване на рецептата ще се добавят тук. При натискането им те ще водят до съответните шаблони, които вече бяха описани при модула *cookingBook.recipe* (4.2.1). Функционалностите от този модул (*cookingBook.recipe*) изискват да знаят уникалния номер на съответната рецепта. Това се постига като отново се използва директивата *ui-sref* и се посочва адреса, който трябва да се извика и параметъра му. По този начин след натискане на съответния бутон, направената конфигурация за адресите на приложението в *app.routes.js* ще зареди правилно шаблоните от модула *cookingBook.recipe* и ще им предаде цялата нужна информация. В конкретния случай подавания параметър ще е идентификатора на конкретната рецепта (*id*):

```
<a ui-sref="editRecipe({recipeID: id})" class="btn">
  <i class="icon-clipboard-edit2"></i> Редактирай
</a>
<a ui-sref="deleteRecipe({recipeID: id})" class="btn">
  <i class="icon-clipboard-remove2"></i> Изтрий
</a>
```

Остава да се направи връзката между темплейта и сървиса чрез контролера *cb-singleView.controller.js*. В него ще трябва да се извика функцията от сървиса *findRecipe*, за да се получат данните за конкретната рецепта и да се

покажат от шаблона. Тук също може да се добави проверка за показване на темплейта за грешка 404.html, ако потребителя се опитва да достъпи несъществуваща рецепта.

```
var currentRecipe = cbSingleViewService.findRecipe(currentRecipeId, $scope.recipeList);
if(currentRecipe === null ) {
    $location.path('/404');
}else{
    $scope.name = currentRecipe.name;
    $scope.ingredients = currentRecipe.ingredients;
    $scope.description = currentRecipe.description;
    $scope.id = currentRecipeId;
}
```

4.2.4. Функционалност за търсене - cb-search

Последният модул на приложението ще позволи търсенето на рецепта по определени продукти. Както за всички останали така и за този модул ще трябва да се създаде файл `cb-search.module.js`, в който ще се дефинира името му *cookingBook.search* и ще се посочат допълнителните модули, на които разчита, ако има такива:

```
(function () {
    'use strict';

    angular.module('cookingBook.search ', [
    ]);
})();
```

Сървиса *cb-search.service.js* на този модул трябва да може да търси рецепта по зададени продукти и да връща списък с всички използвани уникални продукти в рецептите, т.е. ако даден продукт го има в няколко рецепти той ще се съдържа в списъка само веднъж. Наличните уникални съставки на рецептите ще се показват на потребителя, за да се знае какво може да въвежда в полето за търсене. За да може кода да е изчистен и разбираем, функцията за търсене *returnFoundRecipes* ще разчита на две спомагателни функции *search* и *checkRecipe*. Първата ще обработва списъка с продукти който е въведен и по който ще се търси рецептата. За всеки един продукт ще извиква функцията *checkRecipe*, която ще провери, в коя рецепта се използва

този продукт и ще създава нова колекция списък с рецепти *foundRecipesId*, в които се използва конкретния продукт.

```
$this.checkRecipe = function (entryValue, allRecipes) {
    for (var i = 0; i < allRecipes.length; i++) {
        var recipeIn = $this.ingredientsList(allRecipes[i].ingredients);
        if (recipeIn.indexOf(entryValue) > -1) {
            if ($this.foundRecipesId.indexOf(allRecipes[i].id) == -1) {
                $this.foundRecipesId.push(allRecipes[i].id);
            }
        }
    }
};

$this.search = function(allIngredients, availableRecipes) {
    var input = allIngredients.split(',');
    $this.foundRecipesId = [];
    for (var x = 0; x < input.length; x++) {
        var currentInput = input[x].trim();
        $this.checkRecipe(currentInput, availableRecipes);
    }
};
```

Търсенето на рецепта ще е възможно по няколко въведени съставки едновременно и съответно всяка съставка може да се използва в няколко рецепти едновременно. За да се избегне връщането на списък със дублиране на рецептите във функцията *returnFoundRecipes* ще се взима резултата от предните две и ще се проверява дали има повтаряща се рецепта. Накрая ще върне изчистен от повторения списък с рецепти:

```
$this.returnFoundRecipes = function(input, recipeList) {
    $this.search(input, recipeList);
    $this.allFoundRecipes = [];
    for (var i = 0; i < recipeList.length; i++) {
        if ($this.foundRecipesId.indexOf(recipeList[i].id) > -1) {
            $this.allFoundRecipes.push(recipeList[i]);
        }
    }
    return $this.allFoundRecipes;
};
```

Функцията за извеждане на всички уникални продукти *availableUniqueIntegrates* също ще извиква допълнителна функция *ingredientsList*, за да може да се спази конвенцията за писане на кратки и лесни

за разчитане парчета код. Тя ще има входен параметър списъка със всички съставки на една рецепта *recipeIntegrates*. Ще намира името на всеки продукти и ще го записва в масив *allRecipeIn*.

```
$this.ingredientsList = function(recipeIntegrates) {
    $this.allRecipeIn = [];
    for( var item in (recipeIntegrates)){
        $this.allRecipeIn.push((recipeIntegrates)[item].ingredientName);
    }
    return $this.allRecipeIn;
};
```

Така функцията за намиране на всички уникални съставки ще минава през целия списък с рецепта и за всяка една ще взема продуктите ѝ и ще извиква допълнителна функция *ingredientsList*. Ще обработва получения резултат, като от него ще се попълва нов масив *allUniqueIntegrates*, в който всички продукти ще ги има само по един път.

```
$this.availableUniqueIntegrates = function (allRecipesList) {
    $this.allUniqueIntegrates = [];
    for (var i=0; i < allRecipesList.length; i++) {
        var arrayIngredients = $this.ingredientsList(allRecipesList[i].ingredients);
        for (var y = 0; y < arrayIngredients.length; y++){
            if ($this.allUniqueIntegrates.indexOf(arrayIngredients[y]) == -1) {
                $this.allUniqueIntegrates.push(arrayIngredients[y]);
            }
        }
    }
    return $this.allUniqueIntegrates;
};
```

В темплейта *search.html* за търсенето на рецепти ще се създаде форма със стандартно текстово поле (*input*) за въвеждане на продукти и бутон за търсене. И тук както при *saveRecipe.html* ще се използва директивата *ng-submit*, която ще позволи контролера да знае какви данни са попълнени във формата и да ги предаде на сървиса. Бутона на формата ще е от типа *submit*, стандартен за подобна конструкция:

```
<form ng-submit="searchRecipe(ingredientsInput)">
    <label for="search_field">Изброй продукти разделяйки ги с ", ": </label>
    <input type="text" id="search_field" ng-model="ingredientsInput" required>
```

```

<button type="submit" class="btn">
  <i class="icon-document-search2"></i> Търси
</button>
</form>

```

За показването на наличните продукт ще се използва директива *ng-hide*, която ще позволи, ако няма въведени рецепти и съответно продукти елемента да не се показва. И всички съставки ще се изписват, чрез фигурните скоби за двупосочно обновяване на информацията, а за да може динамично да се показват всички намерени продукти ще се използва *ng-repeat*:

```

<div class="gray-wrapper" ng-hide="isRecipeListEmpty">
  <span ng-repeat="ingredient in availableUniqueIntegrates">{{ingredient}}, </span>
</div>

```

За показването на списъка с намерените рецепти принципа ще бъде същия. Отново ще се използва *ng-hide*, за да може да не се показва елемента ако няма намерени рецепти и *ng-repeat* за динамично показване на всички рецепти (Приложение №8). Всяка рецепта ще бъде поставена в елемент линк (**), за да може от този списък потребителя директно да прегледа информацията за избраната рецепта. Навигацията отново ще се осъществи чрез *ui-sref*:

```

<div class="list-wrap" ng-hide="isFoundListEmpty">
  <ul>
    <li>
      <i class="icon-clipboard-list2"></i><p>Намерени рецепти</p>
    </li>
    <li ng-repeat="recipe in foundList">
      <a ui-sref="singleView({recipeID: recipe.id})"> {{recipe.name}}</a>
    </li>
  </ul>
</div>

```

В контролера `cb-search.controller.js` има две основни функционалности, които ще се изпълняват. Първата е при търсене (*searchRecipe*) да взима списъка с намерените рецепти, за да може шаблона да ги покаже:

```
$scope.searchRecipe = function (ingredientsInput) {  
    // зануляване на спусъка с намерене рецепти  
    $scope.foundList = [];  
    $scope.foundList = searchService.returnFoundRecipes(ingredientsInput, $scope.recipeList);  
    $scope.isFoundListEmpty = $scope.foundList.length === 0;  
};
```

Втората функция *availableUniqueIntegrates* ще взема от сървиса всички уникални рецепти, за да може темплейта да покаже и тях:

```
$scope.availableUniqueIntegrates = searchService.availableUniqueIntegrates($scope.recipeList);
```

4.3. Създаване на тестове за приложението

Функционалността за приложението е създадена. Следващата стъпка е да се премине и към тестовете. Те са неделима част от цикъла на разработване на всеки SPA проект. Стъпките, които трябва да се изпълнят са:

- да се направи конфигурация
- да се организира методика за съхраняване на файловете с тестовете
- да се създадат тестове.

4.3.1. Конфигурация

Според описанието във втора глава, първо трябва да се инсталира средата *Karma*. За целта трябва да се инсталира интерфейса с команден ред, като е препоръчително това да стане глобално за машината, на която се разработва с командата *npm install -g karma-cli*. След това може да се инсталира и самата Karma, отново през NPM с флаг *--save-dev*, за да се запази в конфигурационния файл на проекта *package.json* (*npm install karma --save-dev*). Остава да се инсталират помощни модули както за изпълнението така и за писането на тестовете:

- Инсталиране на Jasmine - фреймуърк, с който ще се пишат тестовете. За него има два модула, които да се инсталират *jasmine-core* и *karma-jasmine*

- Инсталиране на модули за пускане на тестовете в браузъри - по този начин при стартиране на тестовете те сами ще се изпълняват в нужните браузъри, без необходимост ръчното им стартиране. Нужни са модули за най-често използваните браузъри - Chrome (*karma-chrome-launcher*), Mozilla Firefox (*karma-firefox-launcher*) и Internet Explorer (*karma-ie-launcher*). При всяко стартиране на тестовете ще се стартира всеки един от тези браузъри. В процеса на разработка това не винаги е оптимално, затова има и модул, който изпълнява тестовете в браузър, но на заден фон, без да пречи. За целта трябва да се свали *PhantomJS*, като нужните модули за това са *karma-phantomjs-launcher* и *phantomjs-prebuilt*.
- Инсталиране на модул за проверка на обхвата на тестовете - за да са от полза тестовете трябва да има начин, по който да се измерва каква част от кода покриват те. Един от вариантите е с модула *karma-coverage*. Той ще дава информация при всяко стартиране на тестовете колко процента от кода се покриват, която ще се изписва директно в конзолата при приключване на тестовете. Освен това ще създава така наречените доклади (*reports*). Това са файлове, чрез който може лесно да се провери за всяко парче код дали има тестове за него или не.
- Инсталиране на AngularJS модул - За всяко приложение работещо с данни е от критично значение операциите работещи с тях да функционират коректно. Тестването им обаче не е много лесно, тъй като това може да доведе до ненужни нови записи в базата данни. Модула *angular-mocks* позволява да се създават псевдо данни само за нуждите на тестовете (*moks*), които имитират поведението на истинските. С него става възможно и тестването на специфичните за AngularJS компоненти - контролери (*controllers*), сървиси (*services*), филтри (*factories*) и директиви (*directives*)

Всички тези модули се свалят чрез NPM и се инсталират с допълнителен флаг към командата `--save-dev`. След това се преминава към конфигурационния файл на Karma, като се използва `karma init` командата. При наличието вече на готов конфигурационен файл, ръчно може да се направят допълнителни фини настройки. Така например където са записани браузърите, на които ще се пускат тестовете може да се добави един допълнителен ред :

```
// start these browsers
// available browser launchers: https://npmjs.org/browse/keyword/karma-launcher
// browsers: ['Chrome', 'Firefox', 'IE'],
browsers: ['PhantomJS'],
```

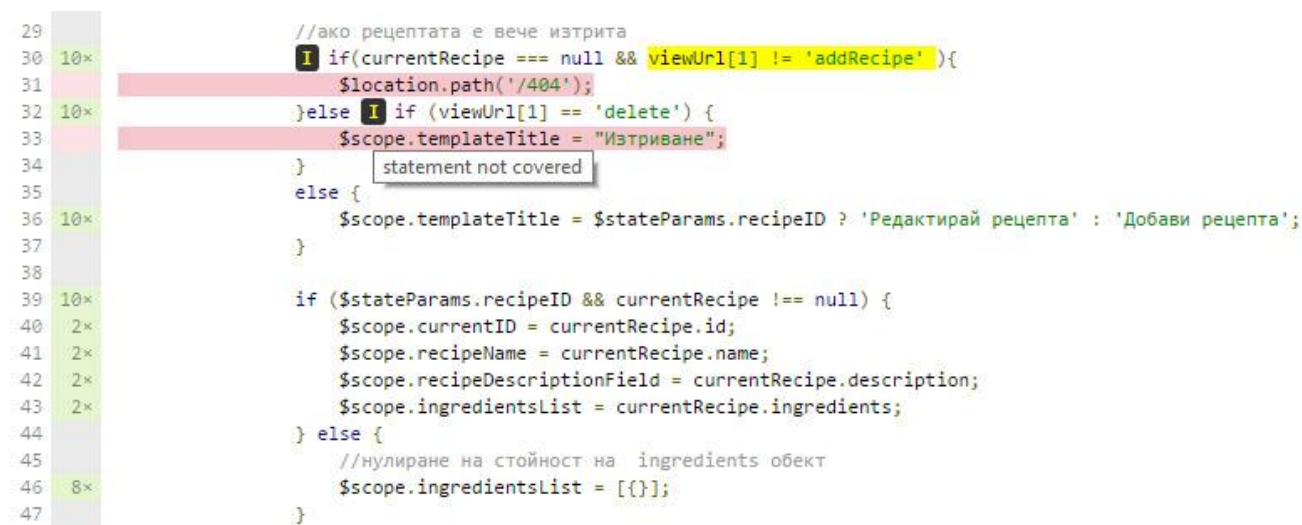
По този начин когато се пишат тестовете е достатъчно да се остави само PhantomJS. Щом писането на теста е завършено този ред съответно да се коментира и да се откоментира горния, за да се изпълнят тестовете на желаните браузъри.

Конфигурацията за обхвата на тестовете също трябва да се допълни ръчно. Има различни варианти и зависи от нуждите на проекта. Един от простите и изчистен вариант е да се генерира автоматично папка *coverage*, където да се съхраняват всички доклади от тестването във html формат (Приложение №9).

```
coverageReporter: {
  // директория където да се съхранява резултата
  dir: 'coverage',
  reporters: [
    //директория за html докладите (report)
    {type: 'html', subdir: 'html'},
    {type: 'text-summary', subdir: '.'}
  ]
},
```

От тези доклади може да се види подробно за всеки файл от функционалността на приложението дали има пропуснат ред от тестовете и кой точно. Самите доклади се запазват в html формат, което ги прави лесни за отваряне в браузъра и поддържане на функционалност като *hover*, т.е. при

поставяне на мишката върху маркиран участък с жълт или червен цвят ще се покаже подсказка за значението на този участък (Фиг. 16).



```

29 //ако рецептата е вече изтрита
30 10x if(currentRecipe === null && viewUrl[1] != 'addRecipe'){
31     $location.path('/404');
32 10x }else if (viewUrl[1] == 'delete') {
33     $scope.templateTitle = "Изтриване";
34     statement not covered
35 }
36 10x else {
37     $scope.templateTitle = $stateParams.recipeID ? 'Редактирай рецепта' : 'Добави рецепта';
38 }
39 10x if ($stateParams.recipeID && currentRecipe !== null) {
40     2x $scope.currentID = currentRecipe.id;
41     2x $scope.recipeName = currentRecipe.name;
42     2x $scope.recipeDescriptionField = currentRecipe.description;
43     2x $scope.ingredientsList = currentRecipe.ingredients;
44 } else {
45     //нулиране на стойност на ingredients обект
46     8x $scope.ingredientsList = [{}];
47 }

```

Фиг. 16

Стартирането на тестовете е изключително лесно. След като всички конфигурации са готови е необходимо да се изпълни команда в конзолата *karma start*. Отново в конзолата ще може да се види как са приключили тестовете, дали има някаква грешка, къде се намира тя. (Фиг. 17).

```

d:\wamp\www\cooking-book>karma start
26 08 2016 21:15:31.580:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/
26 08 2016 21:15:31.587:INFO [launcher]: Starting browser PhantomJS
26 08 2016 21:15:33.209:INFO [PhantomJS 2.1.1 (Windows 8 0.0.0)]: Connected on socket /#HbWQ2YI6V
JOWK9JMAAAA with id 44414032
PhantomJS 2.1.1 (Windows 8 0.0.0): Executed 60 of 60 SUCCESS (0.13 secs / 0.144 secs)

```

Фиг. 17

4.3.2. Методика за съхраняване на файловете

Самите файлове с тестове са JavaScript. По правило те се наименоуват по същия начин, по който се казва файла, чийто код ще се тества, като се добавя *.spec* към името. Така на пример ако се тества *cb-singleView.service.js* то файла, в който ще се напишат тестовете за него ще се казва *cb-*

singleView.service.spec.js (Фиг. 10). Два са основните принципи при съхраняване на файловете с тестове [7]:

- Всички тестове в отделна папка в проекта, която има името *tests*
- Всеки тест се съхранява в съответната папка където се намира файла, за който са написани тестовете

Разглеждайки документацията на AngularJS се вижда, че тяхната препоръка е именно втория подход. Затова и в текущото приложение ще бъде използван точно той. Плюсовете на тази организация са [7]:

- Лесно може да се намери конкретен файл с тестове
- Лесно може да се види дали липсват тестове
- Останалите тестове в близост помагат да се разбере как дадена функционалност работи
- Когато се преименува файла със кода не се забравя да се преименува и файла с тестовете

След изясняването на тази структура не трябва да се забравя, че трябва да се обнови и *Gruntfile.js* файла. Текущата конфигурация в него за оптимизиране броя на файловете и събиращ ги в едно следи за всички файлове, които имат разширение в името *.js*. По този начин на условието отговарят и файловете с тестовете. В крайния файл обаче не трябва да влизат тези с тестовете, защото те са излишни за самата функционалност и не са нужни. За това трябва да се допише ред в конфигурацията, която изрично да посочва всички файлове завършващи на *.spec* да не се включват.

```
dist: {
  src: [
    //първо modules
    'src/*.module.js', 'src/**/*.module.js',
    //след това routes
    'src/*.routes.js', 'src/**/*.routes.js',
    //след това всички останали
    'src/*.js', 'src/**/*.js',
    //външни ресурси
    '!src/*.spec.js', '!src/**/*.spec.js'
  ],
  dest: 'build/js/<%= pkg.name %>.js'
},
```

Същата промяна трябва да се направи и за задачата `watch`. Не е нужно при промяна във файловете с тестове да се извикват отново всички останали операции. Обикновено при писането на тестове задачите на GruntJS не работят. Независимо от това е добре в задачата `watch` да се посочи, че тестовите файлове не трябва да се следят:

```
watch: {  
  js: {  
    files: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js'],  
    tasks: ['jshint:beforeconcat', 'concat:dist']  
  },  
  scss: {  
    files: ['scss/*.scss'],  
    tasks: ['sass']  
  }  
}
```

4.3.3. Създаване на тестове

При писането на тестовете се използва помощта на фреймуърка Jasmine. Основните компоненти, от които се създава един тест са [14]:

- Група от тестове (*Suites*) - Основна градивна структура, чрез нея се групират отделните тестове. Състои се от низ (описание на теста) и функция (самите тестове):

describe(string, function)

- Конфигурация на тест (*Specs*) - градивната структура на всеки тест. Състои се от име и функция, която съдържа едно или повече предположения (очаквания):

it(string, function)

- Предположения (*Expectations*) - чрез тях се проверява дали конкретната функционалност връща очаквания резултат. Предположенията винаги връщат резултат от тип булева стойност *true* или *false*:

expect(actual).toBe(expected)

- Предварително дефинирани общи твърдения - тази конструкция прави оценки и тя решава дали тестът е успешен или не:

toEqual(expected)

- Подготвителна процедура - градивна структура на тестовете, в нея се извършват подготвителни операция за всяка група от тестове. Тази процедура се поддържа преди (*beforeEach*) и след (*afterEach*) всеки тест (*specs*):

```
beforeEach(function() {           afterEach (function() {
    foo = 0;                        foo = 0;
    foo += 1;                       }
})
```

Това, което се тества в AngularJS приложенията са всички файлове без тези завършващи на `module.js` и `router.js`. Всеки файл с тестове започва с едно групиране на всичко в него с *describe*. Всяка отделна под група тестове също започва с *describe* и описание на какво точно тества конкретната група. По този метод се описват както файл, така и всяка функционалност, които се тестват и ако се появи грешка, в конзолата ще се появят тези описания. Така лесно ще може да се намери конкретния провалил се тест и да се провери защо (Фиг. 18).

```
d:\wamp\www\cooking-book>karma start
26 08 2016 22:00:15.964:INFO [karma]: Karma v0.13.22 server started at http://localhost:9876/

26 08 2016 22:00:15.964:INFO [launcher]: Starting browser PhantomJS
26 08 2016 22:00:17.564:INFO [PhantomJS 2.1.1 (Windows 8 0.0.0)]: Connected on socket /#MFR6I6x-
w000IlXKAAAA with id 47754249
PhantomJS 2.1.1 (Windows 8 0.0.0) Controller: SingleViewController currentRecipe is
NOT empty all scope variables should be defined correctly $scope.description FAILED

Expected 'string' to be 'object'.

d:/wamp/www/cooking-book/src/cb-singleView/cb-singleView.controller.spec.js:9:5058
PhantomJS 2.1.1 (Windows 8 0.0.0): Executed 60 of 60 (1 FAILED) (0.171 secs / 0.147 secs)
```

Фиг. 18

Първия тест ще е на сървиса `cb-singleView.service.js` и както вече беше споменатото според конвенцията за наименоуване на файловете, ще се ползва следното име на файла с тестовете - `cb-singleView.service.spec.js`. Започва се със създаване на първи описателен блок:

```
describe('Service: cbSingleViewService', function() {  
    ...  
});
```

Описанието на тестовете трябва да е кратко и достатъчно ясно. В случая теста е за сървис затова започва с *Service:* и след това неговото име. В тази конструкция ще се поместят всички тестове за него. Преди самите тестове трябва да се създадат подготвителните процедури за модула и за самия сървис.

```
beforeEach(module('cookingBook.singleView'));  
  
var cbSingleViewService,  
    mockInitData;  
  
beforeEach(inject(function(_cbSingleViewService_) {  
    cbSingleViewService = _cbSingleViewService_;  
}));
```

Първата подготвителна процедура е за модула, който се тества, за да бъде достъпен в тестовете. След това се прави същото и за самия сървис.

След това се тества дали сървиса съществува и е достъпен за тестване. Така ще се провери дали конфигурацията работи.

```
it('Should have a service', function () {  
    expect(cbSingleViewService).not.toEqual(null);  
});
```

За всяка функция е добра практика да се създава отделен групиращ блок за тестовете. В конкретния сървис има само една функция затова ще се създадем само един нов групиращ блок:

```
describe('Get findRecipe call', function() {  
    ...
```

```
});
```

Разглеждайки самата функция *findRecipe* в сървиса се вижда, че има два входни параметъра - уникален номер на рецептата (*recipeID*) и списък с наличните рецепти (*allRecipes*). Както беше вече споменато при работа с данни, е добра практика да се създават примерни такива в тестовете. Затова ще добавим още една подготвителна процедура, която ще създаде обект с налични рецепти само за нуждите на тестовете, без да има достъп до истинските. Тази процедура ще се помести в групиращи блок за самата функция *findRecipe*.

```
beforeEach(function() {
  mockInitData = [
    {
      id: 1,
      name: "Кекс",
      description: "някакво описание",
      ingredients: [
        {"ingredientName": "масло", "amount": "100", "amountUnits": "гр" },
        {"ingredientName": "захар", "amount": "100", "amountUnits": "гр"}
      ]
    },
    {
      id: 2,
      name: "Хрупкаво пиле",
      description: "Пече се на фирна на 175 градуса.",
      ingredients: [
        {"ingredientName": "корнфелкс", "amount": "1", "amountUnits": "чаша" },
        {"ingredientName": "пармезам", "amount": "3/4", "amountUnits": "чаша" }
      ]
    },
    {
      id: 3,
      name: "Чийзкейк",
      description: "Пече се на фирна на 175 градуса.",
      ingredients: [
        {"ingredientName": "яйца", "amount": "2", "amountUnits": "бп" },
        {"ingredientName": "захар", "amount": "1/2", "amountUnits": "чаша" }
      ]
    },
    {
      id: 4,
      name: "Мъфини",
      description: "Комбинирайте заквасена сметана и захар; Смесете добре. Добавете кокос и разбъркайте.",
      ingredients: [
        {"ingredientName": "заквасена сметана", "amount": "2", "amountUnits": "чаша" }
      ]
    }
  ];
});
```

Първия тест (*specs*), който ще се създаде ще проверява дали функцията връща очакван резултат, когато има правилно подаден уникален номер на рецепта. За целта в самия тест ще се създаде параметър *repeiptId*, който ще играе ролята на идентификатора. Стойността на параметъра, трябва да отговаря на стойност *id* от вече направения обект със псевдо данни *mockInitData*.

```
it('Should have call to return specific receipt by ID', function () {
    var repeiptId = 1;
    ...
});
```

Първото предположение (*Expectations*), което трябва да се добави в теста, е дали функцията *findRecipe* е извикана с правилните входни параметри. Това се постига чрез опция от Jasmine, която се ползва за „шпиониране“ (проследяване) на изпълнението на функциите *spyOn*:

```
spyOn(cbSingleViewService, 'findRecipe').and.callThrough();
cbSingleViewService.findRecipe(repeiptId, mockInitData);
expect(cbSingleViewService.findRecipe).toHaveBeenCalled(repeiptId, mockInitData);
```

Трябва да се провери и дали стойността, който се връща като резултат от функцията е от тип обект. Може да се провери също дали съдържа част от задължителните свойства на обекта като *id* и *name*. И понеже благодарение на псевдо данните се знае коя рецепта ще се върне като резултат от изпълнението на функцията, може да проверим дали стойностите на свойствата на обекта съвпадат:

```
expect(typeof cbSingleViewService.foundRecipe).toBe('object');
expect(cbSingleViewService.foundRecipe.id).toBeDefined();
expect(cbSingleViewService.foundRecipe.id).toEqual(repeiptId);
expect(cbSingleViewService.foundRecipe.name).toBeDefined();
expect(cbSingleViewService.foundRecipe.name).toEqual("Кекс");
```

Остава да се провери дали ако се подаде несъществуващ уникален номер на рецепта, приложението ще продължава да работи коректно. За целта

се създава нов тест, в който се декларира отново вътрешен само за него параметър *repeiptId*, на който се подава невалидна стойност. Отново ще се провери дали функцията *findRecipe* се извиква с правилните параметри, като след това ще има само една проверка. Резултата от функцията трябва да бъде *null*:

```
it('Should return NULL if ID does not match', function () {
  var repeiptId = 15;
  spyOn(cbSingleViewService, 'findRecipe').and.callThrough();
  cbSingleViewService.findRecipe(repeiptId, mockInitData);
  expect(cbSingleViewService.findRecipe).toHaveBeenCalledWith(repeiptId, mockInitData);
  // $rootScope.$apply();
  expect(cbSingleViewService.foundRecipe).toBeNull();
});
```

Тестването на контролера не се различава особено. Важните моменти при него са, че трябва да се тества само данните и функциите, които са негови. Функционалността идваща от сървиса не се тества. За нея подобно на данните също се създава с псевдо обект (*mocks*), т.е. създава се минималистична версия на функционалности, която да симулира поведението на сървиса:

```
beforeEach(function() {
  mockCurrentRecipe = {
    id: 1,
    name: "Кекс",
    ingredients: [
      {"ingredientName": "масло", "amount": "100", "amountUnits": "гп" },
      {"ingredientName": "захар", "amount": "100", "amountUnits": "гп"}
    ],
    description: "някакво описание"
  };
  mockedSingleViewService = {
    findRecipe: function () { return mockCurrentRecipe }
  };
});
```

Както и при предния тест, така и тук се започва с групиране на тестовете с *describe* и се задава име, създават се подготвителните процедури с *beforeEach* и ако трябва се дефинират променливи и им се подават стойности, създават се псевдо данни. Важното за тестването на контролерите, е че има

един задължителен блок *beforeEach*, с който се създава контролера за самите тестове:

```
beforeEach(inject(function($controller, _$rootScope_, $location){
    $rootScope = _$rootScope_;
    $scope = $rootScope.$new();
    location = $location;
    ctrl = $controller('SingleViewController', {
        $scope: $scope,
        $stateParams: stateParams,
        $location: location,
        'cbSingleViewService': mockedSingleViewService
    });
    $rootScope.$apply();
})));
```

След това е добре да се провери дали контролера съществува, за да се знае, че всичко е наред с конфигурацията.

```
it('ctrl should be defined', function() {
    expect(ctrl).toBeDefined();
});
```

Остава само да се провери дали всички променливи, който попълват данните в темплейта са правилно дефинирани и дали стойността им е от очаквания тип.

```
describe('all scope variables should be defined correctly', function() {
    it('$scope.name', function() {
        expect($scope.name).toBeDefined();
        expect(typeof $scope.name).toBe('string');
    });
    it('$scope.ingredients', function() {
        expect($scope.ingredients).toBeDefined();
        expect(typeof $scope.ingredients).toBe('object');
    });
    it('$scope.description', function() {
        expect($scope.description).toBeDefined();
        expect(typeof $scope.description).toBe('string');
    });
    it('$scope.id', function() {
        expect($scope.id).toBeDefined();
    });
    it('$scope.getLength', function() {
        expect($scope.getLength).toBeDefined();
        expect(typeof $scope.getLength).toBe('function');
    });
});
```

На този принцип се тестват всички файлове. Колкото повече тестове се приключват, толкова по-висок ще е и процента на кода, който покриват. Резултата ще се вижда в конзолата (Фиг. 19).

```
PhantomJS 2.1.1 (Windows 8 0.0.0): Executed 60 of 60 SUCCESS (0.13 secs / 0.144 secs)

===== Coverage summary =====

Statements   : 99.14% ( 690/696 )
Branches     : 88.24% ( 30/34 )
Functions    : 96.55% ( 168/174 )
Lines        : 99.14% ( 690/696 )

=====

d:\wamp\www\cooking-book>
```

Фиг. 19

Това, което се вижда от горната фигура е, че тестовете покриват 98.85% от условните конструкции (*Statements*), 88.24% от разклоненията (*Branches*), 96.55% от функциите (*Functions*) и 98.85% от общия брой редове в кода (*Lines*). Стойности от 85% и нагоре са показател за добре написани тестове.

4.4. Създаване на документация за проложението

Документацията е основополагаща част от всеки софтуерен продукт. Нейното създаване и поддържане често бива пренебрегвано или изпускано, защото е тежка и тромава задача, а понякога е почти равносилна на създаване на самостоятелен проект.

В проекти ползващи AngularJS има модул, който улеснява създаването на документация - *grunt-ngdocs*, който използва стандарта за писане на документиращи коментари в кода за JavaScript проекти - *jsdoc*. Стандарта *jsdoc* представлява система от правила, по които да се описват функции, променливи, обекти и въобще всичко в един JavaScript код. По този начин коментарите са еднотипни и разбираеми. Колкото и добре да е написан

или ясно четим един код, коментари описващи неговата функционалност са добра практика. Модула за AngularJS ползва тези коментари и след изпълнение на команда в конзола *grunt ngdocs* ще генерира документация в темплейт подобен на, този който ползва самата документация на фреймуърка. По този начин документацията на проекта ще се поддържа актуална. При промяна на кода, ще трябва да се промени и съответния коментар, а това ще обнови и съдържанието на документацията.

За интегрирането на модула в проекта е нужно първо да се инсталира по вече познатия начин, чрез NPM и със флаг `--save-dev`:

```
npm install grunt-ngdocs --save-dev
```

След това трябва в самия конфигурационен файл *Gruntfile.js* да се допише настройката за тази задача. Това, което трябва да се посочи, е в кои файлове модула да следи за коментари, за да генерира документацията. Това са всички файлове, които са *.js*, като изрично се посочва да не се взима предвид файловете на тестовете:

```
ngdocs: {  
  all: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js']  
}
```

Извикването на самата задача трябва да се добави и в командата за цялостно генериране и оптимизиране на всички файлове *grunt build* от раздел 4.1.3.4 (Приложение №2).

```
grunt.registerTask('build', [  
  'jshint:beforeconcat', 'concat',  
  'jshint:afterconcat', 'uglify', 'ngdocs'  
]);
```

Така при всяко генериране на проекта, ще бъде сигурно, че и документацията ще е актуална. Автоматично ще се генерира папка *docs* в

директорията на проекта, където ще се съхраняват всички генерирани файлове за документацията.

Освен това може да се добави генерирането на документация и в задачата `watch`. Така дори и по време на разработка, ако е нужно да се прегледа документацията, тя ще бъде актуална:

```
watch: {
  js: {
    files: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js'],
    tasks: ['jshint:beforeconcat', 'concat:dist', 'ngdocs']
  },
  scss: {
    files: ['scss/*.scss'],
    tasks: ['sass']
  }
}
```

Самата документация ще бъде достъпна на същия url адрес като този на приложението като се добавя само `/docs` към адреса. Лесно и удобно ще може да се проверява всяка функция, обект или параметър за какво служат и как се използват, дори и ако се стартира на локалната машина на програмиста (Приложение №10).

Основните данни, които трябва да има във всеки коментар, за да е полезен и да може документацията да се генерира правилно са [22]:

- Какъв е типа на парчето код, което се описва с параметър `@ngdoc`
- Името на парчето код, което се описва с параметър `@name`
- Ако се описва функция, метод на коя функционалност е `@methodOf`
- Описание на парчето код, което се документира с параметър `@description`
- За функциите се описват входящите параметри и се посочва техния тип във фигурни скоби (`@param {String}`)
- Ако функцията връща някаква стойност се описва и тя, с нейното име и типа, от който е (`@return {Object}`)

За да може вида на документацията да следва структурирана на проекта, в конкретния случай на модули, трябва да се опишат всички файлове - модули, сървиси и контролери. Така например, документиращия коментар за `cb-singleView.module.js` ще изглежда по следния начин:

```
/**
 * @ngdoc overview
 * @module cookingBook.singleView
 * @name cookingBook.singleView
 *
 * @description
 * Модул за управление на view за единична рецепта в приложението cookingBook.
 */
```

Както различните функционалности на приложението се обединяват в модули, така описвайки по този начин модула ще се създаде подобно групиране и в самата документация (Приложение №10).

В сървисите и контролерите се коментират важна и основна функционалност за приложението, за да се улесни разбирането как работи. В тези файлове се започва с поставяне на коментари и за самия сървис или контролер. За сървиса `cb-singleView.service.js` описващия го коментар ще изглежда по следния начин:

```
/**
 * @ngdoc service
 * @name cookingBook.singleView.service:cbSingleViewService
 * @module cookingBook.singleView
 * @description
 * Управление на зареждането на данните за конкретна рецепта
 */
```

И съответно за контролера `cb-singleView.controller.js`:

```
/**
 * @ngdoc controller
 * @name cookingBook.singleView.controller:SingleViewController
```

```
* @module cookingBook.singleView
* @description
* Управление за view за единична рецепта
*
*/
```

Спазвайки изискванията за документиране на методи, функцията в сървиса `cbSingleViewService` ще има следния коментар:

```
/**
 * @ngdoc property
 * @name currentRecipe
 * @propertyOf cookingBook.singleView.controller:SingleViewController
 * @description
 * Обект с данни за единична рецепта
 */
```

Добавянето на коментарите към кода не отнема много време. Обикновено при писането на функционалност винаги се добавя описание, а по този начин се въвеждат и правила за писането ѝ.

Показания тип на настройка на GruntJS задачата за документация (*ngdocs*) е от възможно най-лесния и бърз начин за създаването ѝ. Разбира се, всеки проект е различен и понякога изисква допълнителни специфични конфигурация на документацията. Този модул е достатъчно гъвкав и може да се използва и за по-специфични случаи, както и да се добавят допълнителни страници и навигации не идващи от записаните коментари.

Заклучение

Развитието на Интернет и всичко свързано с него предоставят все по-голяма възможност за разнообразие при разработката на приложения. В днешно време голяма популярност придобиват технологии базирани на JavaScript, а с това се появяват и много нови възможности. В представения проект беше разгледано приложение, разработено изцяло именно с такива технологии.

Тенденциите в разработването на уеб приложения е свързано все повече с ясно разделяне на клиентската част от сървърната. Все повече готови решения се появяват, които да са от помощ на разработчика, а водещите такива на пазара се развиват и предоставят все по-голяма функционалност.

Пазара на мобилни устройства също се разраства, появяват се различни операционни системи. Разработката на приложение за тях става все по-голямо предизвикателство и е трудоемко, защото трябва да се напише няколко пъти за различните среди. В отговор на това все по-често се предпочита използването на фреймуъркси като AngularJS за създаването им. Това позволява написания код да се стартира както в уеб среда, така и да се превърне в мобилно приложение за всяка от наличните операционни системи.

Разработеното приложение следва именно тези тенденции. От разгледаните архитектурни модели и стратегия за изграждане на JavaScript приложения се приложиха най-често използваните - SPA и MVC. По този начин беше постигната модулност на функционалността. В процеса на планиране стана ясна комплексността и сложността на функционалността, която може да се постигне с език за програмиране, първоначално създаден просто за направата на по-динамични страници. Използването на фреймуърк като AngularJS улесни изграждането на архитектурата на приложението. В разработката се приложиха и описаните похвати техники за оптимизиране както на проекта, така и на средата за разработка, посредством NPM и GruntJS. Така се постигна чиста и организирана среда на разработка, а в същото време

бяха създадени процедури, които лесно да подготвят приложението за пускане в експлоатация.

Показаха се и други две добри практики при разработване на подобни проекти - създаване на тестове и автоматично генериране на документация. Написаните тестове гарантират, че приложението работи достатъчно стабилно, както и че при промяна или добавяне на нова функционалност няма да се наруши правилното му функциониране. А чрез автоматично генерираната документация се подсигури, че за приложението тя винаги ще бъде актуална и ще се създава с минимални усилия.

Като краен резултат се получи интуитивно и лесно за употреба от потребителя приложение, изпълняващо се в браузър. В същото време остава независимо от сървърната технология, което води до много нови възможности. В бъдеще, за това уеб приложение, може да се изгради връзка с всяка сървърна среда или да се надгради и преобразува в мобилно, с минимални усилия от страна на всеки разработчик.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

1. Светлин Наков, Веселин Колев и колектив - Въведение в програмирането със С#. Академия на Телерик за софтуерни инженери София, 2011г. - Издателство: Фабер, Велико Търново, 2011 г.
2. Addy Osmani, Learning JavaScript Design Patterns, 2015.
3. Brad Green and Shyam Seshadri - AngularJS, 2013 - O'Reilly
4. Chris Sevilleja - A Simple Guide to Getting Started With Grun, scotch - April 4, 2014 <https://scotch.io/tutorials/a-simple-guide-to-getting-started-with-grunt>
5. GitHub <https://github.com/blog/2047-language-trends-on-github>
6. Guide to AngularJS Documentation - <https://docs.angularjs.org/guide/>
7. Guide to Angular Documentation - <https://angular.io/docs/>
8. Guide to angular-ui - <http://angular-ui.github.io/ui-router/site/>
9. Grunt: The JavaScript Task Runner - <http://gruntjs.com/>
10. John Papa blog - <https://johnpapa.net/pageinspa>
11. jshint - <http://jshint.com/docs/>
12. Karma Documentation - <https://karma-runner.github.io/1.0/index.html>
13. O'Reilly Media
http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html
14. Pluralsight - Hristo Georgiev, Introduction to Angular test-driven development
<http://tutorials.pluralsight.com/front-end-javascript/introduction-to-angular-test-driven-development?hearted=1>
15. Telerik Academy - Курс "JavaScript приложения",
<http://telerikacademy.com/Courses/Courses/Details/182>
16. Sandeep Panda - AngularJS: Novice to Ninja, sitepoint, 2014
17. Sandeep Panda - A Practical Guide to AngularJS Directives - January 14, 2014, sitepoint <http://www.sitepoint.com/practical-guide-angularjs-directives/>
18. Scanomat - <http://www.scanomat.com/int/topbrewer/introduction>
19. Sitepoint - <https://www.sitepoint.com/practical-guide-angularjs-directives/>
20. StackOverflow - <http://stackoverflow.com/research/developer-survey-2016>
21. wikipedia - www.wikipedia.org

22. Writing AngularJS Documentation –

<https://github.com/angular/angular.js/wiki/Writing-AngularJS-Documentation>

23. w3schools <http://www.w3schools.com/angular>

ПРИЛОЖЕНИЕ 1 – Финален вид на package.json

```
{
  "name": "cooking-book",
  "description": "angular cooking book",
  "version": "1.0.0",
  "author": "Eleonora Dimchevska",
  "privet": true,
  "dependencies": {
    "angular": "^1.5.2",
    "angular-animate": "^1.5.2",
    "angular-local-storage": "^0.2.7",
    "angular-ui-router": "^0.2.18"
  },
  "devDependencies": {
    "angular-mocks": "^1.5.3",
    "grunt": "^0.4.5",
    "grunt-contrib-concat": "^0.5.1",
    "grunt-contrib-jshint": "^1.0.0",
    "grunt-contrib-sass": "^0.9.2",
    "grunt-contrib-uglify": "^0.9.2",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-ngdocs": "^0.2.10",
    "jasmine-core": "^2.4.1",
    "karma": "^0.13.22",
    "karma-chrome-launcher": "^0.2.3",
    "karma-cli": "^0.1.2",
    "karma-coverage": "^0.5.5",
    "karma-firefox-launcher": "^0.1.7",
    "karma-ie-launcher": "^0.2.0",
    "karma-jasmine": "^0.3.8",
    "karma-phantomjs-launcher": "^1.0.0",
    "phantomjs-prebuilt": "^2.1.7"
  }
}
```


ПРИЛОЖЕНИЕ 2 – Финален вид на Gruntfile.js

```
module.exports = function(grunt) {

    // Project configuration.
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        concat: {
            options: {
                process: function(src, filepath) {
                    return '/*! Source: ' + filepath + ' */' + '\n' +
                        src.replace(/(^|\n)[ \t]*('use strict'|"use strict");?\s*/g, '$1');
                },
                sourceMap: true
            },
            dist: {
                src: [
                    //първо се събират файловете modules
                    'src/*.module.js', 'src/**/*.module.js',
                    //после routes
                    'src/*.routes.js', 'src/**/*.routes.js',
                    //след това всички останали
                    'src/*.js', 'src/**/*.js',
                    //външни ресурси
                    '!src/*.spec.js', '!src/**/*.spec.js'
                ],
                dest: 'build/js/<%= pkg.name %>.js'
            },
            thirdparty: {
                src: [
                    //външни ресурси
                    'node_modules/angular/angular.js',
                    'node_modules/angular-ui-router/release/angular-ui-router.js',
                    'node_modules/angular-local-storage/dist/angular-local-storage.js',
                    'node_modules/angular-animate/angular-animate.js'
                ],
                dest: 'build/js/thirdparty.js'
            },
            thirdpartyMin: {
                src: [
                    //външни ресурси
                    'node_modules/angular/angular.min.js',
                    'node_modules/angular-ui-router/release/angular-ui-router.min.js',
                    'node_modules/angular-local-storage/dist/angular-local-storage.js',
                    'node_modules/angular-animate/angular-animate.js'
                ],
                dest: 'build/js/thirdparty.min.js'
            }
        }
    });
```

```

    }
  },
  watch: {
    js: {
      files: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js'],
      tasks: ['jshint:beforeconcat', 'concat:dist', 'ngdocs']
    },
    scss: {
      files: ['scss/*.scss'],
      tasks: ['sass']
    }
  },
  uglify: {
    my_target: {
      options: {
        sourceMap: true,
        preserveComments: 'some'
      },
      files: {
        'build/js/<%= pkg.name %>.min.js': ['build/js/<%= pkg.name %>.js']
      }
    }
  },
  sass: {
    dist: {
      options: {
        style: 'compact',
        "sourcemap=none": '',
        noCache: true
      },
      files: {
        'build/css/app.css': 'scss/app.scss'
      }
    }
  },
  jshint: {
    options: {
      curly: true,
      eqeqeq: false,
      eqnull: true,
      browser: true,
      funcscope: false,
      unused: true
      // globals: {
      //   jQuery: true
      // }
    },
    beforeconcat: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js'],
    afterconcat: ['build/js/<%= pkg.name %>.js']
  }
}

```

```
    },
    ngdocs: {
      all: ['src/*.js', 'src/**/*.js', '!src/*.spec.js', '!src/**/*.spec.js']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-sass');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-ngdocs');

  grunt.registerTask('build', ['jshint:beforeconcat', 'concat', 'jshint:afterconcat', 'uglify',
                                'sass', 'ngdocs']);
  grunt.registerTask('dev', ['build', 'watch']);
};
```

ПРИЛОЖЕНИЕ 3 – Екран добавяне на рецепта от разработеното приложение

Книга с рецепти

Търси

Добави рецепта

Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Добави рецепта

Попълни име:

Попълни описание:

Добави продукти

Количество	Мерна единица	Продукт	Изтрий
<div>Добави</div>			

Запази

ПРИЛОЖЕНИЕ 4 - Екран редактиране на рецепта от разработеното приложение

Книга с рецепти

 Търси Добави рецепта Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Редактирай рецепта

Попълни име:

Попълни описание:

Добави продукти



Изтрий



Изтрий



Добави



Запази

ПРИЛОЖЕНИЕ 5 – Екран изтриване на рецепта от разработеното приложение

Секция едно – потребителя трябва да потвърди дали е сигурен, че иска да изтрие рецептата

Книга с рецепти

 Търси

 Добави рецепта

 Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Изтриване

Сигурен ли си че искаш да изтриеш рецептата: Хрупкаво пиле

Да

Не

Секция две – потвърдително съобщение, че рецептата е изтрита.

Книга с рецепти

 Търси

 Добави рецепта

 Налични рецепти

- Кекс
- Чийзкейк
- Мъфини

Изтриване

Рецептата е изтрита

ПРИЛОЖЕНИЕ 6 – Екран показване на грешка от разработеното приложение

Книга с рецепти



Търси



Добави рецепта



Налични рецепти


- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Внимание

Рецептата е вече изтрита или не съществува!

ПРИЛОЖЕНИЕ 7 – Екран преглед на рецепта от разработеното приложение

Книга с рецепти

 Търси Добави рецепта Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Хрупкаво пиле

Количество	Мерна единица	Продукт
1	чаша	корнфелкс
3/4	чаша	пармезам

Описание:

Пече се на фурна на 175 градуса.

 Редактирай Изтрий

ПРИЛОЖЕНИЕ 8 - Екран търсене на рецепта от разработеното приложение

Книга с рецепти

Търси

Добави рецепта

Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Търси рецепта по продукти

Изброй продукти разделяйки ги с ", ":

Търси

Избери от вече добавените продукти:

масло, захар, корнфлейкс, пармезам, яйца, заквасена сметана,

Книга с рецепти

Търси

Добави рецепта

Налични рецепти

- Кекс
- Хрупкаво пиле
- Чийзкейк
- Мъфини

Търси рецепта по продукти

Изброй продукти разделяйки ги с ", ":

Търси

Избери от вече добавените продукти:

масло, захар, корнфлейкс, пармезам, яйца, заквасена сметана,

Намерени рецепти

- Хрупкаво пиле

ПРИЛОЖЕНИЕ 9 – Финален вид на karma.conf.js

```
// Karma configuration
// Generated on Mon Mar 28 2016 17:50:19 GMT+0300 (FLE Daylight Time)

module.exports = function(config) {
  config.set({

    // base path that will be used to resolve all patterns (eg. files, exclude)
    basePath: '',

    // frameworks to use
    // available frameworks: https://npmjs.org/browse/keyword/karma-adapter
    frameworks: ['jasmine'],

    // list of files / patterns to load in the browser
    files: [
      'node_modules/angular/angular.js',
      'node_modules/angular-ui-router/release/angular-ui-router.js',
      'node_modules/angular-local-storage/dist/angular-local-storage.js',
      'node_modules/angular-animate/angular-animate.js',

      'node_modules/angular-mocks/angular-mocks.js',

      'src/**/*.module.js',
      'src/**/*.js',
      'views/**/*.html',
      'views/*.html'
    ],

    // list of files to exclude
    exclude: [
    ],

    // preprocess matching files before serving them to the browser
    // available preprocessors: https://npmjs.org/browse/keyword/karma-preprocessor
    preprocessors: {
      'src/**/*.js': ['coverage']
    },

    // test results reporter to use
    // possible values: 'dots', 'progress'
    // available reporters: https://npmjs.org/browse/keyword/karma-reporter
```

```
reporters: ['progress', 'coverage'],

coverageReporter: {
  // default to build dir for output
  dir: 'coverage',
  reporters: [
    //output html report in build dir
    {type: 'html', subdir: 'html'},
    //output text summary to stdout (no filename specified)
    {type: 'text-summary', subdir: '.'}
  ]
},

// web server port
port: 9876,

// enable / disable colors in the output (reporters and logs)
colors: true,

// level of logging
// possible values: config.LOG_DISABLE || config.LOG_ERROR || config.LOG_WARN ||
// config.LOG_INFO || config.LOG_DEBUG
logLevel: config.LOG_INFO,

// enable / disable watching file and executing tests whenever any file changes
autoWatch: false,

// start these browsers
// available browser launchers: https://npmjs.org/browse/keyword/karma-launcher
// browsers: ['Chrome', 'Firefox', 'IE'],
browsers: ['PhantomJS'],

// Continuous Integration mode
// if true, Karma captures browsers, runs the tests and exits
singleRun: true,

// Concurrency level
// how many browser should be started simultaneous
//concurrency: Infinity
})
}
```

ПРИЛОЖЕНИЕ 10 – Екран от генерираната документация от разработеното приложение

cooking-book
API Documentation

search the docs

API Documentation / cookingBook.singleView / cbSingleViewService

cookingBook module

controller

CookingBookController

cookingBook... module

controller

RecipeController

service

cbRecipeService

cookingBook... module

controller

SearchController

service

cbSearchService

cookingBook... module

controller

SingleViewController

service

cbSingleViewService

cbSingleViewService

service in module `cookingBook.singleView`

Description

Управление на зареждането на данните за конкретна рецепта

Methods

findRecipe(recipeID, allRecipes)

Намиране конкретна рецепта

Parameters

Param	Type	Details
recipeID	String	о уникален номер на конкретна рецепта
allRecipes	Array	о всички налични рецепти

Returns

Object о конкретна рецепта

Properties

foundRecipe

Обект за стойности на конкретна рецепта. Първоначална стойност null.

Списък на използваните съкращения

Apps - applications
API - Application programming interface
BDD - behavior-driven development
CSS - Cascading Style Sheets
DI - Dependency Injection
DOM - Document Object Model
HTML - Hyper Text Markup Language
IDE - integrated development environment
MVC - Model–view–controller
MVP - Model-View-Presenter
MVVM - Model-View-View-Model
NPM - node package manager
SASS - Syntactically Awesome Stylesheets
SPA - Single-page application
TDD - Test-driven development