

**ТЕХНОЛОГИЧЕСКО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**  
**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

Тема: Интернет сайт за автоматизиране на поръчки за изработването  
на лазерно рязани стенсили

Дипломант:

*Иво Стратев*

Научен ръководител:

*маг. инж. Кирил Митов*

СОФИЯ

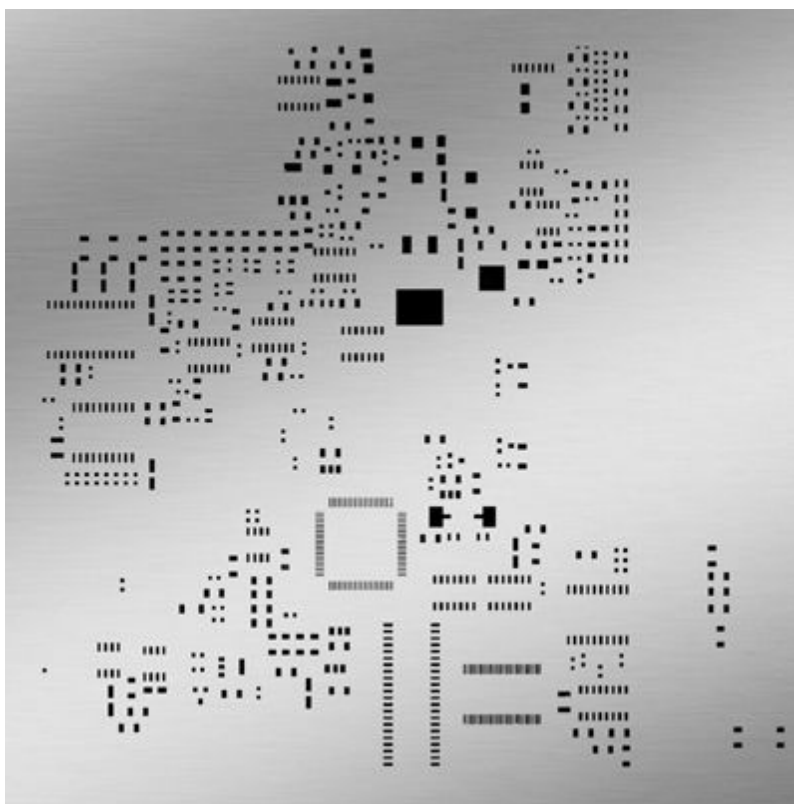
2016





## Увод

Стенсилите представляват метален лист с отвори (Фиг. 0.1), известни като падове или апартюри. Стенсилите се използват за изработка на печатни платки. Апертурите са местата, на които трябва да остане содер паста при изработка на печатните платки. За целта металният лист се поставя върху празна (без електрически компоненти) печатна платка и върху него се нанася тънък слой содер паста. Съществуват две технологии за изработка на стенсили: чрез лазерно рязане и чрез ецване. Лазерно рязаните са със значително по-високо качество, тоест апертурите им са значително по-прецизни, но и цената им е по-висока. Съществуват и два под типа на лазерно рязаните стенсили: “adhesive” и “smd”. Инструкциите, които оказват как да бъдат изрязани апертурите на един стенсил се съхраняват в Gerber файлове.



*(Фиг. 0.1) Примерен стенсил*

Gerber файловете са текстови файлове, използващи “ASCII” символи, които описват бинарни (двухцветни) векторни изображения. Форматът се е наложил като де факто стандарт при изработка на печатни платки и води своето начало от фото плотерите. Разработен е от фирма “Ucamso” [1]. Съществуват два стандарта на формата, като и двата са на фирмата Ucamso. По-старият формат е почти изцяло отпаднал, а за новия стандарт постоянно излизат ревизирани версии [2].

Gerber файловете разделят печатната платка на отделни слоеве (“layers”), които също могат да бъдат разделени на подслоеви. За центрирането и правилната подредба на отделните слоеве се използват центриращи маркери (“fiducials”). При изработката на стенсили се използват само три от всичките слоеве: “top paste” (горна паста), “bottom paste” (долна паста) и “outline” (очертание). За да бъде изрязан един стенсил са нужни поне два от тези три слоя, като задължителен е слоя очертание, който носи информация за формата на печатната платка. Върху печатната платка могат да се поставят електронни компоненти само от едната страна или и от двете ѝ страни. Това се избира от проектанта на електронното изделие. Ако компонентите са разположени и от двете страни са необходими два отделни стенсила за изработката на печатната платка.

За изработката на лазерно рязани стенсили освен Gerber файлове е нужна и конфигурация, описваща характеристиките на желанния стенсил. Някои от характеристиките са: дебелина на металния лист, от който стенсила бива изрязан, размер на листа, разположение на текста върху стенсила.

Целта на разработваната дипломна работа е пълната автоматизация на процеса по поръчване на лазерно рязани стенсили на фирма “IvasTech” чрез Интернет приложение. То трябва да предоставя визуализация на поръчката, механизъм за нейното следене от два типа потребители, статистики за броя направени поръчки, и посещенията на приложението.

## **ПЪРВА ГЛАВА**

### **Преглед на съществуващи конкурентни продукти и запознаване с бизнес модела на фирма “IvasTech”**

#### **1.1. Преглед на съществуващи конкурентни продукти**

Преди разработката на приложението бяха разгледани няколко конкурентни приложения и бе стигнато до заключение, че всички си приличат в решаването на общите проблеми.

##### **1.1.1 Преглед на приложението на фирма “WURTH ELEKTRONIK” (WEdirekt)**

Използваното от фирма “WURTH ELEKTRONIK” (WEdirekt) Интернет приложение [3] предоставя възможност за избор между adhesive, smd и eco (минималистичен) видове стенсили. Дава възможност за избор на следните опции при създаването на конфигурация:

- начин за пренасяне: правоъгелен (без допълнителна защита), в рамка и собствен вид защита за пренос;
- брой и вид на центращите маркери;
- разположение на стенсила върху металния лист;
- покритие;
- позиция на текста

Промяната на повечето (тези, които имат визуализация) опции променя и изгледа на поръчката. За тяхното запазване е нужно след промяната на всяка опция тя да бъде записвана, чрез натискането на бутона до нея (Фиг. 1.1.1.).

След създаването или избирането на конфигурация, потребителят преминава към попълване на информация за стенсила, където вижда и цената, на базата на избраната конфигурация. Тук той може да попълни информацията относно: описание, номер на поръчката (в системата на поръчващия), размери на стенсила, брой апертюри, текст на поръчката, коментар към нея, както и информацията относно доставката на поръчката. Ако няма промяна на опция, не се променя и изгледът на поръчката. Потребителят трябва сам да въведе броя на апертюрите, което може да доведе до грешно попълнена информация, защото не всеки софтуер за работа с Gerber файлове предоставя възможност за автоматично пресмятане на броя на апертюрите. Отново за да бъде отразена промяната е нужно натискането на бутона до нея. След като всички детайли за поръчката са попълнени, потребителят може да качи архивен файл, който да съдържа нужните файлове за изработка на поръчката.

The screenshot shows the 'Basic configuration' step of the Würth Elektronik Stencil Shop. The 'SMD stencil' option is selected. The configuration parameters are as follows:

Step	Parameter	Value	Action
1	Form (frame/self-tensioning frame system)	rectangular	edit
2	Fiducials	2 (cutting through)	edit
3	Layout position	Layout centered	edit
4	Surface	Standard	edit
5	Legend position	PCB side, 6, a	save
6	Save basic configuration as	—	edit

A red button at the bottom says 'continue and enter PRODUCTION DATA'.

(Фиг.

1.1.1.) Създаване на конфигурация за стенсил чрез интернет приложението на фирмата WEdirekt

### 1.1.2 Преглед на приложението на фирма “multi-cb”

Интернет приложението на фирма “multi-cb” използва дълга колона (Фиг. 1.1.2.1), разположена в лявата част на приложението, за конфигурацията на цялата поръчка [4]. Дължината на колоната се променя, когато потребителя реши да избере допълнителни възможни опции. От дясната страна на приложението, винаги се намира поле с изчислената цена, което променя своето разположение, така че постоянно да бъде видимо от потребителя. Приложението предоставя възможност за оказване на следните опции:

- слоеи, за който ще е необходимо изработването на стенсил;
- размери на желаня стенсил;
- разположение върху металния лист;
- покритие на стенсила;
- начин на съхранение при превоз;
- брой на апертурите;
- вид на центриращи марки;
- разположение на текст (като позицията на изглед е фиксирана на “Squeegee side”);
- възможност за избор на редуциране на броя на апертурите;
- опции относно доставката на поръчката.

Приложението не позволява записването на характеристиките на желаня стенсил в конфигурация, която да бъде преизползвана при следваща поръчка. То не предоставя никаква визуализация на поръчката. За два отделни избора предоставя статични картинки за улеснение на потребителя - избор на позиция на текст и разположение на стенсила върху металния лист. Дава възможност за качване на архивен файл, който да съдържа слоевете необходими за изработката на поръчвания стенсил и отново изисква потребителя сам да въведе броя на апертурите.



**multi-cb**  
LEITERPLATTEN

25 Jahre  
SSL

My Account Login

PCB SMD-Stencil Flex Rigid-Flex Metal core

Reset Calculate

**SMD-Stencil**

Name: Test

Working days: 4WD Standard 2WD

Estimated dispatch: Wed 17.02.2016

Size: 300 X 200 mm

Quantity: 1 Pieces

SMD-Stencil for: TOP+BOT (on single Stencil)

Type: quick clamping frames

Type (System): QuattroFlex

Type: asdad

Number of SMD pads: sadaad pads

Pad reduction: Yes 0 %

Thickness: 120µm

Existing fiducials: none

Text (Squeeze side): text

Text type: half lasered

Text position: choose

Position / readable from: 3 (upper right) b (from right)

Diagram: 1 2 3  
d 4 5 b  
6 7 8  
c

**Prices**

Price for 1 units	Unit price	Total
first/reorder in 4 WD	46.79 €	46.79 €

Add to basket

Price valid only when using the online portal plus VAT and shipping costs

Save quotation

(Фиг. 1.1.2.1) Изглед на приложението на фирма *multy-cb*, който служи за поръчване на стенисили

### 1.1.3 Преглед на приложението на фирма LeitOn

Интернет приложението на фирмата LeitOn предоставя изключително богат избор от възможности за съхранение при превоз на доставката [5]. Освен тях предоставя и следните възможности за избор:

- размер на поръчвания стенси́л;
- брой на апертурите;
- разположение върху металния лист;
- покритие на стенси́ла;

- възможности за оптимизация на апертюрите (заобляне на ъглите им и намаляване на размера им);
- брой на центриращи марки (фиксиран между 2 и 8 включително);
- вид на центриращите марки;
- позициониране на текст (като не се предоставят възможности относно изгледа).

Приложението не предоставя никаква визуализация на поръчката и възможност за преизползване на конфигурация. След избирането на опциите за желания стенсил потребителя вижда изчислената цена и следва да попълни информация относно доставката и да влезе в своя потребителски профил, за да може да качи желаните файлове. Има възможност за качване на единичен файл, което предполага той да е архивен и да съдържа необходимите за изработката файлове (Фиг. 1.1.3.1). Освен качване на файл потребителя може да въведе номер на поръчката за използваната от него вътрешна система, както и коментар към поръчката. Над полето за качване на файл са изброени файловите формати, които приложението може директно да обработи.

## **1.2 Проучване и запознаване с бизнес модела на фирма “IvasTech”**

При бизнес модела използван от фирма “IvasTech”, преди разработката на текущата дипломна работа, се използва PDF форма (Фиг 1.2.1) за желаните характеристики на поръчвания стенсил/и. Клиента я попълва и прикача към нея архивен файл, съдържащ всички необходими Gerber файлове за направата на поръчка. Всяка поръчка се обработва от служител на фирмата, като се пресмята нейната ценна. Базовата цена се определя от размера на поръчвания стенил и броя на апертюрите, техния брой се пресмята автоматично от софтуера, който фирмата използва за изработка на лазерно рязани стенсили.

Deutsch / English

You have gone full screen.

Exit full screen (F11)

LEITON

ON BEST SERVICE

4.9 out of 5 with 257 ratings

Company ▾

Prices ▾

Technology ▾

Service & Information ▾

FAQ & Tools ▾

My Account ▾

1 CALCULATE PRICE >

2 SELECT SHIPPING >

3 ORDER SUMMARY >

4 LOGIN SIGN UP >

5 UPLOAD & YOUR NOTES

FILE FORMATS

logged in as: adaw adwd | Log out

Now we need the files for the different positions!

We can process the following data formats directly:

GERBER RS-274X

Extended Gerber Universalformat (RS-274X)

Eagle

Target3001

Sprint Layout

Sprint

KiCad

KICAD 3

Stable 2013.07.07-BZR4022

FILE UPLOAD

1) test

1 x SMD Stencil

Production within 2 Working Days, plus shipment time

Details

Type:

SMD Stencil

Type:

quick fastener system

Type:

Quattroflex / Stencilman

Text:

text

Size:

479 x 497 mm

Stencil Thickness:

100 µm

Number of Pads:

1500

Edge protection:

no

Surface finishing:

brush both sides

Layout position:

View on coating knife side, centered orientation to PCB outline, page-long orientation to long PCB side rounded: no, reduced: no

Pad optimization:

Fiducials:

3

Fiducials:

surface lasered

Text position:

top / left

Packaging:

standard packaging

File:

The file bottom.pho 22.51 KB has been successfully uploaded.

Delete

Your internal order number

Your internal order number

Optional Comments

Optional Comments

For example, additional information about this order..

(Фиг. 1.1.3.1) Качване на Gerber файлове чрез приложението на фирма Leit-On

10

## Solder Paste Stencil Order Form

Company name :  Tel number :   
 Delivery address :  Fax number :   
 Contact name :  E-mail address :

### Stencil Details

File name :  *(please ensure all data and this form are zipped together before sending)*

Quantity :  Stencil thickness (µm) :  Fiducial marks :   
 Data view on squeegee side ? ☐ Fiducial marks side :

Please provide any aperture modifications and image positioning details below:  
 (If you would prefer us to recommend modifications, please indicate this)

**Inscription** Squeegee side ☐ Text :   
 PCB side ☐ Text :

### Multiple panel

Number of panels In X  In Y  Step(mm) Step in X  Step in Y

**Positioning**  Adjust Layout to stencil :  
☐ ☐  


### Delivery Details

Desired delivery date:

(Фиг. 1.2.1) PDF форма използвана от фирма IvasTech

## ВТОРА ГЛАВА

### Преглед на избраните технологии и аргументация за техния избор

Наборът от технологии за реализацията на Интернет приложение започва с избора на сървърна технология. Съществуват много и различни технологии, сред които са: “Rails” [6] (“Ruby” [7]), “Django” [8] (“Python” [9]), “Laravel” [10] (“PHP” [11]), “Kore” [12] (“C/C++” [13]), “Node.js” [14] (“JavaScript” [15]) и много други.

#### 2.1 Избор на сървърната технология "Node.js"

За разработка на настоящата дипломна работа бе избрана мултиплатформената среда за разработка на сървърни и мрежови приложения Node.js (Node), защото имам известен опит с използването на технологията [16, 17, 18], в комбинация с други две: “express.js” [19] (express) и “MongoDB” [20] (Mongo) и реших, че разработката на дипломната работа е добра възможност да ги науча в дълбочина и детайлност.

Node представлява среда за изпълнение на JavaScript код извън браузъра с добавени библиотеки за входно-изходни операции. Той използва V8 двигател за изпълнението на JavaScript код, което позволява към езика да бъде добавена почти всяка C/C++ библиотека. За целта е нужно да се напише C++ addon (приставка), който да “каже” на Node как да преобразува C/C++ кода към JavaScript код, така че той да бъде използван сред останалия JavaScript код. Приставките, за разлика от останалия JavaScript код, не се интерпретират по време на изпълнението на кода, който ги използва, а директно се използва C/C++ кода им. Пример за C++ приставки са класовете Stream [21] и Buffer [22], които са част от езика.

За обмяна на код между проекти, работещи с Node, се използва официалният й пакетен мениджър - npm [23]. За оказването на пакетите и техните верси, които едно Node приложение ползва, е необходим package.json [24] файл, който служи и за

описание на разработваното приложение. Описанието може да бъде използвано, ако разработваният проект се сподели с други програмисти, независимо дали това става на официалния сайт на системата npm или на затворен (частен) нейн вариант. Файлът `package.json` позволява дефинирането на команди за стартиране на приложението и пускане на тестове, ако има такива. В него може да се окаже и версията/ите на Node, с които приложението може да работи. Освен това той се използва и от технологията “browserify” [25].

Технологията browserify е налична под формата на npm пакет и целта ѝ е да позволи преизползването на npm пакети, налични за използване в средата Node извън нея - в браузърите. Повечето модули, които изпълняват входно-изходни операции, имат свои браузърни варианти. Това става посредством идентичния програмен интерфейс: “require” и “module.exports (exports)”. Освен това тази технология позволява използването на споменатия интерфейс и сред всички JavaScript файлове, които са предназначени за изпълнение в браузъра, и тяхното обединение в един единствен файл, който да бъде изпратен до браузъра, което значително намалява времето за зареждане на едно Интернет приложение.

## **2.2 Избор на фреймуърк към Node.js - "express.js"**

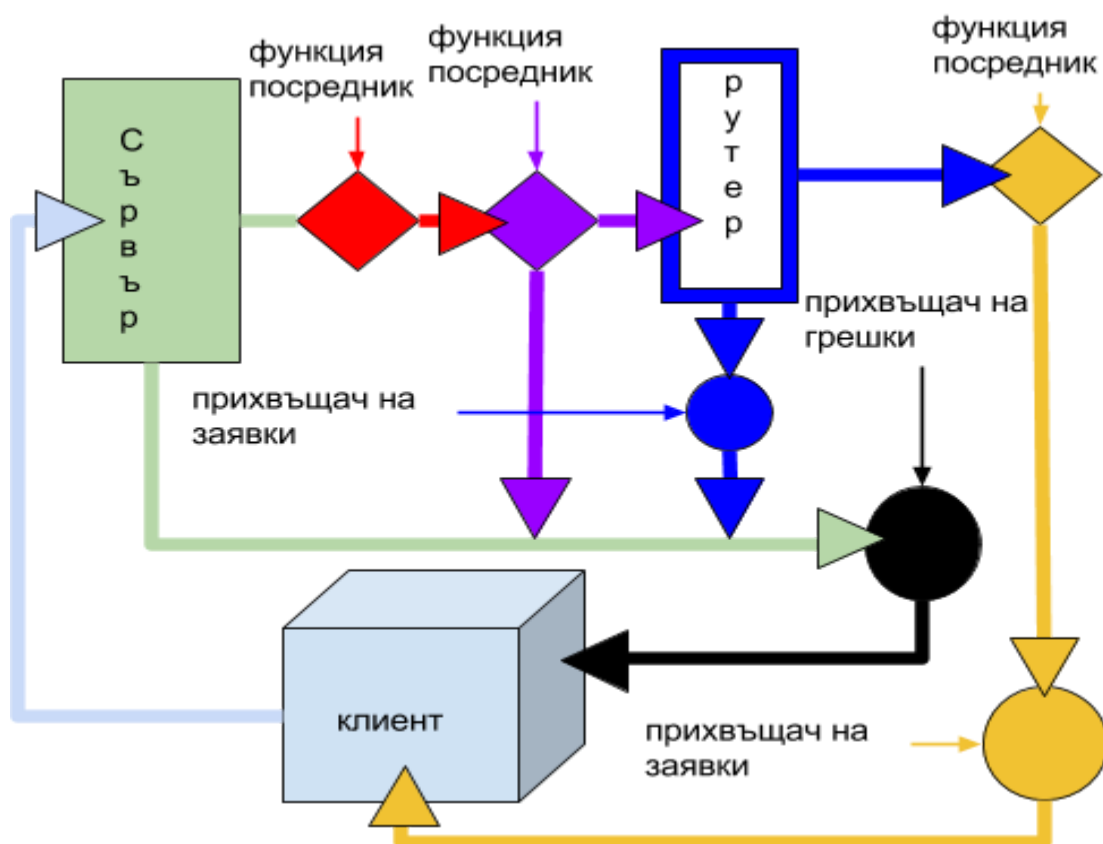
Към Node могат да бъдат добавени много уеб фреймуарци [26], които да олеснят писането на Интернет приложения. От всички бе избран express, с който съм работил. Той от своя страна също може да бъде разширяван с така наречените функции “посредници”.

Express е фреймуърк, който дава възможност да се изгради “система от тръби” през, която преминават http заявки. Това става благодарение на “рутерите”, които фреймуърка предоставя. Те представляват обекти, чрез които се описват всички възможни заявки, като към тях се прикачат “middleware” (посреднически) функции и “request handler” (прихващач на заявка), чрез извикването на методите им. Прихващачите на заявки и функциите посредници имат една и съща абривиатура `function (req, res, [next])`, но прихващачите на заявки се различават по липсващ последен аргумент или липса на неговото извикване. Последния аргумент е и

механизмът, който позволява на заявките да продължат към следващата тръба. Той може да бъде извикан с или без аргумент. Ако е извикан без аргумент това значи, че при извикването на текущата функция, която посредничи, не е възникнала грешка и заявката продължава към следващата тръба. Ако е извикана с аргумент, то той представлява грешката, която е възникнала, а заявката продължава към следващия “error handler” (прихващач на грешки). Прихващачът на грешки е подвид на прихващача на заявки. Той има следната абривиатура `function (err, req, res, [next])`, където `err` е аргументът, с който предходната функция е извикала функцията `next`.

Накратко всяка функция, която се подава по някакъв начин към `express` се явява посредница, а всяка функция, която не извиква механизма `next` - прихващач. Ролята на посредниците е да подготвят заявката, за преминаването ѝ към някой прихващач, който от своя страна е длъжен да спре заявката от нейното продължение, като изпрати отговор до машината направила тази заявка (Фиг 2.2.1).

Пример за посредническа функция е методът `json`, който е част от обекта, който ‘`body-parser`’ експортира, чиято роля е да превърне тялото на заявката в JavaScript обект, който да се използва като източник на информация. Най-добрият пример за прихващач е прихващача на грешки, неговата задача е да съобщи за настъпила грешка. Например когато в резултат на някоя заявка към базата данни се получи грешка, съответно подходящо известие би било изпращането на статус-код 500. Или ако клиента се опитва да достъпи страница на приложението, която не съществува, тогава е подходящо, като резултат на заявката да се изпрати страница, която съобщава за тази грешка. Тук е важно да се спомене, че за `express` е важен редът, в който се подават функциите, отговарящи за заявките, тоест важно е прихващачите на грешки да се подадат последни, за да бъдат извикани когато възникне грешка.



(Фиг. 2.2.1) примерна система от траби за http заявки на едно express приложение

## 2.3 Избор на базата данни "MongoDB"




За база данни бе избрана базата данни MongoDB (Mongo), с която съм работил и познавам. Тя се съчетава изключително добре със средата за разработка на сървърни и мрежови приложения. Mongo е мултиплатформена NoSQL база данни, която се различава от традиционите, релационни бази данни, които съхраняват своята информация в таблици. При нея се използват колекции от документи - всеки документ отговаря на един запис в дадената колекция. Документите, се съхраняват във формат подобен на текстовия JSON, наречен BSON [27]. BSON идва от "бинарен JSON", което е и основната разлика между двата формата, освен това BSON в някой случай съхранява и допълнителна информация. Допълнителна информация се съхранява например, когато базата от данни прецени, че дължината на някой символен низ е прекалено голяма, тогава към информацията от низа се добавя и неговата дължина.



Друга характерна особеност е, че масивите от данни се запаметяват под формата на бинарно репрезентиране на JSON обект, който като свой ключови полета има стойности на индексите на масива в нарастващ ред (ключовите полета на всеки BSON обект са подредени в нарастващ ред). Пример за BSON репрезентирането на JSON обекти могат да бъдат намерени в раздела “FAQ” на официалната страница на спецификацията на формата - [bsonspec.org](http://bsonspec.org).

Mongo със своята динамична структура е решение на един от основните проблеми на релационните SQL бази от данни - празните полета в таблиците, които те използват за съхранение на данни. Така например вместо да се създават 3 таблици за автор, жанр и заглавие на книга, те могат да бъдат обединени в един единствен документ, при който всяка книга пази характеристиките си. Mongo е и най-използваната NoSQL база от данни и към момента на описването на текущата дипломна работа е и 4-та най-използвана база данни според сайта [db-engines.com](http://db-engines.com) .(Фиг. 2.3.1).

296 systems in ranking, February 2016

Rank			DBMS	Database Model	Score		
Feb 2016	Jan 2016	Feb 2015			Feb 2016	Jan 2016	Feb 2015
1.	1.	1.	Oracle	Relational DBMS	1476.14	-19.94	+36.42
2.	2.	2.	MySQL	Relational DBMS	1321.13	+21.87	+48.67
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1150.23	+6.16	-27.26
4.	4.	4.	MongoDB 	Document store	305.60	-0.43	+38.36
5.	5.	5.	PostgreSQL	Relational DBMS	288.66	+6.26	+26.32
6.	6.	6.	DB2	Relational DBMS	194.48	-1.89	-7.94
7.	7.	7.	Microsoft Access	Relational DBMS	133.08	-0.96	-7.47
8.	8.	8.	Cassandra 	Wide column store	131.76	+0.81	+24.68
9.	9.	9.	SQLite	Relational DBMS	106.78	+3.04	+7.22
10.	10.	10.	Redis 	Key-value store	102.07	+0.92	+2.86

(Фиг. 2.3.1) Подредба на базите данни по използваемост, според DB-Engies за февруари и януари 2016г.

### 2.3.1 Избор на драйвера "Mongoose" за връзка с базата данни

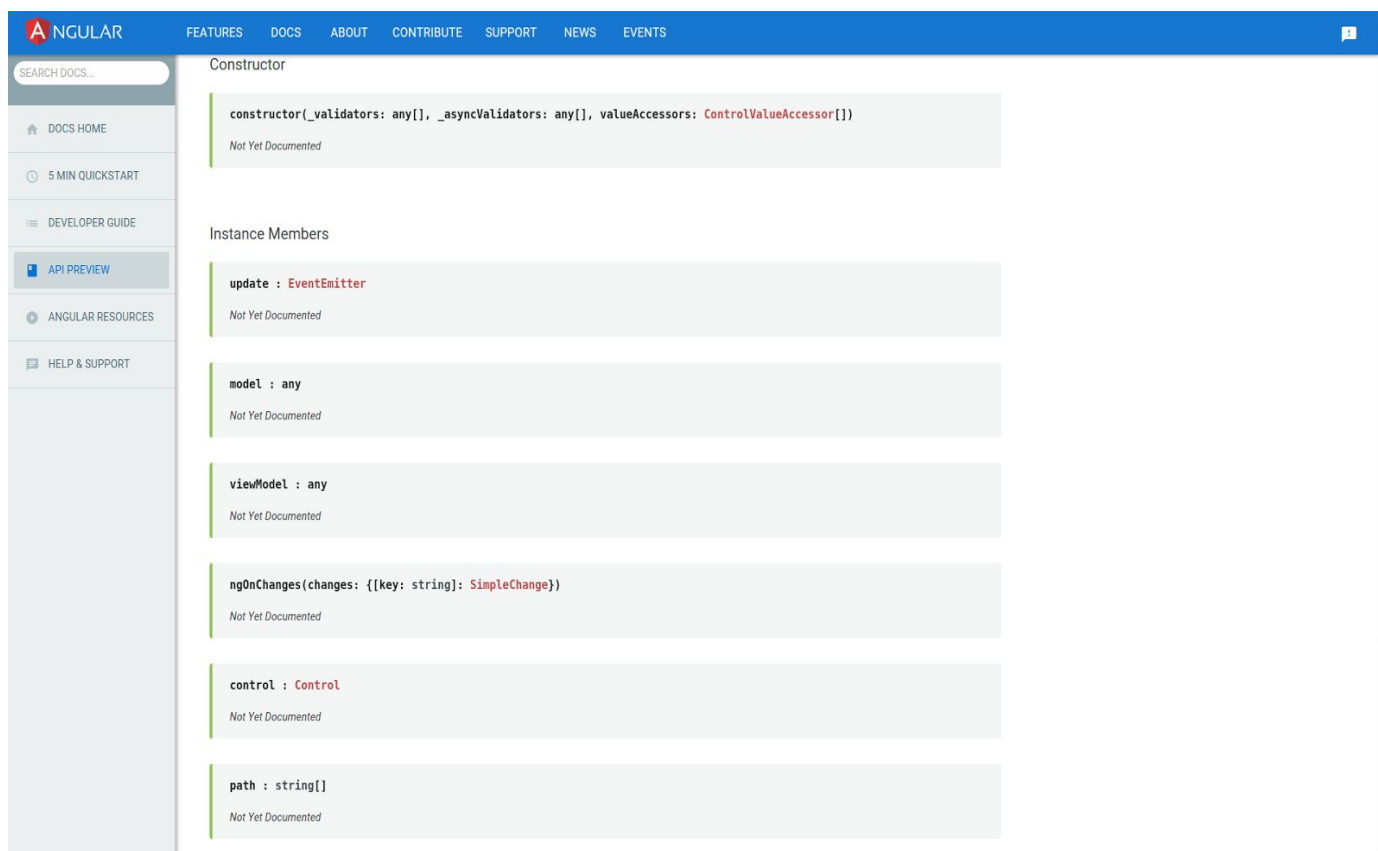
За връзка с базата данни бе избран npm пакета "Mongoose" [28], който представлява "Object-document mapping" (ODM), който е еквалента на използваните при релационните бази данни технологии "Object-relational mapping" (ORM). Mongoose е ориентиран изцяло за работа с базата данни Mongo, за разлика от повечето ORM-та, които предоставят вградена поддръжка на няколко вида релационни бази данни.

ORM е технология, която позволява работата с база данни (заявки, описание на структурата на използваните таблици и връзките между тях) да се извършва, чрез обекти характерни за дадения език за програмиране и по този начин създава улеснение и пълна абстракция над избраната база данни, като всеки момент тя може да бъде заменена от друга, без да се налага значителна промяна в кода.

Mongoose предоставя възможност формата на документите (колекциите) да бъде описана и дефинирана чрез JavaScript обект - схема. Създадената схема може да бъде превърната в модел, който представлява връзката с базата данни. Всеки модел отговаря на една колекция в базата данни и може да бъде директно използван за връзка с нея, като поддържа всички методи на официално предоставяния драйвер. Другият начин, по който моделът може да бъде използван, е като конструктор на обекти. Mongoose е единственият драйвер за Mongo, който разполага с механизъм симулиращ свързвания (оператора join в езика SQL). Този механизъм се нарича популация и позволява директната замяна на ключови полета на документ от дадена колекция с такива от документи от друга колекция. За оказването на тази връзка се използва ключовото поле "ref" в описанието на схемите. Избраният драйвер има вградена поддръжка за Обещания (при заявките трябва да бъде извикан метода "exec" без аргументи, във всички останали случаи Mongoose връща обещание по подразбиране) и е единственият mongo драйвер, използващ Обещания и позволяващ тяхната имплементация да бъде заменена с избрана от разработчика такава (по подразбиране в новите верси се използва "Promise/A+" имплементация, а в по-старите "mpromise"), като за целта може да се използва дори вградената в новите верси на Node имплементация.

## 2.4 Избор на JavaScript фреймуърка "Angular.js"

Към вече избраните технологии бе добавен “Model - View - Controller” (MVC) фреймуърка “Angular.JS” [29, 30] (Angular), които ги допълва до така нареченият “MEAN” стек. Причината за това решение е концепцията, която фреймуъркът предоставя и шанса, чрез разработката на настоящата дипломната работа да го изпробвам и науча. Към момента на този избор голяма популярност бе добила и втората версия на фреймуърка - “Angular” [31] (Angular 2), но той бе отхвърлен като вариант, понеже тогава се намираше в пре-алфа състояние и липсваше официално предоставената документация на повечето части от него. Към момента на описването на процеса по разработката на текущата дипломна работа Angular 2 се намира в състояние на бета версия и документация на повечето елементи продължава да липсва (Фиг. 2.4.1).



(Фиг. 2.4.1) Липсата на документация на една от основните директиви - *ngModel*

Angular е MV\* фреймуърк и е ориентиран изцяло към клиентската част на едно приложение. Той свежда традиционния архитектурен модел MVC, до MV\* , което според екипа разработил технологията се чете като “Model - View - Whatever”.

Традиционният MVC модел разделя едно Интернет приложение на 3 основни части:

- модел - това са данните, които приложението ползва (най-често идват от сървъра на приложението);
- изглед - представлява информацията, която се визуализира на екрана (това, което потребителя вижда като интерфейс на приложението);
- контролер - софтуерният код, който осъществява връзката между данните, които се пазят на сървъра (Модела) и това кои от тях да се визуализират на екрана (Изгледа).

Той също така използва еднопосочно сливане на данните, тоест слива данните (модела) и шаблона (изгледа) в едно. Този модел топично се реализира по следния начин: всеки един изглед представлява шаблон, който съдържа полета. Тези полета са места, които моделът трябва да запълни. Всеки контролер представлява един клас или функция която по подадени данни запълва предназначените за целта полета в шаблона с предоставените от модела дани. В резултат се получава изглед, който се изпраща до потребителя. Основният проблем на този архитектурен модел е, че след настъпване на сливането, промени в модела или съответните му раздели НЕ се отразяват в изгледа. Още по-лошо, всички промени, които потребителят прави на изгледа, не се отразяват в модела. Това означава, че разработчикът на приложението сам трябва да напише код, който постоянно да синхронизира гледката с модела и модела с изгледа или да използва външна библиотека, ако съществува такава.

Архитектурния модел, който Angular използва, решава основния проблем на традиционния MVC модел, като предоставя двупосочно сливане на данните - променяйки модела, това се отразява на изгледа и обратното, промяната в изгледа води до промяна на модела. Тоест може да се приеме, че изгледа винаги е проекция на модела, която потребителя на приложението вижда.

Angular реализира своя MV\* модел, чрез един от компонентите, който предоставя - “директиви”. Директивите от една страна представляват HTML маркери,

които оказват на компилатора, който фреймуърка ползва, как да променя изгледа и те могат да бъдат под формата на: HTML таг, HTML атрибут, HTML коментар, както и CSS клас, като за предпочитане е да се използват първите две. От друга страна всяка една директива представлява функция, която дава точни инструкции към вградените във фреймуърка компилатор. Свързването на името на една директива (HTML маркера, който отговаря на нея) и функцията, която оказва инструкциите на компилатора става посредством API (програмен интерфейс): `module.directive(name, directive)`. Към името на директивата има едно единствено изискване - да следва JavaScript приетата конвенция за именуване, тоест тя да следва така наречения "camelCase" [32]. Това условие е необходимо да бъде спазено, защото Angular използва механизъм за разграничаването на директиви от останалия HTML код (за който няма разлика от малки и големи букви). Фреймуърка използва следните разделители "-", "\_" и ":", за да разпознае, кой HTML маркер, за коя директива се отнася. Ако използваме за име на директива "someName", както е посочено горе, то в HTML кода ще имаме възможност да използваме `some-name`, `some_name` и `some:name` и Angular ще знае, че това е директива, както и нейните инструкции. Добра практика е да се използва първия разделител и имената на директивите да не започват с `ng`, което се използва от фреймуърка за вградените в него директиви. Някой от най-често използваните от тях са `ngModel`, `ngClick`, `ngClass`, `ngStyle` и други. Съществува едно единствено изключение на това правило и то е "interpolation" (вмъкваща) директивата. Тя се използва за директното оказване на места за запълване от модели. Тази специална директива няма име, но за нейното откриване се използват специални символи, които оказват началото и края на директивата, а израза между тях използва име на модел, който да замени своето име със стойността си. Тези специални символи са предефинирани, като "{{" за начало и "}}" за край, но могат да бъдат променяни от разработчика, ако той иска да използва език, различен от HTML, за описание на шаблоните на изгледите.

Директивите са един от механизмите за преизползване на код в едно Angular приложение, както и от едно приложение в друго, използват се и за "разширение" на описателния език HTML, използван за изграждане на шаблони, които фреймуърка ползва. Възможните инструкции, които една директива може да отправи към компилатора са следните:

- “restriction” (ограничение) - оказва кой/и от възможните HTML маркери е/са валидни за директивата;
- “priority/terminal” (приоритет) - оказват реда, в който се компилират отделните директиви;
- “template/Url” (изглед) - шаблонът, който ще се използва като визуализация на директивата (той може да е както символен низ съдържащ HTML фрагмент, така и интернет адрес, на който такъв фрагмент може да бъде намерен и за неговото извличане е нужно да бъде направена заявка към сървъра, за която фреймуърка е отговорен да направи);
- “link” (свързваща функция) - функция, която свързва текущия контекст (данните от него) и изгледа, която служи за синхронизация между модела и изгледа.;
- “transclude” (заменяща функция) - функция, която свързва изгледа с контекст различен от текущия;
- “compile” (компилираща функция) - функция, която манипулира и променя Document Object Model (DOM) създаден от HTML фрагмента (изгледа на директивата);
- “controller” (контролер) - функция, която придава специфично поведение на директивата или символен низ, който оказва на Angular функцията която да използва за контролер;
- “controllerAs” (име на контролера) - име под, което контролера ще е наличен в текущия контекст, ако такова липсва контролера ще бъде прикачен директно на текущия контекст (добра практика е да се ползва такова);
- “scope” (контекст) - когато стойността е вярна булева стойност се създава нов “детски” контекст, който наследява своя “родител”, когато е обект - той бива използван за създаването на изолиран контекст, на който комбинацията от ключове и техните стойности оказва кои входни данни да ползва директивата и как да ги интерпретира (възможностите те са: символен низ, израз и двупосочна връзка), във всички останали случаи не се създава контекст, а вместо това се използва родилския;
- “bindToController” (свързвания към контролера) - това е обект, който свързва входните данни директно с контролера като вместо към контекста (както прави контекст инструкцията) ги прикача към името под, което контролера е бил

прикачен към текущия контекст (ако се дефинира такъв обект като инструкция, то име на контролер също трябва да бъде дефинирано;

- “require” (изисквания) - необходими са , когато директивата трябва да използва механизми дефинирани на други директиви, които се намират на същия елемент или на елементи над него (могат да бъдат използвани в свързващата функция и са достъпни като нейн четвърти аргумент);
- “name” (име) - това е инструкция, позволяваща контролера на директивата да бъде динамично свързан с нея, най-често чрез стойност на атрибут, интерпретирана като символен низ (тази инструкция не е документирана на официалния сайт на фреймуърка [33], но е изключително полезна, например когато се използва наследяване на контролери, по този начин се придава различно поведение на един и същ изглед);

За да работи едно Angular приложение е необходимо то първо да премине през “Bootstrap” процес (зареждане). Този процес е отговорен за първоначалното свързване на елементите на приложението - първото компилиране и свързване на модели и изгледи, както и зареждане на модули, които приложението ползва.

Модулите представляват основния механизъм за разделяне на Angular код на части, както и преизползването му от един проект в друг. Това преизползване става като всеки модул може да декларира своите зависимости от други модули. Модулите предоставят единен интерфейс за регистрация на отделните елементи, който е: `module[elementType] (Name |String|: “name”, Element |Any|: element)`

Отделните елементи, които фреймуърка ползва са:

- “provider” (доставчик) - функция, която се грижи за създаването на нови услуги, което става посредством конфигурационни настройки;
- “factory” (фабрика) - функция, която дефинира нова услуга и винаги връща едно и също копие на услугата (препратка към нея) при нейното инжектиране;
- “service” (услуга) - функция, която дефинира нова услуга (начин да се споделя и преизползва код между отделните компоненти, посредством механизма за инжектиране който фреймуърка предлага), но заразлика от фабриката, резултата от нейното инжектиране винаги е нова версия на услугата (нова нейна инстанция);

- “value” (стойност) - стойност, която може да се споделя с останлия код, чрез нейното инжектиране като зависимост (не е налична в конфигурационните функции и стойността и може да бъде заменя чрез декоратор);
- “constant” (константа) - (стойност) - стойност, която може да се споделя с останлия код, чрез нейното инжектиране като зависимост (стойността и не може да бъде променяна чрез декоратор);
- “decorator” (декоратор) - функция, която променя поведението на дадена услуга и връща променета услуга;
- “animation” (анимация) - фабрика, която връща обект, описващ анимация (този елемент е наличен, само когато “ngAnimate” модула е зареден);
- “filter” (филтър) - фабрика, която връща механизъм за трансформирането на модел (данни), в подходящ за потребителя формат (може да се използва директно в шаблоните посредством “data | filterName:arg”, където filterName е името под, което филтъра е бил регистриран, а arg е аргумент, който не е задължителен ако филтъра няма нужда от такъв, в противен случай е необходим на филтъра за правилната трансформацията на data;
- “controller” (контролер) - функция, чрез която могат да бъдат създавани нови инстанции на класове, които дефинират логиката на приложението и свързват модела с изгледа посредством контекста, към който са прикачени;
- “directive” (директива) - функция, която връща обект, който описва инструкции към компилатора на Angular за свързването на модела с изгледа и тяхната синхронизация;
- “config” (конфигурираща функция) - функция, която се грижи за настройките на приложението (на доставчиците);
- “run” (изпълняваща функция) - функция, която трябва да свърши своята работа след зареждането на всички модули;

Angular използва механизъм наречен “dependency injection” (инжектиране на зависимости), който позволява споделянето и по този начин преизползването на код. Това става като всички елементи различни от константа, стойност и компонент (те не могат да приемат зависимости) използват конструктивна функция, която приема като аргументи, своите зависимости и връща същинския елемент. Angular, може да позна, кои са зависимостите на един елемент, по името на зависимостта, този механизъм на



разпознаване спира да работи, когато кода на приложението се минифицира (повече за минификацията в следващата глава), защото се минифицират и имената на аргументите, което води до невъзможността те да бъдат разпознати от фреймуърка. Като решение на този проблем е тяхното изрично деклариране чрез прикачането на масив (`$inject`), който оказва всички зависимости към всеки елемент. приемащ таквима чрез символни низове отговарящи на имената, под които са били регистрирани. Аргументите на всяка JavaScript функция, могат да бъдат преставени и достъпвани чрез специален масив, локален за всяка функция - `arguments` (Angular се възползва от това), елементите, на който ще съответстват на елемента, който е бил регистриран под името на съответния елемент, от `$inject` масива. По този начин всяка една зависимост може да бъде достъпена като аргумент на конструктивната функция, дори след минифицирането на имената на аргументите ѝ. Различните елементи приемат различни зависимости:

- фабриките, услугите, директивите, филтрите, анимациите и изпълняващата функция могат да приемат всичко освен доставчици и специалната услуга `$scope`, но имат достъп до сулугата `$rootScope`, която дава достъп до механизмите за синхронизация на модел и изглед от най-горния (началния) контекст;
- контролерите, отново могат да приемат всичко освен доставчици, но са и единствените, които имат достъп до специалната услуга `$scope`, която отговаря на текущия контекст, в който се намира контролера (изключение от това правило са свързващата и заменящата функция, на директивата и заменящата функция на компонента, които също имат достъп до тази специална услуга, защото тя предоставя механизмите за синхронизиране на модела и изгледа от съответния контекст);
- доставчиците и конфигурационите функции могат да примат само константи и доставчици (доставчиците могат да приемат всичко, което може и една услуга, но не директно, това става като нужната зависимост се инжектира като зависимост на полето `$get`, които всеки доставчик е задължен да има, защото той е отговорен за създаването, на нови услуги, а те са достъпни през това поле).

Към процеса на инжектиране на зависимости влизат и други под процеси, като например създаването на единна инстанция на услуга, за пръв път когато такава се декларира като зависимост и нейното разпространение сред останалите елементи, имащи зависимост от нея.

Компилятора, услугата `$compile`, който Angular използва е инструмента, който всъщност свързва отделните елементи. Процесът на компилация се разделя на два отделни под процеса: компилация и свързване. По време на компилация фреймуърка обхожда и разпознава всички директиви по подаден символен низ или DOM елемент и връща свързваща функция.

При зареждане на приложението на компилатора се подава часта от HTML код, която декларира, че ще се обработва от фреймуърка. Това може да стане по два начина

- чрез директивата `“ng-app”` (така нареченото автоматично зареждане), тя приема име на модул, който да се използва при зареждането на приложението, фреймуърка ще подаде елемента, върху който е поставена директивата към компилатора;
- чрез програмния интерфейс, който Angular предоставя - `angular.bootstrap(element, modules)`, като по този начин могат да бъдат заредени повече от един модул, и към компилатора ще бъде подаден елемента подаден към `bootstrap` метода.

Свързващата функция приема 3 параметра, като само първия е задължителен: контекст (обект от тип `Scope`), клонираща функция и специфични инструкции към процеса на свързване. Свързващата функция използва данните от подадения контекст, за да запълни всички полета от шаблона, които случая е това, което е било подадено на компилатора, преди това. След първоначалната компилация компилатора може да бъде викан от всяка точка на приложението имаща достъп до него за компилацията на всички останали директиви, които не са били налични при зареждането. Контекста в Angular същност представлява часта на модела от MVC архитектурата, до който във фреймуърка само контролерите и споменатите по-горе 3 изключения имат достъп и могат да променят.

Чрез целия процес на компилация архитектурния модел, който Angular ползва драстично се различава от традиционния MVC модел, при който контролера е този, който типично свързва модел и изглед в едно.

## 2.5 Избор на CSS фреймуърка "Angular-Material"

Взимането на решение, кой точно CSS фреймуърк да ползва едно Интернет приложение в днешни дни е труден, защото съществуват много възможности. Най-известните сред тях са "Bootstrap" [34], "Semantic-UI" [35], "Foundation" [36] и много други. В резултат, на което изборът бе сведен до: "Angular-Material" [37] (Material), "UI-Bootstrap" [38], "Semantic-UI-Angular" [39] и "Foundation-Angular" [40], които имат интеграция с избрания фреймуърк - Angular. Като последните два бяха отхвърлени: Semantic-UI-Angular, защото се намираше в процес на разработка и нямаше нито една официално излязла версия, а Foundation-Angular, заради липсата на опит с фреймуърка Foundation.

UI-Bootstrap се разработва от "Angular-UI" и целта му е да улесни интегрирането на Bootstrap CSS фреймуърка в приложения, използващи Angular. За целта голяма част от Bootstrap е опакован в директиви, които са лесни за използване. Това опаковане е нужно, защото от Angular препоръчват промяната на стилове и класове на елементи да се извършва само и единствено в директиви.

Angular-Material е фреймуърк разработван от екипа, който разработва Angular и Angular 2. Целта му е да предостави лесно интегриране на правилата относно дизайна на приложения разработен от "Google", наречен "Material design" [41], в приложения ползващи двата фреймуърка.

Material и UI-Bootstrap използват модула "ngAnimate" [42], за анимациите, които предоставят. И двата фреймуърка използват наставки за контролирането на видимостта на съдържанието на приложението на различни размери екрани (наставките с еднакви имена не отговарят за един и същ размер на екрана), но Bootstrap ги разделя само на 4 категории, докато Material на 9. Bootstrap използва само "grid" (гريد) система, за наместването на HTML елементите, докато Material, макар да има вградена таква, залага повече на технологията "flexbox" [43], която е много по-лесна за ползване. И двата фреймуърка използват стандартни имена на цветовете, които ползват. Bootstrap използва 6 цвята ("default", "primary", "success", "info", "warning" и "danger"), докато

Material ги свежда само до 3 (“primary”, “warn”, “accent”) , което създава по-лесно разграничение, кой кога да бъде използван. За промяната на цвета, който отговаря на дадено име в Bootstrap е нужна промяна на CSS файла, който фреймуърка ползва (обикновено става чрез сайта на фреймуърка или тегленето на готова тема). В Material този проблем е решен посредством програмния интерфейс, който фреймуърка предоставя под формата на доставчик, за създаването на услуга, който може да създава и конфигурира теми, което от своя страна позволява ползването на отделни теми с един единствен css файл в едно и също приложение, докато това е не възможно с Bootstrap. Material има интеграция с допълнителния модул “ngMessages”, а UI-Bootstrap няма.

След сравнението на двата фреймуърка бе избран Material, въпреки факта, че имам опит с Bootstrap технологията.

## **2.6 Избор на "UI-Router" като технология за виртуализация на страниците на приложението**

Едно от предназначенията на Angular е да спомага за създаването на така наречените “Single page application” (едно-странични) приложения. Едно-страничните приложения представляват приложения, при които за промяната на изгледа или данните не е нужно повторното презареждане на страницата на приложението или промяната и с друга. Най-популярната имплементация на този архитектурен модел е чрез разделянето на приложението на виртуални страници и една реална. Механизмите, които Angular предоставя, за реализацията на този подход са налични в допълнителния модул “ngRoute” [44], който за виртуализация използва директивата “ngView”, а за оказване на конкретния изглед се използват виртуален адрес отговарящи на съответната виртуална страница. Използваната от модула директива не може да бъде използвана повече от веднъж в един изглед, което затруднява виртуализирането на виртуалните страници. За симулирането на вложена виртуализация се налага използването на други директиви, като: “ngSwitch”, “ngIf” и “ngInclude”. Друг проблем свързан с модула е необходимостта всяка страница да има отделен адрес, което не винаги е нужно, съществуват случаи, в които това е дори нежелано от разработчика. В случая на разработваната дипломна работа това е виртуалната страница отговорна за

създаването на нови поръчки. Проблемът, който възниква когато всяка страница си има уникален адрес е, че той може да бъде достъпен от потребителя, по всяко време, а за преминаването от една виртуална страница към друга е необходимо задължителното минаване през предходната страница, тъй като всяка страница добавя нещо към поръчката и всяко едно е зависимо от предходното. Като решение на този проблем бе взето решение да се използва външния за фреймуърк модул “ui.router” [45], разработван от Angular-UI. Той предоставя по-богати механизми за контролиране на виртуалните страници, като освен, че дава решение на описаните по-горе проблеми, представя и възможност за управление на паралелни виртуални страници и не е необходимо ползването на директива различна от тази, която модула ползва - uiView, неща невъзможни с предоставеният от Angular модул. Вместо виртуални адреси и съответни страници, избраният модул организира механизмите за промяна на изгледа около състояния, в които може да се намира приложението.

## 2.7 Избор на "Stylus" за CSS препроцесор

За визуализация на поръчката е необходимо добавянето на допълнителни стилови файлове, което доведе до решението, че към избраните технологии трябва да бъде добавен подходящ стилос препроцесор, който да олесни разработката на дипломната работа. Най-използвани съществуващи решения за стилос препроцесор са: “Sass” [46], “Less” [47], “Stylus” [48] и други. След проучване и сравнение на съществуващите решения бе стигнато до заключението, че повечето препроцесори предлагат сходни механизми олесняващи писането на стилови файлове и трудно ще може да се определи, кой от тях е най-добър и на базата на липсващ предишен опит с използването на подобна технология бе избран Stylus, който е създаден със замисъл да се използва в приложения ползващи средата Node. Във Stylus използването на запетай, кръгли и къдрави скоби е не задължително. Той поддържа така наречената блокова нотация, при която чрез индентация на кода се описва йерархичната структура на CSS селектори, чистия CSS позволява тази структура да бъде оказана само чрез последователно изброяване на селектори. Възможностите, които Stylus предоставя като олеснение за писането на стилови файлове са:

- Променливи;
- Интерполация;
- Оператори;
- Миксини;
- Функции;
- Аргументи с оказания под формата на ключови думи;
- Вградени функции;
- Условя (условни оператори);
- Директното използване на JavaScript обекти (хашове);
- Итератори (цикли);
- Механизми за свързване на отделни файлове в други посредством “@require” и “@import”;
- “@...”, всички механизми, които чистия CSS поддържа и започват с “@”, могат да бъдат използвани в комбинация с блоковата нотация, с която препроцесора идва;
- Механизъм “@extend”, който един от основните начини за преизползване на код (вдъхновен от едноименния механизъм използван от Sass);
- Механизъм “@block”, който позволява присвояването на всеки валиден Stylus блок като стойност на променлива, даващо възможност той да бъде извикван навсякъде и така преизползван по всички възможни начини;
- JavaScript функция url(), която да замени едноименната в Stylus код, за олесняването на построяването на адреси на използвани ресурси;
- Механизъм “@css”, за директното използване на чист CSS код.

## 2.8 Избор на HTML препроцесора "Jade"

След избора на CSS препроцесора Stylus бе взето решение да се проучат и използват подобни технологии, които да олеснят писането на код на другите два основни компонента (HTML и JavaScript) на едно Интернет приложение. Съществуват много и различни HTML решения за препроцесор, но повечето от тях представляват отделни езици за писане на шаблони. Повечето са не съвместими със шаблоните, които

Angular използва, затова бе избрана технологията “Jade” [49], използвана само като препроцесор за езика HTML

Jade представлява напълно отделен език за писане на шаблони, който се компилира до HTML. Подобно на блоковата нотация, която Stylus използва Jade също разполага с такъв механизъм, който се използва за описание на йерархичната структура от елементи, която HTML описва. Но докато при HTML ползването на консистентната индентация на кода с цел по-ясното описание на йерархията е не задължително, то при Jade то е необходимост. Друга основна разлика между двата езика е липсата на затварящи тагове при препроцесора. При него пропускането на правилна индентация на кода дори само на едно място може да доведе до два тотално различни резултата. Като в допълнение на изброените причини, избрания препроцесор има способността по подразбиране да намалява количеството код на изходния HTML файл след компилацията на съответния Jade файл.

## **2.9 Избор на езика "Coffee-Script"**

Решението за избор на технология, която да олесни писането на JavaScript бе направено изключително лесно, въпреки дългия списък [50] от технологии, които могат да заменят чистия JavaScript, като впоследствие се компилират до него. Като технология, която да замени JavaScript бе избран езика за програмиране “CoffeeScript” [51], който води своето начало от езика за програмиране Ruby, впоследствие е взел доста неща и от езика за програмиране Python. И двата езика са ми добре познати и се вписват в мотото “write less, do more” (пиши по-малко, прави повече), което беше една от причините за олеснението на избора.

## **2.10 Избор на изграждаща система "Gulp"**

Взимането на решение за изграждаща система на едно приложение е труден избор, защото съществуват много технологии, които могат да се използват. Избрания език за програмиране включва изграждащата система “Cake”, която е аналог на изграждащите системи “Make” [51] (Unix/C/C++) и “Rake” [52] (Ruby). Тъйкато

Coffee-Script се компилира до чист JavaScript може да бъде използвана и всяка изграждаща система, която използва този език, което затруднява избора още повече. Такива системи са “Jake” [53], “Grunt” [54], “Gulp” [55, 56] и други. Заради много съществуващи решения избора бе сведен само до Grunt и Gulp, които са най-използваните сред Node приложенията. Изграждащата стемата Gulp бе избрана пред конкурентната Grunt, защото според статистиките (Фиг. 2.10.1), които npm предоставя за двата пакета, Gulp е използвана повече и има по-малко проблеми (“open issues”). И двете системи използват “plugins” (добавки), за да олеснят процеса на изграждане на приложения, но Grunt е чисто описателен, като за целта използва JSON формат, докато Gulp е изграден около идеята за работа с потоци и трансформацията на данните в тях.



(Фиг. 2.10.1) Статистики от сайта на пакетния мениджър npm за Gulp и Grunt

## ТРЕТА ГЛАВА

### Описание на разработения продукт

#### 3.1 Алгоритъм на изграждане на приложението

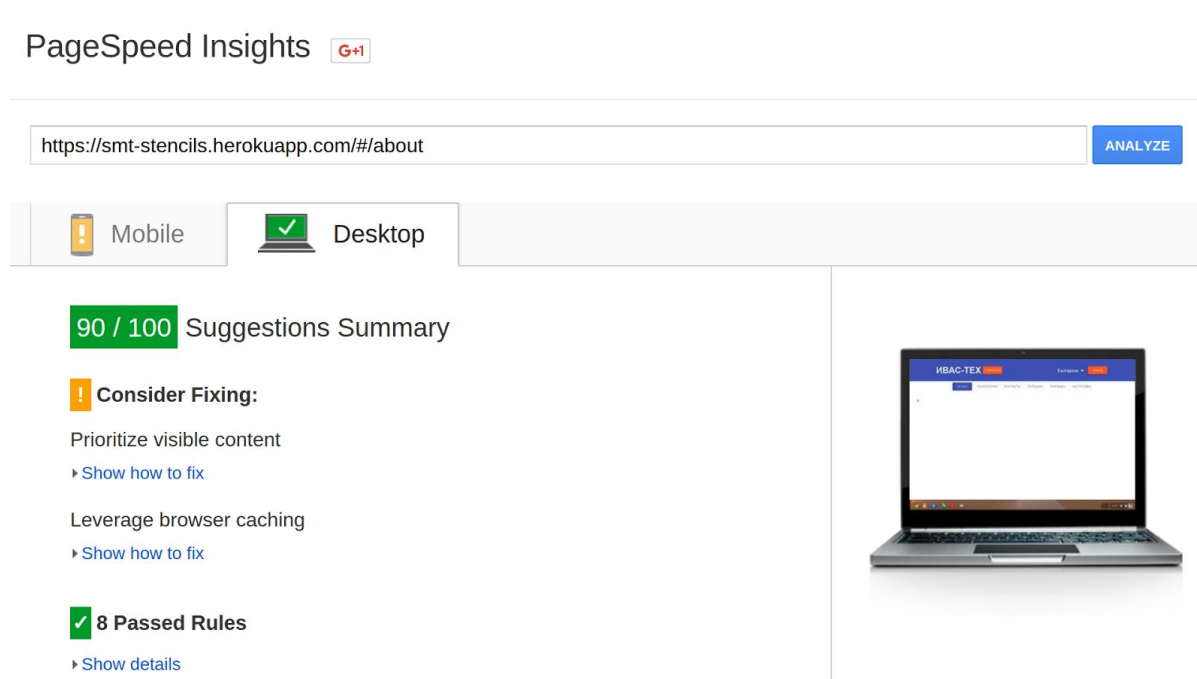


Задачите дефинирани във файла “gulpfile.coffee” описват целия процес по изграждането на приложението.

1. Премахват се директориите: “./templates”, “./build”, “./resources” и “./deploy”;
2. Всички Jade файлове (завършващи на “.jade”) от папката “./client” се компилират до техните еквивалентни HTML файлове в папката “./templates”, като се запазва тяхната йерархична структура (подредбата им в папката “./client”);
3. Файла “index.html” от папката “./templates” се премества в папката “./build/inline”;
4. Всички останали файлове се буферират директно в “кеш паметта” на Angular “\$templateCache”, чрез генерирания от Gulp добавката “gulp-angular-templatecache” модул “templates”;
5. Всички Coffee файлове (завършващи на “.coffee”) от папката “./client” се компилират до техните еквивалентни JavaScript файлове в папката “./build”, като се запазва тяхната ерархична структура (подредбата им в папката “./client”);
6. Всички Coffee файлове от папката “./server” се компилират до техните еквивалентни JavaScript файлове в папката “./deploy”, като се запазва тяхната първоначална ерархична структура от папката “./server”;
7. Всички JavaScript файлове от папката “./build” се събират в един единствен файл, чрез технологията browserify, заедно със файловете представляващи зависимости на приложението (фреймуърка Angular, всички използвани допълнителни модули и библиотеката “chart.js” (използвана от модула “angular-chart.js”)). В разработванта дипломна работа оказаните пътища до файловете от ключовото поле “browser” на файла package.json служат за оказване на browserify да използва минифицирните варианти на зависимостите или за съкратено изписване на техните имена при използването на механизма require. Обединения файл се запаметява като “bundle.js” в същата папка;
8. Файла съдържащ обединените JavaScript файлове се минифицира и премества в папката “inline”;

9. Stylus файла "style.styl" от папката "./client/styles/", който събира останалите стилкови файлове написани на езика Stylus се компилира до еквивалентния "style.css" и се минифицира използвайки приставката "gulp-uglifycss", като по този начин всички допълнителни стилове се съдържат в минифицирания файл;
10. Получения CSS файл се обединява заедно със CSS файловете на Angular-Material и Angular-Chart.js, като той отново се минифицира и премества в папката "inline";
11. Трите файла ("index.html", "bundle.js" и "style.css") се обединяват заедно в "index.html", чрез използването на приставката "gulp-inline", който се премества в папката "send" намираща се в папката "./deploy"; Файла "index.html" се компресира до формата "gzip", който се поддържа от всички браузъри;
12. Клонира се git хранилището "IvasTech/SMT-Stencils\_resources" , качено на сайта github [57], което е част от разработката на дипломната работа, като то служи за съхранението на файла "favicon.ico", който представлява иконката на приложението и файла "top.html", който представлява предварително направена трансформирания на Gerber файлове и служи за демонстрация на промяната на изгледа на стенсила при създаването и промяната на конфигурации от потребителя;
13. От клонираното хранилище се премества и компресира във формат gzip файла "top.html" в папката "./deploy/send/templates", за да бъде изпратен до потребителя на приложението, само когато той създава или променя дадена конфигурация за характеристиките при изработването на стенсил файл/ове, като по този начин времето за зареждане на приложението се намлява;
14. Файла "favicon.ico" също претърпява компресация в gzip формат и се премества, в папката './deploy/send', от където може да бъде изпратен при заявка от браузъра на потребителя на приложението;

За изпълнението на задачите се използва командата "npm run build", която стартира описания алгоритъм и ако завърши успешно се извиква командата "npm run clean", която повтаря стъпка 1 от описания алгоритъм на изграждане. Целта на дефинираните задачи е времето за зареждане на приложението да бъде максимално намалено (Фиг. 3.1.1).



(Фиг. 3.1.1) Резултатите от тестването на приложението с PageSpeed

## 3.2 Комуникация между клиент и сървър на приложението

Клиента и сървъра на приложението си комуникират чрез HTTP заявки. Имплементирани са механизми за описание на заявките и в двете страни на приложението.

### 3.2.1 Динамично изграждане на възможните заявки с цел премахване на повторения и лесен начин за добавянето на нови

В двете страни на приложението е използвано динамично изграждане на заявките, чрез описание на тяхната структура с цел премахване на повторения и лесно бъдещо развитие на приложението.

### **3.2.1.1 Избягване на повторения във файловете описващи заявките, които сървърът може да приеме**

Беше взето решение множеството малки, но често повтарящи се фрагменти от код сред файловете, описващи възможните заявки, които сървърът приема, да бъдат избегнати. За целта беше създаден алгоритъм, описан и реализиран във файла “/server/lib/routerTree.coffee”, който освен премахването на повторения олеснява и описването на възможните заявки. Функцията, която този файл експортира, позволява всички възможни пътища на заявките, идващи към сървърът на приложението, да бъдат описани чрез обект, чиято структура описва отделните рутери - как те трябва да бъдат композирани и как отделните функции посредници трябва да бъдат монтирани към съответния рутер. Функцията използва рекурсивен метод за обхождане на ключовите полета на подадения обект и връща рутера, който те описват, като при “развиването” на рекурсията рутерите биват монтирани един към друг и резултатът е един единствен рутер.

### **3.2.1.2 Избягване на повторения при описването на заявките, които клиента на приложението може да направи към сървърът чрез тяхното описване**

Аналогично на избягването на повторения при описването на заявките, които могат да бъдат отправени към сървърът на приложението беше използван подход на тяхното описване, чрез обект и в кода на Интернет приложението. Структурата на който драстично се различава от тази на обекта, използван за описване на рутери. Обекта, който служи за описание на възможните заявки се задава, чрез метода “setRequests” на доставката “RESTHelperServiceProvider”. Повторенията, които се избягват чрез използването на обект за описание са свързани с данните, които се изпращат до сървърът на приложението и логиката по определянето коя функция да бъде извикана след правенето на всяка една заявка и получаването на отговор от сървърът - дали да се извика функцията за обратна връзка на заявката или функцията отговорна за обработка на грешки. Решението, коя от двете функции да бъде извикана

става чрез метода “then” на обекта от тип Обещание, който всяка заявка връща и проверка на статус кода на заявката. Методът then приема две функции - първата функция бива извикана, ако не настъпи грешка, а втората съответно ако настъпи. Angular по подразбиране приема за успешно преминала заявка, всяка чийто статус код е между 200 и 299, в разработваната дипломна работа за заявка, при която не е настъпила грешка се счита само тази, чиято статус код е 200. Структурата на използвания за описание обект е такава, че той едновременно описва възможните заявки и динамично изграждания интерфейс на услугата “RESTHelperService”. Всяко ключово поле с изключение на полето “alias” има за стойност масив от символни низове, които заедно с имената на полета описват интерфейса на услугата RESTHelperService. Имената на полета отговарят и на ресурсите, които се използват в адресите на отправяните заявки. Услугата RESTHelperService използва услугите “RESTService” и “UploadService” за правенето на заявки, като UploadService се използва само за елементите от масива, който е стойност на ключовото поле “upload” на обекта използван за описание на заявките. Полето alias има за стойност обект, чиито ключови полета са имена на HTTP методи, които имат за стойности масиви от символни низове. Те отговарят на стойностите от останалите ключови полета (с изключение на полето upload) с цел абстракция на заявката, която трябва да бъде отправена. Имената на методите подсказват конкретно действие, което трябва да се предприеме, а всеки обект, носи име, което подсказва формата на информацията, с която се бори. По този начин се избягва запаметяване на адреси и методи на заявките и се повишава четимостта на кода. Пример е метода “register” (Фиг. 3.2.1.2.1) на диалоговия контролер “registerController”, в който е пределно ясно, какво действие трябва да бъде предприето и какъв е формата на изпращаните данни - да се регистрира нов потребител на приложението.

```
ctrl.register = (valid) ->
  if valid
    ctrl.user.language = $translate.use()
    RESTHelperService.user.register user: ctrl.user, (res) ->
      ctrl.hide "success"
```

(Фиг. 3.2.1.2.1) Метода *register* на контролера *registerController*

### **3.2.1.3 Заключение и обяснение как се постига лесното бъдещо разширение на заявки**

Благодарение на описването на заявките, от двете страни на приложението чрез обекти лесно могат да бъдат добавени нови. За разширението на подържаните заявки трябва да бъде създаден обект отговарящ на новия рутер, който ще описва заявката/ите от страната на сървъра и той да бъде добавен към файла “/server/routes/routes.coffee”. А от страната на клиента във файла “/client/requests.coffee” трябва само да се опише/ат новата/ите заявки по задения модел, което значително олеснява бъдещето развитие на приложението, премахва всички възможни паразитни повторения и повишава четимостта на кода.

### **3.2.2 Описание на услугите използвани за връзка между клиент и сървър**

Заявки от Интернет приложението към сървъра му се изпращат чрез динамично изгражданите методите на услугата “RESTHelperService”, която използва услугите “RESTService” и “UploadService” за самото изпращане на заявките.

#### **3.2.2.1 Описание на услугата "RESTService", използвана за подаването на заявки, които не включват прехвърляне на файлове, към сървъра**

Услугата RESTService представлява “closure” функция (Фиг. 3.2.2.1.1), която улеснява правенето на заявки. Услугата се конструира от доставчик, който се използва за конфигурирането на базов адрес на заявките с цел лесното разграничаване на заявките, служещи за комуникация между сървъра и приложението (синхронизация на данни). Базовия адрес се конфигурира посредством извикване на метода “setBase” на доставчика, неговата стойност може да бъде взета чрез метода “getBase”. Услугата връща функцията приемаща име на ресурс при нейното инжектиране и връща функция,

която приема два аргумента: метод на заявка и аргумент, който е информацията, която да бъде изпратена до сървъра и се грижи за правилното построяване на адреса на заявката. Изпращаната информация може да бъде както данни във JSON формат (JavaScript обект, който автоматично се преобразува до неговия еквивалент от използваната услуга “\$http”), така и параметър, когато заявката е с метод “get” или “delete”.

```
(resource) ->
url = _base + "/" + resource
(method, data) ->
$http
    method: method.toUpperCase()
    data: if ($filter "isEmpty") data then data else undefined
    url: if method in ["get", "delete"] and typeof data is
"string" then url + "/" + data else url
```

*(Фиг. 3.2.2.1.1) Функцията връщана при инжектирането на услугата RESTService*

### **3.2.2.2 Описание на услугата “UploadService” използвана за изпращането на заявки, за прехвърляне на файлове към сървъра**

Услугата UploadService представлява “опаковка” (Фиг. 3.2.2.2.1) на предоставената от използвания модул “ng-file-upload” услуга “Upload” за изпращане на файлове по Интернет. Тя е обособена в доставчик, чрез който може да се окаже частичния адрес, на който да се изпращат файлове до сървъра на приложението. Това е с цел разграничаване на заявките, които приемат файлове. Отново този адрес може да бъде зададен чрез метода setBase на доставчика. Функцията, която се връща в резултат на инжектирането на услугата подобно на услугата RESTService приема като аргумент име на ресурс и е отговорна за правилното построяване на пълния адрес на заявката. Особеност на разработваното приложение е оказването на всеки от изпращаните файлове на кой конкретен слой отговаря. Това става като функцията, която се връща

от услугата UploadService приема обект, чиито полета означават имената на слоевете, а стойностите им са файловете, които да бъдат изпратени. Това е с цел решаването на следния проблем: услугата Upload може да изпраща няколко файла до сървъра само като масив. Информацията от подадения обект се преобразува, за да се запази указанието: кой файл на кой слой отговаря. След преобразуването до сървъра на приложението се изпращат масив от файлове и обект, на който полетата са имена на файловете, а стойностите им са имена на слоевете. Сървърът от своя страна е натоварен със задачата отново да преобразува изпращаната информация до първоначалния ѝ формат, с разликата, че имената на файловете ще отговарят на пътищата, на които те могат да бъдат достъпни на сървъра на приложението.

```
service = (Upload) ->
  (url) ->
    base = RESTServiceProvider.getBase()
    chain = [base, _base, url]
    noBase = _base is ""
    noUpload = base is ""
    if noBase then chain.splice 1, 1
    if noUpload then chain.splice 0, 1
    URL = if noBase and noUpload then "/" + url else chain.join "/"

  (files) ->
    data = map: {}, files: []
    for layer, file of files
      data.map[file.name] = layer
      data.files.push file
    Upload.upload url: URL, data: data
```

*(Фиг. 3.2.2.2.1) Код на услугата UploadService*



### 3.3 Обработка на грешки при връзката клиент - сървър и тяхното съхранение на сървъра

Грешките настъпили при връзката клиент - сървър се записват във файлове, отворени като потоци, на сървъра на приложението.

#### 3.3.1 Обработка на грешки настъпили при клиента

Услугата `errorHandleService` служи за изпращане на информация, която да бъде запазена на сървъра под формата на логове, когато в резултат на направена заявка се получи грешка от страната на Интернет приложението. При настъпването на грешка услугата изпраща до конфигурирания адрес данни, оказващи пълна информация за заявката: адрес, метод, заглавия и информация, върната като отговор от сървъра на направената заявка. Функцията, която прихваща направената заявка, носи информация относно настъпила грешка и я записва във файл, който е отворен като поток. В него постоянно може да се добавя информация относно регистрирана нова грешка. Функцията е част от обекта, който описва рутер, приемащ само `post` заявка и се връща в резултат на извикването на функцията, експортирана от файла `"/server/routes/logErrorHandle.coffee"` (Фиг. 3.3.1.1), чрез подаването ѝ на път до лог файла, в който да се записват настъпилите грешки.

```
module.exports = (log) ->
  stream = createWriteStream log, flags: "a"
  post: (req, res, next) ->
    stream.write (JSON.stringify req.body) + "\n", "utf-8", (err) ->
      if err then next err else query res
```

(Фиг. 3.3.1.1) Код на функцията, експортирана от файла `/server/routes/logErrorHandle.coffee`

### 3.3.2 Обработка на грешки настъпили при обработката на заявки от сървъра

За логването на грешки, настъпили при обработка на постъпили заявки към сървъра се използва комбинация от функции посредници. Ключовото поле “beforeEach” на обекта, който описва заявките приемани от сървъра, обозначава две посреднически функции, които да се извикват за всяка заявка идваща към сървъра на приложението. Посредническата функция, връщана от извикването на метода `json` на обекта `bodyParser` е необходима защото тя създава обект, отговарящ на тялото на всяка една заявка. Функцията “`errorLogger`” приема аргумент, който представлява поток и служи за конфигурация на посредническата функция, експортирана от пакета “`morgan`”. Този пакет предоставя механизми за логването на заявките идващи към сървъра и конфигурирането на формата за записване на информация. Използваната конфигурация служи за логване само на заявки, при които е възникнала грешка, тоест техният статус код е по-голям или равен на 400. Полето “afterEach” съдържа две функции (Фиг. 3.3.2.1). Първата се извиква единствено когато за идващата към сървъра заявка не е дефиниран път. Тя създава нов обект от тип грешка и го подава на механизма `next`, който директно го препраща към втората функция, която се използва като механизъм за прихващане на грешки. Функцията експортирана във файла “`/server/errorHandler/coffee`” приема два аргумента: поток, в който грешките да бъдат логнати и адрес, който да пренасочи заявките, за които не е дефиниран път и връща функция за прихващане на грешки.

```
afterEach: [  
  (req, res, next) -> next new Error "Not Found"  
  errorHandler errorStream, "/#!/notfound"  
]
```

(Фиг. 3.3.2.2) Полето *afterEach* от описанието на рутера, описващ всички заявки

Прихващача на грешки проверява за съобщението на грешката. Съобщенията биват класифицирани в три отделни групи:

1. Направена е заявка към ресурс, който не достъпен (дефиниран) на сървъра, в резултат, на което на променливата *status* се присвоява числото 404;
2. Направена е заявка за ресурс, който изисква изпращача ѝ да е разпознат от сървъра на приложението, за статус се използва стойността 401;
3. Възникнала е друг вид грешка при обработката на получена заявка, за статус се използва числото 500 и се добавя съобщение към грешката.

Следва проверка на статус кода на грешката и се проверява, дали тя е възникнала в резултат на ненамерен ресурс и ако тя е възникнала в следствие на такава заявка, като отговор се изпраща препращане към подадения адрес на функцията, създава прихващача. Ако грешката е категоризирана в една от другите две категории, то в резултат на получената заявка се изпраща обект, съдържащ съобщението на грешката и нейния код (Фиг. 3.3.2.2).

```
module.exports = (errorStream, redirect) ->
  (err, req, res, next) ->
    switch err.message
      when "Not Found" then status = 404
      when "Unauthorized Access" then status = 401
      else
        status = 500
        err.message = "Internal Server Error"
    send = res.status status
    if status is 404 then send.redirect redirect
    else send.send error: status, message: err.message
    errorStream.write err.stack + "\n", "utf-8"
```

(Фиг. 3.3.2.2) Кода на функцията, която връща използвания от сървъра прихващач на грешки

Грешка, която попада в категорията на статус код 401 може да възникне, след като дадена заявка премине през използваната посредническа функция и бъде отхвърлена, защото нейният изпращач не е разпознат от сървъра (не е аутентициран). Като за проверка се използва полето `user`, което бива прикачено към заявката ако потребителя е разпознат от системата на сървъра, тоест има създадена за него сесия, която се създава, след като той се логне в системата. Ако то е дефинирано, проверяваната заявка успешно преминава към следващата функция дефинираща път за нея. Ако такова поле липсва, се проверява дали заявката не попада в списъка на заявки, за които не е нужно разпознаване от сървъра. Това са заявките отговарящи за регистрация на нови потребители, вход в системата и проверката за разпознаване и в такъв случай те също продължава напред, в противен случай се сигнализира за настъпила грешка 401 (Фиг. 3.3.2.3), която се обработва от вече описания прихващач на грешки.

```
module.exports = (req, res, next) ->
  if not req.user?

    checkForLogin = (req) ->
      req.url is "/login" and req.method is "GET" or req.method is
"POST"

    checkForUser = (req) ->
      (req.url.match "/user")? and req.method is "POST" or
req.method is "PUT"

    if (checkForLogin req) or (checkForUser req) then next() else
next new Error "Unauthorized Access"
```

```
else next()
```

*(Фиг. 3.3.2.3) Кода на функцията, която ограничава обработката на заявки, за които е нужно разпознаване от системата*

### **3.4 Обработка на грешки настъпили в клиентската част на приложението (не настъпили в резултат на отправяне на заявка към сървър)**

За записването на грешки настъпили по време на работа на Интернет приложението се използва декоратор, който да промени поведението на услугата “\$ExceptionHandler”, която Angular използва за известяване след получена грешка. Поведението на услугата се променя, като вместо да съобщи за настъпила грешка, чрез съобщение в конзолата на браузъра, показващо пътя на грешката, тя изпраща заявка до сървър, на който да се отрази този път. Този вид грешки се записват в лог файла “/deploy/client.log”.

### **3.5 Премахване на паразитни повторения във файловете използвани за създаването на Mongoose модели**

Подобно на решението за премахване на повторения от файловете, дефиниращи пътищата на заявките бе създаден файла

“/server/lib/makeModel.coffee”, който решава следните проблеми:

1. Позволява като ключово поле да се използва “type”, което Mongoose разпознава единствено като обозначение на тип на поле от схема;
2. Позволява съкратен запис за обозначаване на връзки между колекциите, като за целта е необходимо само обозначаването на името на модела, към който трябва да бъде направена връзка;
3. Позволява съкратен запис за обозначаване на стойности по подразбиране на полета от схема, като се подаде масив от два елемента. Първия елемент е типа на полето, а втория стойността по подразбиране;

Функцията, която този файл експортира приема като аргумент име на модел, който трябва да бъде създаден и обект, който представлява описание на схема за създаване на модела и връща създадения Mongoose модел.

### **3.6 Услугата “showDialogService” използвана за създаване и управление на диалозите, които Интернет приложението използва**

В разработваното Интернет приложение силно се използват персонализирани диалози, които се визуализират до потребителя чрез използване на услугата “\$mdDialog”, предоставена на разработчика от фреймуърка Angular-Material, за създаването и управлението на нестандартни диалози. За улеснение бе създадена услугата “showDialogService”, която е отговорна за създаването и управлението на използваните в приложението диалози. Тя улеснява създаването на диалозите, като позволява преизползване на конфигурационния обект за създаване на нови диалози като шаблонизира символните низове оказващи адрес на изгледа, име на контролера отговарящ за този изглед, име под което той да бъде инжектиран в контекста на диалога, локални променливи, които да бъдат директно свързани с контролера на диалога и от къде да се отвори и затвори диалога (Фиг. 3.6.1). От друга страна тя улеснява управлението на различните диалози като предоставя механизъм, който може да променя поведението на диалога след натискането на някой бутон от изгледа му или като придава поведение по подразбиране на диалога, което от ново може да бъде променяно чрез същия механизъм. Тази промяна на поведението на диалозите, позволява тяхното пълноценно преизползване, без промяна в техния код. Механизма позволяващ всичко това се основава на факта, че всеки диалог представлява обект от тип Обещание, който се връща от услугата \$mdDialog след създаването на всеки диалог. Обещанията са в състояние “pending” докато диалога е видим от потребителя на приложението и се скрива само когато състоянието се промени в състояние “resolved” или в състояние “rejected”, чрез използването на методите “hide” или “cancel” на услугата \$mdDialog. При промяна на състоянието на Обещанието, съответния диалог се затваря и като причина за затварянето към функциите, които се използват за обратна връзка се подава същия аргумент, който е бил подаден при

затварянето на диалога. За улеснение бе избрано да се ползва само метода `hide`, който променя състоянието на диалога от `pending` в `resolved`. Като за причина за затварянето на диалога бе избрано да се ползват или символен низ, който отговаря на поле от подаден към услугата `showDialogService` обект като нейн 4-и или 5-и аргумент в зависимост от ефекта, който се търси - промяната на поведение на даден диалог или добавянето на поведение към него или обект, чийто имена на ключови полета отговарят на съответно подаден аргумент на услугата, а техните стойности се използват като аргументи при извикването на функцията за обратна връзка. Промяната на поведението или добавянето на ново става като се итерира през всяко поле и стойност от обекта `extend` и съответните стойности от обекта `handle` се заменят, ако вече съществуват или биват добавени. След затварянето на диалога се извиква функция, която приема като аргумент причината за затварянето на диалога с цел взимане на решение, коя или кои функции на обекта `handle` да бъдат извикани.

```
(event, dialog, locals, handle = {}, extend = {}) ->
  locals.hide = $mdDialog.hide
  if extend? then handle[key] = value for key, value of extend
  $mdDialog
    .show
      templateUrl: dialog + "View"
      targetEvent: if event.target? then event else undefined
      controller: dialog + "Controller"
      controllerAs: dialog + "Ctrl"
      bindToController: yes
      locals: locals
      openFrom: "body"
      closeFrom: "body"
      escapeToClose: no
    .then (reason) ->
      if ($filter "isEmpty") reason
```

```

for key, value of reason
    if handle[key]? then handle[key] value
else if handle[reason]? then handle[reason]()

```

(Фиг. 3.6.1) Код на услугата *showDialogService*

### 3.6.1 Демонстрация на услугата "showDialogService", чрез описване начина на работа на услугата "loginService"

Услугата "loginService" се конструира от услугата "circuitDialogService" и се използва за показване на диалога за вход в системата. Услугата circuitDialogService приема име на диалог, ключ за превод на извежданото съобщение за грешка и не задължителна функция, която да се извика, когато диалога се затвори с причина "success". В случая на разглежданата услуга (loginService) се подава функция, която приема обект, използван за аутентикация на потребителя, посредством услугата "authenticationService". Връщаната услуга от circuitDialogService приема като аргументи event (обект, който представлява настъпилото събитие) и handle (обект, който да се използва за промяна на поведението на конструираната услуга, служеща за показване на нов диалог) (Фиг. 3.6.1.1). Показването на новия диалог става посредством услугата showDialogService, като към нея се подават функции отговарящи за причините "success" и "fail", чрез подавания обект handle. За поведението на диалога при затварянето му с причина success се подава директно аргумента success, ако той не бил подаден по потразбиране стойността му ще бъде undefined, което не е проблем за услугата showDialogService, защото няма да бъде направено извикване на функция ако стойността отговаряща на дадена причина не е дефинирана. А като поведение при затваряне на диалога с fail ще бъде показан диалога tryAgainService (Фиг. 3.6.1.2), който ще покаже съобщение за настъпилата грешка. Той от своя страна, ако бъде затворен с причина success отново ще покаже на потребителя предходния диалог, в случая на разглежданата услуга, това е диалога за вход в системата. Това става като на функцията, връщана като услуга за показване на нов диалог се даде име, като по този



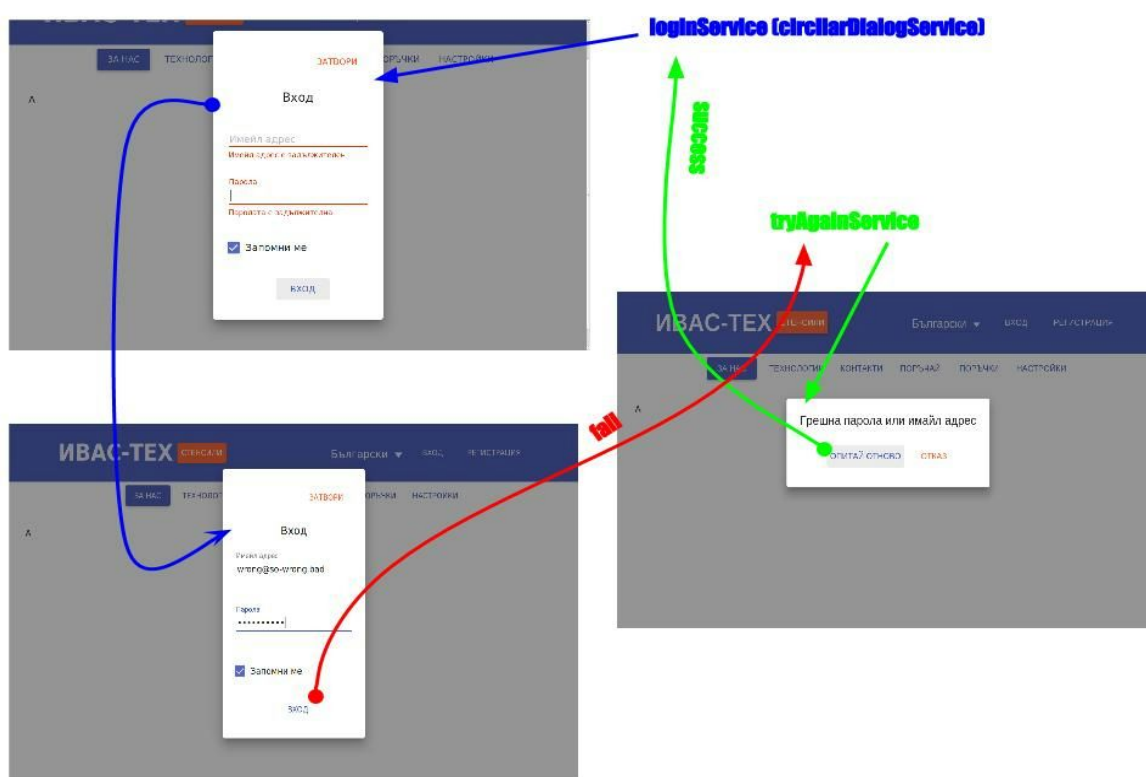
начин връщаната услуга и извикваната услуга при затварянето ще бъдат една и съща функция, защото и двете ще представляват препратка към тази функция. По този начин се заобикаля потенциалния кръг на зависимостите (“circular dependency”), от където идва и името на услугата.

```

service = (showDialogService, tryAgainService) ->
  (dialog, wrong, success) ->
    circular = (event, extend) ->
      handle = success: success, fail: ->
        tryAgainService event, "title-wrong-" + wrong, success: (->
circular event, extend), cancel: handle.cancel
        showDialogService event, dialog, {}, handle, extend
      circular

```

(Фиг. 3.6.1.1) Код на услугата circularDialogService



(Фиг. 3.6.1.2) Принцип на работата на услугата circularDialogService, чрез услугата loginService

Когато услугата `loginService` бъде извикана, се отваря нейния изглед, който дава възможност на потребителя да влезе в системата на сървъра. Като за целта потребителя трябва да въведе адреса на електронната си поща и паролата си. При натискане на бутона за вход в системата, се извиква метода `login` на контролера на диалога, който изпраща тази информация до сървъра (Фиг. 3.6.1.3), който връща като отговор обект, съдържащ две полета - `login` (което отразява дали данните, които потребителя е въвел могат да бъдат асоциирани с регистриран потребител на приложението) и поле `user` (което отразява данните на потребителя, ако той е бил разпознат, в противен случай то има стойност `null`). Ако потребителя е разпознат диалога се затваря с обект съдържащ две причини - `login` и `success`, който се обработва от услугата `showDialogService`. Ако потребителя не е разпознат, диалога се затваря с причина `fail` и на екрана на потребителя се отваря диалог да извести за евентуална грешка на въвежданите от потребителя данни.

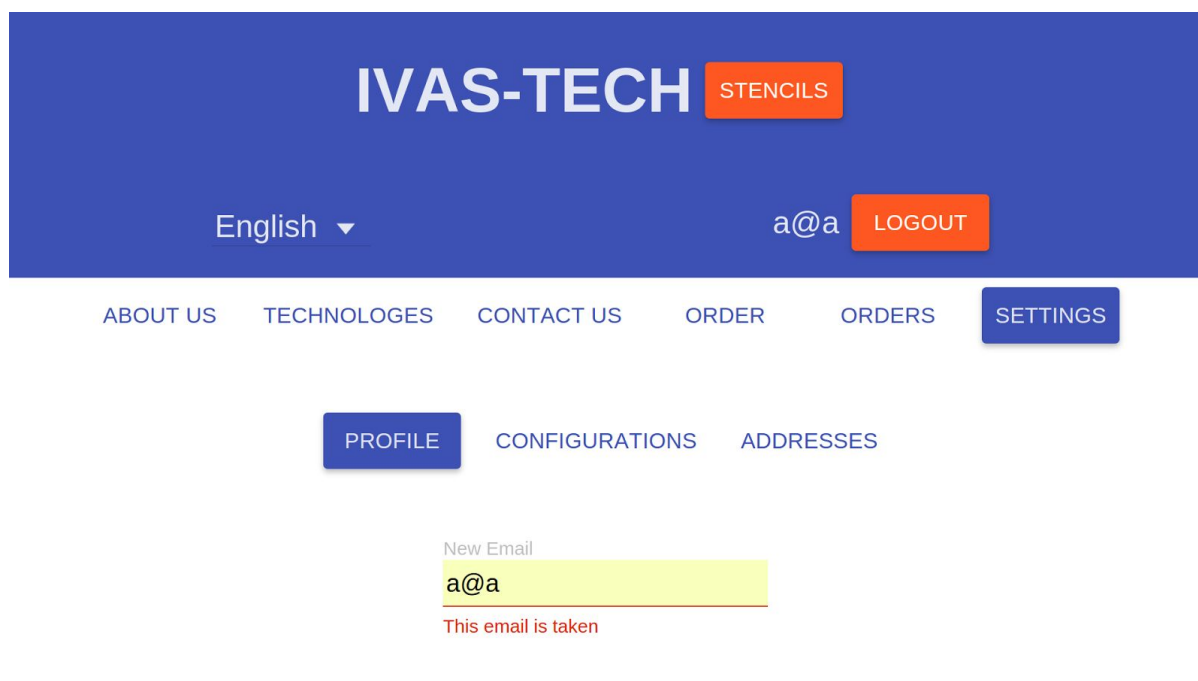
```
ctrl.login = (valid) ->
  if valid then RESTHelperService.login.login user: ctrl.user, (res)
->
  login = -> ctrl.hide login: null, success: user: res.user,
session: ctrl.session
  if res.login then login() else ctrl.hide "fail"
```

*(Фиг. 3.6.1.3) Метода login на контролер loginController*

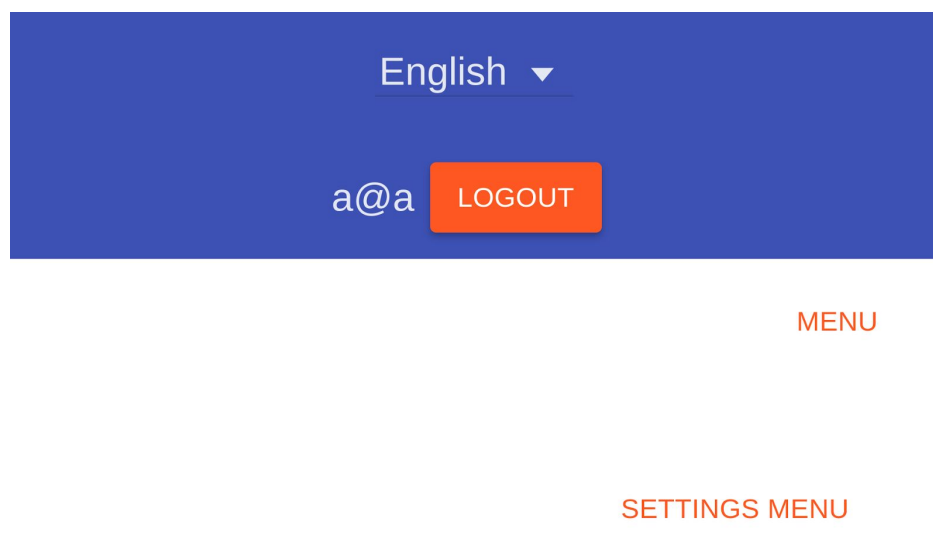
### 3.7 Навигация между виртуалните страници на приложението чрез директивата "ivstStateSwitcher"

За промяната на състоянията, в които може да се намира разработваното Интернет приложението бе създадена директивата "ivstStateSwitcher", която като свой изглед използва "responsive" (реагиращо) меню (Фиг. 3.7.1), (Фиг. 3.7.2), и (Фиг. 3.7.3) чрез което лесно може да се преминава от едно състояние в друго. Целта на

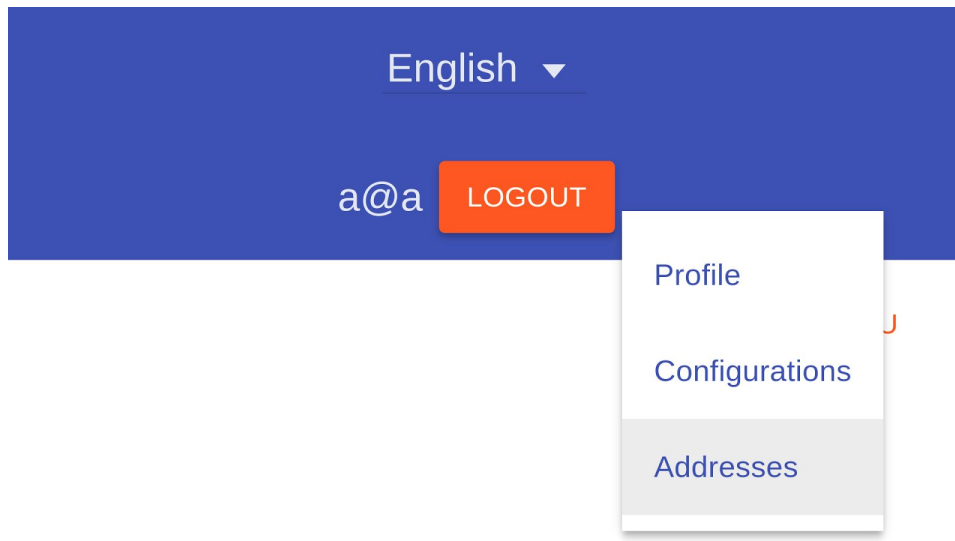
създадената директива е да предостави преизползваем код, който служи за смяната на състоянието, в което се намира приложението. Нейният контролер динамично създава бутоните от менюто за навигация между състоянията, като използва децата на подадено име на състояние. Кой точно деца могат да се използват за създаването на бутони може да бъде контролирано, чрез подаването на масив, на който елементи оказват състоянията, за които навигационен бутон не трябва да бъде генериран. Масива се подава като стойност на атрибута “remove” на елемента на директивата. Освен пълния контрол над генерираните бутони съществува атрибут “override”, който може да бъде използван за промяна на състоянието, на което приложението трябва да отиде при натискането на деден бутон, тоест чрез него може вместо да се премине към конкретно състояние да се премине към дете на това състояние. За целта се подава обект, чийто полета отговарят на състоянията, които искат да се променят, а стойностите на тези полета отговарят на имената на децата на състоянията, на които трябва да се отиде след преминаването към състоянието зададено като ключово поле. В началото на инициализиращата функция “init”, се задават стойности по подразбиране, ако не е използван атрибута remove, то му се присвоява стойност на празен масив, а ако override, не е използван му се присвоява стойност на празен обект. Услугата “statesForStateService” е отговорна за генерирането на бутони от изгледа на директивата. Локалната функция override, едновременно служи за правилното отбелязване на текущото състояние, чрез обозначение на бутона, който съответства на текущата страница и за прехвърлянето от състояние в състояние, ако това е оказано, чрез едноименния атрибут на директивата. Първото извикване на тази функция е непосредствено след нейната дефиниция с цел правилното отбелязване на състоянието, в което се намира приложението, когато навигацията до конкретно състояние е настъпила след въвеждането на желан от потребителя адрес на състояние, на което той е искал да отиде при зареждането на приложението (Фиг. 3.7.4).



(Фиг. 3.7.1) Изгледа на менютата създавани от директивата *ivstStateSwitcher*, гледани от настолен компютър



(Фиг. 3.7.2) Изгледа на менютата от (Фиг. 3.7.1), гледан от мобилно устройство



(Фиг. 3.7.3) Отваряне на менюто "SETTINGS MENU" от (Фиг. 3.7.2)

```
init = ->
  if not ctrl.override? then ctrl.override = {}
  if not ctrl.remove? then ctrl.remove = []

  ctrl.states = statesForStateService ctrl.state, ctrl.remove

  override = (event, toState, toParams, fromState, fromParams) ->
    current = toState.name
    name = current.split separator
    check = name[name.length - 1]
    change = ctrl.override[check]
    ctrl.selected = (Boolean current.match state + "(?!s)" for state
in ctrl.states)
    if change? then $state.go [ctrl.state, check, change].join
separator

  override null, $state.current
```

```
stop = $scope.$on "$stateChangeSuccess", override
```

*(Фиг. 3.7.4) Функцията init от контролера stateSwitcherController*

Част от кода на шаблона на базовата страница “homeView” е показана на (Фиг. 3.7.5), в която е използвана директивата ivstStateSwitcher и описва горното навигационно меню от (Фиг. 3.7.1).

```
ivst-state-switcher(  
  ng-hide="homeCtrl.admin || homeCtrl.notFound"  
  state="home"  
  menu="menu"  
  override="{order: 'configuration', settings: 'profile'}"  
  remove=["admin', 'notfound']"  
)
```

*(Фиг. 3.7.5) Част от кода на шаблона homeView, в която е използвана директивата ivstStateSwitcher*

### 3.8 Интернационализация на Интернет приложението

За изпълнението на едно от изискванията към приложението - двуезичността му, с възможност за лесно добавяне на други езици като превод на приложението бе избрана технологията “angular translate”, която е налична като външен, допълнителен модул към Angular, регистриран под името “pascalprecht.translate” [58]. Модула предоставя изключително лесната интеграция на “i18n” към приложението. Като за целта само трябва да бъдат създадени отделни обекти, за всеки от желаните езици, за които приложението трябва да има поддръжка. Имената на ключовите полета на тези обекти, са символните низове, които разработчика на приложението ползва, а техните стойности са това, което потребителя на приложението вижда. За трансформацията на

тези полета, към стойностите, които потребителя трябва да вижда са налични различни средства: филтър, услуга и директива. Като се препоръчва да се ползва предоставената директива “translate”. Непрепоръчително е да се използва филтъра translate, тъй като филтъра добавя прекалено много функции, които да се извикват при промяна на контекста на различните елементи от изгледа на приложението. Препоръчително е избягването на ползването на услугата за превод “\$translate”, защото по този начин не се разделя логиката, на приложението от неговия изглед, което е в разрез с идеята да се ползват различни елементи, за различните слоеве на приложението. За промяната на езика е необходимо извикването на метода “use” на услугата \$translate с уникалния идентификатор, с който обектите описващи превод за даден език са били регистрирани. Регистрацията на обекти става посредством извикването на метода “translations” на доставчика “\$translateProvider” и подаването му като първи аргумент уникален идентификатор на езика и като втори обекта, носещ информация за превод на съответния език.

### **3.9 Интернет приложението, след като премине през процеса "Bootstrap"**

След като приложението премине през процеса Bootstrap се извиква изпълняващата функция на модула “homeModule” на приложението, която ограничава достъпа на потребителите, които не са разпознати от системата на сървъра. Функцията подава функция, която да бъде извикана, когато приложението започне да променя своето състояние (Фиг. 3.9.1). Независимо от въведения адрес в адресното поле на брауъра от потребителя, приложението винаги променя своето състояние след като се зареди, такаче винаги да се намира в едно от дефинираните състояния. Ако въведения адрес е само адреса на Интернет страницата на приложението, за който не е дефинирана виртуална страница и бива разпознат като “/”, то състоянието се променя в “home.about”. Това става чрез метода “otherwise” на доставчика “\$urlRouterProvider” използван в конфигурационната функция на homeModule, чрез който се оказва, че ако не бъде намерена виртуална страница за даден адрес, да се премине към адреса “/about”, за който съществува виртуална страница, описана в състоянието home.about.

Функцията, която се извиква, когато приложението започне да променя своето състояние проверява дали за адреса, на който потребителя иска да отиде е нужно потребителя да бъде разпознат, като ако е нужна аутентикация за проверяваната страница се вика метода “preventDefault” на аргумента event на функцията, като по този начин се спира пълното преминаване към състоянието, което отговаря на дадения адрес. След спирането смяна на състоянието се показва изглед, служещ да индикира, че приложението се зарежда. Тоест на контролера на текущия изглед може да е нужна информация от сървъра на приложението. Показването на изгледа за зареждане става чрез създаването на диалог. Той е единствения, за чието създаване не се използва услугата showDialogService, вместо това директно се извиква услугата \$mdDialog. Това е и единствения диалог, който не се затваря от потребителя, вместо това той се затваря, когато настъпи събитие "cancel-loading", което настъпва следкато услугата “stopLoadingService” бъде извикана от контролера на текущия изглед. Следва проверка за аутентикация на потребителя. Ако той е разпознат в системата се проверява дали той е администратор, ако е, приложението сменя своето състояние, преминавайки към администраторския изглед. Ако потребителя не е администратор, но все пак е разпознат от системата, съществува създадена за него сесия на сървъра, приложението започва да следи за промяна в състоянието на някоя от поръчките на потребителя и се вдига флаг с цел презареждане на състоянието, на което потребителя е искал да отиде, ако за него е нужна аутентикация. Ако потребителя не е разпознат, се проверява дали е искал да отиде на страница, за която е нужна аутентикация и ако е, се отваря диалог за вход в системата, който когато потребителя успешно влезе в системата се презарежда страницата, на която е отивал, ако не е администратор, защото в такъв случай той вече е пренасочен към администраторската страница. Ако не успее да влезе в системата приложението отива към състоянието по подразбиране, което е страницата “За нас”. Добавя се функция, която да следи за аутентикация на потребителя, която е нужна в случаите, когато потребителя излезе от системата и отново влезе в нея, без да затваря прозореца на своя браузър. Добавя се и функция, която да следи за започване на смяна на състоянията с цел ако потребителя не е разпознат от системата, но се опита да премине към страница, за която е нужно той да бъде, отново се спира промяната на състоянието и му се показва диалог за вход, като при успешно влизане в системата се презарежда страницата, на която е отивал. Следва проверка на флага индикиращ



презареждане на първоначалното състояние, на което потребителя е искал да отиде и ако той е дигнат, съответното състояние се презарежда. Функцията завършва с добавянето на функция, която да бъде извикана, когато приложението бъде затворено, тоест най-главния контекст бъде унищожен, която спира всички функции, които се извикват при настъпването на дадено събитие.

```
stop = $rootScope.$on "$stateChangeStart", (event, going) ->
  notRestricted = ["/about", "/technologies", "/contacts",
"/notfound"]
  admin = no
  goTo = no

  force = (state) -> $state.go state.name, {}, reload: yes

  login = (state, close) -> loginService {}, login: (-> $timeout (->
if not admin then force state), 1), close: close, cancel: close

tryBecomeAdmin = ->
  admin = authenticationService.isAdmin()
  if admin then transitionService.toAdmin()
  else notificationService.listenForNotification()

stop()

if going.url not in notRestricted
  event.preventDefault()
  $mdDialog.show templateUrl: "loadingView", fullscreen: yes,
hasBackdrop: no, escapeToClose: no
  stopLoading = $rootScope.$on "cancel-loading", ->
  $mdDialog.hide()
```

```

    stopLoading()

authenticationService.authenticate().then ->
    if authenticationService.isAuthenticated()
        tryBecomeAdmin()
        if not admin then goTo = yes
    else if event.defaultPrevented then login going,
transitionService.toHome

stopAuth = $rootScope.$on "authentication", tryBecomeAdmin

stopRestriction = $rootScope.$on "$stateChangeStart", (evnt,
toState, toParams, fromState, fromParams) ->

    if not authenticationService.isAuthenticated() and toState.url
not in notRestricted
        evnt.preventDefault()
        login toState

    if goTo then force going

```

(Фиг. 3.9.1) Функцията, която се извиква при започване промяна на смяна на състоянието на приложението

### 3.10 Разпознаване на потребители на системата (Аутентикация)

Разпознаването на потребители се извършва от сървъра на приложението. Интернет приложението при стартирането се прави заявка към сървъра, за да провери дали интернет адреса на потребителя е разпознат от него. Информацията за разпознаваните адреси се пази в отделна колекция на базата данни.

### 3.10.1 Разпознаване на потребители от Интернет приложението

Разпознаването на потребители на системата от Интернет приложението става чрез метода “isAuthenticated”, който връща стойността на ключовото поле “auth” на вътрешния за услугата обект state, служещо като флаг, оказващ дали потребителя е разпознат. За аутентикирането на потребител се използва “authenticate” на услугата “authenticationService” (Фиг. 3.10.1.1). Метода приема само един аргумент, който представлява обект, описващ аутентикация на потребителя. Когато метода е извикан без аргумент, той прави заявка към сървъра на приложението, която да провери дали той е разпознат. Ако потребителя е разпознат се извиква вътрешната за метода функция “broadcast” с отговара на заявката, която се използва с цел преизползване на кода за разпознаване на потребителя от Интернет приложението. Когато метода authenticate е извикан без аргумент, той връща обект от тип Обещание, който индикира кога проверката е завършила. Когато към метода се подаде аргумент, това значи, че той успешно е влязал в системата чрез диалога за вход в системата и директно бива извикана функцията broadcast със същия обект. Функцията broadcast променя вътрешното състояние на услугата, с цел запомнянето, че потребителя е разпознат, какви са неговите данни, дали да се пази сесия за него (дали да бъде изпратена заявка за прекратяване на сесията или не) и дали е администратор,. Функцията променя езика на приложението, такаче той да бъде избрания от потребителя и съобщава за настъпването на събитие, че потребителя е бил разпознат.

```
authenticate: (authentication) ->
  broadcast = (auth) ->
    state = auth: yes, user: auth.user, session: auth.session ? yes,
admin: auth.user.admin
  $translate.use auth.user.language
  $rootScope.$broadcast "authentication"

if authentication? then broadcast authentication
```

```

else $q (resolve, reject) ->
  RESTHelperService.login.logged (res) ->
    if res.login then broadcast res
    resolve()

```

*(Фиг. 3.10.1.1) Метода authenticate на услугата authenticationService*

## 3.10.2 Разпознаване на потребители от сървъра на приложението

За разпознаването на потребители от сървъра на приложението се използват две функции, част от имплементирания модел на сесии и колекция от базата данни, която да съхранява информация за разпознатите потребители

### 3.10.2.1 Създаване и управление на сесии

Избрания и имплементиран архитектурен модел на сървъра на приложението е такъв, че сървъра сам знае данните, с който потребителя работи при всяка заявка. Това се постига чрез прикачането на запис съответстващ на даден потребител към обекта на всяка заявка, тоест “\_id” на потребителя винаги е достъпно. От друга страна всеки модел съдържа специално поле “user” отговарящо на \_id на запис на потребител към всеки документ от всяка колекция (с изключение на три колекции: потребителска, колекция на посещенията и колекция на известията), оказващо на кой потребител принадлежи дадения запис и чрез правенето на заявка, в която участва стойността на \_id на потребителя към базата данни. Винаги може да се работи с конкретните данни на конкретния потребител.

Сесиите се използват за разпознаване на устройството, използвано от потребителя на приложението, тоест на интернет адреса, от който идва всяка заявка и по този начин на самия потребител. Те позволяват потребителя едновременно да използва приложението от различни устройства и да работи с едни и същи данни на тях.

Използваната от сървъра сесия неимплементира типичен модел, при който се пазят стойности, принадлежащи на даден потребител. Вместо това се пази информация

за самия потребител, използвана за извличането на неговите данни. Имплементираната сесия е разделена на три отделни функции посредници (Фиг. 3.10.2.1.1). Функция “remove” се използва за прекратяване на сесия за даден потребител, като тя изтрива документ от сесийната колекция, отговарящ на текущата сесия и се извиква, когато се прихване заявка за изход от системата. Функция “get” се извиква преди всяка заявка, която служи за комуникация между сървъра и Интернет приложението. Ако съществува сесия за даден потребител (съществува документ в сесийната колекция, който съдържа интернет адреса на потребителя), към обекта на заявка бива прикачен обект, отговарящ на дадения потребител, като се прави заявка към базата данни за извличането на тази информация. Като името на ключовото поле чрез, което се добавя информация на потребителя може да бъде конфигурирано и има стойност по подразбиране “user” с цел избягване на конфликти с външни функции посредници, ако възникнат такива. Последната функция “set” се използва за създаването на нова сесия, като тя свързва интернет адрес на дадено устройство на потребителя и самия потребител, чрез създаването на нов запис в сесийната колекция. Тя се извиква след успешна проверка на това дали потребителя съществува в системата (дали изпращаните данни отговарят на регистриран потребител). Първото нещо, което функцията прави е да провери дали съществува потребител, което се отбелязва чрез добавяне на поле към заявката от предходната функция, което има за стойност “null”, ако не е намерен потребител. Ако полето не е null се създава нов запис в сесийната колекция. Ако не възникне грешка при изпълнението на функцията set, следва извикването на функция прихващач на заявката, която да върне отговор на направената заявка. Изпращаният отговор оказва, дали потребителя е бил разпознат и неговата информация, ако е (Фиг. 3.10.2.1.2).

```
remove: (req, res, next) ->
  (sessionModel.remove _id: req[field]._id)
    .exec().then (-> query res), next

get: (req, res, next) ->
  ip = requestIp.getClientIp req
```

```

(sessionModel.findOne ip: ip)
  .populate "user"
  .exec()
  .then (doc) ->
    req[field] = doc
    if not doc? then req.userIP = ip
    next()
  .catch next

set: (req, res, next) ->
  if not req.user? then next()
  else
    ip = requestIp.getClientIp req
    user = req[field]
    (sessionModel.create user: user._id, ip: ip)
      .then (doc) ->
        req[field] = user: user, ip: ip, _id: doc._id
        next()
      .catch next

```

*(Фиг. 3.10.2.1.1) Методите за работа със сесии*

```

post: [
  (req, res, next) ->
    (userModel.findOne req.body.user).exec()
      .then (doc) ->
        req.user = doc
        next()
      .catch next

```

```

session.set

(req, res) ->
  login = req.user?
  query res, login: login, user: if login then req.user.user else
req.user
]

```

*(Фиг. 3.10.2.1.2) Описание на заявката за вход в системата, която създава нова сесия за разпознат потребител*

### 3.10.2.2 Заявка за проверка на разпознаване от системата

Заявката за проверка на разпознаване от системата се изпълнява при получаването на GET на адреса за вход в системата. Освен, че служи за разпознаване от ситемата, тя се използва и за запис на посещение на Интернет страницата на приложението (Фиг. 3.10.2.2.1). Това е първата заявка, която Интернет приложението прави след своето зареждане и бе преценено, че е по-подходящо записването на посещението на приложението да се извършва следкато проверката за разпознаване от ситемата е направена и нейният резултат е известен, вместо посещението да се записва при приемането на заявката, служеща за изпращане на страницата на приложението.

Функцията, която проверя за разпознаване в системата, проверява дали съществува ключово поле, отговарящо на сесията на потребителя, към обекта на заявката и присвоява резултата на поле индикиращо разпознаване, чрез добавянето на ново поле към обекта на заявката, който представлява информацията, която да бъде върната като отговор на приета заявка. Ако потребителя е рапознат, неговата информация се извлича от базата данни чрез заявка и получената информация се добавя към полето, пазещо информацията, която представлява отговора на заявката.

```

get: [
  (req, res, next) ->

```

```

req.send = login: req.user?
if req.send.login
  (userModel.findById req.user.user._id).exec()
    .then (doc) ->
      req.send.user = doc
      next()
    .catch next
else next()

(req, res, next) ->
  find = date: date.format(), ip: if req.userIP? then req.userIP
else req.user.ip
  (visitModel.update find, user: req.send.login, {upsert: yes})
    .exec().then (-> query res, req.send), next
]

```

(Фиг. 3.10.2.2.1) Част от обектът описващ рутера, който приема заявка за разпознаване от системата

Функцията, която записва посещение на страницата на интернет приложението, обновява запис от колекцията на посещенията, който отговаря на Интернет адреса на потребителя и дата на посещение. Ако такъв запис не съществува, той се създава благодарение на ключовото поле “upsert” на обекта, подаван като аргумент, носещ допълнителни оказания към правенето на заявката към базата данни. Използва се обновяване на запис, вместо директно създаване на такъв, защото посещенията на страницата на се свеждат до посещения от уникални интернет адреси за всеки ден, което се постига чрез форамтирането на моментната дата. Форматирането е с цел лимитиране на информацията от дата до ден, чрез метода “format” на експортирания от файла “/server/share/dateHelper” обект, връщан от функцията “\$get”, който се връща и от доставчика на услугата “dateService”, благодарение на използваната технология browserify. Услугата dateService бе създадена с цел визуализация на дати, чрез тяхното



форматиране в символни низове, които да бъдат показвани на потребителя на приложението, но бе взето решение да бъде преизползвана от сървъра на приложението за решаването на проблема: “как да се запазят само посещения от уникални Интернет адреси за всеки ден”.

### **3.11 Вход и изход в системата**

Преди да може даден потребител да влезе в системата, той трябва да се регистрира.

#### **3.11.1 Регистрация на нови потребители**

Регистрацията на нов потребител от Интернет приложението става единствено и само чрез натискането на бутона “РЕГИСТРАЦИЯ”, който е част от изгледа на директивата “ivstUser”. При натискане на бутона се извиква метода “register” на контролера на директивата, който използва услугата “registerService”, за да покаже диалога за регистрация на потребителя. Диалога за регистрация, както и останалите диалози, които показват грешка за не въведена информация, която е задължителна или неправилно въведена информация използват филтъра “isEmpty”, за да окажат на директивата “mdInputContainer”, използвана за “опакване” на полета за въвеждане на информация, предоставена от Angular-Material, че диалога се намира в състояние на грешка. Ръчното оказване на това състояние е необходимо, защото избрания стил на фрэймуърк има вградена поддръжка за използването на директивите от модула “angular-messages” и по този начин задължава тяхното използване, но съществува конфликт между директивата “ngMessage” и директивата “translate” (използвана за интернационализация на приложението), пречещ на правилното показването на известието за грешка на избрания от потребителя език. Използвания филтър в комбинация с поставянето на двете директиви на различни HTML елементи не решава напълно конфликта, но значително увеличава случаите, в които известието бива правилно показано.

За да може потребителя да се регистрира, той трябва първо правилно да попълни полета с информация от диалога за регистрация. Полетата са три и тяхното попълване е задължително. Това се постига като към всяко “input” поле се добави атрибут “required”, който оказва, че са задължителни. Под всяко input поле се добавя HTML елемент с атрибут, директивата “ngMessages”, чиято стойност представлява пътя от контекста до грешката за съответното поле за въвеждане на данни. В Angular формите от изгледите представляват инстанции от “FormController”, а полета за въвеждане на данни имащи атрибут, директивата “ngModel” представляват инстанции от тип “ngModelController” и когато формите или съответните полета съдържат атрибут “name” те биват инжектирани в контекста на изгледа, на който принадлежат под името на стойността, подадена на атрибута “name”, което позволява те да бъдат използвани като останалите модели свързани, чрез директивата ngModel. Това позволява обекта описващ настъпила за даден модел грешка (полето “\$error”) да бъде подаден на директивата ngMessages и чрез подаването на име на ключово поле на директивата ngMessage да се извеждат съобщения за конкретна грешка. Полето за въвеждане на емейл адрес е единственото в разработваното Интернет приложение, за което е създадена специална директива за валидация. За останалата валидация на полета се използват вградени във фреймуърка директиви. Използваната валидация в диалога регистър проверява за дължината на въведената парола и стойността ѝ да съвпада на полето за потвърждение на парола.

Специално създадената с цел валидация директива “ivstEmailTaken” проверява дали въведения от потребителя емейл адрес е използван от друг потребител, като прави заявка към сървъра, което налага да се използва обекта “\$asyncValidators” на контролера на модела, защото всяка заявка се изпълнява асинхронно, от кода направил заявката. Съответно валидатора връща обект Обещание, който отразява състоянието на направената валидация. Като ключовото поле “email-taken”, под което се добавя новия валидатор ще отговаря на полето от обекта \$error, което ще отразява дали емейла е използван и стойността му ще бъде true, когато емейла е зает (Обещанието е преминало в състояние resolved) и false, когато емейла е свободен (Обещанието е преминало в състояние rejected).

След като потребителя правилно е попълнил информацията за своята регистрация при натискането на бутона РЕГИСТРАЦИЯ на диалога се изпраща заявка,

който регистрира потребителя, като към данните, които той е въвел се добавя езика избран в момента на натискане на бутона.

### **3.11.2 Вход в системата**

Влизането в системата става, чрез диалога за вход, който може да бъде отворен при натискане на бутона “Вход” на приложението или това става автоматично, когато потребителя не е влязал в системата, а се опитва да отиде на състояние, за което се изисква да бъде. При успешно влизане в системата услугата authenticationService съобщава за настъпило събитие “authentication”, при което контролера на услугата ivstUser скрива бутоните за вход и регистрация и на тяхно място показва адреса на електронната поща на потребителя и бутон за изход от системата. При настъпването на събитие известяващо вход в системата контролера проверява и дали потребителя е искал да остане в системата при затваряне на приложението, което става чрез извикването на метода “isSession” на услугата за аутентикация, който при влизане чрез диалога връща стойност оказваща дали полето за маркиране до надписа “Запомни ме” е било попълнено (по подразбиране, когато се отваря диалога то е автоматично попълнено). Ако е било празно при затваряне на приложението се извиква метода “unauthenticate” на същата услуга, който прави заявка към сървър на приложението сесията на потребителя да бъде “унищожена”. По подразбиране се създава нова сесия когато потребителя влезе в системата и тя се премахва единствено след направата на заявка от приложението към сървър.

### **3.11.3 Изход от системата**

Когато потребителя е влязъл в системата той има възможност ръчно да излезе от нея, чрез натискането на червения бутон “ИЗХОД”, който извиква метода logout на контролера “userController”, който извиква метода unauthenticate на услугата authenticationService, подавайки му функция, която да бъде извикана когато приключи процеса на излизане. Подаваната функция служи за връщането на изгледа на

директивата `ivstUser` към нейния първоначален облик и пренасочването на потребителя към началната страница на приложението.

### 3.12 Преизползване на MVC

Създадените механизми за преизползването на модели, изгледи и котролери са взаимно обвързани и в повечето случаи резултата от преизползването на един елемент оказва влияние в преизползването на друг (например директивата `"ivstBase"` използва комбинация от трите вида преизползване) и дори някои от механизмите за преизползване на един елемент използват механизъм за преизползване на друг (например директивата `"ivstIncludeDirective"` използва услугата `"scopeControllerService"`).

#### 3.12.1 Преизползване на модели

За преизползването на модели бяха създадени услугите `"scopeControllerService"` и услугата създавана от `"progressServiceProvider"`.

##### 3.12.1.1 Преизползване на модели чрез услугата `"scopeControllerService"`

За преизползването на модели в изгледи, независимо на кой контролер принадлежат, бе създадена услугата `"scopeControllerService"`, която приема като аргумент контекст и се опитва да намери контролер инжектиран в контекста под име, стойността на ключовото поле `"controller"`. Ако намери такъв го присвоява като стойност на полето `"scopeCtrl"`, като по този начин позволява моделите от него да бъдат преизползвани в изгледа, отговарящ на подавания към услугата контекст. Стойностите на моделите могат да бъдат променяни, защото вместо да бъде присвоена стойността на контролера се присвоява препратка към нея (в JavaScript резултата от присвояването на стойността на една променлива на друга е самата стойност когато

типа и е бил един от примитивните, във всички останали случай резултата е препратка към самата променлива). Името на полето, под което са налични преизползваните модели носи името `scopeCtrl`, защото едни и същи модели могат да идват от контролери с различни имена, което доведе до избор на унифицирано име, под което те са налични в изгледа. Името на контролера, от който се преизползват модели идва от стойността на полето `controller` на подавания контекст, което в повечето случаи е директно дефинирано като вход на директивата използваща услугата.

### **3.12.1.2 Преизползване на модели чрез услугата "progressService"**

Услугата "progressService" (Фиг. 3.12.1.2.1) едновременно изпълнява две предназначения, които са взаимно свързани. Тя позволява преизползването на модели от контекста на деца на едно състояние в неговия контекст. Другото предназначение на услугата е навигацията напред и назад в състояние на приложението между децата на едно състояние и запамятаване на състоянието, в което дадено дете се намира, преди промяна на състоянието на приложението и възобновяването му при връщане на съответното състояние. Услугата се създава от нейният доставчик, чрез който се конфигурира базовото (родителското) състояние. Първото нещо, което прави услугата при създаването си е да вземе информация за конфигурационния обект на родителското състояние и да намери всички деца на конфигурираното състояние, използвайки услугата "statesForStateService". Резултатът от инжектирането на услугата е функция приемаща четири аргумента: контекст, име на контролер, имена на полета, които да не се преизползват и имена на полета, които в момента на създаването на съответния контролер не са налични, но при промяна на състоянието на Интернет приложението ще бъдат. Подаването на аргументи към функцията е незадължително и когато такива липсват, услугата предоставя възможност само за смяна напред и назад на състоянието на приложението. Смяната на състоянията става посредством връщания от функцията обект с два метода за смяна в двете посоки. При подаване на стойности поне на първите два аргумента, услугата добавя функция, която се извиква, когато контролера бъде създаден в контекста под подаденото като втори аргумент име. Функцията, която се извиква при създаването на контролера създава масив съдържащ имената на всички

ключови полета от контекста на контролера, които не са функции и не се съдържа в масива с имена, които да не бъдат преизползвани. Не се преизползват функции, защото те не са модели, не се променят при всяко създаване на контролера, при смяна на състоянията и може да не са налични за част от приложението, което използва преизползване на модели. Пример за последното от изброените е директивата "ivstConfigurationInfo", която преизползва модел описващ конфигурация за стенсил и бива използвана на две места. При едното модела, който преизползва директивата, използва за преизползване на друг модел разглеждана услуга, а в другия не, при което само в единия случай тя би имала методите за промяна на изгледа, а в другия не. Следва проверка дали за създавания контролер не се съхранява информация за предишното му състояние. Ако се пази такава информация, на моделите, за които се съхранява информация се присвоява пазената стойност и се съобщава за това събитие. Ако към функцията връщана при инжектирането на услугата е бил подаден контекст при използването на един от двата метода за смяна на състоянието на Интернет приложението ще се запази информация за състоянието на контролера преди да се извърши смяната на състояния на приложението.

```
service = ($state, statesForStateService) ->
  separator = "."
  parent = $state.get parentState
  move = statesForStateService parentState

(scope, current, exclude = [], awaiting = []) ->
  state = $state.current.name.replace parentState + separator, ""
  currentScope = parentScope = properties = null

  if scope? then restored = scope.$watch current, (controller) ->
    currentScope = scope[current]
    parentScope = scope.$parent[(parent.controller.split " as
")][1]]
```

```

    properties = (Object.keys currentScope).filter (property) ->
      typeof currentScope[property] isnt "function" and property
not in exclude
    properties.push deferred for deferred in awaiting

    if parentScope[properties[0]]?
      currentScope[property] = parentScope[property] for property
in properties
      scope.$emit "update-view"
    restored()

    progress = (forward) ->
      if scope? then parentScope[property] = currentScope[property]
for property in properties
      $state.go parentState + separator + move[(move.indexOf state)
+ if forward then 1 else -1]

    next: -> progress yes

    back: -> progress no

```

(Фиг. 3.12.1.2.1) Код на услугата progressService

### 3.12.2 Преизползване на изгледи

За преизползването на изгледи се използват няколко механизма. Основният е чрез създаването на директиви, като по този начин се преизползва цели MVC модел. Друг използван механизъм в разработваната дипломна работа е създадената директивата “ivstIncludeDirective”.

### 3.12.2.1 Преизползване на изгледи чрез директивата “ivstIncludeDirective”

Директивата “ivstIncludeDirective” бе създадена по подобие на включената в Angular директива “ngInclude”, която добавя към шаблона, в който е използвана друг шаблон. Директивата ngInclude позволява шаблон да бъде добавен към друг шаблон чрез използването на интернет адрес, подкойто е наличен добавяния шаблон. Създадената директива позволява вместо чрез Интернет адрес към който да бъде направена заявка за шаблона, той директно да бъде изваден от “кеш паметта” на Angular за шаблони (“\$templateCache”) или стойността на модел да бъде използвана за шаблон. Когато към директивата е добавен атрибут “template” със стойност “true” свързващата й функция (Фиг. 3.12.2.1.1) изважда шаблона от паметта чрез извикването на метода “get” на услугата \$templateCache и подаването му стойността на атрибута “include” на директивата. Ако към директивата не е добавен атрибут template или стойността му е различна от “true” свързващата функция на директивата добавя функция, която да бъде извиквана при промяна на стойността на атрибута include, който е модела използван за добавянето на шаблона. Ако чрез атрибута “controller” на директивата е оказано, че добавяния шаблон има нужда да преизползва модели или директно логика на контролер за промяна на изгледа на компилирания шаблон се извиква услугата scopeControllerService.

```
link: (scope, element, attrs) ->
  insertTemplate = (html) ->
    element.html html
    ($compile element.contents()) scope

if scope.controller? then scopeControllerService scope
```



```

if scope.include?
  if attrs.template is "true"
    insertTemplate $templateCache.get scope.include
  else scope.$watch "include", insertTemplate

```

(Фиг. 3.12.2.1.1) Свързващата функция на директивата *ivstInclude*

### 3.12.3 Преизползване на контролери

Частен случай на преизползването на изгледи чрез обособяването им в директиви е добавянето на динамичен контролер към тях, което позволява най-пълното преизползване на код от всички, като по този начин преизползването на MVC модела става по-пълно от използването на директивива със статично свързан към нея контролер. Използването на динамично свързване на контролера на дадена директива в разработваната дипломна работа е използвано само в комбинация с услугата “\$controller” при “наследяване” на контролери. По този начин се преизползват едновременно модели, изглед и контролер, което значително намлява количеството на повтарящ се код в разработваното Интернет приложение.

Услугата \$controller, позволява използването в Angular на характерния за Обектно ориентираното програмиране (ООП) модел - наследяване. Като чрез нея може да се постигне наследяване на контролери и по този начин един контролер директно може да използва логика и модели дефинирани в друг контролер. Наследяването на контролери в Angular не следва типичния за ООП модел на наследяване, защото услугата \$controller се използва само за създаване на нова инстанция от наследявания контролер, към която да се добави ново поведение и да бъде върната от конструктивната функция на наследявания контролер. Услугата приема два аргумента: първият е функция служеща за създаване на контролер или символен низ обозначаващ регистриран контролер, вторият аргумент е обект представляващ аргументите, които да бъдат подадени на първия аргумент при наследяването. Втория аргумент се използва за директното инжектиране на зависимости, нужни на конструктивната

функция на наследявания контролера и по този начин се съкратява процеса по инстанциране на наследявания контролер.

За обозначаване на динамично свързване на контролер към дадена директива се използва недокументираната инструкция към компилатора на Angular - ключовото поле "name". Стойността на полето name трябва да е символен низ, който отговаря на атрибута, който ще се използва за оказване името на контролера, който трябва да бъде свързан със съответната директива. Стойността на ключовото поле "controller" трябва да е символа "@", използван за оказване на интерпретация като символен низ към фреймуърка.

### **3.13 Имплементация на функционалността за създаване на поръчки**

За създаването на нова поръчка потребителя на Интернет приложението трябва да премине през всичките състояния, които са деца на състоянието "home.order": "configuration", "specific", "addresses", "price" и "finalize". Като за състоянията configuration и addresses се използва директива с динамично свързване на контролера, която позволява общият за двете състояния код да бъде преизползван.

#### **3.13.1 Преизползване на код за елиминнирането на повторения в кода на конфигурациите за стенсил и адреси за доставка**

В процеса на разработка на дипломната работа бе забелязано, че кода използван за създаване и използване на конфигурации за стенсили и адреси се повтаря. За отстраняването на повторения бе създадена директивата "ivstBase", която премахва повторенията в кода на Интернет приложението и функцията експортирана от файла "/server/routes/order/basicCRUDHandle.coffee", която елиминира повторенията в обекта използван за описание на рутера отговарящ за работа с данните на съответния вид конфигурация.

### 3.13.1.1 Премахване на повторения в сървъра на приложението

Функцията експортирана от файла “/server/routes/order/basicCRUDHandle.coffee” приема два аргумента - модел, описващ колекцията на конфигурацията и име на конфигурацията. Описвания рутер приема само четири вида заявки: get, post, patch и delete (Фиг. 3.13.1.1.1).

```
get: (req, res, next) ->
  (model.find user: req.user.user._id)
    .exec().then ((docs) -> query res, "#{name}List": docs), next

post: (req, res, next) ->
  obj = req.body[name]
  obj.user = req.user.user._id
  (model.create obj).then ((doc) -> query res, doc), next

patch: (req, res, next) ->
  update = req.body[name]
  id = update._id
  delete update._id
  (model.findByIdAndUpdate id, $set: update, {new: yes})
    .exec().then ((doc) -> query res, doc), next

delete: (req, res, next) ->
  (model.remove _id: req.params.id).exec().then (-> query res), next

params: delete: "id"
```

(Фиг. 3.12.1.1.1) Структурата на обектът, описващ произволния рутер

GET заявката извлича документите от подадената колекция и ги връща като отговор на получената заявка. POST заявката създава нов документ в колекцията от получените

данни, като прибавя към тях уникалния идентификационен номер. PATCH заявката обновява документ от колекцията, като изтрива неговия уникален идентификационен номер от данните за обновяване, защото в противен случай избрания драйвер за връзка с базата данни ще хвърли грешка, че може да съществува само един документ отговарящ на съответния идентификационен номер. Към заявката се подава обект с ключово поле "new" със стойност true, което казва, че Mongoose при успешно преминала заявка ще извика съответната функция за обратна връзка с обновения документ. DELETE заявката приема параметър - уникален идентификационен номер на документ, който да изтрие от колекцията.

### 3.13.1.2 Отстраняване на повторения в Интернет приложението

За отстраняване на повторения в Интернет приложението бе създадена директивата "ivstBase", която използва динамично свързване на контролера и предоставя базовата функционалност за създаване, използване, обновяване и премахване на конфигурация. Пълната функционалност за работа с конфигурации на директивата е достъпна когато атрибута "settings" на директивата има вярна булева стойност. Свързващата функция на директивата взема решение, коя функция да извика, за да подготви изгледа за потребителя на приложението (Фиг. 3.13.1.2.1).

```
templateUrl: "baseView"
restrict: "E"
scope: yes
controller: "@"
name: "inherit"
controllerAs: "baseCtrl"
bindToController: settings: "="
link: (scope, element, attrs, controller) ->
  controller[if controller.settings then "getObjects" else
"restore"]()
```

(Фиг. 3.13.1.2.1) Обектът, описващ директивата ivstBase

Функцията конструктор на контролера на директивата освен своите зависимости приема и четири аргумента, които позволяват тя да бъде “шаблонизирана” и по този начин кода ѝ да бъде преизползван. Цялата логика между изгледа и контролера се основава на пет ключови полета, чиито имена са шаблонизирани с цел избягване на конфликти, които иначе биха настъпили при смяна на състоянието на приложението чрез услугата `progressService`. Имената им се образуват като към аргумента “link” използван като име на ресурса, с който контролера работи се добави като символен низ, едно от следните пет: "Object", "List", "Index", "Action" и "Disabled" (ще бъдат използвани само стойностите на символните низове с цел улеснение).

Полето `Object` се използва като модел, съхраняващ данните за конфигурацията. Полето `List` е модела, в който се пазят всички конфигурации на потребителя (получени от сървъра и създадени от потребителя, докато той не смени състоянието или затвори приложението). Полето `Index` е модела, който отговаря на индекса на избраната конфигурация, ако потребителя я е избрал от менюто за избор или това ще бъде индекса на последния елемент от масива `List`, когато нова конфигурация бъде запазена. Полето `Action` съдържа символен низ отговарящ на моментното състояние на изгледа, което може да е едно от следните “create” (когато се създава нова конфигурация), “preview” (когато дадена конфигурация бъде избрана или запазена) и “edit” (когато потребителя избере да обнови информацията на избрана от него конфигурация). Полето `Disable` се използва като булева стойност индикираща дали състоянието се намира в режим `preview`. Стойностите на полето `Action` заедно със ключовото поле “settings” се използват за определяне коя част от интерфейса на изгледа потребителя вижда. Различните методи на контролера променят по различен начин петте описани полета, като по този начин се променя изгледа на директивата.

Част от методите на контролера се дефинират динамично в зависимост от стойността на полето `settings`, което оказва дали директивата се използва при създаване на поръчка или не. Това се налага отново с цел преизползване на код, защото директивата освен, при създаване на нова поръчка се използва и за управление на конфигурации от потребителя в частта за настройки на приложението. Функциалността на контролера, която е достъпна само при създаване на поръчка е изцяло обвързана около услугата `progressService` и прехвърлянето на модели с цел тяхното преизползване

до края на направата на нова поръчка. Съответно методите, които служат за управление на конфигурации са достъпни само когато полето settings има вярна булева стойност. Неговата стойност идва чрез едноименния атрибут на директивата.

Изгледа на директивата ivstBase се преизползва чрез шаблона й, който съдържа директивата ivstInclude, чрез която се добавя специфичния за двата вида конфигурации изглед. Контролера на директивата се използва като базов контролер, към който чрез наследяване се добавя нова функционалност, която да управлява добавения към шаблона изглед. Базовата функционалност, която се наследява включва методи за извличане на съществуващи конфигурации, избор на конфигурация, запазване на нова конфигурация и метод за валидация "isValid", проверяващ дали всички полета са попълнени (Фиг. 3.13.1.2.2). Метода isValid проверява дали стойността на всички стойности на масива "valid" (предоставян до контролерите наследяващи контролера "baseInterface" за помнене на информацията относно всички попълнени и непопълнени полета от специфичния) е вярна булева стойност и дали полето за име на конфигурация от изгледа на директивата е попълнено. Ако всички полета са попълнени се извиква функцията подадена на метода, която трябва да бъде извикана когато всички полета са попълнени, в противен случай се извикват функцията, която се извиква, когато има не попълнено поле и функцията, която показва диалог оказващ, че всички полета са задължителни.

```
ctrl.isValid = (event, resolve, reject) ->
  tryToCall = (fns) -> (if fn? then fn()) for fn in fns
  checkValid = -> ctrl.valid.length and (ctrl.valid.every (e) -> e
is yes)
  if checkValid() and ctrl[link + "Object"].name then tryToCall
[resolve]
  else tryToCall [reject, -> simpleDialogService event,
"required-fields"]
```

(Фиг. 3.13.1.2.2) Метода isValid на контролера baseInterface

### 3.13.2 Избиране на конфигурация за стенсил

Избирането на конфигурация за стенсил става или чрез избор между вече създадени конфигурации или чрез създаването на нова. Визуализацията на конфигурацията става посредством изглед на стенсил, който се променя при промяната на конфигурацията. За изгледа на стенсила се използва директивата “ivstStencilPreview”, а за изгледа на конфигурацията се използва директивата “ivstConfigurationInfo”, като двете конфигурации използват общи модели, принадлежащи на контролера "configurationInterface", благодарение на услугата scopeControllerService. Чрез едновременно достъп на двете директиви до общите модели се постига синхронизация между двата изгледа. И двете директиви нямат собствени контролери, но докато директивата ivstStencilPreview използва само модели на контролера configurationInterface, директивата ivstConfigurationInfo се възползва и от препратката към контролера, който се явява общ за двете директиви и използва неговите методи за промяна на моделите му.

#### 3.13.2.1 Синхронизация между директивите ivstConfigurationInfo и ivstStencilPreview

Директивата ivstConfigurationInfo има достъп до методите на контролера configurationInterface, но този контролер е достъпен единствено и само когато съществува негова инстанция в контекста на директивата, което се случва само в две състояния на приложението. За да се запази синхронизацията между двете директиви независимо от състоянието, в което се намира приложението (директивата ivstStencilPreview се използва в изгледите на други състояния на приложението), бе взето решение директивата използвана за визуализация на конфигурация да използва само и единствено чужди за нея модели, които винаги да бъдат достъпни. Моделите, които директивата ivstStencilPreview използва представляват CSS класове, към които се пази препратка през целия процес на правене на нова поръчка, а когато дадена поръчка бъде направена стойностите на моделите, отговарящи на конфигурацията за съответната поръчка биват запазени на сървър на приложението, заедно с останалата

информация на поръчката. Чрез запазването на моделите на сървъра на приложението, те винаги могат да бъдат извлечени и по този начин изгледа на поръчката винаги остава синхронизиран с информацията от използваната в поръчката конфигурация за стенсил независимо от състоянието, в което се намира приложението.

#### **3.13.2.1.1 Директивата “ivstStencilPreview”**

Директивата `ivstStencilPreview` се използва за визуализация на стенсил. При избора на конфигурация директивата се използва за визуализация на конфигурацията, във всички останали случаи тя се използва за визуализация на поръчки. Директивата използва три атрибута за входни данни: “text”, “view” и “controller”. Атрибутът `controller` се използва за оказване на името, под което моделите, които използва директивата ще бъдат налични в родителския контекст на директивата, чрез услугата `scopeControllerService` се добавя препратка към тях в контекста на директивата. Атрибутът `view` е необходим за визуализацията на апертюрите на стенсила, като той представлява шаблон, получен в резултат на рендирането на Gerber файлове. При визуализацията на конфигурация за стенсил се използва предварително рендиран изглед на стенсил, който е един от файловете, които се свалят от хранилището за ресурси на приложението. При визуализацията на поръчки към атрибута `view` се подава резултата от рендирането на качените от потребителя файлове. Атрибутът `text` се използва за получаване на текста, който се визуализира като гравирани текст към стенсила, стойността му трябва да е масив от символни низове, като при избирането на конфигурация за стенсил стойността му е масив от един елемент със стойност “Text”. За визуализацията на стенсил изгледа на директивата използва вградената във Angular директива “`ngClass`”, която позволява CSS класове към HTML елементи да бъдат динамично подавани. Начините за динамично подаване на класове са: стандартният за HTML начин чрез символен низ използващ празно място за разделител на желаните класове, чрез използването на масив, чийто елементи са желаните класове, чрез подаването на обект, чийто ключови полета са имената на CSS класове, а кои да бъдат добавени към елемента става чрез проверка, на стойността на кои класове е логически



вярна, позволява се използване на изрази, чиито резултат е един от изброените или масив със стойности, които могат да бъдат едновременно символни низове и обекти.

В директивата `ivstStenilPreview` са използвани само първите два начина. Когато се връща само символен низ се използва израз, съдържащ така наречения “троен оператор за истина” и използване на модели, чиято стойност бива преизползвана в директивата, в проверката на условието му. При другия начин за стойности на масивите директно се използват модели, не принадлежащи на директивата.

### 3.13.2.1.2 Директивата “`ivstConfigurationInfo`”

Изгледа на директивата “`ivstConfigurationInfo`” се използва за описване на конфигурации за стенсил, чрез използването на полета за въвеждане на информация. В шаблона на изгледа на директивата е използвана вградената във Angular директива “`ngIf`”, която се използва за динамичното добавяне на HTML елементи, когато подаваната ѝ стойност е логически вярна. Всяко от полета за въвеждане на информация съдържа атрибута `required`, който оказва, че полето е задължително да бъде попълнено от потребителя. Съобщение за непълнено поле се извежда само когато някое от полетата, които потребителя вижда на своя екран не е попълнено. Полетата, на които е използвана директивата `ngIf` и подаваната ѝ стойност е логически невярна няма да бъдат създадени и по този начин директивата `required` се отнася само до видимите от потребителя полета. При промяна на валидността на формата, в която са разположени всички полета, свързващата функция на директивата съобщава за това, като подава новата стойност на полето отговарящо за валидността на формата. В изгледа на директивата е използвана директивата “`ngChange`”, част от Angular, за извикване на методите на контролера `configurationInterface`, когато негова инстанция съществува, при промяна на информация от полета. За унифициране на стойностите на информацията, която се изпраща, при запазване на дадена конфигурация е използвана директивата “`ngValue`”, отново предоставяна от избрания фреймуърк. Унифицирането на информация е необходимо, защото стойностите, които потребителя вижда са на езика, който той е избрал от интерфейса на приложението.

### 3.13.2.2 Контролера “configurationInterface”

Методите на контролера configurationInterface се използват за промяна на изгледа на конфигурацията за стенсил при нейното избиране. Контролера дефинира четири метода, които се използват за синхронизация между изгледите на конфигурацията и нейното графично изобразяване пред потребителя на Интернет приложението.

Метода “changeStencilTransitioning” присвоява вярна булева стойност на променливата “ctrl.configurationObject.style.frame”, когато потребителя избере стенсила да бъде рамкиран, което води до промяна в стойността, подавана на директивата ngClass, на всички елементи използваща троен оператор, което се визуализира като рамкиране с противоположно засенчване, което максимално да доближи визуализацията.

Метода “textAngle” присвоява масив съответстващ на възможните избори на ъгъл, от който да бъде четен гравирания върху стенсила надпис, на променливата “ctrl.options.textAngle”. Когато е избрано позицията на текста да е центрирана по вертикала на стенсила възможните избори са ляво и дясно, при центриране по хоризонтала: долу и горе и във всички останали случай и четирите посоки са възможен избор.

Метода “changeText” се използва за пълна промяна на гравирания текст, като чрез него се визуализира промяна в ъгъла на четене, позиция и цвят, съответстващ на начина на гравирание: “бял” когато съвпада със страната на стенсила използвана за ориентир при изработване на печатната платка съответстваща на стенсила, сив когато е от обратната страна и черен, когато текста се изрязва вместо да се гравира. Резултата от извикването на функцията е промяната на обекта, стойност на променливата “ctrl.configurationObject.style.text”. Обекта присвояван на ключовото поле text съдържа две полета: “color”, чиято стойност е един от трите класове за цвят използвани в Интернет приложението и “view”, чиято стойност се получава при събирането на: “text-”, поция, “-” и ъгъл.

Метода “changeStencilPosition” се използва за промяна на стенсила върху металния лист, от който се изрязва. Възможни промени на позицията на стенсила са за

подравняване, позициониране и очертаване. Подравняването може да е пейзажно и портретно. Позиционирането, подобно на възможността за гравирание на текст, може да е двустранно: от страната, която се приема за “горна” (центрираща) и от обратната страна. Очертаването може да е по формата на изрязвания стенсилен правоъгълник или без очертаване, като в такъв случай стенсила се центрира върху металния лист. Резултатът от извикването на метода е промяната на две булеви стойности на обекта “ctrl.configurationObject.style”, които определят очертаването. Промяната в позиционирането се изобразява чрез промяна на цвета на визуализираните апертюри на стенсила, за която промяна не е нужно извикването на функцията. Вместо това модела свързващ полето за въвеждане на информация директно се използва в директивата ngClass и промяната му води до автоматична промяна на цвета. За визуализиране на подравняването на стенсила се използва информация за избраното от потребителя подравняване, в комбинация с информация, която се извлича от очертаването, за това дали да се центрира стенсила върху металния лист.

### **3.13.2.3 Избиране на конфигурация за стенсилен**

Същинският избор на конфигурация се извършва, след като потребителят натисне бутона “NEXT”, който проверява дали всички полета са попълнени преди да премине към следващото състояние, ако потребителят е избрал да създаде нова конфигурация или директно преминава към смяна на състоянието на Интернет приложението. Формата съдържа поле за попълване “save”, което по подразбиране е попълнено. Когато полето save е попълнено, потребителят вижда поле за попълване на името на конфигурацията, което е задължително да се попълни, в такъв случай. Полето служи за информация дали конфигурацията на потребителя да бъде запазена на сървъра на приложението, като по този начин тя може да бъде преизползвана при направата на друга поръчка. Ако потребителят избере полето да остане празно, конфигурацията няма да бъде запазена и ще бъде еднократно използвана при завършването на поръчката. Ако потребителят избере да използва създадена от него конфигурация единствената възможност, която има е да натисне бутона NEXT, защото няма възможност да променя информацията на избраната от него конфигурация.

Стойността на полето save не се взема предвид при изпращането на цялата информацията на поръчка при нейното завършване. Решението информацията на двата вида конфигурации да се записва заедно с останалата информация на поръчката, за всяка поръчка, бе взето с цел отстраняване на възможни конфликти и предоставяне на възможността потребителя да прави поръчка без да е задължен да запамята информацията на всяка конфигурация. Конфликт може да възникне, ако вместо информацията на конфигурацията се пази връзка към конфигурация, запазена на сървъра. Ако при направата на поръчка бъде запазена връзка към дадена конфигурация и тя след това бъде променена от потребителя ще се получи не съответствие в информацията, която е била използвана за направата на поръчка и новата информация. Още по-голям проблем би се появил, ако потребителя след направата на дадена поръчка реши да изтрие конфигурацията използвал при направата на съответната поръчка.

### **3.13.3 Качване на Gerber файлове от потребителя и рендирането им в SVG шаблони с цел визуализацията им**

След като потребителя е избрал конфигурация за стенсил на поръчката, той получава възможност да качи желаните Gerber файлове, да въведе текстът, който да бъде гравирани и да попълни специфичната за поръчката информация относно размери на стенсила и брой на центриращи марки.

Възможните размери, които потребителя може да попълни са дебелина, височина и широчина на стенсила и заедно с броя на центриращите марки са задължителни, за да може цената на поръчката да бъде изчислена.

За да може потребителя да попълни текста, който да бъде гравирани, той първо трябва да качи Gerber файловете за изработка на поръчката. Изискването първо да бъдат качени файловете и след това да се попълни текста за гравирание е, защото при качване на файлове Интернет приложението разбира кое поле за въвеждане на текст да покаже.

### **3.13.3.1 Качване на Gerber файлове за визуализирането им в процеса на поръчване**

Интернет приложението дава възможност за качване на три Gerber файла: файл задаващ формата на стенсила (контура обгръщаш апертюрите), файл за горен слой и долен, като ако поръчвания стенсил ще бъде използван за произвеждането на едно-слойна платка няма значение кой от двата слоя ще бъде използван за целта. Интерфейса за качване и преглед на качени файлове се състои от три полета за отделните файлове, всяко от които представлява директивата “ivstFileContainer”, обединени в директивата “ivstFiles”, която се използва само за поддръждането на трите полета едно до друго и тяхното обособяване в една директива с цел преизползване на код.

#### **3.13.3.1.1 Директивата “ivstFileContainer”**

Директивата ivstFileContainer използва наследяване на контекста на нейния родител и приема входна информация чрез три атрибута: “layer” е името на слоя отговарящ на съответния Gerber файл (“outline” за контура на стенсила, “top” за апертюрите необходими за горния слой и “bottom” за апертюрите от долния слой), “order” отговарящ на обекта на поръчката, когато директивата е използвана в директивата “ivstViewOrder” и инстанцията на контролера “specificController”, когато директивата е използвана в изгледа на състоянието “home.order.specific” и “remove”, който сменя функционалността използвана при натискане на иконата, използвана за изобразяване на дадения файл. Иконата използвана за изобразяване на файлове е част от директивата “ivstFile” и представлява иконата “insert drive file” от “Material Design Icons”, чийто код е пренаписан от SVG на Jade (което е упоменато във файла “LICENSE”), за да може директно да бъде използвана в изгледа на директивата, от която е част. Тъй като и трите полета за качване на файлове до сървъра държат препратка към инстанцията на контролера specificController, който притежава обекта, който се подава на услугата uploadService и към който всяко поле прикача избрания за

него файл. Чрез услугата `RESTHelperService`, избирането на файл, за всяко от полета може да доведе правенето на заявка към сървъра и визуализирането на поръчката.

Шаблона на изгледа на директивата `ivstFileContainer` декларира, че чрез него може да бъде избран файл за качване чрез използване на директивите `“ngfSelect”` и `“ngfDrop”` в главния елемент на шаблона, предоставяни от използваният модул `“ng-file-upload”`, към фреймуърка `Angular`, за качване на файлове (Фиг. 3.13.3.1.1.1). Директивата `ngfSelect` оказва, че при кликване, върху елемента на директивата, от потребителя на Интернет приложението ще се отвори прозорец, изобразяващ файловата система на потребителя, от къде той може да избере кой файл да бъде подготвен за качване. Директивата `ngfDrop` оказва, че от прозорец, изобразяващ файловата система на потребителя може да бъде притеглен файл, който да бъде поставен в елемента на директивата и така той да бъде приготвен за качване до сървъра на приложението. Чрез шаблона изгледа и директивата `ngIf` се показват съобщения до потребителя, които оказват какво може да прави: да качва файл, да премахва избран файл и да сваля дадения файл.

```
div(ng-model="fileCnrCtrl.file" ng-change="fileCnrCtrl.upload()"
ngf-select ngf-drop class="dashed-border" layout="column"
layout-align="center center")
  div(translate="{{'layer-' + fileCnrCtrl.layer}}")
  div(ng-if="!fileCnrCtrl.file" layout="column" layout-align="center
center" layout-margin layout-padding)
    div(translate="div-select-file")
    div(translate="div-drop-file")
  div(ng-if="fileCnrCtrl.file" layout="column")
    div(ng-if="fileCnrCtrl.remove" layout="column"
layout-align="center center" layout-padding)
      div(translate="div-selected-file")
      div(translate="div-remove-file")
    div(ng-if="!fileCnrCtrl.remove" translate="div-download-file")
```

```

layout-margin layout-padding)

    div(layout="column" layout-align="center center")

        ivst-file(name="fileCnrCtrl.fileName()"
ng-click="fileCnrCtrl.action($event)")

```

(Фиг. 3.13.3.1.1.1) Шаблона на директивата ivstFileContainer

Контролера на директивата ivstFileContainer е отговорен за качването на Gerber файлове и интерпретирането на отговора на сървъра, което се извършва от вътрешната за контролера функция “preview” (Фиг. 3.13.3.1.1.2). Първото нещо, което функцията preview прави е да присвои стойност на ключовото поле “invalid” на обекта “ctrl.order”, която оказва дали файлът за очертаващия слой заедно с поне един от другите два слоя, не са избрани за качване. Следва извикване на метода “ifInvalid” на контролера “specificController” и ако той връща невярна булева стойност функцията изпраща файловете до сървъра. Метода ifInvalid проверява дали резултата от предходната операция е вярна булева стойност (условието необходимите слоеве нужни за рендирането да не са избрани, е вярно) и ако е, показва диалог целящ да съобщи на потребителя да избере файлове за нужните слоеве. Ако функцията е направила заявка, тя прави интерпретация на получения от сървъра отговор, като преди това запазва стойността на броя апертюри, получени при обработката на избраните файлове. За интерпретирането на отговора на сървъра на приложението се проверява типа на полетата “top” и “bottom”. Ако типа е символен низ, функцията приема, че стенсилът за съответния слой е бил правилно рендиран, като присвоява стойност на променливата “ctrl.order[layer].view”. Показването на рендираните Gerber файлове на потребителя става, чрез директивата “ivstOrderPreview”. Директивата ivstOrderPreview използва директивата ivstStecilPreview за визуализация на всеки стенсил, получен след рендирането на отделните Gerber файлове, като използва стойността, присвоявана на променливата ctrl.order[layer].view, чрез нейния атрибут view, за съответния слой. Ако стойността е невярна булева се приема, че файла е или празен Gerber файл или въобще не е Gerber файл и потребителя бива известен за това. Ако стойността е равна на null,

това значи че е настъпила грешка при обработката на някой от избраните файлове и потребителя бива известен за грешката, чрез показването на диалог.

```
preview = ->
    ctrl.order.invalid = not(ctrl.order.files.outline? and
(ctrl.order.files.top? or ctrl.order.files.bottom?))
    if not ctrl.order.invalidate() then
RESTHelperService.upload.preview ctrl.order.files, (res) ->
    ctrl.order.apertures = res.apertures
    for layer in ["top", "bottom"]
        if typeof res[layer] is "string" then ctrl.order[layer].view =
res[layer]
        else if res[layer] is null or res[layer] is no
            what = if res[layer] is no then "empty" else "error"
            simpleDialogService {}, "title-#{what}-layer-#{layer}"
```

*(Фиг. 3.13.3.1.1.2) Метода preview на контролера fileContainerController*

Метода “fileName” на контролера “fileContainerController” се грижи потребителя винаги да вижда само името на избрания от него файл. Ако файлът представлява качен на сървъра Gerber файл, когато директивата се използва при преглед на направена от потребителя поръчка, функцията връща името на файла, каквото е било преди той да бъде качен на сървъра.

Метода “upload” проверява размера на файла, ако той е прекалено голям, потребителя се известява, чрез диалог (Gerber файловете са текстови файлове и по подразбиране се приема, че размера им е сравнително малък, въпреки това за лимит е зададен размер от 10 MB), в противен случай избрания файл се добавя към файловете за качване и се извиква функцията preview.

Метода “action” динамично се добавя, когато бъде избран файл, което се постига чрез директивата ngIf от шаблона на директивата ivstFileContainer. Първото нещо, което прави action е да извика метода “stopPropagation” на събитието за качване



на файл, което ще предотврати отварянето на файловата система на потребителя, защото бива извикан когато потребителя кликне върху иконата на избрания файл (Фиг. 3.13.3.1.1.3). Следва проверка на стойността на полето “remove“, което е свързано към контролера от Angular и стойността му идва от подадената на едноимения атрибут на директивата. Когато тя е вярна булева стойност (директивата е използвана за избор на файл, а не за преглед на качен към дадена поръчка) избрания файл се премахва, за да може друг да бъде избран.. Ако файла, който се премахва не е бил контур на стенсила се извиква функцията preview, която визуализира на ново поръчката, ако е възможно и нужно. Когато стойността е невярна се приема, че потребителя иска да прегледа файла и той се изпраща за сваляне чрез отварянето на нова страница на браузъра на потребителя, която прави заявка към сървъра за извличането на файла. Използвано е връщане без стойност, защото по подразбиране CoffeeScript връща резултата от последния израз, който в случая е опакования от Ангулар “window” обект, който е забранен да бъде използван в изрази подавани (извикването на функцията чрез директивата ngClick се брой за израз и ако последния return липсва, резултата би бил самия window обект) в директивите на Angular от съображения за сигурност.

```
ctrl.action = (event) ->
  event.stopPropagation()
  if ctrl.remove
    delete ctrl.order.files[ctrl.layer]
    delete ctrl.file
    if ctrl.layer isnt "outline" and ctrl.order[ctrl.layer].view?
  then delete ctrl.order[ctrl.layer].view
    preview()
  else
    $window.open "api/order/download/" + ctrl.file, ctrl.fileName()
    return
```

(Фиг. 3.13.3.1.1.3)

### **3.13.3.2 Рендиране на Gerber файлове, чрез превръщането им в SVG шаблони**

Преди да се премине към рендирането на получените Gerber файлове, те трябва да бъдат приети и записани на сървъра на приложението. След, което благодарение на програмата Gerber файловете се конвертират в SVG файлове, към които в последствие биват добавени Angular директиви, за получаването на шаблони, които да отговарят на избраната от потребителя конфигурация за стенсил.

#### **3.13.3.2.1 Получаване и съхранение на Gerber файлове от сървъра на приложението**

За получаването и съхранението на файлове от сървъра на приложението бе използван npm пакета “multer”, който позволява обработка на заявки с прикачени към тях файлове. Файловете в случая на разработваната дипломна работа биват прикачвани от услугата Upload, използвана в услугата UploadService. Multer позволява как да бъдат конфигурирани и съхранявани файловете на сървъра чрез функции посредници. Като към всяка заявка, която се прикачи връщана от multer функция посредница прикача обект отговарящ на приетите файлове, под ключово поле с име “files”.

##### **3.13.3.2.1.1 Конфигуриране на multer за начина на съхраняване на файлове**

В разработваната дипломна работа конфигурацията, която е използвана оказва, че броя на качваните файлове не може да надвишава три, като размера на всеки от тях не може да надвишава 10 MB. За съхранение на файловете бе избрано записване на хардиска, на който оперира сървъра на приложението, като папката, в която се записват файловете зависи, дали те се качват на сървъра временно с цел визуализация на поръчката в нейния прогрес на извършване или се записват за по-голям период от време при направата на поръчка. Конфигурацията също оказва как да се построяват уникалните имена на файловете при тяхното записване с цел ограничение на възникване на

конфликти (Фиг. 3.13.3.2.1.1.1). За трайно съхранение на файлове се използва папката, подавана на функцията, експортирана от файла “/server/routes/order/file/multerConfig”, а за временно се използва папката “tmp”, в същата тази папка. За построяването на уникално име на всеки качван файл се използва \_id-то на потребителя, случайно генериран символен низ от 8 символа и пълното името на файла, ако съдържа разширение или към името на файла се добавя разширение, съвпадащо с името му, ако такова липсва.

```
config = (dir, prev) -> ->
  multer
    storage: multer.diskStorage destination: (dir + if prev then
"/tmp" else ""), filename: (req, file, cb) ->
  name = file.originalname
  fileName = name.match /\.[a-zA-Z]+/
  cb null, [req.user.user._id, randomString(), name + if
fileName? and fileName.length > 2 then "" else "." + name].join
"___"
  limits: files: 3, fileSize: 10000000
```

(Фиг. 3.13.3.2.1.1.1) Функцията използвана за конфигуриране на пакета multer

### 3.13.3.2.2 Обработка на получени Gerber файлове, чрез превръщането им в SVG шаблони

При получаване на прикачени файлове с цел визуализация на поръчка, в нейния прогрес на правене от потребителя, се изпълняват седем функции посредници и една като прихващач на заявки, която да върне отговор на направената заявка ако при рендирането на Gerber файловете не е възникнала грешка. Използвано е разделяне на логиката по обработка на заявката чрез обособяването на отделните “част” във функции посредници с цел преизползване на код.

Първата функция посредник се използва за временно съхраняване на файлове. Временно, защото ако настъпи грешка при рендирането на Gerber файловете записаните файлове биват изтрети от сървъра на приложението, а файловете, причинили грешката остават на сървъра с цел обратна връзка, защото дадена грешка е настъпила.

Втората функция посредник се грижи за преобразуването на получените данни с цел идентификация, кой файл на кой слой отговаря, за да може преобразуваната информация да бъде използвана за правилното рендиране на файловете.

Третата и четвъртата функции се опитват да визуализират стенсила за съответния слой, ако това е възможно. Следващите три функции изтриват файловете, качени на сървъра при обработката на постъпилата заявка. Последната функция, която се явява прихващач на заявки връща отговор на направената заявка, представляващ резултата от трансформацията на Gerber файлове в SVG шаблони, които могат да бъдат използвани от Интернет приложението за визуализация на поръчката.

### **3.13.3.2.2.1 Превръщане на Gerber файлове в SVG шаблони**

Целия процес по превръщането на Gerber файлове в SVG шаблони, които да бъдат върнати, като отговор на получената заявка се извършва от функцията, експортирана от файла `“/server/lib/GerberToSVG/GerberToSVGMiddleware.coffee”` (Фиг. 3.13.3.2.2.1.1). Функцията изпраща отговор, когато бъде извикана със символния низ `“send”`, като изпращания отговор е стойността на полето `“stencil”`, прикачено към обекта на заявката при предходните две извиквания на функцията, което пази резултата от опита за създаването на изглед за съответния стенсил. В останалите случаи, когато функцията бъде извикана със символен низ отговарящ на стенсил, за който може да се създаде изглед, се добавя полето `stencil`, ако не съществува на обекта на заявката. Следва проверка дали е наличен файл за съответния стенсил. Ако не е наличен се извиква механизма `next`, за да премине заявката към следващата функция посредник. Ако е наличен файл се извиква функцията експортирана от файла `“/server/lib/GerberToSVG/transform.coffee”`, която връща обект от тип Обещание. Когато връщаното Обещание премине във състояние `resolved` се извиква функция, която

интерпретира резултата от опита за създаването на изглед и адекватно записва резултата в полето “stencil”, прикачено към обекта на заявката, така че тя да бъде правилно интерпретирана, когато Интернет приложението получи отговора. За правилно създаден изглед се счита резултат, който е обект, съдържащ поле с име “preview”, което трябва да е символен низ, защото представлява произведения SVG шаблон, който в последствие се използва от приложението. Ако успешно е генериран SVG шаблон, той заедно с броя на апертюрите се запазват в полето stencil, в противен случай директно се запазва стойността на резултата, която представлява възникналия проблем при обработката на качените файлове. Ако е настъпила грешка при извикването на функцията “transform” се извиква механизма next, който ще извести за настъпилата грешка и ще попречи заявката да достигне до следващата функция, която би трябвало да бъде извикана при нормалната обработка на заявката.

```
module.exports = (middleware) ->
  if middleware isnt "send" then (req, res, next) ->
    if not req.stencil? then req.stencil = apertures: {}
    if req.gerbbers[middleware]?
      (transform req.gerbbers[middleware], req.gerbbers.outline)
        .then (stencil) ->
          if stencil? and typeof stencil.preview is "string"
            req.stencil[middleware] = stencil.preview
            req.stencil.apertures[middleware] = stencil.apertures
          else req.stencil[middleware] = stencil
          next()
        .catch next
    else next()
  else (req, res) -> query res, req.stencil
```

(Фиг. 3.12.3.2.1.1.1) Кода на функцията, експортирана от файла  
/server/lib/GerberToSVG/GerberToSVGMiddleware.coffee

Функцията експортирана от файла `“/server/lib/GerberToSVG/transform.coffee”` връща обект от тип Обещание, който представлява резултата от опита за даден стенсил да бъде създаден SVG шаблон, който да се използва от Интернет приложението за визуализация на поръчка (Фиг. 3.13.3.2.2.1.2). Първото нещо, което прави функцията подавана на конструктора на класа Обещание е да извика функцията, експортирана от файла `“/server/lib/GerberToSVG/convert.coffee”` със същите аргументите, подадени на описваната функция. Функцията `“convert”` също връща обещание, което представлява трансформация на подадените Gerber файлове във SVG файл. Когато обещанието, връщано от функцията `convert` премине в състояние `rejected` се извиква функцията `reject`, чрез която се известява за настъпилата грешка. Когато Обещанието премине в състояние `resolved` се проверява дали не е настъпила грешка при конвертирането на файловете, като се проверява дали стойността на резултата не е `null` и ако е, тя се подава и на функцията `resolve`, която известява за настъпилия проблем. Ако стойността на резултата не е `null` се използва метода `“load”` на обекта, експортиран от `prtm` пакета `“cheerio”`. Пакета `cheerio` предоставя интерфейса на библиотекта `“jQuery”` за ползване в Node, а чрез метода `my load` се зарежда парче от HTML код, върху което да се извършват манипулации, след което отново може да бъде получено парче HTML код, отразяващо направените модификации чрез метода `“html”`. В случая на разработваната дипломна работа се зарежда SVG кода на файла създаден, след извикването на програмта `gerbv`, при който след премахването на XML валидиращата схема става напълно валиден HTML код. След като се зареди резултата от трансформацията на подаваните файлове, се взимат всички елементи от тип `“path”` и ако няма такива се известява, че или подадения файл, посредством аргумента `“paste”` е или празен Gerber файл или въобще не е Gerber файл. Ако зарежданото парче код съдържа елементи със таг `path` се филтрират всички елементи, които са част от контура на стенсила и се премахват CSS селектори с цел цвета на филтрираните елементи да се определя от моделите използвани в една от добавяните директиви. Преди да се добави директива, контролираща дали очертаването да е видимо, се буферират всички `path` елементи на очертаването, след което те се премахват и се добавят като деца на елемент с таг `“g”` (`“group”`), към който са добавя директиваата `“ngClass”`. Обособяването на филтрираните елементи в `g` елемент, към който се добавя директивата `ngClass` е с цел бързодействие, което се постига благодарение на факта, че всички `path` елементи

наследяват трансформациите (добавянето на стилове чрез клас се смята за трансформация) извършвани на елемента g, в който са обединени. По този начин добавяната директива се изпълнява само веднъж, а резултата от нея рефлектира директно върху всички path елементи. Следва промяна на размера с цел съравмерност с елемента, към който се добавя крайния шаблон. Най-накрая се добавя директива контролираща цвета на всички path елементи и дали главния svg елемент да е центриран или не и съответно се премахва CSS селектора "fill" от path елементите отговарящи на апертюрите на стенсила. Преди крайния резултат от манипулациите да се обърне в HTML код, представляващ създадения шаблон, от него се премахват всички нови редове и символа "&apos;" използван от cheerio за единична кавичка се заменя със самия символ за единична кавичка с цел оказване на Angular, че става въпрос за символен низ, а не просто текст.

```
module.exports = (paste, outline) ->
  new Promise (resolve, reject) ->
    (convert paste, outline)
      .then (svg) ->
        if not svg? then resolve svg
        else
          $ = cheerio.load svg
          paths = $ "path"
          apertures = paths.length
          if not apertures then resolve no
          else
            filter = (paths, search) ->
              paths.filter (i, element) -> if
                element.attribs.style.match search then element
            replaceAll = (str, search, replace) -> str.replace (new
              RegExp search, "g"), replace
            removeAll = (str, search) -> replaceAll str, search, ""
            svg = $ "svg"
```

```

    attr = "ng-class"
    out = filter paths, /0%,0%,0%/
    out.css "stroke-width", ""
    out.css "stroke", ""
    outHTML = "<g
#{attr}=\"scopeCtrl.configurationObject.style.outline
    ? 'stencil-outline' :
'stencil-no-outline'\">>#{out.toString()}</g>"
    out.remove()
    ($ "g").append outHTML
    svg.attr "width", "80%"
    svg.attr "height", "90%"
    svg.attr attr,
"[(scopeCtrl.configurationObject.position.side || 'pcb-side'),
    (scopeCtrl.configurationObject.style.layout ?
    'stencil-layout' : 'stencil-centered')]"
    (filter paths, /100%,100%,100%/).css "fill", ""
    resolve apertures: apertures, preview: replaceAll
(removeAll $.html(), "\n"), "&apos;", "'"
    .catch reject

```

(Фиг. 3.13.3.2.2.1.2) Код на функцията, експортирана от файла  
 /server/lib/GerberToSVG/transform.coffee

Функцията експортирана от файла “/server/lib/GerberToSVG/convert.coffee” връща обект от тип Обещание, който представлява резултата от опита за даден стенсил да се произведе изглед, чрез рендирането на два Gerber файла в един SVG (Фиг. 3.13.3.2.2.1.3). Функцията извиква асинхронно програмата gerbv, като и подава аргументи оказващи изходния файл да е SVG и къде да се създаде, какъв да бъде цвета на апертюрите и файла използван за тях, както и какъв да е цвета на очертанието на стенсила и файла за него. Подават се различни цветове, за да могат елементите на изходния файл, да бъдат разпознати на кой Gerber слой отговарят. Ако някой от подаваните файлове не може да бъде прочетен се променя стойността на променливата



“error”, което gerbv съобщава чрез съобщение на стандартния поток за грешки, използван от програмата. Ако се получи грешка при стартирането на програмта gerbv връщаното Обещание преминава в състояние rejected. Когато програмата приключи своето изпълнение се проверява дали е настъпила грешка и ако е връщаното Обещание променя своето състояние в resolved. Ако не е настъпила грешка, се проверява дали използваната програма е създала желанния изходен файл и ако не е, се променя състоянието на връщаното Обещание. Ако файла съществува, той се прочита асинхронно и ако настъпи грешка при това четене обещанието сменя своето състояние в rejected. След като файла бъде успешно прочетен, той се изтрива, защото вече не е нужен и ако възникне грешка при изтриването му връщаното обещание преминава в състояние rejected. Ако не е настъпила никаква грешка при извикването на експортиратора функция, Обещанието, което тя връща преминава в състояние resolved, като за резултат от целия процес на конвертиране на входните файлове се получава SVG кода от изходния файл, без валидиращата схема, която се премахва. Връщаното от функцията Обещание преминава в състояние rejected, само когато настъпи грешка, която е по вина на сървъра на приложението, при което се извиква функцията reject със получената грешка. Грешките, които могат да се получат при асинхронното изпълнение на програмта gerbv не се борят за критични, защото най-вероятно са настъпили в резултат на грешно качени файлове от страна на потребителя на Интернет приложението и той се известява за това, като приложението интрепетира стойността null, която се подава на функцията resolve за качен файл с грешно съдържание. Като възможната грешка при извикването на gerbv се брой за критична и Обещаното преминава в състояние rejected и отново се извиква функцията reject с настъпилата грешка.

```
module.exports = (paste, outline) ->
  new Promise (resolve, reject) ->
    output = "./files/tmp/#{randomString()}.svg"
    error = no
    gerbv = spawn "gerbv", ["-x", "svg", "-o", output, "-a",
"--foreground=#FFFFFF", paste, "--foreground=#000000FF", outline]
```

```

gerbv.stderr.on "data", (data) ->
  str = data.toString()
  if str.includes "Unknown file type" or str.includes "could
not read" then error = yes
gerbv.on "close", ->
  progress = (err, server, cb) -> if err then (if server then
reject err else resolve null) else cb()
  progress error, no, -> fs.access output, (accessErr) ->
  progress accessErr, no, -> fs.readFile output, "utf8",
(readErr, data) ->
  progress readErr, yes, -> fs.unlink output, (removeErr) ->
  progress removeErr, yes, -> resolve data.replace '<?xml
version="1.0" encoding="UTF-8"?>', ""

```

*(Фиг. 3.13.3.2.1.3) Кода на функцията, експортирана от файла*

*/server/lib/GerberToSVG/convert.coffee*

Процеса на рендирането на качени от потребителя Gerber файлове в SVG шаблони е логически разделен в три отделни функции, всяка от които връща обект от тип Обещание, защото кода им се изпълнява асинхронно. В Node е препоръчително да се използва колкото се може повече асинхронен код. Неговото използване да бъде сведено до минимум - само при операции, за които е сигурно, че ще върнат стойност в сравнително кратък интервал от време. Node е изграден с цел да бъде максимално ефективен при конкурентното програмиране. JavaScript кода изпълняван от Node се изпълнява само в една нишка, в един процес и по този начин синхрония JavaScript код се явява блокиращ за използвания от Node модел, базиран изцяло около така наречения “Event Loop”.

### 3.13.4 Избор на конфигурация за адреси на доставката

Избора на конфигурация за адреси на доставката става от състоянието “home.order.adresses” на Интернет приложението. Изгледа на състоянието се състои от

директивата `ivstBase`, като за динамично свързан контролер се използва контролера `"addressesInterface"`, който наследява контролера `'ivstBaseInterface'` и добавя само един нов метод към него. Добавения метод се използва за автоматично попълване на едно от трите полета за адрес, част от изгледа, добавен в шаблона на директивата `ivstBase`, при маркирането на съответното поле за избор и изчистване на информацията при отхвърлянето на избора. Контролера извиква вътрешната за него функция `"listen"` преди да върне своя конструктор, която извиква услугата `stopLoadingService`, за да окаже, че зареждането на състоянието е завършило и добавя функция, която следи валидацията на полета за адрес и присвоява стойност в масива `"ctrl.valid"` за съответното поле. Последното нещо, което прави функцията е да добави функция, която да се извика при унищожаването на контекста на контролера, която да спре абонирането към събитието на предходната функция. Полетата за информация за адреси представляват директивата `"ivstAddress"`.

Директивата `ivstAddress` е създадена с цел преизползване на код. Тя се използва за едно от трите полета за попълване на адреси към доставката: адрес на самата доставка, адрес, на който се издава фактурата и адрес на фирмата поръчител. Свързващата функция на директивата съобщава когато всички полета бъдат попълнени или след като бъдат попълнени, някое остане отново празно. Контролера на директивата се използва за предоставяне на списък от държави и техните градове, от които потребителя може да избира.

### 3.13.5 Пресмятане на цената на поръчката

За пресмятането на цената на поръчката бе създадена директивата `"ivstPriceInfo"`, която използва само един атрибут за входни данни. Свързващата функция на директивата извиква услугата `scopeControllerService` със своя контекст, за да има достъп до метода `"calculatePrice"` на контролера `"orderController"`. Свързващата функция извиква и услугата `progressService` с цел навигация и добавя двете функции за смяна на състоянието към контекста на услугата. Директивата се използва за симулирането на продължително изчисление на цената на поръчката, което всъщност става за изключително кратък прериод от време. Симулирането на продължително

пресмятане е с цел по-добро потребителско изживяване и е постигнато чрез използването на услугата “\$inteveal”, вградена в Angular. Цената на поръчката се изчислява, чрез извикването на метода `calculatePrice` на контролера на състоянието “home.order”, който добавя обект, към обекта на поръчката, съдържащ информация за цената на всеки елемент от поръчката, за който се изчислява цена и общата сума на поръчката. Резултата от разделянето на стойността на всяка цена на броя на периодите за симулиране на продължително изчисление се запамятава чрез обекта “update”.

Услугата \$interval представлява опаковка на JavaScript функцията “setInterval”, която се използва за изпълнението на дадена функция през определен времеви интервал. Функцията, която се подава на услугата \$interval инкрементира стойността на променливата “count” с едно и стойността на ключовото поле “progress” от обекта на контекста с 20. Променливата count се използва за брояч на броя извиквания на функцията, която симулира изчисление на цената, а ключовото поле progress се използва като модел, който се подава на директивата “mdProgressCircular” за обновяване на полето изобразяващо пресмятането. Следва проверка дали функцията се е изпълнила определения брой пъти и ако е моделите на отделните цени от контекста, приемат стойностите на вече изчислените цени, добавят се нов модел към свързания с директивата контролера, които да съхранят крайната цена на поръчката и се премахва директивата mdProgressCircular от изгледа. Ако функцията подадена на услугата \$interval не била извикана определения брой пъти моделите на цените се обновяват. Функцията се извиква на всеки 400 мили секунди, което бе преценено, че е достатъчно като времеви период за да се забележи обновяването на цената от потребителя на приложението.

Всяко поле за показване на изчислена цена представлява директивата “ivstPriceField”, към която се свързват модели за вида на цената и нейната стойност, а тя се грижи те да бъдат представени в подходящ за потребителя формат. Директивата ivstPriceField е създадена с цел да бъде преизползвана в директивата, която показва пълната информация на всяка поръчка.

### 3.13.6 Преглед на поръчката и завършването ѝ

За преглед на поръчката от потребителя, преди тя да бъде завършена се използва директивата “ivstViewOrder”, която се преизползва и при преглеждането на направените от потребителя поръчки. Финализирането на дадена поръчка става посредством бутонна “FINISH”, който извиква метода “makeOrder” на контролера orderController и замества бутона “NEXT” от предходните състояния на приложението, през които потребителя е преминал за създаването на нова поръчка.

Метода makeOrder прави заявка към сървъра на приложението за трайно съхраняване на файловете, които потребителя е прикачил към поръчката. Отговора на направената заявка съдържа имената, дадени на качваните файлове, при получаването им от сървъра, които заедно с крайната цена на поръчката се добавят към обекта на поръчката и след това заедно с останалата информация на поръчката се изпращат до сървъра за създаването на нова поръчка. При получаването на отговор на втората заявка, Интернет приложението показва диалог на потребителя, че успешно е направил своята поръчка.

Директивата ivstViewOrder използва следните директиви, в своя шаблон: ivstOrderPreview, за да визуализира поръчката, ivstPriceField за да покаже крайната цена на поръчката, ivstFiles за да покаже прикачените към поръчката файлове, ivstOrderTexts за да покаже текста, който потребителя иска да бъде гравирен върху всеки стенсил, ivstConfigurationInfo за да покаже информацията на избраната конфигурация за стенсил, ivstOrderSpecific за да покаже размерите на стенсила и броя на центриращите марки, избрани от потребителя и ivstAddress, за всеки от адресите на доставката. След всяка от изброените директиви е поставена директивата “mdDivider” с цел логическо разделение между отделните компоненти на всяка поръчка.

Информацията за всяка директивата, от шаблона на директивата ivstViewOrder, идва от входния атрибут “order” на директивата ivstViewOrder, който получава препратка към контролера orderController, чрез изгледа на състоянието “home.order.finalize”. Благодарение на услугата progressService цялата информация, необходима за създаването на нова доставка е прикачена към инстанцията на

контролера на родителското състояние на състоянията, през които е преминал потребителя за попълването на отделни части от информацията за поръчката.

### **3.14 Преглед на направените поръчки, филтриране на показваните поръчки и следене на промяна на статуса на поръчките**

Всеки потребител може да вижда всички поръчки, които той е направил, а всеки администратор на приложението може да вижда всички направени поръчки. Всеки потребител може да изтрие всички поръчки, които бъдат отхвърлени и да заплати за всички одобрени поръчки, а администраторите съответно могат да променят статуса на всяка поръчка и да добавят коментар за промяната. При промяната на статуса на дадена поръчка се създава известие за тази промяна. Запазваната информация за промяната бе наречена известие, защото в кратък период от време приложението показва диалог известяващ потребителя, направил дадената поръчка, за промяната в статуса на съответната поръчка.

#### **3.14.1 Преглед на направени поръчки**

Подобно на директивата `ivstBase`, която се използва за преизползването на общия код за създаване, избиране, обновяване и изтриване на конфигурации за стенсил и адреси, бе подходено по аналогичен начин с цел преизползване на кода за прегледа на направените поръчки от потребители и администратори, с разликата, бе създадена директива `“ivstOrders”`. Шаблона на директивата, подобно на шаблона на директивата `ivstBase` позволява, чрез директивата `ivstInclude` да се добави специфична функционалност към общата, но за разлика от нея добавянето на нова функционалност е не задължително условие, защото такава е само администраторската функционалност на приложението. За прегледа на поръчки отново е използван подхода за наследяване на базов контролер, но заради разлика от контролера `baseInterface`, базовия контролер директно се използва, без да бъде наследяван, за прегледа на поръчки от потребителя.

Взимането на всички поръчки става чрез направата на заявка към сървъра на приложението, като те се извличат от базата данни. Ако текущия потребител на

Интернет приложението е администратор се извличат всички поръчки, ако не е само направените от съответния потребител, което става чрез проверка на стойността на полето “admin” от документа на потребителя, прикачен към обекта на заявката от някоя от сесийните функции посредници. Всяка поръчка представлява един ред от псевдо таблицата, част от изгледа, получен след компилирането на шаблона “ordersBaseView” (Фиг. 3.14.1.1). За създаването на изгледа на таблица три пъти е използвана директивата `ngRepeat`, вградена в `Angular`. Първото използване на директивата е за имената на колоните, второто за редовете за поръчките и трето за колоните на редовете. За да изглежда създадената таблица, като истинска бе използвана директивата `flex`, която е част от фреймуърка `Angular-Material`, чрез която се постига ефекта на “grid system” (мрежеста система), около който са изградени CSS фреймуърци като `Bootstrap`. Мрежовата система позволява да бъде оказано всеки HTML елемент, каква част от размера на своя родител да заема. Директивата типично приема число, интерпретирано от нея като символен низ, което отговаря на процента от ширната на родителя, който съответния елемент ще има като своя дължина. Освен число директива може да приема и ключови думи, чрез които се оказва каква да е дължината. Такава е използваната дума “auto”, която оказва, че размера ще се променя до нужния, като започва от базовата дължина и ширина на елемента, към който е добавена директивата. Повреме на разработваната дипломна работа `flex` системата, която `Angular-Material` използва, не позволяваше всеки HTML елемент да бъде използван като `flex “container”` (контейнер), тоест към някой от HTML елементите не бе възможно да се добавят CSS селектори, част от `flex` системата, която е добавена е CSS3. Един такъв елемент е “button” елемента, който се използва от директивата “mdButton”, чрез която се добавят бутони към шаблоните използващи `Angular-Material`. За решаването на този проблем всички елементи от колоните на всеки ред представляват “span” елемент, който може да бъде `flex` контейнер. За да изглеждат span елементите като бутоните `mdButton` са добавени съответните CSS класове, които различните директиви, част от `Angular-Material` добавят към елемента на бутоните, създавани чрез директивата `mdButton`. Бе създаден и CSS класа “fake-button”, който премахва визуалните ефекти при минаване през даден `mdButton` или при натискането му, които се добавят от класовете “md-button” и “md-raised” използвани върху `span` елементите. Изключение правят елементите от първата колона, който са използвани

като истински бутони и при тяхното натискане се показва информацията за съответната поръчка и span елемента използван за плащане или изтриване на съответната поръчка от потребителя и обновяване от администратора.

```
md-content(layout="column" flex="auto" layout-wrap)
  md-divider
  div(layout="row" flex="auto")
    span(ng-repeat="(key, value) in ordersCtrl.labels"
flex="{{value}}" ng-class="key === '_id' ? 'md-primary' : 'md-warn'"
class="md-button fake-button" translate="{{'span-' + key}}")
    span(flex="15" class="md-button md-primary fake-button"
translate="span-action")
  md-divider
  div(ng-repeat="order in ordersCtrl.listOfOrders"
ng-if="!ordersCtrl.showing || order._id === ordersCtrl.showing"
layout="column" flex="auto")
    div(ng-mouseover="ordersCtrl.removeNotifcation(order)"
layout="row" flex="auto")
      span(
        ng-repeat="(key, value) in ordersCtrl.labels"
        ng-click="key === '_id' ? ordersCtrl.choose($event, order) :
null"
        flex="{{value}}"
        ng-class="[(key === '_id' ? 'md-primary' : 'md-warn
fake-button'), (order.notify ? 'md-raised': '')]"
        class="md-button"
        translate="{{key === 'status' ? 'span-status-' + order[key]
: order[key]}}"
      )
      span(
```



```

    flex="15"
    class="md-button md-raised"
    ng-click="ordersCtrl.doAction($event, order)"
    ng-if="ordersCtrl.adminPanel ||
ordersCtrl.helpStatus(order)"
    ng-class="ordersCtrl.adminPanel || order.status ===
'rejected' ? 'md-warn' : 'md-primary'"
    translate="{{'button-' + (ordersCtrl.adminPanel ? 'update' :
(order.status === 'accepted' ? 'pay' : 'delete'))}}"
  )
  span(flex="15" disabled="true" class="md-button fake-button"
ng-if="!ordersCtrl.adminPanel && !ordersCtrl.helpStatus(order)")
  md-divider

```

(Фиг. 3.14.1.1) Част от шаблона *ordersBaseView*, която създава изглед на псевдо таблица на поръчките

При натискане на първия елемент от редовете на поръчките се извиква метода “choose” на контролера *ordersInterface*, който присвоява уникалния идентификационен номер на избраната поръчка на модела “showing”. Когато на модела *showing* е присвоена стойност, информацията за избраната поръчка се показва, чрез директивата *ivstViewOrder*, в редовете от таблицата остава само реда на избраната поръчка и се добавя бутон, който когато бъде натиснат отново показва всички поръчки. Метода *choose* прави заявка към сървъра за рендирането на прикачените към поръчката файлове, след което обединява получените изгледи с текста за гравирание в обект, който да може да се използва от директивата *ivstOrderPreview* използвана в директивата *ivstViewOrder* и накрая показва описание към поръчката, като прави нова заявка към сървъра. Ако съществува описание се използва диалоговата услугата “*showDescriptionService*” за показването му.

### 3.14.2 Филтриране на показваните поръчки

Поръчките могат да се филтрират по дати и ключова дума, която се съдържа в тяхната информация, като единствено изключение е информацията от избраната конфигурация за стенсил, защото тя се пази в формат независим от езика, който потребителя на приложение е избрал, с цел интернационализацията на Интернет приложението. Като двата вида филтрация могат да бъдат използвани едновременно и резултата на показваните поръчки ще бъде вътрешното сечението на поръчките, чиято дата е във избрания диапазон и поръчките, които съдържат търсената дума.

Филтрирането на поръчките по дата става посредством избирането на диапазон от дати, който се явява затворен интервал. За избирането на двата края на интервала бе използвана директивата “mdDatepicker”, която използва услугата “\$mdDateLocale” за визуализация на избраната дата, форматът, в който тя се показва на потребителя. Услугата \$mdDateLocale се създава от нейния доставчик, чрез който формата за визуализация може да бъде променян. С цел уеднаквяване на показването на датите на създаване и доставяне на поръчката и на датите, избрани за филтриране на поръчките бе създаден доставчик на услуги “dateServiceProvider”, който да улесни работата с дати на Интернет приложението и методите му да бъдат използвани за конфигуриране на услугата \$mdDateLocale. Избрания формат за визуализация на дати е разделянето на годината, месеца и деня на съответната дата с “/”.

Филтрирането на поръчките бе реализирано посредством използването на вградения в Angular филтър “filter” и използването на услугата “\$filter”, също част от избрания фреймуърк, която позволява използването на филтри извън шаблоните. Филтъра filter позволява масив от елементи да бъде филтриран, като за целта могат да се използват символен низ за търсене на съвпадения, обект за съвпадения между съответни ключове и стойностите им и функция, която връща булева стойност оказваща дали елемента, за който се вика функцията трябва да остане в масива, който филтъра връща. Филтрацията на поръчки се извършва от метода “filterFn” на контролера на директивата ivstOrders, който първо филтрира поръчките в зависимост от въведената ключова дума в полето за филтрация. Ако потребителя е въвел стойност в

него, след, което поръчките се филтрират по дата, като на филтъра `filter` му се подава функция, която връща вярна булева стойност само когато датата на направената поръчка е в избрания диапазон (Фиг. 3.14.2.1). Преди функциите да бъдат филтрирани по дата се проверява дали въведените от потребителя дати не са с разменени места, тоест крайната дата е по-малка от началната и ако е се използва деструктивността, с която CoffeeScript разширява JavaScript. Защитата от обръщане на интервала на датите, за които се показват поръчки е част от метода `filterFn`, защото тя се извиква при промяната на двата модела.

```
ctrl.filterFn = (newValue) ->
  filtered = if ctrl.filter? and ctrl.filter.length then filter
ctrl.fullListOfOrders, ctrl.filter else ctrl.fullListOfOrders
  if ctrl.toDate < ctrl.fromDate then [ctrl.fromDate, ctrl.toDate] =
[ctrl.toDate, ctrl.fromDate]
  ctrl.listOfOrders = filter filtered, (order) ->
    if order? then ctrl.toDate >= (dateService.parse
order.orderDate) >= ctrl.fromDate else no
```

*(Фиг. 3.14.2.1) Метода filterFn на контролера ordersInterface*

За да работи филтрирането по дати на поръчки независимо дали потребителя е въвел стойности, които определят крайщата на интервала за филтриране се задават начални стойности на двата модела, които съответстват на текущата дата и им се присвояват нови стойности, след като поръчките бъдат извлечени от сървъра на приложението, които съответстват на датите на поръчване на първата и последна поръчка, от извлечените, което се извърва от вътрешната за контролера `ordersInterface` функция “init” (Фиг. 3.14.2.2). Функцията се използва и за сортирането на масива от поръчки, който се подава на функцията, форматирането на датите на поръчките и добавянето на ключово поле, към доставките, за които има известие. Първото сортиране на поръчките се извършва при извличането на поръчките от базата данни, преди да бъдат върнати като отговор на направената от функцията `init` заявка. Вторично поръчките се сортират спрямо техния статус и ако две поръчки имат един и

същи статус, допълнително се сортират в зависимост от това, за коя поръчка има непроведено известие. За прилагането на двата филтъра приложението следи за промяна на полето за филтриране по ключова дума и полетата за задаване на диапазон на дати, а за показването на известия контролера следи за съобщаването на събитието notification. За поръчките, които има известия се добавя класа “md-raised”, към полетата от реда на съответната поръчка, в зависимост от това дали към съответната поръчка има прикачено поле “notify” от функцията “transform”.

```
init = ->
  RESTHelperService.order.find (res) ->
    orders = res.orders
    begin = orders[0]
    end = orders[orders.length - 1]
    if begin? then ctrl.toDate = dateService.compatible
begin.orderDate
    if end? then ctrl.fromDate = dateService.compatible
end.orderDate

transform = (full) ->
  transformFn = (order) ->
    if full then order[type + "Date"] = dateService.format
order[type + "Date"] for type in ["order", "sending"]
    order.notify = notificationService.notificationFor order._id
    order

(transformFn order for order in orders).sort (a, b) ->
  value = (some) -> if some.notify? then 1 else 0
  index = (ctrl.status.indexOf a.status) -
(ctrl.status.indexOf b.status)
  if not index then (value a) - (value b) else index
```

```

ctrl.fullListOfOrders = transform yes
trl.listOfOrders = ctrl.fullListOfOrders
stopLoadingService "orders"

listeners = ($scope.$watch "ordersCtrl." + watch, ctrl.filterFn
for watch in ["filter", "fromDate", "toDate"]])

$scope.$on "notification", ->
    ctrl.fullListOfOrders = transform no
    ctrl.filterFn()

$scope.$on "$destroy", -> listener() for listener in listeners

```

(Фиг. 3.14.2.2) Функцията *init* от контролера *ordersInterface*

### 3.14.3 Следене на промяна в статуса на поръчките

За следене на промяна в статуса на поръчките бе създаден доставчик на услуга “notificationServiceProvider”. Доставчика се използва за конфигуриране на състоянието, което може да отрази дали има известия за промяна на статуса, за които потребителя не е оведомен. Тоест състоянието, на което Интернет приложението трябва да се намира за показването на новите известия. В разработваното Интернет приложение това е същото състояние, което показва направените поръчки на потребителя - “home.orders”.

Услугата предоставя метод “notify”, който се използва за извличането, съхранението и показването на невидяните известия. То проверява дали потребителя на приложението е влязал в системата и не е администратор. Ако резултата от последната проверка е логически вярна стойност метода прави заявка до сървъра за извличането на всички известия, които са за конкретния потребител на Интернет приложението. Следва проверка дали има такива известия и ако има променливата

“notifications”, която съхранява известията получава същата стойност, която е имала при инжектирането на услугата - празен обект. Следва запазването на всяко известие, като към обекта notifications се добавят уникалните идентификационни номера на поръчката като ключ и известието, като стойност на новия ключ. Последното, което notify прави е да съобщи за получените известия чрез диалоговата услуга “showNotificationService”, която показва на потребителя, че има промяна в статуса на някоя от неговите поръчки, когато потребителя натисне синия бутон на показвания диалог. Известието става посредством уведомяване за настъпване на събитието "notification", ако приложението се намира на конфигурираното състояние, в противен случай приложението отива на конфигурираното състояние.

За следенето на промяна на статуса на направените от потребителя поръчки се използва метода “listenForNotification” на услугата “notificationService”, който използва метода й notify. Метода listenForNotification спира следенето на стаус, ако такова има чрез метода stopListen на същата услуга. След 10 мили секунди се опитва да покаже дали има известия, за които потребителя не е уведомен, подавайки метода notify на услугата \$timeout и повтаря този опит на всяка минута, отново подавайки същия метод, но на услугата \$interval.

Поръчките за които има известие лесно могат да бъдат разграничени от потребителя, защото към всеки елемент от реда на всяка поръчка, за която има известие, се прибавя класа md-raised. Когато потребителя мине с мишката през такъв ред се извиква метода “removeNotification” на контролера ordersInterface, който извиква едноименния метод на услугата notificationService, който отправя заявка към сървъра на приложението да премахне от базата данни съответното известие към дадената поръчка.

### **3.15 Администрация на приложението**

Когато някой администратор бъде разпознат от системата, Интернет приложението променя своето състояние. Състоянието се променя на “home.admin”, от където администраторите имат възможност да променят статуса на поръчките, да преглеждат изготвяните статистики, да добавят нови администратори и да изтриват

потребители. Цялата тази функционалност се добавя от шаблона “adminPanelView”, който добавя два таба (“mdTab”) към този за преглед на поръчките (Фиг. 3.15.1).

Първият таб представлява три отделни таба за всеки вид статистика, които динамично се изграждат, използвайки директивата ngRepeat. Тялото на всеки от табовете представлява директивата “ivstBarChart”. Решението е да се използват вложени табове, защото по този начин най-лесно статистиките могат да се разделят в отделни изгледи. Това може да се приеме за вид виртуализация на страниците на приложението, но без да е нужно конфигуриране на състояния, защото всеки изглед представлява само една графика.

Втория таб се състои от навигационно меню, което се използва за смяна на състоянията на приложението, добавени чрез директивата uiView. Във втория таб бе решено да се използват вложени състояния, защото по този начин функционалността от състоянието “home.settings.profile” директно може да бъде преизползвана и предоставена за ползване на администраторите и същевременно лесно може да бъде добавена функционалността за управление на потребители от администратори.

md-tab

md-tab-label

span(translate="label-statistics")

md-tab-body

md-tabs(md-selected="0" md-dynamic-height="true" md-center-tabs)

md-tab(ng-repeat="chart in ['revenue', 'count', 'visit'])

md-tab-label

span(translate="{{'span-' + chart}}")

md-tab-body

ivst-bar-chart(chart="scopeCtrl.charts[chart]")

md-tab(ng-click="scopeCtrl.editProfile()")

md-tab-label

span(translate="span-admin")

md-tab-body

ivst-state-switcher(state="home.admin" menu="admin")

```
ui-view(layout="column" layout-wrap layout-padding
layout-margin)
```

(Фиг. 3.15.1) Шаблон *adminPanelView*, добавян към шаблона на директивата *ivstOrders*

### 3.15.1 Промяна на статуса на поръчка

Промяната на статуса на дадена поръчка става посредством диалога създаван от услугата “showDescriptionService”, който се използва и за преглед на описание към смяната на статус на дадена поръчка. Шаблонът на диалога използва директивата *ivstOrderText* за въвеждане на описанието и прегледа му. Когато към диалога се свързва поле “admin” с вярна булева стойност се добавя възможност за смяна на статуса на дадена поръчка, ръчно въвеждане на цената на поръчката и преглед на избория от потребителя език, посредством добавянето на модела “adminPanel”, чиято стойност е името на добавения изглед. Когато синия бутон на диалога бъде натиснат, диалога се затваря, ако е натиснат от обикновен потребител, а ако е натиснат от администратор се обновява информацията за дадената поръчка. Диалога автоматично се показва когато на уникалния идентификатор на дадена поръчка бъде кликнато. Ако администраторът е кликнал и статуса на поръчката е бил “нова” се показват възможностите за обновяване на статуса на поръчката, в противен случай се показва само описанието към нея, ако има такова. Диалога, заедно с възможностите за обновяване на информацията за дадена поръчка, може да бъде отворен и при натискане на бутон “Update”, от реда на съответната поръчка.

Ако липсва описание, при обновяване на информацията на дадена поръчка от администратора, сървърът на приложението ще използва шаблона на описание за съответния статус на избория от потребителя език. Добавянето от сървъра описание е във вид на шаблони, защото подобно на шаблоните, които Angular използва и компилира за да създаде изгледи, които представляват интерфейса на приложението, бе създадена система, която позволява в описанието да бъдат използвани свързвания с цел създаване на улеснение за администраторите. Свързванията в имплементацията на шаблони, която бе реализирана се означава с “&@” и след него се добавя името на ключовото поле, чиято стойност да замести означението.



Функцията, експортирана от файла `"/server/routes/order/description/resolveDescriptionBindings.coffee"` заменя всички свързвания (Фиг. 3.15.1.1). Първият аргумент, който тя приема е масив от символни низове, който представляват получения от сървъра текст въведен от администратора или резултата от прочитането на шаблон за описание. Вторият е обекта, който се използва за свързванията в разработваното приложение, това е обекта, изпращан от разработваното Интернет приложение, представляващ обновената информация на дадена поръчка. Функцията се опитва да провери дали даден елемент от входния масив съдържа свързване и ако съдържа да го замени със съответната стойност от обекта, подаван като втори аргумент, ако той съдържа необходимото ключово поле. Резултата връщан от функцията представлява входния масив със заместени свързвания, ако такива са били открити и това е било възможно.

```
module.exports = (template, populate) ->
  for index of template
    if (template[index].match /&@[a-zA-Z]+/)?
      bind = populate[template[index].replace "&", ""]
      if bind? then template[index] = "" + bind
  template
```

*(Фиг. 3.15.1.1) Код на функцията, експортирана от файла /server/routes/order/description/resolveDescriptionBindings.coffee*

Когато сървър на приложението получи заявка за обновяване на статуса на дадена поръчка, той проверява дали администратора е въвел описание (Фиг. 3.15.1.2). Ако администратора не е въвел (тоест първия елемент от изпращания модел на описанието е празен символен низ) се извиква функцията `"getDescriptionTemplate"` и резултата от нея се добавя към обекта на заявката. В противен случай масива, който представлява въведеното описанието се добавя към обекта на заявката. Функцията `getDescriptionTemplate` приема име на статус и език и връща обект от тип Обещание. Тя прочита асинхронно шаблона, който отговаря на подадените аргументи и ако не настъпи грешка, променя състоянието на връщаното Обещание в `resolved`, подавайки

на функцията `resolve` масив, чийто елементи са текста от всеки ред на прочетения файл, а ако настъпи грешка извиква функцията `reject` с настъпилата грешка. Преди да се добави описание към обекта на заявката задължително се обновява нейния статус, ако той е “sent” (поръчката е била изпратена) се обновява и датата на изпращане, а ако администратора ръчно е въвел цена тя също се добавя към обекта подаван на MongoDB оператора “\$set”. След добавянето на описание, заявката преминава към функция посредница, която замества всички свързвания чрез извикването на функцията “`resolveDescriptionBindings`” и обновява документа от колекцията на описанията, ако съществува запис, който да съдържа уникалния идентификационен номер на поръчката. Функцията добавя нов документ, ако такъв не съществува. Прихващача на заявки създава нов документ към колекцията на известията и връща отговор на заявката.

```
handle.patch = [
  (req, res, next) ->
    text = req.body.text
    order = status: req.body.status
    if req.body.price? then order.price = req.body.price
    if order.status is "sent" then order.sendingDate = new Date()

    (orderModel.findByIdAndUpdate req.body.id, $set: order, {new:
yes}).exec()
      .then (doc) ->
        if text[0] is ""
          (getDescriptionTemplate order.status,
req.body.language).then (txt) ->
            req.text = txt
            next()
        else
          req.text = text
          next()
```

```

        .catch next

    (req, res, next) ->
        binded = resolveDescriptionBindings req.text, req.body
        (descriptionModel.update order: req.body.id, {text: binded},
{upsert: yes})
            .exec().then (-> next()), next

    (req, res, next) ->
        (notificationModel.create order: req.body.id, user:
req.body.user)
            .then ((doc) -> query res, doc), next
]

```

*(Фиг. 3.15.1.2) Част от описанието на рутера, експортиран от файла  
/server/routes/user/userHandle.coffee, която изтрива потребители на приложението*

### 3.15.2 Преглед и изготвяне на статистики

Статистиките за приложението могат да бъдат видяни само от администраторите, като за целта те трябва да навигират до таба за преглед на статистики. Приложението предоставя три вида статистики:

1. за генерирана печалба;
2. за брой на направени поръчки и брой на успешно доставени поръчки (тези които генерират печалба);
3. за брой на всички посещения на приложението и посещения от потребители, разпознати от системата.

Периода, за който да се показват статистиките съвпада с периода, за който се показват поръчки.

### 3.15.2.1 Преглед на статистики

За показването на статистиките бе създадена директивата `ivstBarChart`, която използва само един атрибут за входни данни, представляващ обект, описващ графика. Шаблона на директивата представлява “`canvas`” елемент с директиви от модула “`chart.js`”. Решението да се създаде директива, вместо шаблона директно да бъде използван в `ngRepeat` цикъла от изгледа `adminPanelView`, бе породено от желанието показваните статистики да бъдат с реагиращ дизайн. Модула `chart.js` има вградена поддръжка за реагиращ дизайн на създаваните графики, но това, което той прави е да използва цялото пространство на родителския елемент. Това на мобилни устройства не изглеждаше добре, защото се получава ефект на сбиване на графиката, подобен на ефекта, който се получава при намаляне няколко пъти на резолюцията на дигитална снимка. За това статистиките бяха обособени в директива, чиято свързваща функция се грижи графиките да бъдат реагиращи. Първото нещо, което бе направено за решението на проблема е да се окаже на модула да изключи механизма, който ползва за да прави графиките реагиращи и да се сменят цветовете на графиките, за да са в тон с използваната цветовата темата на приложението, чрез метода “`setOptions`” на доставчика “`ChartJsProvider`”.

За постигането на реагиращ дизайн, при първата компилация на директивата и при промяна на размера на екрана се извиква вътрешната функция “`resize`” от свързващата функция на директивата. Функцията `resize` добавя изгледа на графиката като дете на елемента на директивата, компилирайки го чрез извикването на услугата `$compile` и променяйки размерите на неговия `canvas` елемент (Фиг. 3.15.2.1.1). Като новата височина на `canvas` елемента е два пъти по-малка от неговата дължина, която е 80% от дължината на прозореца на браузъра.

```
link: (scope, element, attrs) ->
    wrapper = element.children()

    resize = ->
```

```

wrapper.html $templateCache.get "barChart"
($compile wrapper.contents()) scope
canvas = wrapper.find "canvas"
part = width: 4, height: 2
canvas.prop prop, $window.screen[prop] * value / 5 for prop,
value of part

resize()

```

```

(angular.element $window).on "resize", resize

```

*(Фиг. 3.15.2.1.1) Код на свързващата функция на директивата ivstBarChart*

В процеса на измисляне на решение на проблема как графиките да станат с истински реагиращ дизайн бе открито, че размера на canvas елемента може да бъде променян само веднъж, защото при неговото добавяне браузъра пресмята неговата ширина и дължина, добавяйки го като нов елемент и само първата промяна на размерите му се отразява от браузъра, когато браузъра не поддържа технологията “WebGL”. С цел приложението да има едно и също поведение при различните браузъри бе решено да се използва решение, което работи по един и същ начин независимо от вида на браузъра.

### 3.15.2.2 Изготвяне на статистики

Статистиките се изготвят от функцията “init”, която се явява вътрешна за контролера “adminController”, който наследява контролера “ordersInterface”. Функцията init извлича от сървъра на приложението всички посещения, като получи отговор запазва резултата и използва услугата stopLoadingService, за да окаже, че изгледа е готов за ползване. При промяна в модела на показваните поръчки, изготвя нова информация за графиките на статистиките (Фиг. 3.15.2.2.1). Първото нещо, което прави функцията, която следи за промяната е да прекъсне изпълнението си ако модела е празен. След което създава обект, чрез извикването на метода “iterator” и а

у с л у г а т а `dateService`, който като концепция реализира “forward iterator” (итератор) от езика C++, за обхождане на затворен интервал от дати. Създавания итератор има член променлива “value”, която съхранява дата, до която е достигнал итератора и метод “next”, който отмества дата с един ден и връща булева стойност, която оказва дали е достигнат края на интервала. Статистиките се водят за интервали от време. Дължината на интервала се изчислява от функцията, така че броя на интервалите да е сравнително постоянен, в зависимост от дължината на интервала, определян от моделите `fromDate` и `toDate` на контролера. Всеки интервал се състои от “label” (етикет), който се използва за ключово поле в обекта “intervals” и обект от стойности, които се следят, записвани като стойност за съответния етикет. Имената на следените стойности са записани в масива “about”, а променливата “interval” съхранява етикета на интервала, за който се пресмятат стойности. За създаването на обекта, който описва всички графики се използва функцията “buildCharts”, като преди това за всяка дата от определения от администратора диапазон се извиква функцията “buildIntervals”.

```
init = ->
  RESTHelperService.visit.find (res) ->
    ctrl.listOfVisits = res.visits
    stopLoadingService "admin"

    stopStatistics = $scope.$watch "ordersCtrl.listOfOrders",
  (orders) ->
    if not orders? then return
    beggin = ctrl.fromDate
    end = ctrl.toDate
    it = dateService.iterator beggin, end
    gap = end.getMonth() - beggin.getMonth()
    years = end.getFullYear() - beggin.getFullYear() + 1
    normalize = Math.abs (Math.floor end.getDate() / 3) -
```

```

(Math.floor beggin.getDate() / 3)

    diff = Math.abs (Math.floor ((3 * gap * years) + 3 -
normalize) / 3) + 3

    interval = current: diff, label: ""
    intervals = {}
    about = ["count", "delivered", "revenue", "visits", "users"]

    ...

    buildIntervals it.value while it.next()

    ctrl.charts = buildCharts()

```

(Фиг. 3.15.2.2.1) Съкратен вариант на функцията *init* на контролера *adminControlller*

Функцията *buildCharts* (Фиг. 3.15.2.2.2) връща обект, описващ три графики, като всяка от графиките е обект, който съдържа три ключови полета: “series” - масив, който описва легендата за съответната диаграма, с имената на различните стълбове; “data” - масив от масиви, съдържащи стойностите за всеки стълб; “labels” - масив с етикетите за всеки интервал. Всички графики се изготвят за един и същ интервал от време, за това и всички графики, получават препратка към масива “labels”, като стойност на едноименното поле. Но преди това се разпределят стойностите от всеки интервал в масивите от полето “data” от всеки обект.

```

buildCharts = ->
    labels = []
    charts =
        count:
            series: ["line-orders-count", "line-delivered"]
            data: [[], []]

```

```

revenue:
  series: ["line-revenue"]
  data: [[]]
visit:
  series: ["line-unique-visits", "line-unique-users"]
  data: [[], []]

for label, data of intervals
  labels.push label
  charts.count.data[0].push data.count
  charts.count.data[1].push data.delivered
  charts.revenue.data[0].push data.revenue
  charts.visit.data[0].push data.visits
  charts.visit.data[1].push data.users

charts[chart].labels = labels for chart in ["count", "revenue",
"visit"]

charts

```

(Фиг. 3.15.2.2.2) Кода на функцията *buildCharts*, част от функцията *init* на контролера *adminController*

Функцията *buildIntervals* (Фиг. 3.15.2.2.3) приема обект от тип “Date” и използва функцията “*statisticData*”, която връща информация за следените данни, подавайки ѝ форматираната дата. След което проверява дали е достигнала края на поредния интервал и ако е, ресетва стойността на полето “*current*” на обекта *interval*. Присвоява стойността на форматираната дата на полето *label* на същия обект и добавя нов празен интервал към обекта *intervals*, чрез извикването на функцията “*emptyRecord*”. Ако края на поредния интервал не е достигнат, тоест полето *current* на обекта *interval* е по-малко от изчислената стойност на променливата “*diff*”, се



инкрементира стойността на проверяваното поле и на променливата `label` се присвоява стойността на едноименното поле от обекта `interval`. Най-накрая добавя стойностите запазени в локалната променлива `“data”` към интервала с етикета (стойността на променливата `“label”`).

```
buildIntervals = (date) ->
  label = dateService.format date
  data = statisticData label
  if interval.current is diff
    interval.current = 0
    interval.label = label
    intervals[label] = emptyRecord()
  else
    label = interval.label
    interval.current++
  intervals[label][info] += data[info] for info in about
```

*(Фиг. 3.15.2.2.3) Кода на функцията buildIntervals, част от функцията init на контролера adminController*

Функцията `statisticData` приема форматирана дата и връща обект със следената информация за подадения аргумент (Фиг. 3.15.2.2.4). Функцията използва функцията `emptyRecord` за да създаде обект със стойност 0 за всички следени стойности. Итерираща през всички показвани доставки и за всяка доставка, на която дата на доставка отговаря на входния аргумент на функцията: инкрементира стойността на полето `“count”` на обекта `“data”`, съхраняващо броя на доставките за съответната дата, прибавя цената на доставката към стойността на полето `“revenue”` от същия обект, което се използва за отчитане на приходите. Ако статуса на поръчката е `“delivered”`, тоест тя е успешно доставена инкрементира стойността на полето `“delivered”`, пазещо тази информация. Следва итерация през всички посещения на приложението, като отново за всяко посещение съответстващо на проверяваната форматирана дата се инкрементира

стойността на полето “visits”.То отброява броя на уникалните посещения от всеки потребител за съответната дата и ако потребителя е разпознат при отчитането на посещениято инкрементира стойността на полето “users” от обекта data.

```
statisticData = (search) ->
  data = emptyRecord()
  for order in orders
    if order.orderDate is search
      data.count++
      data.revenue += order.price
      if order.status is "delivered" then data.delivered++
  for visit in ctrl.listOfVisits
    if visit.date is search
      data.visits++
      if visit.user then data.users++
  data
```

*(Фиг. 3.15.2.2.4) Кода на функцията statisticData, част от функцията init на контролера  
adminController*

### 3.15.3 Управление на потребители

Когато администратора избере третия таб се извиква метода “goToUsers” на контролера adminController, който променя състоянието на Интернет приложението на “home.admin.users”. Контролера на новото състояние прави заявка към сървъра на приложението за извличане на всички потребители от базата данни, сортирани спрямо стойността на полето admin в намаляващ ред. Шаблона на състоянието извежда адреса на електронната поща на всеки потребител в два бутона. Синия бутон позволява на дадения потребител да се промени вида на достъп, а червения изтрива потребителя. При натискането на червения бутон се отваря диалога за потвърждение на действието, който изисква потребителя да въведе своята парола и да я повтори с цел потвърждение

на изтриването. При натискане на синия бутон се използва услугата “changeAccessService”, която отваря диалог, който показва уникалния идентификационен номер на потребителя и поле, от което може да се избира вида на достъп на потребителя. При промяна на достъпа се използва услугата “confirmService”, за показването на диалог за потвърждение. Ако действието, което дадения администратор е предприел бъде успешно потвърдено се изпраща заявка до сървъра.

Заявката за промяна на достъпа е същата, която се използва за смяна на парола или електронна поща, единственото, което се различава са изпращаните данни. Винаги се изпраща обект с две ключови полета: “id” - уникалния идентификационен номер на потребителя и “user” - обект, на който комбинацията от полета и стойности, оказва как да се обнови информацията на потребителя.

Заявката за изтриване на потребител приема като аргумент символен низ, който представлява параметъра към заявката. Ако стойността му е “id” функцията посредница, отговаряща за параметъра на заявката добавя към обекта на заявката уникалния идентификационен номер на разпознатия потребител на приложението като стойност на ключово поле “userID”, в противен случай стойността на полето userID отговаря на параметъра на заявката. Всички останали функции, които се извикват при обработката на заявката за изтриване на потребителя използват стойността на полето userID (Фиг. 3.15.3.1). Първата от тези функции добавя масив към обекта на заявката със имената на всички файлове, които принадлежат на потребителя. Последната изтрива всички файлове на потребителя, а всички останали изтриват информацията за него от определена колекция от базата данни.

```
wipeFromCollection = (collection) ->
  (req, res, next) ->
    (collection.remove user: req.userID).exec().then ( -> next()),
  next

wipeFromDB = [
  (req, res, next) ->
    req.deletes = []
```

```

    check = (p) -> typeof p is "string" and p.length and p not in
req.deletes
    (models[0].find user: req.userID).exec()
      .then (docs) ->
        for doc in docs
          for layer in ["top", "bottom", "outline"]
            path = doc.files[layer]
            if check path then req.deletes.push path
          next()
      .catch next
]

```

```

wipeFromDB.push wipeFromCollection model for model in models

```

```

wipeFromDB.push (req, res, next) ->
  (userModel.remove _id: req.userID).exec().then (-> next()), next

```

```

wipeFromDB.push (req, res, next) ->
  deleted = (for file in req.deletes
    new Promise (resolve, reject) ->
      fs.unlink (join files, file), (err) ->
        if err then reject err else resolve())
  (Promise.all deleted).then (-> query res), next
  ...

```

```

delete: wipeFromDB

```

```

params: delete: name: "userID", callback: userIDParam

```

*(Фиг. 3.15.3.1) Част от файла /server/routes/user/userHandle.coffee, която изтирва потребители, при обработката на съответната заявка*

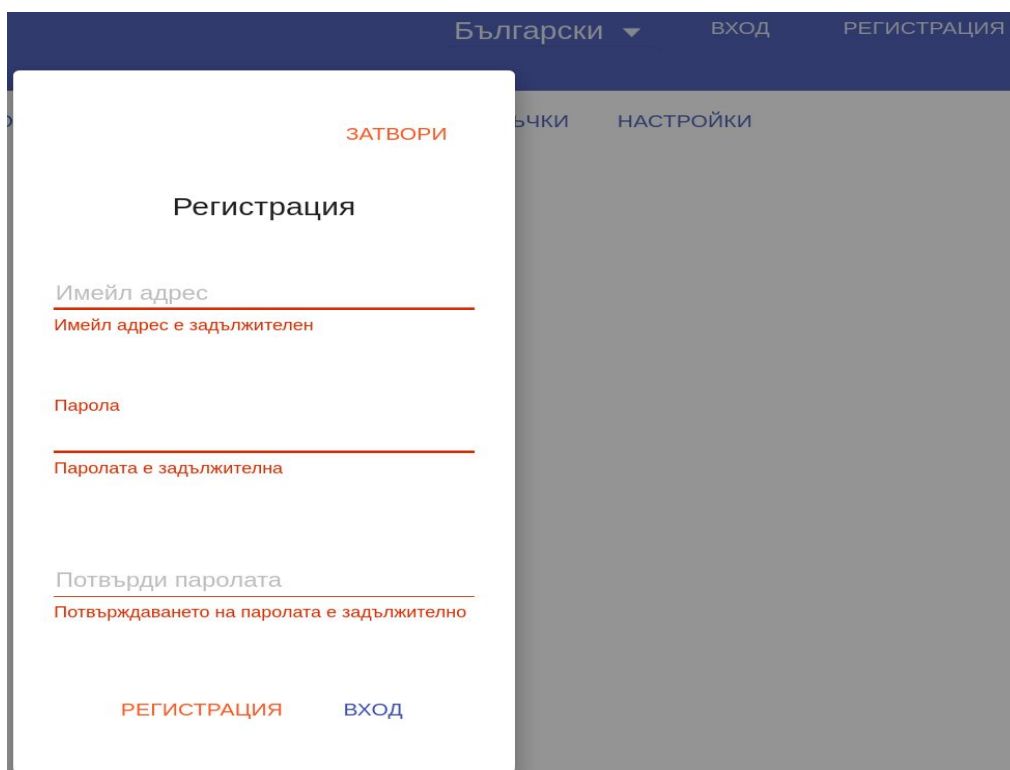
## Четвърта глава

### Ръководство на потребителя

Потребителите, които не са влезли в системата имат достъп само до първите три страници на приложението. За да влезе в системата даден потребител, той първо трябва да се регистрира. Единствено изключение е администратора, който се създава при стартиране на приложението.

#### 4.1 Регистрация

За да се регистрира дадения потребител, той трябва да натисне бутона с надпис РЕГИСТРАЦИЯ и правилно да попълни полетата с информация (Фиг. 4.1.1).



The image shows a registration dialog box overlaid on a blurred background of the application's main interface. The dialog box has a white background and a dark blue header bar. In the header bar, there is a dropdown menu labeled 'Български' with a downward arrow, and two buttons labeled 'ВХОД' and 'РЕГИСТРАЦИЯ'. The main content area of the dialog box is titled 'Регистрация' and contains three input fields. The first field is labeled 'Имейл адрес' and has a red error message below it: 'Имейл адрес е задължителен'. The second field is labeled 'Парола' and has a red error message below it: 'Паролата е задължителна'. The third field is labeled 'Потвърди паролата' and has a red error message below it: 'Потвърждаването на паролата е задължително'. At the bottom of the dialog box, there are two buttons: 'РЕГИСТРАЦИЯ' and 'ВХОД'.

(Фиг. 4.1.1) Диалог за регистриране

## 4.2 Вход

За да влезе в системата потребителя, след като успешно се е регистрирал, той трябва да попълни данните, с които се е регистрирал в диалога за вход (Фиг. 4.2.1). Този диалог може да се отвори при натискането на бутона ВХОД или при опита за отиване на страница, за която е нужно разпознаване.

The image shows a login dialog box. At the top, there is a red button labeled 'ЗАТВОРИ' (Close). Below it is the title 'Вход' (Login). There are two input fields: the first is labeled 'Имейл адрес' (Email address) and has a red error message 'Имейл адрес е задължителен' (Email address is required) below it; the second is labeled 'Парола' (Password) and has a red error message 'Паролата е задължителна' (Password is required) below it. Below the input fields is a checkbox with a blue checkmark and the text 'Запомни ме' (Remember me). At the bottom, there is a blue button labeled 'ВХОД' (Login).

(Фиг. 4.2.1) Диалог за вход

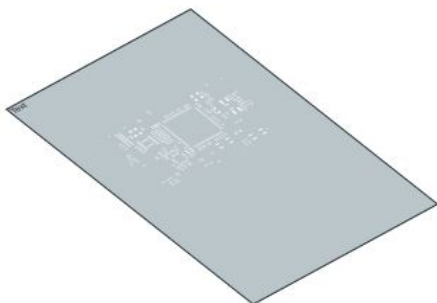
## 4.3 Създаване на нова поръчка

След като потребителя е влязал в системата, той може да създаде нова поръчка, като натисне бутона ПОРЪЧАЙ. За целта трябва да премине през всички състояния на Интернет приложението за създаване на нова поръчка.

### 4.3.1 Избиране на конфигурация за стенсил

Потребителя може да избере да използва някоя от вече създадените от него конфигурации за стенсил, или да натисне бутона с временен надпис “BUTTON-NEW-CONFIGURATION” и да попълни формата с информация за конфигурацията (Фиг. 4.3.1.1).

BUTTON-NEW-CONFIGURATION label-choose ▼



label-stencil-type ▼ label-stencil-transitioning ▼ label-stencil-impregnation ▼

label-fudical-marks ▼ label-fudical-marks-side ▼

label-text-position ▼ label-text-readable-from ▼ label-text-type ▼

label-position-alignent ▼ label-position-position ▼ label-position-side ▼

☒ checkbox-save-configuration

label-configuration-name

BUTTON-NEXT

(Фиг. 4.3.1.1) Избиране на конфигурация за стенсил

Ако потребителя реши да запази въведената информация, той може да го направи като остави маркирано полето с временен надпис “checkbox-save-configuration” и даде име на конфигурацията. След като потребителя е избрал своята конфигурация, той може да премине към състоянието за попълване на специфична за поръчката информация.

### 4.3.2 Попълване на специфична информация

Специфичната информация, която потребителя трябва да попълни са размера на стенсила и броя на центриращите маркери. Както и да качи файловете, необходими за изработване на поръчката и да попълни текста, който да бъде гравирен на всеки стенсил (Фиг. 4.3.2.1).

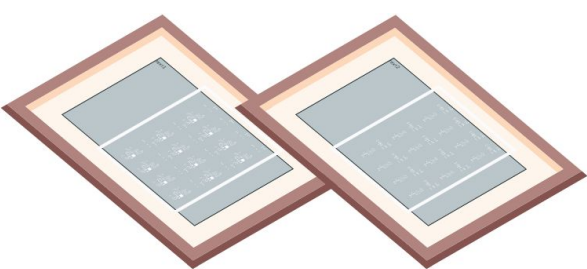





Diagram showing two stencils with dimensions and a configuration form.

Stencil dimensions:

- label-stencil-thickness: 120  $\mu$ m
- label-stencil-height: 300
- label-stencil-width: 200
- label-fudical-marks-number: 3

Form fields:

- layer-outline: Избран файл: (Клики на файла за да го премахнеш)  outline.pho
- layer-top: Избран файл: (Клики на файла за да го премахнеш)  top.pho
- layer-bottom: Избран файл: (Клики на файла за да го премахнеш)  bottom.pho
- label-text-top: текст1
- label-text-bottom: текст2

Buttons: BUTTON-BACK BUTTON-NEXT

(Фиг. 4.3.2.1) Попълване на специфична за поръчката информация и качване на Gerber файлове



Потребителя може да продължи към избора на конфигурация за адреси на доставката чрез натискане на синият бутон или да се върне назад чрез натискане на червеният бутон.

### 4.3.3 Избиране на конфигурация за адреси

Потребителя има същите възможности, както при избора на конфигурация за стенсил. За улеснение на потребителя, ако информацията от някое поле за доставка съвпада с друго, той може да маркира полето над него и по този начин полето с информация за адрес автоматично ще се попълни (Фиг. 4.3.3.1).

BUTTON-NEW-ADDRESSES

label-choose ▼

p-delivery

Държава

label-city

label-post-code

option-country-bulgaria ▼

option-city-sofia ▼

1000

label-address1

жк. Дружба 2, ул. Христо Войвода №14

label-address2

бл. 192, ет. 13

label-firstname

label-lastname

Кристиян

Хаджистоянов

☒ same-as-above

p-invoice

Държава

label-city

label-post-code

option-country-bulgaria ▼

option-city-sofia ▼

1000

label-address1

жк. Дружба 2, ул. Христо Войвода №14

label-address2

бл. 192, ет. 13

label-firstname

label-lastname

Кристиян

Хаджистоянов

(Фиг. 4.3.3.1) Част от изгледа за избиране на конфигурация за адреси на доставка

Потребителя отново разполага с два бутона за навигация - за продължаване към следващото състояние, което пресмята информацията на поръчката или за връщане към предходното.

#### 4.3.4 Преглед на изчислената информация

След като потребителя е попълнил цялата информация, необходима за създаване на поръчката, той може да види детайлна информация за изчислената цена на поръчката (Фиг. 4.3.4.1). Като отново разполага с два бутона за навигация.

header-price

**h-price-fudicals 0**

**h-price-size 131.78**

**h-price-apertures 0**

**h-price-text 0**

**h-price-glued 78.7**

**h-price-impregnation 58.68**

**h-price-total 269.16**

**BUTTON-BACK**

**BUTTON-NEXT**

(Фиг. 4.3.4.1) Преглед на изчислената цена

### 4.3.5 Финализиране на поръчката

Последното от състоянията на Интернет приложението за създаване на поръчка показва пълната информация на поръчката (Фиг. 4.3.5.1) . Като отново предоставя възможност за връщане на зад, но синия бутон създава поръчката, вместо да променя състоянието на приложението.

### 4.4 Преглед на направените поръчки

Всички направени поръчки могат да бъдат видяни от обикновените потребители, когато потребителя натисне бутона ПОРЪЧКИ. На (Фиг. 4.4.1) са показани поръчките на потребителя, като за втората има не видно известие.

Филтрирай

13/3/2016 13/3/2016

SPAN_ID	SPAN-STATUS	SPAN-PRICE	SPAN-ORDERDA	SPAN-SENDINGE	SPAN-ACTION
56E5CD52994C83115F6B2DF	SPAN-STATUS-NI	277	13/3/2016	13/3/2016	
56E5CD56994C83115F6B2DF	SPAN-STATUS-N	277	13/3/2016	13/3/2016	

(Фиг. 4.4.1) Преглед на направени поръчки

### 4.5 Настройки на профила

Когато потребителя натисне бутона НАСТРОЙКИ може да смени своята парола и адрес на електронна поща. Ако желае може дори да изтрие своя профил (Фиг. 4.5.1).

<div>layer-outline</div> <div>Клики за да сваляш файла</div> <div>outline.png</div>		<div>layer-top</div> <div>Клики за да сваляш файла</div> <div>top.png</div>		<div>layer-bottom</div> <div>Клики за да сваляш файла</div> <div>bottom.png</div>	
---	--	---	--	---	--

label-text-top		label-text-bottom	
Текст1		Текст2	

label-stencil-type	label-stencil-transferring	label-stencil-frame-size	label-stencil-frame-clean	label-stencil-impregnation
option-sm-d-stencil	* Загледана в рамка	* 750 x 750	* option-with-clean	* option-with-impregnation
label-lutcal-marks		label-lutcal-marks-side		
option-cut-through		* option-pcb-side		
label-text-position	label-text-readable-from	label-nox-type		
option-top-right	* option-right	* option-drilled		
label-position-alignmen	label-position-position	label-position-side		
option-landscape	* option-layout-centered	* option-pcb-side		
label-stencil-thickness	label-stencil-height	label-stencil-width	label-lutcal-marks-number	
120 µm	* 300	200	3	

p-delivery		
Държава	label-city	label-post-code
option-country-bulgaria	* option-city-sofia	* 1000
label-address1		
жк. Дружба 2, ул. Христа Войвода №14		
label-address2		
бл. 182, ет. 13		
label-firstname	label-lastname	
Кристиан	Хаджистоянов	

p-invoice		
Държава	label-city	label-post-code
option-country-bulgaria	* option-city-sofia	* 1000
label-address1		
жк. Дружба 2, ул. Христа Войвода №14		
label-address2		
бл. 182, ет. 13		
label-firstname	label-lastname	
Кристиан	Хаджистоянов	

p-firm		
Държава	label-city	label-post-code
option-country-bulgaria	* option-city-sofia	* 1000
label-address1		
жк. Дружба 2, ул. Христа Войвода №14		
label-address2		
бл. 182, ет. 13		
label-firstname	label-lastname	
Кристиан	Хаджистоянов	

(Фиг. 4.3.5.1) Част от изгледа за преглед на поръчка, преди да бъде завършена

Нов имейл адрес

ПРОМЕНИ ИМЕЙЛ АДРЕСА

Нова парола

Паролата е задължителна

ПРОМЕНИ ПАРОЛАТА

BUTTON-REMOVE-ACCOUNT

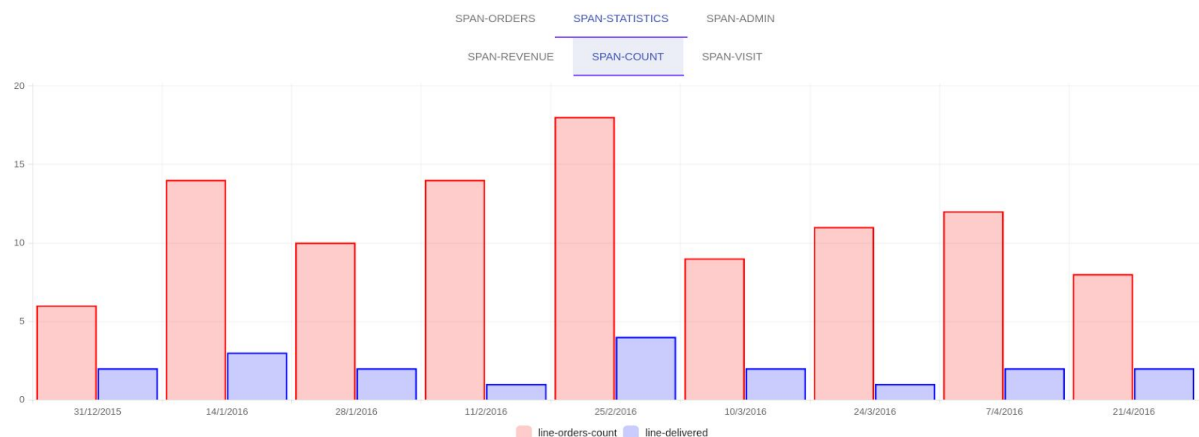
*(Фиг. 4.5.1) Изглед на настройки на профила*

## 4.6 Администрация

Администраторите могат да преглеждат изготвяните статистики за избран от тях времеви интервал и да управляват потребителите на Интернет приложението.

### 4.6.1 Преглед на статистики

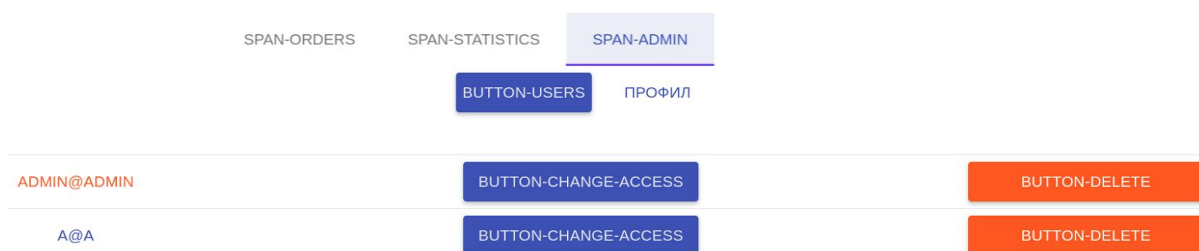
Администраторите могат да видят статистиките, чрез натискането на бутона с временен текст “SPAN-STATISTICS”. Могат да променят интервала от време, за който се изготвят графиките, като променят датите, за които се показват поръчки. На (Фиг. 4.6.1.1) е показана графиката на общия брой поръчки и на доставените поръчки.



(Фиг. 4.6.1.1) Създадена графика на броя поръчки

## 4.6.2 Управление на потребители

При натискане на последния бутон от горното навигациона лента, администраторите могат да отидат на страницата за управление на потребителите (Фиг. 4.6.2). От там те могат да променят правата на достъп на всеки потребител или да изтрият някой.



(Фиг. 4.6.1.2) Изглед на управление на потребители

## Заклучение

С разработката на дипломната работа бе реализирано Интернет приложение, което позволява да се създават и управляват конфигурации за стенсил и адресна информация, поръчване и следене на промяната на статуса на поръчките и изготвяне на статистика. Не бе имплементирана интеграция на плащания с “Paypal” и “Stripes”, за това тя е първото нещо, което да бъде добавено към приложението за неговото бъдещо развитие. Останалите неща, които са включени в бъдещото развитие на проекта са:

- Добавяне на страниците, за които не се изисква разпознаване от системата, защото към момента на описване на разработката всяка от тях представлява символен низ;
- Добавяне на обекти, описващи интернационализацията на приложението за всички шаблони, за които липсват;
- Добавянето на директиви за постигане на реагиращ дизайн на всякъде където липсват;
- Добавяне на шаблони за описание на статусите, за които липсват;
- Имплементация на ограничение на достъп на администраторите, спрямо техният вид достъп
- Добавяне на функционалност за автоматизирано изпращане на електронни писма, за потвърждаване на направена регистрация, направа на поръчка и промяна на статуса на поръчките.

## Използвана литература

1. Интернет страница на фирмата Ucamco, <https://www.ucamco.com/en/>
2. Ucamco, The Gerber Format Specification, Revision 2015.09,  
[https://www.ucamco.com/files/downloads/file/81/the\\_gerber\\_file\\_format\\_specification.pdf](https://www.ucamco.com/files/downloads/file/81/the_gerber_file_format_specification.pdf),  
2015
3. Интернет приложение за поръчване на лазерно рязани стенсили на фирма WEdirekt, <http://www.wedirekt.de/en/stencil>
4. Интернет приложение за поръчване на лазерно рязани стенсили на фирма multi-cb, <https://portal.multi-circuit-boards.eu/Customr/Calculator/>
5. Интернет приложение за поръчване на лазерно рязани стенсили на фирма Leit-On, <http://www.leiton.de/en-kalkulation.html>
6. Getting Started with Rails, [http://guides.rubyonrails.org/getting\\_started.html](http://guides.rubyonrails.org/getting_started.html)
7. Ruby Core Reference, <http://ruby-doc.org/core-2.3.0/>
8. Getting started with Django, <https://www.djangoproject.com/start/>
9. The Python Language Reference, <https://docs.python.org/3/reference/index.html>
10. Basic Task List, <https://laravel.com/docs/5.2/quickstart>
11. PHP Manual, <https://secure.php.net/manual/en/>
12. Documentation, <https://kore.io/doc/>
13. C++ reference, <http://en.cppreference.com/w/>
14. Node.js v5.x Documentation, <https://nodejs.org/dist/latest-v5.x/docs/api/>
15. JavaScript reference,  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>
16. Christopher Buecheler, THE DEAD-SIMPLE STEP-BY-STEP GUIDE FOR FRONT-END DEVELOPERS TO GETTING UP AND RUNNING WITH NODE.JS, EXPRESS, JADE, AND MONGODB,  
<http://cwbuecheler.com/web/tutorials/2013/node-express-mongo/>, 2013



17. Christopher Buecheler, CREATING A SIMPLE RESTFUL WEB APP WITH NODE.JS, EXPRESS, AND MONGODB, <http://cwbuecheler.com/web/tutorials/2014/restful-web-app-node-express-mongodb/>, 2014
18. Chris Sevilleja, Easily Develop Node.js and MongoDB Apps with Mongoose, <https://scotch.io/tutorials/using-mongoosejs-in-node-js-and-mongodb-applications>, 2015
19. Express 4.x API reference, <http://expressjs.com/en/4x/api.html>
20. The MongoDB 3.2 Manual, <https://docs.mongodb.org/manual/>
21. Node.js Stream addon code, [https://github.com/nodejs/node/blob/master/src/js\\_stream.cc](https://github.com/nodejs/node/blob/master/src/js_stream.cc)
22. Node.js Buffer addon code, [https://github.com/nodejs/node/blob/master/src/node\\_buffer.cc](https://github.com/nodejs/node/blob/master/src/node_buffer.cc)
23. Интернет страница на npm, <https://www.npmjs.com/>
24. Specifics of npm's package.json handling, <https://docs.npmjs.com/files/package.json>
25. Browserify Handbook, <https://github.com/substack/browserify-handbook>
26. Hand-picked registry of Node.js frameworks, <http://nodeframework.com/>
27. Specification Version 1.0, <http://bsonspec.org/spec.html>
28. Mongoose guide, <http://mongoosejs.com/docs/guide.html>
29. AngularJS Developer Guide, <https://docs.angularjs.org/guide>
30. AngularJS API Docs, <https://docs.angularjs.org/api>
31. 5 MIN QUICKSTART, <https://angular.io/docs/ts/latest/quickstart.html>
32. Variable naming, <http://javascript.info/draft/variable-naming>
33. Todd Motto, Dynamic Controllers in Directives with the undocumented "name" property, <https://toddmotto.com/dynamic-controllers-in-directives-with-the-undocumented-name-property/>
34. Bootstrap Getting started, <http://getbootstrap.com/getting-started/>
35. Semantic-UI Getting Started, <http://semantic-ui.com/introduction/getting-started.html>
36. Foundation Getting Started, <http://foundation.zurb.com/apps/docs/#!/>
37. Angular-Material Getting Started, <https://material.angularjs.org/latest/getting-started>
38. UI-Bootstrap Getting started, [https://angular-ui.github.io/bootstrap/#/getting\\_started](https://angular-ui.github.io/bootstrap/#/getting_started)

- 39. Semantic-UI-Angular, <https://github.com/Semantic-Org/Semantic-UI-Angular>
- 40. Foundation Angular, <http://foundation.zurb.com/apps/docs/#!/angular>
- 41. Material design Introduction,  
<https://www.google.com/design/spec/material-design/introduction.html#>
- 42. Angular module ngAnimate reference, <https://docs.angularjs.org/api/ngAnimate>
- 43. A Complete Guide to Flexbox, <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>
- 44. Angular module ngRoute reference, <https://docs.angularjs.org/api/ngRoute>
- 45. UI-Router In-Depth Guide, <https://github.com/angular-ui/ui-router/wiki>
- 46. Sass Reference, [http://sass-lang.com/documentation/file.SASS\\_REFERENCE.html](http://sass-lang.com/documentation/file.SASS_REFERENCE.html)
- 47. Features of the Less language, <http://lesscss.org/features/>
- 48. EXPRESSIVE, DYNAMIC, ROBUST CSS, <http://stylus-lang.com/>
- 49. Jade Language Reference, <http://jade-lang.com/reference/> List of languages that  
compile to JS,  
<https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS>
- 50. CoffeScript Language Reference, <http://coffeescript.org/>
- 51. Alex Allain, Makefiles, <http://www.cprogramming.com/tutorial/makefiles.html>
- 52. Rake - Ruby Make, <http://rake.rubyforge.org/>
- 53. Jake Documentation, <http://jakejs.com/docs>
- 54. Grunt Getting started, <http://gruntjs.com/getting-started>
- 55. Gulp Getting Started,  
<https://github.com/gulpjs/gulp/blob/master/docs/getting-started.md>
- 56. gulp API docs, <https://github.com/gulpjs/gulp/blob/master/docs/API.md>
- 57. SMT-Stencils\_resources repository,  
[https://github.com/IvasTech/SMT-Stencils\\_resources](https://github.com/IvasTech/SMT-Stencils_resources)
- 58. Angular translate Guide, <http://angular-translate.github.io/docs/#/guide>

# Съдържание

Увод .....	3
ПЪРВА ГЛАВА Преглед на съществуващи конкуретни продукти и запознаване с бизнес модела на фирма "IvasTech" .....	5
1.1. Преглед на съществуващи конкуретни продукти .....	5
1.1.1 Преглед на приложението на фирма "WURTH ELEKTRONIK" (WEdirekt) .	5
1.1.2 Преглед на приложението на фирма "multi-cb" .....	7
1.1.3 Преглед на приложението на фирма LeitOn .....	8
1.2 Проучване и запознаване с бизнес модела на фирма "IvasTech" .....	9
ВТОРА ГЛАВА Преглед на избраните технологии и аргументация за техния избор .....	12
2.1 Избор на сървърната технология "Node.js" .....	12
2.2 Избор на фреймуърк към Node.js - "express.js" .....	13
2.3 Избор на базата данни "MongoDB" .....	15
2.3.1 Избор на драйвера "Mongoose" за връзка с базата данни .....	17
2.4 Избор на JavaScript фреймуърка "Angular.js" .....	18
2.5 Избор на CSS фреймуърка "Angular-Material" .....	26
2.6 Избор на "UI-Router" като технология за виртуализация на страниците на приложението .....	27
2.7 Избор на "Stylus" за CSS препроцесор .....	28
2.8 Избор на HTML препроцесора "Jade" .....	29
2.9 Избор на езика "Coffee-Script" .....	30
2.10 Избор на изграждаща система "Gulp" .....	30
ТРЕТА ГЛАВА Описание на разработения продукт .....	31
3.1 Алгоритъм на изграждане на приложението .....	31
3.2 Комуникация между клиент и сървър на приложението .....	34
3.2.1 Динамично изграждане на възможните заявки с цел премахване на повторения и лесен начин за добавянето на нови .....	34

3.2.1.1 Избягване на повторения във файловете описващи заявките, които сървъра може да приеме .....	35
3.2.1.2 Избягване на повторения при описването на заявките, които клиента на приложението може да направи към сървъра чрез тяхното описване .....	35
3.2.1.3 Заключение и обяснение как се постига лесното бъдещо разширение на заявки .....	37
3.2.2 Описание на услугите използвани за връзка между клиент и сървър .....	37
3.2.2.1 Описание на услугата "RESTService", използвана за подаването на заявки, които не включват прехвърляне на файлове, към сървъра .....	37
3.2.2.2 Описание на услугата "UploadService" използвана за изпращането на заявки, за прехвърляне на файлове към сървъра .....	38
3.3 Обработка на грешки при връзката клиент - сървър и тяхното съхранение на сървъра .....	40
3.3.1 Обработка на грешки настъпили при клиента .....	40
3.3.2 Обработка на грешки настъпили при обработката на заявки от сървъра ...	41
3.4 Обработка на грешки настъпили в клиентската част на приложението (не настъпили в резултат на отправяне на заявка към сървъра) .....	44
3.5 Премахване на паразитни повторения във файловете използвани за създаването на Mongoose модели .....	44
3.6 Услугата "showDialogService" използвана за създаване и управление на диалозите, които Интернет приложението използва .....	45
3.6.1 Демонстрация на услугата "showDialogService", чрез описване начина на работа на услугата "loginService" .....	47
3.7 Навигация между виртуалните страници на приложението чрез директивата "ivstStateSwitcher" .....	49
3.8 Интернационализация на Интернет приложението .....	53
3.9 Интернет приложението, след като премине през процеса "Bootstrap" .....	54
3.10 Разпознаване на потребители на системата (Аутентикация) .....	57
3.10.1 Разпознаване на потребители от Интернет приложението .....	58
3.10.2 Разпознаване на потребители от сървъра на приложението .....	59
3.10.2.1 Създаване и управление на сесии .....	59

3.10.2.2 Заявка за проверка на разпознаване от системата .....	62
3.11 Вход и изход в системата .....	64
3.11.1 Регистрация на нови потребители .....	64
3.11.2 Вход в системата .....	66
3.11.3 Изход от системата .....	66
3.12 Преизползване на MVC .....	67
3.12.1 Преизползване на модели .....	67
3.12.1.1 Преизползване на модели чрез услугата "scopeControllerService" .....	67
3.12.1.2 Преизползване на модели чрез услугата "progressService" .....	68
3.12.2 Преизползване на изгледи .....	70
3.12.2.1 Преизползване на изгледи чрез директивата "ivstIncludeDirective" .....	71
3.12.3 Преизползване на контролери .....	72
3.13 Имплементация на функционалността за създаване на поръчки .....	73
3.13.1 Преизползване на код за елиминнирането на повторения в кода на конфигурациите за стенсил и адреси за доставка .....	73
3.13.1.1 Премахване на повторения в сървъра на приложението .....	74
3.13.1.2 Отстраняване на повторения в Интернет приложението .....	75
3.13.2 Избиране на конфигурация за стенсил .....	78
3.13.2.1 Синхронизация между директивите ivstConfigurationInfo и ivstStencilPreview .....	78
3.13.2.1.1 Директивата "ivstStencilPreview" .....	79
3.13.2.1.2 Директивата "ivstConfigurationInfo" .....	80
3.13.2.2 Контролера "configurationInterface" .....	81
3.13.2.3 Избиране на конфигурация за стенсил .....	82
3.13.3 Качване на Gerber файлове от потребителя и рендирането им в SVG шаблони с цел визуализацията им .....	83
3.13.3.1 Качване на Gerber файлове за визуализирането им в процеса на поръчване .....	84
3.13.3.1.1 Директивата "ivstFileContainer" .....	84
3.13.3.2 Рендиране на Gerber файлове, чрез превръщането им в SVG шаблони ..	89
3.13.3.2.1 Получаване и съхранение на Gerber файлове от сървъра на приложението .....	89

3.13.3.2.1.1 Конфигуриране на multer за начина на съхраняване на файлове .....	89
3.13.3.2.2 Обработка на получени Gerber файлове, чрез превръщането им в SVG шаблони .....	90
3.13.3.2.2.1 Превръщане на Gerber файлове в SVG шаблони .....	91
3.13.4 Избор на конфигурация за адреси на доставката .....	97
3.13.5 Пресмятане на цената на поръчката .....	98
3.13.6 Преглед на поръчката и завършването ѝ .....	100
3.14 Преглед на направените поръчки, филтриране на показваните поръчки и следене на промяна на статуса на поръчките .....	101
3.14.1 Преглед на направени поръчки .....	101
3.14.2 Филтриране на показваните поръчки .....	105
3.14.3 Следене на промяна в статуса на поръчките .....	108
3.15 Администрация на приложението .....	109
3.15.1 Промяна на статуса на поръчка .....	111
3.15.2 Преглед и изготвяне на статистики .....	114
3.15.2.1 Преглед на статистики .....	115
3.15.2.2 Изготвяне на статистики .....	116
3.15.3 Управление на потребители .....	121
Четвърта глава Ръководство на потребителя .....	124
4.1 Регистрация .....	124
4.2 Вход .....	125
4.3 Създаване на нова поръчка .....	125
4.3.1 Избиране на конфигурация за стенсил .....	126
4.3.2 Попълване на специфична информация .....	127
4.3.3 Избиране на конфигурация за адреси .....	128
4.3.4 Преглед на изчислената информация .....	129
4.3.5 Финализиране на поръчката .....	130
4.4 Преглед на направените поръчки .....	130
4.5 Настройки на профила .....	130
4.6 Администрация .....	132

<b>4.6.1 Преглед на статистики .....</b>	<b>132</b>
<b>4.6.2 Управление на потребители .....</b>	<b>133</b>
<b>Заключение.....</b>	<b>134</b>
<b>Използвана литература .....</b>	<b>135</b>
<b>Съдържание .....</b>	<b>138</b>