**РУСЕНСКИ УНИВЕРСИТЕТ "АНГЕЛ КЪНЧЕВ"**

Факултет: Природни науки и образование
Катедра: Информатика и информационни технологии

**ДОПУСКАМ ДО ЗАЩИТА**

Ръководител на специали-
зиращата катедра:……..……
(доц. д-р Румен Русев)

# ОБЯСНИТЕЛНА ЗАПИСКА НА ДИПЛОМЕН ПРОЕКТ

**Тема**: **Уеб приложение за управление на лични финанси**

**Роли**: **Дизайн мениджър, тест дизайнер, тестер, технически документатор**

Дипломант: ……
(Д. Трифонов)

**ОДОБРЯВАМ**
Ръководител:.....
(гл. ас. д-р. В. Великов )

Русе, 2020 г.

# РУСЕНСКИ УНИВЕРСИТЕТ “АНГЕЛ КЪНЧЕВ”

Факултет: **Природни науки и образование**

Катедра: **Информатика и информационни технологии**

**РЕГИСТРИРАЛ:**
Ст. инспектор:
/ М. Чолакова/
...../....., 20__ г.

**УТВЪРЖДАВАМ:**
Ръководител на катедра ИИТ:
/доц. д-р Р. Русев/

# З А Д А Н И Е
# ЗА
# ДИПЛОМЕН ПРОЕКТ

на студента: Димитър Валериев Трифонов, Фак. No: 186402
Специалност: **Софтуерно инженерство**
ОКС: **Магистър**

1. Тема:
   Уеб приложение за управление на лични финанси: дизайн мениджър, тест дизайнер, тестер, технически документатор
2. Изходни данни:
   Java, Eclipse, Maven, Spring Boot, Git, Confluence, BitBucket, Jira
3. Съдържание на обяснителната записка:
   - Описание на проекта и изпълняваните в него роли
   - Свършена работа по проекта
   - Продукти от свършената работа
   - Възникнали проблеми и взети мерки за решаването им
   - Визия за бъдещето на проекта

Срок за предаване на дипломния проект: 15.06.2020

Зададено на: 01.11.2019

**Дипломант: ……**
/ Д. Трифонов /

**Ръководител:**
/ гл. ас. д-р. В. Великов /

# Съдържание

# 1. Описание на проекта и изпълняваните в него роли

Проектът представлява приложение за управление на лични финанси (уеб приложение и мобилно приложение). Накратко избрахме да го наричаме “PFM“ (от “Personal Finance Manager” – тоест мениджър на лични финанси).

Целта на проекта беше да реализираме уеб система, която позволява на един обикновен човек да управлява парите си по-ефективно. По-конкретно да позволи на един потребител да отчита колко пари има по различни сметки, да отчита своите приходи и разходи по тези сметки, да категоризира и ограничава тези приходи/разходи и още.

Открихме, че вече има съществуващи такива приложения, които всъщност вършат работа, но всяко от тях си има по някой минус. Някои бяха само на английски, други изискваха да добавиш реална банкова сметка, за да работят, трети всъщност бяха прекалено сложни (имаха прекалено много функционалност за някой, който просто иска бързо и лесно да управлява транзакциите си).

В крайна сметка решихме да си поставим задачата да създадем наше приложение, което е на български и позволява на един потребител да изпълнява основни операции за отчитане на парите си, за да може благодарение на това отчитане да взема по-добри финансови решения.

Идеята ни за „отличителна черта“ на приложението беше да добавим възможност за поставяне на „цел по сметка“ (сума, която потребителят иска да стигне по дадена сметка) – при поставяне на цел системата да изчислява прогнозен период, в който е реалистично да бъде осъществена тази цел (като това изчисление да бъде на база на активността по избраната сметка в миналото).

Планът ни не беше да разработим някое революционно и невиждано досега приложение, а по-скоро да разработим приложение, което е достатъчно предизвикателно, за да ни помогне да стигнем следващото ниво като софтуерни инженери. Желаехме нещо, което ще ни предизвика от гледна точка на имплементация, документация и работа в екип.

В началото на разработката на проекта разпределихме ролите си според силните страни и желанията на всеки член от екипа. Водихме се по RUP методологията и от там произлизат наименованията на ролите, които аз поех. Тези роли включват:

1. Design Manager (дизайн мениджър)
2. Test Designer (тест дизайнер)
3. Tester (тестер)
4. Technical Writer (технически документатор)

В следващата глава са описани извършените от мен дейности по различните ми роли.

# 2. Свършена работа по проекта

## *2.1. Като „Design Manager"*

### 2.1.1. Дизайн и разработка на back-end на системата

Самата система се дели на 3 програмни проекта: back-end проект (включва слоя на данните и бизнес логиката по възможните операции в системата), front-end проект (включва слоя на потребителския интерфейс на системата) и проект за мобилното приложение (като трите проекта комуникират помежду си чрез HTTP REST заявки).

Аз разработих преобладаващата част от *back-end* проекта на системата. Това включва следните дейности (категоризирани според различните елементи от разработката на проекта):

- Класове за описание на данни – корекции, подобрения и валидация
- Класове за REST услуги – проектиране и имплементация
- Класове за бизнес логика – проектиране и имплементация
- Класове за защита – настройване, корекции и подобрения
- Управляване на версиите на проекта – работа с Git и Bitbucket.

По **описанието на данните** коригирах и подобрих първоначалните Java entity класове, които бяха добавени от другите разработчици. Това включваше добавяне, премахване,

преименуване на полета, поправка на JPA анотации (за правилно генериране на таблици в БД от Java класовете), добавяне на валидация за данните и добавяне на допълнителни DTO класове за ползване от класовете, които реализират REST услугите.

По **REST услугите** - създадох Java класовете, които реализират REST API, което се използва от front-end проекта и мобилното приложение за изпълнението на различните възможни операции в системата (начинът му на работа е подробно обяснен в следващата глава от записката).

По-конкретно измислих и имплементирах това какви заявки да може да праща клиентът (какви „endpoints” или “изходни точки“ да има в API-то) и как да бъдат обработени от back-end тези заявки.

Например зададох клиентът да може да праща GET заявка до „/accounts”, POST заявка до „/categories”, PUT заявка до „/transactions/{id}” и още (като при някои заявки да трябва да се подадат допълнителни данни в тялото или в URL адреса на заявката),. След това, когато клиентът изпрати такава заявка, определих какво точно да се случи в back-end (да се получи заявката, да се провери дали подадените данни в нея са валидни и накрая да се върнат някакви полезни за клиента данни като отговор на заявката).

По **бизнес логиката** - създадох Java класовете, които реализират бизнес логиката на системата. Това са класовете, които определят какво се случва „зад кулисите“ с данните при изпълнението на различните операции, които системата позволява. По-конкретно измислих и имплементирах това какво трябва да се случва в базата данни при изпълнението на различните операции (например при регистрация, при добавяне на сметка и още).

Част от бизнес логиката беше предварително определена в създадените user story-та, но липсваха някои детайли за желаната функционалност, които трябваше сам да реша как да реализирам. Например за да е изпълнима операция „Синхронизиране на наличност“, имаше нужда да се изпълни служебна транзакция, но не беше напълно ясно как да стане това. За целта реших при регистрация на потребителя да му създам

системна приходна и разходна категория, които да се използват за извършването на тази транзакция и съответно за извършването на синхронизацията.

Списъкът с имплементираните от мен функционалности може да бъде открит в следващата глава на записката (в описанието на документ „Дизайн модел“, където е написано подробно какво и как е реализирано).

Някои от функционалностите, които реализирах в back-end проекта, НЕ СА реализирани във front-end проекта (ползваеми са чрез HTTP REST заявки към back-end, но липсват като възможни операции в потребителския интерфейс). Причината за това е обяснена в главата от записката за възникнали проблеми по време на разработката на проекта.

По **защитата** – настроих първоначалните добавени класове за защита на приложението, коригирах някои грешки по тях и направих някои подобрения.

По-конкретно зададох това кои ресурси да са защитени, кой какъв достъп да има до различните ресурси и на какъв принцип да се задават правата на даден потребител (първоначалните класове за защита работеха с потребители добавени във файл, а аз ги промених да работят с регистрираните потребители в системата, тоест тези в базата данни). Също преработих част от кода за JWT автентикация (направих refactoring) и добавих anti-bruteforce защита, която е описана в следващата глава.

По **управлението на версиите на проекта** работих с Git и Bitbucket. Това включваше:

- добавяне на промени по кода в локално и remote хранилище (чрез Git Bash и SourceTree);
- създаване на клонове (branches) за имплементацията на различните функционалности и отстраняването на бъгове;
- rebase & squash на множество запазени промени (commits) в даден клон за максимално лесни кодови ревюта от другите разработчици;
- пускане на pull requests със завършени функционалности / премахнати бъгове;
- съобщаване на другите разработчици за новости по кода и проблеми свързани с хранилището.

### 2.1.2. Документация на back-end на системата

Аз реализирах 99% от документацията на back-end проекта на системата (останалият един процент включва коментарите на първоначалните класове, които бяха добавени от другите разработчици). Документацията се намира на две места:

- в самия код
- в документа „Дизайн модел“

Документацията на кода включва добавени от мен Java коментари за всеки клас и метод в тях. Коментарите описват механизма на работа на различните класове и методите в тях, а също така и предназначението им, какво означават различните параметри, анотации, полета и т.н. Освен това се постарах кодът да е максимално „самодокументиращ се“ (самото му прочитане да е достатъчно, за да може някой да разбере как работи) – за целта използвах описателни имена на променливи, константи, методи и класове.

Документацията, която добавих в „Дизайн модел“ съдържа цялостен поглед върху начина на работа на back-end проекта (какви класове са включени, какво е предназначението им, как са реализирани различните функционалности и защо са реализирани така). Подробно описание на този документ има в следващата глава на записката.

### 2.1.3. Обратна връзка по разработката на front-end

Не съм участвал директно в *разработката* на front-end проекта. Участието ми там беше в *предоставяне на обратна връзка* на front-end разработчиците. Това най-вече означава съобщаване за бъгове при интеграцията с back-end проекта и предложения за отстраняването им (например случай, при който front-end се опитва да изпрати заявка до back-end, но тази заявка не е в правилен формат и затова отговорът й не е очаквания – забелязах грешката, погледнах какво я предизвиква и съобщих на front-end разработчиците, а също им предложих и решение).

## *2.2. Като „Test Designer”*

Като тест дизайнер реализирах дизайна и поддръжката на тестовите сценарии за различните функционалности на системата.

Категоризирах тестовите сценарии според функционалностите на системата и за всяка от тях добавих по няколко сценария (поредици от стъпки, които да се изпълнят при тестване) под формата на таблици. При промяна в изискванията за реализирането на дадена функционалност, обновявах и тестовия модел, за да е актуален.

Тестовите сценарии са добавени в документа „Тестов модел“, който е описан в следващата глава на записката.

## *2.3. Като „Tester”*

Като тестер основният ми принос беше, че имплементирах unit & интеграционни тестове за back-end на системата. Общо имплементирах 390 теста, които са част от back-end проекта (достъпни са в Bitbucket хранилището). Тестовете са направени чрез JUnit и тестват различните функционалности на системата (в следващата глава е описано какво точно е тествано).

Тествах частично и front-end проекта заедно с мобилното приложение. Не съм писал автоматизирани тестове за тях, а просто следях дали интеграцията между проектите е успешна (тоест дали заедно работят правилно). При забелязването на някой проблем съобщавах на другите разработчици.

## *2.4. Като „Technical Writer”*

Като технически документатор направих следното:

- Написах упътване за ползване на системата (документ *„Материали за обучение“*);
- Формирах списък с термини в проекта и техните дефиниции (документ *„Речник“*);
- Написах упътване за ползване на Confluence (документ *„Примерен документ (+ някои насоки за Confluence)”*)

Първите два споменати документа са разгледани в следващата глава на записката, а последният представлява просто кратко ръководство за това как се създават и оформят документи в Confluence (написах го за улеснение на екипа ми, защото в началото не бяхме запознати с този инструмент).

### *2.5. Допълнително свършена работа*

Успях да свърша и малко допълнителна работа, която не беше директно свързана с ролите ми в проекта:

- **Обратна връзка по документи на другите членове от екипа** – това включваше съобщаване за липсваща / неточна / неясна информация в документацията на другите, откриване на правописни грешки, грешки във форматирането и други. Направих това чрез добавяне на Confluence коментари в документите и чрез директна комуникация с останалите членове на екипа.
- **Документиране в Jira на изпълнени задачи по проекта** – включва добавяне и отчитане на това, което съм свършил и върху което в момента работя по проекта. Целта ми с това беше да улесня постигането на синхрон в екипа - тоест да може всеки да знае какво правя в момента и да планира по-добре собствените си задачи, ако по някакъв начин те зависят от моята свършена работа. Също исках да допринеса за по-детайлната документация на проекта (да е ясно кое кога е свършено, за колко време и така нататък).
- **Преглед и ревюта на код добавен от другите разработчици** – това включваше преглед на pull requests направени от други разработчици в Bitbucket и оставяне на обратна връзка за кода, който са добавили (коментари с информация за това, което трябва да бъде поправено, или одобряване на кода за сливане с главния Git клон на съответния проект).
- **Участие във формирането на софтуерните изисквания** – участвах в уточняването на изискванията за проекта и по-точно в дискусиите за това какво трябва да съдържа системата, как трябва да работи тя, кои функционалности са най-важни, кои са с по-нисък приоритет и т.н.

## 3. Продукти от свършената работа

Крайният резултат от работата ми по проекта включва:

- **документи**
- **програмен код**

Документите ми са качени на Confluence страницата на проекта:

1. Дизайн модел - http://mse2019.ami.uni-ruse.bg:8081/pages/viewpage.action?pageId=3211301
2. Тестов модел - http://mse2019.ami.uni-ruse.bg:8081/pages/viewpage.action?pageId=3211291
3. Речник - http://mse2019.ami.uni-ruse.bg:8081/pages/viewpage.action?pageId=1605661
4. Материали за обучение - http://mse2019.ami.uni-ruse.bg:8081/pages/viewpage.action?pageId=3211287

Кодът ми е качен на Bitbucket хранилището на проекта:

- http://mse2019.ami.uni-ruse.bg:8082/projects/PFM/repos/backend_v2/browse - последна версия (в отделно хранилище заради технически проблеми с първоначалното хранилище)
- http://mse2019.ami.uni-ruse.bg:8082/projects/PFM/repos/backend/browse - предишни версии (включва и всички клонове, които са били създадени по време на разработката)

В тази глава представям крайния резултат от дейностите, които съм извършил по проекта (като от документите включвам най-съществения, а останалото е достъпно на горните адреси).

## *3.1. Документация*

### *3.1.1. Дизайн модел*

В този документ описах механизма на работа на back-end проекта (как и защо е направено това, което е направено). За всяка функционалност от проекта е включено описание, придружено от клас диаграми и sequence диаграми (диаграмите не са включени в записката, за да се спазят ограниченията за обема ѝ)

По време на защитата на Практикум 3 получих забележката, че ако диаграмите са направени след имплементацията, това не е правилно от гледна точка на дизайн (по-добре е първо да се направят диаграми, на база на които да се пише код след това).

Разбирам и виждам смисъла в това, но реших да добавя диаграмите след имплементацията поради следната причина. Считам диаграмите за инструмент, който има предназначението да улесни разработчиците на системата, за да може всички да знаят към какво се стремят и по-лесно да свършат работата си, да не си пречат и т.н. Тъй като аз бях единствения back-end разработчик, създаването на такива диаграми преди имплементация според мен не би улеснило или ускорило процеса на разработка, а точно обратното – би го усложнило и забавило излишно.

Реших, че предварителното създаване на диаграми не би било полезно в този случай (макар да разбирам, че в друг контекст може да е много полезно – например когато се започва работа по наистина сложен проект и участват много разработчици в него). Прецених, че ще е по-ефективно да имплементирам необходимото и след това да документирам това, което съм създал, за бъдещи разработчици, на които може да им се наложи да работят с моя код.

Ето и най-съществените части от документа:

3.1.1.1. ***Дизайн на back-end проекта***

Проектът се състои от Java пакети. Главният пакет се казва "*com.mse.personal.finance*", а останалите като "*db.entity*", "*security*" и "*service*" са подпакети на главния. Разделението на пакети се използва, за да може ясно да се разграничат различните компоненти на проекта и да се улесни разработката и поддръжката му. Пакетите съдържат в себе си Java класове. Тези класове се използват за:

- **Описание на данните в приложението** - наименование и тип на данните, които се използват; ограничения за стойностите им и т.н.
- **Имплементация на бизнес логиката** - как точно се използват данните за постигането на целите на приложението.
- **Имплементация на REST услуги** - за комуникация с front-end проекта и мобилното приложение (за да може те да използват бизнес логиката от back-end проекта).
- **Защита на приложението** - управление на достъпа до различните ресурси в приложението.
- **Тестове** - unit и integration тестове на логиката в back-end проекта

Използва се Maven за управление на различните зависимости в проекта (за изтегляне на необходимите външни библиотеки и правилните им версии).

#### 3.1.1.2. ***Класове за описание на данните***

Това основно включва класовете в пакетите:

- *"db.entity"*
- *"db.validation"*
- *"model"*
- *"model.request"*

В пакета **db.entity** са включени JPA entity класовете. Това са класовете, от които автоматично се генерират таблици в базата данни чрез Hibernate (това включва и поставяне на връзките между таблиците).

Класовете отговарят на модела на данните - това включва имената на таблицата и полетата й, типа на полетата и ограниченията за стойностите им (зададени чрез анотации). Всички класове в този пакет наследяват от BaseEntity класа, който отговаря за генерирането на първичен ключ (ID).

Включени са класове за всяка единица, която участва в системата:

- сметка - *AccountEntity*
- категория - *CategoryEntity*
- транзакция - *TransactionEntity*
- потребител - *UserEntity*
- настройка на потребител - *UserSettingEntity*
- отчетен период – *ReportingPeriodEntity*

В пакета **db.validation** са включени 2 класа - EmailValidator и ValidEmail. Те се използват за валидация на email на потребител в системата. ValidEmail дефинира анотация (@ValidEmail), която да се използва в entity класовете, а EmailValidator предоставя самата имплементация за валидирането на email (използва се регулярен израз). Тези класове се използват като алтернатива за @Email анотацията на Hibernate, която предоставя по-слаба валидация в сравнение.

В пакета **model** са включени:

- DTO класове (*Account, Category, Transaction, ReportingPeriod, User, UserSetting, UserAuthenticationDetails, UserAuthenticationToken*)
- Enum-и (*AccountType, CategoryType, FamilyStatusType, GenderType, TransactionFromType, TransactionToType, UserSettingKey*)

Инстанции на DTO класовете се връщат от методите на класовете управляващи бизнес логиката на приложението (тези в пакет service, обяснени надолу).

Идеята на това е да не се изпращат директно entity обекти на REST контролерите, а да се използват DTO-та, които да скриват част от информацията, която се съдържа в entity обекта (за да може например при създаване на потребител и връщане на информацията за него, паролата му да не бъде включена в JSON обекта, който идва като отговор на заявката).

UserAuthenticationDetails и UserAuthenticationToken са специални DTO класове, които се използват от Spring Security при автентикация на потребител (съответно за пренасяне на username/password на потребител и за пренасяне на JWT).

Enum-ите се използват за определяне на различни типове (тип на категория, тип на сметка и така нататък) и се използват като типове на полетата в entity класовете. UserSettingKey е по-особен enum, който включва списък с позволени ключове за настройки на потребител.

В пакета **model.request** са включени DTO класове, които се използват при изпращане на REST заявки от клиента (намират приложение в REST контролерите, които са описани надолу). Един такъв клас описва какви полета и стойности (прилага се валидация чрез JPA анотации) трябва да съдържа JSON request body на една REST заявка.

3.1.1.3. ***Класове за имплементация на бизнес логиката***

Това включва класовете в пакетите:

- *"service"*
- *"service.mapper"*

- *"db.repository"*

В пакета **service** са включени класовете съдържащи бизнес логиката на приложението.

Всеки service клас си има своето предназначение. AccountService / CategoryService / TransactionService / ReportingPeriodService / UserService се занимават съответно с операции свързани със сметките / категориите / транзакциите / отчетните периоди / потребителите. Това основно включва CRUD операции за всички тези единици.

AuthenticationService се занимава с генерирането и изтриването на JWT. UserProfileService се занимава с операциите свързани с потребителя, които в момента е влязъл в системата (смяна на парола, преглед на данни, настройки и още).

Подробна информация за начина на работа на тези класове има в коментарите на кода им.

В пакета **service.mapper** са включени интерфейси, от които чрез MapStruct автоматично се генерира имплементация за "мапъри" (mappers) при изпълнение на команда "mvn clean install" (Maven команда).

Имплементацията на тези mapper-и предоставя възможността лесно да се преобразуват entity обекти в DTO и обратно, без да се налага ръчно да се пише бизнес логиката за това.

В декларираните методи в интерфейсите просто се задава от какъв обект се преобразува (подаден като аргумент) и в какъв обект трябва да стане преобразуването (типа на върнатия обект от метода). На база на това MapStruct генерира имплементацията за преобразуването (генерираните класове са в папката /target/generated-sources/annotations), която след това може да се ползва в service класовете.

В пакета **db.repository** са включени интерфейси, които позволяват лесната комуникация между базата данни и service класовете.

Декларирането на методи в тези интерфейси генерира имплементация от Spring Data JPA, която улеснява максимално извършването на CRUD операции с entity-тата. Например декларирането на метод deleteByEmail() в UserRepository генерира имплементация от Spring Data JPA, която сега UserService може да използва, за да изтрие потребител по e-mail (като просто извика метода и му подаде правилен аргумент).

В някои случаи декларирането на метод не е достатъчно (при по-сложни заявки - както например в TransactionRepository) и затова се използва JPQL заявка и метод, който е асоцииран с нея (изписва се заявката в @Query анотация, а отдолу се декларира метод, който я изпълнява, като след това Spring Data JPA отново генерира имплементацията).

3.1.1.4. ***Класове за имплементация на REST услугите***

Това включва класовете в пакетите:

- *"rest"*
- *"rest.exception"*

В пакета **rest** са включени класове за REST контролери, които имплементират endpoints за различните възможни операции.

Това означава, че контролерите правят възможни операции като GET към "/users", POST към "/accounts" и така нататък, и определят какво се случва при подаването на такава заявка от клиента (и съответно какъв е отговорът от нея). Заявките връщат HTTP код, който отговаря на извършената операция, и JSON обект (или масив от обекти) с информация получена от изпълнението на заявката.

Самата бизнес логика не е в класовете контролери - те само се обръщат към service класовете, които съдържат логиката.

Всеки клас контролер си има своето предназначение. AccountController / CategoryController / TransactionController / ReportingPeriodController / UserController се занимават съответно с операции свързани със сметките / категориите / транзакциите /

отчетните периоди / потребителите. Това основно включва CRUD операции за всички тези единици.

AuthenticationController се занимава с генерирането и изтриването на JWT (при login / logout от потребителя). UserProfileController се занимава с операциите свързани с потребителя, които в момента е влязъл в системата (смяна на парола, преглед на данни, настройки и още). Подробна информация за начина на работа на тези класове има в коментарите на кода им.

В пакета **rest.exception** са имплементирани класове изключения, които се използват, когато възникне някаква грешка при изпращане на заявка до REST контролерите (например при опит за достъп до несъществуващ ресурс или до ресурс, който не принадлежи на потребителя, който се опитва да го достъпи). Чрез тези класове се задава подходящо съобщение за грешка и подходящ HTTP код, който да връща заявката.

3.1.1.5. ***Класове за защита на приложението***

Това включва класовете в пакетите:

- *"security"*
- *"security.jwt"*

В **security** пакета са включени класове за конфигурация на защитата на приложението. Използва се функционалността на Spring Security.

Най-съществения клас е SecurityConfiguration, в който се задават настройки за защита като например кой до какви ресурси има достъп, кои ресурси са защитени, какво се случва при login/logout. Важно е да се подчертае, че не се ползват HTTP сесии, защото се ползва JWT автентикация.

AuthenticationFailureListener е клас, който следи за невалидни опити за вход в системата и определя какво да се случи при настъпването на такова събитие. Аналогично за AuthenticationSuccessListener класа, който следи за валидни опити за вход. Тези класове се използват за реализация на защита от bruteforce атаки.

DatabaseUserAuthenticationDetails класът позволява автентикацията да става единствено с данни на потребители, които съществуват в базата данни.

PasswordEncoderConfiguration предоставя възможността да се шифроват паролите на потребителите. Самите пароли не се пазят в plain text вид в базата данни, а хеширани (използва се BCrypt, защото е надежден модерен алгоритъм за хеширане на пароли).

В **security.jwt** пакета са включени класове, които имплементират JWT автентикация. Идеята на JWT е да се генерира уникален секретен символен низ за потребителя, който е влязъл в системата, чрез който да се осигурява достъп до различните ресурси. Този низ съдържа в себе си (кодирана) информация за текущия потребител, чрез която системата определя какъв достъп има потребителят.

3.1.1.6. ***Класове за тестване на проекта***

Това включва класовете в папката /src/test/java на проекта (имената на пакетите в нея съответстват на пакетите, които биват тествани). Тествани са само класовете, които съдържат методи с някаква логика. Това основно са класовете в пакети rest и service.

В rest пакета се съдържат integration тестове, които проверяват дали като цяло имплементираното REST API работи както трябва (дали заявката се обработва правилно и дали отговорът й е този, който се очаква).

В service пакета се съдържат unit тестове, които тестват бизнес логиката в различните методи в service класовете (дали данните биват правилно манипулирани).

3.1.1.7. ***Дизайн на изискване "Регистрация и вход в системата за потребители без Facebook/Google профил"***

Регистрацията става като се изпрати POST заявка до "/users" с информация за потребителя (име, парола, email и още). Подадената информация първо бива валидирана от REST контролера (UserController) и ако е валидна, се изпраща до service класа.

Service класът (UserService) създава потребител в базата данни с подадената информация (като защитна мярка паролата на потребителя се запазва в хеширан вид). Сега комбинацията на e-mail / парола за този потребител може да се използва за вход в системата.

При регистрацията също така се създават служебни категории за потребителя (една за приходи и една за разходи), които участват в служебни транзакции (например при синхронизиране на наличност по сметка на потребител). Освен това се добавя настройка за потребителя, с която се следи дали е влизал някога в системата (за да може при първото му влизане да му бъде предложено да бъдат създадени примерни категории и сметки).

Входът става като се изпрати POST заявка до "/auth/login" с request body, което съдържа email и парола на потребителя. Подадената информация първо бива валидирана от REST контролера (AuthenticationController) и ако е валидна, се изпраща до service класа.

Service класът (AuthenticationController) създава JWT (JSON Web Token) за потребителя, ако подадените данни са валидни (има такъв потребител), а след това го връща като отговор на HTTP заявката.

Сега клиентът може да използва генерирания token string, за да си осигури достъп до различните ресурси в системата, които му принадлежат. Това става като към всяка следваща заявка се добави хедър "Authorization" със стойност "Bearer *token-string*", където *token-string* е генерирания token string.

Имплементирана е и anti-bruteforce защита, която ограничава броя на поредни неуспешни опити за вход от един IP адрес. Ако лимитът за пореден брой неуспешни опити за вход е надвишен, IP адресът на потребителя се блокира за зададено време и той временно губи правото си за вход в системата (дори и да въведе коректни данни през това време). Ако потребителят успешно влезе в системата преди да надвиши този лимит, броячът на неуспешни поредни опити за вход се занулява. Тази логика се реализира от класовете AuthenticationFailureListener (следи за неуспешен опит за вход в системата), AuthenticationSuccessListener (следи за успешен вход в системата) и

LoginLimitService (управлява блокирането/отблокирането на IP адресите и брояча на поредни неуспешни опити за вход за даден IP адрес).

### 3.1.1.8. ***Дизайн на изискване "Преглед на профилна информация за регистрирани потребители"***

Профилната информация за текущия потребител се извлича с проста GET заявка до "/profile". Алтернативно информацията може да се извлече с GET заявка до "/users/{id}" (където id е уникалният номер на потребителя в базата данни).

REST контролерът (UserProfileController) получава заявката и я изпраща до service класа. Service класът (UserProfileService) проверява кой потребител е влязъл в системата в момента, извлича данните му от съответстващия му entity обект, преобразува ги в DTO (и същевременно не включва паролата на потребителя в този обект, поради съображения за сигурност), а накрая го връща на REST контролера, за да послужи като отговор на заявката.

### 3.1.1.9. ***Дизайн на изискване "Смяна на парола за потребители без Facebook/Google профил"***

Смяната на паролата се осъществява с пращане на PUT заявка до "/profile/password" (операцията е достъпна само за потребители, които са влезли в системата).

Заявката включва подадени старата и новата парола (старата един път, а новата два пъти за потвърждение), като старата трябва да съответства на текущата парола на потребителя, а новата трябва е изписана еднакво два пъти и да има дължина поне 6 символа (за по-голяма сигурност).

REST контролерът (UserProfileController) извършва част от валидацията (проверява дали са попълнени стойности и дали новата парола отговаря на изискването за дължина). Ако заявката е валидна, тя се изпраща до service класа.

Service класът (UserProfileService) проверява дали старата парола съответства на паролата, която се пази в базата данни за текущия потребител, а също така проверява

дали двете нови пароли са еднакви. Ако това е така, паролата се обновява (и отново се запазва в базата данни в хеширан вид).

### 3.1.1.10. ***Дизайн на изискване "Управление (създаване, редактиране, деактивиране/активиране, изтриване) на сметки"***

Създаването става чрез POST заявка до "/accounts". Подава се информацията за сметката в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (AccountController) и ако е валидна, се изпраща до service класа. Service класът (AccountService) първо проверява дали няма сметка с това име за текущия потребител. Ако няма, създава нова сметка в базата данни с подадената информация в заявката (а за собственик на сметката се счита потребителят, който е изпратил заявката) и връща данните й като отговор на заявката.

Редактирането става чрез PUT заявка до "/accounts/{id}" (където id е ID-то на сметката, която да бъде обновена). Подава се обновената информация за сметката в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (AccountController) и ако е валидна, се изпраща до service класа.

Service класът (AccountService) първо проверява дали въобще съществува такава сметка. Ако съществува, проверява дали сметката принадлежи на потребителя, който е изпратил заявката. Ако и това е налице, накрая проверява дали новото подадено име на сметката вече не се използва за друга сметка на текущия потребител. Ако и името е валидно (не се използва все още), данните на сметката се обновяват и запазват, а накрая се връщат като отговор на заявката.

Деактивирането става чрез PATCH заявка до "/accounts/{id}/deactivate" (където id е ID-то на сметката, която да бъде деактивирана).

REST контролерът (AccountController) директно се обръща към service класа. Service класът (AccountService) първо проверява дали съществува сметка с подаденото ID и

дали принадлежи на текущия потребител. Ако това е така, проверява дали сметката не е маркирана като изтрита (в такъв случай по изискване деактивирането не е възможно и затова не се извършва). Ако не е, сметката се маркира като деактивирана и се връщат обновените й данни като отговор на заявката. Когато сметката е деактивирана, тя спира да участва в изчислението на общия баланс по всички сметки на потребителя.

Активирането става чрез PATCH заявка до "/accounts/{id}/activate" (където id е ID-то на сметката, която да бъде активирана). Логиката е същата като при деактивирането, но с едната разликата, че накрая сметката се маркира като активирана и след това участва в изчислението на общия баланс по всички сметки на потребителя.

Изтриването става чрез DELETE заявка до "/accounts/{id}" (където id е ID-то на сметката, която да бъде изтрита).

REST контролерът (AccountController) директно се обръща към service класа. Service класът (AccountService) първо проверява дали съществува сметка с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, проверява дали сметката вече не е маркирана като изтрита и дали балансът по нея има стойност различна от 0 (в такъв случай по изискване изтриването не е възможно и затова не се извършва). Важно е да се отбележи, че сметката не се изтрива от базата данни, а само се маркира като изтрита (по изискване).

Има възможността и за преглед на данните на сметките. Това става с GET заявки до "/accounts" и до "/accounts/{id}" (където id е ID-то на дадената сметка, за която да се извлече информация).

REST контролерът (AccountController) получава заявката и я изпраща до service класа. Service класът (AccountService) проверява дали сметката съществува и дали принадлежи на текущия потребител, а ако да, връща данните й като отговор на заявката. Ако се извличат данни за всички сметки, се връща списък с данните им (или празен списък, ако няма създадени сметки).

### 3.1.1.11. ***Дизайн на изискване "Редактиране на профилна информация за регистрирани потребители"***

Редактирането на данни на потребителя става чрез PUT заявка до "/profile" с подадени в request body обновените данни на потребителя. Алтернативно информацията може да се обнови с PUT заявка до "/users/{id}" (където id е уникалният номер на потребителя в базата данни).

В request body на заявката трябва да са включени стойности за всички полета на потребителя с изключение на email, ID и парола. Това е така, защото email-ът на потребителя не може да бъде променян (по изискване), ID-то на потребителя се извлича автоматично (от service класа), а паролата на потребителя се обновява чрез отделна HTTP заявка.

REST контролерът (UserProfileController) извършва валидацията (проверява дали са попълнени коректни нови стойности за данните на потребителя). Ако заявката е валидна, тя се изпраща до service класа. Service класът (UserProfileService) просто обновява данните на текущия потребител и връща тези обновени данни (без паролата) под формата на DTO, който да послужи като отговор на заявката.

3.1.1.12. ***Дизайн на изискване "Управление (създаване, редактиране, изтриване) на пера (категории)"***

Създаването става чрез POST заявка до "/categories". Подава се информацията за категорията в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (CategoryController) и ако е валидна, се изпраща до service класа.

Service класът (CategoryService) първо проверява дали няма категория с това име за текущия потребител. Ако няма, създава нова категория в базата данни с подадената информация в заявката (а за собственик на категорията се счита потребителят, който е изпратил заявката) и връща данните й като отговор на заявката.

Редактирането става чрез PUT заявка до "/categories/{id}" (където id е ID-то на категорията, която да бъде обновена). Подава се обновената информация за категорията в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (CategoryController) и ако е валидна, се изпраща до service класа.

Service класът (CategoryService) първо проверява дали въобще съществува такава категория. Ако съществува, проверява дали категорията принадлежи на потребителя, който е изпратил заявката. Ако и това е налице, накрая проверява дали новото подадено име на категорията вече не се използва за друга категория на текущия потребител. Ако и името е валидно (не се използва все още), данните на категорията се обновяват и запазват, а накрая се връщат като отговор на заявката.

Изтриването става чрез DELETE заявка до "/categories/{id}" (където id е ID-то на категорията, която да бъде изтрита).

REST контролерът (CategoryController) директно се обръща към service класа. Service класът (CategoryService) първо проверява дали съществува категория с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, категорията се изтрива и данните на изтритата категория биват върнати като отговор на заявката.

Има възможността и за преглед на данните на категориите. Това става с GET заявки до "/categories" и до "/categories/{id}" (където id е ID-то на дадената категория, за която да се извлече информация).

REST контролерът (CategoryController) получава заявката и я изпраща до service класа. Service класът (CategoryService) проверява дали категорията съществува и дали принадлежи на текущия потребител, а ако да, връща данните й като отговор на заявката. Ако се извличат данни за всички категории, се връща списък с данните им.

### 3.1.1.13. ***Дизайн на изискване "Управление (създаване, редактиране, изтриване) на транзакции"***

Създаването става чрез POST заявка до "/transactions". Подава се информацията за транзакцията в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (TransactionController) и ако е валидна, се изпраща до service класа.

Service класът (TransactionService) първо проверява дали подадените данни в заявката за "fromType" & "fromId" и "toType" & "toId" са валидни. Това са данните за сметките/категориите, по които ще се правят манипулации чрез транзакцията.

Видът на транзакцията се определя от подадените стойности за тези полета:

- Ако "fromType" е КАТЕГОРИЯ, а "toType" е СМЕТКА, транзакцията е ПРИХОД.
- Ако "fromType" е СМЕТКА, а "toType" е КАТЕГОРИЯ, транзакцията е РАЗХОД.
- Ако "fromType" е СМЕТКА, а "toType" също е СМЕТКА, транзакцията е ТРАНСФЕР.

За да се считат за валидни тези данни, е нужно да са изпълнени следните условия:

- "fromType" и "toType" не може заедно да са от тип КАТЕГОРИЯ;
- "fromId" и "toId" трябва да съдържат ID на съществуващи сметки/категории, които принадлежат на текущия потребител (и са активирани, ако са сметки);
- ако транзакцията е от тип ПРИХОД, "from категорията" й трябва също да е от тип ПРИХОД;
- ако транзакцията е от тип РАЗХОД, "to категорията" й трябва също да е от тип РАЗХОД;
- ако транзакцията е от тип ТРАНСФЕР, "from сметката" трябва да има баланс по-голям или равен на сумата посочена в заявката на транзакцията.

Ако подадените данни са валидни, service класът изпълнява транзакцията (тоест извършва манипулации в базата данни по сметките/категориите, които участват в транзакцията).

Логиката за изпълняване на транзакцията зависи от типа на транзакцията:

- Ако транзакцията е ПРИХОД, се увеличават балансът на "to сметката" и сумата на "from категорията" със сумата посочена в заявката.
- Ако транзакцията е РАЗХОД, се намалява балансът на "from сметката" и се увеличава сумата на "to категорията" със сумата посочена в заявката.
- Ако транзакцията е ТРАНСФЕР, се намалява балансът на "from сметката" и се увеличава балансът на "to сметката" със сумата посочена в заявката.

Накрая service класът запазва транзакцията в базата данни и връща данните й като отговор на заявката.

Редактирането става чрез PUT заявка до "/transactions/{id}" (където id е ID-то на транзакцията, която да бъде обновена). Подава се обновената информация за транзакцията в request body на заявката.

Подадената информация първо бива валидирана от REST контролера (TransactionController) и ако е валидна, се изпраща до service класа. Service класът (TransactionService) първо проверява дали въобще съществува такава транзакция. Ако съществува, проверява дали транзакцията принадлежи на потребителя, който е изпратил заявката. Ако и това е налице, service класът проверява дали новите подадени "from" и "to" данни са валидни. Ако да, се отменят предишните манипулации извършени от транзакцията и се изпълняват новите. Накрая се обновяват данните на транзакцията в базата данни и тези данни се връщат като отговор на заявката.

Изтриването става чрез DELETE заявка до "/transactions/{id}" (където id е ID-то на транзакцията, която да бъде изтрита).

REST контролерът (TransactionController) директно се обръща към service класа. Service класът (TransactionService) първо проверява дали съществува транзакция с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, манипулациите извършени от транзакцията се отменят и транзакцията се изтрива, а данните на изтритата транзакция биват върнати като отговор на заявката.

3.1.1.14. ***Дизайн на изискване "Синхронизиране на наличност"***

Синхронизирането на наличност по сметка става с PATCH заявка до "/accounts/{id}" с request параметър "balance" (новата наличност по сметката), където id е ID-то на сметката.

REST контролерът (AccountController) директно се обръща към service класа. Service класът (AccountService) първо проверява дали съществува сметка с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, се извършва или служебна приходна транзакция (ако новата наличност по сметката е по-голяма от старата), или служебна разходна транзакция (ако новата наличност е по-малка от старата) със сума

разликата между новата и старата наличност. За целта се използват системните разходни и приходни категории, които се създават при регистрация на потребителя.

3.1.1.15. ***Дизайн на изискване "Преглед на детайли за транзакция"***

Детайлите за дадена транзакция се извличат с проста GET заявка до "/transactions/{id}" (където id е ID-то на транзакцията, за която да се извлече информация). Има и опцията да се извлече информация за много транзакции (разделени на страници, тъй като се очаква, че транзакциите ще са много на брой) - това става чрез GET заявка до "/transactions" с подадени request параметри за определяне на колко транзакции да бъдат върнати на страница.

REST контролерът (TransactionController) получава заявката и я изпраща до service класа. Service класът (TransactionService) проверява дали транзакцията съществува и дали принадлежи на текущия потребител, а ако да, връща данните й като отговор на заявката.

3.1.1.16. ***Дизайн на изискване "Обща статистика за наличност по всички сметки и по всички зададени категории"***

Общата наличност по всички сметки се извлича с GET заявка до "/accounts/balance-sum".

REST контролерът (AccountController) получава заявката и я изпраща до service класа. Service класът (AccountService) проверява кой потребител е влязъл системата, изчислява общата наличност по сметките му (чрез JPQL заявка реализирана в AccountRepository) и я връща като отговор на заявката. Важно е да се подчертае, че се вземат предвид само АКТИВИРАНИТЕ сметки (наличността по деактивирани и изтрити сметки не се добавя към общата стойност).

Общата сума по всички категории се извлича с GET заявка до "/categories/total-current-period-sum" с request параметър "type" (типа на категорията - разходна или приходна).

REST контролерът (CategoryController) получава заявката и я изпраща до service класа. Service класът (CategoryService) проверява кой потребител е влязъл системата,

изчислява общата сума по категориите му от зададения тип (чрез JPQL заявка реализирана в CategoryRepository) и я връща като отговор на заявката.

3.1.1.17. ***Дизайн на изискване "Поставяне на цели по сметки"***

Частично имплементирано.

За момента може да се зададе цел за дадена сметка чрез PATCH заявка до "/accounts/{id}" с request параметър "goal" (сумата цел за сметката), където id е ID-то на сметката.

REST контролерът (AccountController) директно се обръща към service класа. Service класът (AccountService) първо проверява дали съществува сметка с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, се обновява целта за сметката в базата данни, а обновените данни на сметката биват върнати като отговор на заявката.

Предстои да се имплементира логика за използване на стойността за целта на сметката.

3.1.1.18. ***Дизайн на изискване "Бюджетиране по пера за разходи"***

Частично имплементирано.

За момента може да се зададе лимит за харчене по дадена категория за разходи чрез PATCH заявка до "/categories/{id}/limit" (където id е ID-то на дадената категория).

REST контролерът (CategoryController) директно се обръща към service класа. Service класът (CategoryService) първо проверява дали съществува категория с подаденото ID и дали принадлежи на текущия потребител. Ако това е така, се обновява лимита за категорията в базата данни, а обновените данни на категорията биват върнати като отговор на заявката.

Предстои да се имплементира логика за използване на стойността за лимита на категорията.

3.1.1.19. ***Дизайн на изискване "Списък на всички приходи за отчетен период"***

Частично имплементирано.

За момента има възможност да се изчислят общите приходи за даден интервал от време по дадена сметка.

Общите приходи за даден период от време се извличат с GET заявка до "/accounts/{id}/income-sum" с request параметри startDate и endDate (дати във формат dd.MM.yyyy; начална и крайна дата включително), където id е ID-то на сметката.

REST контролерът (AccountController) получава заявката и я изпраща до service класа. Service класът (AccountService) проверява дали съществува такава сметка и дали принадлежи на текущия потребител, а ако да, изчислява общите приходи по сметката за посочения период. Сумата се изчислява на база на изпълнените транзакции, в които участва посочената сметка. Това става чрез JPQL заявка реализирана в AccountRepository (заявката включва в крайния резултат и приходи от трансферни транзакции). Накрая тази изчислена сума се връща като отговор на заявката.

Също могат да се извлекат всички приходни транзакции между две дати. Това става с GET заявка до "/transactions/between-dates" с request параметри "type" (тип на транзакцията), startDate и endDate (дати във формат dd.MM.yyyy; начална и крайна дата включително). Има и опцията да се извлече информацията разделена на страници (тъй като се очаква, че транзакциите ще са много на брой) - това става чрез подаден допълнителен request параметър за определяне на колко транзакции да бъдат върнати на страница.

REST контролерът (TransactionController) получава заявката и я изпраща до service класа. Service класът (TransactionService) проверява дали подадения тип на транзакцията е валиден и ако да, извлича транзакциите, които са от този тип и са извършени в подадения интервал от време (благодарение на JPQL заявка, която е реализирана в TransactionRepository). Накрая връща данните на тези транзакции като отговор на заявката.

Предстои да се имплементира логика за отчетни периоди и извличане на приходни транзакции за дадена сметка.

3.1.1.20. ***Дизайн на изискване "Списък на всички разходи за отчетен период"***

Частично имплементирано.

За момента има възможност да се изчислят общите разходи за даден интервал от време по дадена сметка.

Общите разходи за даден период от време се извличат с GET заявка до "/accounts/{id}/expense-sum" с request параметри startDate и endDate (дати във формат dd.MM.yyyy; начална и крайна дата включително), където id е ID-то на сметката.

REST контролерът (AccountController) получава заявката и я изпраща до service класа. Service класът (AccountService) проверява дали съществува такава сметка и дали принадлежи на текущия потребител, а ако да, изчислява общите разходи по сметката за посочения период. Сумата се изчислява на база на изпълнените транзакции, в които участва посочената сметка. Това става чрез JPQL заявка реализирана в AccountRepository (заявката включва в крайния резултат и разходи от трансферни транзакции). Накрая тази изчислена сума се връща като отговор на заявката.

Също могат да се извлекат всички разходни транзакции между две дати. Логиката е същата като при извличането на приходни транзакции.

Предстои да се имплементира логика за отчетни периоди и извличане на разходни транзакции за дадена сметка.

3.1.1.21. ***Дизайн на изискване "Списък на всички разходи за отчетен период"***

Частично имплементирано.

За момента има възможност да се изчисли общата добавена сума за даден интервал от време по дадена приходна/разходна категория. Също могат да се извлекат всички

приходни/разходни транзакции между две дати (логиката е описана в предишните две изисквания).

Това става с GET заявка до "/categories/{id}/added-sum" с request параметри startDate и endDate (дати във формат dd.MM.yyyy; начална и крайна дата включително), където id е ID-то на категорията.

REST контролерът (CategoryController) получава заявката и я изпраща до service класа. Service класът (CategoryService) проверява дали съществува такава категория и дали принадлежи на текущия потребител. Ако да, проверява какъв е типът на категорията и след това изчислява общата добавена сума по категорията за посочения период. Сумата се изчислява на база на изпълнените транзакции, в които участва посочената категория. Това става чрез JPQL заявка реализирана в TransactionRepository. Накрая тази изчислена сума се връща като отговор на заявката.

Предстои да се имплементира логика за отчетни периоди.

### *3.1.2. Тестов модел*

Предназначението на документа "Тестов модел" е да представи и опише тестовите случаи, с които проектът е тестван. Тестовите случаи обхващат всички функционалности на системата. За всяка функционалност добавих по един или повече тестови случая.

Във всеки тестов случай са описани общите стъпки, които ще бъдат извършени по време на теста на съответната функционалност. Това означава, че един тестов случай в тестовия модел може да отговаря за повече от един изпълнен тест. Например от тестов случай "Неуспешна регистрация на потребител (въведени некоректни данни)" може да се реализират няколко теста:

- Неуспешна регистрация заради невъведен e-mail;
- Неуспешна регистрация заради въведен вече съществуващ email;
- Неуспешна регистрация заради въведена невалидна парола

... и още.

В тестовия случай са описани възможните комбинации от данни, които могат да се използват при имплементация на автоматизирани тестове или при ръчно тестване.

Всеки тест се счита за успешен, ако очакваният резултат съвпада с реалния получен резултат. При разминаване тестът се счита за неуспешен.

Тестовите случаи описани тук са предназначени за тестове върху front-end проекта на системата, който се обръща до back-end проекта. Back-end проектът има имплементирани свои собствени unit и интеграционни тестове, които се базират на тестовите случаи описани в този документ.

За валидни се считат всички данни, които отговарят на следните изисквания:

- да са в определената дължина (например паролата на потребителя има минимум за дължината си, потребителското име не може да е празно и т.н.);
- да съдържат само символи, които принадлежат на правилната азбука (например потребителското име не може да съдържа интервал)

За верни се считат данни, които съответстват на данните, които са съхранени в базата данни (например въведената парола за вход на потребителя трябва да съответства на паролата му, която е в базата данни). За коректни се считат данни, които са *валидни* и *верни*.

Ето няколко тестови случая от тестовия модел (останалите могат да бъдат прегледани в самия документ):

### 3.1.2.1. ***Неуспешна регистрация за потребители без Facebook/Google профил (въведени невалидни данни)***

Предусловия:
→ 1. Потребителят не е влязъл в системата.

| Номер на стъпка | Извършено действие | Очакван резултат |
|---|---|---|
| 1 | Отваряне на страницата за регистрация | Визуализиране на форма за регистрация |

| | | |
|---|---|---|
| 2 | Попълване на формата за регистрация с невалидни данни:<br>• невалидно / вече съществуващо / непопълнено потребителско име (email)<br>• невалидна / непопълнена парола | Въведени некоректни данни за регистрация на потребител |
| 3 | Избиране на бутон за потвърждаване на регистрацията | Извеждане на съобщение за грешка |

Общ очакван резултат: Неуспешен опит на потребителя за регистрация в системата.

3.1.2.2. ***Успешно забраняване на вход в системата (за интервал от време след няколко поредни неуспешни опита за вход)***

Предусловия:

→ 1. Потребителят не е влязъл в системата.

→ 2. Забраната е настроена да настъпва след 3 поредни неуспешни опита.

| Номер на стъпка | Извършено действие | Очакван резултат |
|---|---|---|
| 1 | Отваряне на страницата за вход | Визуализиране на форма за вход |
| 2 | Попълване на формата за вход с некоректни данни:<br>• несъществуващо / непопълнено потребителско име (email)<br>• невярна (несъответстваща на потребителското име) / непопълнена парола | Въведени некоректни данни за вход на потребител |
| 3 | Избиране на бутон за вход | Извеждане на съобщение за |

| | | |
|---|---|---|
| | | грешка |
| 4 | Второ попълване на формата с некоректни данни | Въведени некоректни данни за вход на потребител |
| 5 | Избиране на бутон за вход | Извеждане на съобщение за грешка |
| 6 | Трето попълване на формата с некоректни данни | Въведени некоректни данни за вход на потребител |
| 7 | Избиране на бутон за вход | Извеждане на съобщение за грешка (с допълнителна информация, че потребителят временно е загубил правото си за опит за вход в системата);<br><br>Правене на бутона за вход под формата неактивен |

Общ очакван резултат: Неуспешен опит на потребителя за вход в системата и временна загуба на правото му за опит за вход в системата.

3.1.2.3. ***Успешно редактиране на сметка***

Предусловия:

→ 1. Потребителят е влязъл в системата.

→ 2. Потребителят има поне една създадена сметка.

| Номер на стъпка | Извършено действие | Очакван резултат |
|---|---|---|
| 1 | Отваряне на страницата със списъка на сметките на потребителя | Визуализиране на страница, която съдържа списък със сметките на потребителя |
| 2 | Избиране на бутон за редакция на дадена сметка | Показване на форма, която е попълнена с текущата налична информация за избраната сметка |
| 3 | Заменяне на данните във | Попълнена форма за редакция |

| | | |
|---|---|---|
| | формата с нови коректни данни:<br>• валидно и неизползвано име на сметката<br>• валидна наличност | на сметка |
| 4 | Натискане на бутон за запазване на данните | Показване на съобщение за успешно редактирана сметка |

Общ очакван резултат: Сметката е успешно редактирана и потребителят вече я вижда с обновените й данни.

3.1.2.4. ***Неуспешно синхронизиране на наличност по сметки (пречещи предусловия)***

Предусловия:
→ 1. Потребителят е влязъл в системата.
→ 2. Потребителят има поне една създадена сметка.
→ 3. Някоя от избраните сметки за синхронизация е била изтрита/деактивирана в друг прозорец / Загубена е връзка със системата.

| Номер на стъпка | Извършено действие | Очакван резултат |
|---|---|---|
| 1 | Избиране на меню за сметки | Визуализиране на подменю за синхронизация на сметките |
| 2 | Избиране на подменю за синхронизация на сметките | Визуализиране на форма за синхронизация на сметките |
| 3 | Избиране на сметки за синхронизация и попълване на новата им наличност | Попълнена форма за синхронизация |
| 4 | Избиране на бутон за изпълняване на синхронизацията | Показване на съобщение за грешка;<br><br>Обновена е наличността само на избраните сметки, които са съществуващи и активирани (при наличие на връзка със |

| | | |
|---|---|---|
| | | системата) |

Общ очакван резултат: Синхронизирана е наличността само по избраните сметки, които са съществуващи и активирани (при наличие на връзка със системата).

3.1.2.5. ***Успешно създаване на транзакция***

Предусловия:

→ 1. Потребителят е влязъл в системата.

→ 2. Потребителят има създадени поне една сметка и една категория.

| Номер на стъпка | Извършено действие | Очакван резултат |
|---|---|---|
| 1 | Избиране на меню за транзакции | Визуализиране на подменюта за добавяне на транзакция и за списък с транзакциите |
| 2 | Избиране на подменю за добавяне на транзакция | Показване на форма за попълване на данни на нова транзакция |
| 3 | Попълване на формата с коректни данни:<br>• валидна дата на изпълнение<br>• валиден тип на транзакцията<br>• валидна сума<br>• валидни данни за "от" и "до" на транзакцията | Попълнена форма за създаване на нова транзакция |
| 4 | Натискане на бутон за запазване на транзакцията | Показване на съобщение за успешно добавена транзакция;<br><br>Препращане към обновения списък на транзакциите на потребителя;<br><br>Сумата/наличността по засегнатите категории/сметки е |

| | | |
|---|---|---|
| | | обновена |

Общ очакван резултат: Транзакцията е добавена успешно в базата данни и потребителят вече я вижда в списъка си с транзакции, а сумата/наличността по засегнатите категории/сметки е обновена.

### *3.1.3. Речник*

Първоначалният речник, който създадох, беше документ в Confluence, който съдържа таблици с важни дефиниции, абревиатури и акроними използвани в проекта.

В последствие този документ беше заменен от разширението “Smart Terms for Confluence – Glossary“, защото то предоставя по-голямо удобство за хората, които пишат документите, и за тези, които ще ги четат (четящите по-лесно разбират определението на някой термин, а пишещите по-лесно добавят нови термини и определения).

Принципът му на работа е следният. Първо се добавя някакъв термин в общия списък с термини. След това може да се види определението на този термин като се кликне на мястото където е използван (в който и да е Confluence документ).

Ето експортирания от разширението списък с термини (с удебелен шрифт са изписани имената на термините, които са специфични за нашия проект):

| **Име** | **Определение** |
|---|---|
| Android Studio | Официалната среда за интегрирана разработка (IDE) за разработка на приложения за Android, базирана на IntelliJ IDEA. |
| Angular | Платформа за разработване на уеб приложения (web framework). |
| Apache Tomcat | Реализация с отворен код на Java Servlet, JavaServer Pages, Java Expression Language и WebSocket технологии. |
| API | "Application programming interface" - набор от практики, протоколи и инструменти за изграждане на софтуерни приложения |
| Bitbucket | Система за уеб-базиран контрол на версиите на софтуера. |
| BPMN | "Business Process Model and Notation" - графично представяне и специфициране на бизнес процесите и бизнес модела |

| | |
|---|---|
| Confluence | Софтуер за управляване на документацията по проекти. |
| Drive | Среда за съхраняване и споделяне на документи |
| DTO | "Data Transfer Object" - обект за пренос на данни (повече информация тук: https://en.wikipedia.org/wiki/Data_transfer_object) |
| Eclipse | Интегрирана среда за разработка, използвана в компютърното програмиране. Съдържа основно работно пространство и разширяема система за приставки за персонализиране на средата. |
| Enterprise Architect | Инструмент за визуално моделиране и проектиране, базиран на OMG UML. Използван за проектиране и изграждане на бизнес процеси и моделиране на UML. |
| Google Docs | Текстов процесор, включен като част от безплатен уеб базиран софтуерен офис пакет, предлаган от Google в рамките на услугата му Google Drive. |
| HTTP | "Hypertext Transfer Protocol" - протокол за предаване на информация по World Wide Web (Световната мрежа). |
| IntelliJ IDEA | Интегрирана среда за разработка, написана на Java за разработване на компютърен софтуер. Той е разработен от JetBrains и е достъпен с безплатен лиценз Apache 2, така и като платен софтуер. И двете версии могат да се използват с търговски цели. |
| Java Code Conventions | Правила и насоки за писане на качествен Java код |
| Jira | Софтуер за следене на задачи. |
| JPA | "Java Persistence API" - Java API за създаване на класове, които се преобразуват в таблици в базата данни. Нужна е конкретна имплементация на JPA (например Hibernate). |
| JVM | Виртуална машина, която позволява на компютъра да стартира както Java програми, така и програми написани на други езици, които също са компилирани в байт-код на Java. |
| Maven | Инструмент за автоматизация на изграждане, използван главно за Java проекти. |
| MVC | "Model-View-Controller"<br><br>Трислойна архитектура - помага за лесно разделяне на системата на компоненти<br><br>MVC е архитектурен шаблон за дизайн в програмирането, който е основан на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение.<br><br>Моделът (Model) е централен компонент в шаблона. Това е динамичната структура от данни на приложението, независима от потребителския интерфейс. Моделът управлява данните, логиката и правилата на приложението.<br><br>Изгледът (View) е изходящият поток от информация (това, което приложението изпраща като отговор до дисплея, респективно – до потребителя, в следствие на неговата заявка). Възможни са няколко различни изгледа на една и |

| | |
|---|---|
| | съща информация, като например различни диаграми за мениджмънт на даден ресурс или различни таблици.<br><br>Контролерът (Controller) е третата част от този шаблон. Той приема потребителския вход (т.е. данните, които потребителя въвежда, неговите заявки и т.н.) и ги преобразува в команди към модела или изгледа. |
| Node.js | Среда за изпълнение на JavaScript, която изпълнява JavaScript код извън уеб браузър. |
| **PFM** | "Personal Finance Manager" - Приложение за управление на лични финанси |
| pgAdmin | Богата на функции платформа за администриране и разработка на Open Source за PostgreSQL. |
| PostgreSQL | Безплатна и отворена система за релационна база данни за управление, наблягаща на разширяемостта и спазването на SQL (известна още като Postgres). |
| QA | "Quality Assurance" - Осигуряване на качество |
| RDBMS | "Relational Database Management System" - система, която предоставя средства за работа с релационни бази данни. |
| React Native | React Native е framework за мобилни приложения. Използва се за разработване на приложения за Android, iOS, Web и UWP, като позволява на разработчиците да използват React за постигането на тази цел. |
| REST | "Representational State Transfer" - стил софтуерна архитектура за реализация на уеб услуги. |
| RUP | "Rational Unified Process" - методология за разработка на софтуерни проекти. |
| Spring | Технологична рамка (framework) за Java платформата. Предоставя много функции, които улесняват разработването на Java-базирани enterprise приложения. |
| TypeScript Style Guide | Правила и насоки за писане на качествен TypeScript код |
| UI | "User Interface" - потребителски интерфейс |
| UML | "Unified Modeling Language" - графичен език за визуализиране и специфициране на елементите на софтуерната система. |
| Vagrant | Инструмент, който дава възможност на потребителите да създават и конфигурират леки, възпроизводими и преносими среди за разработка |
| Visual Studio Code | Редактор на изходен код създаден от Microsoft за Windows, Linux и macOS. Характеристиките включват поддръжка за отстраняване на грешки, подчертаване на синтаксис, интелигентно попълване на код, фрагменти, преработка на код (refactoring) и вграден Git. |
| **Бюджет** | Колко пари може да си позволи да изхарчи потребителят (обикновено за даден период –например бюджет за месеца – и за дадена категория –например бюджет за дрехи) |
| Инструментални среди | Системата за програмиране се състои от инструментални среди, предоставящи средства, предназначени за създаване на програми. |

| | |
|---|---|
| | Създаването на дадено приложение най-общо преминава през няколко стъпки, като за всяка от тях са необходими съответни инструментални средства.<br><br>Към тези средства като минимум могат да се включат: езици за програмиране, текстови редактори, графични редактори, транслатори, програми за настройка и тестване на приложенията и др. |
| **Категория** | (В контекста на PFM проекта)<br><br>За какво харчи потребителят парите си, разделено на групи (например категория „Разходи за храна“, категория „Разходи да дрехи“ и т.н.) ИЛИ от какво получава пари (например категория „Приходи от заплата“). |
| **Наличност** | Колко пари има потребителят (по някоя сметка или като цяло) |
| **Отчетен период** | Периодът, в края на който системата изчислява разликата между приходите и разходите; периодът, за който се бюджетира, т.е. се определят лимити за разходи по категории |
| **Приход** | Увеличаване на парите на потребителя |
| ПУК | "План за управление на качеството" |
| **Разход** | Намаляване на парите на потребителя |
| **Сметка** | Група заделени пари на потребителя (например спестовна сметка, сметка свързана с дебитна карта, сметка за PayPal средства и т.н.) |
| **Транзакция** | Операция свързана с парите на потребителя (приход, разход, трансфер) |
| **Трансфер** | Прехвърляне на парите на потребителя от една сметка в друга сметка |
| **Цел по сметка** | Сума, която потребителят е задал като очаквана и е сбор от наличностите по избраните от него сметки |

### *3.1.4. Материали за обучение*

Документът „Материали за обучение“ представлява кратко ръководство за потребителя, което показва как се ползва системата (уеб приложението и мобилното приложение) – как става регистрацията, как се редактира или добавя сметка и т.н. Освен инструкции под формата на текст, в документа има добавени и снимки за максимално улеснение на потребителя (снимките не са включени в записката, за да се спазят ограниченията за обема ѝ).

Ето самото ръководство в синтезиран вид:

#### 3.1.4.1. ***Регистрация в системата***

При зареждане на главната страница на системата се зарежда форма за вход, а най-отдолу има линк за регистрация.

След натискане на линка се появява форма за регистрация. Задължително е да се попълнят само полетата, които са маркирани със звезда, а останалите са по желание:

- Email (*)
- Име (*)
- Възраст
- Пол
- Семейно положение
- Образование
- Парола
- Потвърждение на паролата (*)

С натискането на бутона за запазване приключва регистрацията и сега може да се използват данните на създадения потребител за вход в системата.

3.1.4.2. ***Вход и изход от системата***

Входът в системата става от формата на главната страница на системата.

Попълват се email и парола на съществуващ потребител, а след това се натиска бутона за вход. След натискането на бутона се визуализира начална страница на потребителя, която съдържа обща статистика.

Може да се излезе от системата като потребителят кликне на името си в горния десен ъгъл и след това кликне на "Изход".

3.1.4.3. ***Сметки – добавяне, преглед и редактиране***

А) Добавяне на сметка

Добавянето на сметка става с кликване на меню "Сметки" и след това кликване на подменю "Добави сметка". При кликване върху "Добави сметка" се визуализира форма за добавяне на сметка. Задължително е да се попълнят само полетата със звездичка:

- Име на сметка (*)
- Текущ баланс (*)
- Цел на сметката

С натискане на бутона за добавяне приключва добавянето на сметката и тя вече може да се използва за извършването на транзакции.

Б) Преглед на сметка

Може да се прегледа информацията за дадена сметка като се кликне на меню "Сметки" и след това на подменю "Списък сметки". След това се визуализира списък със сметките на потребителя.

До всяка сметка има бутон за преглед на информацията й. При натискането му се вижда информацията за дадената сметка.

В) Редактиране на сметка

Може да се редактира информацията за дадена сметка като се кликне на меню "Сметки" и след това на подменю "Списък сметки". След това се визуализира списък със сметките на потребителя.

До всяка сметка има бутон за редактиране на информацията й. При натискането му се визуализира форма за редактиране на информацията на сметката.

В новопоявилата се форма може да се въведе нова информация за сметката. След натискане на бутона за запазване, информацията на сметката ще бъде обновена.

3.1.4.4. ***Категории – добавяне, преглед и редактиране***

А) Добавяне на категория

Добавянето на категория става с кликване на меню "Категории" и след това кликване на подменю "Добави категория". При кликване върху "Добави категория" се визуализира форма за добавяне на категория. Задължително е да се попълнят само полетата със звездичка:

- Име на категория (*)
- Тип на категория (*)
- Ограничение
- Тип на ограничението

С натискане на бутона за добавяне приключва добавянето на категорията и тя вече може да се използва при извършването на транзакции.

Б) Преглед на категория

Може да се прегледа информацията за дадена категория като се кликне на меню "Категории" и след това на подменю "Списък категории". След това се визуализира списък с категориите на потребителя.

До всяка категория има бутон за преглед на информацията й. При натискането му се вижда информацията за дадената категория.

В) Редактиране на категория

Може да се редактира информацията за дадена категория като се кликне на меню "Категории" и след това на подменю "Списък категории". След това се визуализира списък с категориите на потребителя.

До всяка категория има бутон за редактиране на информацията й. При натискането му се визуализира форма за редактиране на информацията на категорията.

В новопоявилата се форма може да се въведе нова информация за категорията. След натискане на бутона за запазване, информацията на категорията ще бъде обновена.

3.1.4.5. ***Транзакции – добавяне и преглед***

А) Добавяне на транзакция

Добавянето на транзакция става с кликване на меню "Транзакции" и след това кликване на подменю "Добави транзакция". При кликване върху "Добави транзакция" се визуализира форма за добавяне на транзакция. Задължително е да се попълнят само полетата със звездичка:

- Тип на транзакция (*)
- Сума (*)
- От (*)
- До (*)
- Описание

С натискане на бутона за добавяне приключва добавянето на транзакцията и наличността/сумата по засегнатите сметки/категории е обновена.

Б) Преглед на транзакция

Може да се прегледа информацията за дадена транзакции като се кликне на меню "Транзакции" и след това на подменю "Списък транзакции". След това се визуализира списък с транзакциите на потребителя.

До всяка транзакция има бутон за преглед на информацията й. При натискането му се вижда информацията за дадената транзакция.

3.1.4.6. ***Статистики – преглед и настройване***

А) Преглед на статистика за наличност по сметки и сума по категории

Може да се прегледа статистиката за сметките/категориите като се кликне на меню "Статистики" и след това на подменю "Преглед на статистики". След това се визуализира страница със статистиката.

Вижда се информация за общата наличност по сметките на потребителя и общата сума по категориите му. При задаване на начална и крайна дата може да се покаже статистика само за дадения период.

Б) Преглед на статистика за транзакции

Може да се прегледа статистиката за транзакциите като се кликне на меню "Статистики" и след това на подменю "Статистики за транзакции". След това се визуализира страница, от която може да се избере начална и крайна дата (и тип на транзакциите) за периода, за който да се генерира статистиката.

При натискане на бутон за обновяване се показва статистика за транзакциите извършени през посочения период.

3.1.4.7. ***Смяна на езика***

Системата може да бъде ползвана и на английски език. Това става с просто кликване върху иконата за език в горния десен ъгъл. По същия начин може да се превключи и отново към български.

3.1.4.8. ***Упътване за ползване на мобилното приложение***

Мобилното приложение работи на същия принцип като уеб системата, с основната разлика, че през него не могат да се въвеждат ръчно транзакции.

Като алтернатива на това има възможност за сканиране на баркод на касова бележка с камерата на устройството. Като се доближи камерата до такъв баркод, приложението автоматично разпознава данните на транзакцията и позволява добавянето й в базата данни.

## *3.2. Програмен код*

Кодът, който написах, е включен в Java проект с Maven & Spring Framework (по-точно Spring Boot със Spring Boot Starter Web). Този проект се състои от класовете, които са описани в документа „Дизайн модел“. Както вече споменах, това е проектът, който реализира back-end на системата. Освен документацията в „Дизайн модел“, има и коментари в самия код, които пояснят кое как работи.

Ето някои от най-важните технологии използвани в проекта и тяхното предназначение:

- Spring Data JPA (с Hibernate) - за максимално лесно манипулиране на данни в БД чрез Java код и преобразуване на Java класове в таблици в БД.
- Spring Security - framework предоставящ функционалност за защита на приложението – удостоверяване, упълномощаване и още.
- JJWT – библиотека за генериране и проверяване на JSON Web Tokens в Java проект
- H2 – база данни за тестовата среда
- PostgreSQL – база данни за средата за разработка и за продукционната среда
- Swagger 2 – генератор на документация за REST услугите
- DevTools – инструмент за по-лесна разработка на проекта (мигновено опресняване на файловете на сървъра, без да има нужда проектът да се стартира наново; по-детайлни съобщения за грешка в отговорите на REST заявките)
- Jackson - JSON процесор за Java
- Spring Boot Starter Test (с JUnit) – инструменти за максимално лесно тестване на приложението

Проектът може да се стартира като нормално Java приложение (main методът е в класа PersonalFinanceApplication, който пък е в пакета *com.mse.personal.finance*), но първо трябва да се изпълни Maven командата *mvn clean install*, която генерира имплементация за mapper интерфейсите (обяснени горе).

След стартиране на проекта може да се прегледа списък с възможни заявки и как трябва да са оформени. Това става с отварянето на следния адрес: http://localhost:8800/swagger-ui.html. Чрез софтуер като Postman може да се пращат заявки.

# 4. Възникнали проблеми и взети мерки за решаването им

## *4.1. Проблеми и решения при документацията*

Най-големият проблем в документирането, който аз лично срещнах, беше, че **не знаех какво точно трябва да бъде документирано и как точно**. Това е първият софтуерен проект, който документирам наистина сериозно, а затова не знаех каква точно работа се изисква да свърша. Отделно не знаех кога е удачно да започна да документирам – дали трябва да изчакам някой колега да завърши своя документ, от който аз да извлека информация? Или е по-добре да започна веднага и след това да надграждам малко по малко? Всеки, който питах, имаше различно мнение за това какво трябва да включва даден документ, как трябва да е структуриран и още. Накрая реших просто да прегледам примерни такива документи в интернет и стари магистърски проекти, по които да се водя. Използвах собствената си преценка, комбинирана с опита на хора, които са минали вече по този път, и просто добавих в документите това, което сметнах, че ще е най-полезно за хората, които ще ги четат.

Друг проблем беше **постигането на синхронизация с останалите членове на екипа**. Често промените в документацията на един човек се отразяват върху документацията на останалите. Например малко преди предаването на проекта се наложи да направя доста промени в документ „Тестов модел“, защото се оказа, че от добавените (или липсващи) тестови сценарии в този мой документ зависят документи на други членове от екипа. Открих, че най-доброто решение е проста комуникация. Питах другите какво трябва да добавя или да дам като информация, а след това просто предоставих каквото е необходимо и изгладих нещата. По същия начин процедирах, когато ми трябваше информация от документите на другите от екипа – казах им какво ми трябва и изчаках да ми отговорят, а след това просто продължих напред.

Трети съществен проблем беше, че **подцених колко време и усилия ще ми отнеме да напиша документацията**. Трудно е да изчислиш колко време ще ти отнеме да направиш нещо, което не си правил досега. Освен това когато започнеш работа, осъзнаваш, че често има доста повече неща за свършване, отколкото си предполагал. В резултат има доста работа по документацията, която не успях да завърша (описано по-подробно в следващата глава), защото не ми стигнаха времето и енергията. Мисля, че не се справих напълно успешно с този проблем, но „решението“, което приложих, беше всеки ден да работя по дипломната работа поне за около половин час. Имаше периоди, в които работих доста повече, а през други покривах само минимума, но се стараех на всяка цена всеки ден да влагам поне малко усилия за завършването на дипломната работа.

Имах и някои дребни технически проблеми като например „Как се прави X в Confluence?”, но тях ги реших лесно чрез най-обикновено проучване с помощта на Google.

## *4.2. Проблеми и решения при разработката*

В разработката мисля, че най-големият проблем, който срещнах, беше това **да преценя кои функционалности реалистично ще мога да имплементирам до зададения ни краен срок**. Някои от функционалностите ги бях имплементирал и в минали проекти (например защита на приложението, добавяне на прости CRUD операции), така че имах представа колко време ще ми отнеме да ги имплементирам отново, но за други си нямах идея. Общо взето реших просто често да давам отчети на другите от екипа за свършената от мен работа, за да имат представа как се справям, а след това да преценят те самите как да планират собствените си задачи.

Имах и трудности **при имплементирането на някои от функционалностите**. Някои функционалности изглеждаха сложни, но се оказаха прости за имплементиране, а други изглеждаха прости, но се оказаха сложни за имплементиране. Понякога се появяваше неочакван бъг, който ме забавяше и в резултат на това дълго време не можех да продължа работата си по имплементацията на дадена функционалност. Общо взето се справих с това като търсих решения в интернет за възникналите технически проблеми, а също така питах другите разработчици за помощ и/или временно превключвах към работа по други части от проекта (забелязвам, че временното отдалечаване от проблема ми помага след това по-лесно да се сетя за решение на този проблем).

Трети проблем беше, че **понякога не знаех какво точно трябва да имплементирам**. Водех се по user story-та, които бяха добавени в JIRA, но понякога липсваха детайли за това как точно трябва да работи дадената функционалност. Реших този проблем като направих справка със софтуерните изисквания и като питах останалите членове от екипа за мнение. Един ценен урок от това е, че ако не знаеш към какво се стремиш, няма как да го постигнеш. Ако не знаеш какво точно искаш да прави софтуерът, който разработваш, няма как да го имплементираш и да работи.

Друг проблем беше, че **подцених колко време ще ми трябва да имплементирам unit & integration тестове за back-end проекта**. Самото имплементиране не е сложен процес, ако човек е работил с JUnit преди (а аз бях), но просто писането на тестове отнема време и тъй като не бях тествал проект толкова обстойно досега, не прецених правилно колко време ще ми е необходимо. Общо имплементирах 390 теста, които тестват всеки метод в класовете реализиращи REST услуги и бизнес логика. Самите класове нямат чак толкова много методи и повечето методи в тях не съдържат някаква невероятно сложна логика, но въпреки всичко писането на тестове ми отне доста време. За това допринесе и фактът, че при самото тестване открих някои бъгове, които трябваше да отстраня. За да минимизирам времето за имплементация, използвах някои готови тестове, които съм писал за минали проекти с подобна функционалност. Също така използвах готовите тестове като основа за следващите (например използвах тестовете на сметките като основа за имплементацията на тестовете на категориите и просто промених кода където имаше необходимост, тъй като тестваната логика е сходна).

Имаше и някои други технически проблеми като например това, че от време на време се губеше връзката към предоставения ни от университета сървър, но като цяло това не беше голяма пречка, защото имах и локално копие на кода, върху който работя (и през това време просто работих локално)

## *4.3. Проблеми и решения в екипната работа*

Според мен най-големият проблем в екипната ни работа беше **слабата ни комуникация**. Проблемът беше, че не знаехме какво правят другите от екипа. Дали привършват с работата си? Дали не са забравили, че имат да свършат нещо? Дали пък всъщност не знаят как да свършат задачите си и нямат смелостта да помолят за помощ? Просто липсваше ефективна комуникация, чрез която според мен можехме да свършим проекта много по-бързо и по-ефективно. Разчитахме единствено на срещите ни в университета и на седмични конферентни разговори в Skype, за да следим прогреса си, но това мисля, че не беше достатъчно (особено предвид извънредната ситуация, която допълнително ограничи срещите ни на живо). Също така когато някой попиташе нещо в общия Skype чат, често отговорите идваха прекалено бавно (или въобще не идваха).

Друг проблем, който е свързан с горния, беше това, че ни **липсваше система за отговорност**. Липсваше напрежение да работим усилено. Според мен не е добре да има прекалено голямо напрежение върху хората, които работят, защото това ги стресира и понижава качеството на труда им, но не е добре и да няма *никакво* поставено напрежение, защото така те се отпускат прекалено (оптималният вариант е нещо по средата). Първоначално идеята беше да правим редовни (или поне седмични) „Scrum срещи“, в които набързо споделяме какво сме свършили, по какво работим в момента и кое ни затруднява, обаче на практика това не се случи. Планът беше да използваме седмичните Skype разговори за тази цел, но реално почти винаги липсваше поне един от членовете на екипа и не беше ясно как върви работата му. Всеки просто тихичко си работеше (или бездействаше), а по някое време изведнъж всички разбирахме, че има много несвършена работа и оставащото ни време за свършването ѝ е много малко. Най-съществения пример за това, който се сещам, е случая, в който единият от разработчиците заяви към края на проекта, че няма време и желание да изпълни възложените му задачи по разработката на front-end проекта. В резултат на това трябваше друг разработчик да поеме задачите му, но дори и така нямаше достатъчно време всичките задачи да бъдат изпълнени. Освен това почти никой не ползваше Jira, за да отчете задачите, които е свършил и по които работи в момента. Просто екипът ни не изпитваше достатъчно напрежение да свърши задачите си навреме, защото нямаше система за отговорност (или ако е имало, тя е била много слабо приложена).

Трети проблем според мен бяха **нереалистичните очаквания на екипа ни**. Всички от екипа бяха прекалено оптимистично настроени по отношение на това колко време ще ни отнеме да завършим проекта, кои функционалности ще можем да имплементираме и още. На практика всичко отне много повече време от очакваното и реализирахме много по-малко функционалности от тези, които обещахме на възложителя. Като решение поместихме голяма част от функционалностите, които не реализирахме, в категория „nice to have”, а се фокусирахме само върху най-важните функционалности на системата и ги маркирахме като приоритетни. Това обаче според мен беше решение породено от отчаяние и истинското решение би било още в самото начало да не вдигаме прекалено много очакванията на възложителя и да бъдем реалистични по отношение на това колко време ще ни отнеме разработката на проекта и какво можем да свършим за този период (особено като се има предвид, че повечето членове на екипа работят на пълно работно време). Според мен е много по-добре хората да имат

за теб ниски очаквания, които да надхвърлиш, отколкото да имат високи очаквания и да ги разочароваш след като не отговориш на тях.

Още един според мен подценяван проблем е **липсата на страст в проекта**. Мисля, че повечето от нас нямаха голяма страст за проекта, по който работихме. Мотивацията ни не беше да реализираме наистина изключителен продукт, а просто да изпълним дипломната си задача. Това мисля, че също допринесе за забавянето в работата по проекта и за многото нереализирани функционалности. Когато човек прави нещо от страст, той е готов да даде всичко от себе си, за да може крайният продукт да е максимално добър и полезен. Ако няма тази страст, човек не дава всичко от себе си и затова крайният продукт е посредствен. В началото на проекта мисля, че всички бяхме доста мотивирани да направим нещо наистина качествено, но постепенно тази мотивация се загуби и затова започнаха да се появяват голяма част от проблемите в работата ни.

## *4.4. Лично виждане за предотвратяване и решаване на проблеми*

Ето списък с някои мерки, които бих въвел, за да може да се предотвратят или решат голяма част от споменатите горе проблеми (най-вече тези в екипната работа, защото проблемите при имплементацията и документацията общо взето си имат ясни решения, с които вече сме запознати).

1) Реалистични очаквания. Мисля, че е по-ефективно да работим с доза реализъм или даже лек песимизъм по отношение на това колко време ще ни отнеме да завършим проекта или някоя задача в него, и кои функционалности ще можем да реализираме за времето, с което разполагаме. Ако например си мислим, че нещо ще ни отнеме седмица, нека по-добре да кажем, че ще ни отнеме две седмици. По този начин минимизираме шанса да попаднем в ситуация, в която не сме изпълнили работата си навреме или не сме изпълнили всичко, което сме казали, че ще изпълним. За постигането на това е необходимо всеки член от екипа да се замисли дълбоко за силните и слабите си страни, а след това да прецени и да сподели колко точно може да допринесе за реализирането на проекта.

2) Увеличаване и подобряване на комуникацията в екипа. Това включва по-активно използване на писмена комуникация и по възможност по-чести срещи на живо с екипа. Предимството на писмената комуникация е, че написаното може лесно да бъде прочетено и разбрано по всяко време. Дори и член от екипа да не може да присъства на някой конферентен разговор, пак може с 2 думи да напише в чата какво е свършил и по какво работи, а и да разбере как се справят другите. Да, разговорите също могат да бъдат записвани и прослушвани след това, но от тях по-трудно и по-бавно се извлича съществената информация.

3) Въвеждане на система за отговорност. Никой не иска да е единствения, който не върши никаква работа. Ако всеки започне по-често да отчита това, което е свършил и това, по което работи в момента по проекта, мисля, че мотивацията на всеки от екипа да работи усилено ще се увеличи значително. Мисля, че най-лесният начин това да се случи е като се използва Jira по-активно и в чата редовно всеки написва с по няколко думи как върви прогресът му. Дори и някой от екипа да няма страст за проекта, който разработваме, този човек пак няма да иска да мързелува, защото не желае да се излага пред другите.

# 5. Визия за бъдещето на проекта

В тази глава ще споделя още някои неща, които мисля, че могат да бъдат поправени или добавени в проекта, за да стане по-качествен (най-вече от гледна точка на код/функционалност, документация и екипна работа).

## *5.1. Промени в кода и функционалността*

### *5.1.1. Промени в собствения ми код*

На практика не успях да изпълня всичко, което исках да направя по back-end проекта (основно заради неправилно разпределение на времето и енергията ми), но все пак имам някаква идея какво бих подобрил.

1) Бих добавил логика за отчетни периоди. От тази логика зависи реализирането на голяма част от функционалностите, които все още не са имплементирани (бюджетиране, изчисляване на прогнозна дата за постигане на цел и други).

2) Бих използвал тип BigDecimal тип за полетата, които пазят парични суми. В момента тези полета са от тип Double (по изискване в „Модел на данните“), но така се появяват проблеми с прецизността. BigDecimal е по-подходящ тип за такива данни.

3) Бих подобрил съобщенията за грешка, които се връщат от REST заявката, когато клиентът се опитва да създаде транзакция с невалидни „from” и “to” данни. В момента се изпраща едно общо съобщение за невалидни „from” и “to” данни, но тъй като проблемът може да е породен от доста неща (липсваща „from“ сметка, изтрита „from” сметка, липсваща „to” категория“ и т.н.), съобщенията за грешка може да са много по-информативни.

4) Бих добавил локализация (съобщения от REST услугите на български). В момента REST API-то връща съобщения само на английски. Проблемът с това е, че когато възникне грешка във front-end проекта, появилото се съобщение отново е на английски (а пък интерфейсът на приложението е на български и няма логика съобщенията за грешка да са на английски). Това може да бъде решено и във front-end проекта, но мисля, че е хубаво да е реализирано и в back-end проекта.

5) Бих подобрил валидацията на данните. Смятам, че като цяло в момента добавената валидация на данните е добра, но все пак има някои малки неща, които могат да бъдат изгладени. Например мисля, че за приходни категории в системата не трябва да има ограничения за постъпилата сума, а затова може да се добави някаква валидация.

Също така в класа ServiceUtils има метод, който определя дали дадена сметка/категория/транзакция принадлежи на потребителя, който в момента е влязъл в системата. Този метод предполага, че в момента ИМА влязъл в системата потребител (тъй като методът никога не бива извикван, ако това условие не е налице, макар unit & integration тестовете, които го използват, да са изключение). Затова методът хвърля NullPointerException, когато няма влязъл потребител. Вместо да се хвърля изключение, може просто да се провери в началото на метода дали в момента има влязъл потребител в системата.

6) Бих преработил кода на back-end тестовете и бих добавил още тестове. В момента повечето тестове използват assertTrue() метода за проверка дали дадено условие е изпълнено. Бих заменил това с assertEquals(), когато се проверява някаква стойност на поле, което не е от булев тип. Предимството на това е, че по-лесно се вижда къде даден тест се проваля, ако има несъответствие в очакваната и реалната стойност на някое поле.

Също бих запазвал всички hard-coded стойности в променливи, които просто да използвам, когато правя някакво сравнение (в противен случай тези стойности се повтарят отново и отново, а ако поради една или друга причина се измени стойността, трябва да се правят промени на много места и поддръжката на тестовете става трудна).

Има и много методи, чиято имплементация се повтаря в различните класове за тестване (например методи за вмъкване на примерен потребител в системата, на примерна сметка и така нататък). Най-удачно мисля, че ще е да се създаде един клас TestUtils с методи, които да са достъпни за всички класове, които съдържат имплементирани тестове.

Освен това макар да има имплементирани доста тестове за back-end проекта, смятам, че могат да бъдат добавени още, които тестват логиката по-обстойно.

7) Бих имплементирал интеграционни тестове за front-end проекта. Не ми остана време да направя това, но като тестер мога да имплементирам автоматизирани интеграционни тестове и за front-end проекта чрез Cypress (unit тестовете бих ги оставил за front-end разработчиците, които са писали кода).

8) Бих имплементирал останалите функционалности. Някои от функционалностите, които все още не са имплементирани, смятам, че първо трябва да бъдат обсъдени с другите членове на екипа (например споделените сметки и прегледа на прогнозна дата за достигане на цел). Обаче други от функционалните изисквания останаха нереализирани просто заради неправилното разпределение на времето и енергията ни и защото не се считаха за приоритетни, когато всъщност можеха да бъдат направени бързо и без да струват прекалено много усилия на екипа като цяло (например входа и

регистрацията чрез Facebook/Google). Бих се фокусирал върху това да завърша вече добавените в изискванията функционалности, а чак след това бих помислил за добавяне на други новости в системата, които все още не са ясно определени.

### *5.1.2. Промени в кода на другите*

Не мога да коментирам прекалено много кода и имплементираното от другите, защото не съм го анализирал в голяма дълбочина, но все пак мога да споделя някои неща, които ми направиха впечатление, когато правих ревюта и преглеждах добавеното в Bitbucket, и някои проблеми, които бих искал да видя решени.

1) Бих добавил документация в самия код. Направи ми впечатление, че другите разработчици не документират кода си (или ако има някои коментари, те обикновено са повърхностни и не обясняват добре това, което се опитват да обяснят). Така става доста трудно някой външен разработчик да разбере механизма на работа на функционалностите, които са реализирани. Според мен е хубаво да има коментар както за всеки клас, така и за всеки съществен имплементиран метод.

2) Бих добавял свършената си работа по-често в Bitbucket. Забелязах също, че другите разработчици не публикуват кода си, ако вече не е в окончателен вид (тоест не добавят често commits в проектите си, а направо пускат pull request, когато са готови с това, което разработват). Не е задължително да публикуват всичко, разбира се, но мисля, че може да е доста полезно, защото става по-лесно да се следи прогреса в разработката на проекта и да се отстраняват различни бъгове. Освен това на мен лично ми помага, защото ми дава някаква представа за това как върви работата на другите разработчици и по този начин мога по-добре да планирам своята. Ако те не публикуват нищо, не знам какво точно се случва с кода и дали има някакъв напредък, а понякога това може да затрудни работата ми (например аз имплементирам различни функционалности по back-end, но не знам дали работят през front-end проекта и дали трябва да променя нещо, за да е успешна интеграцията).

3) Бих спазвал по-добре конвенциите за качествен код. „Качествен код“ може да се каже че донякъде е субективно понятие, но според има някои принципи, които са универсални и трябва да се спазват. Смислени имена на променливите, правилно

форматиране, стремеж към писане на четлив код и така нататък. На доста места ми се стори, че тези правила не са спазени.

4) Бих променил някои неща по интерфейса на приложението. На места в интерфейса на уеб приложението забелязах някои неточни изрази (например при регистрация се използва „Години“ вместо „Възраст“). Мобилното приложение пък е изцяло на английски. На места има следи от програмен код (например в БД типът на една активирана сметка се пази като „ACTIVATED“ и front-end показва типа й като „ACTIVATED” в списъка със сметки на потребителя, а по-добре би било да пише „активирана“). Изглаждането на малки неща като тези може да направи проекта по-качествен и успешен като цяло.

## *5.2. Промени в документацията*

### *5.2.1. Промени в собствената ми документация*

По документацията си също забелязах някои пропуски, които бих искал да изгладя. Пропуските се изразяват по-често в това, че може да се добави още информация по документите, отколкото в това, че съществуващата информация в тях е грешна.

1) Бих доразвил документ „Дизайн модел“ с обновени диаграми. Дори малки промени по кода малко или много изменят структурата на клас диаграмите и sequence диаграмите. Бих обновил тези диаграми, за да може да са максимално актуални и да отговарят на финалната версия добавена в Bitbucket (в момента също са общо взето актуални, но има някои разминавания в имена на методи, подавани параметри в тези методи и така нататък).

2) Бих доразвил документ „Тестов модел“ с добавяне на още тестови сценарии. Сегашните добавени тестови сценарии в документа просто не са достатъчни. Може да се добавят още много такива за функционалните изисквания (дори и някои от тези изисквания да не са имплементирани все още). За нефункционалните изисквания също може да се добавят тестови сценарии, а в момента липсват изцяло.

3) Бих доразвил документ „Речник“ с още добавени термини. Понеже в проекта използваме доста съкращения и технологии, според мен може да се добавят още термини в речника, които да направят документацията по-лесна за разбиране. Също може да се напишат малко по-ясно някои от съществуващите определения (особено тези, които обясняват какво представлява дадена технология и за какво се използва).

4) Бих доразвил документ „Материали за обучение“ с актуализиране на информацията в него и добавяне на малко по-подробна информация за ползването на мобилното приложение. Заради проблемите описани в предишната глава, голяма част от функционалността на front-end проекта на системата липсваше почти до крайния срок за предаване на системата. Освен това интерфейсът се изменяше рязко и често (менютата придобиваха нови имена и ново разположение, формите за попълване на данни се променяха, добавяха се функционалности и се махаха такива). Затова написах документ „Материали за обучение“ в последния момент (за да е максимално актуален, тъй като ако интерфейсът се обнови, снимките и информацията в документа също трябва да се обновят). Имайки това предвид, може да има леки разминавания между написаното и това, което присъства във финалната версия на проекта в Bitbucket, а следователно може да има нужда и от периодично обновяване на ръководството за ползване на системата (дори и механизмът на работа на системата да не се е изменил съществено). По инструкциите за ползване на мобилното приложение също може да се добави още малко информация и леко да се изрежат части от добавените снимки. Може също да се добавят кратки видеа, които показват как се ползва системата.

### *5.2.2. Промени в документацията на другите*

Както и при кода, не съм анализирал в огромна дълбочина документите на другите, но ми направиха впечатление някои неща, докато преглеждах документацията като цяло. Ако се изгладят следните пропуски, мисля, че качеството на документацията ще се увеличи (а в резултат и качеството на проекта също).

1) Бих обръщал повече внимание на детайлите. Направи ми впечатление, че на много места в документите има правописни грешки, граматически грешки, неточно изписани имена на технологии (нещо като „Typescript” и “Spring boot” вместо „TypeScript” и “Spring

Boot”), неточни изрази останали от copypaste и още. Ако се обърне внимание на тези малки детайли, според мен документацията ще бъде много по-добра.

2) Бих следял дали документите са в съответствие с другите документи и с последната версия на системата. Това е доста предизвикателна задача, защото документите се менят постоянно, а и системата също. За постигането на тази цел мисля, че е най-удачно всеки член на екипа периодично да преглежда последната версия на системата и също така онези чужди документи, които са пряко свързани с документите, които той самият поддържа (например човекът, който отговаря за документа за резултати от тестването, да преглежда документ „Тестов модел“ периодично, за да не изпусне да тества с някой наскоро добавен тестов сценарий). Също така мисля, че отново би помогнало въвеждането на система за отговорност – така всеки лесно може да сподели промените, които е направил по своите документи, а и да разбере за промени по документите на другите.

## *5.3. Промени в работата в екип*

В предишната глава споменах основните промени, които бих въвел в работата в екип (**реалистични очаквания, подобряване на комуникацията и въвеждане на система за отговорност**). Това са мерки, които бих въвел още от самото начало на проекта, но дори и това да не е възможно, въвеждането им в по-късен етап според мен пак би довело до положителен резултат (както се казва „Най-доброто време да засадиш дърво е преди 20 години. Следващото най-добро време е днес.”).

Още една промяна, която според мен би довела до положителен резултат, е **да се разпредели работата по проекта наново**. Смятам, че в първата фаза на проекта някои членове на екипа имаха малко работа за вършене, а други прекалено много. Ако работата се разпредели малко по-равномерно, мисля, че продуктивността ни ще се увеличи и качеството на проекта също.