Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ
Кафедра інформатики факультету інформатики

СТВОРЕННЯ СИСТЕМИ ЗБОРУ АНАЛІТИКИ ПРО СТУДЕНТІВ УНІВЕРСИТЕТУ

Текстова частина до курсової роботи за спеціальністю „Комп'ютерні науки та інформаційні технології" 122

Керівник курсової роботи:
д.т.н., доцент, Глибовець Андрій Миколайович
_____/підпис/
"_____" _____ 2020 р.

Виконав студент:
Козачук Анастасія Олександрівна
_____ /підпис/
"_____" _____ 2020 р.

КИЇВ– 2020 рік

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
канд. фіз.-мат. наук, доцент
_____С. С. Гороховський
(підпис)
„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Козачук Анастасії Олександрівні факультету інформатики 4-го курсу
ТЕМА Створення системи збору аналітики про студентів університету
Вихідні дані:

- Застосунок – система збору аналітики про студентів університету

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
Розділ 1: Використання Spring Boot для реалізації серверу застосування
Розділ 2: Microsoft Azure OAuth2 авторизація в Spring Boot застосуваннях
Розділ 3: ReactJs та Redux як інструменти для розробки UI частини застосунку
Висновки
Список літератури

Дата видачі „___” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Створення системи збору аналітики про студентів університету

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	15.10.2019	
2.	Огляд технічної літератури за темою роботи. Пошук курсів та методичних матеріалів.	05.11.2019	
3.	Встановлення необхідного програмного забезпечення.	10.12.2019	
3.	Створення схеми БД для застосунку. Затвердження схеми з керівником.	15.12.2019	
4.	Написання серверної частини застосунку.	30.12.2019	
5.	Написання першої текстової частини курсової роботи.	15.01.2020	
6.	Ознайомлення з OAuth2 flow. Реєстрація застосунку в Microsoft Azure Active directory. Ознайомлення з роботою Microsoft Azure.	01.02.2020	
7.	Реалізація авторизації в Spring Boot застосуванні.	10.02.2020	
8.	Написання другої частини курсової роботи.	25.02.2020	
9.	Реалізація UI частини застосування.	15.03.2020	
10.	Написання третьої частини курсової роботи.	30.03.2020	
11.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	03.04.2020	
12.	Перегляд змісту роботи з керівником	04.04.2020	
13.	Створення слайдів для доповіді та написання доповіді.	19.04.2020	
14.	Захист роботи	24.04.2020	

Студент Козачук А. О.

Керівник Глибовець А. М.

“_____”_____

Зміст

Анотація

Метою даної курсової роботи є створення застосунку для збору аналітики про студентів університету.

Зміст курсової роботи розкриває поступові етапи створення веб застосунку за допомогою Spring фреймворку та ReactJS бібліотеки.

В першій частині описано розбиття застосунку на рівні і реалізація цих рівнів на основі Spring Boot. В другій частині детально розглянуто внутрішню роботу та налаштування Spring security, а також на основі діаграм описано етапи роботи OAauth2 фреймворку з Microsoft Azure. Третя частина представляє собою опис роботи таких інструментів як ReactJs та Redux для створення UI застосунку.

Вступ

Основною ціллю даної курсової роботи є створення застосунку для зберігання відомостей про студентів університету. Наразі університет надає широкий спектр можливостей і активностей для студентів: різні олімпіади, хакатони, участь в студентських організаціях, організація різних заходів і ще багато іншого. У кожного студента за всі роки навчання в університеті є цілий список того чим він займався окрім навчання. Було би корисним зберігати всю цю інформацію в одному місці. На основі цієї інформації можна оцінити активність студента в університеті. А зберігаючи додатково інформацію про всі прослухані курси та контактну інформацію можна мати централізоване сховище даних для студентів університету.

Також дана система може бути використана для того щоб викладачі могли залишати свої відгуки про студентів. Ці відгуки будуть доступні лише для викладачів і дозволять глибше розуміти мотивацію студента до навчання.

Для реалізації даного застосунку було обрано Spring Boot фреймворк для реалізації серверної частини та ReactJs для реалізації UI частини, а також авторизація на основі Microsoft Azure OAuth2. Ці технології є часто вживаними в великих ентерпрайз проектах і будуть корисними для детального вивчення.

Цілями даної курсової роботи є:

1) Створення архітектури застосунку на основи Spring Boot фреймворку.
2) Вивчення роботи OAuth2.
3) Вивчення роботи платформи Microsoft Azure для реєстрації власного застосунку та користувачів.
4) Огляд роботи ReactJs та Redux бібліотеки для розробки UI застосунку.
5) Створення серверної та клієнтської частини застосунку.
6) Проектування і створення бази даних.

РОЗДІЛ 1: Використання Spring Boot для реалізації серверу застосування

1.1 Коротко про Spring Boot

Spring широко застосовується для створення масштабованих застосувань. Для веб-застосунків Spring пропонує Spring MVC, який є широко застосовуваним модулем Spring фреймворку, який використовується для створення масштабованих веб-застосунків. Але основним недоліком проектів на основі цього фреймворку є їх конфігурація, яка може займати багато часу та бути занадто складною для початківців. Вирішенням цієї проблеми став Spring Boot.

Spring Boot (Рисунок 1.1) – це open-source фреймворк, який розробляється і підримується компанією Pivotal. Spring Boot побудований на верхівці Spring фреймворку і містить в собі всі його можливості. З його допомогою розробники можуть швидко розпочати роботу не витрачаючи багато часу на налаштування конфігурацій для запуску свого веб-застосунку.

Рисунок 1.1 – Структура Spring Boot [1]

Отже Spring Boot вирішує наступні проблеми:

1) Робить легшим процес початку роботи зі Spring framework.
2) Мінімізує кількість створення конфігурацій застосунку вручну.
3) Здійснює автоконфігурацію на основі файлів property і JAR classpath.
4) Допомагає вирішити конфлікти з залежностями для Maven чи Gradle.

5) Пропонує вбудований Tomcat сервер.
6) Пропонує легке створення та підтримку для REST end points.
7) Процес деплою застосунку дуже простий. War або jar файл може бути легко задеплоєний на tomcat сервер.
8) Мікросервісна архітектура.

Точкою входу в програму є клас відмічений анотацією @SpringBootApplication.

1.2 Архітектура Spring Boot застосунку

Spring Boot має 4 основних рівні (Рисунок 1.2, Рисунок 1.3):

1) Presentation Layer – обробляє http запити, переводить JSON в Java об’єкти і автентифікує запит, передаючи його до бізнес рівню.
2) Business Layer – обробляє всю бізнес логіку застосунку. Він складається з класів сервісів та використовує методи надані рівнем доступу до даних.Також цей рівень виконує валідацію та авторизацію.
3) Persistence Layer – містить всю логіку роботи з БД, а також конвертує Java об’єкти з рядків БД та назад в ці рядки БД.
4) Database Layer – На цьому рівні виконуються всі CRUD операції.

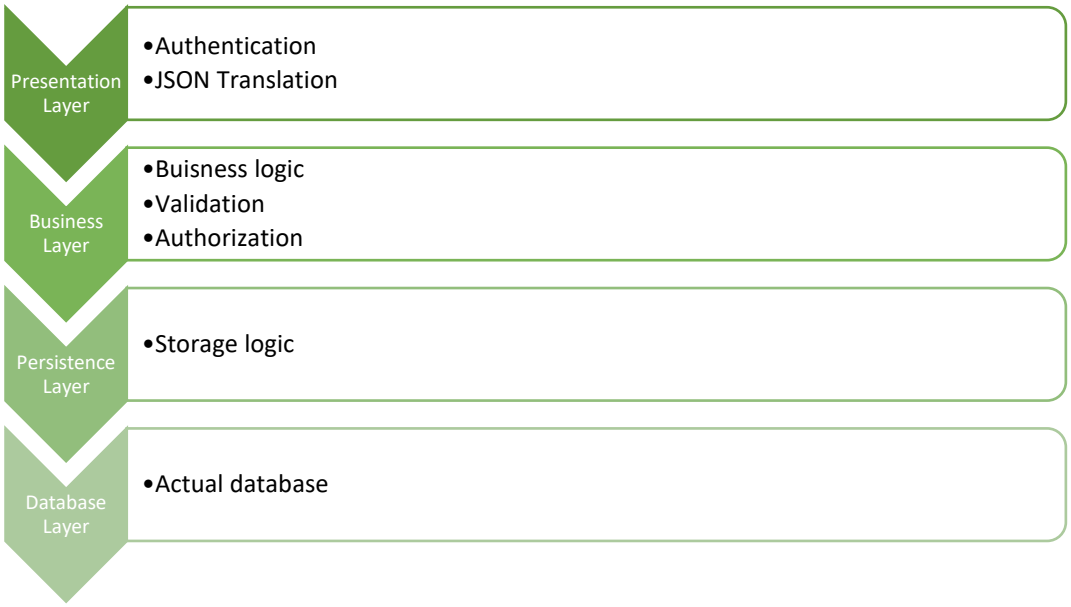

Рисунок 1.2 – Рівні архітектури Spring Boot

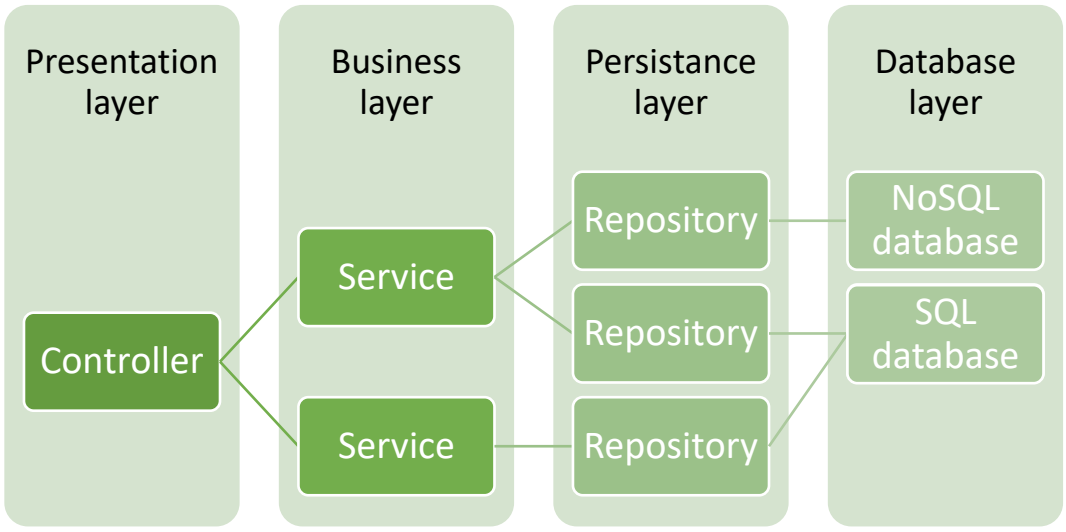

Рисунок 1.3 – Рівні архітектури Spring Boot застосунку

Spring boot flow (Рисунок 1.4):

1) Клієнт виконує http запит.
2) Запит йде до контроллера, контроллер знаходить метод, який може обробити даний запит і обробляє його. Після цього він викликає відповідний сервіс, якщо це необхідно.

3) На рівні сервісів виконується вся необхідна бізнес логіка. Цей рівень виконує операції над даними, які відображені в JPA за допомогою класів моделей.
4) Повертається відповідь клієнту у вигляді View або JSON даних.

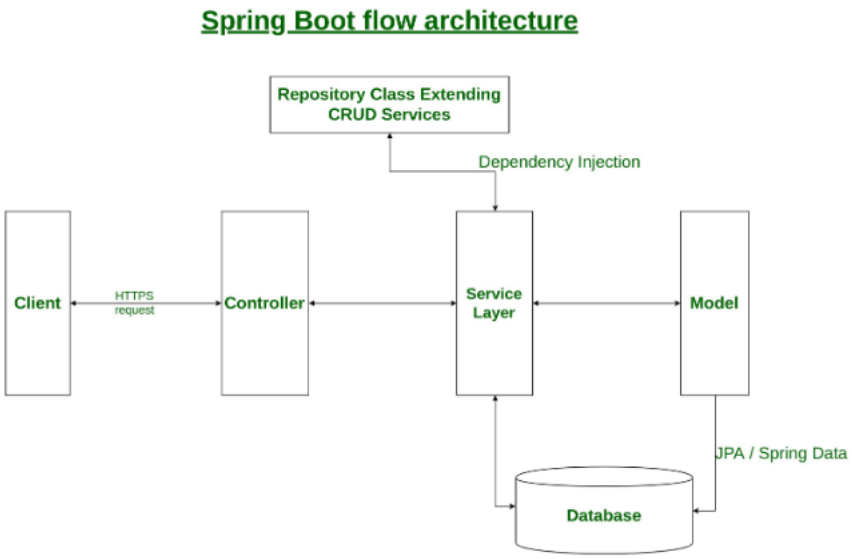

Рисунок 1.4 – Spring boot flow [2]

Дані зберігаються і передаються між рівнями та клієнту у вигляді DTO та моделей. Модель представляє формальні базові конструкції даних. DTO (data transfer object) створений для того, щоб транспортувати дані від сервера до клієнта і навпаки.

1.3 Проектування застосунку

Перед тим як починати створювати будь-який застосунок необхідно визначити вимоги до цього застосунку та функції, які він повинен буде виконувати, визначити групи користувачів та спроектувати базу даних застосунку на основі описаних умов.

Вимоги до системи збору аналітики про студентів:

- Збереження контактної інформації про студента, контактних даних батьків.
- Збереження прослуханих студентом курсів.
- Можливість створення викладачами відгуків про студента.
- Збереження інформації про участь в олімпіадах, хакатонах.
- Збереження інформації про участь в студентських організаціях.
- Рік вступу\випуску студента. Програма на якій навчається студент.
- Збереження інформації про інші активності.
- Авторизація через Office365. Роль для користувача буде повертатись з окремого серверу.
- Методисти університету мають мати можливість додавати відомості про студента.

Групи користувачів системи збору аналітики про студентів:

1) Студенти – можуть заповнювати та переглядати інформацію про себе.
2) Викладачі – можуть заповнити інформацію про себе. Переглядати інформацію про студентів і залишати відгуки про студентів.
3) Методисти – мають можливість переглядати дані по студентам, додавати дані про студентів, додавати до системи курси, студентські організації, викладачів.
4) Керівники студентських організацій – мають можливість переглядати учасників організації, створювати опис до своєї організації.

База даних системи збору аналітики про студентів (Рисунок 1.5):

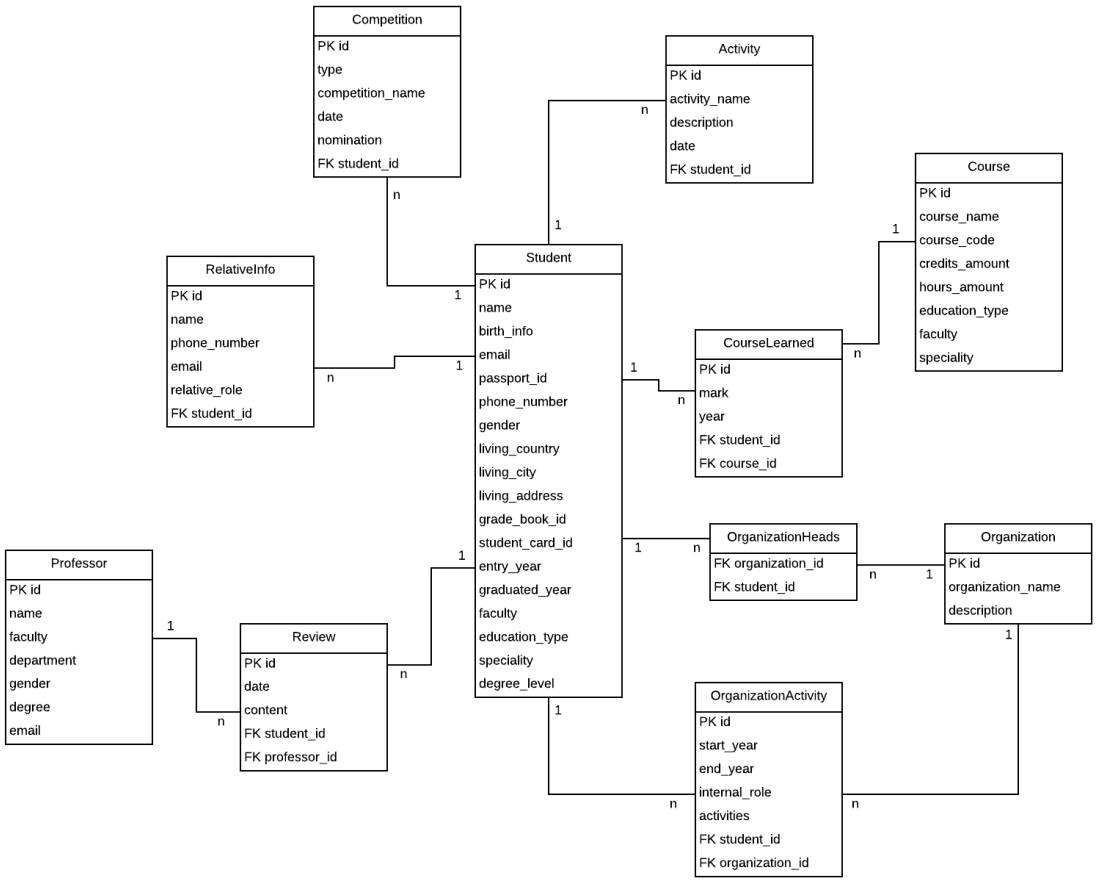

Рисунок 1.5 – Схема бази даних

Відповідно до схеми було спроектовано наступні таблиці:

1) Student – призначена для зберігання докладних даних про студента, його контактну інформацію, місце проживання, факультет та спеціальність на якій навчається.
2) Activity –зберігає інформацію про будь-яку діяльність в університеті (допомога в організації днів факультетів, різних подій тощо).
3) Competition – зберігає інформацію про участь студента в різних змаганнях, олімпіадах, хакатонах.
4) RelativeInfo – зберігає контактну інформацію родичів студента.
5) CourseLearned – зберігає інформацію про вивчені студентом курси.
6) Course – зберігає інформацію про наявні курси в університеті.
7) OrganizationActivity – зберігає інформацію про діяльність студента в тій чи іншій організації.

8) Organization – зберігає інформацію про студентські організації.
9) OrganizationHeads – зберігає інформацію про керівників студентських організацій.
10) Review – зберігає інформацію про відгуки на студентів від викладачів.
11) Professor – зберігає інформацію про викладачів КМА.

1.4 Створення моделей

Приклад створення моделі (Рисунок 1.6) на основі таблиці бази даних Student (для компактності представлення в моделі показано не всі поля, які наявні в таблиці бази даних).

Анотація @Entity вказує що клас і сутністю і є відображенням таблиці Student бази даних. Кожне поле класу позначене анотацією @Column представляє рядок в базі даних. Також в значеннях анотації @Column можна прописувати валідацію для кожного з полів. Поле позначене анотацією @Id є відображенням PrimaryKey в базі даних.

```
@Entity
public class Student extends GettableById<Integer> {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(nullable = false)
   private Integer id;

   @Column(nullable = false, length = 70)
   private String name;

   @Column
   private LocalDate birthInfo;

   @Column
   @Enumerated(value = EnumType.STRING)
   private Gender gender;

   @OneToMany(mappedBy = Activity.Fields.student)
   private List<Activity> activities;

   @ManyToMany
   @JoinTable(name = "organization_head",
           joinColumns = @JoinColumn(name = "student_id", referencedColumnName = "id"),
           inverseJoinColumns = @JoinColumn(name = "organization_id", referencedColumnName =
"id"))
   private List<Organization> headFor;
}
```

Рисунок 1.6 – Реалізація моделі

Позначення поля за допомогою анотації @GeneratedValue вказує, що значення буде автоматично згенеровано для цього поля. Використовується ця анотація зазвичай для primary key fields, як і в даному випадку. Можливі значення стратегій для анотації:

1) GenerationType.AUTO – під час збереження даних до бази даних persistence provider використовує global number generator щоб згенерувати первинний ключ для кожної нової entity. Згенероване значення є унікальним на рівні бази даних і ніколи не буде використано повторно.
2) GenerationType.IDENTITY – є схожою до AUTO стратегії. Вона також генерує автоматичне значення для первинного ключа під час збереження entity до бази даних. Але це значення є унікальним лише в межах певного типу (таблиці).
3) GenerationType.SEQUENCE – вказує, що persistence provider повинен присвоювати первинні ключі використовуючи database sequence.
4) GenerationType.TABLE - вказує, що persistence provider повинен присвоювати первинні ключі використовуючи таблицю бази даних для того щоб забезпечити унікальність.

Анотація @Enumerated дає можливість використовувати тип java enum для представлення типу даних для колонки в базі даних. JPA підтримує можливість перетворення з enum типу в тип вказаний в анотації та навпаки. Можливі значення типів:

1) EnumType.STRING – в базу даних записується значення методу name() enum типу.
2) EnumType.ORDINAL - в базу даних записується значення методу ordinal() enum типу. Цей тип використовувати ризиковано, тому що визначений на початку порядок констант може змінитись і тоді

порушиться відповідність значень між тими що записані в базі даних і тими, що зберігаються в enum типі.

Відповідно до схеми бази даних ми повинні поєднати таблицю Student з іншими таблицями. Це можна зробити за допомогою наступних анотацій:

1) @OneToOne – відповідає зв’язку 1-1 в таблиці бази даних.
2) @OneToMany, @ManyToOne – відповідають зв’язкам 1-n, n-1 в таблиці бази даних.
3) @ManyToMany – відповідає зв’язку n-m в базі даних.

1.5 Життєвий цикл entity

Кожна створена entity проходить свій життєвий цикл (Рисунок 1.7).

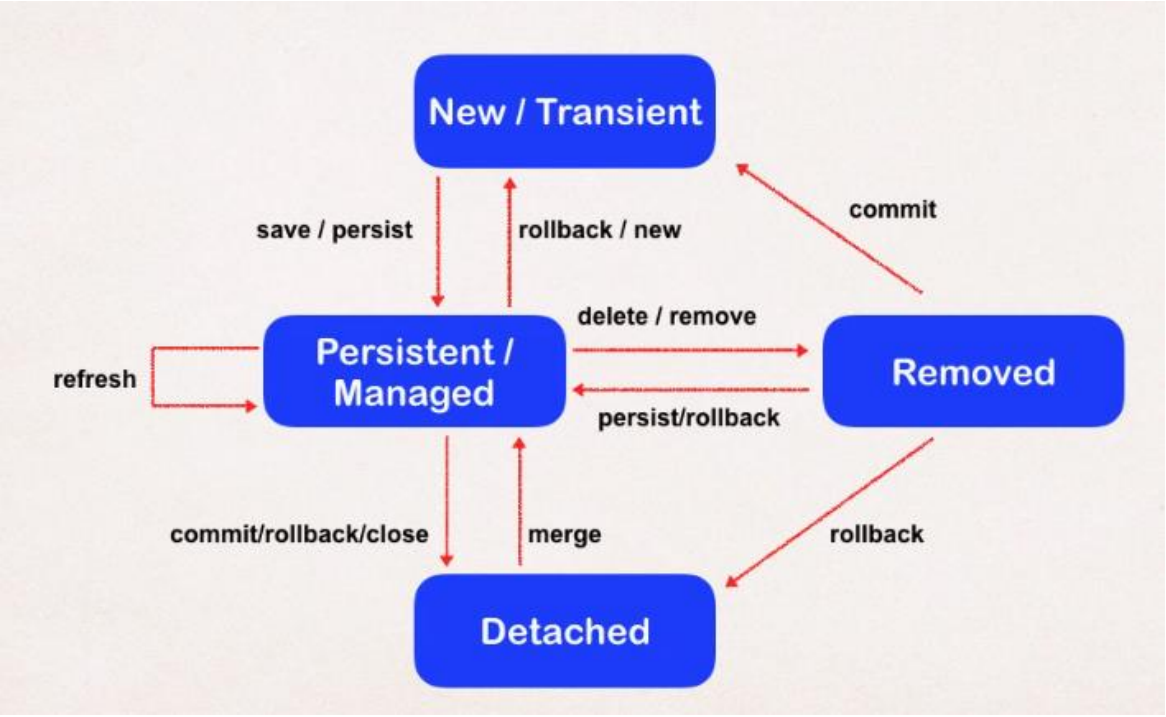

Рисунок 1.7 – Схема життєвого циклу моделі [1]

Transient

Коли об’єкт знаходиться в transient стані, persistant manager нічого не знає про цей об’єкт. Тобто це звичайний POJO об’єкт.

Persistant/Managed

Коли об’єкт перебуває в persistant стані, то persistant manager знає про його існування і тримає значення з бази даних синхронізованими зі значенням полів цього об’єкта. Є декілька способів перевести об’єкт в цей стан:

1) Завантаження об’єкту з бази даних за допомогою JPA.
2) Збереження об’єкта в базу даних за допомогою JPA.

Тільки тоді коли ви оновлюєте значення об’єкта в persistant стані, вони будуть оновлені в базі даних.

Об’єкт може знаходитись в цьому стані тільки тоді коли він знаходиться в transactional context. Для того щоб створити цей контекст ми можемо використати анотацію @Transactional на рівні методу, або класу.

Detached

Після того як всі операції в transactional context закінчують свою роботу, він закривається. Всі об’єкти які знаходились в persistant state і залишили transactional context переходять в detached state. Об’єкт, який знаходиться в detached state може бути повернутий в persistant state в новому transactional context.

Цей стан відрізняється від transient в тому, що JPA все ще зберігає запис про цей об’єкт в своїй пам’яті. Тому коли ви захочете оновити значення в базі даних відповідно до нових значень в цьому об’єкті, то JPA буде намагатись оновити тільки ті поля, які змінились з минулого разу коли цей об’єкт був в persistant стані.

Removed

Останній стан в життєвому циклі об’єкта це removed. Об’єкт переходить в цей стан коли ви позначаєте об’єкт, який знаходиться в persistant стані як видалений. Після того, як об’єкт перейшов до цього стану, його більше не можна використовувати.

1.6 JPA cascade types

Entities які залежні від інших entity, часто мають залежність від операцій які виконуються над entity від яких вони залежні. Наприклад, певна позиція є частиною замовлення. Якщо замовлення видаляють, то цю позицію також слід видалити. Такий взаємозв’язок називається cascade delete.

javax.persistence.CascadeType визначає каскадні операції, які можуть бути застосовані до child елементів об’єкту.

Cascade Operations для Entities:

1) ALL - усі каскадні операції будуть застосовані до пов'язаного об'єкта з батьківським об'єктом. ALL еквівалентно = {DETACH, MERGE, PERSIST, REFRESH, REMOVE}
2) DETACH – якщо батьківський об’єкт знаходиться в detached стані, то пов’язані з ним об’єкти також переходять у цей стан.
3) MERGE - якщо батьківський об’єкт повернутий до persistent стану, то пов’язані з ним об’єкти також переходять у цей стан.
4) PERSIST - якщо батьківський об’єкт зберігається в persistent стані, то пов’язані з ним об’єкти також переходять у цей стан.
5) REFRESH - якщо батьківський об’єкт буде оновлений в поточному persistent стані, то пов’язані з ним об’єкти також будуть оновлені у цьому стані.
6) REMOVE - якщо батьківський об’єкт буде видалений в поточному persistent стані, то пов’язані з ним об’єкти також будуть видалені.

1.7 Створення Repository для моделей

Для того щоб створити репозиторій (Рисунок 1.8) для моделі неохідно створити інтерфейс і позначити його анотацією @Repository. Цей інтерфейс повинен наслідувати один з існуючих репозиторіїв:

1) CrudRepository – інтерфейс для загальних CRUD операцій на сховищі конкретного типу.
2) PagingAndSortingRepository – розширення CrudRepository для надання додаткових методів отримання результатів, таких як розбиття результатів на сторінки чи сортування.
3) JpaRepository – розширення PagingAndSortingRepository, що забезпечує методи пов'язані з JPA.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {
    Optional<Student> findByEmail(String email);
    boolean existsStudentByEmail(String email);
}
```

Рисунок 1.8 – Реалізація репозиторію

Є декілька способів створення методів запитів в репозиторії, які використовуються найчастіше:

1) Створення запитів по імені методів. Використовуючи визначені правила назви методів Spring JPA може автоматично генерувати запити до бази даних.
2) @NamedQuery анотація. Запит визначається над entity, а потім оголошується в самому репозиторії (Рисунок 1.9).

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
        query = "select u from User u where u.emailAddress = ?1")
public class User {

}

public interface UserRepository extends JpaRepository<User, Long> {

   List<User> findByLastname(String lastname);

   User findByEmailAddress(String emailAddress);
}
```

Рисунок 1.9 – Приклад використання @NamedQuery

3) @Query анотація. Використовується над методами репозиторію і визначає SQL запит, який має виконатись при виклику даного методу (Рисунок 1.10).

```
public interface UserRepository extends JpaRepository<User, Long> {

   @Query("select u from User u where u.emailAddress = ?1")
   User findByEmailAddress(String emailAddress);
}
```

Рисунок 1.10 – Приклад використання @Query

1.8 Створення сервісів

Для того, щоб створити сервіс (Рисунок 1.11) необхідно створити клас і позначити анотацією @Service. Сервіси призначені для того щоб виконувати всю основну бізнес логіку застосунку. Зазвичай сервіси використовують методи з репозиторіїв для виконання операцій зв’язаних з базою даних.

```
@Service
public class StudentServiceImpl extends StudentService {

   @Autowired
   private StudentRepository studentRepository;

   @Override
   public Student getByEmail(final String email) {
      return studentRepository.findByEmail(email).orElseThrow(() -> new NoSuchEntityException(
            messageBundleService.getMessage("student.with.email.not.exists", email)));
   }
}
```

Рисунок 1.11 – Реалізація сервісу

1.9 Створення контролерів

Для того щоб створити REST контролер (Рисунок 1.12) необхідно створити клас і позначити його анотацією @RestController та @RequestMapping, в параметрах якої вказується шлях до контролера. В самому класі визначаються методи ендпоінти, кожний з яких позначається однією з анотацій @GetMapping, @PostMapping, @PutMapping,

@DeleteMapping. Ці методи приймають параметри, або DTO об’єкти з даними. Повертають методи дані у вигляді DTO об’єктів.

```
@RestController
@RequestMapping("/students")
public class StudentController {

   @Autowired
   private StudentFacade studentFacade;

   @GetMapping("/{email}/info")
   public StudentDto getStudentByEmail(
         @PathVariable
         final String email) {
      return studentFacade.getStudentByEmail(email);
   }
}
```

Рисунок 1.12 – Реалізація контроллеру

Для того щоб правильно обробити помилки, які можуть виникати в процесі виконання операцій на будь-якому рівні можна створити спеціальний клас, який буде їх оброблювати.

Клас, що позначений анотацією @ControllerAdvice стає глобальним перехоплювачем виключних ситуацій з усіх контролерів застосунку.

У цьому класі оголошені обробники виключень, по одному на метод, оскільки кожне виключення має свій власний шлях його обробки.

Такий підхід дозволяє зібрати всі обробники виключень в одному місці, що робить код чистішим і зрозумілішим для читання та розробки (Рисунок 1.13).

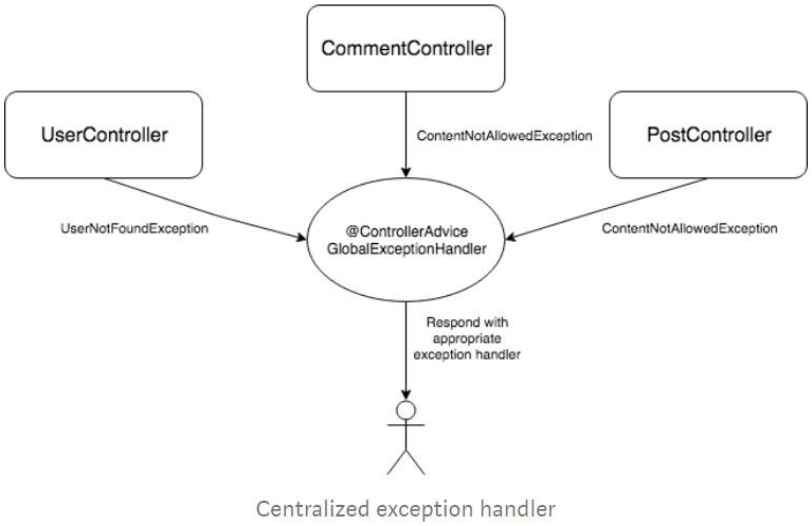

Рисунок 1.13 – Схема роботи @ControllerAdvice [3]

Для того щоб створити метод обробник виключень необхідно позначити його анотацією @ExceptionHandler. В методі описується обробка виключення, а також зазначається який статус помилки повернути користувачу разом з текстом про помилку (Рисунок 1.14).

Метод обробник позначений анотацією @ExceptionHandler може мати дуже гнучку сигнатуру і приймати аргументи різних типів, наприклад exception argument, request and/or response objects, session object, locale object, model object etc. Так само метод може мати різні типи для повернення результату, наприклад ModelAndView object, Model object, View object, String, Response entity etc.

```
@ControllerAdvice
public class GeneralExceptionHandler {


   @ExceptionHandler
   @ResponseStatus(HttpStatus.BAD_REQUEST)
   @ResponseBody
   public ServerErrorResponse handleRuntimeException(final Throwable th) {
      return new ServerErrorResponse(ErrorResponseType.UNEXPECTED, th.getMessage(),
th.getClass().getSimpleName());
   }
}
```

Рисунок 1.14 – Реалізація обробника помилок

1.10 Фільтрація даних за допомогою специфікацій

Spring Data надає інтерфейс Specfication, який може бути використаний для того щоб створювати і викликати JPA criteria queries. Імплементуючи даний інтерфейс (Рисунок 1.15) необхідно перевизначити метод toPredicate(), де створити необхідний предикат. Інші методи в інтерфейсі визначені за замовчуванням і слугують допоміжними методами при створенні специфікацій:

1) not(Specification<T> spec)
2) where(Specification<T> spec)
3) and(Specification<T> spec)
4) or(Specification<T> spec)

```
public static Specification<Employee> getEmployeesByNameSpec(String name) {
   return new Specification<Employee>() {
      @Override
      public Predicate toPredicate(Root<Employee> root,
                                   CriteriaQuery<?> query,
                                   CriteriaBuilder criteriaBuilder) {
         Predicate equalPredicate = criteriaBuilder.equal(root.get(Employee_.name), name);
         return equalPredicate;
      }
   };
}
```

Рисунок 1.15 – Реалізація специфікації

Після того, як предикат визначений в специфікації його можна застосовувати в методах репозиторія (Рисунок 1.16). Для цього репозиторій повинен наслідувати JpaSpecificationExectutor інтерфейс. Створена специфікація передається в середину метода і всю іншу роботу Spring виконує самостійно.

```
@Repository
public interface CompetitionRepository extends IBaseRepository<Competition, Integer>,
      JpaSpecificationExecutor<Competition> {

   List<Competition> findAll(Specification<Competition> specification);
}
```

Рисунок 1.16 – Використання специфікації в репозиторії

Також існує спосіб автоматичного створення специфікацій за допомогою Specification Argument Resolver. Це бібліотека, яка дозволяє сконфігурувати на рівні контролеру яку специфікацію нам треба отримати і вона на основі цих конфігурацій сама створить необхідну специфікацію.

В прикладі показаний метод контролера (Рисунок 1.17), якому напряму передається вже згенерована специфікація. Специфікація генерується на основі визначеної конфігурації. Декілька з існуючих анотацій:

1) @Or – анотація означає, що необхідна модель повинна відповідати хоча б одній з вказаних специфікацій.
2) @And - анотація означає, що необхідна модель повинна відповідати усім вказаним специфікаціям.
3) @Spec – описує специфікацію для певного параметру (приклади значень для spec: Equal, Like, GreaterThan, Null, NotNull etc.).

```
@GetMapping
public List<CompetitionDto> findCustomers(
      @And({
                  @Spec(path = "type", spec = Equal.class),
                  @Spec(path = "competitionName", spec = Like.class),
                  @Spec(path = "nomination", spec = Like.class),
                  @Spec(path = "date", spec = Equal.class)
            })
            final Specification<Competition> competitionSpec) {

   return competitionFacade.getCompetitionsBySpec(competitionSpec);
}
```

Рисунок 1.17 – Специфікація як параметр контролеру

Для того щоб Spring міг згенерувати специфікацію, ми повинні вказати в його веб конфігурації додатковий argument resolver (Рисунок 1.18).

```
@Override
public void addArgumentResolvers(final List<HandlerMethodArgumentResolver> argumentResolvers) {
    argumentResolvers.add(new SpecificationArgumentResolver());
}
```

Рисунок 1.18 – оголошення argument resolver

Запити для такого типу ендпоентів створюються у наступному вигляді (1.1):

http://localhost:8080/competitions?type=HACKATHON&competitionName=Olymp (1.1)

Відповідь від сервера на запит (Рисунок 1.19):


```
{
    "id": 2,
    "type": "HACKATHON",
    "competitionName": "Olymp2",
    "date": "2020-04-09",
    "nomination": "Учасник"
}
```


Рисунок 1.19 – Відповідь сервера на запит з специфікацією

РОЗДІЛ 2: Microsoft Azure OAuth2 авторизація в Spring Boot застосуваннях

2.1 Spring security concepts

Автентифікація

Автентифікація це валідація ваших даних користувача таких як ім’я користувача\пароль для того, щоб підтвердити вашу особу. Цей процес дозволяє визначити хто ви є, перед тим як давати вам доступ до будь-якого ресурсу в системі. Є декілька видів автентифікацій:

1) Knowledge based authentication – автентифікація за допомогою логіну\паролю про який знаєте і ви, і система куди ви хочете потрапити.
2) Possession based authentication – автентифікація за допомогою пристрою, яким володієте лише ви (наприклад телефон, або персональна карта з ключем, тощо).

Комбінація цих двох видів автентифікацій дає можливість створювати мультифакторну автентифікацію, коли використовується декілька рівнів вищезазначених способів для аавтентифікації.

Автентифікація відповідає на запитання: хто цей користувач?

Авторизація

Авторизація відбувається після успішної автентифікації вашої особи. Цей процес визначає ваші права в системі: до яких ресурсів ви маєте доступ і які ресурси можете змінювати.

Авторизація відповідає за запитання: чи цей користувач має дозвіл на виконання даної операції?

Principal

Це користувач, який вже залогінений до системи. Один користувач може мати багато principals в системі. Наприклад користувач залогінився під різними сесіями або ж має декілька аккаунтів в системі.

Granted authority

Це список прав, які користувач має в системі.

Role

Це група прав, які зазвичай згруповані разом.

Наприклад застосунок зі збору аналітики для студентів має наступні ролі та права:

1. STUDENT_ROLE
    - Перегляд інформації про себе
    - Зберігання інформації про себе (свої контактні дані, інформація про будь яку активність тощо)
2. PROFESSOR_ROLE
    - Перегляд інформації про себе
    - Зберігання інформації про себе
    - Перегляд інформації по студентам
    - Створення відгуків по студентам
    - Перегляд відгуків про студентів
3. ADMIN_STAFF_ROLE
    - Перегляд інформації про студентів
    - Заповнення інформації про студентів
    - Перегляд інформації про викладачів
    - Перегляд відгуків про студентів
    - Додавання нових курсів до системи
    - Додавання нових студентських організацій до системи

- Верифікувати заповнену студентами інформацію

4. ORG_HEAD_ROLE
    - Додавати інформацію про організації
    - Переглядати учасників своєї організації
    - Верифікувати інформацію заповнену учасниками своєї організації у них в профілі

2.2Як працює Spring security

Коли користувач робить запит до веб-серверу, контейнер сервлетів визначає чи є необхідний сервлет, який може обробити запит і якщо він є перенаправляє цей запит до необхідного сервлету. Контейнер сервлетів також має фільтри, які перехоплюють кожний запит, що дає можливість виконувати певні операції над цими запитами ще до того як вони будуть передані до самого контейнера. Spring security використовує фільтри для того щоб перехоплювати кожний запит і дозволяти чи забороняти його подальшу обробку (Рисунок 2.1). Можна також визначити патерни для url, які дозволять spring security на рівні фільтрів визначати чи доступ до даного ресурсу можуть отримати всі чи тільки залогінені користувачі.

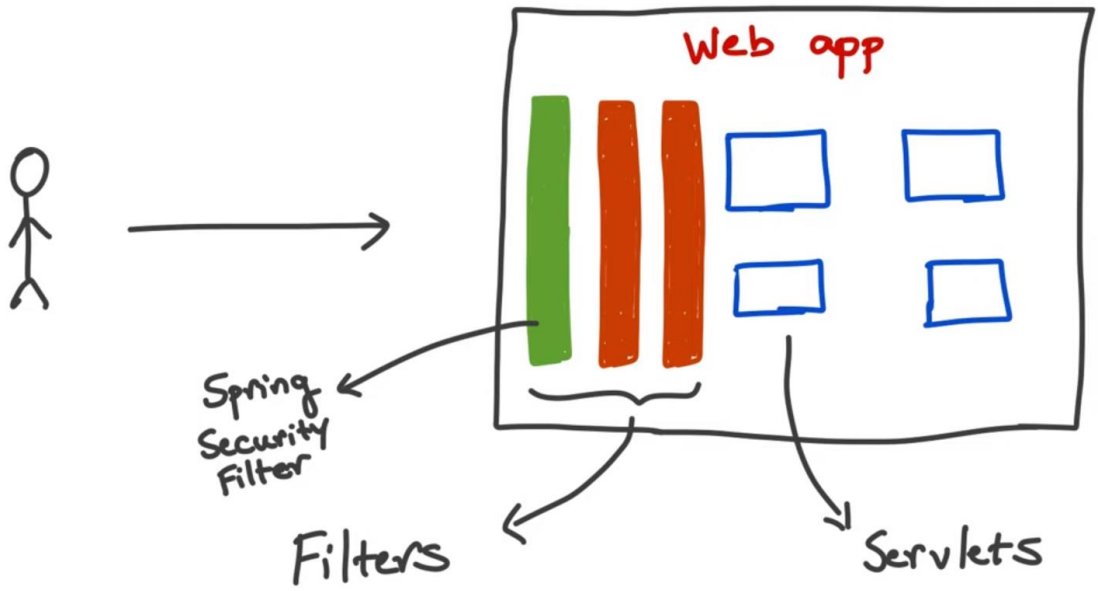

Рисунок 2.1 – Робота Spring security на рівні фільтрів [5]

Автентифікація в Spring security

Spring security має Authentication manager, який управляє автентифікацією в застосунку. Цей менеджер має метод authenticate(), який повертає успішну автентифікацію, або повертає помилку для неуспішної автентифікації (Рисунок 2.2).

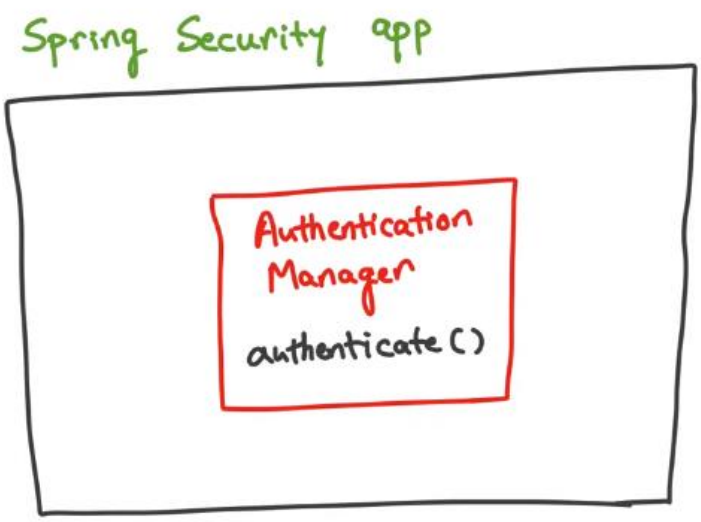

Рисунок 2.2 – Робота Spring security на рівні Authentication manager [5]

Немає необхідності створювати Authentication manager вручну, достатньо його сконфігурувати за допомогою AuthenticationManagerBuilder класу. Для того щоб отримати об’єкт цього класу необхідно створити клас для конфігурації security застосунку. Даний клас повинен наслідувати WebSecurityConfigurerAdapter. В середині цього класу можна перевизначити метод, який на вхід отримує AuthenticationManagerBuilder і виконати всі необхідні налаштування для автентифікації в нашому застосунку.

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
        super.configure(auth);
    }
}
```

Рисунок 2.3 – Конфігурація Authentication manager

Авторизація в Spring security:

Для того щоб сконфігурувати авторизацію застосунку необхідно перевизначити метод configure(HttpSecurity) в класі який наслідується від WebSecurityConfigurerAdapter (Рисунок 2.3).

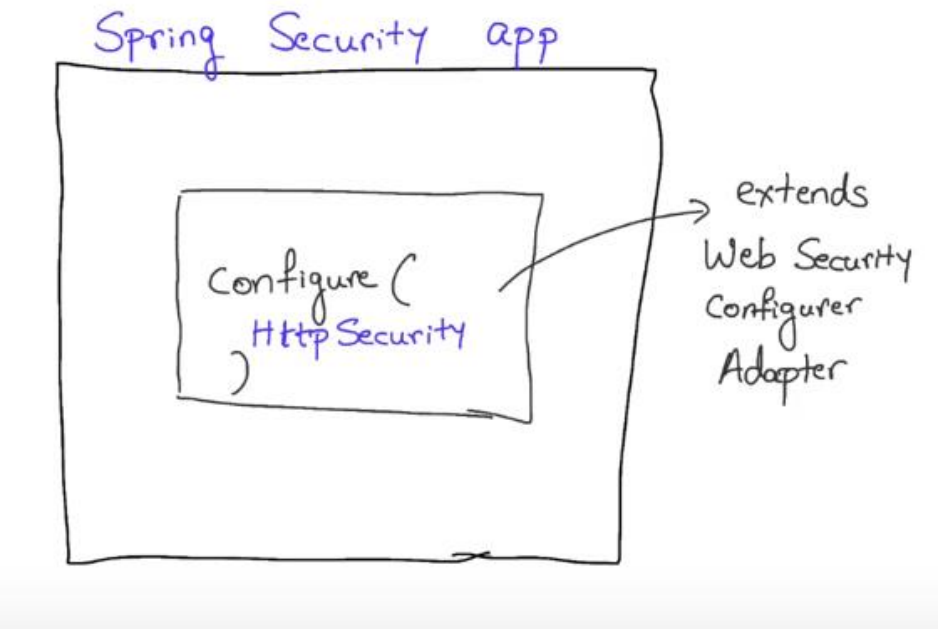

Рисунок 2.4 – Конфігурація HttpSecurity [5]

Як працює автентифікація та авторизація всередині:

Насправді на рівні фільтрів Spring security має не один фільтр, а декілька. Ці фільтри ще називають security filter chain. Один з цих фільтрів і

відповідає безпосередньо за автентифікацію користувачів в системі (Рисунок 2.4).

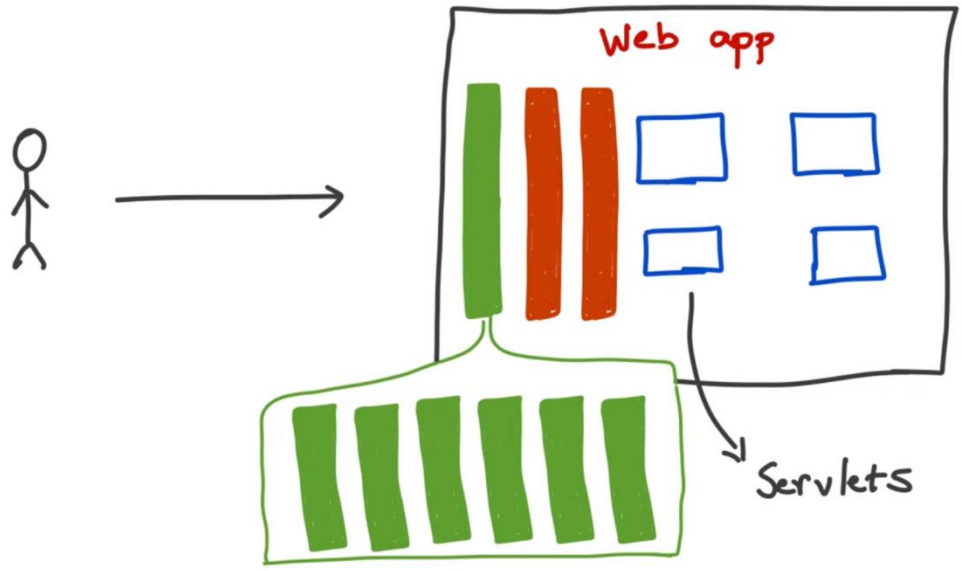

Рисунок 2.5 – Spring security filter chain [5]

Автентифікацію виконує клас під назвою AuthenticationProvider, який реалізує метод authenticate(). Цей метод приймає на вході user credentials, а на виході віддає principal, користувача, який вже залогінений в систему. User credentials і principal передаються у вигляді Authentication об’єкта, який слугує обгорткою для цих даних. Spring security вже має набір наперед визначених провайдерів для найбільш часто вживаних методів автентифікації, але якщо є необхідність, то можна реалізувати власний провайдер і показати spring security, що треба використовувати саме його (Рисунок 2.6).

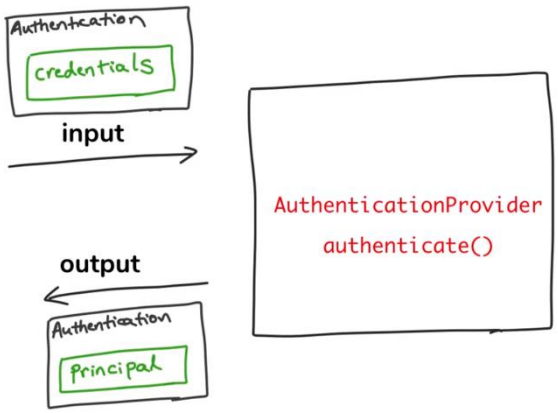

Рисунок 2.6 – Робота Authentication Provider [5]

Один застосунок може мати декілька стратегій автентифікації (логін\пароль, OAuth, Ldap etc). В результаті 1 застосунок може мати декілька AuthenticationProvider, кожен з який буде мати специфічний механізм автентифікації (Рисунок 2.7).

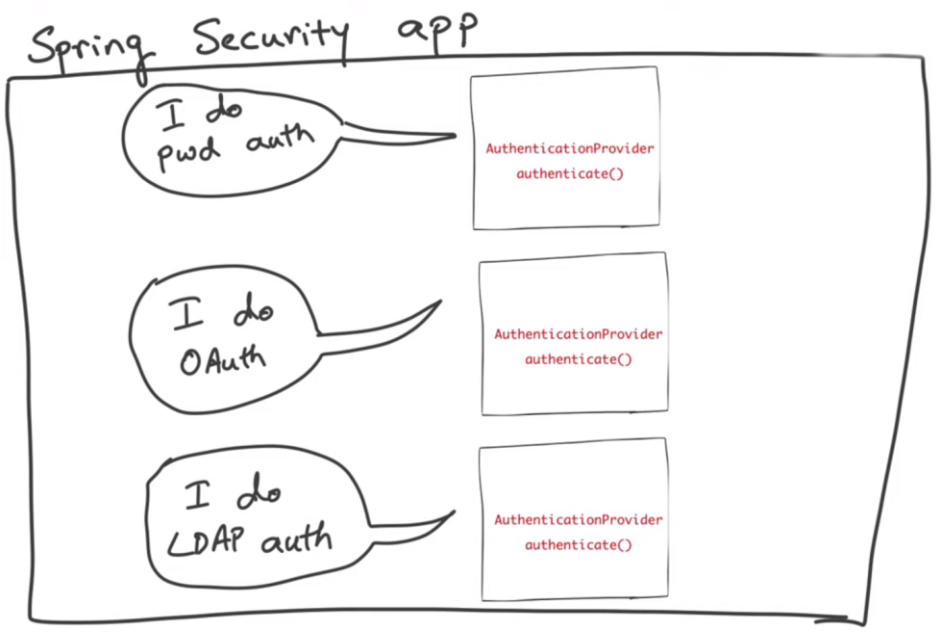

Рисунок 2.7 – Робота Authentication providers [5]

Кожен з Authentication Provider буде приймати Authentication об’єкт, але кожен з них знає, які саме він очікує отримати user credentials і як використати ці дані для автентифікації користувача. Кожен Authentication Provider відповідає за певний тип автентифікації.

Для того щоб координувати роботу всіх цих провайдерів, Spring security має спеціальний об’єкт під назвою Authentication manager. Authentication manager не автентифікує користувача самостійно, він лише делегує цей процес відповідному Authentication Provider залежно від типу автентифікації. Для того щоб мати змогу визначити тип автентифікації у конкретного провайдера, кожен провайдер повинен реалізувати метод supports(), який повертає підтримуваний провайдером тип автентифікаії (Рисунок 2.8).

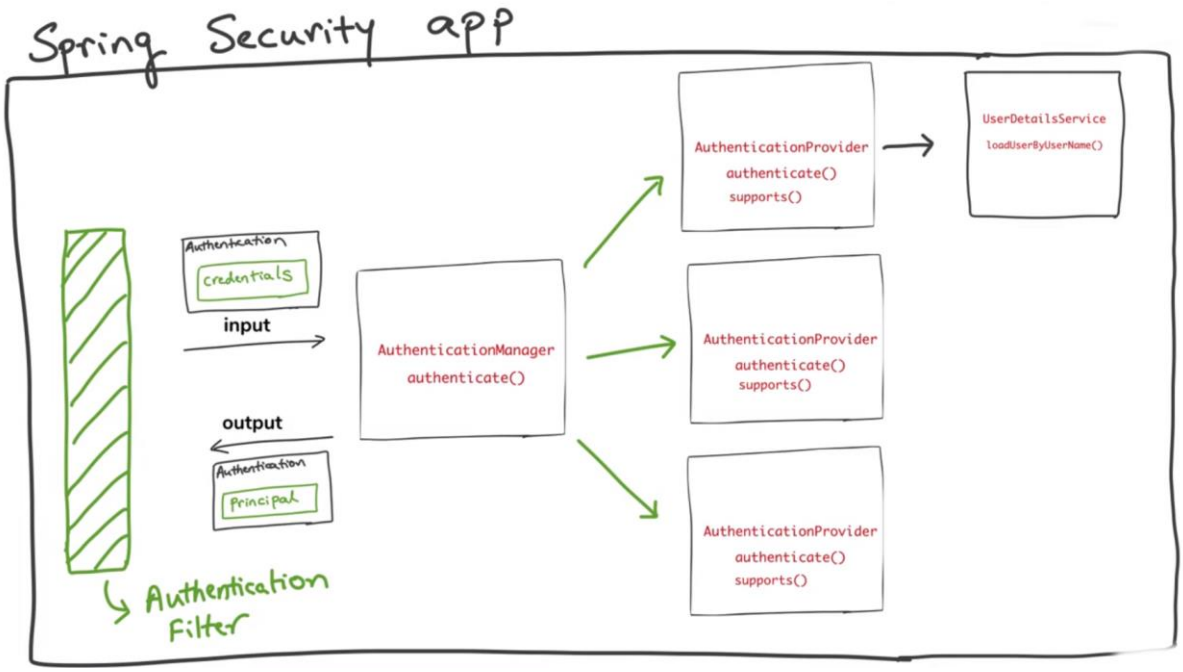

Рисунок 2.8 – Загальна робота Spring security [5]

Після того як Authentication Provider отримав дані користувача, він повинен автентифікувати його за цими даними. Для того що зробити це йому необхідно зробити запит до сховища, де зберігається інформація про зареєстрованих користувачів, на основі userid, яке було передано провайдеру в якості даних користувача. Після того як Authentication Provider отримав дані про цього користувача зі сховища, він може перевірити відповідність credentials збережених в сховищі до credentials вказаних в запиті (Рисунок 2.9).

Рисунок 2.9 – Облікові дані для Authentication provider [5]

Автоматизуючи цей процес можна сказати, що для різних сервісів буде різним тільки запит на інформацію про користувачів до сховища. Вся інша перевірка, яка знаходиться в Authentication Provider залишається незмінною. Саме тому Spring security надає інтерфейс UserDetailsService, за допомогою якого реалізується запит до сховища, щоб дізнатись інформацію про користувача. Тому, якщо застосунок має звичайну автентифікацію і інформація про користувачів збережена в базі даних, то достатньо реалізувати лише клас, що імплементує UserDetailsService, а всю іншу роботу виконає Spring security (Рисунок 2.10).

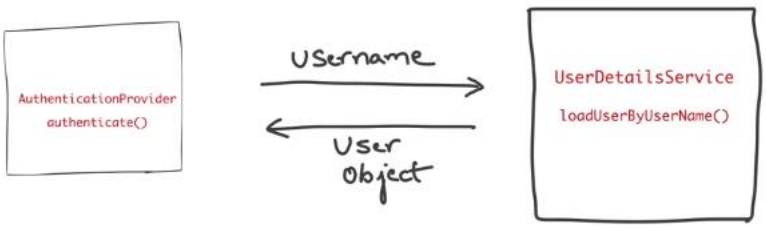

Рисунок 2.10 – Робота UserDetailsService [5]

Якщо автентифікація пройшла успішно, Authentication manager повертає Authentication об’єкт в якому збережена вся інформація про залогіненого користувача і цей об’єкт зберігається в SecurityContext, за допомогою якого можна знати на кожному кроці нашого застосунку хто є залогіненим користувачем в системі на даний момент.

2.3OAuth2 authorization flow

OAuth визначає 4 ролі:

1) Resource owner – об’єкт здатний надати доступ до захищеного ресурсу. Коли власником ресурсу є людина, вона також є end-user.
2) Resource server – сервер, на якому розміщені захищені ресурси, здатний приймати та відповідати на запити до захищених ресурсів використовуючи аксес токени (access tokens).
3) Client – застосунок, який виконує запити на захищені ресурси від імені власника ресурсу та з його доволу. Термін «клієнт» не передбачає якоїсь конкретної реалізації.
4) Authorization server – сервер, який видає аксес токени клієнту після успішної автентифікації власника ресурсів.

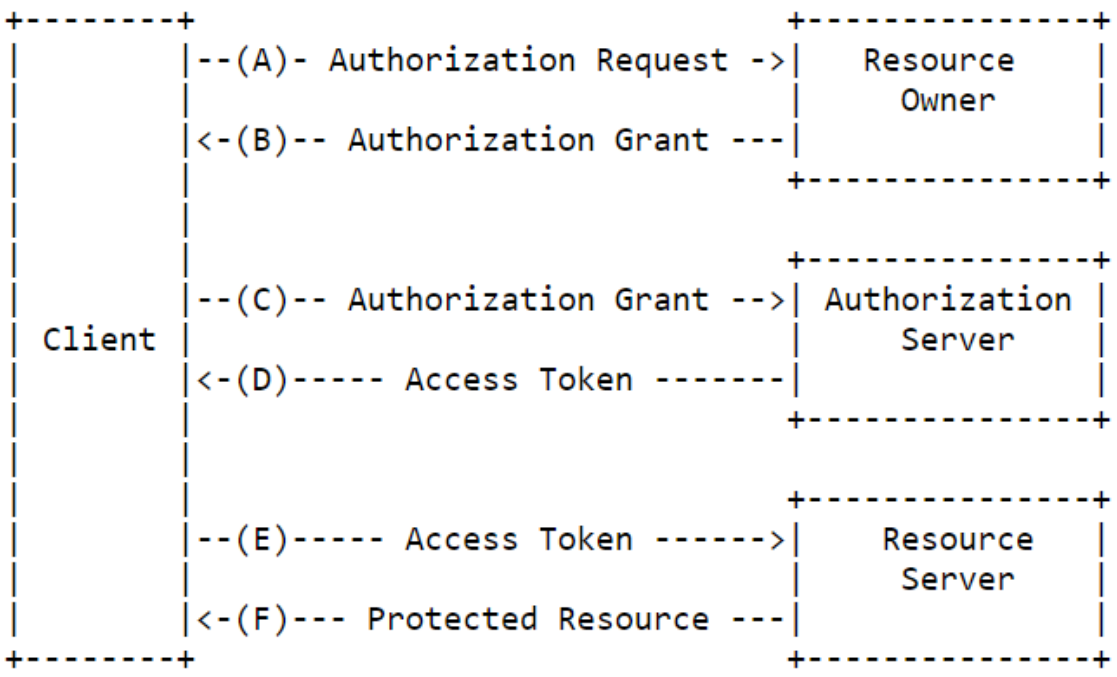

Рисунок 2.11 – OAuth2 flow [6]

Абстрактний OAuth2 flow пояснює взаємодію між ролями описаними вище (Рисунок 2.11):

(A) Клієнт робить запит на авторизацію від власника ресурсу. Цей запит може бути зроблено напряму до власника ресурсу, або ж опосередковано через сервер авторизації (що є більш надійно).

(B) Клієнт отримує authorization grant, який показує що власник ресурсу дав дозвіл клієнту на авторизацію до його захищеного ресурсу. Тип authorization grant залежить від методу використаного клієнтом для запиту на авторизацію і типів доступних на сервері авторизації.

(C) Клієнт робить запит до серверу авторизації на аксес токен, авторизуючись за допомогою отриманого authorization grant.

(D) Сервер авторизації автентифікує клієнт і валідує authorization grant, і якщо він валідний, то надсилає клієнту аксес токен.

(E) Клієнт робить запит на захищений ресурс з сервера ресурсів і автентифікується за допомогою аксес токена.

(F) Сервер ресурсів валідує аксес токен, і якщо валідний, обробляє запит.

Authorization grant (далі «дозвіл на авторизацію») це credentials, які показують що власник ресурсу успішно авторизувався в системі (для того щоб мати доступ до його захищених ресурсів). Специфікація визначає 4 типи дозволів на авторизацію:

1) Authorization code – код авторизації отримується за допомогою сервера авторизації, який є посередником між клієнтом та власником ресурсу. Замість того щоб запросити дозвіл від власника ресурсу напряму, клієнт перенаправляє власника ресурсу на сервер авторизації, який в свою чергу авторизує клієнта і повертає код авторизації. Перед тим як відправити код авторизації, сервер авторизації автентифікує

власника ресурсу і виконує авторизацію. Таким чином credentials власника ресурсу ніколи не передаються клієнту.

2) Implicit – це спрощений flow коду авторизації оптимізований для клієнтів, які реалізовані в браузері використовуючи мову скриптів, таку як JavaScript. В цьому варіанті, замість того щоб видавати клієнту код авторизації, йому відразу видається аксес токен. Дозвіл на авторизацію називається неявним оскільки ніяких проміжних кодів авторизації не генерується і клієнт одразу отримує аксес токен. При implicit flow сервер авторизації не автентифікує клієнта.
3) Resource owner password credentials – облікові дані (логін та пароль) користувача можуть бути використані напряму як дозвіл на авторизацію для клієнта. Такий тип дозволу може бути використаний лише тоді, коли власник ресурсу і клієнт мають високий рівень довіри, або немає іншого способу організувати дозвіл на авторизацію. Не дивлячись на те, що облікові дані користувача є достатньою умовою для доступу на захищений ресурс, клієнт все рівно отримує аксес токен після першого запиту на захищений ресурс з обліковими даними користувача.
4) Client credentials – облікові дані клієнта використовуються як дозвіл на авторизацію зазвичай тоді, коли клієнт є власником ресурсу, або коли клієнт робить запит на доступ до захищених ресурсів на основі авторизації раніше узгодженої з сервером авторизації.

Існує також декілька видів токенів:

1) Access token – це облікові дані власника ресурсу, які використовуються для отримання доступу до захищених

ресурсів. Цей токен має певний термін дії, який може закінчуватись.

2) Refresh token – це облікові дані власника ресурсу, які використовуються для того щоб повторно отримати аксес токен, після того як термін його дії закінчився. Цей тип токенів клієнт може отримати від сервера авторизації. Цей вид токенів використовується тільки сервером авторизації і ніколи не може бути використаний для отримання захищених ресурсів

Процес оновлення аксес токену (Рисунок 2.12):

(A) Клієнт робить запит на аксес токен, автентифікуючись за допомогою серверу авторизації надаючи йому дозвіл на авторизацію.

(B) Сервер авторизації автентифікує клієнта і валідує дозвіл на авторизацію, і якщо вона валідна, повертає аксес токен та рефреш токен клієнту.

(C) Клієнт робить запит на захищений ресурс на сервер ресурсів надаючи йому аксес токен.

(D) Сервер ресурсів валідує аксес токен і якщо валідний обробляє запит.

(E) Кроки (C) і (D) повторюються, поки термін дії аксес токену не закінчується. Якщо клієнт знає, що термін дії аксес токену закінчився, він переходить до кроку (G). Якщо він цього не знає, то він робить ще один запит до захищеного ресурсу.

(F) Так як у аксес токену закінчився термін дії, сервер ресурсів повертає помилку про те що аксес токен не правильний.

(G) Клієнт робить запит на новий аксес токен, автентифікуючись на сервері авторизації і надаючи йому рефреш токен.

(H) Сервер авторизації автентифікує клієнт і валідує рефреш токен, і якщо токен валідний – генерує новий аксес токен, який відправляється до клієнта.

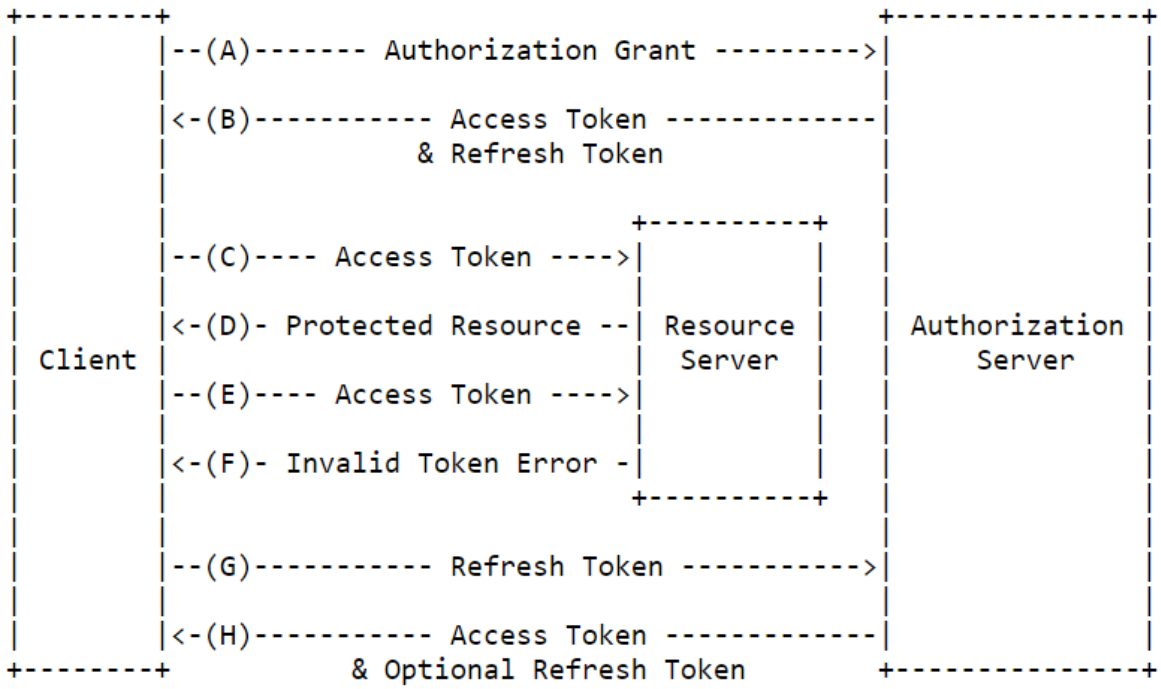

Рисунок 2.12 – OAuth2 flow with refresh tokens [6]

2.4 Реєстрація застосунку в Microsoft Azure active directory

Заходимо в AAD і вибираємо вкладку реєстрації застосунку. На цій вкладці обираємо «Нова реєстрація» (Рисунок 2.13).

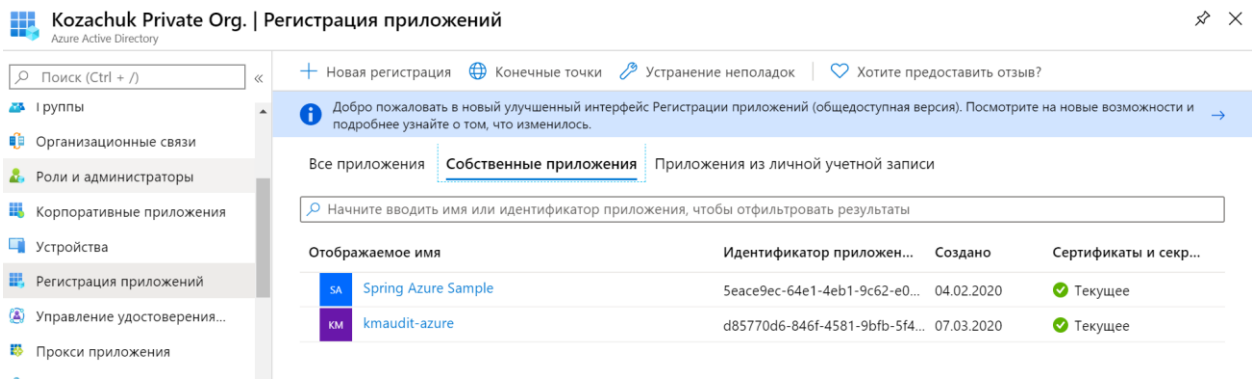

Рисунок 2.13 – Реєстрація застосунку в Microsoft Azure

Вводимо ім’я застосунку і вибираємо які типи облікових записів буде підтримувати наш застосунок. Клікаємо створити (Рисунок 2.14).

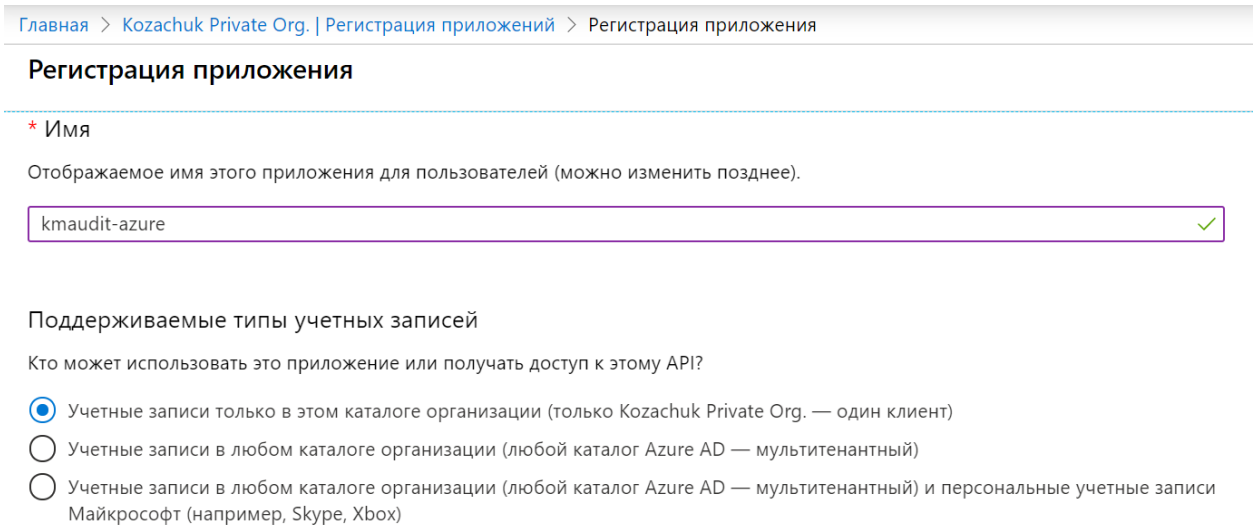

Рисунок 2.14 – Реєстрація застосунку в Microsoft Azure 2

На вкладці «Сертифікати і клієнти» нам необхідно створити новий секрет для клієнта. Цей секрет постійно скопіювати після генерації, тому що після цього його вже не можна буде відкрити на перегляд (Рисунок 2.15).

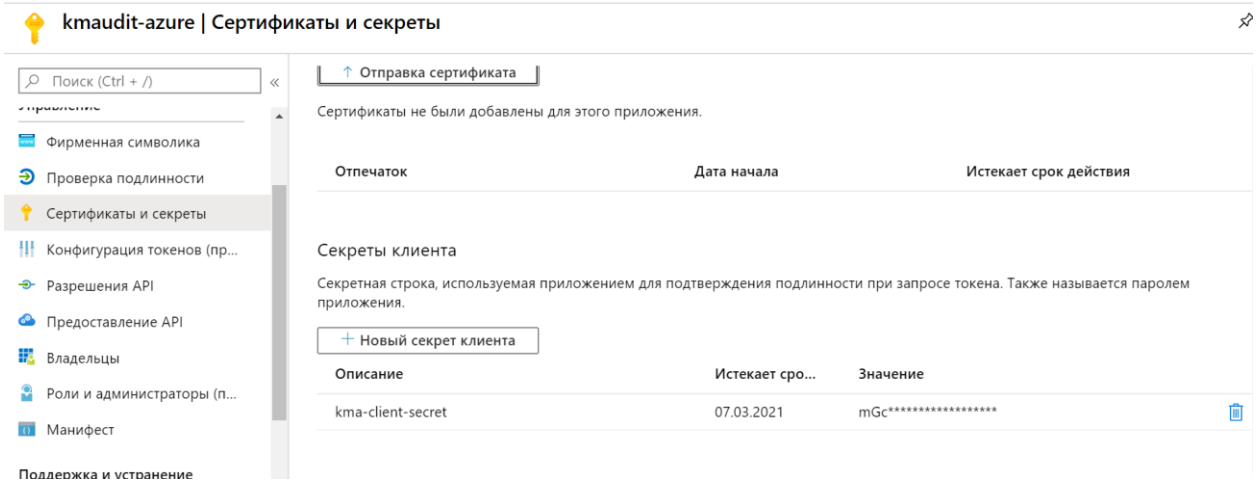

Рисунок 2.15 – Створення секретного ключа

За необхідності можна додати права API, які можуть мати користувачі (Рисунок 2.16).

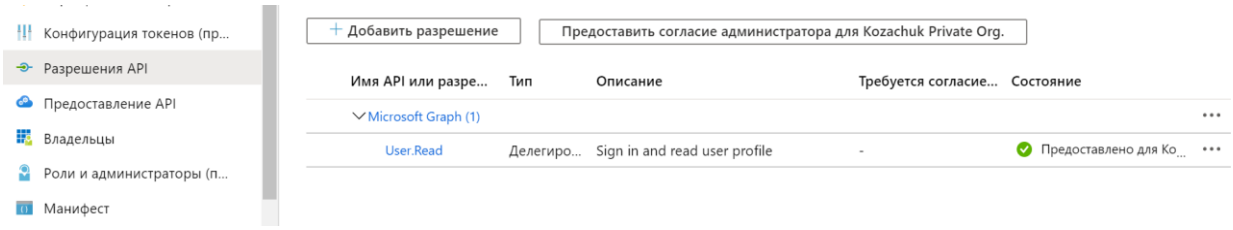

Рисунок 2.16 – Надання прав

Також необхідно вказати Url на який буде перенаправлятись застосунок після успішної автентифікації користувачів (Рисунок 2.17).

Рисунок 2.17 – Конфігурація Url перенаправлення

Для того щоб дозволити Implicit flow в своєму застосунку в файлі Маніфесті ви повинні поставити true для полів oauth2AllowIdTokenImplicitFlow і oauth2AllowImplicitFlow (Рисунок 2.18).

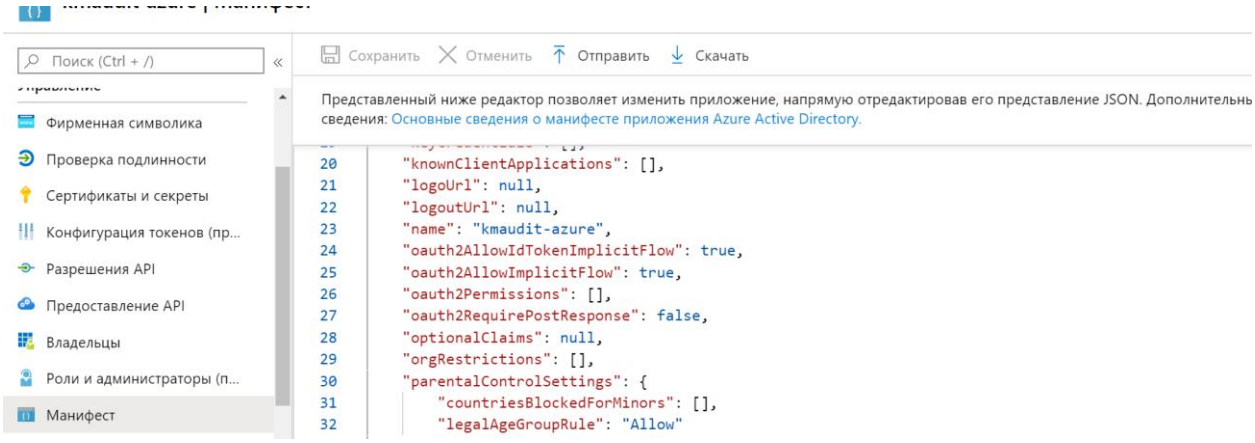

Рисунок 2.18 – Маніфест застосунку

Тепер необхідно створити користувачів, які зможуть логінитись до застосунку (Рисунок 2.19).

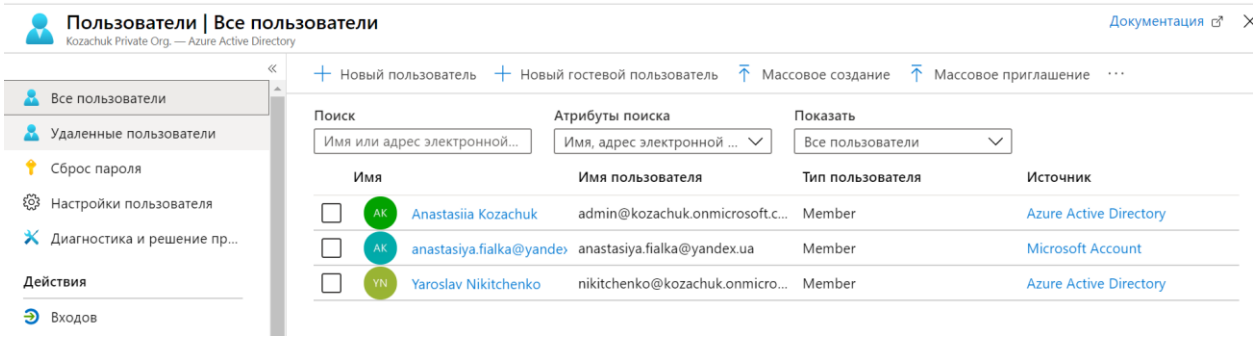

Рисунок 2.19 – Створення користувача

Для того щоб сконфігурувати Spring boot застосунок для роботи з зареєстрованим Azure застосунком необхідно скопіювати наступні дані:

1) tenant-id
2) client-id
3) client-secret

2.5 OAuth2 with Azure

Схема роботи OAuth2 для Microsoft Azure (Рисунок 2.20):

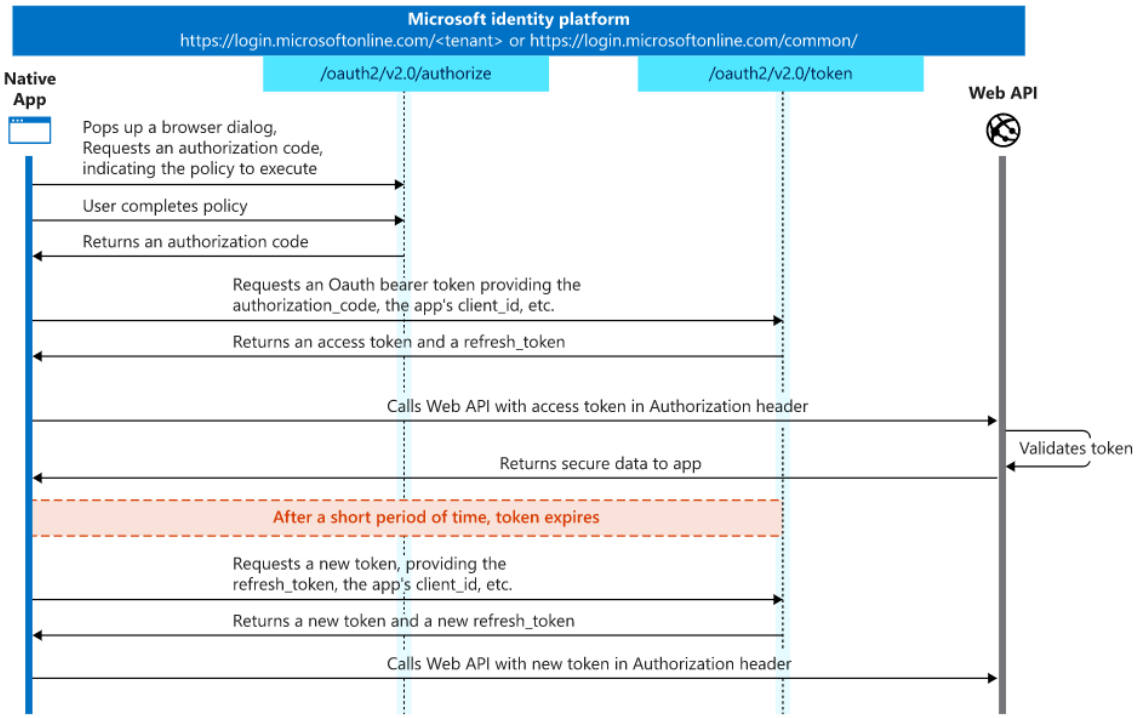

Рисунок 2.20 – Microsoft Azure OAuth2 flow [7]

Детальний опис Microsoft Azure AOuth2 flow:

1) Запит на код авторизації (Рисунок 2.21). Клієнт перенаправляє користувача на /authorize ендпоінт. На цьому етапі користувач буде повинен ввести свої облікові дані і завершити автентифікацію. Ендпоінт Microsoft identity платформи також перевіряє чи користувач дав згоду на permissions зазначені в параметрі запиту scope. Якщо користувач не давав раніше згоди на використання цих permissions, Microsoft identity попросить підтвердити надання необхідних permissions клієнту. Після того як користувач автентифікувався і надав всі необхідні дозволи, платформа Microsoft identity поверне відповідь на Url вказаний в параметрі redirect_uri, використовуючи метод вказаний в параметрі response_mode.

Request an authorization code

https://login.microsoftonline.com/{tenant}/oauth2/v2.0/authorize?
client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&response_type=code
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&response_mode=query
&scope=openid%20offline_access%20https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&state=12345

Successful response

GET https://login.microsoftonline.com/common/oauth2/nativeclient?
code=AwABAAAAvPM1KaPlrEqdFSBzjqfTGBCmLdgfSTLEMPGYuNHSUYBrq...
&state=12345

Error response

GET https://login.microsoftonline.com/common/oauth2/nativeclient?
error=access_denied
&error_description=the+user+canceled+the+authentication

Рисунок 2.21 – Запит на код авторизації [7]

2) Запит на аксес токен(Рисунок 2.22). Після того як клієнт отримав authorization_code від сервера авторизації, можна робити запит на

отримання аксес токену. Це можна зробити виконавши запит POST на /token ендпоінт.

Request an access token

```
POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&code=OAAABAAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq3n8b2JRLk4OxVXr...
&redirect_uri=http%3A%2F%2Flocalhost%2Fmyapp%2F
&grant_type=authorization_code
&client_secret=JqQX2PNo9bpM0uEihUPzyrh
```

Successful response


```
{
   "access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...",
   "token_type": "Bearer",
   "expires_in": 3599,
   "scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.read",
   "refresh_token": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
   "id_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTctOD...",
}
```


Error response


```
{
  "error": "invalid_scope",
  "error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not valid. The scope https://foo.microsoft.com/mail.read is not valid.\r\nTrace ID: 255d1aef-8c98-452f-ac51-23d051240864\r\nCorrelation ID: fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7\r\nTimestamp: 2016-01-09 02:02:12Z",
  "error_codes": [
    70011
  ],
  "timestamp": "2016-01-09 02:02:12Z",
  "trace_id": "255d1aef-8c98-452f-ac51-23d051240864",
  "correlation_id": "fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7"
}
```


Рисунок 2.22 – Запит на аксес токен [7]

3) Використання аксес токену(Рисунок 2.23). Після того як сервер авторизації повернув аксес токен, його можна використовувати для доступу на захищені ресурси.

Use the access token

```
GET /v1.0/me/messages
Host: https://graph.microsoft.com
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...
```

Рисунок 2.23 – Використання аксес токену [7]

4) Оновлення аксес токену(Рисунок 2.24). Аксес токени діють не довго і повинні бути оновлені після закінчення терміну дії. Це можна зробити, виконавши ще один POST запит на /token ендпоінт. Тільки на цей раз в параметрах необхідно вказати refresh_token замість code. Рефреш токени не мають визначеного терміну дію, тому вони досить довговічні, але все ж можуть бути відкликані сервером авторизації.

Refresh the access token

POST /{tenant}/oauth2/v2.0/token HTTP/1.1
Host: https://login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

client_id=6731de76-14a6-49ae-97bc-6eba6914391e
&scope=https%3A%2F%2Fgraph.microsoft.com%2Fmail.read
&refresh_token=OAAABAAAAiL9Kn2Z27UubvWFPbm0gLWQJVzCTE9UkP3pSx1aXxUjq..
.
&grant_type=refresh_token
&client_secret=JqQX2PNo9bpM0uEihUPzyrh

Successful response

{
"access_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Ik5HVEZ2ZEstZnl0aEV1Q...",
"token_type": "Bearer",
"expires_in": 3599,
"scope": "https%3A%2F%2Fgraph.microsoft.com%2Fmail.read",
"refresh_token": "AwABAAAAvPM1KaPlrEqdFSBzjqfTGAMxZGUTdM0t4B4...",
"id_token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyZDRkMTFhMi1mODE0LTQ2YTct
OD...",
}

Error response

{
"error": "invalid_scope",
"error_description": "AADSTS70011: The provided value for the input parameter 'scope' is not
valid. The scope https://foo.microsoft.com/mail.read is not valid.\r\nTrace ID: 255d1aef-8c98-
452f-ac51-23d051240864\r\nCorrelation ID: fb3d2015-bc17-4bb9-bb85-
30c5cf1aaaa7\r\nTimestamp: 2016-01-09 02:02:12Z",
"error_codes": [
70011
],
"timestamp": "2016-01-09 02:02:12Z",
"trace_id": "255d1aef-8c98-452f-ac51-23d051240864",
"correlation_id": "fb3d2015-bc17-4bb9-bb85-30c5cf1aaaa7"
}

Рисунок 2.24 – Оновлення аксес токену [7]

2.6 OAuth2 Azure Spring boot configuration

Для того щоб налаштувати Spring boot для роботи з OAuth2 Azure flow необхідно вказати в файлі application.properties client-id, client-secret, directory-groups (Рисунок 2.25).

```
# Specifies your App Registration's Application ID:
azure.activedirectory.client-id=d85770d6-846f-4341-9bfb-5f46473b764c

# Specifies your App Registration's secret key:
azure.activedirectory.client-secret=mGc1W9Rxobt-[cHUX7d1Saaw..

# Specifies the list of Active Directory groups to use for authorization:
azure.activedirectory.active-directory-groups=Users
```

Рисунок 2.25 – Конфігурація OAuth2 в application.properties

Також в pom.xml необхідно додати наступні залежності:

1) azure-active-directory-spring-boot-starter
2) spring-security-oauth2-jose
3) spring-security-oauth2-client

В класі, який наслідує WebSecurityConfigurerAdapter необхідно перевизначити метод для того щоб сконфігурувати авторизацію за допомогою Microsoft Azure (Рисунок 2.26).

```
@Autowired
private AADAuthenticationFilter aadAuthFilter;

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS,"/**").permitAll()
        .anyRequest()
        .authenticated();

    http.cors().and().csrf()
        .disable()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.addFilterBefore(aadAuthFilter, UsernamePasswordAuthenticationFilter.class);
}
```

Рисунок 2.26 – Конфігурація авторизації

В цьому методі описуються патерни Url, які можуть бути доступні без аторизації і ті які будуть доступні лише авторизованим користувачам. Також відключаються csrf токени, а також cors політики. Створюється STATELESS сессія. А також додається фільтр, який приймає на вхід токени авторизованого користувача і дістає ролі для цього користувача з Microsoft Azure (цей токен буде отримувати UI частина застосунку і передавати його серверу).

Додавши свій власний постфільтр після AADAuthenticationFilter ми маємо можливість надати користувачу ролі, які будуть отримані не з Microsoft Azure, а з нашого власного ресурсу.

РОЗДІЛ 3: ReactJs та Redux як інструменти для розробки UI частини застосунку

3.1 Компоненти в ReactJs

Компоненти є блоками, з яких будується будь-який React застосунок. Компоненти бувають двох типів:

1) Компонент функція (Рисунок 3.1) – це компонент, який написаний у вигляді звичайної JavaScript функції. Цей тип компонентів слугує суто для відображення даних. Компонент функція може небов'язково приймати параметри і повинна повертати реакт елемент, який буде відображено на сторінці. Основні характеристики компонента функції:
    - Functional – тому що в основі цього компонента лежить функція.
    - Stateless – тому що цей тип компонентів не зберігає\керує станом.
    - Presentational – тому що все що вони роблять, це відображають UI елементи.

```
const UserDetails = ({userDetails}) => (
  <Card small className="mb-4 pt-3">
    <CardHeader className="border-bottom text-center">
      <h4 className="mb-0">{userDetails.name}</h4>
      <span className="text-muted d-block mb-2">Студент</span>
    </CardHeader>
    <ListGroup flush>
      <ListGroupItem className="px-4 text-center">
        <strong className="text-muted d-block mb-2">
          {userDetails.email}
        </strong>
      </ListGroupItem>
    </ListGroup>
  </Card>
);
```

Рисунок 3.1 – Приклад компоненту функції

Компонент функція, описана вище, приймає на вхід параметр userDetails у вигляді JSON об’єкту і все що робить цей компонент, це відображення даних з цього об’єкту.

2) Компонент клас (Рисунок 3.2) – для того щоб створити компонент клас необхідно використати JavaScript ES6 синтакс класів. Компоненти класи можуть мати логіку, локальний стан, змогу взаємодіяти з іншими компонентами. Компонент клас отримує параметри ззовні за допомогою спеціального об’єкту props. Стан може зберігатись в спеціальному полі state. Компонент повинен реалізувати метод render(), який повертає реакт елемент, який повинен бути відображений на сторінці. Основні характеристики компонентів класів:
    - Class – тому що вони реалізуються у вигляді класів.
    - Smart – тому що вони можуть містити в собі логіку.
    - Stateful – тому що вони можуть зберігати\управляти локальним станом.
    - Container – тому що вони зазвичай містять в собі велику кількість інших компонентів.

Приклад компоненту класу, який зберігає state, для того щоб контролювати кількість childElements, які він має і при натисканні користувачем на кнопку відображати ще один компонент на сторінці. Новий компонент відображається за допомогою іншого компоненту <ActiivityCard/>.

Реакт дозволяє компонувати компоненти між собою, створюючи ієрархію компонентів.

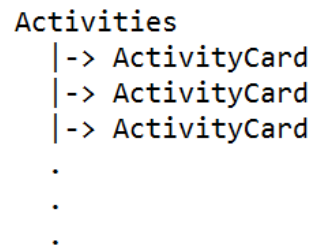

Рисунок 3.2 – Ієрархія компонентів

```
class Activities extends React.Component {

 constructor(props) {
  super(props);
 }

 state = {
  numChildren: 1
 };

 addChildCard = () => {
  this.setState(({numChildren}) => {
   return {
    numChildren: numChildren + 1
   }
  });
 };

 render() {
  const children = [];

   for (let i = 0; i < this.state.numChildren; i += 1) {
    children.push(
     <Row key={i}>
      <Col key={i}>
       <ActivityCard key={i} isNew={i === (this.state.numChildren - 1)}
                     addChild={this.addChildCard}/>
      </Col>
     </Row>);
   }


  return (
   <Container fluid className="main-content-container px-4">
    <Row noGutters className="page-header py-4">
     <PageTitle title="Інформація про загальні активності" md="12"
                className="ml-sm-auto mr-sm-auto"/>
    </Row>
    <ActivityFilter />
    {children}
   </Container>);
 }
}

export default Activities;
```

Рисунок 3.3 – Приклад компоненту класу

3.2 Життєвий цикл компонентів

Кожен компонент має свій життєвий цикл. До кожної фази життєвого циклу компонента реакт надає нам доступ за допомогою спеціальних методів (Рисунок 3.3):

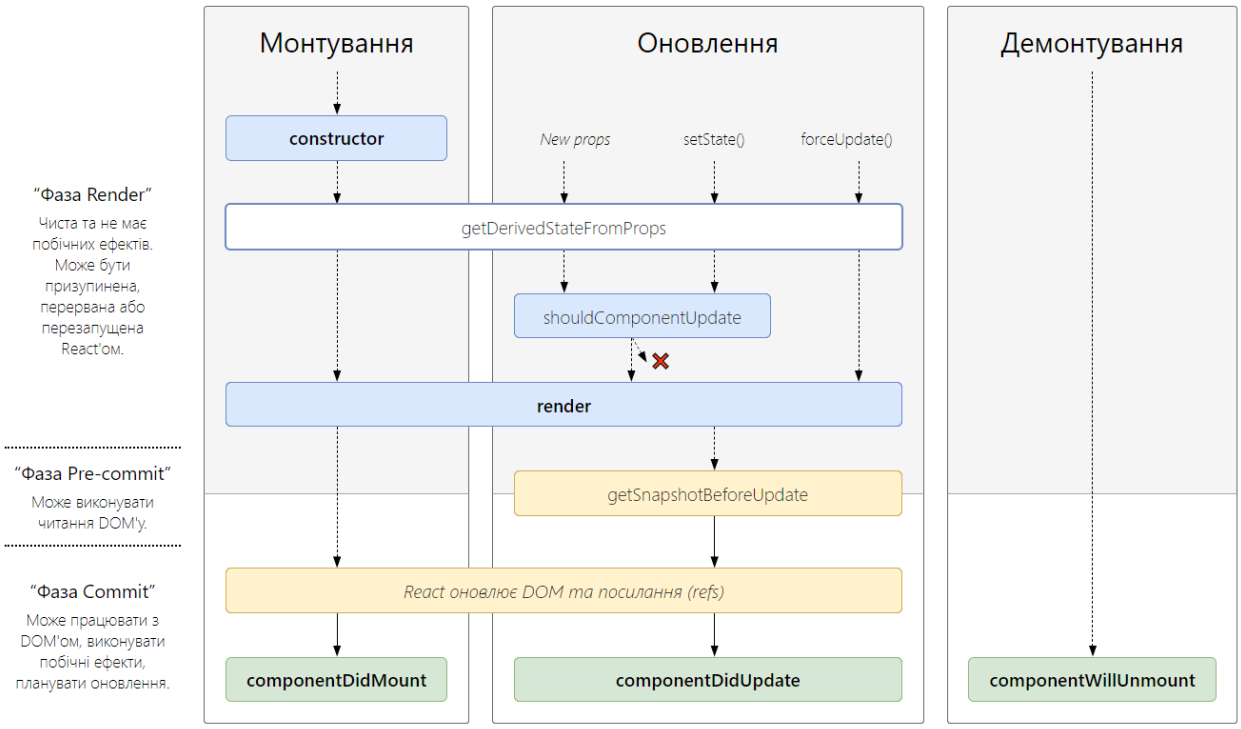

Рисунок 3.4 – Життєвий цикл компоненту [8]

Фази життєвого циклу компонента:

1) Mounting – на цьому етапі відбувається ініціалізація, створюється екземпляр компонента, який встановлюється в DOM дерево. Методи, які викликається на етапі монтування:
    1. Constructor() – конструктор викликається до того, як він буде примонтований. Для того щоб визначити this.props в конструкторі, необхідно викликати super(props). Конструктор як правило використовують в одному з двох випадків: ініціалізація this.state в конструкторі, прив’язка обробників подій до екземпляру класу.
    2. getDerivedStateFromProps() – викликається перед викликом методу render(), як при першому рендері, так і для всіх

наступних. Цей метод необхідний лише для випадків, коли стан компонента залежить від змін в this.props з часом.

3. render() – це метод, який є необхідним для реалізації в компоненті класі. Повертає реакт елементи для подальшого їх відображення на сторінці.
4. componentDidMount() – викликається одразу після монтування компоннета.

2) Updating – кожного разу коли state або props змінюються всередині компоненту, або через API, або через сервер, компонент оновлюється шляхом повторної ініціалізації на сторінці. Методи, які викликається на етапі оновлення:
    1. getDerivedStateFromProps()
    2. shouldComponentUpdate() – за допомогою цього методу можна показати реакт чи поточна зміна стану чи пропсів є причиною для того що заново відрендерити компонент.
    3. render()
    4. getSnapshotBeforeUpdtate() – викликається перед тим, як останній відрендерений елемент буде зафіксовано. Він дозволяє компоненту взяти деяку інформацію з DOM, до того як вона може бути зміненою.
    5. componentDidUpdate() – викликається одразу після оновлення компонента.
3) Unmounting – це остання фаза в життєвому циклі компонента. На цьому етапі компонент видаляється зі сторінки та з віртуального DOM дерева і знищується. Методи, які викликається на етапі оновлення:
    1. componentWillUnmount() – викликається перед тим як компонент буде демонтовано і знищено.

3.3Робота VirtualDom в ReactJs

Virtual Dom це програмна концепція, в якій ідеальне (віртуальне) представлення UI зберігається в пам’яті і синхронізується з справжнім DOM бібліотекою під назвою ReactDom (Рисунок 3.5). Цей процес називається reconciliation.

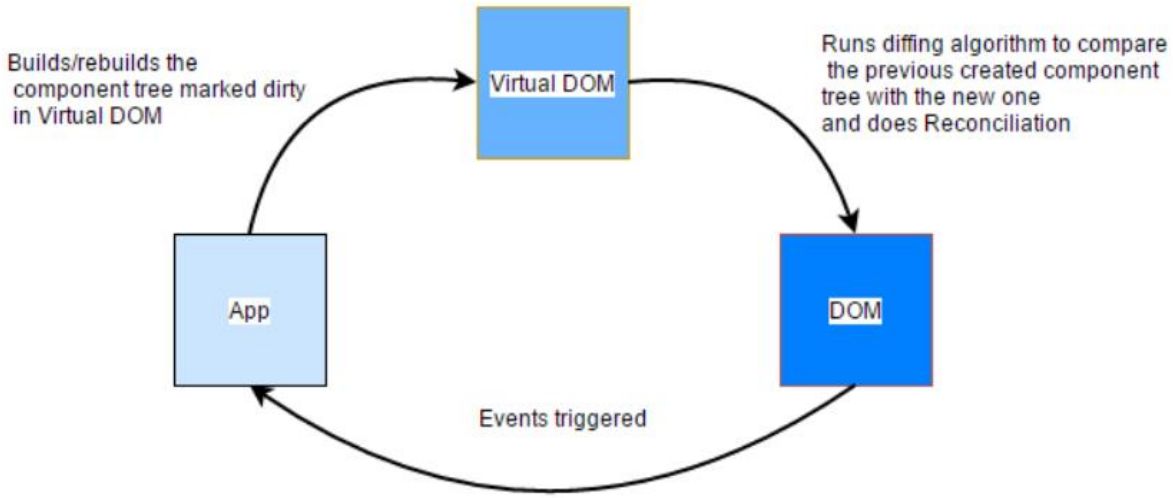

Рисунок 3.5 – Робота VirtualDom [9]

Функція render() створює дерево реакт елементів в певний момент часу. Після наступного оновлення стану функція render() поверне нове дерево елементів. Після того, як render() повертає нове дерево, реакт повинен зрозуміти, як він може найефективніше оновити існуюче дерево Dom().

Реакт реалізує еврестичний алгоритм (O(n)) для вирішення цієї задачі. Цей алгоритм бере свою основу з двох припущень:

1) Два елементи з різними типами будуть мати різні дерева.
2) Розробник може вказати, які дочірні елементи можуть залишатись однаковими між рендерами за допомогою атрибуту key.

Під час порівняння двох дерев реакт спершу порівнює два кореневих елементи. Якщо кореневі елементи мають різні типи, то реакт знищує старе дерево і будує нове з нуля. Під час знищення старого дерева для всіх компонентів класів викликається метод componentWillUnmount().

Під час побудови нового дерева для всіх компонентів викликається метод componentWillMount(), а після цього componentDidMount(). Будь-який стан пов’язаний зі старим деревом втрачається.

При порівнянні двох реакт елементів одного типу, реакт буде дивитись на атрибути обох. Якщо значення атрибутів змінилось, реакт не буде оновлювати весь елемент, а тільки атрибут, що оновився. Після обробки цього вузла, реакт рекурсивно пройдеться по всіх дочірніх елементах. Коли компонент клас оновлюється його екземпляр залишається колишнім, саме тому його стан зберігається між рендерами.

За замовчуванням при рекурсивному обході дочірніх елементів вузла реакт перевіряє два списки потомків одночасно і якщо знаходить відмінність, то в такому випадку створює мутацію і оновлює DOM (Рисунок 3.6).

```
<ul>
  <li>перший</li>
  <li>другий</li>
</ul>

<ul>
<li>перший</li>
<li>другий</li>
<li>третій</li>
</ul>
```

Рисунок 3.6 – Порівняння однакових елементів

При перевірці такого списку реакт буде перевіряти перший елемент з першим, другий з другим, і на третьому він визначить, що треба додати один елемент до списку в DOM.

Проте якщо елемент вставити на початку, а не в кінці, то реакт зрівняє перший елемент з першим, побачить що значення змінилось і оновить його в DOM. Те ж саме відбудеться з другим елементом. Третій елемент буде додано в DOM, як новий елемент (Рисунок 3.7).

Неефективність описана вище може стати проблемою і значно сповільнити час оновлення сторінки.

```
<ul>
  <li>перший</li>
  <li>другий</li>
</ul>

<ul>
<li>третій</li>
<li>перший</li>
<li>другий</li>
</ul>
```

Рисунок 3.7 – Порівняння однаковий елементів 2

Для того щоб вирішити дану проблему реакт підтримує атрибут key, який може бути доданий до однотипних дочірніх елементів. Коли в дочірніх елементів є ключі, реакт використовує ці ключі для того щоб порівняти потомків дерева до і після його оновлення (Рисунок 3.8).

```
<ul>
  <li key="1">перший</li>
  <li key="2">другий</li>
</ul>

<ul>
<li key="3">третій</li>
<li key="1">перший</li>
<li key="2">другий</li>
</ul>
```

Рисунок 3.8 – Порівняння однакових елементів з key

Для даного прикладу реакт буде порівнювати не значення, які записані в реакт елементах, а значення їх key. Тому оновлення такого списку буде ефективним.

3.4Redux state container

Спілкування між реакт компонентами є досить простим, коли компоненти знаходяться в відношенні батько-дитина. Але водночас намагаючись зв'язати компоненти на різних рівнях ієрархії дерева можна досить швидко заплутатись в передаванні даних від компонента компоненту.

Для того щоб організувати спілкування між двома компонентами, які не мають стосунків батько-дитина, можна налаштувати глобальну систему подій. Для цього і слугує Redux. Використовуючи Redux компонентам більш не треба передавати дані один одному вниз по дереву ієрархії (Рисунок 3.9).

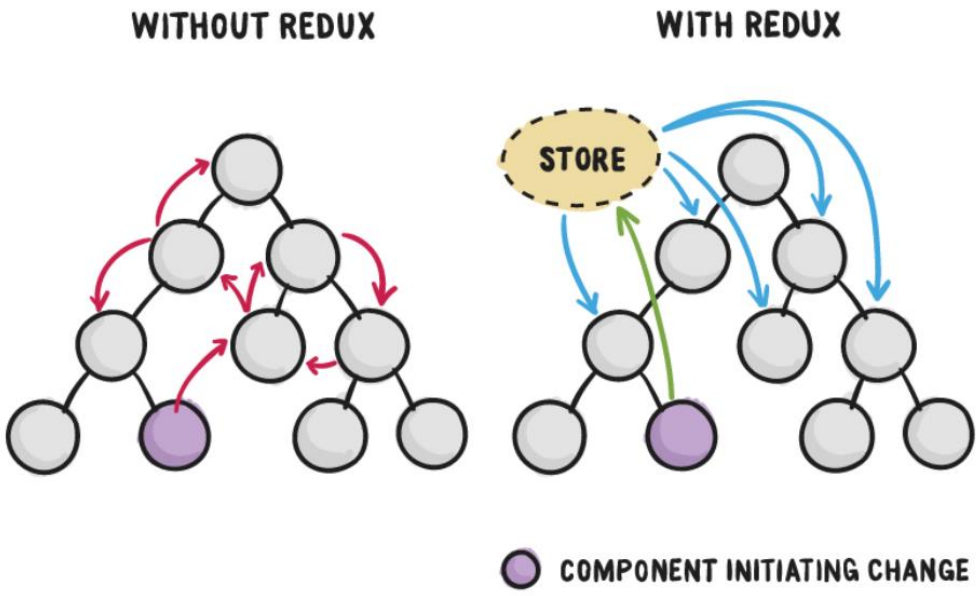

Рисунок 3.9 – Принцип роботи Redux store [10]

Redux – це контейнер стану для нашого застосунку, а дерево стану – це мінімальне представлення нашого додатку в будь-який момент часу. Існує тільки одне дерево стану, яке зберігає стан для всього застосунку. Дерево стану зберігається в сховищі застосунку. Дерево стану є read-only, що означає, що до нього не можна записувати дані напряму. Для того щоб записати дані до дерева стану необхідно використовувати actions.

Якщо коротко, Redux працює наступним чином (Рисунок 3.10):

1) Компоненти мають функції колбеки, які викликаються коли трапляється будь-яка подія на UI.
2) Ці колбеки створюють dispatch actions, основою яких є викликана подія.
3) Reducers обробляють actions і вираховують новий стан.
4) Новий стан для цілого додатку зберігається в єдине сховище.
5) Компоненти отримують новий стан у вигляді пропс і перемальовують себе в необхідних місцях.

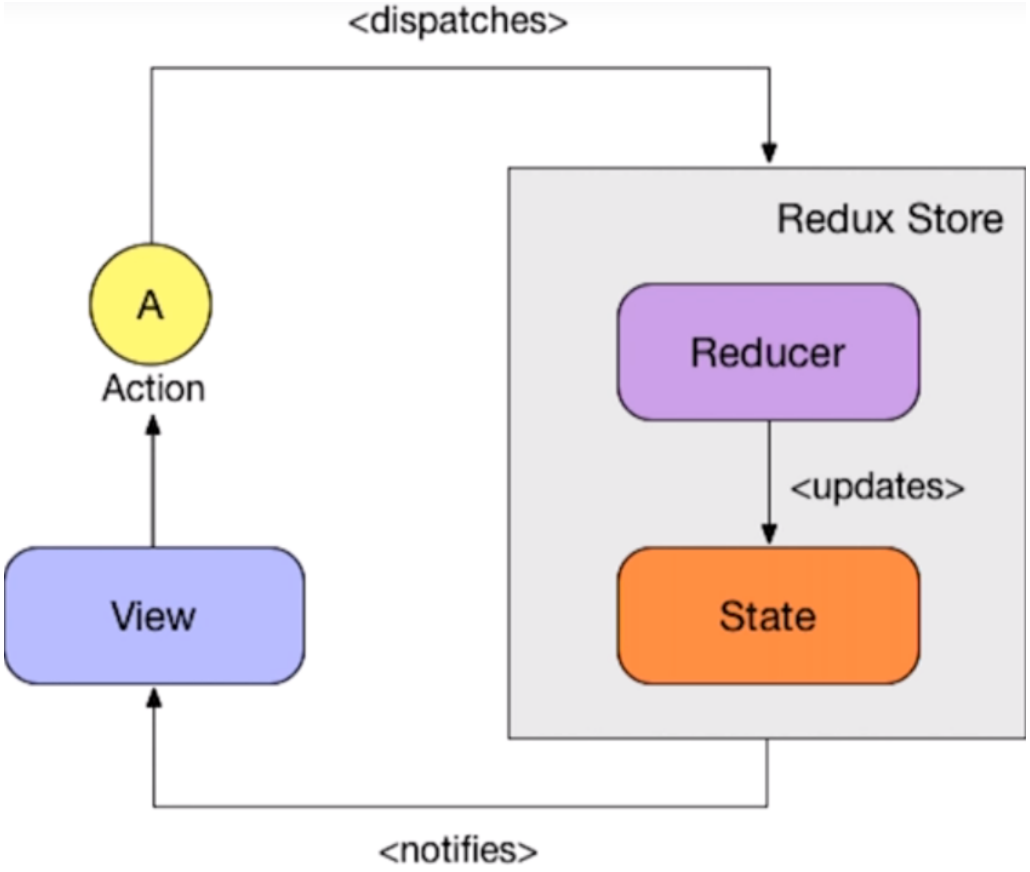

Рисунок 3.10 – Принцип роботи Redux [11]

Actions

Це простий JavaScript об’єкт який описує мінімальний шлях, який є необхідним для того щоб оновити стан застосунку. Actions це спосіб викликати мутацію стану: вони описують, що повинно змінитись в стані, але

не знають як це змінити. Будь-які дані, які потрапляють до редакс сховища, потрапляють туди за допомогою actions.

Actions ідентифікуються значенням, яке називається action type. Цей ідентифікатор є обов'язковим і повинен бути унікальним. Action type повинні мати тип string, тому що повинні мати можливість бути серіалізованими (Рисунок 3.11).

```
export const saveValue = (key, value) => ({
 type: 'SAVE_VALUE',
 key,
 value,
});
```

Рисунок 3.11 – Приклад action

Також actions можуть мати дані, необхідні відповідному редьюсеру для того щоб оновити стан застосунку.

Reducer

Редьюсер це чиста функція, яка описує зміни в стані, які необхідні бути виконані. На відміну від actions, редьюсери знають як змінити стан застосунку, але вони не знають деталей реалізації цих змін (Рисунок 3.12).

```
export default (state, action) => {
 if (action.type === 'SAVE_VALUE') {
  return Object.assign({}, state, {
   [action.key]: action.value,
  })
 }

 return state
}
```

Рисунок 3.12 – Приклад reducer

Редьюсер приймає на вхід поточний стан та поточну actions.

Store

Сховище є своєрідним клеєм, який тримає разом всі блоки редакс: дерево стану, actions dispatcher, і редьюсери застосунку. В додатку може бути лише один інстанс сховища.

Після ініалізації, сховище надає 3 методи:

1) Store.getState() – сховище зберігає дерево стану. Цей метод повертає поточний стан редакс сховища.
2) Store.dispatch({ type: 'SAVE_VALUE' }) – редакс сховище дозволяє нам викликати дії над ним, для того щоб змінити його стан. Все що необхідно, це передати валідний actions об’єкт і все інше редакс сховище зробить самостійно.
3) Store.subscribe(callback) – цей метод приймає на вхід колбек функцію, яка буде викликатись кожного разу, коли стан застосунку буде змінюватись.

```
const store = createStore(persistedReducer);
```

Рисунок 3.13 – Приклад створення сховища

Тепер можна створити сховище, використавши функцію createStore() з бібліотеки redux і передати в цей метод створений нами редьюсер (Рисунок 3.13).

3.5Реалізація сторінки з використанням ReactJS та Redux

Розглянемо як приклад реалізацію сторінки профілю користуача (Рисунок 3.14, Рисунок 3.15).

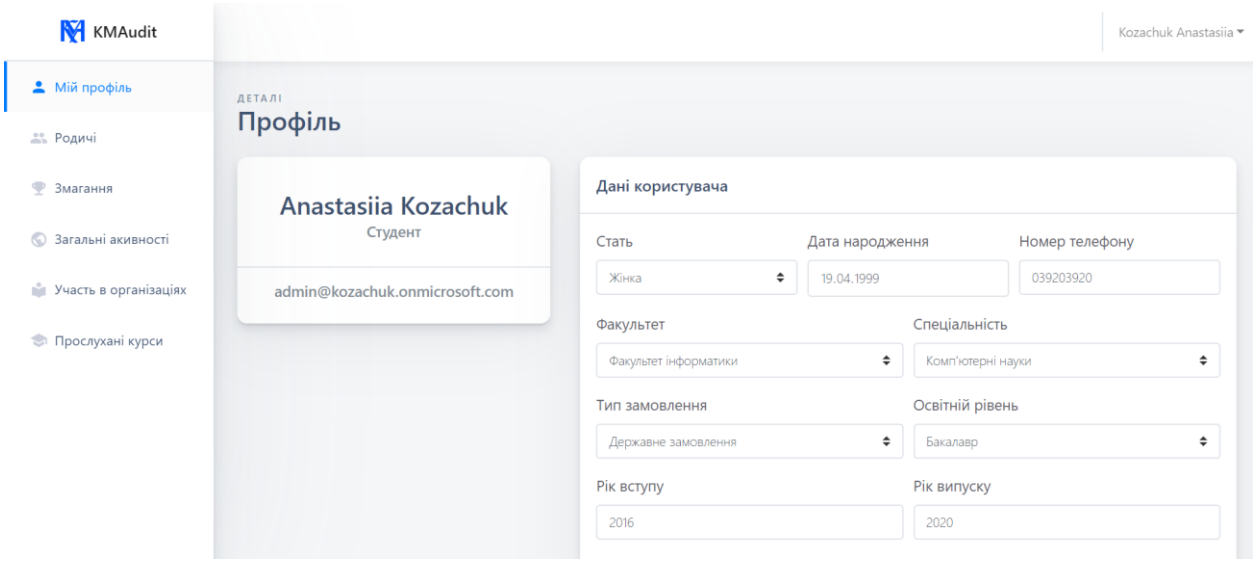

Рисунок 3.14 – Інтерфейс сторінки профілю користувача (верхня частина)

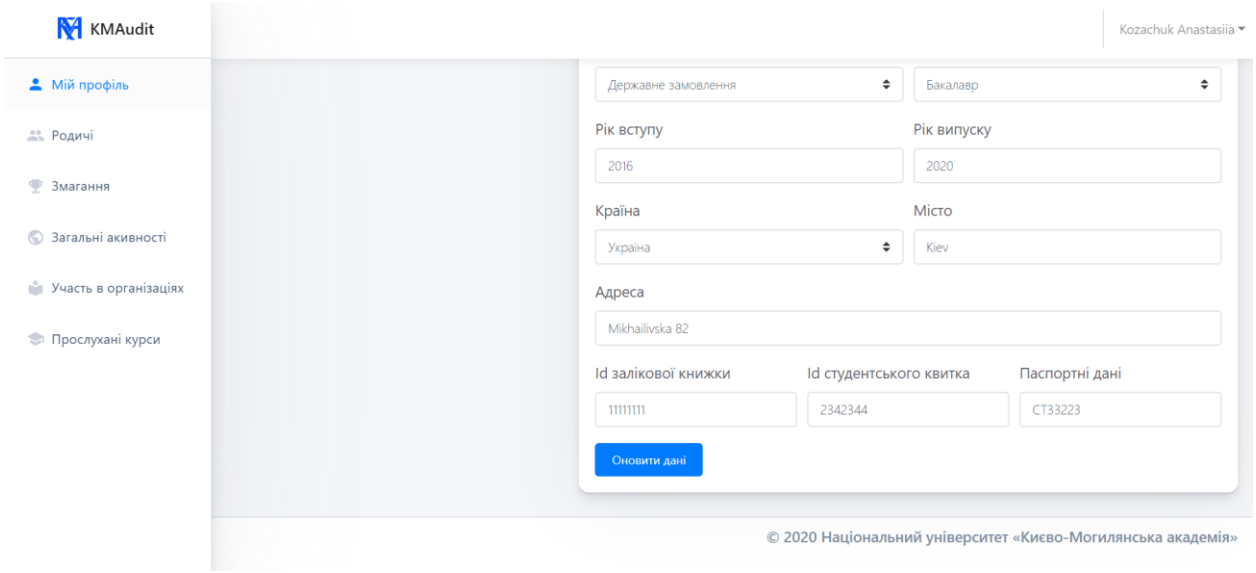

Рисунок 3.15 – Інтерфейс сторінки профілю користувача (нижня частина)

Першим кроком, який необхідно зробити, це вивчити макет і розбити його на компоненти. Кожен компонент в ідеальному варіанті повинен виконувати якусь одну задачу. Якщо з часом задачі для компонента збільшуються, його треба розбити на менші компоненти. Також необхідно визначити тип кожного компонента.

Отже даний макет був розбитий на уявні компоненти (Рисунок 3.16, Рисунок 3.17):

1) MainNavbar (область відмічена синім) – основний хедер сторінки.
2) NavbarNav (область відмічена оранжевим) – випадаючий список, який дозволяє виконувати швидку навігацію та завершити сеанс.
3) MainFooter (область відмічена фіолетовим) – основний футер сторінки.
4) MainSidebar (область відмічена зеленим) - ліва панель сторінки. На цій панелі зберігається вся навігація по веб-сторінці.
5) StudentProfile (область відмічена жовтим) – сторінка профілю.
6) UserDetails (область відмічена червоним зліва) – короткі незмінні дані про студента.
7) UserAccountDetails (область відмічена червоним справа) – всі дані про акаунт студента з можливістю оновлення.

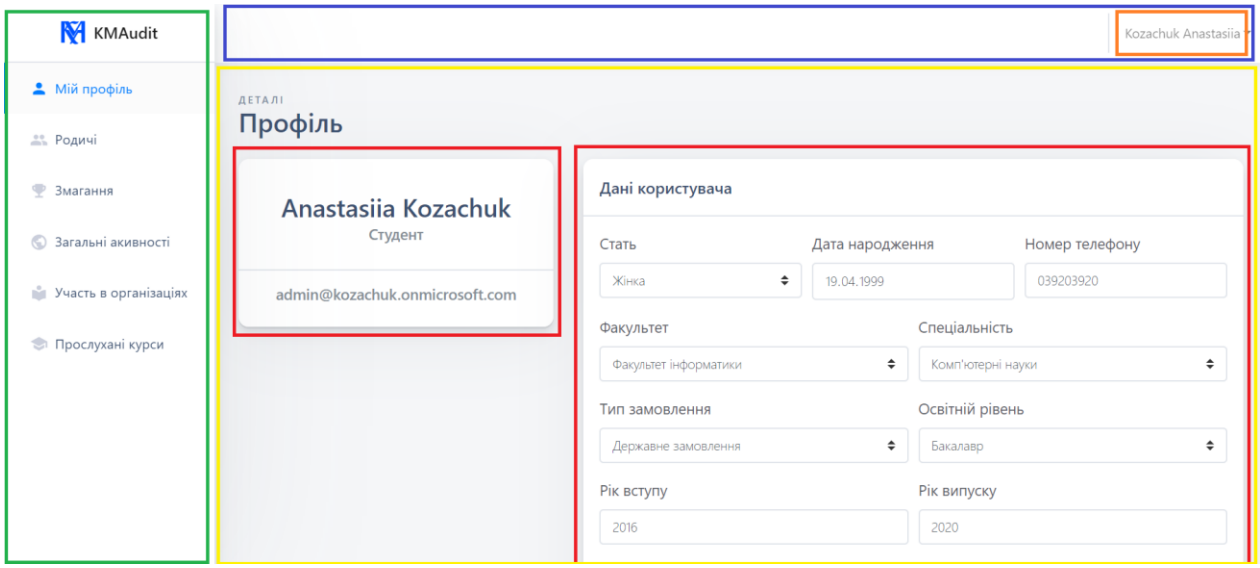

Рисунок 3.16 – Інтерфейс розбитий на компоненти (верхня частина)

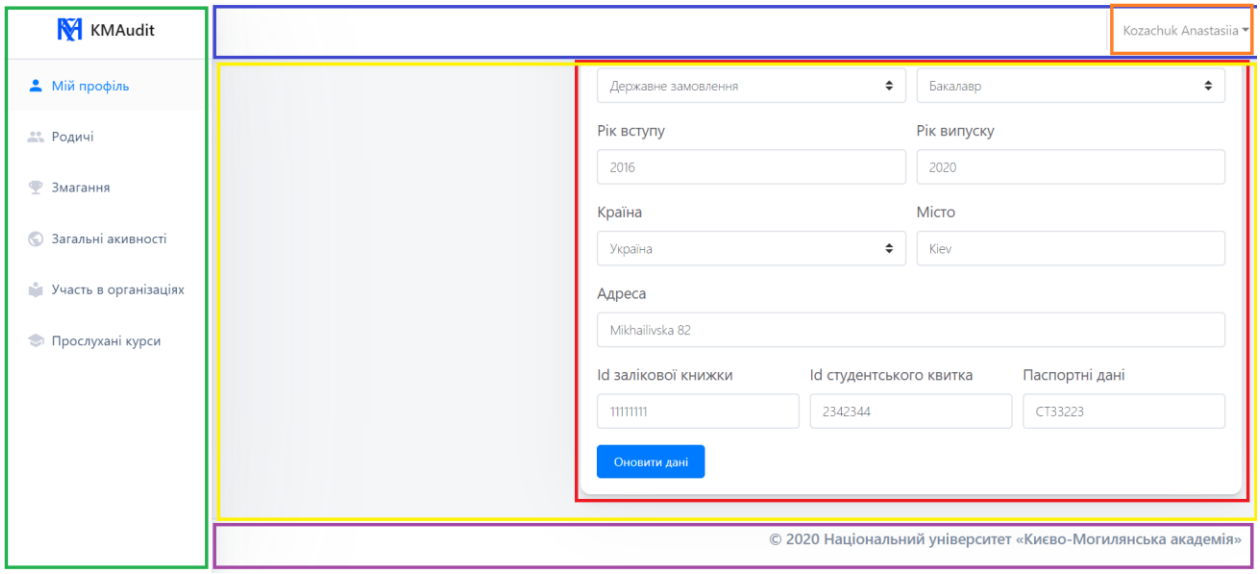

Рисунок 3.17 – Інтерфейс розбитий на компоненти (нижня частина)

Всі описані компоненти, окрім UserAccountDetails, просто відображають наявну інформацію. UserAccountDetails компонент відображає інформацію, а також дозволяє її оновити. А отже повинен зберігати стан з поточними даними про акаунт студента. Саме тому UserAccountDetails повинен бути компонентом класом, всі інші описані компоненти можуть бути компонентами функціями.

Після того як стало зрозуміло, які компоненти необхідно створювати, треба створити ієрархію цих компонентів (Рисунок 3.18).

```
App
    |- DefaultLayout
        |- MainNavbar
            |- NavbarNav
        |- MainSidebar
        |- MainFooter
    |- StudentProfile
        |- UserDetails
        |- UserAccountDetails
```

Рисунок 3.18 – Ієрархія застосунку

Відповідно до схеми, окрім зазначених вище компонентів був також вказаний компонент DefaultLayout. Він може бути перевикористаний для будь-яких інших сторінок, оскільки до нього входять 3 основних компоненти

сторінки, які для всіх сторінок будуть однаковими: верхня панель, нижня панель та бокова панель.

Після того, як було визначено компоненти та дерево компонентів можна їх реалізувати (Рисунок 3.19, Рисунок 3.20, Рисунок 3.21, Рисунок 3.22).

```
const MainNavbar = ({ layout, stickyTop }) => {
 const classes = classNames(
  "main-navbar",
  "bg-white",
  stickyTop && "sticky-top"
 );

 return (
  <div className={classes}>
   <Container className="p-0">
    <Navbar type="light" className="align-items-center flex-md-nowrap p-0 text-end">
     <NavbarNav />
    </Navbar>
   </Container>
  </div>
 );
};
```

Рисунок 3.19 – Компонент MainNavbar

```
const MainFooter = ({ contained, menuItems, copyright }) => (
 <footer className="main-footer d-flex p-2 px-3 bg-white border-top">
  <Container fluid={contained}>
   <Row>
    <span className="copyright ml-auto my-auto mr-2">{copyright}</span>
   </Row>
  </Container>
 </footer>
);
```

Рисунок 3.20 – Компонент MainFooter

```
const DefaultLayout = ({ children, noNavbar, noFooter }) => (
  <Container fluid>
    <Row>
      <MainSidebar />
      <Col
        className="main-content p-0"
        lg={{ size: 10, offset: 2 }}
        md={{ size: 9, offset: 3 }}
        sm="12"
        tag="main"
      >
        {!noNavbar && <MainNavbar />}
        {children}
        {!noFooter && <MainFooter />}
      </Col>
    </Row>
  </Container>
);
```

Рисунок 3.21 – Компонент DefaultLayout

```
const StudentProfile = () => {

 const api = useApi();
 const [student, studentLoading, studentError, getStudent] = useApiRequest({
   apiMethod: api.getStudent,
   initialValue: null,
 });

 useEffect(() => {
   getStudent({});
 }, [getStudent]);


 return (
   <Container fluid className="main-content-container px-4">
     <Row noGutters className="page-header py-4">
       <PageTitle title="Профіль" subtitle="Деталі" md="12"
                  className="ml-sm-auto mr-sm-auto"/>
     </Row>
     <Row>
       <Col lg="4">
         {student && <UserDetails userDetails={student}/>}
       </Col>
       <Col lg="8">
         {student && <UserAccountDetails userDetails={student}/>}
       </Col>
     </Row>
   </Container>
 );
};
```

Рисунок 3.22 – Компонент StudentProfile

Redux в застосунку використано для того щоб зберігати відомості про вже залогінених користувачів (зберігати інформацію про їх аксес токен). Action і reducer використовуються ті що були описані в розділі про редакс. Дана функція відповідає за створення редакс сховища. persistReducer() і persistStore() це функції з спеціальної бібліотеки redux-persist. Ця бібліотека допомагає зберігати інформацію, яка зберігається в редакс сховищі навіть після завершення роботи застосунку. При наступному запуску застосунку в редакс сховище завантажаться всі дані, які були в ньому до попереднього завершення роботи (Рисунок 3.23).

```
export default () => {
 const persistedReducer = persistReducer(persistConfig, rootReducer);
 const store = createStore(persistedReducer);
  const persistor = persistStore(store);
 return { store, persistor }
}
```

Рисунок 3.23 – Створення сховища

Наступна функція показує використання редакс сховища. Функція useSelector() повертає сховище, з якого можна дістати всі необхідні дані (Рисунок 3.24).

```
const entity = useSelector((state) => state[entityName])
```

Рисунок 3.24 – Використання сховища

Висновки

Відповідно до мети і поставлених завдань у роботі було:

1) Проаналізовано роботу з Spring Boot фреймворком для розробки веб серверу. Було розглянуто покрокову побудову архітектури Spring Boot застосунку: створення моделей та їх життєвий цикл, створення репозиторіїв, сервісів та контролерів. Також описано роботу з специфікаціями та Specification Argument Resolver бібліотекою для зручного створення специфікацій відразу в контролері.
2) Детально вивчено та описано внутрішню роботу Spring security: основні поняття, призначення authentication manager та authentication providers, роботу з UserDetailsService інтерфейсом. Розглянуто OAuth2 специфікацію та покрокову роботу OAuth2 з Microsoft Azure.
3) Розглянуто роботу з ReactJs та Redux бібліотеками для створення клієнту застосунку: описано види компонентів, їх життєвий цикл; покрокову роботу з Redux бібліотекою; проаналізовано переваги роботи з VirtualDom в ReactJs.

В результаті виконання курсової роботи було створено систему збору аналітики про студентів університету:

1) Створено серверну частину застосунку з використанням Spring Boot;
2) Застосунок зареєстровано в Microsoft Azure;
3) Налаштовано OAuth2 автентифікацію за допомогою Spring Security та Microsoft Azure OAuth2;
4) Створено клієнтську частину застосунку за допомогою ReactJS та Redux бібліотек.

Кожен з цих кроків описаний в курсовій роботі паралельно до теоретичної частини з використанням написаного для застосунку коду.

Література

1) https://www.udemy.com/course/spring-hibernate-tutorial/
2) https://www.geeksforgeeks.org/introduction-to-spring-boot/
3) https://medium.com/@jovannypcg/understanding-springs-controlleradvice-cd96a364033f
4) https://www.objectdb.com/java/jpa/entity/generated
5) https://www.youtube.com/playlist?list=PLqq-6Pq4lTTYTEooakHchTGglSvkZAjnE
6) https://tools.ietf.org/html/rfc6749
7) https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow#successful-response
8) http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/
9) https://medium.com/@gethylgeorge/how-virtual-dom-and-diffing-works-in-react-6fc805f9f84e
10) https://medium.com/@fknussel/redux-3cb5aac94a66
11) https://www.udemy.com/course/pro-react-redux/
12) https://howtodoinjava.com/hibernate/hibernate-jpa-cascade-types/
13) https://medium.com/datadriveninvestor/authentication-vs-authorization-716fea914d55
14) https://docs.oracle.com/cd/E19798-01/821-1841/bnbqm/index.html
15) https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb
16) https://medium.com/swlh/understanding-component-lifecycle-in-reactjs-ed35d76dab2e
17) https://www.javatpoint.com/spring-boot-architecture