Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Современные платформы программирования

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

Программное средство по учёту доходов и расходов с прогнозированием «Kwit»

БГУИР КП 1-40 01 01 148 ПЗ

Студент: гр. 451006 Снитовец М. В.

Руководитель: Дубко Н. А.

Минск 2017

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой ПОИТ

Лапицкая Н.В.

(подпись)

2017 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Снитовцу Михаилу Владимировичу

1. Тема работы Программное средство для учёта расходов и доходов с прогнозированием

2. Срок сдачи студентом законченной работы 12.06.2017

3. Исходные данные к работе Среда программирования Intellij IDEA

4. Содержание расчётно-пояснительной записки (перечень вопросов, которые подлежат разработке):

Введение;

1. Анализ предметной области;

2. Используемые технологии;

3. Проектирование программного средства;

4. Описание классов и методов;

Заключение.

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Схема базы данных программного средства

6. Консультант по курсовой работе Дубко Н. А.

7. Дата выдачи задания 10.02.2017 г.

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и процентом от общего объёма работы):

введение к 20.02.2017 – 10 % готовности работы;

раздел 1 к 15.03.2017 – 30 % готовности работы;

раздел 2 к 02.04.2017 – 60 % готовности работы;

раздел 3 к 26.04.2017 – 60 % готовности работы;

раздел 4 к 04.05.2017 – 90 % готовности работы;

оформление пояснительной записки и графического материала к 12.05.2017- 100 % готовности работы.

Защита курсового проекта с 12 мая 2017.

РУКОВОДИТЕЛЬ Дубко Н. А.

(подпись)

Задание принял к исполнению М. В. Снитовец 10.02.2017 г.

(дата и подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

За 2016 год потребительские расходы домашних хозяйств Беларуси по категории «прочие товары и услуги» составляет 8.7% от общего дохода [1]. С 2012 года данный показатель вырос на один процент. Такие, казалось бы, небольшие числа говорят о том, что с каждым годом контроль над расходами белорусов постепенно снижается.

Среди общества присутствует проблема контроля за расходами, учётом целей трат и их объемов. Для решения данной проблемы с течением времени использовались разные подходы, такие как простая запись раходов и доходов в тетрадь или ведение таблиц Excel. С развитием технологий разработки мобильных и веб-приложений стали появляться отдельные средства для полного контроля над личными финансами.

Широкое применение в области учёта финансов нашли аналитические технологии для прогнозирования и анализа данных, прогнозирование на основе эконометрических, регрессионных и нейросетевых алгоритмах.

Целью данного курсового проекта является разработка веб-приложения для простого и удобного ведения учёта личных доходов и расходов с внедрением возможностей оценки текущего баланса и прогнозированием будущих раходов. Внимание планируется сконцентрировать на простоту использования и прослеживания текущего состояния счетов.

Задачи, которые предполагается решить в рамках курсового проекта:

- изучение и обобщение знаний о языке программирования Koltin;
- изучение арихитектуры REST и примение её в расзработке API;
- изучение различных алгоритмов прогнозирования временных рядов;
- изучение и использование программной платформа AngularJS.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

В данном разделе будет произведен краткий обзор существующих аналогов приложения; сформулированы требования к разрабатываемому программному средству.

1.1 Обзор аналогов

В результате анализа предметной области было выявлено большое количество приложений по личному учёту доходов и раходов. В данном подразделе будут приведены три ярких представителя.

Основными критериями сравнения являются:

– пользовательский интерфейс;
– удобство и быстрота использования;
– возможность категоризации тразакций;
– работа со счетами;
– возможность просмотра статистики;
– отображение текущего состояния счетов.

Классическим приложением по учёту доходов и расходов является система «Семейный бюджет», расположенная по адресу https://koshelek.org. Данная система предоставляет широкий функционал по контролю личного бюджета. Предоставляет большие возомжности по генерации отчётов, планированию будущих расходов. Однако в следствии широких возможностей системы пострадал пользовательский интерфес. Внешний вид сайта устарел (рисунок 1.1), что также сказывается и на удобстве использования. Также сайт перегружен множеством вложенных меню и мелких, неочевидных иконок, что затрудняет работу с ним неподготовленному пользователю.

Кроме этого существует система под названием «Drebedengi» расположенная по адресу http://drebedengi.org (рисунок 1.2). Данная система позволяет вести учёт доходов и расходов, перемещений между счетами, планировать бюджет и и контролировать текущее состояние счетов. Также приложение позволяет контролировать долги. Пользовательский интерфейс более приятный, по сравнению с «Семейным бюджетом». Однако всё равно присутствует сложность восприятия из-за большого количества чисел.

Третий аналог, отличающийся от описанных выше отсутсвием перегруженного интерфейса — система «Zenmoney» (http://zenmoney.ru). Данное приложение позволяет работать с личными счетами, категориями, транзакциями. Предоставляет возможности генерировать отчёты, планировать бюджет, просматривать прогноз баланса. Главным недостатком данного

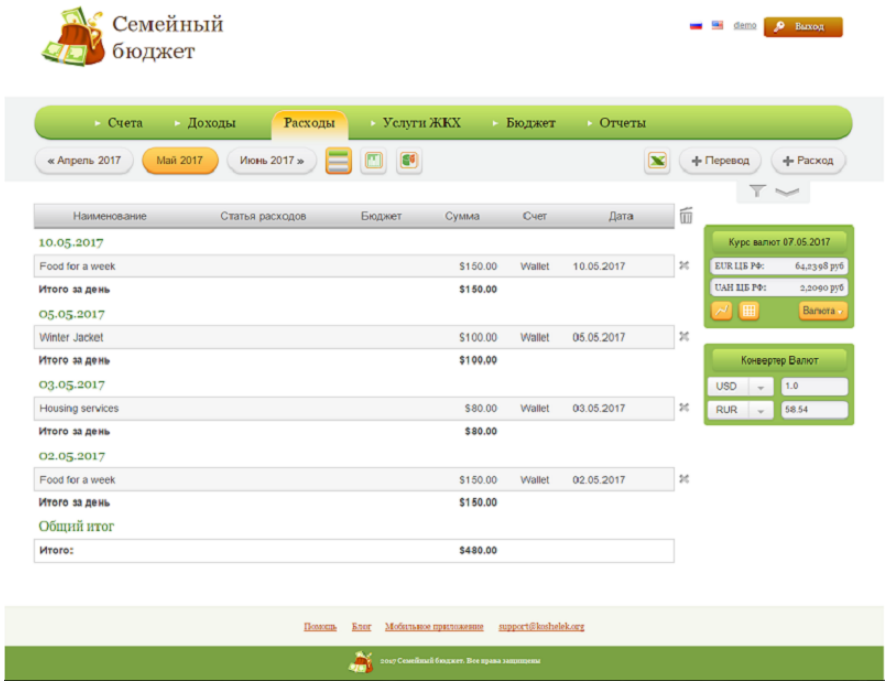

Рисунок 1.1 – Раздел «Доходы» сайта системы «Семейный бюджет»

приложения является внешний вид приложение — большинство элементов интерфейса предоставлены без какой-либо стилизации (рисунок 1.3).

Результат сравнения имеющихся аналогов с разрабатываемым приложением приведён в таблице 1.1.

Таблица 1.1 – Сравнение приведеных аналогов с разрабатываемым приложением

Функция	Kwit	Koshelek.org	Drebedengi.ru	Zenmoney.ru
Современный интерфейс	+	-	+	-
Простота в использовании	+	-	-	+
Мультивалютность	+	+	+	+
Простота в использовании	+	-	-	+
Работа со счетами	+	+	+	+
Работа с категориями	+	+	+	+
Возможность генерировать отчёты	-	+	+	+
Прогнозирование	+	+	+	+

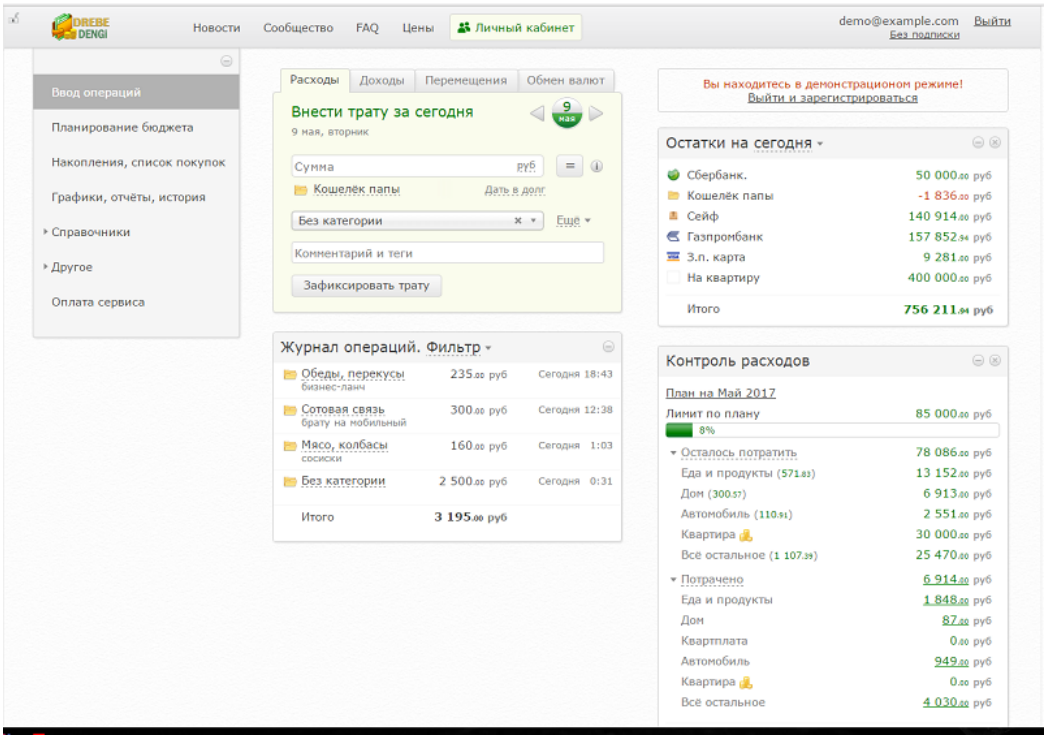

Рисунок 1.2 – Главная страница пользователя сайта системы «Drebedengi»

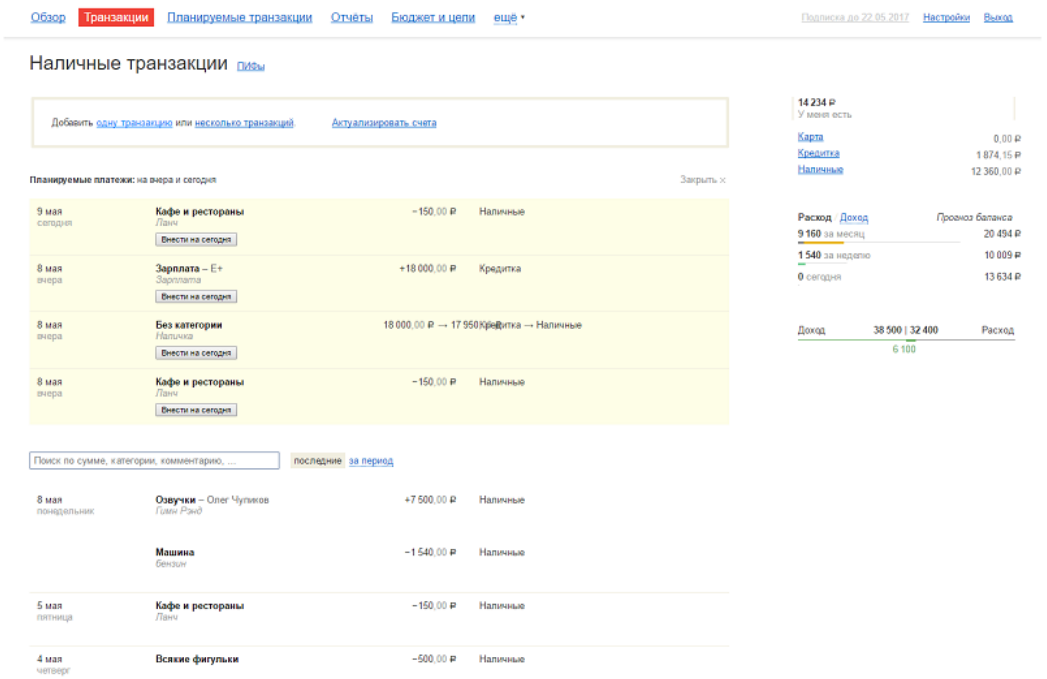

Рисунок 1.3 – Страница транзакций сайта системы «Zenmoney»

При составлении таблицы учитывался весь запланированный функционал, реализация некоторых функций возможна в версиях, которые будут разработаны вне данного курсового проекта.

1.2 Постановка задачи

Целью данного курсового проекта является разработка:

– добавление, удаление и изменение транзакций;

– добавление, удаление и изменение категорий;

– добавление, удаление и изменение счетов;

– возможность удаления категорий и счетов с переносом всех транзакций на другой счёт;

– возможность подсчёта статистики по категориям за произвольный период всемени;

– подсчёт ежедневной суммы до зарплаты, прогноза будущих затрат за последнее время.

Программное средство должно представлять собой сервер с REST API и веб-клиент. Данное сочетание позволит в дальнейшем развить данное приложение в полноценную кроссплатформенную систему.

2 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

В данном разделе будет приведено описание языков программирования, фреймворков, используемых при разработке.

2.1 Язык программирования Koltin

Koltin (Котлин) — статически типизированный язык программирования, работающий поверх JVM и разрабатываемый компанией JetBrains.

Основные приемущества языка Koltin:

– краткость — кострукции и возможности языка разрабатывались с целью уменьшить количество кода, но при этом не уменьшая читаемости этого кода;

– поддержка защиты от NullPointerException на уровне языка;

– полная совместимость с языком программирования Java, что позволяет использовать весь набор библиотек и технологий, накопленных за долгое время существования Java;

– возможность расширять библиотеки, не изменяя их код, что позволяет дописывать необходимую функциональность без нарушений каких-либо лицензий либо поиска исходников уже скомпилированных библиотек (листинг 2.1);

– мультипарадигмость — Koltin можно писать код как в процедурном стиле, так и в объектно-ориентированном, а также благодаря поддержке функций высшего порядка можно писать код и в функциональном стиле. Сочетание лучших качеств у каждого из этих стилей позволяет разрабатывать приложения быстро и эффективно (листинг 2.2);

Листинг 2.1 – Пример Extension-функции на языке Koltin

```
public fun CharSequence.padEnd(length: Int, padChar: Char = ' '): CharSequence
    {
   if (length < 0)
      throw IllegalArgumentException("Desired length $length is less than
         zero.")
   if (length <= this.length)
      return this.subSequence(0, this.length)

   val sb = StringBuilder(length)
   sb.append(this)
   for (i in 1..(length - this.length))
      sb.append(padChar)
   return sb
}
```

Листинг 2.2 – Реализация популярной функции высшего порядка «map» на языке Koltin

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
   val result = arrayListOf<R>()
   for (item in this)
      result.add(transform(item))
   return result
}
```

2.2 Spring Framework

Spring — универсальный фреймворк с открытым исходным кодом для Java-платформы. Spring представляет собой набор из большого количества модулей и расширений, позволяющих эффективно и быстро разрабатывать приложения. Приемущественно он связан с платформой Java Enterprise.

Spring может быть рассмотрен как коллекция меньших фреймворков. Большинство этих фреймворков может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Можно выделить следующие фреймворки, которые делятся на структурные элементы типовых комплексных приложений:

– Inversion of Control-контейнер — конфигурирование компонентов приложений и управление жизненным циклом Java-объектов, поддержка JSR-299;

– фреймворк аспектно-ориентированного программирования (AOP) — работает с функциональностью, которая не может быть реализована возможностями объектно-ориентированного программирования на Java без потерь;

– фреймворк доступа к данным: работает с системами управления реляционными базами данных на Java-платформе, используя JDBC- и ORM-средства и обеспечивая решения задач, которые повторяются в большом числе Java-based environments, обеспечивает широкую поддержку Java Persistence API (JPA);

– фреймворк управления транзакциями: координация различных API управления транзакциями и инструментарий настраиваемого управления транзакциями для объектов Java;

– фреймворк MVC: каркас, основанный на HTTP и сервлетах, предоставляющий множество возможностей для расширения и настройки;

– фреймворк аутентификации и авторизации: конфигурируемый инструментарий процессов аутентификации и авторизации, поддерживающий много популярных и ставших индустриальными стандартами протоколов,

инструментов, практик через дочерний проект Spring Security

В данном курсовом проекте задействованы следующие модули Spring:

– Spring Boot — проект для автоконфигурации и работы с другими модулями самого Spring;

– Spring Core — базовый модуль, включающий в себя IoC-контейнер;

– Spring MVC — требуется для быстрой и качественной реализации REST API;

– Spring Security — обеспечивает защиту ресурсов сервера для неавторизованных пользователей, а также авторизация по протоколу OAuth 2.0;

2.3 AngularJS

AngularJS — JavaScript-фреймворк с открытым исходным кодом. Предназначен для разработки одностраничных приложений. Его цель — расширение браузерных приложений на основе MVC-шаблона, а также упрощение тестирования и разработки.

Фреймворк работает с HTML, содержащим дополнительные пользовательские атрибуты, которые описываются директивами, и связывает ввод или вывод области страницы с моделью, представляющей собой обычные переменные JavaScript. Значения этих переменных задаются вручную или извлекаются из статических или динамических JSON-данных.

Angular придерживается MVC-шаблона проектирования и поощряет слабую связь между представлением, данными и логикой компонентов. Используя внедрение зависимости, Angular переносит на клиентскую сторону такие классические серверные службы, как видозависимые контроллеры. Следовательно, уменьшается нагрузка на сервер и веб-приложение становится легче. Арихитектура клиентского приложения с использованием AngularJS представлена на рисунке 2.1

Благодаря популярности данного фреймворка существует большое количество пользовательских библиотек (2084, по информации с сайта http://ngmodules.org/), которые позволяют ускорить разработку приложения и расширить его функциональность.

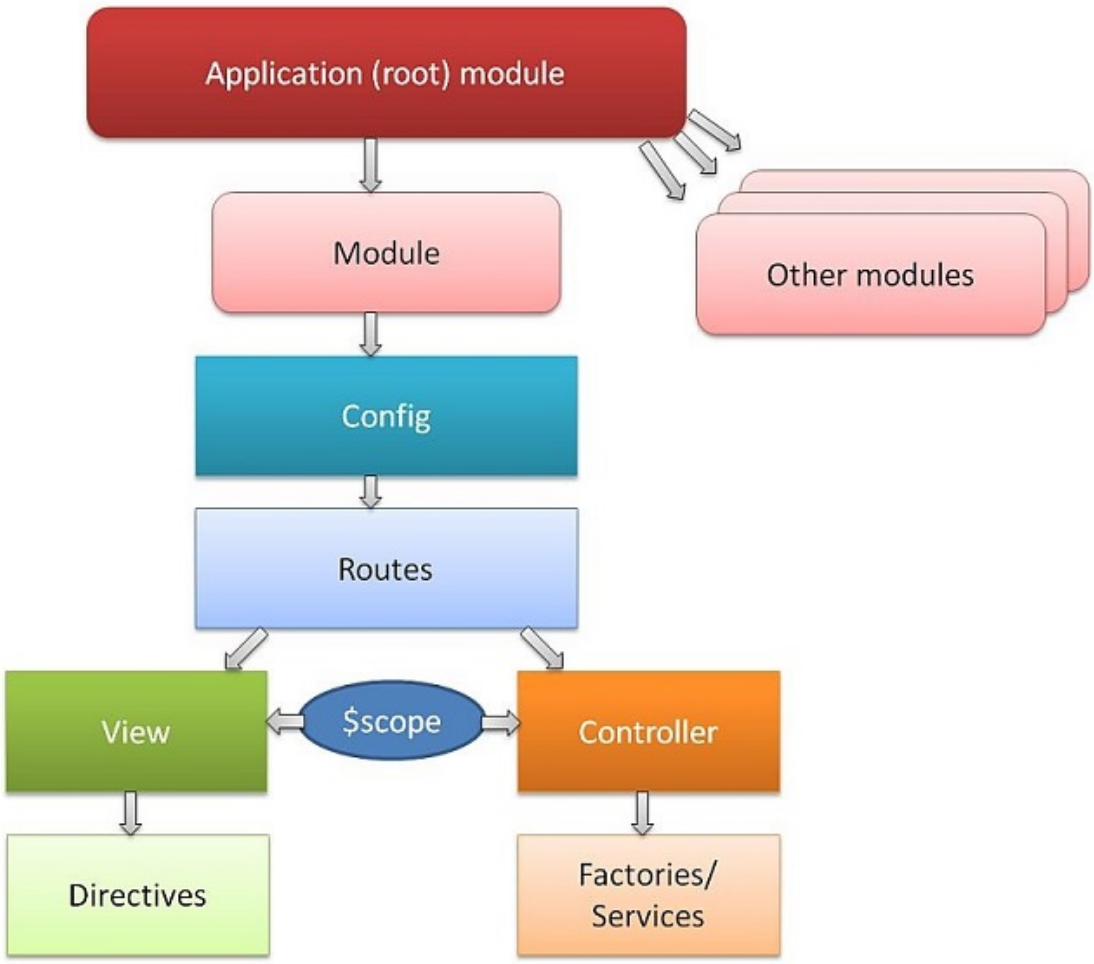

Рисунок 2.1 – Архитектура приложения с ипользованием AngularJS

3 ПРОЕКТИРОВАНИЕ

В данном разделе будет описан процесс проектирования основных модулей программного средства.

3.1 Общее описание серверной архитектуры

Главная задача сервера — предоставить API для работы веб-клиента и, возможно, приложений на других платформах. Данную задачу эффективно решает архитектурный стиль взаимодействия компонентов распределённого приложения REST.

REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы, такие как отсутсвие состояния, единообразие интерфейса, идентификация ресурсов в запросах.

Разработка приложения с помощью данной архитектуры позволяет добиться расширяемости API, уменьшении нагрузки на сервер за счёт вынесения большьшинства логики отображения и обработки данных на веб-клиент.

Общая структура работы клиент-серверного приложения с ипользованием архитеркуты REST представлена на рисунке 3.1.

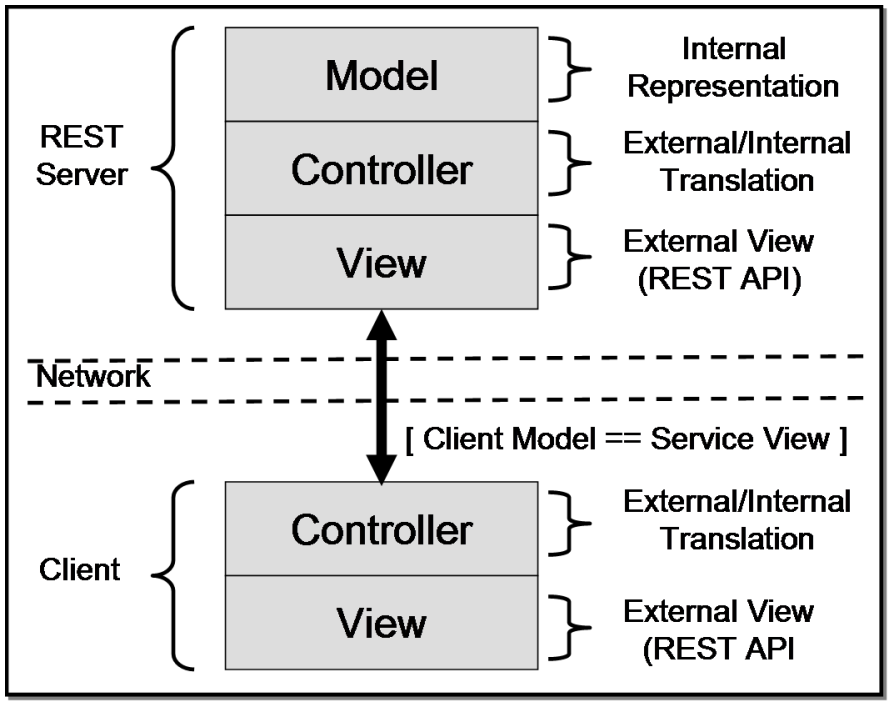

Рисунок 3.1 – Структура работы клиент-серверного приложения с ипользованием REST архитектуры

С точки зрения программной архитектуры приложения, было принято решение использовать классическую монолитную архитектуру, имеющую следующие слои:

– слой контроллеров — часть сервера, предоставляющая API клиенту и принимающая запросы от него;

– слой сервисов — основная часть серверной части приложения, содержащая бизнес логику приложения, обработку ошибок и работу с базой данных;

– слой репозиториев — абстракция над базой данных, предоставляющая возможности работать с ней остальной части приложения.

Приемуществами монолитной архитектуры является скорость разработки, простота, расширяемость. Данная архитектура отлично походит для малых и средних приложений, каким и является данный курсовой проект.

3.2 Функции прогнозирования

Одной из важных особенностей разрабатываемого приложения является функция прогнозирования.

Среди классов методов прогнозирования можно выделить следующие:

– методы, основанные на сглаживании, экспотенциальном сглаживании и скользящем среднем;

– регрессионные методы прогнозирования;

– нейросетевые модели бизнес-прогнозирования.

Для реализации функции прогнозирования в рамках предметной области решено остановиться на методах, основанных на сглаживании, экспотенциальном сглаживании и скользящем среднем. Данные методы отличаются простотой реализации, лёгкостью вычислений и умеренной точностью прогноза, чего вполне достаточно для решения поставленных задач.

Самой простой моделью прогнозирования является

$$Y_{t+1} = Y_t, \tag{1}$$

что соответствует предположению, что «завтра будет как сегодня». Она не только не учитывает механизмы, определяющие прогнозируемые данные (этот серьезный недостаток вообще свойственен многим статистическим методам прогнозирования), но и не защищена от случайных флуктуаций, она не учитывает сезонные колебания и тренды.

Моделью, основанной на простом усреднении является

$$Y_{t+1} = \frac{1}{T+1}(Y_t + Y_{t-1} + ... + Y_{t-T}), \tag{2}$$

и в отличии от самой простой модели, описанной уравнением (1), которой соответствовал принцип «завтра будет как сегодня», этой модели соответствует принцип «завтра будет как было в среднем за последнее время». Такая модель более устойчива к флуктуациям, поскольку в ней сглаживаются случайные выбросы относительно среднего.

В приведенной выше формуле (2) предполагалось, что ряд усредняется по достаточно длительному интервалу времени. Однако? как правило, значения временного ряда из недалекого прошлого лучше описывают прогноз, чем более старые значения этого же ряда. Тогда можно использовать для прогнозирования скользящее среднее:

$$Y_{t+1} = \frac{1}{T+1}(Y_t + Y_{t-1} + ... + Y_{t-T}). \tag{3}$$

Смысл его заключается в том, что модель видит только ближайшее прошлое (на T отсчетов по времени в глубину) и основываясь только на этих данных строит прогноз.

Метод, описнный в формуле (3) больше всего подходит для решения задачи, поставленной в данной курсовой работе. С его помощью предполагается рассчитывать вероятные расходы пользователя исходя из его последних трат, что, в сочетании с средней ежедневной суммой до ближайшей зарплаты позволяет сообщать пользователю о возможной нехватке денег в до зарплаты.

3.3 Разработка API

API приложения должен позволять пользователю полностью взаимодействовать с приложением и всеми доступными ресурсами. Для разработки API требуется выделить сущности приложения, с которыми будет происходит взаимодействие.

В предметной области данного курсового проекта можно выделить следующие сущности:

– User — сущность, представляющая собой самого пользователя системы, хранящая информацию о нём;

– Currency — необходима для поддержки мультивалютности системы, содержит информацию об интернационалом коде валюты и способе

корректного отображения с числами;

– Wallet — представляет собой счёт пользователя;

– Category — категория расходов или доходов пользователя, служит для подсчёта статистики трат;

– Transaction — сущность, представляющая собой денежную операцию со счетам пользователя, основная сущность системы.

Первоочерёдная задача API — поддерживать основные CRUD-операции для каждой из этих сущностей. Для сущности пользователя необходимо добавить возможности зарегестрироваться и авторизоваться. Также необходимо добавить набор необходимых функций для других сущностей. Описание разработанного API приведено в таблице 3.1.

Таблица 3.1 – Описание API приложения

Идентификатор ресурса	HTTP метод	Описание
/api/users/register	POST	Регистрация нового пользоватля
/api/users/logout	POST	Выход пользователя из приложения
/api/users/change-password	POST	Смена пароля пользователя
/api/users/salary-info	GET	Получение информации о зарплате пользователя
	POST	Смена информации о зарплате пользователя
/oauth/token	POST	Получение токена авторизации по протоколу OAuth 2.0
/api/currencies/	GET	Получение всех валют в постраничном режиме
	POST	Создание новой валюты (недоступно обычным пользователям)
/api/currencies/:id	GET	Получение валюты с заданным ID
	PUT	Изменение валюты с заданным ID (недоступно обычным пользователям)
	DELETE	Удаление валюты с заданным ID

Продолжение таблицы 3.1

Идентификатор ресурса	HTTP метод	Описание
/api/currencies/all	GET	Получение всех валют без постраничного режима
/api/currencies/code/:code	GET	Получение валюты с заданным кодом
/api/wallets/	GET	Получение всех счетов пользователя в постраничном режиме
	POST	Создание нового счета
/api/wallets/:id	GET	Получение счета с заданным ID
	PUT	Изменение счета с заданным ID
	DELETE	Удаление счета с заданным ID
/api/wallets/:id ?newWallet=:newId	DELETE	Удаление счета с заданным ID и переносом всех связанных с ним транзакций на другой счёт с такой же валютой
/api/wallets/all	GET	Получение всех счетов без постраничного режима
/api/wallets/forecast	GET	Одна из основных функций приложения. Получение прогноза на будущие затраты, средней суммы в день до зарплаты и количества дней до зарплаты
/api/categories/	GET	Получение всех категорий пользователя в постраничном режиме
	POST	Создание новой категории
/api/categories/:id	GET	Получение категории с заданным ID
	PUT	Изменение категории с заданным ID
	DELETE	Удаление категории с заданным ID

Продолжение таблицы 3.1

Идентификатор ресурса	HTTP метод	Описание
/api/categories/:id ?newCategory=:newId	DELETE	Удаление категории с заданным ID и переносом всех связанных с ним транзакций на другой другую категорию того же типа
/api/categories/all	GET	Получение всех категорий без постраничного режима
/api/categories/type/:type	GET	Получение всех категорий по заданному типу
/api/categories/stats/:type/ :currencyCode	GET	Получение категорий по данному типу со статистикой по данной валюте за всё время
/api/categories/stats/:type/ :currencyCode ?from=:fromDate &to=:toDate	GET	Получение категорий по данному типу со статистикой по данной валюте за время в промежутке между fromDate и toDate
/api/transactions/	GET	Получение всех транзакций пользователя в постраничном режиме, отсортированные в хронологическом порядке, начиная с самых новых
	POST	Создание новой тразакции
/api/transactions/:id	GET	Получение тразакции с заданным ID
	PUT	Изменение тразакции с заданным ID
	DELETE	Удаление тразакции с заданным ID

4 ОПИСАНИЕ МЕТОДОВ И КЛАССОВ

В данном разделе приведены сведения об основных классах и методах серверной части программного средства.

Таблица 4.1 – Основные классы и методы пакета «service»

Класс	Метод	Описание
AbstractCrudService	checkValidNestedEntities- IfNeed(entity: E)	Метод, предназначенный для валидации вложенных сущностей, где они передаются в качестве параметров метода, по умолчанию пустой, нужен для переопределения в наследниках, где это нужно ServiceNotFoundException.
	checkPersonalVisibility(userId: Long, entityId: Long)	Проверяет соответствие принадлежности сущности с entityId пользователю с userId. Если сущность не принадлежит пользователю, бросает ServiceNotFoundException.
	findByIdAndUserId(id: Long, userId: Long): E?	Находит в базе данных сущность с данными id и userId или возвращает null если сущности с таким сочетанием признаков не существует.

Продолжение таблицы 4.1

Класс	Метод	Описание
	create(entity: E): E?	Создаёт сущность, переданную в параметре (если она валидна), если она успешно создана, возвращает эту сущность, с заполненными полями, которые были сгенерированы в процессе создания (например, id). Если произошла ошибка в процессе изменения, то бросает ServiceException.
	update(entity: E): E?	Изменяет сущность, переданную в параметре (если она валидна), если она успешно изменена, возвращает эту сущность, с обновлёнными полями. Если произошла ошибка в процессе изменения, то бросает ServiceException.
	delete(id: Long, userId: Long): Unit?	Удаляет сущность по переданным признакам, ничего не возвращает, если операция была проведена успешно (Unit), возвращает null, если такой сущности не существует. Если произошла ошибка в процессе удаления, то бросает ServiceException.

Продолжение таблицы 4.1

Класс	Метод	Описание
CategoryServiceImpl	delete(id: Long, userId: Long): Unit?	Удаляет категорию по переданным признакам и удаляет все связанные с ней транзакции, ничего не возвращает, если операция была проведена успешно (Unit), возвращает null, если такой сущности не существует. Если произошла ошибка в процессе удаления, то бросает ServiceException.
	softDelete(id: Long, newId: Long, userId: Long): Unit?	Удаляет категорию по переданным признакам и перемещает все связанные с ней транзакции на категорию с newId. Категория с newId должна быть того же типа, что и удаляемая категория. Ничего не возвращает, если операция была проведена успешно (Unit), возвращает null, если такой сущности не существует. Если произошла ошибка в процессе удаления, то бросает ServiceException.

Продолжение таблицы 4.1

Класс	Метод	Описание
	calculateCategoryStats(userId: Long, type: CategoryType, currencyCode: String, range: DateRange?): CategoriesStats	Возвращает объект статистики, содержщий список всех категорий для данных пользователя и типа, содержащий статистику по заданной валюте. Если параметр range равен null, тогда статистика подсчитвается за всё доступное время.
	findByUserIdAndType(id: Long, type: CategoryType): List<Category>	Возвращает список всех категории указанного типа type, принадлежащих пользователю с указаным id.
CurrencyServiceImpl	findByCode(code: String) : Currency?	Ищет в базе данных валюту с указанным кодом code, если такой валюты не существует — возвращает null.
TransactionServiceImpl	findAllByUserId(userId: Long, pageable: Pageable): Page<Transaction>	Возвращает страницу транзакций, содержащий список транзакций данного пользователя, количество элеметнов списка, общее количество транзакций данного пользователя, в соответствии с параметрами объекта Pageable.

Продолжение таблицы 4.1

Класс	Метод	Описание
WalletServiceImpl	delete(id: Long, userId: Long): Unit?	Удаляет счёт по переданным признакам и удаляет все связанные с ним транзакции, ничего не возвращает, если операция была проведена успешно (Unit), возвращает null, если такой сущности не существует. Если произошла ошибка в процессе удаления, то бросает ServiceException.
	softDelete(id: Long, newId: Long, userId: Long): Unit?	Удаляет счёт по переданным признакам и перемещает все связанные с ним транзакции на счёт с newId. Счёт с newId должен содержать ту же валюту, что и удаляемый счёт. Ничего не возвращает, если операция была проведена успешно (Unit), возвращает null, если такой сущности не существует. Если произошла ошибка в процессе удаления, то бросает ServiceException.

Продолжение таблицы 4.1

Класс	Метод	Описание
	calculateCostForecast(userId: Long) : CostForecast?	Вычисляет прогнозируемые расходы для пользователя на основании информации о его зарплате, также вычисляет ежедневную сумму денег до зарплаты и количество дней до неё.
UserServiceImpl	register(registrationDetails : RegistrationDetails)	Регистрирует нового пользователя в приложении. Бросает ServiceBadRequestException если пользователь с таким переданным email уже существует, пароль и подтверждение пароля не совпадают или если вылюты зарплаты не существует в базе.
	logout(principal : OAuth2Authentication)	Удаляет из базы данных токен доступа OAuth 2.0 для данного пользователя, делая тем самым выход из приложения.
	changePassword(changeDetails: PasswordChangeDetails, userId: Long, principal: OAuth2Authentication)	Заменяет текщий пароль пользователя на новый. Бросает ServiceBadRequestException если новый пароль не совпадает с подтверждением пароля, либо если старый пароль неверен.

Продолжение таблицы 4.1

Класс	Метод	Описание
	findSalaryInfo(userId: Long) : SalaryInfo?	Возвращает информацию о зарплате для текущего пользователя.
	setSalaryInfo(salaryInfo : SalaryInfo, userId: Long) : SalaryInfo?	Обновляет информацию о зарплате для текущего пользователя, бросает ServiceBadRequestException, если переданой валюты не существует в базе.

Таблица 4.2 – Классы и методы блока обработки ошибок

Класс	Метод	Описание
WhiteLabelError-PageController	error(request: HttpServletRequest): Any	Представляет собой глобальный обработчик неопознанных ошибок, включая ошибки неверного URL. Возвращает в зависимости от MIME-типа запроса либо JSON-объект с инфрмацией об ошибке, либо html-страницу с ошибкой.
RestExceptionHandler	handleMethodArgument-NotValid(ex: MethodArgument-NotValidException, headers: HttpHeaders, status: HttpStatus, request: WebRequest): ResponseEntity<Any>	Основной обработчик ошибок валидации при первичной обработке запроса, возвращает на клиент стандартизованный JSON-объект со списком ошибок валидации и HTTP-кодом 400.

Продолжение таблицы 4.2

Класс	Метод	Описание
	handleHttpMessage-NotReadable(ex: HttpMessageNotReadable-Exception, headers: HttpHeaders, status: HttpStatus, request: WebRequest): ResponseEntity<Any>	Обрабатывает ошибки десереализации из JSON в Java-объекты.
	handleTypeMismatch(ex: TypeMismatchException, headers: HttpHeaders, status: HttpStatus, request: WebRequest): ResponseEntity<Any>	Обрабатывает ошибки типов в передаваемых с клиента параметрах запроса.
	handleConstraint-ViolationException(ex: Constraint-ViolationException): ResponseEntity<Any>	Обрабатывает ошибки нарушения целостности базы данных, таких как попытка вставки неуникального значения в уникальное поле и др.
	handleBadRequest(ex: RequestException): ResponseEntity<Any>	Обрабатывает обрабатывает ошибки нарушения бизнес логики приложения, такие как остутствие какой-либо сущности в базе данных или невалидных вложенных сущностях.
	handleInternalError(ex: ControllerException): ResponseEntity<Any>	Основной обработчик внутренних ошибок сервера, возвращает HTTP-код 500 Internal Server Error.

Продолжение таблицы 4.2

Класс	Метод	Описание
	handleInternalError-Explicit(ex: Exception): ResponseEntity<Any>	Обработчик непредвиденных внутренних ошибок сервера, которые могут возникнуть в процессе эксплуатации системы, возвращает HTTP-код 500 Internal Server Error.

ЗАКЛЮЧЕНИЕ

В результате курсового проекта было разработано программное средство для простого и удобного учёта расходов и доходов. Были реализованы функции прогнозирования, вносящие неназойливую и очень полезную информацию о текущем состоянии денежного балланса и тенденций расходов.

Были получены навыки написанная клиент-серверных с использованием REST-архитектуры. Получены навыки разработки клиентских приложений с использованием фреймворка AngularJS.

В дальнейшем планируется развивать проект, внося в него новую функциональность, но с учётом главной особенности приложения — простоты и удобства. Планируется добавить возможности контролировать долги и денежные переводы между счетами. Также одной из очень важных целей является разработка приложений на мобильные платформы, использующих API сервера.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Потребительские расходы домашних хозяйств Беларуси [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.belstat.gov.by;

[2] Koltin Documentation [Электронный ресурс]. – Электронные данные. – Режим доступа: https://kotlinlang.org/docs/reference/ ;

[3] Аналитические технологии для прогнозирования и анализа данных [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.neuroproject.ru/forecasting_tutorial.php;

[4] Анализ временных рядов и прогнозирование /Н. А. Садовникова, Р. А. Шмойлова — М.: Московский финансово-промышленный университет «Синергия», 2016 — 152 с.

[5] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — Питер, 2007 — 366 с.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

```
abstract class AbstractPersonalCrudController<E : UserBaseEntity<Long>>(
      private val crudService: PersonalCrudService<E, Long>,
      private val logger: Logger
) : CrudController<E, Long> {


   override fun getAll(@Auth auth: UserDetails, pageable: Pageable): Page<E> =
         wrapServiceCall(logger) { crudService.findAllByUserId(auth.userId,
            pageable) }

   override fun getById(@PathVariable("id") id: Long?, @Auth auth: UserDetails
      ): E = wrapServiceCall(logger) {
      crudService.findByIdAndUserId(id!!, auth.userId).ifNullNotFound(id)
   }

   override fun create(@Valid @RequestBody entity: E, @Auth auth: UserDetails)
      : E = wrapServiceCall(logger) {
      entity.id = null
      entity.userId = auth.userId
      crudService.create(entity) ?: throw badRequestException("Error", "Cannot
          create entity")
   }

   override fun update(@PathVariable("id") id: Long?, @Valid @RequestBody
      entity: E, @Auth auth: UserDetails): E {
      entity.id = id
      entity.userId = auth.userId
      return wrapServiceCall(logger) {
         crudService.update(entity) ?:
               throw badRequestException("Error", "Cannot create entity")
      }
   }

   override fun delete(@PathVariable("id") id: Long?, @Auth auth: UserDetails)
       =
         wrapServiceCall(logger) { crudService.delete(id!!, auth.userId) }
               ?: notFoundException("Error", "Entity not found")

}

@Validated
open class CategoryControllerImpl(
      private val categoryService: CategoryService
) : AbstractPersonalCrudController<Category>(categoryService, logger),
    CategoryController {
   companion object {

      private val logger = LoggerFactory.getLogger(CategoryControllerImpl::
         class.java)!!
```

```
    }

    @InitBinder
    fun initBinder(binder: WebDataBinder) {
        binder.registerCustomEditor(CategoryType::class.java, CategoryTypeBinder
            ())
    }

    override fun calculateCategoryStats(
            @PathVariable("type") type: CategoryType,
            @PathVariable("currencyCode") @CurrencyCode currencyCode: String,
            @RequestParam(name = "from") @DateTimeFormat(pattern = "yyyy-MM-dd")
                from: Date,
            @RequestParam(name = "to") @DateTimeFormat(pattern = "yyyy-MM-dd") to
                : Date,
            @Auth auth: UserDetails
    ): CategoriesStats =
            wrapServiceCall(logger) { categoryService.calculateCategoryStats(auth
                .userId, type, currencyCode, from till to) }

    override fun calculateCategoryStatsAllTime(
            @PathVariable("type") type: CategoryType,
            @PathVariable("currencyCode") @CurrencyCode currencyCode: String,
            @Auth auth: UserDetails
    ): CategoriesStats =
            wrapServiceCall(logger) { categoryService.calculateCategoryStats(auth
                .userId, type, currencyCode, null) }

    override fun getAll(@Auth auth: UserDetails): List<Category> =
            wrapServiceCall(logger) { categoryService.findAllByUserId(auth.userId
                ) }

    override fun getAll(@Auth auth: UserDetails, @PathVariable("type") type:
        CategoryType): List<Category> =
            wrapServiceCall(logger) { categoryService.findByUserIdAndType(auth.
                userId, type) }

    override fun softDelete(@PathVariable("id") id: Long?, @RequestParam("
        newCategory") newId: Long?, @Auth auth: UserDetails) =
            wrapServiceCall(logger) { categoryService.softDelete(id!!, newId!!,
                auth.userId) }
                    ?: notFoundException("Error", "Entity not found")
}

open class CurrencyControllerImpl(
        private val currencyService: CurrencyService
) : CurrencyController {

    companion object {
        private val logger = LoggerFactory.getLogger(CurrencyControllerImpl::
            class.java)!!
    }
```

```
    override fun getAll(): Iterable<Currency> =
          wrapServiceCall(logger) { currencyService.findAll() }

    override fun getAll(@Auth auth: UserDetails, pageable: Pageable): Page<
        Currency> =
          wrapServiceCall(logger) { currencyService.findAll(pageable) }

    override fun getById(@PathVariable("id") id: Long?, @Auth auth: UserDetails
        ): Currency =
          wrapServiceCall(logger) { currencyService.findById(id!!).
              ifNullNotFound(id) }

    override fun getByCode(@PathVariable("code") @CurrencyCode code: String):
        Currency =
          wrapServiceCall(logger) {
              currencyService.findByCode(code)
                    ?: notFoundException("Error", "Entity with code '$code' not
                         found")
          }

    override fun create(@Valid @RequestBody entity: Currency, @Auth auth:
        UserDetails): Currency =
          wrapServiceCall(logger) { currencyService.create(entity) }

    override fun update(@PathVariable("id") id: Long?, @Valid @RequestBody
        entity: Currency, @Auth auth: UserDetails): Currency =
          wrapServiceCall(logger) { currencyService.update(id!!, entity) }

    override fun delete(@PathVariable("id") id: Long?, @Auth auth: UserDetails)
         =
          wrapServiceCall(logger) { currencyService.delete(id!!) }
}

open class TransactionControllerImpl(
      transactionService: TransactionService
) : AbstractPersonalCrudController<Transaction>(transactionService, logger),
    TransactionController {

    companion object {
        private val logger = LoggerFactory.getLogger(TransactionControllerImpl::
            class.java)!!
    }

}

class UserControllerImpl(
      private val userService: UserService
) : UserController {

    companion object {
        val logger = LoggerFactory.getLogger(UserControllerImpl::class.java)!!
    }
```

```
    override fun register(@Valid @RequestBody registrationDetails:
        RegistrationDetails) =
            wrapServiceCall(logger) { userService.register(registrationDetails) }

    override fun getSalaryInfo(@Auth auth: UserDetails): SalaryInfo =
            wrapServiceCall(logger) { userService.findSalaryInfo(auth.userId) }
                ?: SalaryInfo(null, null)

    override fun setSalaryInfo(@Valid @RequestBody salaryInfo: SalaryInfo,
        @Auth auth: UserDetails) =
            wrapServiceCall(logger) { userService.setSalaryInfo(salaryInfo, auth.
                userId) }

    override fun logout(principal: OAuth2Authentication) {
        wrapServiceCall(logger) { userService.logout(principal) }
    }

    override fun changePassword(@Valid @RequestBody passwordChangeDetails:
        PasswordChangeDetails, @Auth auth: UserDetails, principal:
        OAuth2Authentication) =
            wrapServiceCall(logger) { userService.changePassword(
                passwordChangeDetails, auth.userId, principal) }
}

open class WalletControllerImpl(
        private val walletService: WalletService
) : AbstractPersonalCrudController<Wallet>(walletService, logger),
    WalletController {

    @InitBinder
    fun initBinder(binder: WebDataBinder) {
        binder.registerCustomEditor(WalletType::class.java, WalletTypeBinder())
    }

    override fun getCostForecast(@Auth auth: UserDetails): CostForecast? =
            wrapServiceCall(logger) { walletService.calculateCostForecast(auth.
                userId) }

    companion object {
        val logger = LoggerFactory.getLogger(WalletControllerImpl::class.java)!!
    }

    override fun getAll(@Auth auth: UserDetails): List<Wallet> =
            wrapServiceCall(logger) { walletService.findAllByUserId(auth.userId)
                }

    override fun softDelete(@PathVariable("id") id: Long?, @RequestParam("
        newWallet") newId: Long?, @Auth auth: UserDetails) =
            wrapServiceCall(logger) { walletService.softDelete(id!!, newId!!,
                auth.userId) }
                    ?: notFoundException("Error", "Entity not found")

}
```

```
@Service
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
class UserServiceImpl(
      private val userRepository: UserRepository,
      private val currencyRepository: CurrencyRepository,
      private val passwordEncoder: PasswordEncoder,
      private val defaultTokenServices: DefaultTokenServices
) : UserService {
    companion object {

       val logger = LoggerFactory.getLogger(UserServiceImpl::class.java)!!

    }

    override fun changePassword(changeDetails: PasswordChangeDetails, userId:
        Long, principal: OAuth2Authentication) {
       if (changeDetails.password != changeDetails.passwordConfirmation) {
          throw ServiceBadRequestException("Error" to "Passwords don't match")
       }
       val user = wrapJPACall { userRepository.findOne(userId) }
       if (user != null) {
          if (passwordEncoder.matches(changeDetails.oldPassword, user.
             passwordHash)) {
             user.passwordHash = passwordEncoder.encode(changeDetails.password)
             wrapJPAModifyingCall { userRepository.save(user) }
                   ?: throw ServiceBadRequestException("Error" to "Cannot
                      update password")
             logout(principal)
             logger.debug("Password changed for user[${user.id}][${user.email}]
                ")
          } else {
             throw ServiceBadRequestException("Error" to "Wrong old password")
          }
       } else {
          logger.warn("Invalid user id '$userId' passed!")
          throw ServiceBadRequestException("Error" to "User doesn't exist.")
       }
    }

    override fun logout(principal: OAuth2Authentication) {
       val accessToken = defaultTokenServices.getAccessToken(principal)
       defaultTokenServices.revokeToken(accessToken.value)
    }

    override fun register(registrationDetails: RegistrationDetails) {
       val existingUser = wrapJPACall { userRepository.findByEmail(
          registrationDetails.email ?: "") }
       val errors = mutableListOf<RestErrorMessage>()
       if (registrationDetails.password != registrationDetails.
          passwordConfirmation) {
```

```
            errors.add("Error", "Passwords don't match")
        }
        if (existingUser != null) {
            logger.debug("User with email '${registrationDetails.email} already
                exists, registration failed.")
            errors.add("Error", "User with such email is already exists.")
        }
        val currency = wrapJPACall { currencyRepository.findByCode(
            registrationDetails.salaryCurrencyCode ?: "") }
        if (currency == null) {
            logger.debug("Failed setting salary info: invalid currency code '${
                registrationDetails.salaryCurrencyCode}'")
            errors.add("Error", "Currency with code '${registrationDetails.
                salaryCurrencyCode}' doesn't exist.")
        }
        if (errors.isEmpty()) {
            val user = User(
                  email = registrationDetails.email,
                  passwordHash = passwordEncoder.encode(registrationDetails.
                      password)
            )
            wrapJPAModifyingCall { userRepository.save(user) }
                  ?: throw ServiceBadRequestException("Error" to "Cannot register
                       user")
            logger.info("User with email '${registrationDetails.email}
                successfully registered.")
        }
        errors.throwIfNeed()
    }

    override fun findSalaryInfo(userId: Long): SalaryInfo? {
        val user = wrapJPACall { userRepository.findOne(userId) }
        if (user != null) {
            return if (user.salaryDay == null || user.salaryCurrency == null)
                null
            else
                SalaryInfo(user.salaryDay, user.salaryCurrency?.code)
        } else {
            logger.warn("Invalid user id '$userId' passed!")
            throw ServiceBadRequestException("Error" to "User doesn't exist.")
        }
    }

    override fun setSalaryInfo(salaryInfo: SalaryInfo, userId: Long) {
        val currency = wrapJPACall { currencyRepository.findByCode(salaryInfo.
            salaryCurrencyCode ?: "") }
        if (currency != null) {
            val user = wrapJPACall { userRepository.findOne(userId) }
            if (user != null) {
                user.salaryDay = salaryInfo.salaryDay
                user.salaryCurrency = currency
                wrapJPAModifyingCall { userRepository.save(user) }
                      ?: throw ServiceBadRequestException("Error" to "Cannot set
```

```
                    salary info")
            } else {
                logger.warn("Invalid user id '$userId' passed!")
                throw ServiceBadRequestException("Error" to "User doesn't exist.")
            }
        } else {
            logger.debug("Failed setting salary info: invalid currency code '${
                salaryInfo.salaryCurrencyCode}'")
            throw ServiceBadRequestException("Error" to
                  "Currency with code '${salaryInfo.salaryCurrencyCode}' doesn't
                      exist.")
        }
    }
}

@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
abstract class AbstractCrudService<E : UserBaseEntity<Long>>(
      private val crudRepository: PersonalCrudRepository<E, Long>
) : PersonalCrudService<E, Long> {

    @Transactional(propagation = Propagation.SUPPORTS)
    protected open fun checkValidNestedEntitiesIfNeed(entity: E) {}

    @Transactional(propagation = Propagation.SUPPORTS)
    protected fun checkPersonalVisibility(userId: Long, entityId: Long) =
          findByIdAndUserId(entityId, userId).ifNullServiceNotFound(entityId)

    override fun findByIdAndUserId(id: Long, userId: Long): E? =
          wrapJPACall { crudRepository.findByIdAndUserId(id, userId) }

    override fun create(entity: E): E? = wrapJPACall {
        checkValidNestedEntitiesIfNeed(entity)
        wrapJPAModifyingCall { crudRepository.save(entity) }
    }

    override fun update(entity: E): E? = wrapJPACall {
        checkPersonalVisibility(entity.userId!!, entity.id!!)
        checkValidNestedEntitiesIfNeed(entity)
        wrapJPAModifyingCall { crudRepository.save(entity) }
    }

    override fun delete(id: Long, userId: Long): Unit? = wrapJPACall {
        checkPersonalVisibility(userId, id)
        wrapJPAModifyingCall { crudRepository.delete(id) }
    }

}

@Service
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
class WalletServiceImpl(
```

```
    private val walletRepository: WalletRepository,
    private val transactionRepository: TransactionRepository,
    private val userService: UserService
) : AbstractCrudService<Wallet>(walletRepository), WalletService {

    companion object {
        private val logger = LoggerFactory.getLogger(WalletServiceImpl::class.
            java)!!
    }

    override fun checkValidNestedEntitiesIfNeed(entity: Wallet) {
        val errors = mutableListOf<RestErrorMessage>()
        if (entity.currency?.id == null) {
            errors.add("Field 'currency'", "Nested field 'id' is not specified")
        }
        errors.throwIfNeed()
    }

    override fun findByIdAndUserId(id: Long, userId: Long): Wallet? =
            wrapJPACall { walletRepository.findByIdAndUserId(id, userId) }

    override fun findAllByUserId(userId: Long): List<Wallet> =
            wrapJPACall { walletRepository.findByUserId(userId) }

    override fun findAllByUserId(userId: Long, pageable: Pageable): Page<Wallet
        > =
            wrapJPACall { walletRepository.findByUserId(userId, pageable) }

    override fun delete(id: Long, userId: Long): Unit? {
        checkPersonalVisibility(userId, id)
        if (id == userId) throw ServiceBadRequestException("Error" to "Cannot
            shift transactions to delete wallet")
        val affected = wrapJPACall { transactionRepository.deleteByWalletId(id)
            }
        logger.info("$affected transactions deleted from wallet[$id]")
        val isSuccess: Unit? = wrapJPAModifyingCall { walletRepository.delete(id
            ) }
        if (isSuccess != null) logger.info("Wallet[$id] deleted")
        return isSuccess
    }

    override fun softDelete(id: Long, newId: Long, userId: Long): Unit? {
        checkPersonalVisibility(userId, id)
        if (id == newId) throw ServiceBadRequestException("Error" to "Cannot
            shift transactions to delete wallet")
        val newWallet = wrapJPACall { walletRepository.findByIdAndUserId(newId,
            userId) }
        val oldWallet = wrapJPACall { walletRepository.findByIdAndUserId(id,
            userId) }
        newWallet ?: throw ServiceNotFoundException("New wallet" to "Wallet not
            found.")
        oldWallet ?: throw ServiceNotFoundException("Old wallet" to "Wallet not
            found.")
```

```
        if (oldWallet.currency?.id != newWallet.currency?.id) {
            throw ServiceBadRequestException("Invalid currency" to "Wallets must
                have same currencies")
        }
        val affected = wrapJPACall { transactionRepository.shiftToNewWallet(
            newId, id) }
        logger.info("$affected transactions shifted from wallet[$id] to wallet[
            $newId, ${newWallet.name}]")
        val isSuccess: Unit? = wrapJPAModifyingCall { walletRepository.delete(id
            ) }
        if (isSuccess != null) logger.info("Wallet[$id] deleted")
        return isSuccess
    }

    override fun calculateCostForecast(userId: Long): CostForecast? {
        val errors = mutableListOf<RestErrorMessage>()
        val salaryInfo = userService.findSalaryInfo(userId)
        salaryInfo ?: return null
        val average = walletRepository.calculateSumForNormal(salaryInfo.
            salaryCurrencyCode!!)
        average ?: errors.add("Error", "Cannot calculate daily sum")
        val prediction = transactionRepository.calculateMovingAveragePrediction(
            userId,
                salaryInfo.salaryCurrencyCode, PREDICTION_LOOKUP_DAYS)
        prediction ?: errors.add("Error", "Cannot calculate average prediction")
        errors.throwIfNeed()

        val calendar = Calendar.getInstance()
        val dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH)
        val maxDayOfMonth = calendar.getActualMaximum(Calendar.DAY_OF_MONTH)
        val salaryDay = if (salaryInfo.salaryDay!! > maxDayOfMonth)
            maxDayOfMonth else salaryInfo.salaryDay
        val daysTillSalary = if (dayOfMonth >= salaryDay)
           maxDayOfMonth - dayOfMonth + salaryDay else salaryDay - dayOfMonth
        return CostForecast(
              dailySumTillSalary = (average!!.divide(BigDecimal(daysTillSalary),
                  RoundingMode.FLOOR)).setScale(4, RoundingMode.DOWN),
              actualCosts = prediction!!,
              daysTillSalary = daysTillSalary
        )
    }

}

@Service
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
open class TransactionServiceImpl(
      private val transactionRepository: TransactionRepository
) : AbstractCrudService<Transaction>(transactionRepository),
    TransactionService {

    @Transactional(propagation = Propagation.SUPPORTS)
```

```
    override fun checkValidNestedEntitiesIfNeed(entity: Transaction) {
        val errors = mutableListOf<RestErrorMessage>()
        if (entity.wallet?.id == null) {
            errors.add("Field 'wallet'", "Nested field 'id' is not specified")
        }
        if (entity.category?.id == null) {
            errors.add("Field 'category'", "Nested field 'id' is not specified")
        }
        errors.throwIfNeed()
    }

    override fun findByIdAndUserId(id: Long, userId: Long): Transaction?
            = wrapJPACall { transactionRepository.findByIdAndUserId(id, userId) }

    override fun findAllByUserId(userId: Long, pageable: Pageable): Page<
        Transaction>
            = wrapJPACall { transactionRepository.findByUserId(userId, pageable)
                }

}

@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
abstract class AbstractCrudService<E : UserBaseEntity<Long>>(
        private val crudRepository: PersonalCrudRepository<E, Long>
) : PersonalCrudService<E, Long> {

    @Transactional(propagation = Propagation.SUPPORTS)
    protected open fun checkValidNestedEntitiesIfNeed(entity: E) {}

    @Transactional(propagation = Propagation.SUPPORTS)
    protected fun checkPersonalVisibility(userId: Long, entityId: Long) =
            findByIdAndUserId(entityId, userId).ifNullServiceNotFound(entityId)

    override fun findByIdAndUserId(id: Long, userId: Long): E? =
            wrapJPACall { crudRepository.findByIdAndUserId(id, userId) }

    override fun create(entity: E): E? = wrapJPACall {
        checkValidNestedEntitiesIfNeed(entity)
        wrapJPAModifyingCall { crudRepository.save(entity) }
    }

    override fun update(entity: E): E? = wrapJPACall {
        checkPersonalVisibility(entity.userId!!, entity.id!!)
        checkValidNestedEntitiesIfNeed(entity)
        wrapJPAModifyingCall { crudRepository.save(entity) }
    }

    override fun delete(id: Long, userId: Long): Unit? = wrapJPACall {
        checkPersonalVisibility(userId, id)
        wrapJPAModifyingCall { crudRepository.delete(id) }
    }
```

```
}

@Service
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = arrayOf(
    ServiceException::class))
class CurrencyServiceImpl(
      private val currencyRepository: CurrencyRepository
) : CurrencyService {

   override fun create(currency: Currency): Currency = wrapJPACall {
      currency.id = null
      currencyRepository.save(currency)
   }

   override fun findAll(): Iterable<Currency> =
         wrapJPACall { currencyRepository.findAll() }

   override fun findById(id: Long): Currency? =
         wrapJPACall { currencyRepository.findOne(id) }

   override fun findAll(pageable: Pageable): Page<Currency> =
         wrapJPACall { currencyRepository.findAll(pageable) }

   override fun findByCode(code: String): Currency? =
         wrapJPACall { currencyRepository.findByCode(code) }

   override fun update(id: Long, currency: Currency): Currency = wrapJPACall {
      currency.id = id
      currencyRepository.save(currency)
   }

   override fun delete(id: Long) = wrapJPACall { currencyRepository.delete(id)
       }
}

@Repository
interface UserRepository : PagingAndSortingRepository<User, Long> {
   fun findByEmail(email: String): User?
}

interface PersonalCrudRepository<E : BaseEntity<ID>, ID : Serializable> {
   fun findByIdAndUserId(id: ID, userId: ID): E?
   fun <S : E> save(entity: S): S
   fun delete(id: ID)
}

@Repository
interface TransactionRepository :
      PagingAndSortingRepository<Transaction, Long>,
      PersonalCrudRepository<Transaction, Long> {

   @Query("""SELECT t FROM Transaction t
                  INNER JOIN FETCH t.wallet w
```

```
            INNER JOIN FETCH w.currency
            INNER JOIN FETCH t.category
            WHERE  (t.id = :id) AND (t.userId = :userId)""")
override fun findByIdAndUserId(@Param("id") id: Long, @Param("userId")
    userId: Long): Transaction?

@Query("""SELECT t FROM Transaction t
            INNER JOIN FETCH t.wallet w
            INNER JOIN FETCH w.currency
            INNER JOIN FETCH t.category
            WHERE t.userId = :id
            ORDER BY t.date DESC, t.id ASC""",
      countQuery = "SELECT COUNT(t) FROM Transaction t WHERE t.userId = :
          id")
fun findByUserId(@Param("id") id: Long, pageable: Pageable): Page<
    Transaction>

@Transactional
@Modifying(clearAutomatically = true)
@Query("DELETE FROM Transaction t WHERE t.category.id = :id")
fun deleteByCategoryId(@Param("id") categoryId: Long): Int

@Transactional
@Modifying(clearAutomatically = true)
@Query("DELETE FROM Transaction t WHERE t.wallet.id = :id")
fun deleteByWalletId(@Param("id") id: Long): Int

@Transactional
@Modifying(clearAutomatically = true)
@Query("UPDATE transaction SET category_id = :newCategoryId WHERE
    category_id = :oldCategoryId",
      nativeQuery = true)
fun shiftToNewCategory(
      @Param("newCategoryId") newCategoryId: Long,
      @Param("oldCategoryId") oldCategoryId: Long): Int

@Transactional
@Modifying(clearAutomatically = true)
@Query("UPDATE transaction SET wallet_id = :newWalletId WHERE wallet_id =
    :oldWalletId",
      nativeQuery = true)
fun shiftToNewWallet(
      @Param("newWalletId") newWalletId: Long,
      @Param("oldWalletId") oldWalletId: Long): Int

@Query("""SELECT IF(SUM(act.sum) IS NULL, 0, SUM(act.sum)) / (:daysLookup
    + 1)
          FROM transaction AS act
            JOIN wallet ON act.wallet_id = wallet.id
            JOIN currency ON wallet.currency_id = currency.id
          WHERE (act.user_id = :userId) AND
                (currency.code = :currencyCode) AND
                (act.date BETWEEN DATE_SUB(NOW(), INTERVAL :daysLookup DAY)
```

```
    AND NOW())""",
        nativeQuery = true)
    fun calculateMovingAveragePrediction(
        @Param("userId") userId: Long,
        @Param("currencyCode") currencyCode: String,
        @Param("daysLookup") daysLookup: Int): BigDecimal?

}

@Repository
interface WalletRepository :
      PagingAndSortingRepository<Wallet, Long>,
      PersonalCrudRepository<Wallet, Long> {

   @Query("SELECT w FROM Wallet w INNER JOIN FETCH w.currency WHERE (w.id = :
      id) AND (w.userId = :userId)")
   override fun findByIdAndUserId(@Param("id") id: Long, @Param("userId")
      userId: Long): Wallet?

   @Query("SELECT w FROM Wallet w INNER JOIN FETCH w.currency WHERE w.userId
      = :id ORDER BY w.type ASC, w.id ASC",
         countQuery = "SELECT COUNT(w) FROM Wallet w WHERE w.userId = :id")
   fun findByUserId(@Param("id") id: Long, pageable: Pageable): Page<Wallet>

   @Query("SELECT w FROM Wallet w INNER JOIN FETCH w.currency WHERE w.userId
      = :id ORDER BY w.type ASC, w.id ASC")
   fun findByUserId(@Param("id") id: Long): List<Wallet>

   @Query("""SELECT SUM(w.balance) FROM Wallet w
                INNER JOIN w.currency c
                WHERE w.type = 'NORMAL'
                AND c.code = :currencyCode""")
   fun calculateSumForNormal(@Param("currencyCode") currencyCode: String):
      BigDecimal?
}


@Repository
interface CurrencyRepository : PagingAndSortingRepository<Currency, Long> {
   fun findByCode(code: String): Currency?
}


@Repository
interface CategoryRepository :
      PagingAndSortingRepository<Category, Long>,
      PersonalCrudRepository<Category, Long> {

   fun findByUserIdOrderByIdAsc(id: Long, pageable: Pageable): Page<Category>

   fun findByUserIdOrderByIdAsc(id: Long): List<Category>

   fun findByUserIdAndType(id: Long, type: CategoryType): List<Category>
```

```
    fun fetchCategoryStats(
            @Param("userId") userId: Long,
            @Param("currencyId") currencyId: Long,
            @Param("categoryType") categoryType: String,
            @Param("startDate") startDate: Timestamp,
            @Param("endDate") endDate: Timestamp
    ): List<CategoryStats>
}

data class DateRange(
        val start: Date,
        val end: Date
) {
    companion object {
        val NONE = DateRange(MIN_SQL_DATETIME, MAX_SQL_DATETIME)
    }
}

data class RestError(
        val status: Int,
        val error: String,
        val errors: List<RestErrorMessage>
) {
    constructor(status: HttpStatus, vararg errors: RestErrorMessage) : this(
            status.value(),
            status.reasonPhrase,
            errors.asList())

    constructor(status: HttpStatus, errors: List<RestErrorMessage>) : this(
            status.value(),
            status.reasonPhrase,
            errors)

}

data class RestErrorMessage(
        val title: String,
        val message: String
)

data class OauthError(
        val error: String,
        val errorDescription: String
)

inline fun <T> wrapJPACall(block: () -> T): T = try {
    block()
} catch (e: DataIntegrityViolationException) {
    throw ServiceConstraintFailException("Invalid entity" to "Invalid entity
        references")
} catch (e: DataAccessException) {
    throw ServiceException(cause = e)
```

```
}

inline fun <T> wrapJPAModifyingCall(block: () -> T): T? = try {
   block()
} catch (e: EmptyResultDataAccessException) {
   null
} catch (e: DataIntegrityViolationException) {
   throw ServiceConstraintFailException("Invalid entity" to "Invalid entity
      references")
} catch (e: DataAccessException) {
   throw ServiceException(cause = e)
}


inline fun <T> wrapServiceCall(logger: Logger, block: () -> T): T = try {
   block()
} catch (e: ServiceRequestException) {
   logger.debug("Invalid user request")
   throw RequestException(e.status, e.errors)
} catch (e: ServiceException) {
   logger.warn("Error while service call\n", e)
   throw ControllerException(cause = e)
}

fun notFoundException(title: String, message: String): Nothing =
      throw RequestException(HttpStatus.NOT_FOUND, title to message)

fun badRequestException(title: String, message: String): Nothing =
      throw RequestException(HttpStatus.BAD_REQUEST, title to message)

fun MutableList<RestErrorMessage>.throwIfNeed(): Nothing? =
      if (this.isNotEmpty()) throw ServiceBadRequestException(this) else null

fun MutableList<RestErrorMessage>.add(title: String, message: String) {
   this.add(RestErrorMessage(title, message))
}

inline fun <T> T?.ifNull(block: T.() -> T): T {
   return if (this == null) block() else this
}

fun <ID, T : BaseEntity<ID>> T?.ifNullServiceNotFound(id: ID? = null): T {
   if (this == null) {
      throw ServiceNotFoundException("Error" to "Entity ${if (id == null) ""
         else "with id '$id' "}not found")
   }
   return this
}

fun <ID, T : BaseEntity<ID>> T?.ifNullNotFound(id: ID? = null): T {
   if (this == null) {
      throw RequestException(HttpStatus.NOT_FOUND,
            "Error" to "Entity ${if (id == null) "" else "with id '$id' "}not
```

```
                found")
    }
    return this
}

infix fun Date.till(end: Date) = DateRange(this, end)

fun Date.toStamp(): Timestamp = Timestamp.from(this.toInstant())
```