Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет *комп’ютерних наук*

(повна назва)

Кафедра *програмної інженерії*

(повна назва)

**АТЕСТАЦІЙНА РОБОТА (ПРОЕКТ)**

**Пояснювальна записка**

*БАКАЛАВР*

(освітньо-атестаційний рівень)

\_\_\_\_*IoT-застосування «SMART MARKET» автоматизоції здійснення процесу                                      покупок у торгових закладах*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(тема)

Виконав:

студент *4* курсу, групи *ПІ-13-6*

напряму підготовки (спеціальності)

*6.050103 Програмна інженерія*

(шифр і назва напряму, спеціальності)

*Кокорін П.О.*

(прізвище, ініціали)

Керівник *Ворочек О.Г.*

(прізвище, ініціали)

Рецензент *Лановий О.Ф.*

(прізвище, ініціали)

Допускається до захисту

Зав. кафедри \_\_\_\_\_\_\_\_\_\_\_ *Дудар З.В.*

(підпис) (прізвище, ініціали)

2017 р.

Харківський національний університет радіоелектроніки

Факультет *комп’ютерних наук*

Кафедра *програмної інженерії*

Освітньо-атестаційний рівень *бакалавр*

Напрям підготовки *6.050103 Програмна інженерія*

(шифр і назва)

Спеціальність *6.050103 Програмна інженерія*

(шифр і назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри

(підпис)

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2017 р.

**ЗАВДАННЯ**

НА АТЕСТАЦІЙНУ РОБОТУ (ПРОЕКТ)

студентові *Кокоріну Павлу Олександровичу*

(прізвище, ім’я, по батькові)

1. Тема роботи (проекту) *IoT-застосування «SMART MARKET» автоматизації здійснення процесу покупок у торгових закладах* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

затверджена наказом по університету від "*15*" *травня 2017 р*. № 665 Ст.

2. Термін подання студентом роботи (проекту) *01.06.2017*

3. Вихідні дані до роботи (проекту) *розробити програмну реалізацію IoT-застосування, яке складається з веб-додатку та мобільного додатку. Використовувати ОС Windows 10, середовище розробки Visual Studio 2017, СУБД MS SQL Server 2014, мови об’єктно-орієнтованого програмування C# та JavaScript, фреймворки .NET Framework, ASP.NET MVC 5, та технологію XAMARIN.*

4. Зміст пояснювальної записки (перелік питань, що потрібно розробити)

*мета роботи, аналіз предметної області та концептуальне моделювання, опис прийнятих проектних рішень (об‘єктні моделі та методи, структура бази даних), опис програмної реалізації, тестування. Додатки: а) слайди презентації; б) лістинг програми; в) електронні матеріали до проекту на CD.*

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників, плакатів) *Діаграма прецедентів, діаграма класів, слой DAL (Data access layer), слой BLL (Business logic layer), слой UI (User interface), картка коду, результат роботи API api/balances/1, інтерфейс «UserManager», модель аутентіфікації через форми, модель аутентіфікації через соціальні мережі, патерн MVC, головний інтерфейс сайту, реалізація Generic Repository (частина 3, IDisposable), реалізація Generic Repository (частина 2), реалізація Generic Repository (частина 1), контекст бази даних, табличка Balance, табличка User, моделі User та Balance. панель користувача*

6. Консультанти розділів роботи (проекту)

|  |  |  |  |
| --- | --- | --- | --- |
| Найменування  розділу | Консультант  (посада, прізвище, ім’я, по батькові) | Позначка консультанта  про виконання розділу | |
| підпис | дата |
| Спецчастина | доц. Ворочек О.Г. |  |  |
|  |  |  |  |
|  |  |  |  |

7. Дата видачі завдання «18» квітня 2017 р.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів роботи (проекту) | Термін  виконання етапів проекту (роботи) | Позначка про виконання |
| 1 | Аналіз предметної галузі | *01.03.2017* |  |
| 2 | Розробка специфікації ПЗ | *21.03.2017* |  |
| 3 | Об'єктний аналіз поставленої задачі | *18.04.2017* |  |
| 4 | Кодування програми | *18.05.2017* |  |
| 5 | Тестування і налагодження програми | *30.05.2017* |  |
| 6 | Підготовка пояснювальної записки: | *04.06.2017* |  |
| 7 | Підготовка презентації та доповіді | *05.06.2017* |  |
| 8 | Нормоконтроль, рецензування | *06.06.2017* |  |
| 9 | Попередній захист | *07.06.2017* |  |
| 10 | Занесення диплома в електронний архів | *08.06.2017* |  |
| 11 | Допуск до захисту у зав. кафедри | *09.06.2017* |  |
|  |  |  |  |

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник роботи (проекту) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *доц.,* *Ворочек О.Г*

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи бакалавра: 63 с., 20 рис., 1 табл.,15 джерел.

Об'єкт проектування – система організації покупок у розумному магазині за допомогою мобільного додатку.

Метою роботи є створення веб-системи та мобільного додатку організації покупок у розумному магазині.

Методи розробки базуються на технології .NET MVC 5, сервер бази даних MSSQL, XAMARIN, EF6, Git, Web-сервер IIS.

У результаті роботи здійснена реалізація системи з організації покупок у розумному магазині, яка дозволяє користувачам створювати персональний акаунт на сайті, поповнювати кошти цього акаунту та оплачувати товари у розумних магазинах, де інтегрована ця система.

ВЕБ-СИСТЕМА, MSSQL, РОЗУМНИЙ МАГАЗИН, ТОВАРИ, МОБІЛЬНИЙ ДОДАТОК, MSSQL, MVC 5, IIS, GIT, XAMARIN.

Object of planning - system of intelligent purchases in the store with a mobile application.

The point is to create a web-based application and mobile organization purchases a reasonable store.

Methods of development based on technology .NET MVC 5, MSSQL database server and the Web-server IIS.

As a result of implementation of the system made the organization a reasonable shopping store that allows users to create a personal account on the site, to replenish this account to pay for goods and reasonable shops where the system is integrated.

WEB SYSTEM, MSSQL, SMART SHOP, PRODUCTS, MOBILE APPLICATION, MSSQL, MVC 5, IIS, GIT, XAMARIN.

ЗВІТ

[ВСТУП 6](#_Toc484646219)

[1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ 8](#_Toc484646220)

[1.2 Виявлення проблем та актуалізація рішень 10](#_Toc484646221)

[1.3 Аналіз існуючих рішень 11](#_Toc484646222)

[1.4 Аналіз основних технологій розробки 14](#_Toc484646223)

[1.5 Постановка задачі 17](#_Toc484646224)

[2 ФОРМУВАННЯ ВИМОГ ПО ПРОГРАМНОЇ СИСТЕМИ 19](#_Toc484646225)

[3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 21](#_Toc484646226)

[3.1 UML проектування ПЗ 21](#_Toc484646227)

[3.2 Проектування архітектури ПЗ 25](#_Toc484646228)

[3.3 Проектування бази даних 28](#_Toc484646229)

[3.4 Приклади найцікавіших алгоритмів та методів 30](#_Toc484646230)

[3.5 Створення UI / UX або іншого дизайну системи 33](#_Toc484646231)

[4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ 36](#_Toc484646232)

[5 ТЕСТУВАННЯ РОЗРОБЛЕННОГО ПРОГРАМНОГО ЗАБЕСПЕЧЕННЯ 43](#_Toc484646233)

[ВИСНОВКИ 46](#_Toc484646234)

[ПЕРЕЛІК ПОСИЛАНЬ 47](#_Toc484646235)

[ДОДАТОК А. Слайди презентації 48](#_Toc484646236)

[ДОДАТОК Б. Лістинг коду класу «MapService» 55](#_Toc484646237)

# ВСТУП

У сучасному світі усе навколо стрімко автоматизується. Автоматизування ручної праці це дуже важливий процес, який підвищує ефективність роботи, зменшує затрати, відкриває нові можливості и т.д. Тому не так давно з’явилися інтернет-магазини, в яких люди мають можливість робити покупки різноманітних товарів. Але ця інновація не витіснила звичайні магазини, в яких процес купівлі товарів все ж таки потребує ручної праці касира.

Основні функції касирів у супермаркетах це підрахування суми купованих товарів, прийом грошових коштів, та видача решти. Вважається, що цей процес може и має бути автоматизований, за допомогою такого напрямку в IT, як IOT[9].

Актуальність цієї ідеї дуже висока. По-перше, це вигідно звичайним людям, бо автоматизовані процеси виконуються швидше ніж ручні. Це сприяє вирішенню такої проблеми, як великі черги у магазинах. По-друге, це може бути вигідно власникам магазинів, за умовою якщо вони будуть отримувати більший прибуток, або витрачати менше коштів на утримання магазинів (зарплатня касира). Звісно система для автоматизування закупок має бути легко спроектована, щоб по-перше, клієнти дуже швидко розуміли як її використовувати, по-друге, вона легко інтегрувалась до різноманітних супермаркетів. Також вона має бути не дуже дорогою (не більше ніж вартість утримання касирів). При цих умовах є всі підстави, щоб розробити цю систему та пропонувати її людям.

Під автоматизацією процесу покупки товарів розуміється система, яка дозволяє клієнту створити свій персональний акаунт, покласти кошти на цей акаунт, прийти до любого супермаркету, у якому інтегрована ця система, придбати потрібні товари, оплативши їх за допомогою власного смартфона. Як можна побачити при цьому сценарії касир не потрібен. Звісно система передбачає також апаратну реалізацію, але у рамках цього диплому буде зроблено опір на програмну реалізацію системи.

До апаратної реалізації відноситься :

– маркування товарів, для їх ідентифікації;

– створення безпечних виходів з магазину, щоб через них неможливо було пронести

маркованих неоплачених товар.

– створення терміналу оплати товарів. Функції термінала :

1. визначає групу товарів, які бажає оплатити користувач;
2. визначає суму вартості цих товарів;
3. має графічний інтерфейс взаємодії;
4. оплата за карткою;
5. оплата за телефоном;
6. оплата часткової групи товарів.

До програмної реалізації відноситься :

– створення мобільного додатку з можливістю оплати групи товарів (емуляція);

– створення сайту за наступними умовами:

1. реєстрація користувачів, з наступним розділенням на ролі (адміністратор, користувач, власник магазину);
2. аутентіфікація користувачів за соціальними мережами (VK, Facebook, LinkedIn);
3. використання 3-слойної архітектури (UI, BLL, DAL);
4. локалізація сайту за декількома язиками, але обов’язково англійським;
5. використовування автоматичного маппінгу об’єктів;
6. використовування логеру для запису виключних ситуацій;
7. управління персональними даними;
8. управління персональних акаунтом користувача.

Використовуючи дану систему, користувач створює власний акаунт з коштами на даному веб-сайті, який обумовлений можливістю його поповнення, перевірки балансу, прикріплення карток, отримання історії транзакцій тощо. Це є дуже зручним для більшості користувачів. Так як, наприклад, смартфон з останніми можливостями мають не всі, а вихід до інтернету з власної оселі має чи не кожна людина у великому мегаполісі.

Використання даної системи у магазині, звісно потребує мобільного телефону, але за реальної реалізації, вона може бути адаптована, наприклад, під картки банків, які також є майже у всіх сімей.

Отже, виходячи з усього вищеназваного, можна зробити висновок, що тема атестаційної роботи є актуальною. Даний проект надає можливість здійснювати купівлю товарів у магазинах набагато легшим способом, аніж зараз.

Метою атестаційної роботи є розглядання існуючих аналогів, аналіз предметної області, моделювання та проектування програмного рішення та реалізація програмної частини системи, або прототипу. У перспективі цей прототип може буті розвинений до реальної комерційної системи та розгортуватись у будь-яких магазинах з касовою системою оплати.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

Предметною областю даного проекту є «Програмна система автоматизування покупок у магазині – «SMART MARKET».

Основними проблемами цієї галузі є:

– ручна праця касира;

– великі черги у магазинах;

– власники магазинів мають платити касирам заробітну плату, що впливає на ціну їх продукції, що також невигідно покупцям;

– постійна проблема з видачею решти покупцеві, так як не завжди є потрібні купюри та монети.

База даних[8] потрібна для того, щоб зберігати інформацію про користувача (його транзакції, баланс його карток, можливі акції тощо), зберігати інформацію про магазин ( про розташування магазину, графік його роботи, можливі акції тощо), зберігати служебну інформацію (транзакції, обіг клієнтів, та інша інформація, яка може заінтересувати власників магазинів).

База даних повинна містити інформацію про:

– користувачів, їх персональні акаунти тощо;

– магазини, їхні адреси, адреси їх власних сайтів, якщо вони маються, можливо їх

продукція, якщо буде інтегрована база товарів магазину;

– інформація по транзакціям;

– інформація по рахункам користувачів.

Ця система зможе полегшити роботу людини такими функціями:

– значною перевагою у зручності використання;

– більша вигода, як власникам, так і клієнтам магазинів;

– дає можливість користуватись послугами на далекій відстані.

Слід зазначити, що реальна система повинна використовувати IoT технології. IoT (Internet of Things) — це мережа, що складається із взаємозв'язаних фізичних об'єктів (речей) або пристроїв, які мають вбудовані датчики, а також програмне забезпечення, що дозволяє здійснювати передачу і обмін даними між фізичним світом і комп'ютерними системами, за допомогою використання стандартних протоколів зв'язку. Крім датчиків, мережа може мати виконавчі пристрої, вбудовані у фізичні об'єкти і пов'язані між собою через дротові і бездротові мережі. Ці взаємопов'язані об'єкти (речі) мають можливість зчитування та приведення в дію, функцію програмування та ідентифікації, а також дозволяють виключити необхідність участі людини, за рахунок використання інтелектуальних інтерфейсів.

Основною концепцією IOT є можливість підключення всіляких об'єктів (речей), які людина може використовувати в повсякденному житті, наприклад, холодильник, кондиціонер, автомобіль, велосипед і навіть кросівки. Всі ці об'єкти (речі) повинні бути оснащені вбудованими датчиками або сенсорами, які мають можливість обробляти інформацію, що надходить з навколишнього середовища, обмінюватися нею і виконувати різні дії в залежності від отриманої інформації. Прикладом впровадження такої концепції є система «розумний будинок» або «розумна ферма». Ця система аналізує дані навколишнього середовища і в залежності від показників регулює температуру в приміщенні. У зимовий період регулюються інтенсивність опалення, а в разі спекотної погоди будинок має механізми відкривання і закривання вікон, завдяки чому провітрюється будинок, і все це відбувається без втручання людини.

Інтеграція з інтернетом має на увазі, що пристрої будуть використовувати IP-адресу як унікальний ідентифікатор. Проте, через обмежені адресні простори в IPv4 (що дозволяє використовувати 4,3 мільярда унікальних адрес), об'єктам ІР доведеться використовувати IPv6, який забезпечує унікальними адресами мережевого рівня не менше 300 млн. пристроїв на одного жителя Землі. Об'єктами в ІР будуть не тільки пристрої із сенсорними можливостями, але також пристрої, які виконують дії (наприклад, лампочки або замки, якими керують через Інтернет). Значною мірою, майбутнє інтернету речей не буде можливим без підтримки IPv6, отже, глобальне впровадження IPv6 у найближчі роки буде мати вирішальне значення для успішного розвитку ІР в майбутньому.

Для бездротової передачі даних особливо важливу роль в побудові інтернету речей відіграють такі характеристики, як ефективність, відмовостійкість, адаптивність, можливість самоорганізації. Основне зацікавлення в цьому сенсі представляє стандарт IEEE 802.15.4, що управляє доступом для організації енергоефективних персональних мереж, і є основою для таких протоколів, як ZigBee, WiFi, Bluetooth, 6LoWPAN.

ZigBee — це комунікаційна технологія, заснована на протоколі IEEE 802.15.4 для реалізації низькошвидкісних бездротових приватних мереж. ZigBee володіє такими характеристиками, як низьке енергоспоживання, низька швидкість передачі даних, низька вартість і висока пропускна здатність. В даний час ZigBee використовується в основному при передаванні інформації між різними речами електронного обладнання, які знаходяться в межах короткої відстані і швидкості передачі даних не дуже висока. Це, в основному периферійні пристрої (миша, клавіатура) і побутова електроніка (TV, DVD), а також пристрої промислового управління (монітори, датчики і засоби автоматизації).

WiFi — це локальна бездротова технологія, яка використовує 2,4 ГГц надвисокої частоти або 5 ГГц супер-високочастотної радіохвилі. Ця технологія дуже добре підходить для передавання великих обсягів даних по бездротовій мережі між пристроями, але це також вимагає багато енергії для роботи і має невеликий рівень пропускної здатності даних. При використанні цієї технології потрібно буде замінювати батареї у всіх пристроях на регулярній основі.

Bluetooth — це бездротова технологія, яка використовується для передачі даних в персональних мережах. Він передає дані по смузі частот від 2,4 до 2,485 ГГц і працює на коротших відстанях, ніж Wi-Fi. Ви можете синхронізувати пару пристроїв, таких як телефони, навушники, колонки, комп'ютери і багато іншого. З розвитком Bluetooth v4.0 з'явилася можливість реалізувати функцію низького енергоспоживання і збільшений радіус дії до декількох десятків метрів.

Серед провідних технологій важливу роль у розповсюдженні інтернету речей відіграють рішення PLC — технології побудови мереж передачі даних по лініях електропередач, оскільки у багатьох додатках присутній доступ до електромереж (наприклад, торгові автомати, банкомати, інтелектуальні лічильники, контролери освітлення спочатку підключені до мережі електропостачання). 6LoWPAN, який реалізує шар IPv6 як над IEEE 802.15.4, так і над PLC, будучи відкритим протоколом, стандартизованих IETF, відзначається як особливо важливий для розвитку інтернету речей.

## 1.2 Виявлення проблем та актуалізація рішень

Дрібні і середні продуктові магазини сьогодні представлені в такій кількості, що дуже важко вибрати найкращий. Асортимент товару скрізь практично однаковий, клієнтів, в середньому, теж. При такій великій конкуренції на плаву залишаються тільки найкращі, інші раптово закриваються. Вижити в період кризи і не втратити бізнес хоче кожен підприємець, тому необхідно з'ясувати проблеми і, вже сьогодні, почати з ними боротися.

Середньостатистичний магазин стикається з такими проблемами: прострочення товару, постійні зміни в складі персоналу, цінова категорія товару та асортимент. Ці проблеми потрібно розглядати детально і намагатися вирішити.

Основна проблема – постійно нові продавці. Змінюються вони з різних причин, але основна і, на жаль, найпоширеніша - недостача при переобліку. Уникнути цього можна, якщо поставитися до прийому персоналу дуже відповідально: вимагати рекомендації, передзвонити обов'язково на попереднє місце роботи, або, наприклад, автоматизувати процес закупівлі у своєму магазині, та більше уваги приділити важливішим проблемам.

У магазині постійно повинен перебувати старший (будь то господар або завідуюча), котрий повинен знімати касу якомога частіше (аж до 5-6 раз в день). Так підприємці перестраховуються від крадіжок. Ці проблеми зникають самі собою, якщо в магазинах будуть інтегровані IOT рішення.

Вибір асортименту - дуже важливий фактор в хорошому розвитку вашого бізнесу. Неможна дозволяти продавцям самостійно робити заявки (їх набагато простіше умовити взяти непотрібний товар), треба самостійно це контролювати. Дана система не зможе вирішити цю проблему, але якщо у магазині буде автоматизований один процес, та від цього підприємець отримає вигоду, тоді можна буде намагатися автоматизувати інші процеси. Дана проблема може бути вирішена за допомогою штучного інтелекту, якщо буде вестись звіт популярності покупок, така система сама зможе робити замовлення.

Терміни реалізації, зазначені на упаковці будь-якого товару, повинні обговорюватися з постачальниками. Відразу обговорюватися умови обміну або повернення. Підприємці намагаються укласти договори, в яких будете охоплено всі хвилюючі їх пункти. Терміни придатності у продуктів дуже часто підроблюються, але звісно підробити його можливо у будь-який час, під час виготовлення, під час обробки, закупівлі перевезення, інвентаризації тощо. Якщо, наприклад, продукти під час виготовлення будуть маркіруватись радіомаяками, у яких буде закладено інформацію про придатність продукту, риск підробки на наступних стадіях буде знижено, через технічну складність.

Торгових фірм, що пропонують сьогодні свій товар, величезна кількість, тому вибирати є з чого. Для користувача великою проблемою є визначатися з закупівлею потрібного, максимально вигідного товару, звісно це робота усіляких прас-навігаторів, але якщо ця система буде запам’ятовувати історію покупок з цінами та адресами магазинів, це сприятиме вирішенню проблеми.

Якщо серйозно поставитися до даних проблем, бути завбачливим та розбірливим в своїх діях, то магазин буде тільки процвітати з інтеграцією IOT рішень та автоматизацією важливих процесів.

## 1.3 Аналіз існуючих рішень

На даний момент такі системи тільки зароджуються. Широкого застосування вони поки що не отримали, але дуже швидко набирають популярність. Наприклад, компанія Amazon представила концепцію супермаркету майбутнього, повністю позбавленого від касових терміналів і черг, пов'язаних з оплатою покупок.

Нова платформа названа Amazon Go, <https://www.amazon.com/b?node=16008589011> . Ідея полягає в тому, щоб автоматично відстежувати дії відвідувачів у торговельному залі, контролюючи, які товари вони беруть з полиць або ставлять назад.

Для здійснення покупок відвідувачеві на вході в супермаркет буде досить відсканувати спеціальний код в додатку Amazon Go на смартфоні. Після це в справу вступить новітня система комп'ютерного зору з алгоритмами глибокого машинного навчання.

Аналізуючи дії покупців, а також інформацію від численних сенсорів, встановлених в торгових залах та на стелажах, комп'ютерна система зможе формувати точний список товарів, взятих кожним відвідувачем. Якщо користувач виставить який-небудь продукт назад на полицю, він буде автоматично «викреслять» з віртуальної корзини.

Після завершення шопінгу відвідувач може спокійно вийти з супермаркету, не піклуючись про оплату: гроші автоматично спишуться з рахунку, асоційованого з додатком Amazon Go.

В даний час нова система проходить тестування за участю співробітників Amazon. Супермаркет майбутнього пропонує готові до вживання продукти харчування, зокрема, легкі закуски.

Перший магазин Amazon Go, як очікується, розкриє двері на початку 2017 року, він розташується в Сіетлі.

У Москві компанія X5 Retail Group, <https://www.x5.ru/> , що володіє торговими мережами «Перехрестя», «Карусель», «Пятерочка», відкрив влітку 2013 року перший в місті інноваційний супермаркет.

Всі продукти в «супермаркеті майбутнього» забезпечать радіочастотними чіпами RFID, ця технологія дозволить не викладати товари на транспортну стрічку. Вартість вмісту візка порахують на виході, а розплатитися можна буде через спеціальний термінал за допомогою банківської карти або навіть мобільного телефону.

Розумними в магазині стануть навіть полки, які будуть сповіщати менеджерів, які товари закінчуються в залі або на складі. Творці супермаркету також планують ввести систему розпізнавання осіб для ведення вікової статистики по покупцям. Все це дозволить скоротити витрати як мінімум на 20%.

Експерти, втім, налаштовані скептично. По-перше, технологія RFID має свої недоліки: мітки погано читаються з металевих поверхонь і ємностей з рідинами. А якщо товар загорнути у фольгу, сигнал і зовсім не пройде. Але головна проблема - за чий рахунок встановлювати мітки RFID, кожна з яких коштує три рубля. Наприклад, американська сісти Wal-Mart вже багато років впроваджує ці технології, але до сих пір не зуміла домовитися з постачальниками. Також слід відмітити, що в інтернеті більше немає інформації щодо цього проекту, тому можна припустити, що він був провальний.

Магазини без продавців з'являться в Японії до 2025 року. Уряд Японії домовився про співпрацю з п'ятьма найбільшими операторами мереж супермаркетів (Seven & I Holdings, Lawson, Ministop, FamilyMart Uny Holdings і East Japan Railway), щоб спільно розробити систему автоматичних магазинів без продавців до 2025 року. Про це повідомляє видання Nikkei Asian Review.

Нова система дозволить не пробивати штрих-код кожного товару окремо, а зчитувати інформацію продуктів в кошику покупця одночасно з допомогою спеціальних чіпів. Впровадження такої системи в 50 тис. Магазинів країни буде коштувати близько 46 млн. доларів США.

У чіпах буде міститися інформація про дату і місце виготовлення продукту, а також про дату закінчення терміну придатності. Інформація може бути переглянута віддалено, що дозволить виробникам перевіряти стан продукції для коригування планів продажів і виробництва. Також інформація буде доступна і покупцям. Вартість одного чіпа обійдеться в 10 ієн (0,09 доларів).

Міністерство економіки, торгівлі і промисловості Японії прийняв рішення надати підтримку виробникам для масового випуску таких чіпів. Саме впровадження автоматичних магазинів пов'язано з нестачею продавців в Японії.

У Шанхаї відкрився перший магазин без продавців, шведська інноваційна компанія Wheelys відкрила в Шанхаї перший в світі магазин без продавців. Магазин працює 24 години на добу, сім днів на тиждень, і це тільки пілотний проект, компанія розраховує продавати свою концепцію по франшизі. Варто відзначити, що раніше про плани відкрити подібний магазин повідомляв Amazon, але, як можна побачити, гіганта ринку переграла зовсім невелика компанія.

Після успішного регіонального тестування в маленькому шведському містечку в минулому році, Wheelys відкрила магазин Wheely247 в Шанхаї як приклад працездатності концепції в великих містах.

Купувати в такому магазині нескладно. Клієнт встановлює на смартфон додаток, що дозволяє отримати доступ до магазину. Для покупки товару потрібно просто відсканувати штрих-код товару, який він хоче придбати, і при виході з магазину додаток списує з кредитної картки вартість покупок.

З метою попередження крадіжок весь простір магазину проглядається відеокамерами.

Надалі Wheelys планує продавати ліцензії на використання технології, так що будь-який рітейлер може інтегрувати її в уже існуючі магазини. За словами компанії, «те, що Uber зробив для таксі, ми робимо для роздробу». Зараз магазин працює в форматі бета-тестування, тобто для його відвідування ще треба залишити заявку на сайті, це зроблено, щоб уникнути напливу цікавих роззяв.

## 1.4 Аналіз основних технологій розробки

Технологія ASP.NET – технологія створення веб-застосунків і веб-сервісів від компанії Майкрософт. Вона є складовою частиною платформи Microsoft.NET[1] і розвитком старішої технології Microsoft ASP. У цей час останньою версією цієї технології є ASP.NET Core.

Технологія ASP.NET зовні багато в чому зберігає схожість із старішою технологією ASP, що дозволяє розробникам відносно легко перейти на ASP.NET. У той же час внутрішній устрій ASP.NET істотно відрізняється від ASP, оскільки вона заснована на платформі. NET і, отже, використовує всі нові можливості, що надаються цією платформою.

Хоча ASP.NET бере свою назву від старої технології Microsoft ASP, вона значно від неї відрізняється. Microsoft повністю перебудувала ASP.NET, ґрунтуючись на Common Language Runtime (CLR), який є основою всіх застосунків Microsoft .NET. Розробники можуть писати код[13] для ASP.NET, використовуючи практично будь-які мови програмування, що входять у комплект .NET Framework (C#[3], Visual Basic.NET, і JScript.NET). ASP.NET має перевагу у швидкості в порівнянні зі скриптовими технологіями, тому що при першому зверненні код[11] компілюється і поміщається в спеціальний кеш, і згодом тільки виконується, не вимагаючи витрат часу на парсинг, оптимізацію, і т. д.

Переваги ASP.NET:

– ASP.NET має перевагу у швидкості в порівнянні з іншими технологіями, заснованими на скриптах (PHP, тощо);

– розширюваний набір елементів управління і бібліотек класів дозволяє швидше розробляти застосунки;

– ASP.NET спирається на багатомовні можливості .NET, що дозволяє писати код[7] сторінок на C#, VB, C/C++ та інших;

– поділ візуальної частини та бізнес-логіки;

– розширювана модель обробки запитів.

Крім сайту система також матиме мобільний додаток. Основною задачею мобільного додатку є оплата товарів у розумному магазині. Додаток дозволятиме авторизуватись і отримати доступ до частини інформації, яка відображається на сайті, крім цьго через додаток можна буде продивлюватись список продуктів, який користувач хоче купити у магазині, деякі продукти можна буде виключити з оплати тощо. Крім того мобільна версія має рахувати сумму покупки, фіксувати чи вистачає коштів на картці користувача, відображати потрібні повідомлення тощо. Для даних задач вирішено використовувати технологію Xamarin, яка дозволяє будувати кросплатформенні мобільні додатки на платформі Mono.

XAMARIN - це фреймворк для розробки кросплатформених мобільних додатків (iOS, Android, Windows Phone) з використанням мови C#[10]. Використовуючи Xamarin програміст може використовувати звичну для нього мову із знайомими йому особливостями. При чому має повний доступ до всіх можливостей SDK платформи й рідному механізму створення UI, отримуючи на виході додаток, що, нічим не відрізняється від нативних і не поступається їм у продуктивності.Метою роботи є дослідження платформи Xamarin, її використання для розробки кросплатформених додатків, дослідження процесу розробки, можливості та особливості фреймворку. Під об’єктом дослідження розуміємо процеси програмування. Предметом дослідження постають особливості процесу розробки кросплатформених додатків з використанням фреймворку Xamarin. Xamarin заснований на open-source реалізації платформи .NET - Mono. Ця реалізація включає в себе власний компілятор C #, середовище виконання та основні .NET бібліотеки. Важливо, що розробкою Xamarin займаються ті ж люди, що і розробкою Mono. Xamarin враховує розходження між способами попередньої компіляції різних платформ (Android, Windows Phone – Just-in-time, iOS – Aheadof-time), надаючи окремі компілятори для кожної з цих платформ, які дозволяють на виході отримувати справжні нативні додатки, які виконуються поза контекстом браузера і можуть використовувати всі апаратні і програмні ресурси платформи. Для кожної платформи Xamarin надає можливість використовувати нативні засоби розробки UI та нативні елементи призначеного для користувача інтерфейсу.

Також основною задачею системи є зберігання великої кількості інформації, яка використовується у сценаріях системи. Microsoft SQL Server — комерційна система керування базами даних, що розповсюджується корпорацією Microsoft. Мова, що використовується для запитів — Transact-SQL, створена спільно Microsoft та Sybase. Transact-SQL є реалізацією стандарту ANSI / ISO щодо структурованої мови запитів SQL із розширеннями. Використовується як для невеликих і середніх за розміром баз даних, так і для великих баз даних масштабу підприємства. Багато років вдало конкурує з іншими системами керування базами даних SQL Server 2014.

Наступна версія SQL Server—SQL Server 2014. Мета випуску SQL Server 2014 —зробити керування даними самоналаштовуваним, самоорганізованим та самопідтримуваним. SQL Server 2014 також включає підтримку структурованих і напівструктурованих даних, у тому числі цифрові медіа-формати для зображень, звуків, відео й інших мультимедійних даних. Ключовим нововведенням SQL Server 2014 є розвинені засоби управління ресурсами (resource governor), що дозволяють ефективно управляти і розподіляти робоче навантаження за допомогою відстежування рівня завантаження процесора і обсягу пам'яті, що займають працюючі застосунки. Microsoft виділяє засоби управління на основі політик, розширені можливості з складання звітів і проведенню аналізу, а також розвинені засоби управління інтелектуальними ресурсами підприємства. У продукті з'явилася повноцінна підтримка неструктурованих даних і покращена система шифрування інформації. Крім того, варто виділити розширені функції роботи з геолокалізованими даними.

Серед нових можливостей і удосконалень Microsoft SQL Server 2014 також слід зазначити появу нових типів даних, а саме — просторових даних, кращу сумісність з застосунками сторонніх розробників, наприклад Oracle, тіснішу інтеграцію з Office, оптимізовані засоби шифрування даних, засоби управління на основі політик, а також покращені інструменти звітності і аналізу.

Продукт SQL Server 2014 пропонується в семи модифікаціях, орієнтованих на різні категорії користувачів, зокрема, на робочі групи, крупних корпоративних замовників, розробників і інших. Ціни на SQL Server 2014 в порівнянні з попередніми версіями системи управління базами даних залишаться тим самим. Крім того, Microsoft пропонує користувачам безкоштовні версії продукту (з обмеженими можливостями)—SQL Server 2014 Express і SQL Server Compact. SQL Server 2014 включає низку вдосконалень для роботи з критичними бізнес-застосунками і бізнес-аналітикою як в традиційних дата-центрах, так і в приватних, публічних і гібридних хмарних середовищах. Серед нових можливостей SQL Server 2012 виділяються SQL Server AlwaysOn (рішення підтримки високого рівня доступності даних та аварійного відновлення), xVelocity (технологія збільшення продуктивності сховищ даних та програм бізнес-аналітики), нові рішення в області візуалізації PowerPivot і PowerView для створення звітів і аналітичних програм з Excel і SharePoint, покращені інструменти для інтеграції даних і управління ними, включаючи SQL Server Data Quality Services і Master Data Services, нова семантична модель бізнес-аналітики та інструмент для адміністраторів баз даних і розробників застосунків SQL Server Data Tools.

Також Microsoft зробила значні інвестиції в області Big Data, а саме в інтеграцію SQL Server і популярних інструментів для бізнес-аналітики з неструктурованою інформацією.

Для подібних задач потрібно використовувати базу данних MS SQL Server 2014, так як вона буде добре інтегрована до ASP.NET MVC 5 фреймворку, та дуже добре працюъ з ORM-системою Entity Framework 6. ADO.NET Entity Framework (EF) - об'єктно-орієнтована технологія доступу до даних, є object-relational mapping (ORM) рішенням для .NET Framework від Microsoft. Надає можливість взаємодії з об'єктами як за допомогою LINQ у вигляді LINQ to Entities, так і з використанням Entity SQL. Для полегшення побудови web-рішень використовується як ADO.NET Data Services (Astoria), так і зв'язка з Windows Communication Foundation і Windows Presentation Foundation, що дозволяє будувати багаторівневі додатки, реалізуючи один з шаблонів проектування MVC, MVP або MVVM. Для веб-додатку потрібно сформувати слой доступу к даним, який буде використовувати LINQ to Entities.

LINQ to Entities – це альтернативний інтерфейс LINQ API, який використовується для звернення до бази даних. Він відокремлює сутнісну об'єктну модель даних від фізичної бази даних, вводячи логічне відображення між ними. Так, наприклад, схеми реляційних баз даних не завжди підходять для побудови об'єктно-орієнтованих програм і в результаті ми маємо об'єктну модель програми, котра суттєво різниться від логічної моделі даних, в цьому випадку використовується LINQ to Entities, який використовує модель EDM (Entity Data Model). Тобто, якщо вам потрібно послабити зв'язок між вашою сутнісної об'єктною моделлю даних і фізичною моделлю даних, наприклад, якщо ваші сутнісні об'єкти конструюються з декількох таблиць або вам потрібна велика гнучкість в моделюванні ваших сутнісних об'єктів використовуйте LINQ to Entities.

## 1.5 Постановка задачі

На основі проаналізованих даних, з урахуванням усіх переваг та недоліків, у випускній атестаційній роботі бакалавра необхідно спроектувати та релізувати систему оплати товарів за допомогою мобільного телефона та веб-сайту. Веб-система має бути легкою у освоєнні, бо розрахована на використання різними сегментами суспільства. Основною задачею сайта є відображання персональних данних користувача, його рахунків, вдалий опис операції та функцій даної системи. Для даних задач дуже вдало підходить технологія ASP.NET MVC 5.

Дана система буде конкурентоспроможним за допомогою зручного та простого інтерфейсу, матиме цікавий і корисний функціонал.

Даний продукт буде вирішувати наступні проблеми, які присутні в популярних сервісах:

– неефективність процесу купівлі товарів у торгових закладах, через повільну ручну обробку товарів касиром;

– залежність від готівки при покупках;

– постійна проблема пошуку нових працівників на посаду касирів;

– підробка термінів споживання на певних товарах.

Даний продукт буде відрізнятися наступними можливостями:

* можливість сплати використовуючи мобільний телефон;
* чудова можливість вибору продуктів для покупки;
* можливість просмотру історії здійснених покупок.

# 2 ФОРМУВАННЯ ВИМОГ ПО ПРОГРАМНОЇ СИСТЕМИ

Вимоги до програмного забезпечення — набір вимог щодо властивостей, якості та функцій програмного забезпечення, що буде розроблено, або знаходиться у розробці. Вимоги визначаються в процесі аналізу вимог та фіксуються в специфікації вимог, діаграмах прецедентів та інших артефактах процесу аналізу та розробки вимог.

Розробка вимог до програмної системи може бути розділена на декілька етапів:

– знаходження вимог (збір, визначення потреб зацікавлених осіб та систем);

– аналіз вимог (перевірка цілісності та закінченості);

– специфікація (документування вимог);

– тестування вимог.

Аналіз вимог полягає в визначенні потреб та умов, які висуваються щодо нового, чи зміненого продукту, враховуючи можливо конфліктні вимоги різних замовників, таких як користувачі.

Аналіз вимог є критичним для успішної розробки проекту. Вимоги мають бути задокументованими, вимірними, тестовими, пов'язаними з бізнес-потребами, і описаними з рівнем деталізації достатнім для конструювання системи. Вимоги можуть бути архітектурними, структурними, поведінковими, функціональними, та не функціональними.

Види вимог за рівнями:

– бізнес-вимоги — визначають призначення ПЗ, можуть описуватися в документі про бачення (англ. vision) та документі про межі проекту (англ. scope);

– вимоги користувача — визначають набір завдань користувача, які повинна вирішувати програма, а також сценарії їхнього вирішення в системі. Ці вимоги можуть мати вигляд тверджень, варіантів використання, історій користувача, сценаріїв взаємодії;

– функціональні вимоги — визначають «що» повинен робити програмний продукт. Ці вимоги описуються в документі Специфікації програмного забезпечення (англ. SRS).

Види вимог за характером:

– функціональний характер — вимоги до поведінки системи:

1) бізнес-вимоги;

2) вимоги користувача;

3) функціональні вимоги.

– нефункціональний характер — вимоги до характеру поведінки системи:

1) бізнес-правила — визначають обмеження, що витікають з предметної області;

2) системні вимоги — вимоги до програмних інтерфейсів, надійності, обладнанню;

3 ) атрибути якості;

4) зовнішні системи та інтерфейси;

5) обмеження.

Веб-додаток має наступні функціональні вимоги, функціонал:

– зареєструвати власний акаунт, з можливістю віновлення даних за допомогою електронної пошти;

– прикріпити карту свого банку до сайту, через використання api сторонніх сайтів;

– мати відображання балансу рахунку, номеру картки, валюти тощо;

– мати легкий зрозумілий інтерфейс, з боковим згортаючим меню;

– отримувати результат кожної операції відображати у модальному вікні або вспливаючим блоком;

– мати локалізацію додатку якнайменше на два язика;

– мати безпечний алгоритм аутентифікації користувача, заснований на зашифрованому токені доступу, який має срок дії.

Веб-додаток має наступні системні вимоги:

– сайт повинен бути побудований за допомогою технології ASP.NET MVC 5;

– мати адаптований дизайн під планшети, смартфони, лаптопи тощо;

– справно працювати в усіх популярних браузерах (Chrome, Mozilla, Opera).

Мобільний додаток повинен надавати користувачеві:

– авторизуватися через WEB API сайту, захищаючи персональні дані користувачів, можно використовувати для цього протокол HTTPS;

– вибрати продукти, які користувач хоче купити. за кастомізованим ListView, який відображає назву продукту, його ціну, та checkbox вибору;

– вибрати картку за допомогою якой він хоче оплатити продукти;

– оплатити вибрані продукти, з надысланням транзауцыъ до власних api з подальшим занесеням ії до бази даних.

# 3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 3.1 UML проектування ПЗ

В сучасних інформаційних технологіях фаза життєвого циклу, на якій фіксуються вимоги на розробку програмного забезпечення, є визначальною для його якості, термінів та вартості робіт. Саме на цій фазі мають бути зафіксовані реальні потреби користувачів, що стосуються функціональних, операційних та сервісних можливостей.

Для кращого розуміння програмної системи, що розроблюється розглянемо діаграму прецедентів (use case).

Діаграма прецедентів – в [UML](https://uk.wikipedia.org/wiki/UML)[6] [діаграма](https://uk.wikipedia.org/wiki/%D0%94%D1%96%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%B0), на якій зображено відношення між акторами та прецедентами в системі. Також перекладається як діаграма варіантів використання.

Діаграма прецедентів є [графом](https://uk.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), що складається з множини акторів, прецедентів (варіантів використання), асоціацій між акторами та прецедентами, відношень серед прецедентів, та відношень узагальнення між акторами. Діаграми прецедентів відображають елементи моделі варіантів використання.

Суть даної діаграми полягає в наступному: проектовану систему представлено у вигляді безлічі сутностей чи акторів, що взаємодіють із системою за допомогою варіантів використання. Варіант використання (use case) використовується для опису послуг, які система надає актору. Іншими словами, кожен варіант використання визначає деякий набір дій, який виконує система при діалозі з актором.

Розпочинається створення діаграми використання із ідентифікації варіантів використання та дійових осіб. Це необхідно для того, щоб:

– чітко розмежувати систему і її оточення;

– визначити, які дійові особи і як саме взаємодіють з системою, який функціонал (варіанти використання) очікується від системи;

– визначити і описати загальні поняття, які необхідні для детального опису функціонала системи (прецедентів).

Алгоритм створення діаграми прецедентів:

– визначення дійових осіб;

– визначення варіантів використання;

– складання опису кожного варіанту використання;

– опис моделі прецедентів в цілому.

На рисунку 3.1 можна побачити діаграму прецедентів.

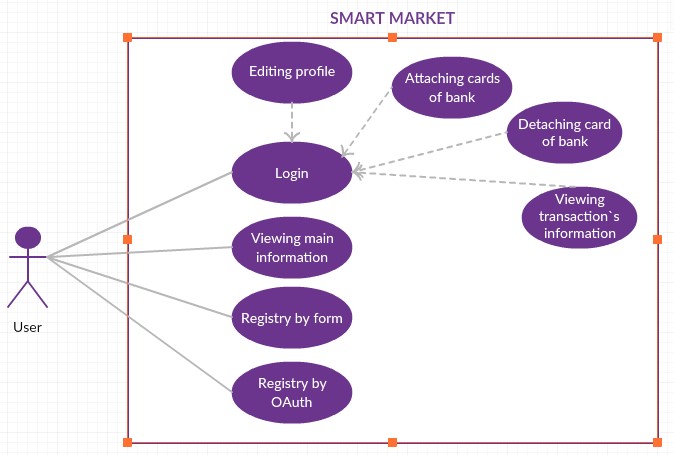


Рисунок 3.1 – Діаграма прецедентів ролі «User»

Таким чином діаграма прецедентів чітко відображує акторів, прецеденти, а також відносини між ними і наочно ілюструє узагальнене представлення функціонального призначення системи.

Діаграма прецедентів містить у собі одну роль User. котрий має можливість зареєструватися декількома шляхами на даному сайті, а також дивитися основну публічну інформацію, яка складається з опису системи, інноваційних рішень, списку підтримуємих магазинів тощо. Аутентифікувавшись (Login) користувач має можливість змінити інформацію у своєму профілі, прикріпити карту свого банку, відкріпити ії, та продивитись інформацію щодо транзакцій.

Використовуючи її, розробнику легше розуміти систему, підсистему або класи, а також поглянути ззовні на переваги використання елементів для того чи іншого контексту.

На етапі аналізу вимог і попереднього проектування системи на основі побудованих моделей системних прецедентів будуються діаграми класів системи.

Діаграма класів, будучи логічним поданням моделі, представляє детальну інформацію про структуру моделі системи з використанням термінології класів об’єктно-орієнтованого програмування, а саме: про внутрішній устрій системи (про архітектуру системи). На діаграмі класів можуть бути вказані внутрішня структура і типи відносин між окремими об’єктами і підсистемами, що призводить до розвитку концептуальної моделі системи.

Клас в мові UML позначає деяку множину об’єктів, що володіють однаковою структурою і взаємозв’язками з об’єктами інших класів. У діаграмах класів системи вказуються об’єкти з моделі системних прецедентів з їх описом і вказівкою взаємозв’язків між класами.

У програмі реалізовані класи, вміст яких зображено на рис. 3.2.

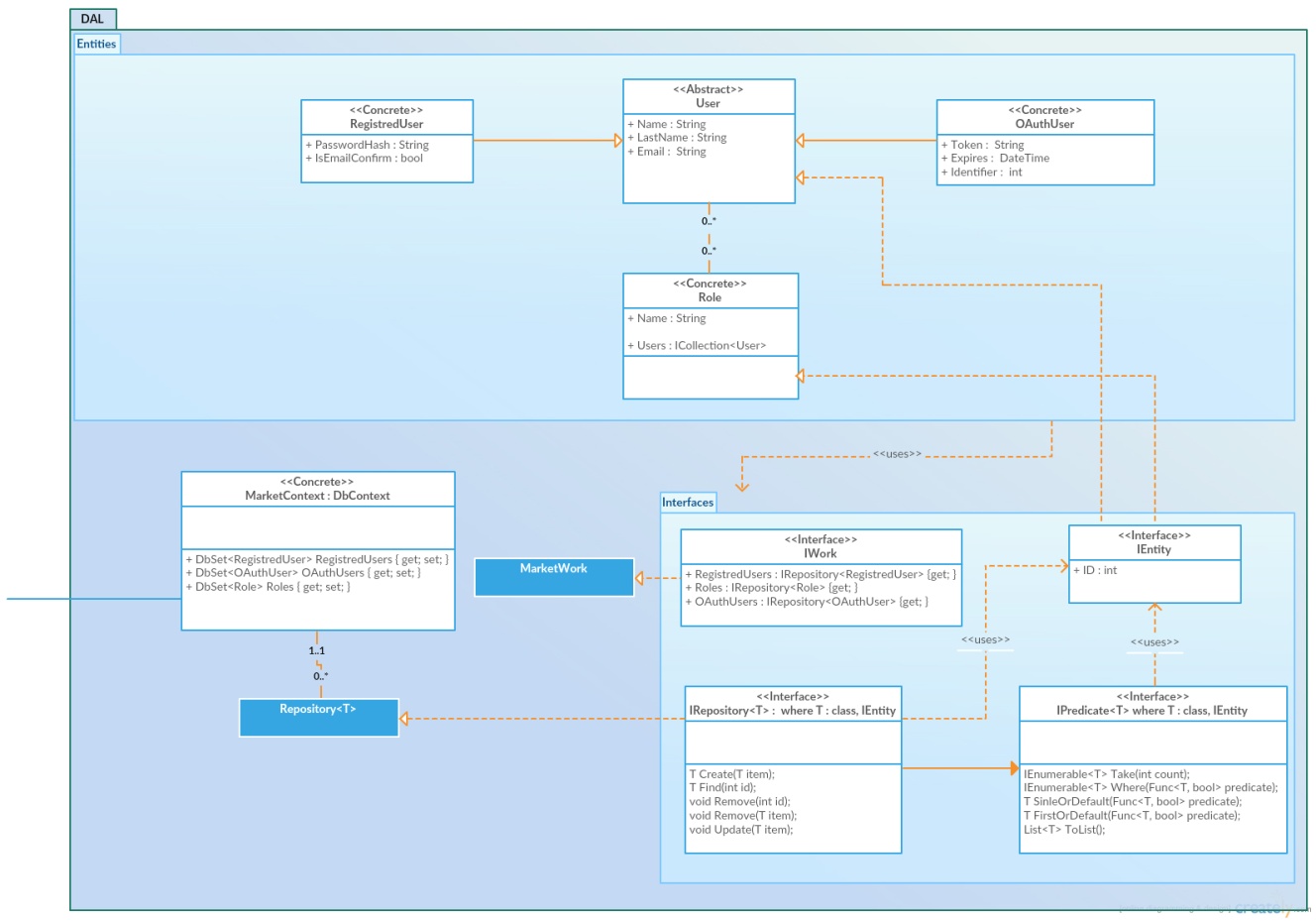


Рисунок 3.2 – Діаграма класів для слою DAL

Синтаксис діаграм класів є ефективним засобом структурування вимог до елементів проектованої системи, до їх даних, інтерфейсів, функціональності.

Як можна побачити слой DAL містить два підпространство імен – Entities та Interfaces. Entities містить у собі класи-сутності, такі як User, Balance, Role тощо. Interfaces містить у собі набір усіх інтерфейсів, які використовуються цим слоєм. Також у корні розміщено клас Repository<TEntity>, котрий детально описаний у розділі архітектури та клас MarketWork, основною задачею якого є створення потрібних репозиторієв.

Основна задача слою DAL (Data Access Layer) це дотримання даних з істочника даних, усунення зайвої інформації та передавання у зручному форматі до верхнього слою.

Цей слой буде використовуватися тільки слоєм вищим за рівнем, тобто BLL. Основна задача BLL (Business Logic Layer) це обробка існуючих моделей, виконання бізнес-логіки, робота з ресурсами, формування ViewModels, та передавання на UI слой зручних для відображання моделей.

Діаграма класів – це тип діаграм, які частіше за все використовуються при моделюванні об’єктно-орієнтованих систем.

Зв’язки у діаграмі класів.

Асоціація показує, що об'єкти однієї сутності (класу) пов'язані з об'єктами іншої сутності.

Якщо між двома класами визначена асоціація, то можна переміщатися від об'єктів одного класу до об'єктів іншого. Цілком припустимі випадки, коли обидва кінці асоціації відносяться до одного і того ж класу. Це означає, що з об'єктом деякого класу дозволено зв'язати інші об'єкти з того ж класу. Асоціація, що зв'язує два класи, називається бінарною. Можна, хоча це рідко буває необхідним, створювати асоціації, що зв'язують відразу кілька класів; вони називаються n-арнимі. Графічно асоціація зображується у вигляді лінії, що з'єднує клас сам з собою або з іншими класами.

Асоціації може бути присвоєно ім'я, яке описує природу відносини. Зазвичай ім'я асоціації не вказується, якщо тільки ви не хочете явно задати для неї рольові імена або у вашій моделі настільки багато асоціацій, що виникає необхідність посилатися на них і відрізняти один від одного. Ім'я буде особливо корисним, якщо між одними і тими ж класами існує кілька різних асоціацій.

Клас, що бере участь в асоціації, грає в ній деяку роль. По суті, це "обличчя", яким клас, що знаходиться на одній стороні асоціації, звернений до класу з іншого її боку. Ви можете явно позначити роль, яку клас грає в асоціації.

Агрегація. Проста асоціація між двома класами, яка відображає структурне відношення між рівноправними сутностями, коли обидва класи знаходяться на одному концептуальному рівні, і ні один не є більш важливим, ніж інший. Але іноді доводиться моделювати відношення типу «частина/ціле», в якому один з класів має більш високий ранг (ціле) і складається з декількох менших за рангом (частин). Ставлення такого типу називають агрегацією; воно зараховане до відносин типу «має» (з урахуванням того, що об'єкт-ціле має кілька об'єктів-частин). Агрегація є окремим випадком асоціації і зображується у вигляді простої асоціації з незафарбованим ромбом з боку «цілого».

Графічно агрегація представляється порожнім ромбом на блоці класу, і лінією, яка від цього ромба до міститься класу.

Композиція - більш суворий варіант агрегації. Відома також як агрегація за значенням.

Композиція має жорстку залежність часу існування екземплярів класу контейнера та примірників містяться класів. Якщо контейнер буде знищений, то весь його вміст буде також знищено.

Графічно представляється як і агрегація, але з зафарбовані ромбиком.

## 3.2 Проектування архітектури ПЗ

Веб-додаток складатиметься з трьох слоїв :

- UI, User Interface, слой графічного інтерфейсу, містить у собі контролери, Views (класи відображення), класи-хєлпери тощо;

- BLL, Business Logic Layer, слой бізнес логіки, містить у собі основну логіку додатку (класи-менеджери), сервіси по управлінню ресурсами, правила маппінгу об’єктів тощо.

- DAL, Data Access Layer, слой доступу даних, містить у собі контекст даних від Entity Framework, класи-репозиторії (repositories), класи-сутності (entities).

Слід зазначити, що під слоєм у додатку ASP.NET MVC 5 розуміється збірка class library, яка буде яка буде інкапсулювати деякі дані, отримувати дані з більш нижчого слою (збірки), оброблювати їх та передавати вище. При цьому дуже важливо проектувати додаток таким чином, щоб слої або збірки не посилалися друг на друга через один рівень, та, щоб компоненти нижчого слою нічого не знали про слої та компоненти верхніх слоїв. Першим реалізованим рівнем (слоєм) буде Data Access Layer або рівень доступу до даних. Цей рівень зазвичай агрегує всі моделі даних, що зберігаються в БД, а також класи, через які йде взаємодія з БД. Проект «SmartMarket.BLL» буде представляти рівень доступу до даних. На рисунку 3.3 показано набір файлів цієї сборки. Зараз слід зазначити, що найважливішими класами є контекст та репозиторій (у даному випадку ми використовуємо generic repository). Generic repository відрізняється від звичайного репозиторія тим, що використовує один інтерфейс, один каркас для всіх моделей. Логікою створення репозиторієв займається класс «MarketWork», саме цю реалізацію ми будемо використовувати у вищих слоях. Слід також зазночити, що для кожного конкретного класу (реалізації) ми оголошуємо інтерфейс, щоб потім можна було використовувати який-небудь IOC-контейнер.

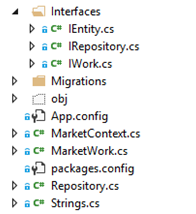
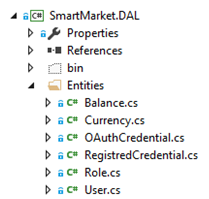


Рисунок 3.3 – Слой DAL (Data access layer)

Другим реалізованим слоєм буде BLL, Business Logic Layer, слой у якому зібрана уся логіка додатку. На рисунку 3.4 можна побачити списки файлів, які містить цей слой.

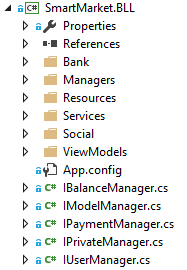


Рисунок 3.4 - Слой BLL (Business logic layer)

Як ми бачимо у корні оголошено інтерфейси менеджерів, які будуть використовуватись на більш вищому рівні (UI, User interface), як правило у контролерах. реалізації для цих інтерфейсів зібрано у папці «Managers», для них також виділено своє підпространство імен. Також цей слой містить «ViewModels» - це спеціальні класи, які використовуються для кожної зі сторінок і містять у собі усі данні які треба буде відобразити на сторінці. Також дуже важливими є сервіси (папка «Services») MapService, UserService, ResourceService, EmailService, SecurityService. Це все класи-помічники, вони вирішують другорядні завдання щодо безпеки, алгоритми та методи шифрування, відправки електронної пошти, управління ресурсами тощо. деякі з них також використовуються віщим слоєм.

І нарешті третій слой - це слой користувальницького інтерфейсу (User interface, UI), він зберігає у собі контролери, api-контролери та views. Мають мінімум логіки та зазвичай викликають менеджери за нижчого слою логіки, формують повідомлення валідації, викликають потрібні View та створюють для них потрібні ViewModels. На рисунку 3.5 ви можете побачити структуру слою UI.

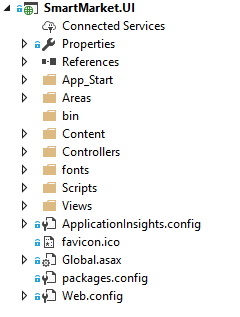


Рисунок 3.5 – Слой UI (User interface)

Мобільний додаток не розбивався на слої, та викликає api-контролери сайту задля проходження процесу аутентіфікації, отримання усіляких даних, запису деяких даних тощо. Він створений за шаблоном Xamarin forms, тому немає сенсу виносити його структуру, бо вона абсолютно ідентична з усіма іншими Android-додатками. Найважливішими класами є класи-activity, які виконують певну логіку, відображають потрібний layout та викликають потрібні функції через api-контролери.

На рисунку 3.6 можна побачити картку коду веб-додатку, вона наглядно ілюструє зв’язки між слоями сайту. Картку коду можна створити за допомогою Visual Studio Enterprise, у якій необхідно відкрити вкладку «Архітектура», далі картка коду, далі відкрити окно рішень вашого цільового проекту, далі перемістити потрібні збірки, класи, інтерфейси, компоненти на картку. Visual Studio сама прочитає цю інформацію та побудує карту коду. Microsoft Visual Studio — серія продуктів фірми Microsoft, які включають інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів. Ці продукти дозволяють розробляти як консольні програми, так і програми з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight.

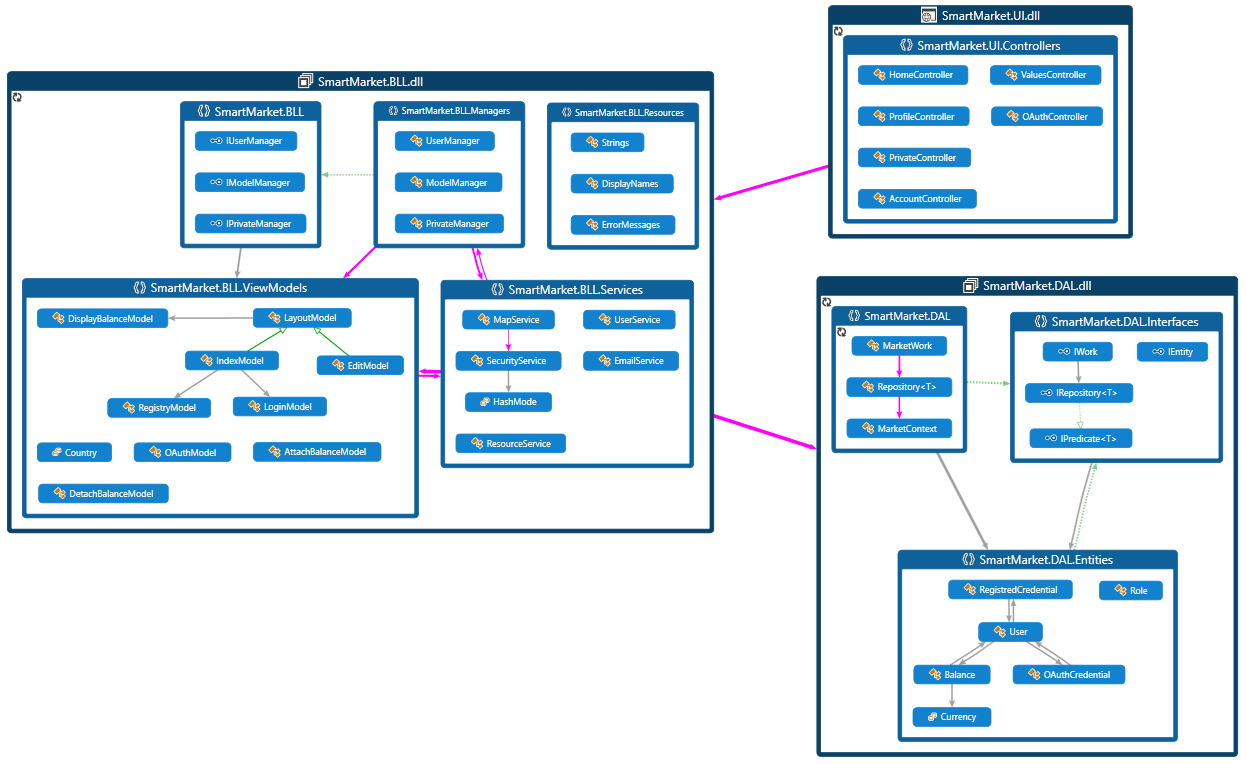


Рисунок 3.6 – Картка коду

Даний малюнок також зображено у презентації та буде детально розглянуто під час захисту перед екзаменаційною комісією. На ньому можна побачити усі залежності між слоями та їх компонентами. У карту коду винесені тільки найважливіші компоненти.

## 3.3 Проектування бази даних

База даних потрібна для сайту, щоб зберігати інформацію про користувачів, їх рахунки, операції тощо. У якості цільової СУБД було обрано MS SQL Server 2014. У якості ORM-системи було обрано технологію Entity Framework 6. У попередньому розділі ми описували слої сайту, серед яких був слой доступу даних, який зберігає у собі класи-моделі, які за допомогою Entity Framework мапяться до бази. На рисунку 3.7 ви можете, як моделі User та Balance виглядають у коді. На рисунку 3.8 та 3.9 ви можете побачити як ці класи конвертувались у таблички у даній базі даних.

Також слід зазначити, що серед своїх моделей ми можемо використовувати наслідування, та мапер Entity Framework створить аналог у базі даних. Зразу відповім, що Entity Framework може створити ієрархію з моделей трьома шляхами, та у даному проекті був випадок, коли дуже потрібно було використовувати цю функцію, але про цей випадок детально описано у наступному розділі.

Також треба добавити, що для правильної роботи Entity Framework потрібно оголосити клас контексту бази даних, це як би мовити аналог бази даних у коді, проксі, через який можна впливати на базу. На рисунку 3.10 ви можете побачити його реалізацію.

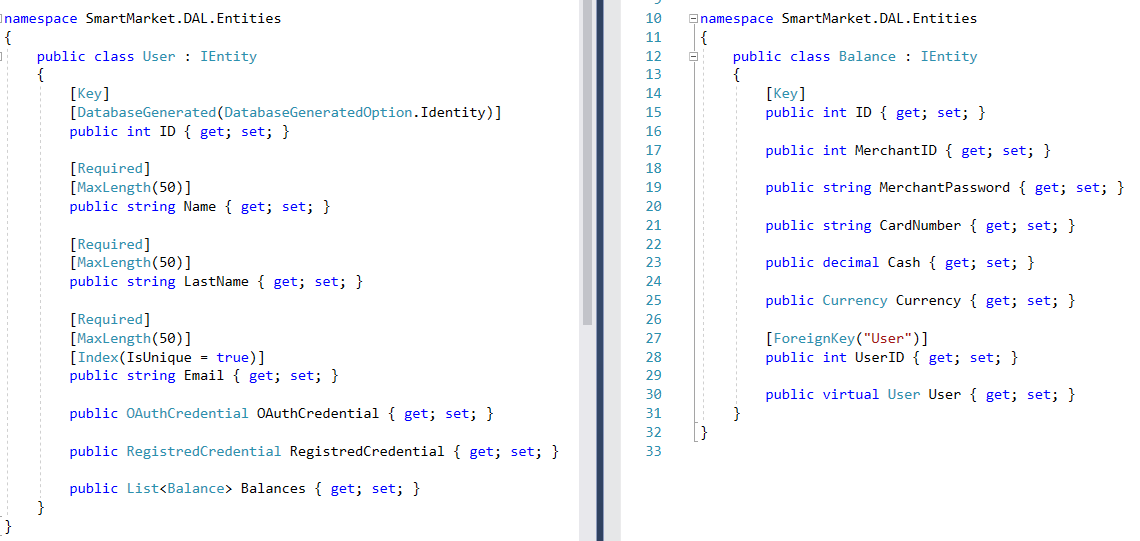


Рисунок 3.7 – Моделі User та Balance

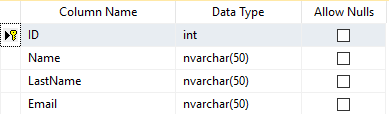


Рисунок 3.8 – Таблиця User

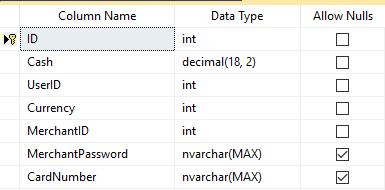


Рисунок 3.9 – Таблиця Balance

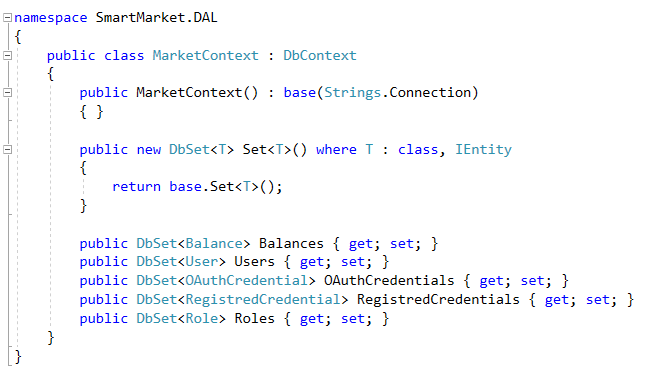


Рисунок 3.10 – Контекст бази даних

З реалізації контексту бази даних видно, що він оголошує свойства типу DbSet<T>, які аналогом табличок у базі даних, також він оголошує метод Set<T>, який додає додаткові правила до типу та визиває реалізацію базового класу. Цей метод повинен за типом повернути колекцію з бази даних, але звісно він не загружає усі дані.

## 3.4 Приклади найцікавіших алгоритмів та методів

Напевно один з самих цікавих класів, які мені довелось реалізувати у цьому проекті, це був Generic Repository. На рисунках 3.11 та 3.12 ви можете побачити його реалізацію. Цікава річ у тому, що загалом під кожну entity створюється свій репозиторій, який інкапсулює в собі клас контексту бази даних та через нього виконує необхідні функції. Тобто зазвичай кожна реалізація репозиторія працює тільки з одним типом даних, з однією сутністю, з одним класом. У даному випадку ми оголошуємо один репозиторій, з одним інтерфейсом (набором методів) для усіх сутностей. І з однієї сторони це дуже вигідно, бо нам потрібно писати менше коду для його реалізації, при використанні ми завжди використовуємо тільки один інтерфейс, у нас немає дублювання коду на рівні репозиторія.

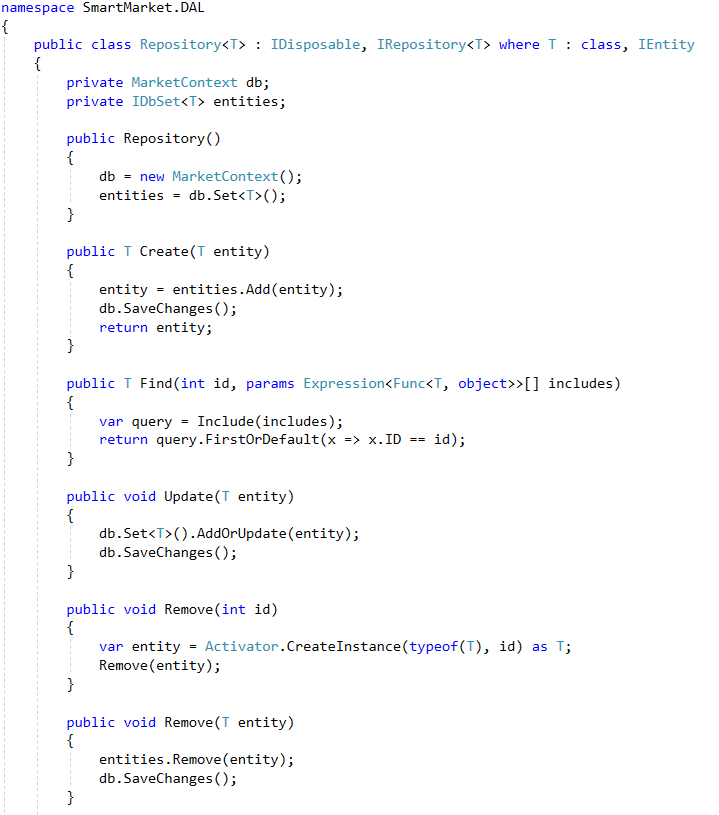


Рисунок 3.11 – Реалізація Generic Repository (частина 1)

Але при цьому є декілька мінусів, по-перше це знову ж таки уніфікований інтерфейс не завжди нам допомагає, бо для різних сутностей інколи треба використовувати різні методи. За умови, що ми створюємо Generic Repository, ми не маємо можливості оголосити метод, наприклад, FindByEmail тільки для сутності User. Якщо ми оголосимо такий метод у даному інтерфейсі, він автоматично буде використовуватись для кожної сутності, і коли ми створимо наприклад репозиторій для рахунків користувачів (Balance), буде дивно якщо він матиме цей метод. Другий мінус заключається у тому, що ми інколи повинні використовувати рефлексію для реалізації якихось методів. А рефлексія працює дуже повільно. Також мінус є напевно складність реалізації деяких методів. Наприклад як нам отримати сутність з усіма її залежними даними, наприклад користувача з усіма його рахунками, бо доступу до метода Include через колекцію DbSet у нас немає.

Як бачимо на рисунку, якщо ми не використовували б Generic Repository, ми повинні були б оголосити методи Find, Update, Remove, Create для всіх реалізацій репозиторієв.

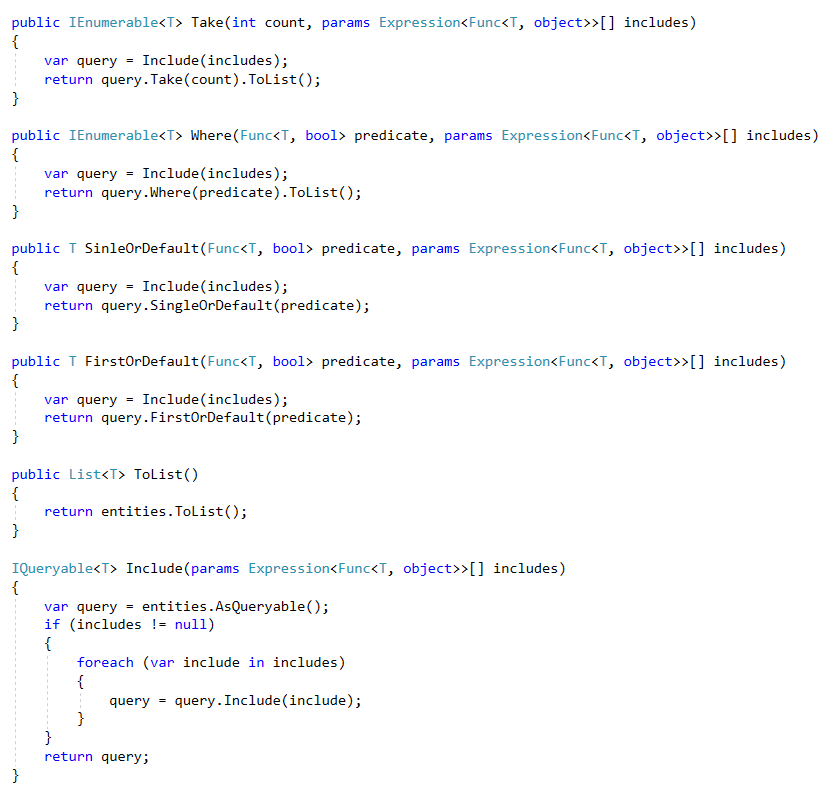


Рисунок 3.12 – Реалізація Generic Repository (частина 2)

На даному рисунку ми бачимо вирішення багатьох даних проблем. Першою проблемою була реалізація метода FindByEmail для сутності User, але не для інших сутностей. Ця проблема вирішується методами розширення колекцій, наприклад FirstOrDefault, прийматиме лямда-вираження, і вже це вираження буде підставляти до антологічного методу колекції з даними. Реалізацію Include можна також зробити через лямда-вираження. Попри все необхідно також було реалізувати інтерфейс IDisposable. Як відомо компанія Майкрософт презентувала розробникам програмного забезпечення уніфікований шаблон реалізації інтерфейсу IDisposable, реалізацію якого можна подивитися на рисунку 3.13.

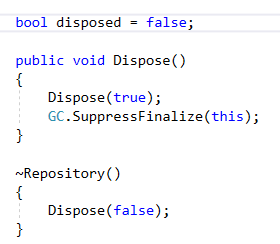
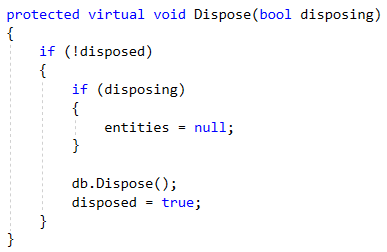
 

Рисунок 3.13 – Реалізація Generic Repository (частина 3, IDisposable)

Основне застосування цього інтерфейсу полягає в звільненні некерованих ресурсів. Складальник сміття автоматично звільняє пам'ять, виділену для керованого об'єкта, якщо цей об'єкт більше не використовується. Проте не може передбачити, коли буде виконано складання сміття. Крім того збирач сміття не має відомостей про некерованих ресурсів, таких як дескриптори вікон або відкриття файлів і потоків.

Використовуйте Dispose метод цього інтерфейсу, щоб явно звільняти некеровані ресурси разом зі складальником сміття. Користувач об'єкта може викликати цей метод, коли об'єкт більше не потрібен.

## 3.5 Створення UI / UX або іншого дизайну системи

Для сучасного інтернет проекту UX-дизайн відіграє важливу роль і визначає успішність його розвитку. Враховувати його ключові моменти потрібно на всіх етапах розробки. UX дизайн (User eXperience Design) – застосування методів проектування, спрямованих на досягнення максимально ефективної взаємодії користувача з системою.

Розбролювана система має наступні вимоги до UX- дизайну:

– landing-page, клієнт взаємозв’язана із сервером за допомогою AJAX, тому повинен мати одну сторінку, яка не потребує перезавантаження для оновлення інформації;

– вона оновлюється частково асинхронно;

– користувач має змогу легко знайти потрібний йому елемент на сайті;

– збоку сайту повинно розташовуватися меню;

– зверху сайту повинно розташовуватися часткове меню;

– контент сайту повинен змінюватися, коли користувач перемикається по посиланням.

На рисунку 3.14 ви можете бачити головний інтерфейс сайту.

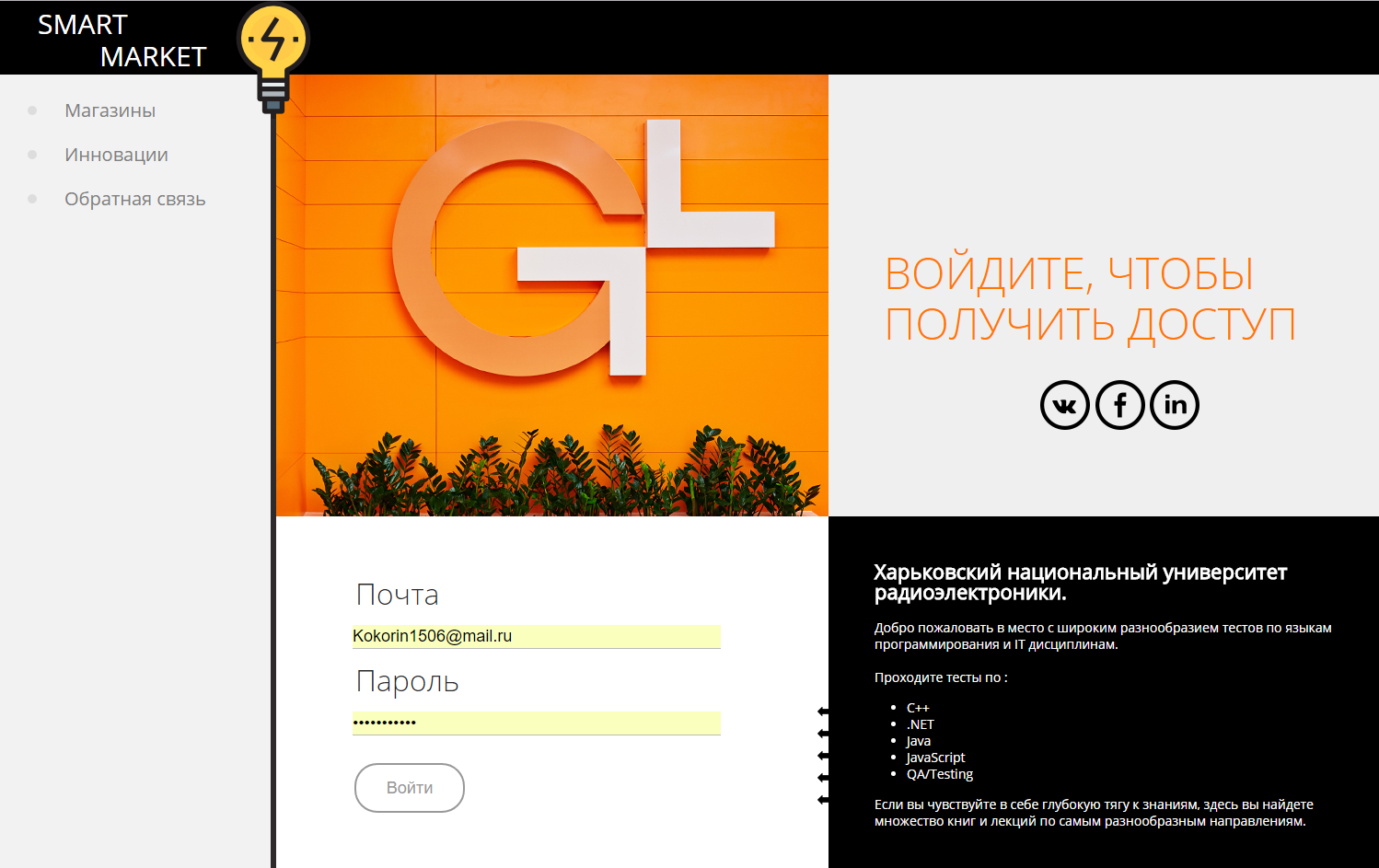


Рисунок 3.14 – Головний інтерфейс сайту

Також на рисунку 3.15 можна побачити інтерфейс панелі аутентифікованого користувача, на якій містяться іконки зі створення рахунку, редагування профілю, відкріплення рахунку, виходу з системи, продавлення додаткової інформації щодо рахунку, тощо.

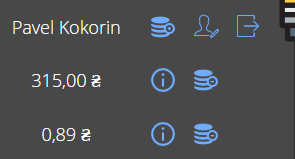


Рисунок 3.15 – Панель користувача

Також відображений поточний баланс по кожній картці банку.

# 4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

Під час вибору технологій для реалізації цього проекту було проаналізовано багато сучасних мов програмування та фреймворків і були обрані найактуальніші з них. Тож було вирішено використати об’єктно-орієнтовану мову програмування C# та фреймворк .NET Framework для серверної сторони та JavaScript[15], HTML 5, CSS для клієнтської сторони.

У якості архітектурного патерну для веб-додатку був обраний MVC. Модель–вигляд–контролер (або Модель–представлення–контролер, англ. Model-view-controller, MVC) — архітектурний шаблон, який використовується під час проектування та розробки програмного забезпечення.

Цей шаблон передбачає поділ системи на три взаємопов'язані частини: модель даних, вигляд (інтерфейс користувача) та модуль керування. Застосовується для відокремлення даних (моделі) від інтерфейсу користувача (вигляду) так, щоб зміни інтерфейсу користувача мінімально впливали на роботу з даними, а зміни в моделі даних могли здійснюватися без змін інтерфейсу користувача.

Мета шаблону — гнучкий дизайн програмного забезпечення, який повинен полегшувати подальші зміни чи розширення програм, а також надавати можливість повторного використання окремих компонентів програми. Крім того використання цього шаблону у великих системах сприяє впорядкованості їхньої структури і робить їх більш зрозумілими за рахунок зменшення складності.

На рисунку 4.1 ви можете бачити ілюстрацію цього патерну.



Рисунок 4.1 – Патерн MVC

У рамках архітектурного шаблону MVC програма поділяється на три окремі, але взаємопов'язані частини з розподілом функцій між компонентами. Модель (Model) відповідає за зберігання даних і забезпечення інтерфейсу до них. Вигляд (View) відповідальний за представлення цих даних користувачеві. Контролер (Controller) керує компонентами, отримує сигнали у вигляді реакції на дії користувача (зміна положення курсору миші, натискання кнопки, ввід даних в текстове поле) і передає дані у модель.

Модель є центральним компонентом шаблону MVC і відображає поведінку застосунку, незалежну від інтерфейсу користувача. Модель стосується прямого керування даними, логікою та правилами застосунку.

Вигляд може являти собою будь-яке представлення інформації, одержуване на виході, наприклад графік чи діаграму. Одночасно можуть співіснувати кілька виглядів (представлень) однієї і тієї ж інформації, наприклад гістограма для керівництва компанії й таблиці для бухгалтерії.

Контролер одержує вхідні дані й перетворює їх на команди для моделі чи вигляду.

Модель інкапсулює ядро даних і основний функціонал їхньої обробки і не залежить від процесу вводу чи виводу даних.

Вигляд може мати декілька взаємопов'язаних областей, наприклад різні таблиці і поля форм, в яких відображаються дані.

У функції контролера входить відстеження визначених подій, що виникають в результаті дій користувача. Контролер дозволяє структурувати код шляхом групування пов'язаних дій в окремий клас. Наприклад у типовому MVC-проекті може бути користувацький контролер, що містить групу методів, пов'язаних з управлінням обліковим записом користувача, таких як реєстрація, авторизація, редагування профілю та зміна пароля.

Зареєстровані події транслюються в різні запити, що спрямовуються компонентам моделі або об'єктам, відповідальним за відображення даних. Відокремлення моделі від вигляду даних дозволяє незалежно використовувати різні компоненти для відображення інформації. Таким чином, якщо користувач через контролер внесе зміни до моделі даних, то інформація, подана одним або декількома візуальними компонентами, буде автоматично відкоригована відповідно до змін, що відбулися.

У даному проекті контролери (Controllers) та відображання (Views) містяться на слою UI. Модель містить у собі декілька компонентів, задач. По-перше це доменна логіка, яка виражена збіркою (слоєм) BLL. По-друге це самі сутності (Entities), які містяться на слою DAL. Усі ці слої описані у попередньому розділі.

Платформу .NET Framework. Microsoft .NET (читається дот-нет) — програмна технологія, запропонована фірмою Microsoft як платформа для створення як звичайних програм, так і веб-застосунків. Багато в чому є продовженням ідей та принципів, покладених в технологію Java. Одною з ідей .NET є сумісність служб, написаних різними мовами. Хоча ця можливість рекламується Microsoft як перевага .NET, платформа Java має таку саму можливість.

Кожна бібліотека (збірка) в .NET має свідчення про свою версію, що дозволяє усунути можливі конфлікти між різними версіями збірок.

Платформа .NET — крос-платформова технологія, в цей час існує реалізація для платформи Microsoft Windows, FreeBSD (від Microsoft) і варіант технології для ОС Linux в проекті Mono (в рамках угоди між Microsoft з Novell), DotGNU .

Захист авторських прав відноситься до створення середовищ виконання (CLR — Common Language Runtime) для програм .NET. Компілятори для .NET випускаються багатьма фірмами для різних мов вільно.

Платформа .NET поділяється на дві основні частини — середовище виконання (по суті віртуальна машина) та інструментарій розробки.

Середовища розробки .NET-програм: Visual Studio .NET (C++, C#, J#), SharpDevelop, Borland Developer Studio (Delphi, C#) тощо. Середовище Eclipse має додаток для розробки .NET-програм. Застосовні програми також можна розроблювати в текстовому редакторі та використовувати консольний компілятор.

Як і технологія Java, середовище розробки .NET створює байт-код, призначений для виконання віртуальною машиною. Вхідна мова цієї машини в .NET називається CIL (Common Intermediate Language), також відома як MSIL (Microsoft Intermediate Language), або просто IL. Застосування байт-кода дозволяє отримати крос-платформовість на рівні скомпільованого проекту (в термінах .NET: збірка), а не на рівні сирцевого тексту, як, наприклад, в С. Перед запуском збірки в середовищі виконання (CLR) байт-код перетворюється вбудованим в середовище JIT-компілятором (just in time, компіляція на льоту) в машинні коди цільового процесора.

Слід зазначити, що один з перших JIT-компіляторів для Java був також розроблений фірмою Microsoft (тепер в Java використовується досконаліша багаторівнева компіляція — Sun HotSpot). Сучасна технологія динамічної компіляції дозволяє досягнути аналогічного рівня швидкодії з традиційними «статичними» компіляторами (наприклад, С++) і питання швидкодії часто залежить від якості того чи іншого компілятора.

У даному проекті всюди використувалась програмна платформа .NET Framework 4.0, 4.5, 4.5.1. Ці версії фреймворку принесли багато корисних функції.

Microsoft анонсувала .NET 4.0 29 вересня 2008 року. Перша бета-версія з'явилася 20 травня 2009 року, разом з бета-версією Visual Studio 2010. Нововведення включають в себе:

– parallel Extensions — PLINQ (Parallel LINQ) і Task Parallel Library, призначені для спрощення програмування для багатопроцесорних і розподілених систем;

– технологія Managed Extensibility Framework (MEF);

– повна підтримка IronPython, IronRuby та F#;

– підтримка підмножин .NET Framework і ASP.NET у варіанті «Server Core»;

– підтримка Code Contracts;

– засоби моделювання Oslo і мова програмування M, призначений для створення предметно-орієнтованих мов і моделей.

Платформа .NET Framework 4.5 версія вийшла 15 серпня 2012 року відсутня офіційна підтримка Windows XP:

– платформа .NET для Metro додатків;

– підмножина в пакеті яка надає можливість для мов C# та VB.NET писати Metro додатки;

– зміни в ядрі;

– консоль тепер повністю підтримує кодування в Unicode UTF-16;

– асинхронні операції;

– в 4.5 версії додано нові асинхронні можливості. В основних мовах C# та VB.NET це здійснено через ключеві слова await та async.

Оновлена версія фреймворку .Net 4.5.1 була презентована разом із Visual Studio 2013 та вийшла в світ 17 жовтня 2013-го року:

Основні нововведення торкались інструментарію налагоджування та покращення продуктивності застосунків:

– налагоджування (debugging) ;

– підтримка Edit & Continue для 64-бітної архітектури;

– підтримка обриву та збою зв'язку в ADO.NET;

– покращення асинхронного налагоджування;

– оптимізація продуктивності застосунків;

– механізм стиснення великих об'єктів у пам'яті по запиту;

– вдосконалення системи побудови програм Just-In-Time для підтримки багатьох платформ;

– новий функціонал ASP.NET App Suspend, який дозволяє автоматично вимикати неактивні сайти та завантажувати їх по запиту.

Дуже цікаве рішення було прийнято при проектуванні та реалізації слою доступу даних. Як відомо на даному сайті було реалізовано реєстрацію користувачів. До цього ж є можливість швидко аутентіфікуватись користувачам, які мають сторінки у соціальних мережах VK, Facebook, LinkedIn. Але для отримання функціоналу сайту, та його використанню, кожного користувача все ж таки потрібно зареєструвати в базі. Тут з’являється велика проблема. Запис у базі даних для користувача має один формат. Наприклад кожен запис має такі колонки як : ім’я, фамілія, ідентифікатор, електронна пошта тощо. Та колонки для аутентіфікації – пароль, логічне поле чи підтверджена пошта. Коли користувач буде аутентіфікуватись через соціальні мережі, немає можливості отримати його пароль. Попри нам все ж таки потрібно зареєструвати користувача, щоб почати роботу з ним, щоб зберігати його дані, його налаштування до наступного разу коли він зайде. Але для аутентіфикації від соціальних мереж можна отримати такі поля як токен доступу до мережі, ідентифікатор користувача, та цілочисленне значення , яке показує коли закінчиться термін токену. Звісно для такої проблеми дуже гарно було б створити ієрархію моделей, де у якості базового класу розмістити користувача з загальними полями, такими як ім’я, фамілія, електронна пошта тощо. А у конкретних класах розмістити його поля для поступу. Наприклад Один наслідник може мати поля пароль та логічне поле електронної пошти, чи підтверджена вона, а інший може мати токен доступу до соціальних мереж. І річ у тім, що Entity Framework може конвертувати це трьома шляхами :

– підхід TPH, Table per Hierarchy табличка на ієрархію;

– підхід TPT, Table per Type табличка на тип;

– підхід TPC, Table per Concrete class, табличка на кожний конкретний клас.

При використанні підходу TPH (Table Per Hierarchy / Таблиця на одну ієрархію класів) для однієї ієрархії класів використовується одна таблиця. Дані базових і похідних класів зберігаються в одну таблицю, а для їх відмінності створюється спеціальний стовпець.

Підхід TPT (Table Per Type / Таблиця на тип) передбачає збереження в загальній таблиці тільки тих властивостей, які загальні для всіх класом-спадкоємців, тобто які визначені в базовому класі. А ті властивості, які відносяться тільки до похідному класу, зберігаються в окремій таблиці.

Підхід TPC (Table Per Concrete Type / Таблиця на кожен окремий тип) передбачає створення для кожної моделі по окремій таблиці. Стовпці в кожній таблиці створюються за всіма властивостями, в тому числі і успадкованим.

Були спроби використати усі ці способи. Якісь з них більш вдалі, вони нормалізують табличку, якісь менше, але при кожному підході не має можливості створити користувача зі одними аутентіфікаційними полями і потім добавити інші, так би мовити добавити у базу декілька конкретних реалізацій. Наприклад, створити користувача з паролем та флагом на пошту, а потім добавити йому аутентіфікаційні поля від соціальних мереж, та надати йому можливість зайти як через аутентіфікаційні форми, так і через соціальні мережі.

Але можна зробити роботу Entity Framework власноруч, та створити моделі для проходження аутентіфікації. На рисунку 4.2 та на рисунку 4.3 ви можете побачити ці моделі.

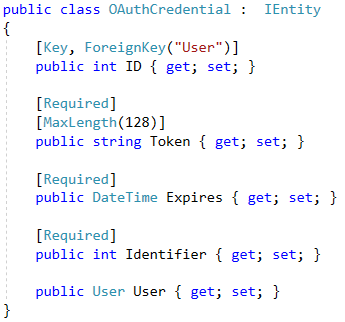


Рисунок 4.2 – Модель аутентіфікації через соціальні мережі

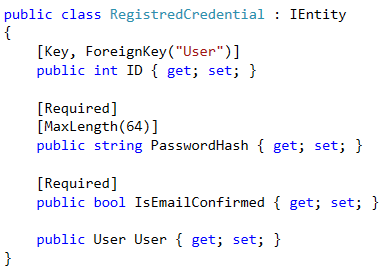


Рисунок 4.3 – Модель аутентіфікації через форми

Модель User може мати ці дві моделі, його код ви можете побачити у розділі про проектування бази даних. Таким чином є можливість створити користувача з обома моделями або якоюсь однією, та надати йому можливість заходити на сайт, як через форми, так і через соціальні мережі. Також слід зазначити, що ці три моделі на слою BLL зводяться до одного класу, який ними управляє – це «UserManager», його інтерфейс реалізацію ви можете побачити на рисунку 4.4.

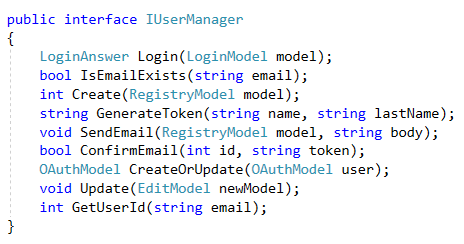


Рисунок 4.4 – Інтерфейс «UserManager»

Він містить у собі методи процесу аутентифікування користувача, перевірки існування електронної пошти у системі, створення користувача через форми, створення користувача через аутентифіковані дані з соціальних мереж, генерація токену для підтвердження електронної пошти, метод відправки повідомлення на електронну пошту, підтвердження електронної пошти оновлення користувача, отримання ідентифікатору користувача.

# 5 ТЕСТУВАННЯ РОЗРОБЛЕННОГО ПРОГРАМНОГО ЗАБЕСПЕЧЕННЯ

Тестування програмного забезпечення (англ. Software Testing) – це процес технічного дослідження, призначений для виявлення інформації про якість продукту відносно контексту, в якому він має використовуватись. Якість не є абсолютною, це суб'єктивне поняття. Тому тестування, як процес своєчасного виявлення помилок та дефектів, не може повністю забезпечити коректність програмного забезпечення. Воно тільки порівнює стан і поведінку продукту зі специфікацією.

Метою тестування розробленого web-додатку є позбавлення помилок та недоліків, а саме: виявлення помилок у функціоналі під час проходження аутентіфікації або взаємодії фахівців між собою, виявлення недоліків інтерфейсу і забезпечення коректного відображення функціоналу на усіх можливих конфігураціях комп’ютера.

Повинен бути здійснений контроль якості web-додатка в цілому, а також його окремих частин, і серверної частини системи.

Так в цілому, повинні бути протестовані:

* інтерфейс головної сторінки;
* інтерфейс aside меню;
* інтерфейс створення нового користувача;
* інтерфейс прикріплення банківської картки;
* функціонал видалення банківської картки;
* функціонал редагування персональних даних;
* зовнішній вид програми при різних конфігураціях операційної системи;
* інтерфейс запитів до API;
* функціонал мобільного додатку:

1) процес аутентіфікації;

2) процес загрузки карток;

3) процес вибору продуктів;

4) процес оплати.

Функціонального тестування – виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації і вимог.

На етапі розробки програмної системи проводилося ручне тестування функціоналу. На таблиці 5.1 можна побачити список функцій, які досліджувались під час тестування і отримані результати.

З результатів таблиці бачимо, що отриманий функціонал розробленої програмної системи задовольняє поставленим вимогам і забезпечує повний функціонал користувача.

Таблиця 5.1 – Функціональні тести

|  |  |  |
| --- | --- | --- |
| Критерій | Статус | Примітки |
| Авторизація та реєстрація в системі | Пройдено | Автоматична авторизація при запуску додатку за допомогою збереження сесійних даних |
| Перегляд головної сторінки | Пройдено | Відображається головна сторінка |
| Збереження користувацьких даних | Пройдено | Дані коректно зберігаються в системі |
| Прикріплення нової картки | Пройдено | Прикріплюється нова картка з поточним рахунком. |
| Редагування користувальницьких даних. | Пройдено | Успішно редагуються та зберігаються усі поля. |
| Відкріплення картки | Пройдено | Карта успішно відкріплюється |
| Відображення та вибір продуктів | Пройдено | Продукти відображаються вдало, працює можливість множинного вибору |

З результатів таблиці бачимо, що отриманий функціонал розробленої програмної системи задовольняє поставленим вимогам і забезпечує повний функціонал користувача.

Конфігураційне тестування – це традиційного тестування продуктивності, в якому замість того щоб тестувати продуктивність системи з точки зору створеного навантаження, тестується ефект впливу на продуктивність змін в конфігурації. Конфігураційне тестування використовується для контролю якості «Мобільності» в розділі «Адаптованості».

Буде проведена перевірка працездатності програмного продукту для різних видів браузерів, перевірка розташування елементів при різних розширеннях екрану/браузеру, досліджена поведінка web-додатку при вмиканні/вимиканні функцій JavaScript.

З таблиці 5.2 бачимо, що коректна робота даного додатку не залежить від обраного використовує мого браузера.

Таблиця 5.2 – Тестування кросс-браузерністі web-додатку

|  |  |
| --- | --- |
| Браузер | Результат |
| Google Chrome | Відображення елементів web-додатку без якихось помилок і повний рабочій функціонал інтерфейсу |
| Mozilla Firefox |
| Opera |
| Internet Explorer |

Для реалізації тестування API була додана спеціальна технологія WEB API. Створюється гарний результат роботи API, рис. 5.1.

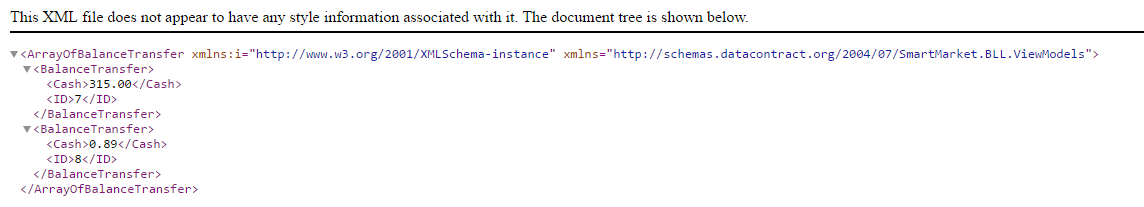


Рисунок 5.1 – Результат роботи API api/balances/1

Було протестовано API системи. За результатами передбачається коректний функціонал серверної частини та інтерфейсу взаємодії клієнту та серверу.

# ВИСНОВКИ

Метою даної роботи була розробка система організації покупок у розумному магазині.

В ході даної роботи проведено докладне дослідження предметної області, обґрунтовано доцільність розробки системи, проведений аналіз існуючих аналогів, описані принципи її роботи та виявлено основні функції.

Реалізація проекту відбувалася за допомогою мов програмування C# та JavaScript, використовувався фреймворк MVC 5 та бібліотек EntityFramework 6, Automapper, NLogger, JQuery, технологій WEB API, XAMARIN. Інтерфейс комунікації між сторонами бек-енду та фронт-енду побудована за допомогою протоколу передачі HTML та XML-даних. Серверний додаток має 3-слойну архітектуру. В якості СУБД виступає MS SQL.

Виконані проектування і реалізація системи і розраховані витрати на її створення. Доведено раціональну обґрунтованість впровадження системи, а також її популярність та необхідність у сучасному світі.

Особливості технології розробки дозволяють при необхідності здійснити перенесення системи на платформи, відмінні від тієї, для якої велася розробка, зі збереженням функціонала.

Розроблена система може використовуватися для професійної роботи фахівців, а саме: інтегрування функціоналу систему та її архітектури під реальні стартами цієї тематики, використовування ідеї з оплати, реалізації взаємодії користувачів системи з магазином тощо.

Потрібно відзначити, що в ході розробки не було реалізовано апаратної частини системи, тому робити повні висновки щодо економічної доцільності неможливо.

У першому виданні додатку впроваджено функціональність для фахівців IT-галузі. З урахуванням гнучкою архітектури, здатністю збільшувати ширину баз даних, система буде здатна підлаштуватись під будь-яку предметну область і надати їй величезну користь. Тобто розглянуте програмне забезпечення може використовуватися людьми за всіма типами професій.

Розроблена система може використовуватися для приклада кодової бази та показу архітектури програмного забезпечення для покращення інших проектів.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Фернандес О. Путь .NET. Подробное руководство по созданию приложений в среде Visual Studio [Текст] / О. Фернандес. – СПб.: Символ-Плюс, 2009. – 768 с.

2. Хэнссон Д.Х. Гибкая разработка веб-приложений в среде Visual Studio 2013 [Текст] / Томас Д., Хэнссон Д.Х. – СПб.: Питер, 2008. – 716 с.

3. Бойд Р. Начало работы с OAuth 2.0 / Раян Бойд. – СПб: Питер, 2012. – 535 с.4. Фитцджеральд М. Изучаем C# на примере приложения [Текст] / М. Фитцджеральд. – СПб.: БХВ-Петербург, 2008. – 336 с.

5. Кнут Д. Э. Искусство программирования. Детальное описание алгоритмов [Текст] / Д. Э. Кнут. – М.: Вильямс, 2002. – 301 с.

6. Фаулер М.б Скотт К. UML Основи. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 192с.

7. Макконнелл С. Совершенный код. Мастер-Класс. Описание подходов разработки [Текст] / С. Макконнел. – М.: Русская Редакция, 2010. – 896 с.

8. Кириллов В.В. Введение в реляционные базы данных / В.В. Кириллов, Г.Ю.Громов. – СПб: БХВ-Петербург, 2009. – 320 с.

9. Голден К. Хороший интерфейс – невидимый інтерфейс, разработка IoT-приложений / К. Голден. – СПб.: Питер,  2016.  – 256 с.

10. Мацумото Ю. Язык программирования C#. Полный сборник [Текст] / Флэнаган Д., Мацумото Ю. – СПб.: БХВ-Петербург, 2008. – 440 с.

11. Фаулер М. Рефакторинг: улучшение существующего кода. Пример применения антипаттернов [Текст] / М. Фаулер. – СПб.: Символ-Плюс, 2003. – 432 с.

12. Глушаков Д.В. Базы данных / С.В. Глушаков, Д.В. Ломотько. – СПб: Символ-Плюс, 2002. – 340 с.

13.Мартин Р. Чистый код. Создание, анализ и рефакторинг программного обеспечения [Текст] / Р. Мартин. – СПб.: Питер, 2010. – 464 с.

14. Приеми объектно-ориентированного проектирования. [Текст] / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб.: Питер, 2001. – 368 с.

15. Челимски Д. Рецепты JavaScript / Д. Челимски, Д. Астелс. – СПб.: Питер, 2010. – 448 с.

# ДОДАТОК А

Слайди презентації

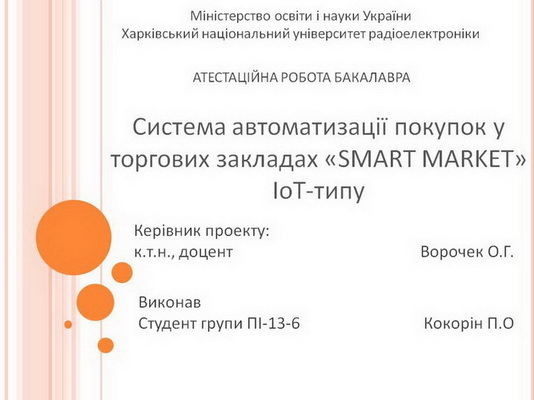


Рисунок А1 – Слайд презентації №1

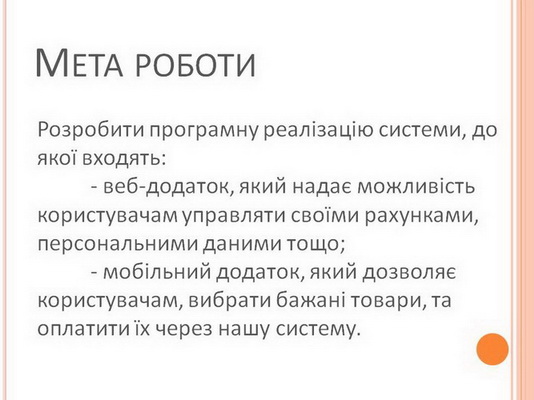


Рисунок А2 – Слайд презентації №2

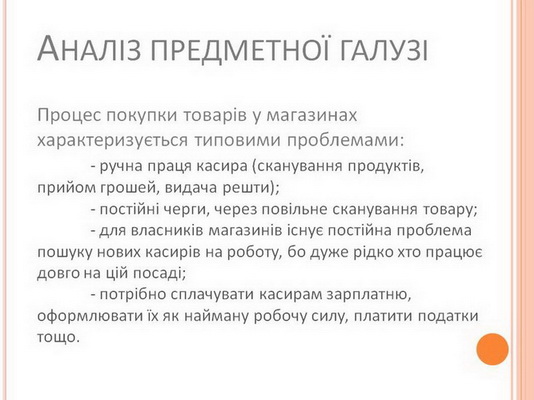


Рисунок А3 – Слайд презентації №3

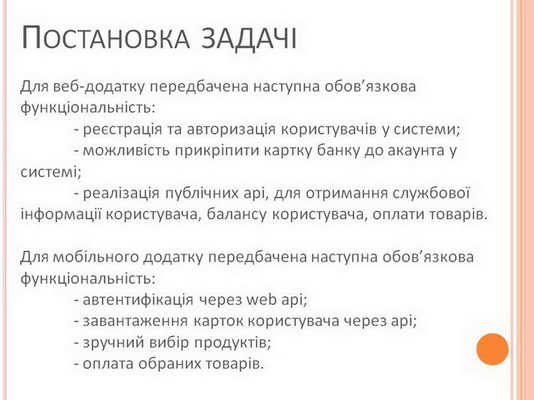


Рисунок А4 – Слайд презентації №4

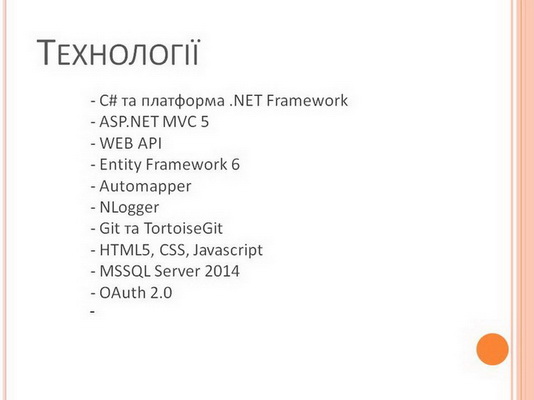


Рисунок А5 – Слайд презентації №5

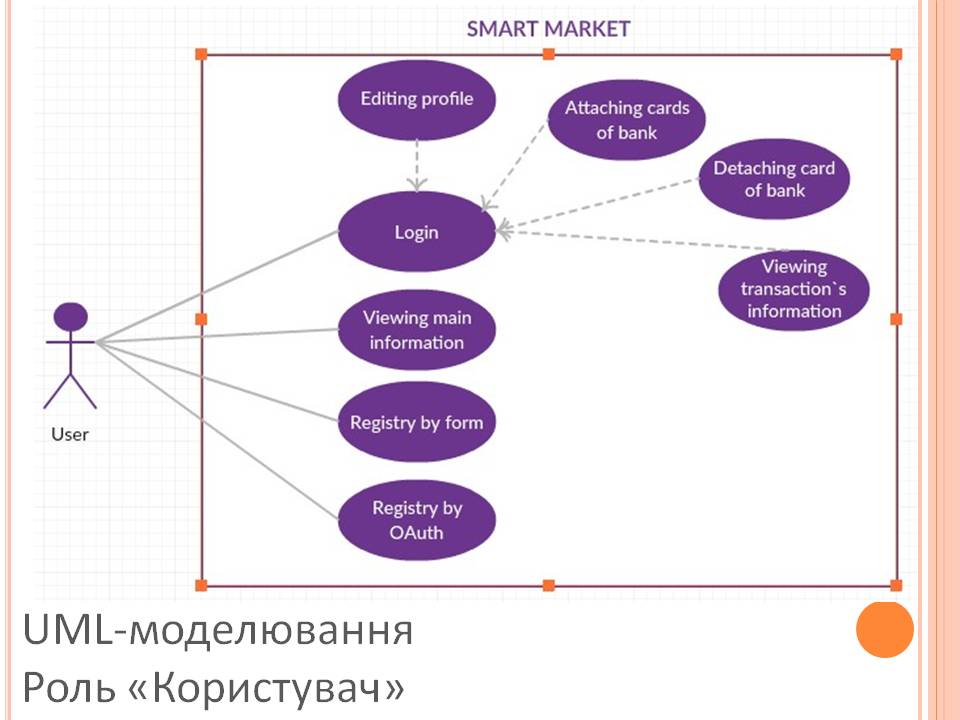


Рисунок А6 – Слайд презентації №6

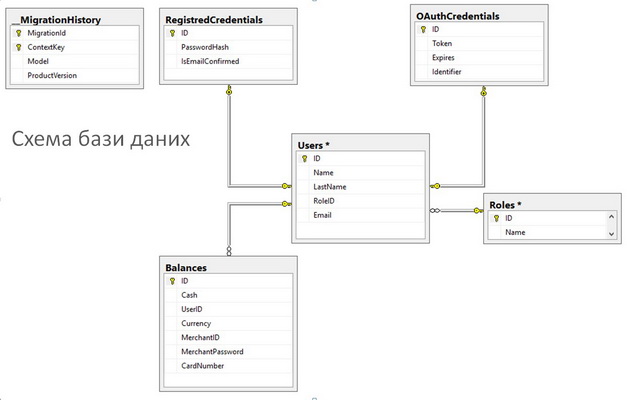


Рисунок А7 – Слайд презентації №7

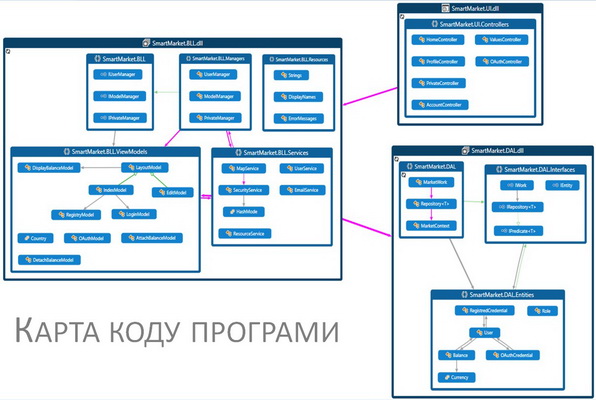
****

Рисунок А8 – Слайд презентації №8

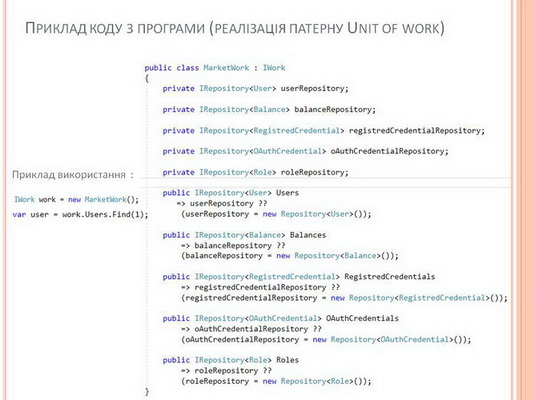
****

Рисунок А9 – Слайд презентації №9

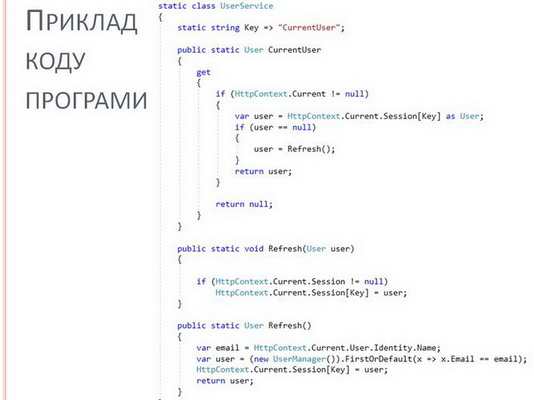
****

Рисунок А10 – Слайд презентації №10

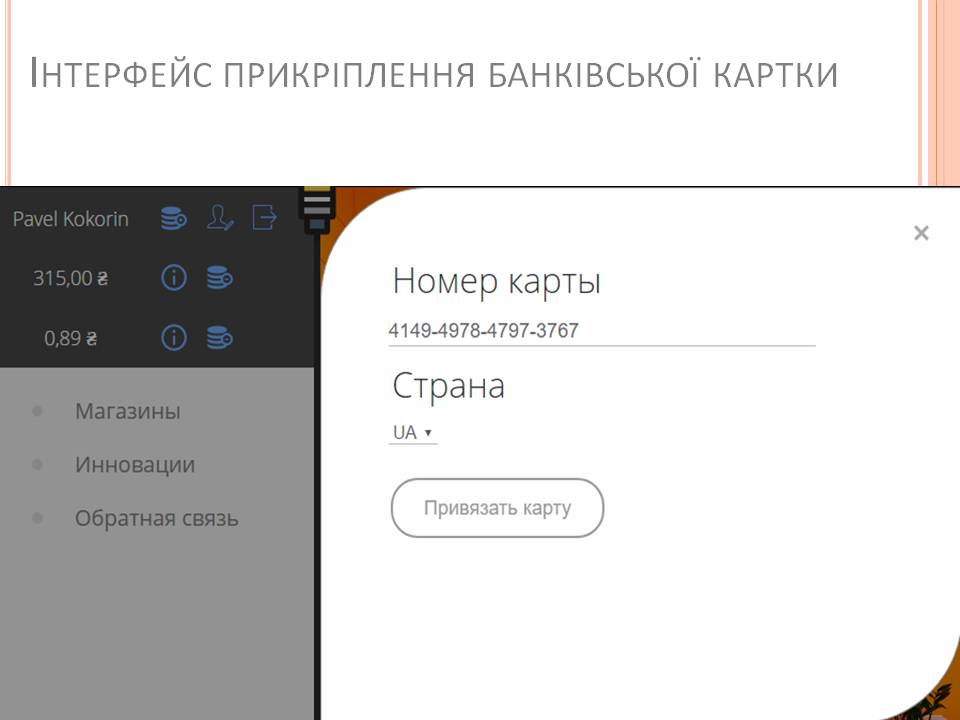


Рисунок А11 – Слайд презентації №11

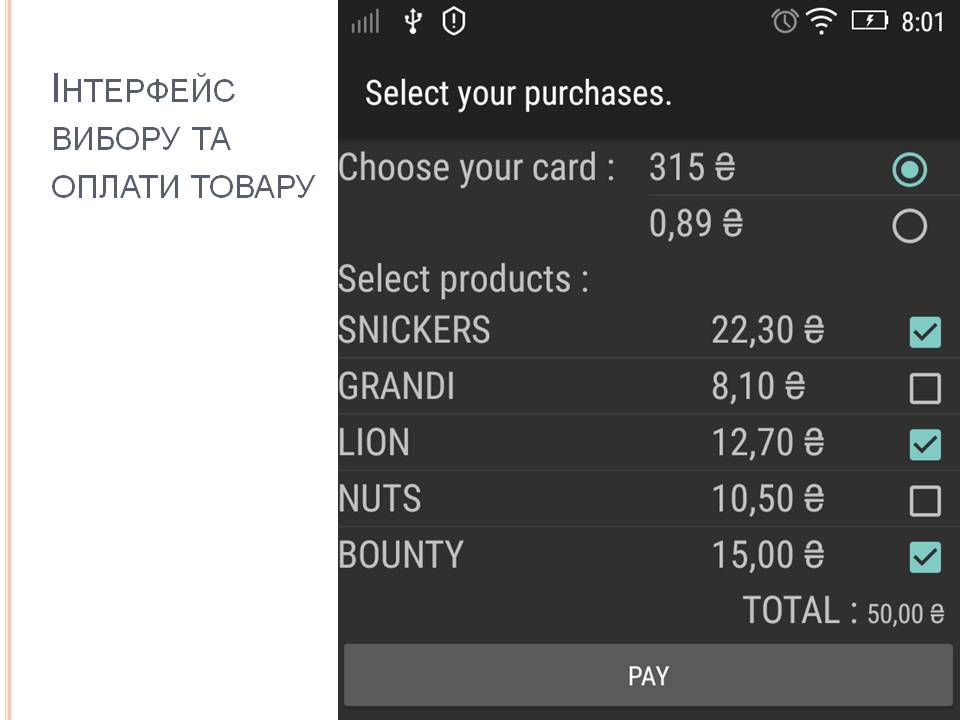


Рисунок А12 – Слайд презентації №12

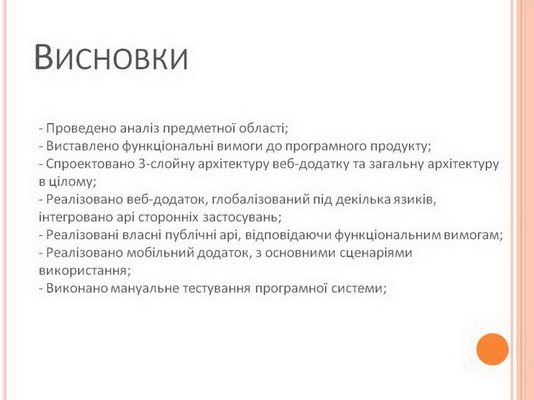
****

Рисунок А13 – Слайд презентації №13

# ДОДАТОК Б

Лістинг коду класу «MapService»

namespace SmartMarket.BLL.Services

{

public static class MapService

{

public static void Config()

{

UserConfigurations();

}

private static void UserConfigurations()

{

Mapper.Initialize(config => {

config.CreateMap<RegistryModel, User>().ConvertUsing<RegistryModelUserConverter>();

config.CreateMap<EditModel, User>().ConvertUsing<EditModelUserConverter>();

config.CreateMap<OAuthModel, User>().ConvertUsing<OAuthModelUserConverter>();

config.CreateMap<User, EditModel>().ConvertUsing<UserEditModelConverter>();

config.CreateMap<User, OAuthModel>().ConvertUsing<UserOAuthModelConverter>();

config.CreateMap<User, IndexModel>().ConvertUsing<UserIndexModelConverter>();

config.CreateMap<Balance, DisplayBalanceModel>().ConvertUsing<BalanceDisplayBalanceModelConverter>();

config.CreateMap<AttachBalanceModel, Balance>().ConvertUsing<AttachBalanceModelBalanceConverter>();

config.CreateMap<Balance, DetachBalanceModel>().ConvertUsing<BalanceDetachBalanceModel>();

});

}

class RegistryModelUserConverter : ITypeConverter<RegistryModel, User>

{

public User Convert(RegistryModel source, User destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new User();

if(source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Email != null)

destination.Email = source.Email;

if (destination.RegistredCredential == null)

destination.RegistredCredential = new RegistredCredential();

if (source.Password != null)

{

var passwordHash = SecurityService.GetPasswordHash(source.Password);

destination.RegistredCredential.PasswordHash = passwordHash;

}

return destination;

}

}

class OAuthModelUserConverter : ITypeConverter<OAuthModel, User>

{

public User Convert(OAuthModel source, User destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new User();

if (source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Email != null)

destination.Email = source.Email;

if (destination.OAuthCredential == null)

destination.OAuthCredential = new OAuthCredential();

if (source.Token != null)

destination.OAuthCredential.Token = source.Token;

if (source.Expires != null)

destination.OAuthCredential.Expires = source.Expires;

if (source.Identifier != 0)

destination.OAuthCredential.Identifier = source.Identifier;

return destination;

}

}

class EditModelUserConverter : ITypeConverter<EditModel, User>

{

public User Convert(EditModel source, User destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new User();

if (source.ID != 0)

destination.ID = source.ID;

if (source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Email != null)

destination.Email = source.Email;

return destination;

}

}

class UserOAuthModelConverter : ITypeConverter<User, OAuthModel>

{

public OAuthModel Convert(User source, OAuthModel destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new OAuthModel();

if (source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Email != null)

destination.Email = source.Email;

if (source.OAuthCredential == null)

return destination;

if (source.OAuthCredential.Token != null)

destination.Token = source.OAuthCredential.Token;

if (source.OAuthCredential.Expires != null)

destination.Expires = source.OAuthCredential.Expires;

if (source.OAuthCredential.Identifier != 0)

destination.Identifier = source.OAuthCredential.Identifier;

return destination;

}

}

class UserEditModelConverter : ITypeConverter<User, EditModel>

{

public EditModel Convert(User source, EditModel destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new EditModel();

if (source.ID != 0)

destination.ID = source.ID;

if (source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Email != null)

destination.Email = source.Email;

var balanceModels = new List<DisplayBalanceModel>();

foreach (var balance in source.Balances)

{

balanceModels.Add(Mapper.Map<Balance, DisplayBalanceModel>(balance));

}

destination.BalanceModels = balanceModels;

return destination;

}

}

class BalanceDisplayBalanceModelConverter : ITypeConverter<Balance, DisplayBalanceModel>

{

public DisplayBalanceModel Convert(Balance source, DisplayBalanceModel destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new DisplayBalanceModel();

if (source.ID != 0)

destination.ID = source.ID;

if (source.Cash != 0)

{

destination.Cash = source.Cash.ToString() + " ₴";

}

destination.Currency = source.Currency.ToString().ToLower();

return destination;

}

}

class UserIndexModelConverter : ITypeConverter<User, IndexModel>

{

public IndexModel Convert(User source, IndexModel destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new IndexModel();

if (source.ID != 0)

destination.UserID = source.ID;

if (source.Name != null)

destination.Name = source.Name;

if (source.LastName != null)

destination.LastName = source.LastName;

if (source.Balances == null)

return destination;

var balanceModels = new List<DisplayBalanceModel>();

foreach (var balance in source.Balances)

{

balanceModels.Add(Mapper.Map<Balance, DisplayBalanceModel>(balance));

}

destination.BalanceModels = balanceModels;

return destination;

}

}

class AttachBalanceModelBalanceConverter : ITypeConverter<AttachBalanceModel, Balance>

{

public Balance Convert(AttachBalanceModel source, Balance destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new Balance();

if (source.UserID != 0)

destination.UserID = source.UserID;

if (source.MerchantID != 0)

destination.MerchantID = source.MerchantID;

if (source.MerchantPassword != null)

destination.MerchantPassword = source.MerchantPassword;

if (source.CardNumber != null)

destination.CardNumber = source.CardNumber;

if (source.Balance != 0)

destination.Cash = source.Balance;

switch (source.Country)

{

case "UA":

destination.Currency = Currency.UA;

break;

case "RU":

destination.Currency = Currency.RU;

break;

case "US":

destination.Currency = Currency.US;

break;

}

return destination;

}

}

class BalanceDetachBalanceModel : ITypeConverter<Balance, DetachBalanceModel>

{

public DetachBalanceModel Convert(Balance source, DetachBalanceModel destination, ResolutionContext context)

{

if (source == null)

return null;

if (destination == null)

destination = new DetachBalanceModel();

if(source.ID != 0)

destination.BalanceID = source.ID;

if (source.CardNumber != null)

{

destination.CardNumber = source.CardNumber;

destination.CardNumber = destination.CardNumber.Insert(4, "-");

destination.CardNumber = destination.CardNumber.Insert(9, "-");

destination.CardNumber = destination.CardNumber.Insert(14, "-");

}

if (source.Cash != 0)

destination.Balance = source.Cash.ToString() + " ₴";

return destination;

}

}

}

}