math\_2019\_009

**Міністерство освіти і науки України**

**Чернівецький національний університет**

**імені Юрія Федьковича**

Факультет математики та інформатики

(повна назва інституту/факультету)

Кафедра математичного моделювання

(повна назва кафедри)

**Інформаційна система судових рішень**

**Дипломна робота**

**Рівень вищої освіти - другий (магістерський)**

Виконав: Махов Сергій

студент (ка) 6 курсу, групи 601

спеціальності 122 – Комп’ютерні науки\_

(назва спеціальності)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я та по-батькові)

Керівник к.ф.-м.н., асистент Перцов А.С.\_

(науковий ступінь, вчене звання, прізвище та ініціали)

**До захисту допущено:**

**Протокол засідання кафедри № 7**

від „ 4 ” грудня 2018 р.

зав. кафедри \_\_\_\_\_\_\_\_\_ доц. Піддубна Л.А.

Чернівці – 2018

# Анотація

Дипломна робота присвячена розробці системи зберігання та пошуку судових рішень судів України. У програмному продукті реалізовано схему бази даних для збереження інформації та пошукову систему.

Зміст

[Анотація 2](#_Toc532139454)

[Вступ 4](#_Toc532139455)

[Постановка задачі 6](#_Toc532139456)

[Основні принципи та підходи організації даних програмного забезпечення 8](#_Toc532139457)

[1. Типи баз даних 8](#_Toc532139458)

[2. Моделі узгодженості даних 13](#_Toc532139459)

[3. CAP Теорема 15](#_Toc532139460)

[4. Проектування бази даних 16](#_Toc532139461)

[5 Масштабування баз даних 20](#_Toc532139462)

[Характеристика розробленого програмного продукту 23](#_Toc532139463)

[1. Характеристика сховища даних 23](#_Toc532139464)

[2. Архітектура додатку та технології які він використовує 25](#_Toc532139465)

[3. Інструкція користувачу 39](#_Toc532139466)

[Висновки 42](#_Toc532139467)

[Список використаної літератури 43](#_Toc532139468)

[Додаток А: Схема бази даних 44](#_Toc532139469)

[Додаток Б: Код серверного додатку 45](#_Toc532139470)

# Вступ

Комп’ютери та інформаційні системи є важливою частиною нашого суспільства. Вони використовуються майже в кожній сфері нашого життя, включаючи наші роботи, для автоматизації послуг та продуктів; в сфері охорони здоров’я, для покращення якості лікування; в освіті, щоб допомогти дітям здобувати нові знання; у фінансовій сфері, для підтримки переказу коштів та операцій над власними рахунками; у сфері розваг, включаючи Інтернет, фільми, музику та інші мистецтва; у транспортній системі, для контролю над системами автомобілів та публічного транспорту, наприклад метро.

Технології розвиваються з феноменальним темпом і не збираються сповільнюватись. Технології вже допомогли робітникам безліч індустрій та ринків працювати якісніше. Інформаційні технології також дозволили покращити канали комунікації що також вплинуло на продуктивність роботи. У більшості сфер індустрії ІТ-технології є невід’ємною частиною процесу роботи.

Так як і у сферах нашого життя, так і у судові владі інформаційна ера розпочала революцію. Пропонуючи можливість опубліковувати величезні кількості судових рішень в мережі, судова влада може істотно збільшити власну прозорість. Саме можливість публікації рішень дасть змогу людям які не були задіяні в конкретному судовому засіданні швидко використати його як прецедент для власних потреб, як то: винести вирок, подати апеляцію. Опубліковані судові рішення є важливим базовим матеріалом для професій пов’язаних з юриспруденцією, академічних дослідників, журналістів та приватних компаній що працюють на ринку юридичної інформації.

Найбільш важливим аргументом проти публікації таких рішень в оригінальній формі є незаконне втручання в особисте життя, тобто порушення статті 8 «Хартії основних прав Європейського Союзу», оскільки зазвичай судові рішення часто містять особисту інформацію. Але цей аргумент може вдоволений якщо публікувати рішення у анонімній формі, тобто з вилученою усією особистою інформацією.

Функції які виконують публікації судових рішень вже існували навіть до появи інтернету: публічна перевірка справедливості судді та публічне знання про розвиток права. До початку поширення інтернету, перша функція, хоча й іноді не ідеально, здійснювалась за допомогою публічного оголошення рішення суду. Друга ж, здійснювалась самим судами або комерційними видавцями які пропонували друковані рішення що були важливими. З швидким розвитком інформаційних технологій дані функцій можуть бути ще більше покращені, оскільки швидкість поширення та кількість інформації значно збільшується.

В Україні, вже існує практика публікацій судових рішень, яка навіть підкріплена законом (Закон України про доступ до судових рішень) який регулює необхідність публікацій до судових рішень (ст.2 п.2) та доступ до них (ст.3, 4) за допомогою реєстру судових рішень. Однак, існуюча система є нестабільною та ненадійною, що призводить до порушення цього ж закону (ст.4, п.1).

Кроком для подолання цієї проблеми є розробка альтернативного рішення що не буде мати вищезгаданих недоліків. Саме тому розробка інформаційної системи реєстру судових рішень і є моє завдання на магістерську роботу.

# Постановка задачі

У 2002 році Верховна Рада схвалила Концепцію Загальнодержавної програми адаптації України до законодавства Європейського Союзу, де передбачалося створення національної інформаційної мережі судових рішень з метою доступу до судової практики та можливості їх публічного обговорення.

Однак перша спроба втілити її у життя була зроблена аж 18 січня 2005 року, коли у Верховній Раді був зареєстрований законопроект «Про забезпечення гласності при здійсненні господарського судочинства». Авторами законопроекту були Леонід Антоненко та Іван Крулько. Законодавча ініціатива належала Борису Тарасюку та Миколі Катеринчуку.

Законопроект отримав підтримку уряду і реалізацію закладених ідей було затверджено 4 лютого 2005 року. Майже одночасно з урядом ідея була підтримана і судовою владою. Так, постановою Президії Верховного Суду України, Президії Ради суддів України, колегії Державної судової адміністрації України «Про стан здійснення судочинства у 2004 р. і завдання на 2005 р.» від 18 лютого 2005 року, Державній судовій адміністрації України доручено запровадити реєстр судових рішень зі створенням у комп'ютерній мережі відповідної бази даних і забезпечити в установленому законом порядку доступ до неї.

31 серпня 2005 року в парламенті був зареєстрований законопроект «Про доступ до судових рішень». Законодавча ініціатива належала Василю Онопенко, текст проекту був розроблений Центром політико-правових реформ.

22 грудня 2005 року цей альтернативний проект було схвалено у другому читанні і в цілому конституційною більшістю — 303 голосами народних депутатів.

Закон набрав чинності 1 червня 2006 року. Саме з цієї дати усі ухвалені судами України процесуальні документи (крім рішень місцевих загальних судів) повинні були заноситись до реєстру і з'являтися на Інтернет-запит будь-якого користувача, а рішення місцевих загальних судів — з 1 січня 2007 року. Проте, перші документи почали з'являтися на запит користувачів з затримкою в принаймні два тижні.

Унаслідок вищезгаданих подій було розроблено Єдиний державний реєстр судових рішень, однак на сьогоднішній день він є малоефективним інструментом для пошуку та перегляду судових рішень, оскільки його пошуковий апарат є неефективним.

Метою магістерської роботи є розробка системи яка дозволить шукати та переглядати судові рішення.

Для реалізації поставленого завдання потрібно:

* знайти набори даних що представляють собою судові рішення,
* розробити концептуальну, логічну та фізичну схеми бази даних,
* визначити та реалізувати фільтри для здійснення ефективного пошуку,
* реалізувати серверну частину системи,
* реалізувати клієнтську частину системи

# Основні принципи та підходи організації даних програмного забезпечення

Сучасне інформаційне поле складається з маси подій, об’єктів та явищ. Воно охоплює такі об’єми, що без чітко працюючої визначеної системи, збереження всіх цих даних було б хаотичним та некерованим.

Визначною рисою бази даних від інших сховищ є те, що інформація в ній обов’язково піддається обробці електронними носіями з наступною систематизацією, яка підпорядкована певним правилам. Вона є важливою для неперервної взаємодії систем що містять інформацію.

### 1. Типи баз даних

Система керування базами даних (СКБД) дозволяє користувачу організовувати, зберігати та отримувані дані з комп’ютеру. Вони є шляхом комунікації з пам’яттю комп’ютера. Бази даних зіграли визначну роль в революції комп’ютерів. Перші комп’ютерні програми були розроблені у рані 50-ті роки та фокусувались здебільшого на мовах кодування та алгоритмах. У ті часи, комп’ютери були здебільшого величезними калькуляторами і дані (номери телефонів, імені) вважались залишками обробленої інформації. Комп’ютери лише починали бути комерційно доступними і коли бізнесмени почали використовувати їх для реальних задач, залишкові дані раптово почали бути важливими.

У 1960 році, Чарльз Бахман розробив першу в світі СКБД – «Інтегровану систему баз даних». У той же час, IBM не маючи бажання залишатись позаду, розробили власну СКБД, відому як IMS. Обидві систему описані як попередники навігаційних баз даних.

У середині 1960-их, коли комп’ютери стали більш швидкими та зручними у використанні та у той же час набрали популярності, безліч способів використання баз даних стало доступним. У результаті цього, люди вимагали розробки стандарту, у відповідь на це, Бахман формує групу що мала б розробити дизайн та стандарт мови яка називається Common Business Oriented Language (COBOL). Група презентувала стандарт у 1971, який також став відомим як «підхід CODASYL».

Підхід «CODASYL» був дуже складною системою і вимагав довгого тренування для користувачів. Він залежав від техніки «ручної» навігації використовуючи зв’язаний набір даних який формує величезну мережу. Пошук записів міг бути досягнутий за допомогою однією з трьох технік:

* Використовуючи первинний ключ (також відомий як CALC ключ)
* Рухаючи відношення (так звані множини) з одного запису на інший
* Скануванням усіх записів по черзі

Згодом «CODASYL» втратив популярність через простіших конкурентів що з’явилися на ринку.

Едгар Кодд який працював у той час у IBM на розробкою жорстких дисків був незадоволений відсутністю пошукового рушія у «CODASYL» та у моделі «IMS». У 1970 він написав серію статей, що описували нові шляхи побудови баз даних. Згодом, його ідеї еволюціонували в доповідь що називалась «Реляційна модель даних для великих банків даних», в якій описувались нові методи збереження даних та опрацьовування великих множин даних. Записи повинні були зберігатись не як безформний список зв’язаних записів, а у вигляді «таблиці з записами фіксованої довжини».

IBM інвестувала багато ресурсів у модель IMS та не була зацікавлена у ідеях Кодда. На щастя, деякі люди які не працювали в IBM виявили інтерес. У 1973, Майкл Стоунбрейкер та Ежен Вонг вирішили дослідити реляційні бази даних. Проект був названий «INGRES» (Interactive Graphics and Retrieval System), він показав що реляційна модель може бути ефективною та практичною. INGRES працював з мовою запитів також відомим як QUEL, який в свою чергу змусив IBM розробити більш просунуту мову - SQL у 1974. SQL швидко замінив QUEL як більш функціональна мова запитів.

Реляційні СКБД були ефективним засобом для зберігання та обробки структурованої інформації. Після росту швидкості обробки призвела до появи більшої кількості неструктурованої інформації (картини, фотографії, музика і т. д.). Неструктуровані дані є нереляційною та не мають схеми, тому реляційні СКБД просто не могли їх обробляти, оскільки були розроблені не для таких типів даних.

**NoSQL («Not only» Structured Query Language) –** з’явилась у наслідок появи інтернету та необхідності збільшення швидкості обробки неструктурованих даних. Іншими словами, NoSQL бази даних є кращими в деяких випадках відносно реляційних баз даних через власну швидкість та гнучкість. Модель NoSQL є не реляційною та використовує «розподілену» систему бази даних, ця система є швидкою, використовує ефективний метод організації даних та може обробляти великі об’єми різних типів даних.

Карло Строцці використав термін NoSQL у 1998 у якості назви для власної реляційної бази даних яка не підтримувала стандарт SQL. Ерік Еванс, працівник компанії Rackspace, повторно ввів термін NoSQL коли Йохан Оскарссон хотів організувати розмову щодо розподілених баз даних у 2007. Цей термін спробували використати для великої кількості нереляційних, розподілених баз які зазвичай не намагаються гарантувати ACID (Atomicity - атомарність, Consistency - узгодженість, Isolation – ізольованість, Durability - довговічність), що є ключовими атрибутами класичних реляційних баз, таких як Sybase, IBM DB2, MySQL, Microsoft SQL Server, Postgres SQL, Oracle RDBMS, Infromix та Oracle RDB.

Часто, NoSQL бази даних групують за шляхом який вони використовують для збереження даних, наприклад: «ключ-значення», документ-орієнтовні та графові бази даних. NoSQL підхід виріс разом з великими інтернет компаніями які працюють з задачами які зазвичай не під силу традиційним реляційним СКБД. Такі бази зазвичай дуже сильно оптимізовані для отримання та дописування інформацію та часто пропонують дуже мало можливостей окрім цих, власне, це і є плата за великі перспективи масштабування та збільшену швидкодію на конкретних моделях даних.

«Not only» (не лише) – означає що існує можливість обробляти структуровані та неструктуровані дані дуже швидко. Широке використання NoSQL можна пов’язати з послугами що надаються такими компаніями як Twitter, LinkedIn, Facebook та Google. Кожна з цих організацій зберігає та обробляє величезну кількість неструктурованих даних. Перевагами NoSQL перед SQL та реляційними СКБД є:

* Висока здатність до масштабування
* Розподілена система обчислень
* Гнучка схема
* Можуть обробляти неструктуровані та напів-структуровані дані
* Немає складних відношень

Нажаль, NoSQL також має деякі проблеми. Наприклад, деякі NoSQL бази даних можуть вимагати багато оперативної пам’яті чи процесорного часу.

Апаратне забезпечення може виходити з ладу, але NoSQL бази розроблені за допомогою розподіленої архітектури яка включає резервне збереження даних та функцій. Це досягається за допомогою використання декількох вузлів (серверів баз даних). Якщо один, чи більше, вузлів перестають працювати, то інші вузли можуть продовжувати працювати в нормальному режимі без втрати даних. Правильно використовувані NoSQL бази даних можуть надати високу швидкодію на великих навантаженнях. Загалом, існує 4 види NoSQL баз даних, кожна з них має специфічні якості та характеристики.

*Документ-орієнтовні бази –* управляє, зберігає та утримує напів-структуровані дані (також відомі як документ-орієнтовні дані). Документи можуть бути описані як незалежні одиниці, що дозволяє збільшити швидкодію і спросити поширення даних серед різних серверів. Документ-орієнтовані бази зазвичай мають потужний рушій запитів та систему контролю індексування що робить запити швидкими та простими. Прикладами таких баз є: MongoDb, Amazon Dynamo DB, Elasticsearch.

*Колонкова база даних* – база даних яка відрізняється від традиційних реляційних СКБД тим що зберігає дані як порції колонок, а не рядків. Така зміна фокусування, від рядків до колонок, дозволяє колонковим базам даних максимізувати їх швидкодію коли величезна кількість даних збережена в єдиній колонці. Прикладами колонкових баз є: Cloudera, Cassandra та HBase.

*Сховище пар типу «ключ-значення»* - такі бази даних є корисними для зберігання даних про корзину чи збереження профілів користувачів. Увесь доступ до бази можливий через використання первинного ключа. Зазвичай, в таких базах відсутня фіксована схема чи модель даних. Такі бази не є ефективними коли в даних присутні складні зв’язки або ж дані необхідно отримувати не за первинним ключем. Прикладами таких баз є: Riak, Berkeley DB та Aerospike.

*Графові бази даних –* системи навігації, диспетчерські системи та соціальні мережі є головними користувачами графових баз даних. Такі бази покладаються на теорію графів та чудово працюють з даними які можуть бути відображені як графи. Вони відрізняються від традиційних реляційних баз тим, що зберігають зв’язків як «актуальні зв’язків». Такий тип сховища зв’язків призводить до меншої кількості розривів у схемі що змінюється та актуальною базою. Прикладами графової бази може слугувати: Neo4j, GraphBase та Titan.

**Поліглотне збереження** - це концепт збереження даних, що походить від ідеї «поліглотної розробки», розроблений Нілом Фордом у 2006 році. Оригінальна ідея пропонувала розробляти додатки використовуючи декілька мов програмування, розуміючи що кожна мова здатна вирішувати відповідну задачу просто, коли інша мова може мати складнощі.

Є безліч баз даних які можуть вирішувати різні задачі. Використовуючи одну базу для того щоб задовільнити всі вимоги може призвести до неефективного результату. Наприклад, реляційні бази даних чудово керують зв’язками що існують між різними таблицями даних, це є непогано до тих пір поки даних не дуже велика кількість, але стає проблемою коли їх стає все більше. Графова база може вирішувати задачу зв’язків на великих об’ємах даних, але страждає відсутністю механізму транзакцій які існують у реляційних базах. Натомість, документ-орієнтовна база може бути використана для збереження неструктурованої інформації для конкретної задачі. Ці та інші задачі можуть бути вирішені різними базами даних, але в контексті одного додатку.

**UnQL (Unstructured Query Language) –** у 2011, розробники баз даних розпочали працювати над специфікацією мови запитів для NoSQL баз. Вона побудована для запитів колекцій документів з визначеними полями напротивагу рядкам та колонкам. UnQL претендує бути надмножиною над SQL, в якому SQL це дуже обмежений тип UnQL для якого запити завжди повертають ті ж поля, номери, імена та типи. Тим не менш, UnQL не описує мову опису даних (Data Definition Language, DDL).

На сьогоднішній день бази даних починають працювати з навіть складнішою логікою. Ще одна особливість яка починає зароджуватись – ідея розподілу розміщення. Вона дозволить базі даних бути більше аніж в одному місці і отримувати запити як цілісна одиниця. Такі бази називаються «розподіленими» або ж федеративними. Наприклад, частина бази даних може знаходитись у Нью-Йорку, а інша частина у Бостоні і запит який вимагає порахувати всіх клієнтів буде виконуватись на обох частинах. Такий розподіл став можливий завдяки збільшенню швидкості мережі.

Майбутнє баз даних надихає можливостями, величезні системи з’являються на ринку та вимагають новаторських підходів у збереженні та обробці даних. Мобільні бази даних вже використовуються у продуктах які вже на ринку. Обробка розподілених транзакцій стає нормою для бізнесу в багатьох сферах.

### 2. Моделі узгодженості даних

У NoSQL базах даних досить модель узгодженості даних дуже сильно відрізняється від тих які використовуються в реляційних базах (настільки ж сильно, наскільки й в інших NoSQL базах).

Дві найбільш популярні моделі відомі за акронімами ACID та BASE. Досить часто їх протиставлять одну одній, однак важливо знати що кожна модель має свої недоліки та переваги – і ні одна з них не є ідеальним інструментом.

**ACID**

Багато розробників хто працював з реляційними базами даних вже знайомі з ACID транзакціями, які були стандартом деякий час. Ідея ACID – гарантувати безпечне оточення в якому клієнти можуть працювати з даними. Акронім ACID розшифровується як:

* Atomic (Атомарний) – всі операції транзакції закінчуються успіхом або ж кожна операція відміняється.
* Consistent (узгоджений) – по завершенню транзакції дані у базі даних відповідає обмеженням цілісності.
* Isolated (ізольованість) – транзакції ізольовані одна від одної. Постійний доступ до даних регулюються базою даних, так що транзакції з’являються послідовно.
* Durable (довговічний) – результат застосування транзакції є перманентним, навіть у випадку наявності помилок.

Властивості ACID означають, що якщо транзакція виконана, то дані які вона принесла є узгодженими. Більшість графових баз даних використовують ACID модель для того щоб бути впевненим в тому що дані записані та узгоджені.

**BASE**

Для багатьох предметних областей, ACID модель є занадто песимістичною (тобто більше хвилювань за узгодженість даних) аніж того насправді вимагає задача.

У NoSQL баз даних, ACID транзакції є менш популярними, причиною тому є те що деякі реалізації баз пожертвували вимогами до негайної узгодженості та порядку виконання задля отримання інших переваг, наприклад масштабування чи стійкості до відмов. Акронім BASE розшифровується наступним чином:

* Basic Availability (базова доступність) – база даних працює більшість часу
* Soft-state (гнучкий стан) – сховище не повинно бути узгодженим на запис.
* Eventual Consistency (Узгодженість з часом) – сховища стають узгодженими з часом (наприклад перед запитом на читання).

BASE властивості є набагато слабкішими аніж гарантії ACID, однак не існує однозначної відповідності між двома моделями узгодженості.

BASE база цінує доступність (так як це важливо для масштабування), але вона не гарантує цілісність копійованих даних під час запису. В той же час, BASE модель пропонує набагато менш строгу модель перестраховки аніж ACID: дані будуть узгоджені в майбутньому або ж під час читання.

BASE модель узгодженості зазвичай використовується документ-орієнтованими базами даних.

Немає однозначної відповіді яку модель узгодженості слід використовувати для усіх додатків BASE чи ACID. Розробникам та архітекторам слід обирати необхідну модель базуючись на конкретній задачі, а не на тому що зараз в трендах чи що використовували до того.

### 3. CAP Теорема

В минулому, коли необхідно було зберегти більше даних або ж збільшити можливості їх обробки найбільш розповсюдженим способом було вертикальне масштабування (тобто використати більш потужні комп’ютери) або ж оптимізувати кодову базу. Тим не менш, з розвитком паралельних обчислень у розподілених системах стає більш звично горизонтальне масштабування, тобто мати більшу кількість комп’ютерів що роблять ту ж задачу паралельно. Уже існують безліч інструментів для обробки даних. Однак, щоб ефективно обрати необхідний інструмент слід знати та розуміти ідею CAP теорему.

**CAP Теорема-** це твердження що у розподіленій базі даних можна мати лише 2 з 3 властивостей: узгодженість, доступність та стійкість до розділення мережі.

**Partition tolerance (стійкість до розділення мережі) –** ця властивість говорить про те, що система продовжує працювати незважаючи на кількість повідомлень між вузлами що не була доставлена по мережі. Система що стійка до розділення мережі може витримати будь-яку кількість збоїв мережі які не приводять до збою мережі в цілому. Записи даних ефективно копіюються через комбінації вузлів та мереж щоб підтримувати систему в працюючому стані незважаючи на тимчасові несправності. Працюючи з сучасними розподіленими системами, стійкість до розподілення мережі це не опція, а необхідність. А отже слід обирати між узгодженістю та доступністю.

**Consistency (узгодженість) –** ця властивість говорить про те що всі вузли надають однакові дані в один момент часу. Іншими словами, виконання операції читання завжди повертає значення останньої операції запису на усіх вузлах системи. Система називається цілісною якщо транзакція починається в системі з цілісним станом та закінчується в системі з цілісним станом. У цій моделі, система переходить у нецілісний стан під час транзакції, але вся транзакція відкочується якщо відбувається помилка під час будь-якого етапу процесу. Тим не менш, для підтримки узгодженості вузлам може знадобитися час щоб оновитись, а отже вони не будуть доступні по мережі.

**Availability (доступність) –** ця властивість говорить про те що кожен запит гарантовано отримає відповідь (будь-то успіх чи помилка). Досягти доступності в розподіленій системі вимагає щоб система залишалась робочою 100% часу. Кожен клієнт повинен отримати відповідь незважаючи на стан будь-якого вузла у системі. А отже, такі бази даних не залежні від часу, оскільки їх вузли повинні бути в мережі постійно.

Розподілені системи дозволяють досягти рівня обчислювальної потужності та доступності яка просто не була можлива у минулому. Сьогоденні системи мають більшу швидкодію, менші затримки та близько 100% час роботи у дата центрах які розкидані по усій земній кулі. Більш того, додатки працюють на обладнанні яке можна з легкістю придбати та налаштувати за доступною ціною. Тим не менш за все є своя ціна, розподілені системи є більш складними ніж їх нерозподілені побратими. Розуміння складності розподілених систем, роблячи правильні вибори зважаючи на CAP теорему є ключовими факторами при розробці горизонтально масштабованих систем.

### 4. Проектування бази даних

**Проектування бази даних –** організація даних згідно з моделлю бази даних. Розробник визначає які дані повинні бути збережені та як дані відносяться один до одного. Проектування баз даних включає в себе класифікацію даних та визначення взаємовідносин. Теоретичне відображення даних називається онтологією.

Основними задачами проектування баз даних є:

* Забезпечення збереження всієї необхідної інформації
* Забезпечення можливості отримання даних
* Зменшення надмірності та дублювання даних
* Забезпечення цілісності даних

Для визначення даних які мають бути збережені в базі даних, зазвичай залучають людину яка займається проектуванням бази даних, це швидше розробник що має експертизу в області теорії баз даних, більше аніж знання про предметну область данні якої будуть зберігатись. Більш того, дані які будуть зберігатись в базі даних повинні бути визначені у взаємодії з людиною яка має експертизу в даній предметній області та хто знає, які дані повинні бути присутні в системі.

Цей процес зазвичай називають «аналізом вимог», він вимагає від розробника бази даних навичку виявляти необхідну інформацію в тих хто володіє знанням про предметну область. Це необхідно, оскільки ті хто володіє предметною областю зазвичай не можуть чітко виразити які є системні вимоги для бази даних оскільки для них є незвичним думати в термінах дискретних даних які мають бути збережені. Також, дані які мають зберігатись можуть бути визначенні в специфікації вимог.

Коли проектувальник бази даних обізнаний про дані які мають зберігатись в базі даних, він повинен визначити залежності між даними. Іноді коли дані змінюються можуть змінитись і дані що не є видимими. Наприклад, нехай в списку імен з адресами декілька людей можуть мати однакову адресу, але одна людина не може мати більш аніж одну адреси – адреса залежна від імені. За допомогою імені можна однозначно визначити адресу, але не навпаки – за адресом не можна однозначно визначити ім’я, оскільки декілька людей можуть відноситись до однієї адреси. Оскільки адреса визначається по імені то вона вважається залежною від імені.

Коли відношення та залежності між частинами інформації визначено стає можливо організувати дані в логічні структури які можуть бути зіставлені в об’єкти сховища що підтримуються СКБД. У випадку реляційної бази даних, такими об’єктами являються таблиці де дані збережені у вигляді рядків з колонками. У об’єктно-орієнтовній базі даних об’єкти збереження прямо відповідають до об’єктів що використовуються в об’єктно-орієнтовних мовах програмування. Відношення можуть бути описані як атрибути класів об’єкту або ж як методи які оперують об’єктами іншого класу.

Кожна таблиця може відповідати реалізації або ж логічного об’єкту або відношенням що об’єднує один чи більше прикладів одного чи багатьох логічних об’єктів. Зв’язок між таблицями може бути збережений як посилання що пов’язує підпорядковані таблиці з батьківськими.

**Концептуальна схема** **–** опис інформаційних потреб бізнесу на високому рівні абстракції. Зазвичай включає лише основні ідеї та зв’язки між ними. Чернова модель, яка недостатньо деталізована для того щоб побудувати справжню базу даних. Вона описує семантику організації і відображає серію припущень про її природу. Конкретно, вона описує речі що є значимими для організації (сутності), про які вона намагається збирати інформацію, характеристики атрибутів сутностей та асоціації між парами цих сутностей.

Оскільки концептуальна схеми відображає семантику організації та не дизайн бази даних, вона може існувати на різних рівнях абстракцій.

**Логічна схема** – модель даних для специфічної задачі об’єктної області описана незалежно від конкретної СКБД, але в термінах структур даних, таких як реляційні таблиці та колонки, об’єктно орієнтовані класи або ж теги XML. Що і відрізняє її від концептуальної схеми, яка описує семантику організації даних без зв’язків з якими-небудь технологіями.

Логічна схема відображає абстрактну структуру предметної області. Досить часто вона є схематичною за природою і часто використовується в бізнес процесах що прагнуть охопити речі які є важливими для організації. Після валідації та підтвердження, логічна схема може стати базою для фізичної схеми.

Логічна схема повинна бути побудована на базі попередньої концептуальної схеми, яка за своїм походженням описує семантику та інформаційний контекст, це має також відображатись в логічній схемі.

**Фізична схема** **–** відображення дизайну даних яка реалізована, або призначена для реалізації у системі керування базами даних. У життєвому циклу проекту вона зазвичай походить від логічної схеми, також може бути отримана внаслідок «реверс-інженерінгу» від існуючої реалізації бази даних. Закінчена фізична схема буде містити всі елементи бази даних для створення зв’язків між таблицями або ж для покращення швидкодії, наприклад індекси, визначення обмежень, розділені таблиці чи кластери. Аналіз зазвичай може використати фізичну схему для оцінки вимог до сховища. Також схема може містити специфічні деталі до розміщення даної системи бази даних.

**Проектування ділиться на три етапи:**

*Концептуальне (інфологічне) проектування* – побудова семантичної схеми предметної області, тобто інформаційної моделі найвищого рівня абстракції. Така модель створюється без орієнтації на яку-небудь конкретну систему керування базами даних (СКБД) чи модель даних. Терміни «семантична модель», «концептуальна схема» та «інфологічна модель» є синонімами.

Частіше за все концептуальна схема містить в собі:

* Опис інформаційних об’єктів чи понять предметної області та зв’язків між ними.
* Опис обмежень цілісності, тобто вимог до допустимих значень даних і до зв’язків між ними.

*Логічне (даталогічне) проектування* – створення схеми бази даних на основі конкретної схемі даних, наприклад, реляційної схеми даних. Для реляційної схеми даних логічна схема – набір схем відношень, зазвичай з вказанням первинних ключів, а також зв’язків між відношеннями, що являють собою зовнішні ключі.

Перетворення концептуальної схеми в логічну схему, як правило, виконується за формальними правилами. Цей етап може бути в більшій мірі оптимізований.

На етапі логічного проектування враховується специфіка конкретної моделі даних, але може не враховуватись специфіка конкретної системи керування базами даних.

*Фізичне проектування –* створення схеми бази даних для конкретної СКБД. Специфіка СКБД може включати в себе обмеження на іменування об’єктів бази даних, обмеження на підтримуванні формати і т. д. Окрім того, специфіка конкретної СКБД при фізичному проектуванні включає в себе вибір рішень пов’язаних з фізичним середовищем збереженням даних (розподілення БД по файлам та пристроям, вибір методів керування дисковою пам’яттю), створення індексів і т. д.

Результатом фізичного проектування на мові SQL зазвичай є сценарій що створює необхідні об’єкти у СКБД.

### 5 Масштабування баз даних

**Масштабування баз даних** – одна з найбільш складних задач під час розвитку системи.

В основі масштабування даних лежить той же принцип, що і в основі масштабування Web-додатків. Це розподілення даних на групи та виділення їх на окремі сервери. Існує дві основні стратегії – реплікація та шардінг.

*Реплікація –* реплікація дозволяє зробити повний дублікат бази даних. Так замість одного сервера буде використовуватись два та більше.

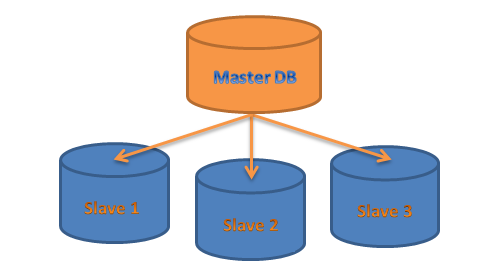
Найчастіше використовується стратегія «хазяїн-невільник». Вона передбачає ролі оригінальної бази даних та її копій.

* *Хазяїн –* це основний сервер бази даних, куди поступають усі дані. Всі зміни в даних (додавання, оновлення, видалення) мають відбуватись на цьому сервері.
* *Невільник –* це допоміжний сервер бази даних, які копіює всі дані з «хазяїну». З цього серверу слід читати дані. Такі серверів може бути декілька.

Реплікація дозволяє використовувати два та більше однакових серверів замість одного. Операцій читання даних часто набагато більше, аніж операцій зміни даних. Саме тому, реплікація дозволяє розгрузити основний сервер за рахунок переносу операцій читання на «невільників».

Також існує стратегія «багато- хазяїв» - цей метод передбачає те що декілька фізичних копій баз можуть обробити запити на зміни. Недоліком такої стратегії є значне збільшення складності системи.

Варто зазначити, що реплікація сама по собі не дуже зручний механізм для масштабування. Причиною тому є втрата синхронізації даних і затримки в копіювання з «хазяїна» на «невільника». Зате це чудовий засіб для забезпечення стійкості до відмов. В разі поломки «хазяїна» можна швидко зробити «хазяїном» будь якого «невільника».

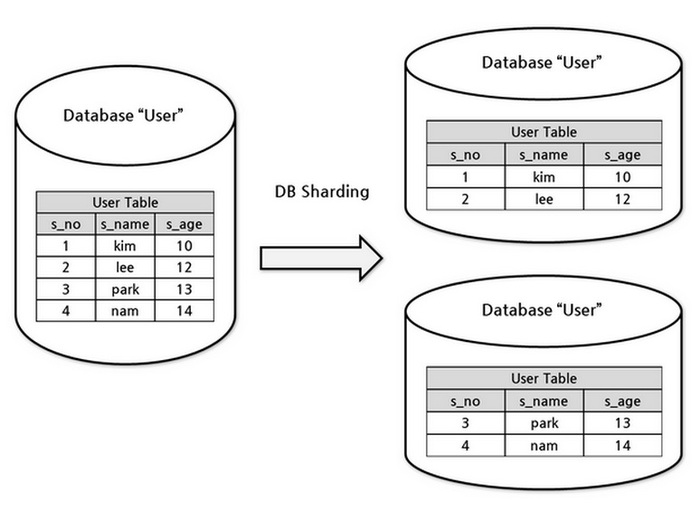


*Рис.1 Реплікація бази даних за стратегією «хазяїн-невільник»*

*Шардінг (іноді шардування) –* це інша техніка масштабування роботи з інформацією. Його ідеєю є розділення (секціонування) бази даних на окремі частини так, щоб кожну з них можна було винести на окремий сервер. Цей процес залежить від структури бази даних і виконується прямо в системі.

*Вертикальний шардінг* – це виділення таблиці чи групи таблиць на окремий сервер.

*Горизонтальний шардінг –* це виділення однієї таблиці на різні сервери. Його слід використовувати для великих таблиць які не поміщаються на одному сервері.



*Рис. 2. Приклад горизонтального шардінгу*

Зазвичай шардінг та реплікацію використовуються разом для ефективного досягнення масштабування та стійкості до відмов.

Для збереження та обробки даних в реєстрі судових рішень використана реляційна СКБД PostgreSQL та нереляційний пошуковий рушій ElasticSearch. У той час як PostgreSQL обробляє збереження списків метаданих: типів рішень, списків судів, типів справ і т. д. ElasticSearch зберігає власне проіндексовані дані документів разом з їх наповненням та метаданими: типи рішень, справ, тексти рішень, дати ухвали та надходження. Таке рішення дозволяє ефективно будувати пошукові фільтри та здійснювати пошук за текстом рішень.

# Характеристика розробленого програмного продукту

### 1. Характеристика сховища даних

Згідно схеми бази даних реєстру судових рішень що зображений у Додатку А, схема складається з наступних таблиць:

* cause\_category
* instance
* judgment\_form
* justice\_kind
* court
* document

Проведемо огляд реляційних таблиць.

Таблиця «cause\_category» що відображає категорію справи складається з наступних полів: код категорії та її імені. Дана таблиця використовується для наданням можливості обирати значення пошуку за відповідним критерієм та перевірки узгодженості типу справи якому належить документ.

Наступна таблиця - «instance» відображає доступні судові інстанції. Ця таблиця включає наступні поля: код інстанції та її назву. Таблиця також використовується для надання списку можливих значень фільтрів та перевірки узгодженості документів.

Розглянемо таблицю «judgment\_form» вона є об’єктом що є проекцією форми судових рішень, її колонками є: ідентифікатор форми рішення та його назву. Дана таблиця визначає список можливих значень для фільтру за формами судових рішень і визначенням узгодженості документів судових рішень.

Таблиця «justice\_kind» проекція форми судочинств що складається з наступних полів: код форми та її назва. Дана проекція надає список доступних значень для фільтрів та узгодженості даних.

Об’єкт «court» відображає список доступних судів. Її колонками є: код суду, назва суду та коду інстанції на яку посилається суд. Дана табличка слугує джерелом значень фільтру та таблицею на яке посилаються інші таблиці.

Таблиці описані вище здебільшого слугують для постачання даних до фільтрів та слідкуванням за узгодженістю даних.

Лідуючою таблицею у додатку слугує таблиця «document» вона зберігає структуровану частину документа яка може бути оброблена у реляційній базі. Властивостями даного об’єкта є: ідентифікатор документу, ідентифікатор суду який опублікував даний документ, ідентифікатор форми судового рішення, ідентифікатор форми судочинства якому належить даний документ, ідентифікатор категорії справи до якої належить цей документ, номер справи до якої прикріплений документ, ПІБ судді якому належить цей документ, дати ухвали, надходження та публікації та власне посилання на файл документу який містить його текст та власне флаг який позначає чи індексований вміст даного документу в нереляційному сховищі. Дана таблиця слугує реляційним відображенням реєстру судових рішень. Її дані будуть використанні для подальшого індексування у нереляційному сховищі додатку — ElasticSearch.

Для збереження структурованих даних додатку було обрано використовувати реляційну СКБД PostgreSQL. Аргументами в її користь були:

* Дана СКБД здатна зберігати велику кількість інформації без значного розміру росту місця що займає база та без втрат швидкодії.
* Вона є безоплатна у використанні.
* Вона оновлюється та підтримується розробниками.
* Наразі існує безліч бібліотек що дозволяють працювати з даною СКБД на різних мовах програмування.
* PostgreSQL для ОС Windows поставляється разом з зручним клієнтським інтерфейсом pgAdmin

Розглянемо нереляційне сховище додатку.

Наразі створено єдиний індекс з іменем «newdocument» який зберігає документи типу «document». Окрім властивостей які наслідуються з таблиці «document» з реляційного сховища також присутня властивість «контент документу» у даній властивості зберігається власне текст прийнятого рішення. Також варто відзначити що до даного поля використаний аналізатор української мови який дозволяє здійснювати повнотекстовий пошук за вмістом документу.

У якості сховища для неструктурованої інформації документів використовується Elasticsearch. Обґрунтуванням його вибору у якості нереляційного сховища слугували наступні аргументи:

* Дане програмне рішення не потребує оплати за власне використання
* Elasticsearch може працювати з великою кількістю документів без втрати швидкодії
* Це сховище, у разі необхідності, має великий потенціал для масштабування
* Має широкий спектр засобів для здійснення пошуку за різними критеріями, в тому числі повнотекстовий пошук

### 2. Архітектура додатку та технології які він використовує

Згідно вимог поставлених законом України про доступ до судових рішень, додаток повинен підпадати під наступні характеристики:

* Здатність до масштабування сховища документів
* Стійкість до відмов у разі стресових навантажень
* Забезпечувати зручний пошук серед існуючих судових рішень
* Мати високу швидкодію

Для забезпечення характеристик вище було вирішено реалізувати веб додаток який складається з двох проектів.

Перший відповідає за користувацький інтерфейс та відображення результатів пошуку у браузері. Для імплементації даної програми було вирішено використати пару зв’язку з бібліотек React та Redux. React – бібліотека що розробляється компанією Facebook для реалізації компонентного підходу у побудові інтерфейсів. Дана бібліотека дозволяє розробляти великі односторінкові додатки. Головними ідеями цієї бібліотеки є простота у використанні та швидкість. У той же час React володіє наступними перевагами:

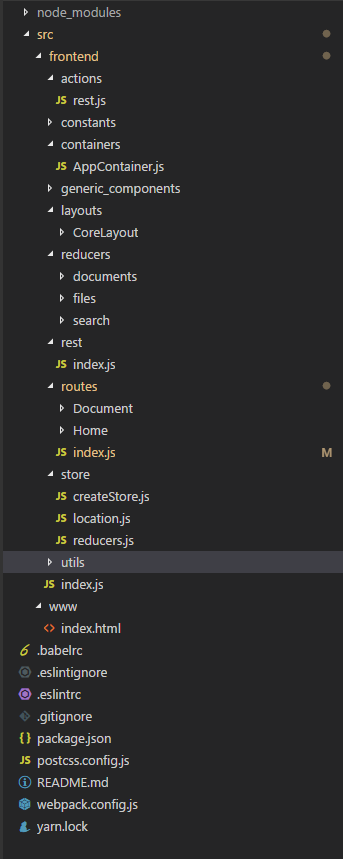
* Однонаправленість даних – тобто всі параметри компонентів можуть пропагуватися лише їх від батьківських компонентів. У той же час дочірні компоненти можуть викликати функції-дії які передаються батьківськими компонентами.
* Віртуальна об’єктна модель документу – оптимізація швидкодії при перемальовуванні сторінки. Саме за допомогою цієї технології браузер буде перемальовувати лише частини документів які змінюються.
* Мова виразів JSX – за допомогою цієї мови розробник може просто описувати ієрархії та поведінку компонентів.

Redux – менеджер стану для Javascript додатків. Дана бібліотека розроблена компанією Facebook та її ціль у допомозі розробникам створювати додатки яким необхідний стан який можна легко тестувати та підтримувати цілісність даних, Redux реалізує однонаправлену архітектуру Flux. Використання даної бібліотеки дозволяє розділити стан від відображення що робить його агностичним до бібліотек які керують відображенням. Бібліотека Redux може бути описана трьома фундаментальними постулатами:

* Єдине джерело даних – стан зберігається всередині єдиного об’єктного дерева всередині сховища Redux.
* Стан є незмінним – тобто бажання змінити будь які дані в стані додатку виливаються в створення нового стану.
* Усі зміни викликаються чистими функціями – для зміни стану додатку під впливом подій розробник повинен створити чисту функцію.

Також для збірки проекту використовується npm та webpack.

Розглянемо структуру даного під проекту.



*Рис. 3. Структура підпроекту клієнтської частини*

Даний підпроект є веб-порталом, а отже його реалізацією є комбінацією HTML, CSS та Javascript. Компоненти архітектури проекту рознесені по різним директоріям для швидкого доступу до необхідного шару функціональності. Розглянемо кожен шар окремо:

* Actions – це шар архітектури що відповідає за публікацію подій у додатку. Тобто у файлах цього шару зберігається функції які запускають події у нашому проекті. Наприклад: завантажити документ, шукати документи і т.д.
* Constants – директорія зберігає перелік констант у проекті. Здебільшого це види подій що можуть статись.
* Containers – зберігає контейнери додатку що є зв’язуючим мостом між React компонентами та Redux станом.
* Layouts – зберігає макети розмітки що використовуються у додатку.
* Reducers – чисті функції бібліотеки Redux які відповідають за зміну стану додатку.
* Rest – шар комунікації між клієнтською частиною додатку та серверною, саме він слугує мостом між двома частинами проекту.
* Routes - шар опису роутів проекту. Кожен роут складається щонайменше з трьох частин: декларативного опису роуту (компонента яка слугує його відображенням та опціональний шлях для доступу до компоненти), контейнер-компоненти яка зв’язує стан додатку з третьою складовою роуту — React-компонентою яка відображає конкретний роут.
* Store – містить опис створення менджеру стану додатку, цей шар комбінує в собі використання функцій шару reducers та додання до менджеру спеціальних плагінів — middleware що слугують для допоміжної функціональності, як то створення можливості для асинхронного публікування подій.

Окрім коду який відповідає за бізнес логіку також присутня директорія node\_modules, у якій зберігаються завантажені бібліотеки що використовується у підпроекті. Також присутні конфігураційні файли, а саме: package.json для опису метаінформації, залежностей проекту та команд для його збірки та запуску, окрім того, webpack.config.js описує конфігурацію збірника проекту.

Розглянемо структуру типового React-компоненту:

import React from 'react';

import { List, ListItem, ListDivider } from 'react-toolbox/lib/list';

import { AppBar, IconButton } from 'react-toolbox';

import { Layout, Panel, Sidebar } from 'react-toolbox';

import Iframe from 'react-iframe'

class DocumentView extends React.Component {

state = {

drawerActive: false,

drawerPinned: false,

sidebarPinned: false

};

goHome = () => {

history.back();

}

toggleSidebar = () => {

this.setState({ sidebarPinned: !this.state.sidebarPinned });

};

componentDidMount() {

this.props.findDocumentFile(this.props.document.get("id"))

}

render() {

const document = this.props.document.toJS()

const unkown = "Невідомо"

const file = this.props.documentFile ? this.props.documentFile.toJS() : null;

return (

<Layout>

<Panel>

<AppBar leftIcon="arrow\_back" rightIcon='info' onRightIconClick={this.toggleSidebar} onLeftIconClick={this.goHome} title={document.causeNumber ?document.causeNumber : unkown}/>

<div style={{ flex: 1, overflowY: 'auto', padding: '1.8rem' }}>

{file && file.content ?

<div dangerouslySetInnerHTML={{\_\_html:`${file.content}`}}></div> : ""}

</div>

</Panel>

<Sidebar pinned={this.state.sidebarPinned} width={11}>

<div><IconButton icon='close' onClick={this.toggleSidebar} /></div>

</Sidebar>

</Layout>

)

}

}

export default DocumentView

Починаючи з першого рядку файлу можемо бачити імпорти необхідних компонентів та функцій з інших бібліотек та файлів. Слідом за ними власне починається опис нашого компоненту у вигляді Javascript класу. Для визначення React-компоненту Javascript-клас має відповідати двум вимогам: наслідуватися від React.Component та мати метод render() що повертає JSX-розмітку яка є відображенням данної компоненти, саме її результат побачить користувач. Слідом за декларацією класу можемо бачити опис стану компоненту, у ньому відображається внутрішні дані компоненту які будуть слугувати для його відображення, у нашому випадку це змінні які слугують індикатором відмальовування сайдбару, слідом знаходять члени класу-методи які відповідають за навігацію та зміну стану компоненти. Далі можемо бачити функцію componentDidMount() - ця функція є особлива у бібліотеці React, вона автоматично викликається під одного з етапів життя компоненту, такі функції називаються “функціями життєвого циклу”. Останньою є функція render() яка у даному випадку відображає вміст документу (якщо він присутній у стані додатку).

Перейдемо до функцій що змінюють стан додатку, типова функція виглядає подібно до наступної:

import { DOCUMENTS\_RETREIVED, SEARCH\_STARTED, SEARCH\_ERROR } from '../../constants/action.types'

import Immutable from 'immutable'

const initialState = Immutable.fromJS({})

export default (state = initialState, action) => {

switch (action.type) {

case SEARCH\_STARTED:

return state.set("isSearching", true)

case DOCUMENTS\_RETREIVED:

let newState = initialState

newState = newState.set("searchInfo", Immutable.fromJS(action.searchInfo))

return newState.set("isSearching", false)

case SEARCH\_ERROR:

return state.set("isSearching", false)

default:

return state

}

}

Перейдемо одразу до функції, її сигнатура вимагає мати два параметри, теперішній стан додатку (або його частини) та подія яка сталась. Базуючись на типі події що сталась та теперішньому стані функція будує новий стан додатку. Завдяки використанню бібліотеки Immutable.js розробник може бути впевненим що старий стан ніколи не буде змінений, натомість буде створений новий.

Розглянемо контейнери які слугують мостом для передачі стану Redux до React компонентів:

import {searchDocuments } from '../../../actions/rest'

import HomeView from '../components/HomeView'

import { connect } from 'react-redux'

const mapStateToProps = (store) => ({

documents: store.documents,

searchInfo: store.search.get("searchInfo"),

isSearching: store.search.get("isSearching", false)

})

const mapDispatchToProps = (dispatch) => ({

search: (query) => dispatch(searchDocuments(query))

})

export default connect(mapStateToProps, mapDispatchToProps)(HomeView)

Даний файл має дві функції mapStateToProps яка ставить у відповідність стан Redux стану до властивостей React-компоненту, mapDispatchToProps у свою чергу створює відображення функцій що створюють події до властивостей того ж React-компоненту. В кінці файлу знаходиться виклик функції яка власне пов’язує три складові компоненти та породжує нову компоненту яка і буде використана у об’єктному дереві порталу.

Другим додатком є сервер що надає REST-сервіси для клієнтської частини проекту та індексування документів. Даний сервер написаний на мові програмування Java з використанням засобів фреймворку Spring Boot, для міграцій бази даних використовується бібліотека Liquibase.

Для розробки даної підсистеми було вирішено використовувати Java версії 8, головними відмінностями якої є:

• Підтримка лямбда-виразів – можливість оголошення анонімних функцій з лаконічним синтаксисом.

• Можливість додавати реалізації методів в інтерфейс, використовуючи ключове слово default, ця особливість також відома розробникам як методи розширення.

• Посилання на методи – можливість передавати у вищезгадані функції посилання на методи різних класів.

• Функціональні інтерфейси (предикати, постачальники і т.д.) – інтерфейс що має один абстрактний метод, їх реалізацію можна приводити до написання лямбда-виразу який і буде відповідати імплементації цього абстрактного методу.

• Потоки для роботи з колекціями – клас java.util.Stream надаєм можливість виконувати велику кількість операції на колекціями за допомогою лаконічних методів що використовують лямбда вирази.

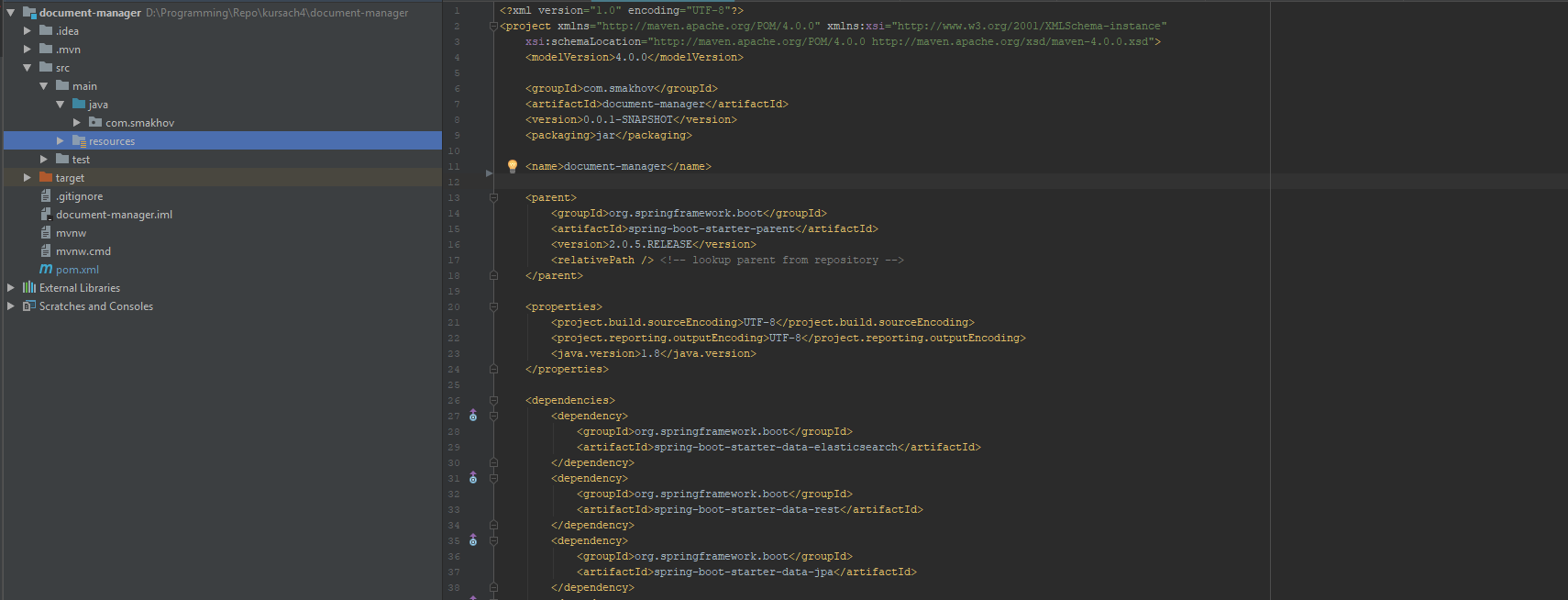
• Java 8 містить новий API для роботи з датами і часом.

Також для спрощення та оптимізації розробки серверних інтерфейсів та інтеграцій з іншими технологіями було використано фреймворк Spring Boot. Даний засіб є збіркою різних додаткових засобів та бібліотек, які можна опціонально підлаштовувати під власні потреби. У нашому випадку, використовуються наступні компоненти даного фреймворку:

* Spring Core – ця компонента є ядром інфраструктури Spring, його роль полягає у реалізації підходів Inversion of Control та Dependency Injection. Дані підходити дозволяють розробнику декларативно описати залежності між класами програми та їх конфігурації, а фреймворк сам створить та підставить їх куди треба, також його реалізація працює з життєвим циклом компонентів системи та дозволяє писати користувацьку логіку в різні етапи роботи системи.
* Spring AOP – модуль для розробки за допомогою парадигми «Аспектно-орієнтованого» програмування. Дана парадигма передбачає можливість втручання в хід роботи додатку для впровадження наскрізною функціональності, як то логування викликів методів, замірів швидкодії, перевірки прав доступу.
* Spring MVC – компонента яка надає можливість описувати REST точки доступу для клієнтів додатку. Також варто зазначити що вона додає можливість для імплементації архітектурного шаблону MVC, який дозволяє розділити бізнес-логіку, логіку доступу до бази та логіку відображення на різні шари додатку.
* Spring Data – механізм для швидкої розробки методів доступу до бази. Дана підсистема дозволяє описувати методи та запити до бази даних без необхідності ручного налаштування підключення до бази даних та керуванням транзакціями.

Також для опису залежностей цього додатку використовується менеджер проекту – Maven, який дозволяє за допомогою плагінів та додатків описати етапи компіляції та збірки проекту, також існує можливість описати бібліотеки які необхідні для роботи проекту. Вся конфігурація проекту описується в файлі pom.xml.

Ознайомимось з структурою проекту.

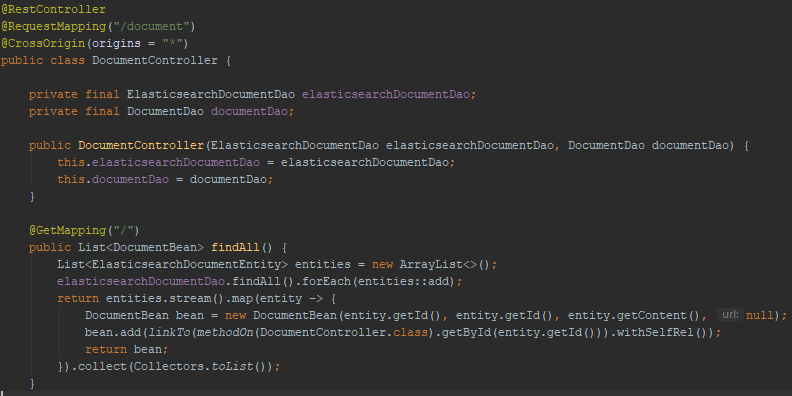


*Рис. 4. Структура проекту*

Даний проект наслідує структуру що пропонується Maven, сирцевий код знаходиться у папці src/main/java, статичні ресурси (конфігурації профілів додатку, сценарії міграції бази даних) можна знайти у директорії src/main/resources, файли для тестів знаходяться у директоріях src/test.

Шари архітектури додатку розбитий на окремі пакети які групують класи за їх відповідальністю, розглянемо приклади кожного з шарів.

У пакеті «controller» знаходяться класи які відповідають за побудову та відповідь на REST-запити клієнтів. Для прикладу розглянемо клас DocumentController:

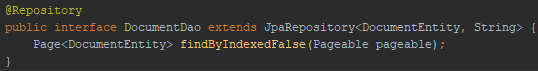


*Рис. 5 Частина коду контролеру*

Цей клас – приклад з шару контролерів, які відповідають за надання відповіді клієнту на конкретні запити. Наступним анотаціям слід приділити увагу:

* @RestController – індикатор для контейнера Spring який показує що даний клас контролером який є імплементацією специфікації REST. Для фреймворку це означає що слід просканувати усі методи на наявність спеціальних анотацій та зпівставити їх з відповідними шляхами URI, ці методи будуть викликатись при запиті на ці URI.
* @RequestMapping – анотація яка інформує фреймворк про те що всі запити які будуть звертатись до цього контролеру повинні бути з префіксом (у нашому випадку «document»)
* @GetMapping – одна з спеціальних анотацій яка інформує про те що метод який нею анотований буде викликаний у разі HTTP-запиту з методом GET на відповідний шлях.

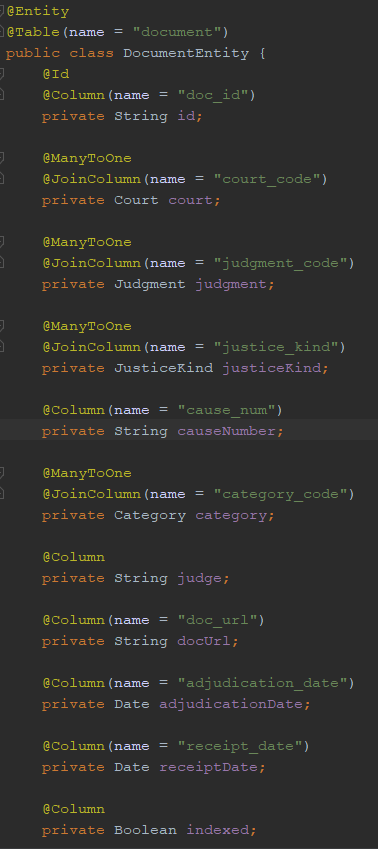
У пакеті dao знаходяться інтерфейси доступу та операцій з базою даних. Дані шар архітектури активно використовує можливості бібліотеки Spring Data, що надає змогу розробнику декларативно описувати необхідні операції за допомогою Java-інтерфейсів без необхідності їх реалізації. Розглянемо один з них (DocumentDao):



*Рис. 6. Лістинг інтерфейсу роботи з документами у базі даних*

Даний клас відмічений анотацією @Repository що інформує Spring-контейнер про те що даний інтерфейс є засобом комунікації з базою. Окрім того, даний інтерфейс наслідується від іншого інтерфейсу – JpaRepository, що свідчить про те що він буде використовувати налаштування JPA у проекті, а отже працювати з реляційною PostgreSQL СКБД, даний інтерфейс має узагальнюючі параметри, у даному випадку використані DocumentEntity – клас моделі з яким працює даний інтерфейс та String – тип який відповідає типу головного ключа в цієї сутності. Всередині інтерфейсу можемо бачити єдиний метод «findByIndexedFalse» який буде шукати документи у яких значення колонки «indexed» дорівнює «false», дана функціональність досягається за допомогою назви методу, більше того метод повертає результати пошуку в базі посторінково та надає можливість отримати сторінку починаючи з необхідного запису та необхідної довжини.

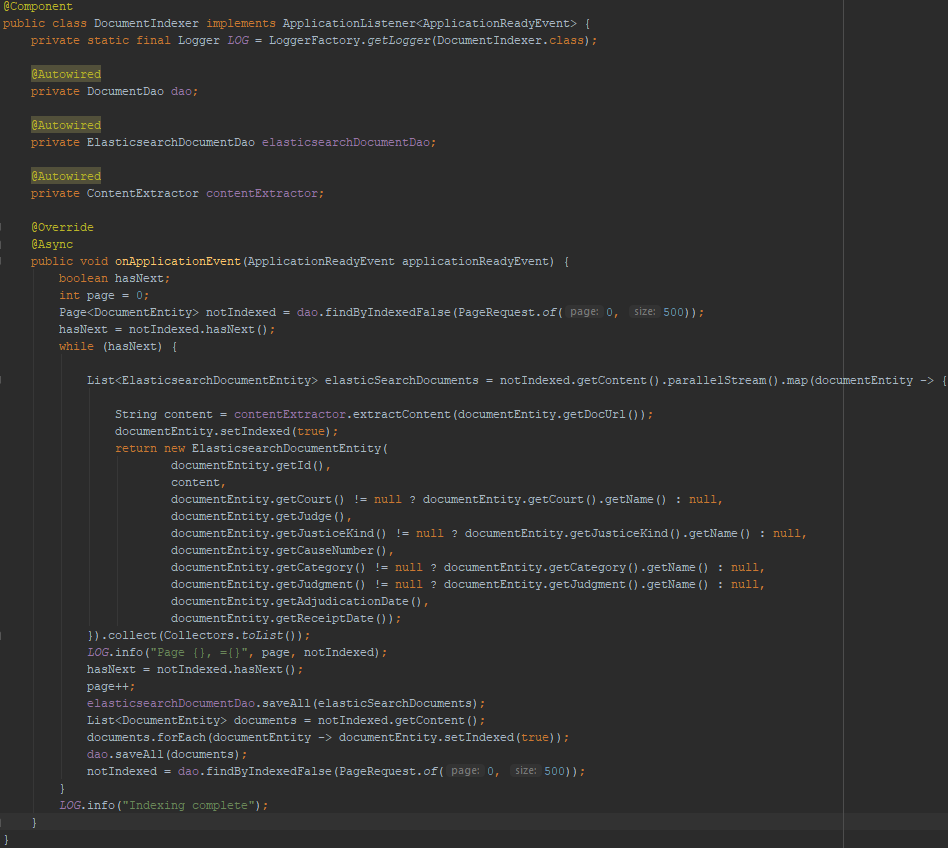
Пакет entity містить усі класи моделей (сутностей) з якими працює наш додаток, дані класи є відображенням таблиць або індексів сховищ даних у нашому додатку. Розглянемо модель «DocumentEntity» яка є відображенням метаданих судових документів у додатку:



*Рис. 7. Лістинг сутності DocumentEntity*

Даний клас використовує анотації специфікації JPA (Java Persistence API) для роботи з базами даних. Першою є анотація @Entity яка сповіщує додаток про те що даний клас є проекцією сутності з бази даних у Java. Наступною слідує анотація @Table(name=”document”) яка сповіщує про те що даний клас представляє конкретну таблиці в базі даних, а саме «document». @Id інформує про те що дана властивість є первинним ключем цієї сутності. @Column(name=”doc\_id”) означає що значення цього поля знаходяться у відповідній колонці «doc\_id» яке знаходиться у параметрі name, якщо даний параметр не використаний то в якості імені колонки використовується ім’я поля. @ManyToOne та @JoinColumn анотації свідчать про те що сутність пов’язана з іншою сутністю у відношенні багато до одного, тобто в базі даних дві таблиці пов’язані зовнішнім ключем.

Окрім вищезазначених основних класів та пакетів у додатку існує функціональність для автоматичного індексування тексту документів у ElasticSearch. Дана можливість представлена класом DocumentIndexer:

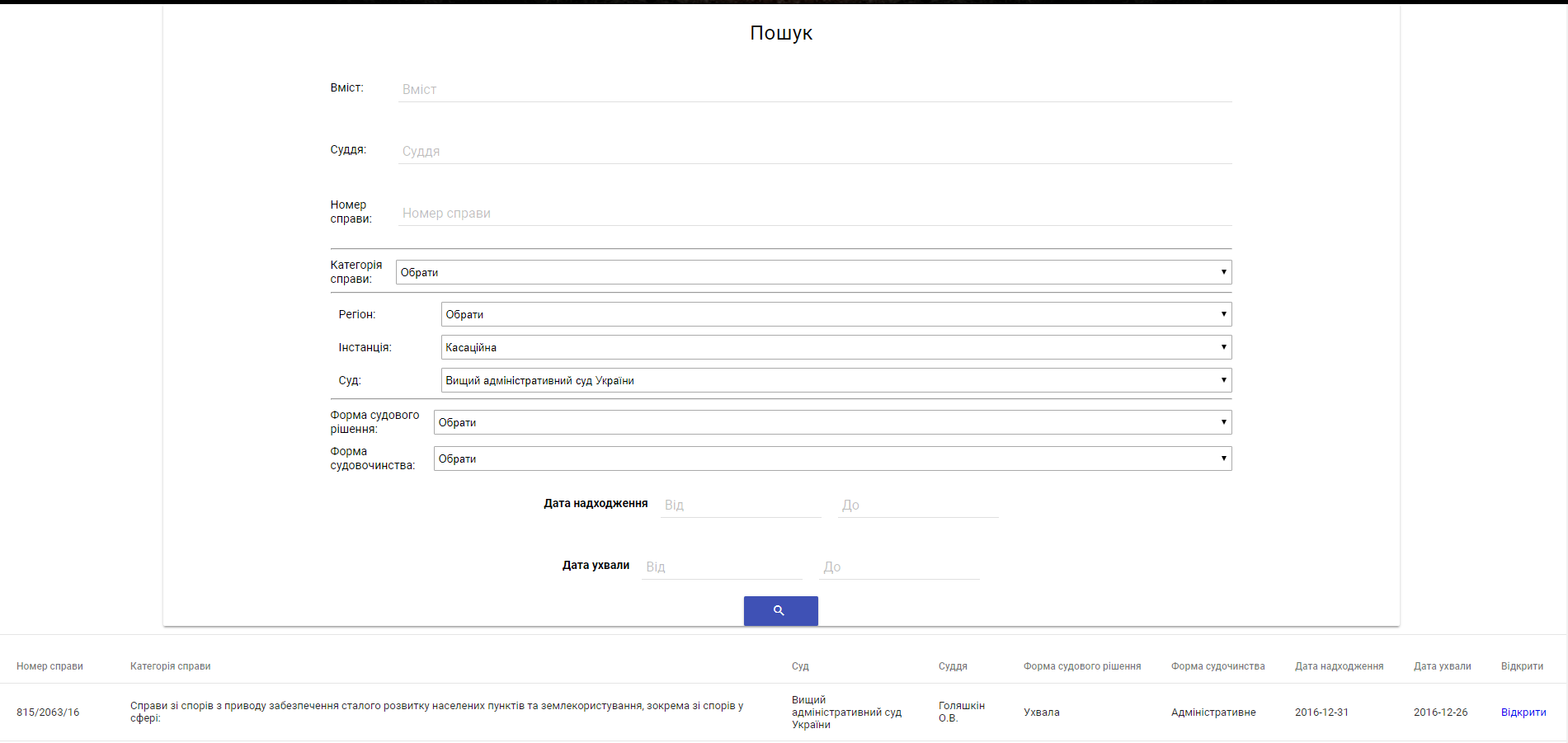


*Рис. 8. Лістинг класу DocumentIndexer*

Даний клас має анотацію @Component яка говорить додатку про те що він є частиною контейнеру який працює з залежностями, також даний клас реалізує інтерфейс ApplicationListener<ApplicationReadyEvent> що має єдиний метод onApplicationEvent який повинен запуститись коли додаток буде готовий до старту. Нижче, над полями класу, використовується анотація @Autowired яка говорить додатку про те що значення цих полів потрібно заповнити з контейнеру додатку. Над методом onApplicationEvent анотація @Async означає те що даний метод повинен виконуватись асинхронно від основного потоку додатку.

### 3. Інструкція користувачу

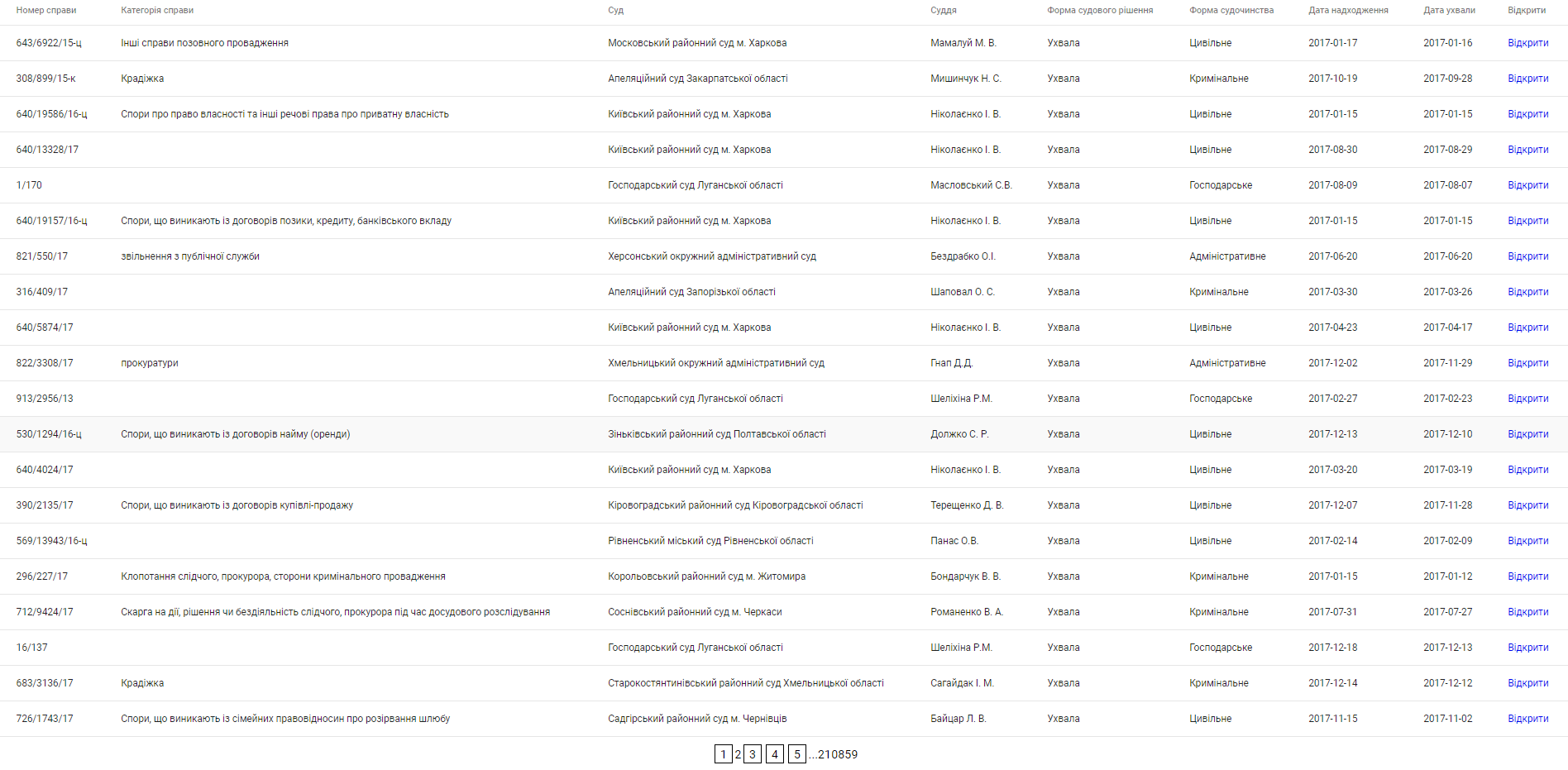
Оскільки судові рішення мають бути відкриті будь-якій людині то для використання даного додатку не потрібно проходити будь яких реєстрацій. Кожний користувач системи матиме доступ до усіх можливостей системи, включаючи елемент пошуку (рис. 9):



*Рис. 9 Елемент пошуку реєстру судових рішень*

З цієї сторінки користувач має змогу задати необхідні критерії для пошуку документів судових рішень. На даний момент можна знаходити за наступними критеріями: текст документу, ПІБ судді, номер справи, категорія справи, суд, форми судочинства та судового рішення, а також обирати часові інтервали для дат надходження та ухвали рішень. Для зручності вибору необхідного суду існують фільтри за регіоном суду та його інстанцією, після їх вибору фільтр «Суд» матиме відфільтровану за цими критеріями кількість результатів. При натисканні поля що належать властивостям «Дата ухвали» чи «Дата надходження» відкривається модальне вікно яке дозволяє користувачу зручно обрати необхідну дату.

Після задання всіх необхідних параметрів користувач отримує результат внизу сторінки який відсортований за найкращим збігом (рис. 10):



*Рис. 10. Приклад результатів пошуку*

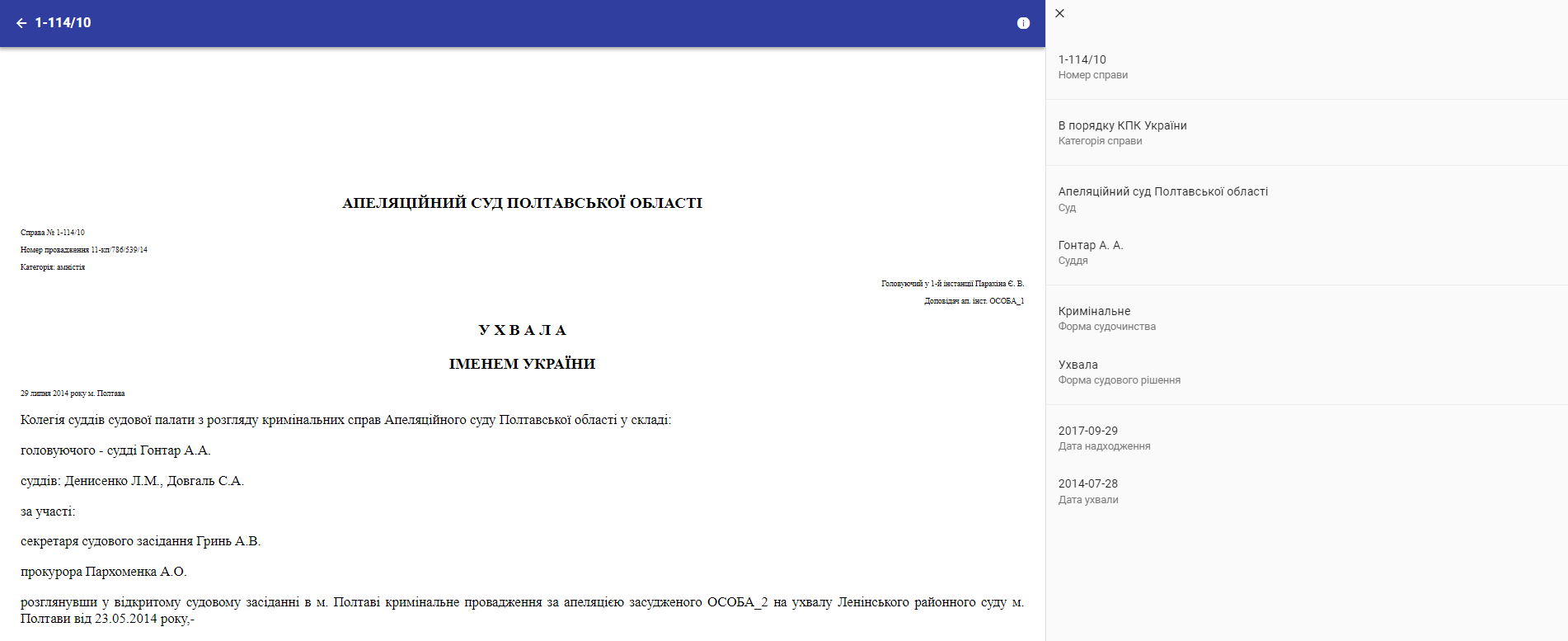
У списку результатів передбачене вивід наступних властивостей того чи іншого документу судового рішення: номер справи, категорія справи, суд та суддя що виніс дане рішення, форма судового рішення та форма судочинства, дати надходження та ухвали. Для зменшення навантаження на канал передачі та сервери в цілому було передбачено вивід посторінкових результатів. Внизу сторінки користувач має змогу скористуватись компонентами сторінкування для навігації за порціями даних, та власне, може побачити загальну кількість сторінок

Окрім того, присутня можливість ознайомитись з кожним документом окремо натиснувши на посилання «Відкрити» (рис. 11):



*Рис. 11 Сторінка перегляду судового рішення*

Натиснувши на посилання користувач переходить на сторінку перегляду документу судового рішення де має змогу ознайомитись з його текстом та ознайомитись з його метаданими натиснувши на кнопку з символом «і» (рис 12):



*Рис. 12 Метадані судового рішення*

Для повернення на попередню сторінку користувачу слід натиснути на кнопку з символом «←».

Вся розробка інтерфейсу відповідала принципам дизайну сайтів запропонованими компанією Google – «Material Design».

# Висновки

Реєстр судових рішень – розподілена інформаційна система, що дозволяє зберігати і використовувати велику кількість неструктурованої інформації що й представляє судові рішення у зручному для користувачів вигляді. Основною задачею реєстру є збереження та надання можливостей ефективної навігації серед великої кількості документів судових рішень.

В процесі виконання магістерської роботи було проаналізовано існуючу систему задіяну в Україні та існуючі конкуруючі системи, більшість з яких є комерційними або ж що не дають змогу вирішити поставлені задачі.

Було проведено аналіз існуючих підходів у збережені реляційних та неструктурованих даних та можливість їх використання для збереження у «Реєстрі судових рішень».

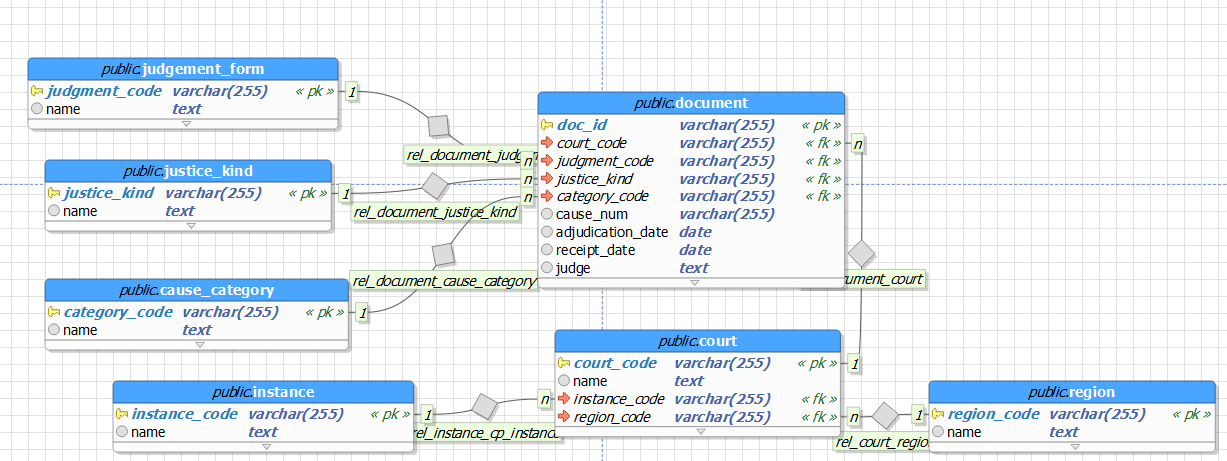
Розроблено:

* схему бази даних для збереження метаданих документів судових рішень;
* архітектуру та власне додаток для реалізації задачі збереження та навігації по судовим рішенням;
* веб інтерфейс використовуючи сучасні та ефективні технології розробки у світі веб-порталів;
* також розроблено серверний додаток який має гнучку та зрозумілу структуру і може бути запущений як на приватних серверах так і на хмарних сервісах.

# Список використаної літератури

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированого проектирования. Паттерны проектирования. – СПб.: Питер, 2015 – 368 с.: ISBN 978-5-496-00389-6
2. Эккель Б. Философия Java. 4-те полное изд. – СПб.: Питер, 2015 – 1168с.: ISBN 978-5-496-01127-3Дронов, В. JavaScript в Web-дизайне; СПб: БХВ, **2011. -** 880 c
3. Erik Wilde, Cesare Pautasso. REST: From Research to Practice. — Springer Science & Business Media, 2011. — 528 p. — ISBN 978-1-4419-8303-9.
4. Clinton Gormley, Zachary Tong. Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine – O’REILLY; 1st ed. 2015 – 724 p. – ISBN - 978-1449358549.
5. Radu Gheorghe Matthew Lee Hinman. Elasticsearch in Action – Manning Publications; 1st ed. 2015 – 496 p. – ISBN - 978-1617291623.
6. Artemij Fedosejev. React.js Essentials– Packt; 2nd ed. 2015 – 208 p. – ISBN - 9781783551620.
7. Java 8 Reference: https://docs.oracle.com/javase/8/docs/api/
8. History of databases: <https://www.researchgate.net/publication/298332910_History_Of_Databases>
9. ACID vs BASE consistency models explained: <https://neo4j.com/blog/acid-vs-base-consistency-models-explained/>
10. CAP theorem and distributed database management systems: https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e

# Додаток А: Схема бази даних



# Додаток Б: Код серверного додатку

Лістинг DocumentManagerApplication:

package com.smakhov;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Import;  
import org.springframework.data.elasticsearch.repository.config.EnableElasticsearchRepositories;  
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;  
import org.springframework.scheduling.annotation.EnableAsync;  
  
@SpringBootApplication  
@EnableJpaRepositories  
@EnableAsync  
@Import(Config.class)  
@EnableElasticsearchRepositories(basePackages="com.smakhov.dao")  
public class DocumentManagerApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(DocumentManagerApplication.class, args);  
 }  
}

Лістинг SearchController:

package com.smakhov.controller;  
  
import com.smakhov.entity.ElasticsearchDocumentEntity;  
import org.elasticsearch.index.query.BoolQueryBuilder;  
import org.elasticsearch.index.query.QueryBuilders;  
import org.elasticsearch.search.sort.SortBuilders;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.domain.PageRequest;  
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;  
import org.springframework.data.elasticsearch.core.aggregation.AggregatedPage;  
import org.springframework.data.elasticsearch.core.query.NativeSearchQueryBuilder;  
import org.springframework.web.bind.annotation.CrossOrigin;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RestController;  
  
import static org.elasticsearch.index.query.QueryBuilders.*queryStringQuery*;  
  
@RestController  
@CrossOrigin(origins = "\*")  
public class SearchController {  
  
 @Autowired  
 private ElasticsearchTemplate template;  
  
 @PostMapping("/search")  
 public AggregatedPage<ElasticsearchDocumentEntity> search(@RequestBody UISearchRequest searchRequest) {  
 Integer page = searchRequest.getPage();  
 NativeSearchQueryBuilder searchQuery = new NativeSearchQueryBuilder().withFields("\*");  
 if (searchRequest.getContent() != null) {  
 searchQuery.  
 withQuery(  
 *queryStringQuery*(searchRequest.getContent()).defaultField("content"));  
 }  
 if (searchRequest.hasAnyFilter()) {  
 BoolQueryBuilder filter = buildFilters(searchRequest);  
 searchQuery.withFilter(  
 filter  
 );  
  
 }  
  
 searchQuery  
 .withPageable(PageRequest.*of*(page != null ? page : 0, 20)).withSort(SortBuilders.*scoreSort*()).withSort(SortBuilders.*fieldSort*("id.keyword"));  
  
 return template.queryForPage(searchQuery.build(), ElasticsearchDocumentEntity.class);  
 }  
  
 private BoolQueryBuilder buildFilters(@RequestBody UISearchRequest searchRequest) {  
 BoolQueryBuilder filter = QueryBuilders.*boolQuery*();  
 if (searchRequest.getJudge() != null) {  
 filter.filter(  
 QueryBuilders.*queryStringQuery*(searchRequest.getJudge()).defaultField("judge")  
 );  
 }  
 if (searchRequest.getCourt() != null)  
 filter.filter(  
 QueryBuilders.*termQuery*("court.keyword", searchRequest.getCourt())  
 );  
 if (searchRequest.getCaseNumber() != null) {  
 filter.filter(  
 QueryBuilders.*termQuery*("causeNumber.keyword", searchRequest.getCaseNumber()));  
 }  
 if (searchRequest.getCauseCategory() != null) {  
 filter.filter(  
 QueryBuilders.*termQuery*("category.keyword", searchRequest.getCauseCategory()));  
 }  
 if (searchRequest.getJudgmentForm() != null) {  
 filter.filter(  
 QueryBuilders.*termQuery*("judgment.keyword", searchRequest.getJudgmentForm()));  
 }  
 if (searchRequest.getJusticeKind() != null) {  
 filter.filter(  
 QueryBuilders.*termQuery*("justiceKind.keyword", searchRequest.getJusticeKind()));  
 }  
 if (searchRequest.getAdjudicationDateFrom() != null || searchRequest.getAdjudicationDateTo() != null) {  
 filter.filter(QueryBuilders.*rangeQuery*("adjudicationDate").from(searchRequest.getAdjudicationDateFrom()).to(searchRequest.getAdjudicationDateTo()));  
 }  
 if (searchRequest.getReceiptDateFrom() != null || searchRequest.getReceiptDateTo() != null) {  
 filter.filter(QueryBuilders.*rangeQuery*("receiptDate").from(searchRequest.getReceiptDateFrom()).to(searchRequest.getReceiptDateTo()));  
 }  
 return filter;  
 }  
}

Лістинг DocumentEntity:

package com.smakhov.entity;  
  
import javax.persistence.\*;  
import java.util.Date;  
import java.util.Objects;  
  
@Entity  
@Table(name = "document")  
public class DocumentEntity {  
 @Id  
 @Column(name = "doc\_id")  
 private String id;  
  
 @ManyToOne  
 @JoinColumn(name = "court\_code")  
 private Court court;  
  
 @ManyToOne  
 @JoinColumn(name = "judgment\_code")  
 private Judgment judgment;  
  
 @ManyToOne  
 @JoinColumn(name = "justice\_kind")  
 private JusticeKind justiceKind;  
  
 @Column(name = "cause\_num")  
 private String causeNumber;  
  
 @ManyToOne  
 @JoinColumn(name = "category\_code")  
 private Category category;  
  
 @Column  
 private String judge;  
  
 @Column(name = "doc\_url")  
 private String docUrl;  
  
 @Column(name = "adjudication\_date")  
 private Date adjudicationDate;  
  
 @Column(name = "receipt\_date")  
 private Date receiptDate;  
  
 @Column  
 private Boolean indexed;  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
  
 public Court getCourt() {  
 return court;  
 }  
  
 public void setCourt(Court court) {  
 this.court = court;  
 }  
  
 public Judgment getJudgment() {  
 return judgment;  
 }  
  
 public void setJudgment(Judgment judgment) {  
 this.judgment = judgment;  
 }  
  
 public JusticeKind getJusticeKind() {  
 return justiceKind;  
 }  
  
 public void setJusticeKind(JusticeKind justiceKind) {  
 this.justiceKind = justiceKind;  
 }  
  
 public Category getCategory() {  
 return category;  
 }  
  
 public void setCategory(Category category) {  
 this.category = category;  
 }  
  
 public Boolean getIndexed() {  
 return indexed;  
 }  
  
 public void setIndexed(Boolean indexed) {  
 this.indexed = indexed;  
 }  
  
 public String getCauseNumber() {  
 return causeNumber;  
 }  
  
 public void setCauseNumber(String causeNumber) {  
 this.causeNumber = causeNumber;  
 }  
  
 public Date getAdjudicationDate() {  
 return adjudicationDate;  
 }  
  
 public void setAdjudicationDate(Date adjudicationDate) {  
 this.adjudicationDate = adjudicationDate;  
 }  
  
 public Date getReceiptDate() {  
 return receiptDate;  
 }  
  
 public void setReceiptDate(Date receiptDate) {  
 this.receiptDate = receiptDate;  
 }  
  
 public String getJudge() {  
 return judge;  
 }  
  
 public void setJudge(String judge) {  
 this.judge = judge;  
 }  
  
 public String getDocUrl() {  
 return docUrl;  
 }  
  
 public void setDocUrl(String docUrl) {  
 this.docUrl = docUrl;  
 }  
  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 DocumentEntity that = (DocumentEntity) o;  
 return Objects.equals(id, that.id) &&  
 Objects.equals(court, that.court) &&  
 Objects.equals(judgment, that.judgment) &&  
 Objects.equals(justiceKind, that.justiceKind) &&  
 Objects.equals(category, that.category);  
 }  
  
 @Override  
 public int hashCode() {  
  
 return Objects.hash(id, court, judgment, justiceKind, category);  
 }  
  
 @Override  
 public String toString() {  
 return "DocumentEntity{" +  
 "id='" + id + '\'' +  
 ", court=" + court +  
 ", judgment=" + judgment +  
 ", justiceKind=" + justiceKind +  
 ", causeNumber='" + causeNumber + '\'' +  
 ", category=" + category +  
 ", judge='" + judge + '\'' +  
 ", docUrl=" + docUrl +  
  
 ", indexed=" + indexed +  
 '}';  
 }  
}

Лістинг ElasticsearchDocumentEntity:

package com.smakhov.entity;  
  
import com.fasterxml.jackson.annotation.JsonFormat;  
import org.springframework.data.elasticsearch.annotations.Document;  
import org.springframework.data.elasticsearch.annotations.Field;  
import org.springframework.data.elasticsearch.annotations.FieldType;  
  
import java.util.Date;  
  
@Document(indexName="newdocument", type = "document")  
public class ElasticsearchDocumentEntity {  
  
 private String id;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Text*)  
 private String content;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String court;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String judge;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String justiceKind;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String causeNumber;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String category;  
 @Field(analyzer = "ukrainian", searchAnalyzer = "ukrainian", type = FieldType.*Keyword*)  
 private String judgment;  
 @Field(type = FieldType.*Date*, store = true)  
 @JsonFormat(shape = JsonFormat.Shape.*STRING*, pattern = "yyyy-MM-dd")  
 private Date adjudicationDate;  
 @Field(type = FieldType.*Date*, store = true)  
 @JsonFormat(shape = JsonFormat.Shape.*STRING*, pattern = "yyyy-MM-dd")  
 private Date receiptDate;  
 public ElasticsearchDocumentEntity() {  
 }  
 public ElasticsearchDocumentEntity( String content) {  
 this.content = content;  
 }  
 public ElasticsearchDocumentEntity(String id, String content) {  
 this.id = id;  
 this.content = content;  
 }  
 public ElasticsearchDocumentEntity(String id, String content, String court, String judge, String justiceKind, String causeNumber, String category, String judgment, Date adjudicationDate, Date receiptDate) {  
 this.id = id;  
 this.content = content;  
 this.court = court;  
 this.judge = judge;  
 this.justiceKind = justiceKind;  
 this.causeNumber = causeNumber;  
 this.category = category;  
 this.judgment = judgment;  
 this.adjudicationDate = adjudicationDate;  
 this.receiptDate = receiptDate;  
 }  
 public String getContent() {  
 return content;  
 }  
 public String getId() {  
 return id;  
 }  
 public void setContent(String content) {  
 this.content = content;  
 }  
 public void setId(String id) {  
 this.id = id;  
 }  
 public String getCourt() {  
 return court;  
 }  
 public void setCourt(String court) {  
 this.court = court;  
 }  
 public Date getAdjudicationDate() {  
 return adjudicationDate;  
 }  
 public void setAdjudicationDate(Date adjudicationDate) {  
 this.adjudicationDate = adjudicationDate;  
 }  
 public Date getReceiptDate() {  
 return receiptDate;  
 }  
 public void setReceiptDate(Date receiptDate) {  
 this.receiptDate = receiptDate;  
 }  
 public String getJudge() {  
 return judge;  
 }  
 public void setJudge(String judge) {  
 this.judge = judge;  
 }  
 public String getJusticeKind() {  
 return justiceKind;  
 }  
 public void setJusticeKind(String justiceKind) {  
 this.justiceKind = justiceKind;  
 }  
 public String getCauseNumber() {  
 return causeNumber;  
 }  
 public void setCauseNumber(String causeNumber) {  
 this.causeNumber = causeNumber;  
 }  
 public String getCategory() {  
 return category;  
 }  
 public void setCategory(String category) {  
 this.category = category;  
 }  
 public String getJudgment() {  
 return judgment;  
 }  
 public void setJudgment(String judgment) {  
 this.judgment = judgment;  
 }  
 @Override  
 public String toString() {  
 return "ElasticsearchDocumentEntity{" +  
 "content='" + content + '\'' +  
 ", id='" + id + '\'' +  
 '}';  
 }  
}

# Додаток В: Код клієнтського додатку

Лістинг файлу actions/rest.js:

import RestService from '../rest'

import { browserHistory } from 'react-router'

import \* as actionTypes from '../constants/action.types'

import rtfToHtml from '@iarna/rtf-to-html'

export function retreiveDocuments () {

return (dispatch) => {

RestService.getDocuments()

.then(documents => dispatch({

type: actionTypes.DOCUMENTS\_RETREIVED,

docs: documents

}))

}

}

export function searchDocuments (query) {

return (dispatch) => {

dispatch({type: actionTypes.SEARCH\_STARTED})

RestService.searchDocuments(query)

.then(response => dispatch({

type: actionTypes.DOCUMENTS\_RETREIVED,

docs: response.content,

searchInfo: response

})).catch(response => {

dispatch({

type: actionTypes.SEARCH\_ERROR

})

})

}

}

export function addDocument (data) {

return (dispatch) => {

RestService.addDocument(data)

.then(document => {

dispatch({

type: actionTypes.DOCUMENT\_ADDED,

doc: document

})

browserHistory.push('/')

})

}

}

export function loadDocument (id) {

return (dispatch) => {

RestService.loadDocument(id)

.then(file => {

dispatch({

type: actionTypes.DOCUMENT\_FILE\_LOADED,

file,

id

})

RestService.loadDocumentContent(file.url).then(content => {

rtfToHtml.fromString(content, (err,doc) =>{

dispatch({

type: actionTypes.DOCUMENT\_CONTENT\_LOADED,

content: doc,

id

})

})

})

})

}

}

export function deleteDocument (data) {

return (dispatch) => {

RestService.deleteDocument(data)

.then(() => {

browserHistory.push('/')

dispatch({

type: actionTypes.DOCUMENT\_DELETED,

id: data.id

})

})

}

}

Лістинг constants/action.types.js:

export const DOCUMENTS\_RETREIVED = 'DOCUMENTS\_RETREIVED'

export const DOCUMENT\_ADDED = 'DOCUMENT\_ADDED'

export const DOCUMENT\_DELETED = 'DOCUMENT\_DELETED'

export const DOCUMENT\_FILE\_LOADED = "DOCUMENT\_FILE\_LOADED"

export const DOCUMENT\_CONTENT\_LOADED = "DOCUMENT\_CONTENT\_LOADED"

export const SEARCH\_STARTED = "SEARCH\_STARTED"

export const SEARCH\_ERROR = "SEARCH\_ERROR"

Лістинг constainers/AppContainer.js:

import React, { Component, PropTypes } from 'react'

import { browserHistory, Router } from 'react-router'

import { Provider } from 'react-redux'

class AppContainer extends Component {

static propTypes = {

routes: PropTypes.object.isRequired,

store: PropTypes.object.isRequired

}

shouldComponentUpdate () {

return false

}

render () {

const { routes, store } = this.props

return (

<Provider store={store}>

<div style={{ height: '100%' }}>

<Router history={browserHistory} children={routes} />

</div>

</Provider>

)

}

}

export default AppContainer

Лістинг reducers/search/index.js:

import { DOCUMENTS\_RETREIVED, SEARCH\_STARTED, SEARCH\_ERROR } from '../../constants/action.types'

import Immutable from 'immutable'

const initialState = Immutable.fromJS({})

export default (state = initialState, action) => {

switch (action.type) {

case SEARCH\_STARTED:

return state.set("isSearching", true)

case DOCUMENTS\_RETREIVED:

let newState = initialState

newState = newState.set("searchInfo", Immutable.fromJS(action.searchInfo))

return newState.set("isSearching", false)

case SEARCH\_ERROR:

return state.set("isSearching", false)

default:

return state

}

}

Лістинг rest/index.js:

import fetch from 'isomorphic-fetch'

import searchResult from '../test\_jsons/state.js';

class RestService {

searchDocuments(query) {

return fetch('http://localhost:8080/search', {

method: 'POST',

headers: {

'Content-type': 'application/json'

},

body: JSON.stringify(query)

})

.then(response => response.json())

.catch(() => {

return searchResult.search.searchInfo;

})

}

loadDocument(id) {

return fetch(`http://localhost:8080/document/${id}`, { method: 'GET' })

.then(response => response.json())

}

loadDocumentContent(url) {

const newUrl = url.replace("od.reyestr.court.gov.ua", "localhost:8008")

return fetch(newUrl).then(response => response.text())

}

openDocument(id) {

const formData = new FormData()

formData.append('content', query)

return fetch('http://localhost:8080/search', { method: 'POST', body: formData })

.then(response => response.json())

}

}

export default new RestService()

Лістинг routes/Document/index.js:

import DocumentContainer from './containers/DocumentContainer'

// Sync route definition

export default {

path: 'document/:id',

component: DocumentContainer

}

Лістинг routes/Home/containers/HomeContainer.js:

import {searchDocuments } from '../../../actions/rest'

import HomeView from '../components/HomeView'

import { connect } from 'react-redux'

const mapStateToProps = (store) => ({

documents: store.documents,

searchInfo: store.search.get("searchInfo"),

isSearching: store.search.get("isSearching", false)

})

const mapDispatchToProps = (dispatch) => ({

search: (query) => dispatch(searchDocuments(query))

})

export default connect(mapStateToProps, mapDispatchToProps)(HomeView)

Лістинг routes/Home/components/HomeView.js:

import React from 'react';

import { Card, CardMedia, CardTitle, CardText, CardActions } from 'react-toolbox/lib/card';

import ProgressBar from 'react-toolbox/lib/progress\_bar';

import Table, { TableCell, TableHead, TableRow } from 'react-toolbox/lib/table';

import { Input } from 'react-toolbox/lib/input';

import { DatePicker } from 'react-toolbox/lib/date\_picker';

import Select from '../../../generic\_components/Select';

import { Button } from 'react-toolbox/lib/button';

import { Link } from 'react-router'

import Pagination from '../../../generic\_components/Pagination';

import theme from './HomeView.css'

import courtsList from '../../../test\_jsons/courts.json';

import justiceKinds from '../../../test\_jsons/justice\_kinds.json';

import regionList from '../../../test\_jsons/regions.json';

import causeCategories from '../../../test\_jsons/cause\_categories.json';

import judgmentForms from '../../../test\_jsons/judgment\_forms.json';

import instances from '../../../test\_jsons/instances.json';

class HomeView extends React.Component {

static propTypes = {

documents: React.PropTypes.object,

retreiveDocuments: React.PropTypes.func,

search: React.PropTypes.func

}

// load data for courts select

loadCourtSelectOptions() {

this.setState({

courtsList: courtsList

});

}

// load data for instances select

loadInstanceSelectOptions() {

this.setState({

instanceList: instances

});

}

// load data for justice kinds select

loadJusticeKindSelectOptions() {

this.setState({

justiceKinds: justiceKinds

});

}

// load data for regions select

loadRegionSelectOptions() {

this.setState({

regionList: regionList

});

}

// load data for Judgment forms select

loadJudgmentFormsSelectOptions() {

this.setState({

judgmentForms: judgmentForms

});

}

// load data for cause categories select

loadCauseCategoriesSelectOptions() {

this.setState({

causeCategories: causeCategories

});

}

componentDidMount() {

this.loadCourtSelectOptions()

this.loadCauseCategoriesSelectOptions();

this.loadRegionSelectOptions();

this.loadJudgmentFormsSelectOptions();

this.loadJusticeKindSelectOptions();

this.loadInstanceSelectOptions();

}

componentWillReceiveProps(nextProps) {}

constructor() {

super()

this.state = {

formData: {}

}

this.search = :: this.search

this.changePage = :: this.changePage

}

search() {

if (this.formValid()) {

this.props.search(this.state.formData)

}

}

goToPage(index) {

this.props.search({...this.state.formData, page: index})

}

// check is at least one field is not empty

formValid() {

let valid = false;

Object.keys(this.state.formData).map((item) => {

if (this.state.formData[item] != null) {

valid = true;

}

});

return valid;

}

// onChange input. Where event is typed text, type - name of input

changeInput(event, type) {

let stateItem = { ...this.state };

stateItem.formData[type] = event === "" ? null : event;

this.setState(stateItem);

}

// onChange select. Where event is choosen option, type - name of select

changeSelect(event, type) {

let stateItem = { ...this.state };

stateItem.formData[type] = event.target.value === "Обрати" ? null : event.target.value;

this.setState(stateItem);

}

// onChange DatePicker. Where event is choosen data, type - key in this state.

changeDate(date, type) {

let stateItem = { ...this.state };

stateItem.formData[type] = date;

this.setState(stateItem);

}

changePage(pageIndex) {

this.setState({ activePage: pageIndex });

}

render() {

const { courtsList : stateCourtsList = [], regionList = [], causeCategories = [], judgmentForms = [], instanceList = [] } = this.state;

const courtsList = stateCourtsList.filter(court => {

if(this.state.formData.region == null)

{

return true;

}

return court.region\_code == this.state.formData.region

}).filter(court => {

if(this.state.formData.instance == null)

{

return true;

}

return court.instance\_code == this.state.formData.instance

})

const renderList = this.props.documents.valueSeq().toJS().map((item, idx) => {

return (

<TableRow key={idx}>

<TableCell>{item.causeNumber}</TableCell>

<TableCell>{item.category}</TableCell>

<TableCell>{item.court}</TableCell>

<TableCell>{item.judge}</TableCell>

<TableCell>{item.judgment}</TableCell>

<TableCell>{item.justiceKind}</TableCell>

<TableCell>{item.receiptDate}</TableCell>

<TableCell>{item.adjudicationDate}</TableCell>

<TableCell><Link className={theme.link} to={`/document/${item.id}`}>Відкрити</Link></TableCell>

</TableRow>

)

});

return (

<div>

<div className={theme.search}>

<Card style={{width: '80%', alignItems: "center"}}>

<CardTitle>Пошук</CardTitle>

<div className={theme.topForm}>

<div>

<div><div>Вміст:</div> <Input label="Вміст" value={this.state.searchQuery} onChange={(event) => this.changeInput(event, "content")} /></div>

<div><div>Суддя:</div><Input label="Суддя" name="judge" onChange={(event) => this.changeInput(event, 'judge')} /></div>

<div><div>Номер справи:</div><Input label="Номер справи" name="case number" onChange={(event) => this.changeInput(event, 'caseNumber')} /></div>

</div>

<hr />

<div>

<Select name="causeCategory" title="Категорія справи" changeSelect={(event) => this.changeSelect(event, 'causeCategory')} value={this.state.formData.causeCategory}>

<option value={null}>Обрати</option>

{causeCategories.map((causeCategory) => {

return (

<option value={causeCategory.name}>{causeCategory.name}</option>

)

})}

</Select>

</div>

<hr />

<div className={theme.courtBlock}>

<div>

<Select name="Region" title="Регіон" changeSelect={(event) => this.changeSelect(event, 'region')} value={this.state.formData.region}>

<option value={null}>Обрати</option>

{regionList.map((region, index) => {

return (

<option value={region.region\_code} key={region.region\_code}>{region.name}</option>

)

})}

</Select>

</div>

<div>

<Select name="Instance" title="Інстанція" changeSelect={(event) => this.changeSelect(event, 'instance')} value={this.state.formData.instance}>

<option value={null}>Обрати</option>

{instanceList.map((instance, index) => {

return (

<option value={instance.code} key={instance.code}>{instance.instance}</option>

)

})}

</Select>

</div>

<div>

<Select name="Court" title="Суд" changeSelect={(event) => this.changeSelect(event, 'court')} value={this.state.formData.court}>

<option value={null}>Обрати</option>

{courtsList.map((court, index) => {

return (

<option value={court.name} key={court.name}>{court.name}</option>

)

})}

</Select>

</div>

</div>

<hr />

<div>

<Select name="JudgmentForm" title="Форма судового рішення" changeSelect={(event) => this.changeSelect(event, 'judgmentForm')} value={this.state.formData.judgmentForm}>

<option value={null}>Обрати</option>

{judgmentForms.map((judgmentForm, index) => {

return (

<option value={judgmentForm.name} key={judgmentForm.name}>{judgmentForm.name}</option>

)

})}

</Select>

</div>

<div>

<Select name="JusticeKind" title="Форма судовочинства" changeSelect={(event) => this.changeSelect(event, 'justiceKind')} value={this.state.formData.justice\_kind}>

<option value={null}>Обрати</option>

{justiceKinds.map((justiceKind, index) => {

return (

<option value={justiceKind.name} key={justiceKind.justice\_kind}>{justiceKind.name}</option>

)

})}

</Select></div>

</div>

<div className={theme.bottomForm}>

<h4>Дата надходження</h4>

<div className={theme.searchElement}>

<div className={theme.formElem}>

<DatePicker label="Від" inputFormat={(value) => `${value.getFullYear()}-${value.getMonth() + 1}-${value.getDate()}`} onChange={(date) => this.changeDate(date, 'receiptDateFrom')}

maxDate={this.state.formData.receiptDateTo} value={this.state.formData.receiptDateFrom} />

</div>

<div className={theme.formElem}>

<DatePicker label="До" inputFormat={(value) => `${value.getFullYear()}-${value.getMonth() + 1}-${value.getDate()}`}onChange={(date) => this.changeDate(date, 'receiptDateTo')}

minDate={this.state.formData.receiptDateFrom} value={this.state.formData.receiptDateTo} />

</div>

</div>

</div>

<div className={theme.bottomForm}>

<h4>Дата ухвали</h4>

<div className={theme.searchElement}>

<div className={theme.formElem}>

<DatePicker label="Від" inputFormat={(value) => `${value.getFullYear()}-${value.getMonth() + 1}-${value.getDate()}`} onChange={(date) => this.changeDate(date, 'adjudicationDateFrom')}

maxDate={this.state.formData.adjudicationDateTo} value={this.state.formData.adjudicationDateFrom} />

</div>

<div className={theme.formElem}>

<DatePicker label="До" inputFormat={(value) => `${value.getFullYear()}-${value.getMonth() + 1}-${value.getDate()}`} onChange={(date) => this.changeDate(date, 'adjudicationDateTo')}

minDate={this.state.formData.adjudicationDateFrom} value={this.state.formData.adjudicationDateTo} />

</div>

</div>

</div>

<Button icon='search' onClick={this.search} raised primary disabled={this.props.isSearching} />

</Card>

</div>

{this.props.isSearching && <div className={theme.loading}><ProgressBar type="circular" mode="indeterminate" /></div>}

{this.props.searchInfo && !this.props.isSearching ?

(<div><Table selectable={false} style={{ marginTop: 10 }}>

<TableHead>

<TableCell>Номер справи</TableCell>

<TableCell>Категорія справи</TableCell>

<TableCell>Суд</TableCell>

<TableCell>Суддя</TableCell>

<TableCell>Форма судового рішення</TableCell>

<TableCell>Форма судочинства</TableCell>

<TableCell>Дата надходження</TableCell>

<TableCell>Дата ухвали</TableCell>

<TableCell>Відкрити</TableCell>

</TableHead>

{renderList}

</Table>

<Pagination goToPage={(index) => this.goToPage(index)} {...this.props.searchInfo.toJS()} /></div>) : null

}

</div>

)

}

}

export default HomeView