МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Савченко А.С.
«___»_____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
«МАГІСТРА»
ЗА СПЕЦІАЛІЗАЦІЄЮ «ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ (ЗА ГАЛУЗЯМИ) »

Тема: «Система кодогенерації в сучасних веб-фреймворках»

Виконавець: Романенко Іван Олександрович
Керівник: к.т.н., доцент Моржов Володимир Іванович

Нормоконтролер: _____ Райчев І.Е.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра Комп'ютерних інформаційних технологій

Галузь знань, спеціальність, спеціалізація: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології (за галузями)" .

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Савченко А.С.
"___"_____2020 р

ЗАВДАННЯ

на виконання дипломної роботи студента

Романенко Івана Олександровича

1. Тема роботи: «Система кодогенерації в сучасних веб-фреймворках»

Затверджена наказом ректора № 1891/ст від 02.10.2020р.

2. Термін виконання роботи: з 5.10.2020 по 31.12.2020р.

3. Вихідні данні до роботи: кодова база системи, для кодогенерації.

4. Зміст пояснювальної записки: вступ, аналітичний огляд і постановка завдання, архітектура проекту, основні діючі компоненти та їх опис, приклад використання Graphql за допомогою системи, висновок.

5. Перелік обов'язкового графічного матеріалу: структура проекту кодогенерації, схема обробки, приклад використання усіх можливостей системи кодогенерації.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Проаналізувати літературу та джерела за темою дипломного проекту.	6.10.20 – 10.11.20	
2.	Розроблення та затвердження плану дипломного проекту.	11.11.20 – 13.11.20	
3.	Провести консультації з науковим керівником щодо створення першого розділу.	14.11.20 – 18.11.20	
4.	Розробка розділу 1: Аналітичний огляд і постановка завдання	01.12.20 – 02.12.20	
5.	Розробка розділу 2: Аналітичний огляд graphql схем та запитів	03.12.20 – 05.12.20	
6.	Розробка розділу 3: Реалізація кодогенератора.	06.12.20 – 07.12.20	
7.	Розробка розділу 4: Генерація коду за допомогою створенної системи	08.12.20 – 10.12.20	
8.	Висновки та оформлення пояснювальної записки дипломного проекту.	11.12.20 – 12.12.20	
9.	Підписання необхідних документів у встановленому порядку.	13.12.20 – 13.12.20	
10.	Оформлення та друк пояснювальної записки дипломної роботи	14.12.20 – 14.12.20	

7. Дата видачі завдання:

Керівник дипломного проекту _____ Моржов В.І.
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Романенко І.О.
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломного проекту роботи «Система кодогенерації в сучасних веб-фреймворках» викладена на 108 сторінках, містить 11 рисунків, 5 літературних джерел, 35 сторінок додатків.

Мета роботи: Створення власної системи, для кодогенерації методів для звертання до серверу, використовуючи GraphQL схеми.

Об'єктом дослідження є сучасні моделі для комунікації між клієнтом та сервером: REST API та GraphQL

Предметом дослідження є REST API та GraphQL засоби для коммунікації між клієнтом та сервером.

Ключові слова: СИСТЕМА КОДОГОНЕРАЦІЇ, ВЕБ ТЕХНОЛОГІЇ, ВЕБ-ФРЕЙМВОРК, GRAPHQL, REST API

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

БД – база даних

ВФ - веб-фреймворк

ВД - веб-додаток

ТЗ – технічне завдання

BE – back end

FE- front end

CP - CROSS PLATFORM

CLI – command line interface

WWW - World Wide Web

JS – JavaScript

ES - EcmaScript

ES6 – EcmaScript6

REST – Representational state transfer

API - Application programming interface

IDL - Interactive Data Language

ВСТУП

На сьогоднішній день, у світі існує величезна кількість складних та великих систем, за кожною із яких стоять десятки , або навіть, сотні тисяч строк коду, із заплутаною логікою. Впоратися із цим допомогає типизація даних.

За допомогою типізації даних ми можемо писати код більш ізольовано, розбивати його на невеликі компоненти, та бути впевненими в тому, что при певних аргументах, API компонента або функції буде завжди повертати дані з одним і тим самим інтерфейсом, що значно полегшує розробку та робить її більш безпечною.

Здебільшого, для побудови складних систем використовуються REST API. Цей архітектурний стиль вдало вирішувати поставлені перед ним задачи, але у нього є суттєві недоліки, а саме: неможливість зв`язати типизацію сервера с клієнтською типізацію. Саме через це між бекендом та клієнтською частиною створюєтьяс умовний контракт інтерфейсів, що у свою чергу призводить до наступних проблем:

- Неможливість клієнта дізнатись про зміни на сервері
- Мануальне копіювання типів бекенду, що повищує вірогідність людьскої помилки
- Неможливість клієнта контролювати інтерфейс моделі даних, що будуть надіслані сервером.

Усі ці проблеми може вирішити архітектура GraphQL, яка у поєднанні з кодогенерацією може значно покращити та прискорити розробку, в якій необхідно комунікувати із сервером.

Тому, для створення системи кодогенерації на базі GraphQL, на прикладах сучасних веб-фреймворків, була обрана саме ця тема.

РОЗДІЛ 1

АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1. Основні положення щодо кодогенерації

У обчисленнях генерація коду - це процес, за допомогою якого генератор коду компілятора перетворює деяке проміжне подання вихідного коду у форму (наприклад, машинний код), який може бути легко виконаний машиною.

Складні компілятори, як правило, виконують кілька проходів над різними проміжними формами. Цей багатоступеневий процес використовується, оскільки багато алгоритмів для оптимізації коду легше застосовувати по одному, або тому, що вхід в одну оптимізацію покладається на завершену обробку, виконану іншою оптимізацією. Ця організація також сприяє створенню єдиного компілятора, який може орієнтуватись на декілька архітектур, оскільки лише останній із етапів генерації коду (бекенд) повинен змінюватися від цільової до цільової.

Вхідні дані до генератора коду зазвичай складаються з дерева синтаксичного аналізу або абстрактного дерева синтаксису. Дерево перетворюється на лінійну послідовність інструкцій, як правило, на проміжній мові, такій як триадресний код. Подальші етапи компіляції можуть називатися чи не називатися "генерацією коду", залежно від того, чи вони включають суттєві зміни у поданні програми.

Наприклад, прохід для оптимізації очей не буде називатися "генерацією коду", хоча генератор коду може містити прохід для оптимізації окуляра.

На додаток до базового перетворення з проміжного подання в лінійну послідовність машинних інструкцій, типовий генератор коду намагається певним чином оптимізувати згенерований код.

Завдання, які зазвичай є частиною етапу "генерації коду" складного компілятора, включають:

1. Вибір інструкції: яку інструкцію використовувати.
2. Розклад інструкцій: в якому порядку розміщувати ці інструкції. Планування - це оптимізація швидкості, яка може мати критичний вплив на конвеєрні машини.
3. Розподіл регістрів: розподіл змінних до регістрів процесора
4. За потреби налагоджуйте створення даних, щоб код міг бути налагоджений.

У компіляторі, який використовує проміжну мову, може бути два етапи вибору інструкцій - один для перетворення дерева синтаксичного аналізу в проміжний код, а другий етап набагато пізніше для перетворення проміжного коду в інструкції з набору інструкцій цільової машини. Ця друга фаза не вимагає обходу дерева; це може бути зроблено лінійно, і, як правило, передбачає просту заміну операцій середньої мови відповідними кодами операцій. Однак, якщо компілятор насправді є мовним перекладачем (наприклад, таким, що перетворює Ейфеля на C), то другий етап генерації коду може передбачати побудову дерева з лінійного проміжного коду.

Коли генерація коду відбувається під час виконання, як при компіляції "точно в час" (JIT), важливо, щоб весь процес був ефективним щодо простору та часу. Наприклад, коли регулярні вирази інтерпретуються і використовуються для генерації коду під час виконання, часто не генерується детермінований кінцевий автомат замість детермінованого, оскільки зазвичай перший може бути створений швидше і займає менше місця в пам'яті, ніж другий. Незважаючи на генерацію менш ефективного коду, генерація коду JIT може скористатися перевагами профілювання інформації, яка доступна лише під час виконання.

Фундаментальне завдання прийняття введення однією мовою та отримання результату нетривіально іншою мовою можна зрозуміти з точки зору основних трансформаційних операцій формальної теорії мови. Отже, деякі методики, які спочатку були розроблені для використання в компіляторах, стали застосовуватись і іншими способами. Наприклад, YACC (ще один компілятор компілятора) бере введення у формі Бакуса-Наура і перетворює його на синтаксичний аналізатор в C. Хоча він спочатку був створений для автоматичного створення парсера для компілятора, уасс також часто використовується для автоматизації написання коду що потрібно змінювати щоразу, коли змінюються технічні характеристики.

Як правило, синтаксичний та семантичний аналізатор намагається отримати структуру програми з вихідного коду, тоді як генератор коду використовує цю структурну інформацію (наприклад, типи даних) для створення коду. Іншими словами, перший додає інформацію, а другий втрачає частину інформації. Одним із наслідків цієї втрати інформації є те, що роздуми стають важкими або навіть неможливими. Для вирішення цієї проблеми генератори коду часто вбудовують синтаксичну та семантичну інформацію на додаток до коду, необхідного для виконання.

1.2. Основні положення концепції REST API

Репрезентативна передача стану (REST) - це програмний архітектурний стиль, який визначає набір обмежень, що використовуються для створення веб-служб. Веб-служби, що відповідають архітектурному стилю REST, які називаються RESTful Web services, забезпечують взаємодію між комп'ютерними системами в Інтернеті. Веб-служби RESTful дозволяють запитуючим системам отримувати доступ до текстових подань веб-ресурсів та маніпулювати ними за допомогою єдиного та заздалегідь визначеного набору операцій без громадянства. Інші види веб-служб, такі як веб-служби SOAP, надають свої власні довільні набори операцій.

Веб-ресурси вперше були визначені у Всесвітній павутині як документи або файли, ідентифіковані за їхніми URL-адресами. Однак сьогодні вони мають набагато більш загальне та абстрактне визначення, яке охоплює кожну річ, сутність чи дію, які можна ідентифікувати, назвати, звернутись до них, обробити чи виконати будь-яким способом в Інтернеті. У веб-службі RESTful запити, надіслані на URI ресурсу, викликатимуть відповідь із корисним навантаженням, відформатованим у форматі HTML, XML, JSON або іншому форматі. Відповідь може підтвердити, що було внесено певні зміни до стану ресурсу, і відповідь може надавати гіпертекстові посилання на інші пов’язані ресурси. Коли використовується HTTP, як це найчастіше, доступні операції (методи HTTP): GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS і TRACE.

Використовуючи протокол без стану та стандартні операції, системи RESTful прагнуть до швидкої продуктивності, надійності та здатності рости шляхом повторного використання компонентів, якими можна керувати та оновлювати їх, не впливаючи на систему в цілому, навіть під час її роботи.

Термін репрезентативна передача даних був введений і визначений у 2000 році Роєм Філдінгом у своїй докторській дисертації. Дисертація Філдінга пояснила принципи REST, які були відомі як "об'єктна модель HTTP", починаючи з 1994 року, і використовувались при розробці стандартів HTTP 1.1 та Uniform Resource Identifiers (URI). Цей термін призначений для створення уявлення про те, як поводиться добре розроблений веб-додаток: це мережа веб-ресурсів (віртуальний автомат стану), де користувач просувається через додаток, вибираючи ідентифікатори ресурсів та операції з ресурсами, такі як GET або POST (переходи стану додатка), в результаті чого подання наступного ресурсу (наступний стан програми) передається кінцевому користувачеві для їх використання.

Обмеження архітектурного стилю REST впливають на наступні архітектурні властивості:

1. Ефективність взаємодії компонентів, яка може бути домінуючим фактором сприйманої користувачем продуктивності та ефективності мережі;

2. Масштабованість, що дозволяє підтримувати велику кількість компонентів та взаємодії між компонентами. Рой Філдінг описує вплив REST на масштабованість наступним чином:

3. Розділення проблем клієнта-сервера REST спрощує реалізацію компонентів, зменшує складність семантики з'єднувачів, покращує ефективність налаштування продуктивності та збільшує масштабованість чистих серверних компонентів. Багаторівневі системні обмеження дозволяють вводити посередників - проксі-сервери, шлюзи та брандмауери в різні точки зв'язку, не змінюючи інтерфейсів між компонентами, що дозволяє їм допомагати в перекладі комунікацій або покращувати продуктивність за допомогою широкомасштабного спільного кешування. REST дозволяє проміжну обробку шляхом обмеження повідомлень бути самоописовою: взаємодія не має стану між запитами, стандартні методи та типи носіїв використовуються для позначення семантики та обміну інформацією, а відповіді прямо вказують на кешування.

4. Простота єдиного інтерфейсу;

5. Модифікація компонентів для задоволення мінливих потреб (навіть під час запуску програми);

6. Видимість зв'язку між компонентами агентами обслуговування;

7. Переносимість компонентів шляхом переміщення програмного коду з даними;

8. Надійність у стійкості до відмов на системному рівні за наявності відмов всередині компонентів, з'єднувачів або даних.

Шість керівних обмежень визначають систему RESTful. Ці обмеження обмежують способи, якими сервер може обробляти та реагувати на запити клієнта, так що, працюючи в рамках цих обмежень, система набуває бажаних нефункціональних властивостей, таких як продуктивність, масштабованість, простота, можливість модифікації, видимість, портативність та надійність.

Якщо система порушує будь-яке з необхідних обмежень, її не можна вважати RESTful.

1. Клієнт-серверна архітектура. Принципом, що лежить в основі обмежень клієнт-сервер, є розділення проблем. Відокремлення проблем користувальницького інтерфейсу від проблем зберігання даних покращує портативність користувальницьких інтерфейсів на декількох платформах. Це також покращує масштабованість за рахунок спрощення серверних компонентів. Мабуть, найбільш важливим для Інтернету є те, що поділ дозволяє компонентам розвиватися незалежно, підтримуючи, таким чином, вимоги до масштабів Інтернету в декількох організаційних доменах.

2. Відсутність стану. У взаємодії клієнт-сервер стан складається з внутрішнього стану та зовнішнього стану. Власний стан, який називається станом ресурсу, зберігається на сервері і складається з інформації, яка не залежить від контексту сервера, тим самим роблячи його доступним для всіх клієнтів сервера. Зовнішній стан, який називається станом програми, зберігається на кожному клієнті і складається з інформації, яка залежить від контексту сервера і, отже, не може бути спільною. Клієнти несуть відповідальність за передачу стану додатка серверу, коли йому це потрібно. Обмеження зберігання стану програми на клієнті, а не на сервері, робить спілкування без громадянства.

3. Шарувата система. Клієнт зазвичай не може визначити, підключений він безпосередньо до кінцевого сервера або до посередника. Якщо між клієнтом і сервером розміщений проксі-сервер або балансир навантаження, це не вплине на їх зв'язок і не буде необхідності оновлювати код клієнта або сервера. Сервери-посередники можуть покращити масштабованість системи, увімкнувши балансування навантаження та надавши спільні кеші. Крім того, безпеку можна додати як шар поверх веб-служб, а потім чітко відокремити бізнес-логіку від логіки безпеки. Додавання безпеки як окремого рівня забезпечує застосування політики безпеки.

Нарешті, це також означає, що сервер може викликати кілька інших серверів, щоб генерувати відповідь клієнту.

4. Код на вимогу (за бажанням). Сервери можуть тимчасово розширити або налаштувати функціональність клієнта, передаючи виконуваний код: наприклад, скомпільовані компоненти, такі як аплети Java, або сценарії на стороні клієнта, такі як JavaScript.

5. Уніфікований інтерфейс. Єдине обмеження інтерфейсу є фундаментальним для проектування будь-якої системи RESTful. Це спрощує та роз'єднує архітектуру, що дозволяє кожній частині розвиватися самостійно. Чотири обмеження для цього єдиного інтерфейсу:

6. Ідентифікація ресурсу в запитах. Індивідуальні ресурси ідентифікуються у запитах, наприклад, використовуючи URI у веб-службах RESTful. Самі ресурси концептуально відокремлені від подань, що повертаються клієнту. Наприклад, сервер може надсилати дані зі своєї бази даних у форматі HTML, XML або як JSON - жоден з них не є внутрішнім представленням сервера.

7. Маніпулювання ресурсами через подання. Коли клієнт має представлення ресурсу, включаючи будь-які вкладені метадані, він має достатньо інформації для зміни або видалення стану ресурсу.

8. Самоописові повідомлення. Кожне повідомлення містить достатньо інформації, щоб описати, як обробити повідомлення. Наприклад, який парсер для виклику можна вказати за типом носія.

9. Гіпермедіа як двигун стану застосування (HATEOAS). Отримавши доступ до початкового URI для програми REST - аналогічно користувачеві веб-мережі, який отримує доступ до домашньої сторінки веб-сайту, клієнт REST повинен мати можливість динамічно використовувати надані сервером посилання для виявлення всіх доступних йому ресурсів. Коли доступ триває, сервер відповідає текстом, який містить гіперпосилання на інші доступні в даний час ресурси. Не потрібно, щоб клієнт чітко кодував інформацію про структуру або динаміку програми.

1.3. Основні положення концепції GraphQL

GraphQL - це мова запитів даних та мова маніпуляцій з відкритим кодом для API, і час виконання для виконання запитів із наявними даними. GraphQL був розроблений компанією Facebook у 2012 році до публічного випуску в 2015 році. 7 листопада 2018 року проект GraphQL був перенесений з Facebook у нещодавно створену програму GraphQL Foundation, розміщену неприбутковою організацією Linux Foundation. Починаючи з 2012 року, підйом GraphQL слідував термінам прийняття, як це визначив Лі Байрон, творець GraphQL, з точністю. Мета Байрона - зробити GraphQL всюдисущим на веб-платформах.

Специфікація GraphQL, яка зазвичай використовується для віддаленого зв'язку клієнт-сервер. На відміну від SQL, GraphQL є агностичним щодо джерел даних, що використовуються для отримання даних та збереження змін. Доступ до даних і маніпулювання ними здійснюється за допомогою довільних функцій, які називаються розв'язувачами. GraphQL координує та узагальнює дані цих функцій розподільника, а потім повертає результат клієнту. Як правило, ці функції вирішувача повинні делегувати рівню бізнес-логіки, відповідальному за зв'язок з різними базовими джерелами даних. Цими джерелами даних можуть бути віддалені API, бази даних, локальний кеш і майже все інше, до чого може мати доступ ваша мова програмування.

Він забезпечує підхід до розробки веб-API, його порівнювали та протиставляли REST та іншим архітектурам веб-служб. Це дозволяє клієнтам визначати структуру необхідних даних, і та сама структура даних повертається із сервера, отже, запобігаючи поверненню надмірно великих обсягів даних, але це має наслідки для того, наскільки ефективним може бути кешування результатів запиту в Інтернеті . Гнучкість та насиченість мови запитів також додає складності, яка може бути нецінна для простих API. Він складається із системи типів, мови запитів та семантики виконання, статичної перевірки та самоаналізу типу.

GraphQL підтримує читання, запис (мутацію) та підписку на зміни даних (оновлення в режимі реального часу - найчастіше реалізовані за допомогою WebHooks). Сервери GraphQL доступні для багатьох мов, включаючи Haskell, JavaScript, Perl, Python, Ruby, Java, C ++, C #, Scala, Go, Rust, Elixir, Erlang, PHP, R та Clojure.

9 лютого 2018 року мова визначення мови схеми GraphQL (SDL) стала частиною специфікації.

1.4. Технології необхідні для написання кодогенератора

JavaScript - це мова програмування, яка відповідає специфікації ECMAScript. JavaScript є високорівневим, часто встигуючим до компіляції та багатопарадигмою. Він має синтаксис фігурних дужок, динамічне введення тексту, орієнтацію на об’єкти на основі прототипу та функції першого класу.

Поряд з HTML та CSS, JavaScript є однією з основних технологій Всесвітньої павутини. JavaScript забезпечує інтерактивні веб-сторінки та є важливою частиною веб-додатків. Переважна більшість веб-сайтів використовують його для поведінки на стороні клієнта, і всі основні веб-браузери мають спеціальний механізм JavaScript для його виконання.

Як мова багатопарадигми, JavaScript підтримує керовані подіями, функціональні та імперативні стилі програмування. Він має інтерфейси прикладного програмування (API) для роботи з текстом, датами, регулярними виразами, стандартними структурами даних та об'єктною моделлю документа (DOM). Однак сама мова не включає жодного введення / виводу (вводу / виводу), наприклад мережевих, сховищних чи графічних засобів, оскільки середовище хоста (як правило, веб-браузер) забезпечує ці API.

Спочатку двигуни JavaScript використовувались лише у веб-браузерах, але зараз вони вбудовані в деякі сервери, як правило, через Node.js. Вони

також вбудовані в різноманітні програми, створені за допомогою таких фреймворків, як Electron та Cordova.

Незважаючи на те, що між JavaScript та Java є подібність, включаючи назву мови, синтаксис та відповідні стандартні бібліотеки, ці дві мови відрізняються і сильно відрізняються за дизайном.

У період домінування Internet Explorer на початку 2000-х сценарії на стороні клієнта були в стагнації. Це почало змінюватися в 2004 році, коли наступник Netscape, Mozilla, випустив браузер Firefox. Багато хто добре прийняв Firefox, взявши значну частку ринку від Internet Explorer.

У 2005 році Mozilla приєдналася до ECMA International, і розпочалась робота над стандартом ECMAScript для XML (E4X). Це призвело до того, що Mozilla працювала спільно з Macromedia (пізніше придбана Adobe Systems), які впроваджували E4X на їх мові ActionScript 3, яка базувалася на проекті ECMAScript 4. Метою стала стандартизація ActionScript 3 як нового ECMAScript 4. З цією метою Adobe Systems випустила реалізацію Tamarin як проект з відкритим кодом. Однак Tamarin та ActionScript 3 занадто відрізнялися від встановлених сценаріїв на стороні клієнта, і без співпраці з боку Microsoft ECMAScript 4 так і не досягнув результатів.

Тим часом дуже важливі події відбувались у спільнотах з відкритим кодом, не пов'язаних з роботою ECMA. У 2005 році Джессі Джеймс Гаррет випустив довідковий документ, в якому ввів термін Ajax і описав набір технологій, основою яких був JavaScript, для створення веб-додатків, де дані можна завантажувати у фоновому режимі, уникаючи необхідності розміщувати цілу сторінку перезавантажує. Це спричинило епоху відродження JavaScript, очолюваної бібліотеками з відкритим кодом та громадами, що утворилися навколо них. Було створено багато нових бібліотек, включаючи jQuery, Prototype, Dojo Toolkit та MooTools.

Google дебютував у браузері Chrome у 2008 році, використовуючи движок V8 JavaScript, який був швидшим, ніж його конкуренти. Ключовим

нововведенням стала своєчасна компіляція (JIT), тому іншим виробникам браузерів довелося переглянути свої движки для JIT.

У липні 2008 р. Ці розрізнені партії зібрались на конференцію в Осло. Це призвело до можливої домовленості на початку 2009 року про поєднання всієї відповідної роботи та просування мови вперед. Результатом став стандарт ECMAScript 5, випущений у грудні 2009 року.

Амбітна робота над мовою тривала протягом декількох років, завершившись великою колекцією доповнень та уточнень, яка була оформлена публікацією ECMAScript 6 у 2015 році.

З 2016 по 2020 рік щороку публікувалася нова версія стандарту ECMAScript, але обсяг змін був значно меншим, ніж 5-е або 6-е видання. Таким чином, JavaScript тепер можна вважати зрілою мовою, яка здебільшого прижилася.

Поточна екосистема JavaScript має безліч бібліотек та фреймворків, усталену практику програмування та розширене використання JavaScript поза веб-браузерами. Плюс, із зростанням односторінкових додатків та інших веб-сайтів, важких для JavaScript, було створено ряд перетворювачів для сприяння процесу розробки.

TypeScript - мова програмування, розроблена і підтримувана Microsoft. Є суворим синтаксичним суперсетом JavaScript і додає в мову необов 'язкову статичну типізацію. TypeScript призначений для розробки великих додатків і транскомпілює на JavaScript. Оскільки TypeScript є суперсетом JavaScript, існуючі програми JavaScript також є допустимими програмами TypeScript.

TypeScript може використовуватися для розробки JavaScript-програм як на стороні клієнта, так і на стороні сервера (як у випадку з Node.js або Deno). Існує кілька варіантів транскомпіляції. Або ви можете використовувати TypeScript за замовчуванням, або викликати компілятор Babel для перетворення TypeScript на JavaScript.

TypeScript підтримує файли визначень, які можуть містити інформацію про типи існуючих бібліотек JavaScript, так само як файли заголовків C++

можуть описувати структуру існуючих файлів об 'єктів. Це дозволяє іншим програмам використовувати значення, визначені у файлах, як якщо б вони були статично типізованими сутностями TypeScript. Існують заголовкові файли сторонніх виробників для популярних бібліотек, таких як jQuery, MongoDB і D3.js. Також доступні заголовки TypeScript для базових модулів Node.js, що дозволяють розробляти програми Node.js в TypeScript.

Компілятор TypeScript сам написаний в TypeScript і скомпільований в JavaScript. Ліцензується за ліцензією Apache License 2.0. TypeScript включений в якості першокласної мови програмування в Microsoft Visual Studio 2013 Update 2 і більше пізніх версій, поруч з C # та іншими мовами Microsoft. Офіційне розширення також дозволяє Visual Studio 2012 підтримувати TypeScript. Андерс Хайльсберг, провідний архітектор C # і творець Delphi і Turbo Pascal, працював над розробкою TypeScript.

Анотації для примітивних типів є числовими, булевими і строковими. Слабко або динамічно типізовані структури мають тип any.

Анотації типів можна експортувати в окремий файл оголошень, щоб зробити інформацію про типи доступною для сценаріїв TypeScript з використанням типів, вже скомпільованих в JavaScript. Анотації можуть бути оголошені для існуючої бібліотеки JavaScript, як це було зроблено для Node.js і jQuery.

Компілятор TypeScript використовує вивід типу для виводу типів, якщо типи не вказані. Наприклад, метод додавання у вищезгаданому коді буде виведений як повертаючий число, навіть якщо не була надана анотація поверненого типу. Це ґрунтується на статичних типах лівих і правих чисел і знанні компілятора, що результатом складання двох чисел завжди є число. Однак явне оголошення поверненого типу дозволяє компілятору перевірити правильність.

Якщо жоден тип не може бути виведений через відсутність оголошень, за замовчуванням використовується динамічний будь-який тип. Значення

будь-якого типу підтримує ті ж дії, що і значення JavaScript, і мінімальна статична перевірка типу виконується для операцій з будь-якими значеннями.

TypeScript також є мовою програмування, яка зберігає поведінку JavaScript під час виконання. Наприклад, ділення на нуль у JavaScript створює Infinity замість створення винятку під час виконання. У принципі TypeScript ніколи не змінює поведінку коду JavaScript під час виконання.

Це означає, що при переміщенні коду з JavaScript в TypeScript він буде виконуватися аналогічним чином, навіть якщо TypeScript вважає, що код має помилки типу.

Збереження тієї ж поведінки під час виконання, що і JavaScript, є основною обіцянкою TypeScript, оскільки це означає, що ви можете легко переходити між двома мовами, не турбуючись про тонкі відмінності, які можуть змусити вашу програму перестати працювати.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі був проведений аналіз технологій, за допомогою яких можна створити власний кодогенератор для API бекенда. Для створення фреймворку були обрані наступні технології.

1. GraphQL – мова запитів, яка була створена на заміну REST API, для більш зручної комунікації між клієнтом та сервером
2. JavaScript (TypeScript) – TypeScript був обраний через те, що він повністю підтримую можливості мови JavaScript, а також надає чимало нових, таких як: строгу типізацію, можливість використовувати інтерфейси, типи, та енами, краще автодоповнення тексту, що дуже зручно для написання великих додатків а також використання стороннього коду, можливість створення власного конфігураційного файлу для опису правил написання коду.

РОЗДІЛ 2

АНАЛІТИЧНИЙ ОГЛЯД GRAPHQL СХЕМ ТА ЗАПИТІВ

2.1. Основні положення щодо принципів GraphQL

GraphQL має ряд принципів проектування:

1. Ієрархічність: Більшість розробок продуктів сьогодні включає створення та маніпулювання ієрархіями поглядів. Для досягнення відповідності структурі цих додатків сам запит GraphQL структурований ієрархічно. Запит формується так само, як і дані, які він повертає. Для клієнтів це природний спосіб описати вимоги до даних.

2. Орієнтований на продукт: GraphQL не виправдовується вимогами переглядів та інтерфейсних інженерів, які їх пишуть. GraphQL починає з їхнього способу мислення та вимог і створює мову та час роботи, необхідні для цього.

3. Надійне введення: Кожен сервер GraphQL визначає специфічну для типу програми систему. Запити виконуються в контексті системи цього типу. Враховуючи запит, інструменти можуть гарантувати, що запит є як синтаксично правильним, так і дійсним у системі типу GraphQL перед виконанням, тобто під час розробки, і сервер може надати певні гарантії щодо форми та характеру відповіді.

Запити, визначені клієнтом: Завдяки своїй системі типів, сервер GraphQL публікує можливості, якими можуть користуватися його клієнти. Клієнт відповідає за те, як саме він використовуватиме ці опубліковані можливості.

4. Ці запити задаються на рівні деталізації на рівні поля. У більшості програм клієнт-сервер, написаних без GraphQL, сервер визначає дані, що повертаються в різних сценаріях кінцевих точок. Натомість запит GraphQL повертає саме те, про що запитує клієнт, і не більше того.

5. Інтроспективний: GraphQL є інтроспективним. Система типу сервера GraphQL повинна запитуватися самою мовою GraphQL, як це буде описано в цій специфікації. Самоаналіз GraphQL служить потужною платформою для побудови загальних інструментів та бібліотек клієнтського програмного забезпечення.

Завдяки цим принципам GraphQL є потужним і продуктивним середовищем для побудови клієнтських додатків. Розробники продуктів і дизайнери, що створюють додатки на робочих серверах GraphQL - за підтримки якісних інструментів - можуть швидко стати продуктивними, не читаючи обширної документації та не маючи офіційного навчання або взагалі не виконуючи його.

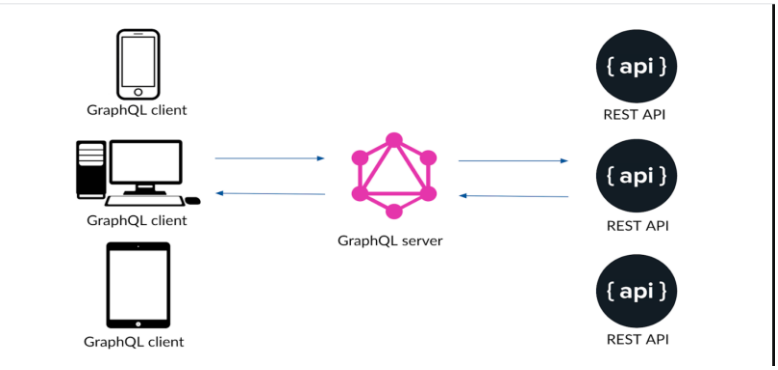

Рис.2.1. Візуальна схема GraphQL

Наступна офіційна специфікація служить посиланням для цих будівельників. Він описує мову та її граматику, систему типів та систему самоспостереження, що використовуються для її запиту, а також механізми виконання та перевірки та алгоритми їх живлення. Мета цієї специфікації - створити основу та основу для екосистеми інструментів GraphQL, клієнтських бібліотек та серверних реалізацій, що охоплює як організації, так

і платформи, які ще не створені. Ми з нетерпінням чекаємо на співпрацю з громадою для цього.

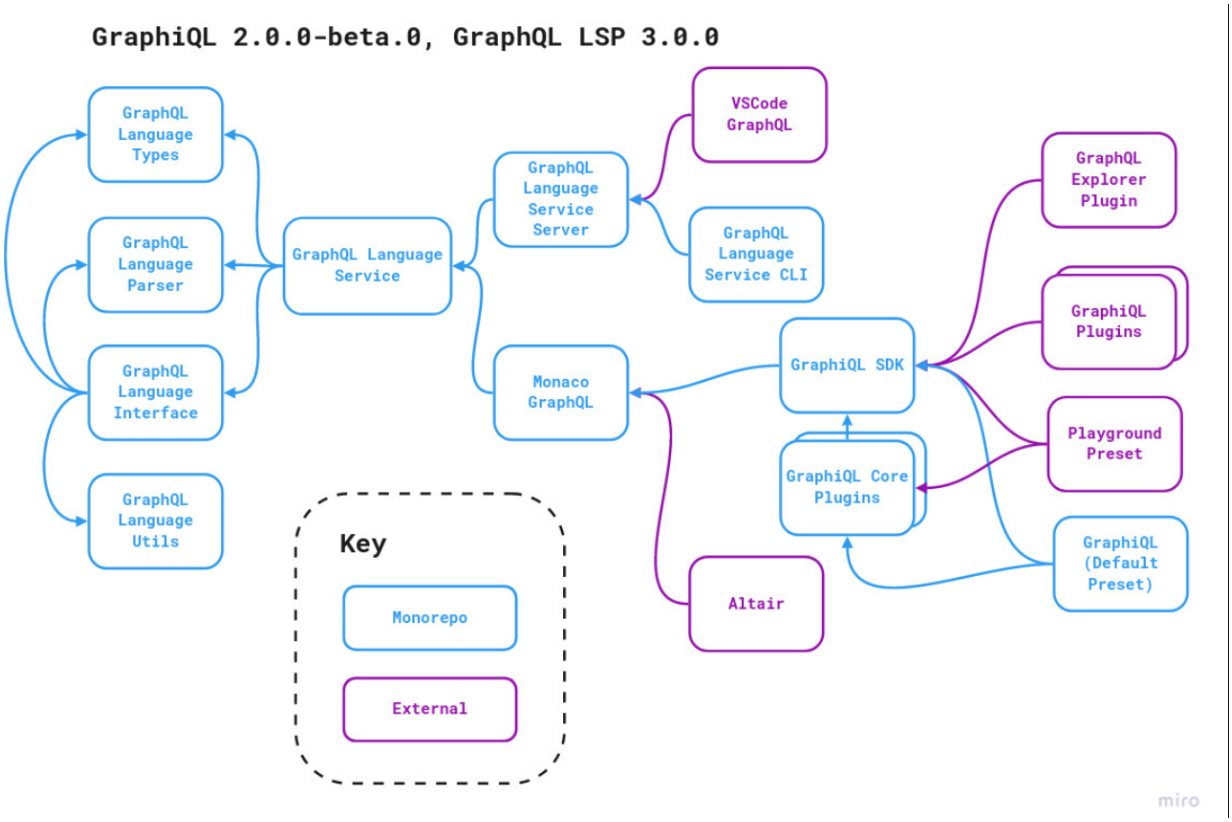

Рис.2.2. Візуалізації алгоритму обробки схем GraphQL

2.2. Мова програмування запитів GraphQL

Клієнти використовують мову запитів GraphQL для надсилання запитів до служби GraphQL. Ми посилаємось на ці джерела запитів як на документи. Документ може містити операції (запити, мутації та підписки), а також фрагменти, загальну одиницю композиції, що дозволяє повторно використовувати запит.

Документ GraphQL визначається як синтаксична граматика, де термінальними символами є лексеми (неподільні лексичні одиниці). Ці лексеми визначені в лексичній граматиці, яка відповідає шаблонам вихідних символів (визначається подвійною двокрапкою: :).

Документи GraphQL виражаються у вигляді послідовності символів Unicode. Однак, за невеликими винятками, більшість GraphQL виражається

лише в оригінальному неконтрольованому діапазоні ASCII, щоб бути якомога ширшим сумісним із якомога більшою кількістю існуючих інструментів, мов та форматів серіалізації та уникнути проблем із відображенням у текстових редакторах та керуванні джерелами.

"Позначка порядку байтів" - це спеціальний символ Unicode, який може з'являтися на початку файлу, що містить Unicode, який програми можуть використовувати для визначення факту, що текстовий потік є Unicode, якої ендіанності знаходиться текстовий потік і який з декількох Unicode кодування для інтерпретації.

Документ GraphQL складається з декількох видів неподільних лексичних лексем, визначених тут у лексичній граматиці за шаблонами вихідних символів Unicode.

Пізніше маркери використовуються як символи терміналу в синтаксичних граматиках Document GraphQL.

Пробіл використовується для поліпшення читабельності вихідного тексту та діє як розділення між маркерами, і будь-яка кількість пробілів може з’являтися до або після будь-якого маркера. Пробіл між маркерами не є значущим для семантичного значення документа GraphQL, проте символи пробілів можуть з'являтися всередині маркера String або Comment.

До і після кожного лексичного маркера може бути будь-яка кількість ігнорованих маркерів, включаючи WhiteSpace та Comment. Жодні проігноровані регіони вихідного документа не є значущими, однак ігноровані вихідні символи можуть суттєво відображатися в межах лексичного маркера, наприклад рядок може містити пробіли.

Жодні символи не ігноруються під час аналізу даного маркера, як приклад, заборонено використовувати пробіли між символами, що визначають FloatValue.

Документи GraphQL наповнені іменованими речами: операціями, полями, аргументами, типами, директивами, фрагментами та змінними. Усі імена повинні мати однакову граматичну форму.

Імена в GraphQL чутливі до регістру. Тобто ім’я, Ім’я та ІМ’Я стосуються різних імен. Підкреслення є суттєвими, що означає, що ім'я_інше та інше ім'я - це два різні імена.

Рис.2.3. Приклад запиту GraphQL

Імена в GraphQL обмежені цією підмножиною ASCII можливих символів для підтримки взаємодії з якомога більшою кількістю інших систем.

Документ GraphQL описує повний файл або рядок запиту, керований службою або клієнтом GraphQL. Документ містить декілька визначень, виконуваних або представників системи типу GraphQL.

Документи виконуються службою GraphQL, лише якщо вони містять OperationDefinition, а в іншому випадку містять лише ExecutableDefinition. Однак документи, які не містять OperationDefinition або містять TypeSystemDefinition або TypeSystemExtention, все ще можуть бути проаналізовані та перевірені, щоб дозволити клієнтським інструментам представляти багато застосувань GraphQL, які можуть відображатися в багатьох окремих файлах.

Якщо Документ містить лише одну операцію, ця операція може бути безіменною або представлена у скороченій формі, яка опускає як ключове слово запиту, так і назву операції. В іншому випадку, якщо документ GraphQL містить кілька операцій, кожна операція повинна бути названа. Під час надсилання Документу з декількома операціями до служби GraphQL також повинно бути вказано назву бажаної операції, яку потрібно виконати.

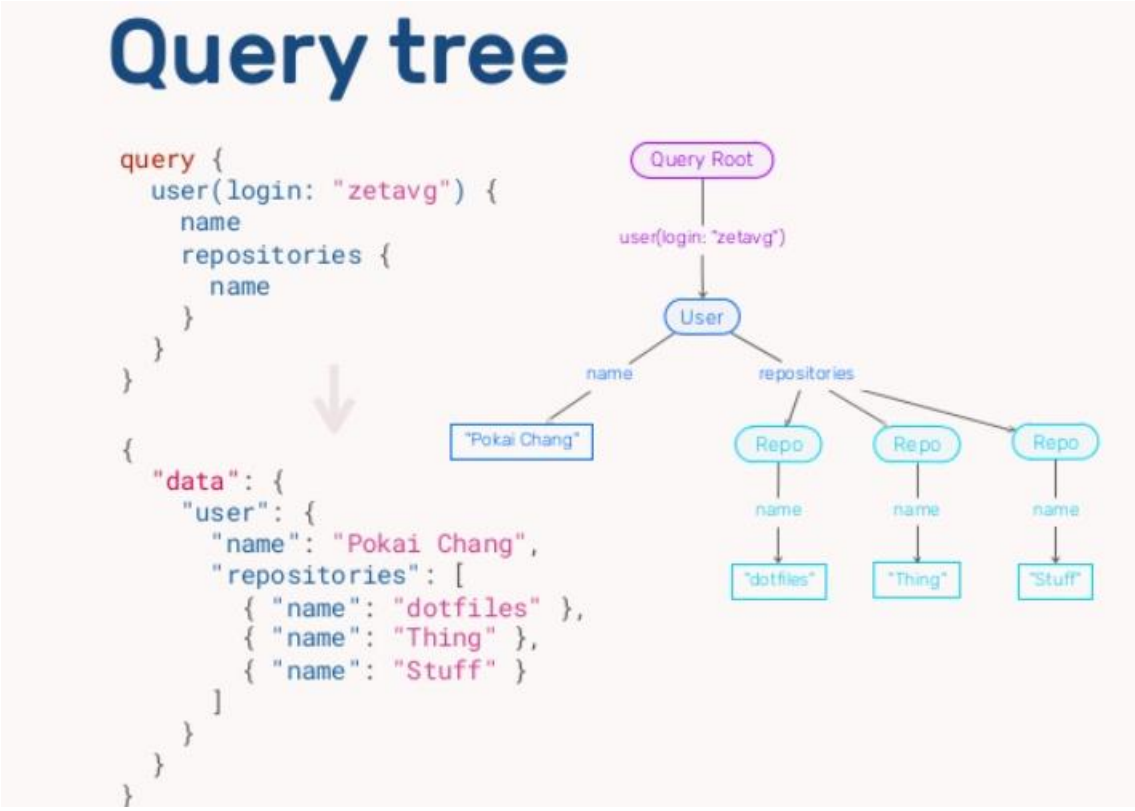

Рис.2.4. Приклад залежностей полів в схемі GraphQL

Служби GraphQL, які прагнуть лише забезпечити виконання запитів GraphQL, можуть вибрати включити лише ExecutableDefinition і опустити правила TypeSystemDefinition та TypeSystemExtension з Definition.

Директиви дають спосіб описати альтернативне виконання і поведінку перевірки типу в документі GraphQL.

Фрагменти повинні вказувати тип, до якого вони застосовуються. У цьому прикладі friendFields можна використовувати в контексті запиту Користувача.

Фрагменти не можна вказати на будь-якому вхідному значенні (скаляр, перелік або введений об'єкт). Фрагменти можна вказати для типів об'єктів, інтерфейсів та об'єднань. Виділення у фрагментах повертають значення лише тоді, коли конкретний тип об'єкта, над яким він працює, відповідає типу фрагмента. Наприклад, у цьому запиті щодо моделі даних Facebook:

```
Example № 21
query FragmentTyping {
  profiles(handles: ["zuck", "cocacola"]) {
    handle
    ...userFragment
    ...pageFragment
  }
}

fragment userFragment on User {
  friends {
    count
  }
}

fragment pageFragment on Page {
  likers {
    count
  }
}
```

Рис.2.5. Приклад фрагментів GraphQL

У деяких випадках вам потрібно надати параметри для зміни поведінки виконання GraphQL таким чином, що аргументів поля буде недостатньо, наприклад, умовно включити або пропустити поле. Директиви передбачають це, описуючи додаткову інформацію виконавцю.

```
Example № 23
query inlineFragmentTyping {
  profiles(handles: ["zuck", "cocacola"]) {
    handle
    ... on User {
      friends {
        count
      }
    }
    ... on Page {
      likers {
        count
      }
    }
  }
}
```

Рис.2.6. Приклад inline фрагментів

Директиви мають назву разом із переліком аргументів, які можуть приймати значення будь-якого типу введення. Директиви можна використовувати для опису додаткової інформації щодо типів, полів, фрагментів та операцій. Оскільки майбутні версії GraphQL застосовують нові можливості конфігурування, вони можуть бути відкриті за допомогою директив.

GraphQL не просто перевіряє, чи є запит синтаксично правильним, але також гарантує, що він є однозначним та без помилок у контексті даної схеми GraphQL. Недійсний запит як і раніше технічно виконується і завжди дасть стабільний результат, як визначено алгоритмами в розділі Виконання, однак цей результат може бути неоднозначним, несподіваним або несподіваним щодо запиту, що містить помилки перевірки, тому виконання має відбуватися лише для дійсних запитів.

Зазвичай перевірка виконується в контексті запиту безпосередньо перед виконанням, однак служба GraphQL може виконувати запит без явної перевірки, якщо відомо, що саме той самий запит перевірявся раніше. Наприклад: запит може бути перевірений під час розробки, за умови, що він згодом не зміниться, або служба може перевірити запит один раз і записати результат, щоб уникнути повторного підтвердження того самого запиту в майбутньому. Будь-який інструмент на стороні клієнта або часу розробки повинен повідомляти про помилки перевірки та не допускати формулювання або виконання запитів, про які відомо, що недійсні на даний момент часу.

Еволюція системи типів. Оскільки системна схема типу GraphQL з часом розвивається шляхом додавання нових типів та нових полів, можливо, запит, який раніше був дійсним, згодом може стати недійсним. Будь-яка зміна, яка може спричинити втрату чинності раніше дійсного запиту, вважається порушенням. Службам GraphQL та супровідникам схем рекомендується уникати порушень змін, однак для того, щоб бути більш стійкими до цих змін, складні системи GraphQL все ще можуть дозволяти виконання запитів, які в певний момент були відомі без помилок перевірки, і з тих пір не змінилися.

Якщо під час операції виникли помилки, карта відповідей повинна містити запис із ключовими помилками. Значення цього запису описано в розділі “Помилки”. Якщо операція завершена без помилок, цей запис не повинен бути присутнім.

Якщо операція включала виконання, карта відповіді повинна містити запис із ключовими даними. Значення цього запису описано в розділі «Дані». Якщо операція не вдалася до виконання через синтаксичну помилку, відсутність інформації або помилку перевірки, цей запис не повинен бути присутнім.

Карта відповідей може також містити запис із розширеннями ключів. Цей запис, якщо встановлений, повинен мати карту як значення. Цей запис зарезервований для розробників для розширення протоколу, як вони

вважають за потрібне, а отже, немає додаткових обмежень щодо його змісту. Для того, щоб майбутні зміни в протоколі не зламали існуючі сервери та клієнти, карта відповідей верхнього рівня не повинна містити жодних записів, крім трьох, описаних вище.

1. Запис помилок у відповіді є непустим списком помилок, де кожна помилка є картою.

2. Якщо під час запитуваної операції помилок не було виявлено, запис про помилки не повинен бути присутнім у результаті.

3. Якщо введення даних у відповіді відсутнє, запис про помилки у відповіді не повинен бути порожнім. Він повинен містити принаймні одну помилку. Помилки, які він містить, повинні вказувати, чому не вдалося повернути дані.

Якщо запис даних у відповіді присутній (у тому числі, якщо це значення null), запис помилок у відповіді може містити будь-які помилки, які сталися під час виконання. Якщо під час виконання сталися помилки, вони повинні містити ці помилки.

Якщо помилка може бути пов'язана з певним полем у результаті GraphQL, вона повинна містити запис із ключовим шляхом, який деталізує шлях поля відповіді, яке зазнало помилки. Це дозволяє клієнтам визначити, чи є нульовий результат навмисним чи спричинений помилкою виконання.

Це поле повинно бути списком сегментів шляху, починаючи з кореня відповіді і закінчуючи полем, пов'язаним з помилкою. Сегменти шляху, що представляють поля, повинні бути рядками, а сегменти шляху, що представляють індекси списків, мають бути 0-індексованими цілими числами. Якщо помилка трапляється у псевдонімному полі, шлях до помилки повинен використовувати псевдонім, оскільки він представляє шлях у відповіді, а не в запиті.

Якщо поле, яке спричинило помилку, було оголошено ненульовим, нульовий результат буде міхуром до наступного поля, що допускає нуль. У цьому випадку шлях до помилки повинен включати повний шлях до поля

результату, де сталася помилка, навіть якщо цього поля немає у відповіді. Наприклад, якби поле імені зверху оголосило ненульовий тип повернення у схемі, результат буде виглядати інакше, але повідомлена помилка буде однаковою:

```
Example № 185
{
  "errors": [
    {
      "message": "Name for character with ID 1002 could not be fetched.",
      "locations": [ { "line": 6, "column": 7 } ],
      "path": [ "hero", "heroFriends", 1, "name" ]
    }
  ],
  "data": {
    "hero": {
      "name": "R2-D2",
      "heroFriends": [
        {
          "id": "1000",
          "name": "Luke Skywalker"
        },
        {
          "id": "1002",
          "name": null
        },
        {
          "id": "1003",
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

Рис.2.6. Приклад формату помилки

2.3. Типізація схем GraphQL

Система GraphQL Type описує можливості сервера GraphQL і використовується для визначення того, чи правильний запит. Система типів також описує типи вхідних змінних запиту, щоб визначити, чи є значення, надані під час виконання, дійсними.

Можливі підходи щодо типізації системи:

1. SchemaDefinition

2. TypeDefinition

3. DirectiveDefinition

Мова GraphQL включає IDL, що використовується для опису системи типу служби GraphQL. Інструменти можуть використовувати цю мову визначення для надання таких службових програм, як генерація коду клієнта або обв'язка завантаження служби.

Інструменти GraphQL, які прагнуть лише забезпечити виконання запитів GraphQL, можуть вирішити не аналізувати TypeSystemDefinition.

Документ GraphQL, який містить TypeSystemDefinition, не повинен виконуватися; Служби виконання GraphQL, які отримують документ GraphQL, що містить визначення систем типу, повинні повертати описову помилку.

Розширення системи типів використовуються для представлення системи типу GraphQL, яка була розширена з якоїсь оригінальної системи типу. Наприклад, це може використовуватися локальною службою для представлення даних, до яких клієнт GraphQL має доступ лише локально, або службою GraphQL, яка сама по собі є розширенням іншої служби GraphQL.

Можливості системи колективного типу служби GraphQL називаються "схемою" цієї служби. Схема визначається з точки зору типів та директив, які вона підтримує, а також кореневих типів операцій для кожного виду операції: запиту, мутації та передплати; це визначає місце в системі типів, де починаються ці операції.

Схема GraphQL сама повинна бути внутрішньо дійсною. У цьому розділі описуються правила цього процесу перевірки, де це доречно.

Усі типи в схемі GraphQL повинні мати унікальні імена. Жоден із запропонованих типів не може мати однакову назву. Жоден із наведених типів не може мати назви, що суперечить будь-яким вбудованим типам (включаючи типи скалярів та самоаналізу).

Усі директиви в схемі GraphQL повинні мати унікальні імена. Усі типи та директиви, визначені в схемі, не повинні мати імені, яке починається на

"__" (два підкреслення), оскільки це використовується виключно системою самоаналізу GraphQL.

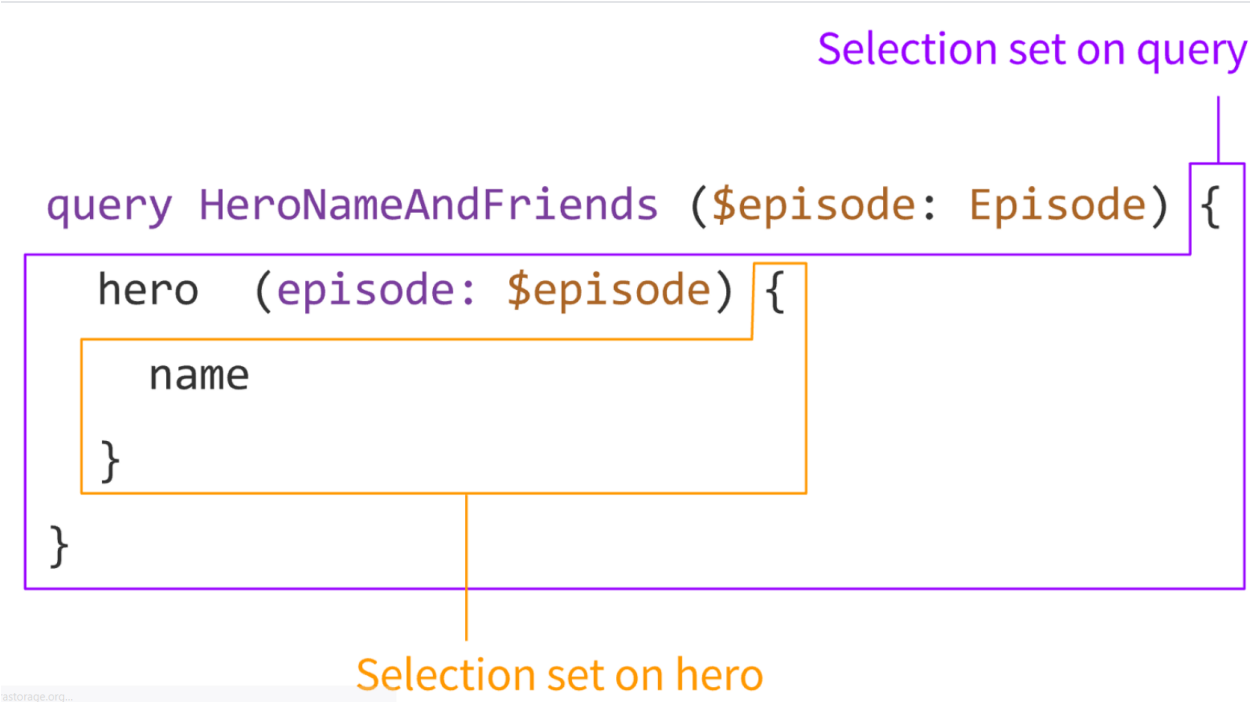

Рис. 2.7. Приклад використання типів у запиті GraphQL

Документація - це першокласна особливість систем типу GraphQL. Щоб документація служби GraphQL залишалася узгодженою з її можливостями, описи визначень GraphQL надаються поряд із їх визначеннями та доступні за допомогою самоаналізу.

Щоб дизайнери служб GraphQL могли легко публікувати документацію поряд із можливостями служби GraphQL, описи GraphQL визначаються за допомогою синтаксису Markdown (як зазначено CommonMark). У мові визначення системи типу, ці рядки опису (часто BlockString) виникають безпосередньо перед визначенням, яке вони описують.

Усі типи GraphQL, поля, аргументи та інші визначення, які можна описати, повинні містити Опис, якщо вони не вважаються самоописовими

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі був проведений аналіз структури мови та структури схем бази данних мови запитіва GraphQL. Мому можна виділити основні причини, через які використання цієї мови є важливим та практичним:

1. Добре підходить для складних систем та мікросервісів. Інтегруючи кілька систем за своїм API, GraphQL об'єднує їх і приховує їх складність. Потім сервер GraphQL відповідає за отримання даних із існуючих систем та упакування їх у формат відповіді GraphQL.

2. Отримання даних за допомогою одного виклику API. Основна відмінність GraphQL від REST полягає в тому, що остання зосереджена навколо окремих кінцевих точок, тому для збору всіх необхідних даних розробник повинен поєднати кілька кінцевих точок. Тоді як GraphQL зосереджується на самому завданні, у цьому випадку розробник може запитувати необхідні дані лише за допомогою одного виклику API.

3. Немає проблем із надмірним та недостатнім завантаженням. Відповіді REST відомі тим, що містять занадто багато даних або їх недостатньо, що створює потребу в іншому запиті.

4. Пристосування запитів до потреб. Як уже зазначалося, документація REST API пояснює окремі кінцеві точки, їх функції та параметри, які розробник може передавати їм

РОЗДІЛ 3

РЕАЛІЗАЦІЯ КОДОГЕНЕРАТОРА

3.1. Опис елементів системи

Система складається з умовного ядра, та плагінів. Ядро програми відповідає виконання кроків, необхідних для створення авто сгенерованого файлу, а саме:

1. Загрузка файла із схемою (через http або локальний файл)
2. Перетворення схеми у формат, зручний для обробки
3. Створення об’єкту с відповідними типами
4. Створення файлу с генерованими типами та методами виклику
5. Виклик плагінів на кожному кроці з відповідним контекстом виклику

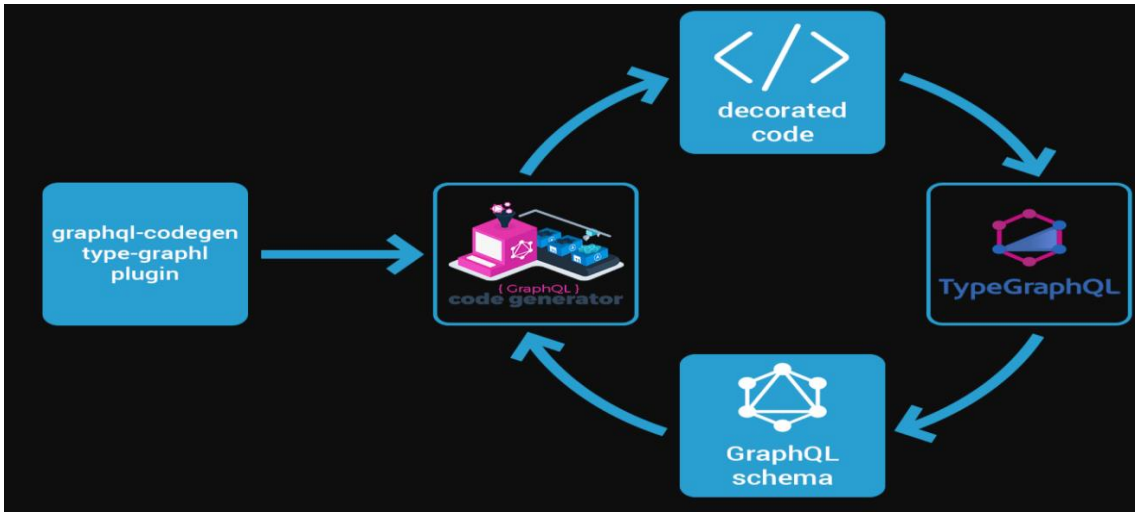

Рис. 3.1. Порядок оброблення схеми

У свою чергу плагіни можуть змінювати дані, які будут передаватися у наступний крок. Тобто, вони існують для того, щоб була можливість як

завгодно змінювати формат даних, та модіфікувати його як необхідно. На кожному кроці генерації файлу с типами та методами виклику буде виклиний відповідний плагін, якщо він був зарегестрований, і йому буде переданий контекст виклику (який необхідний саме для цього кроку, де зарегестрований плагін).

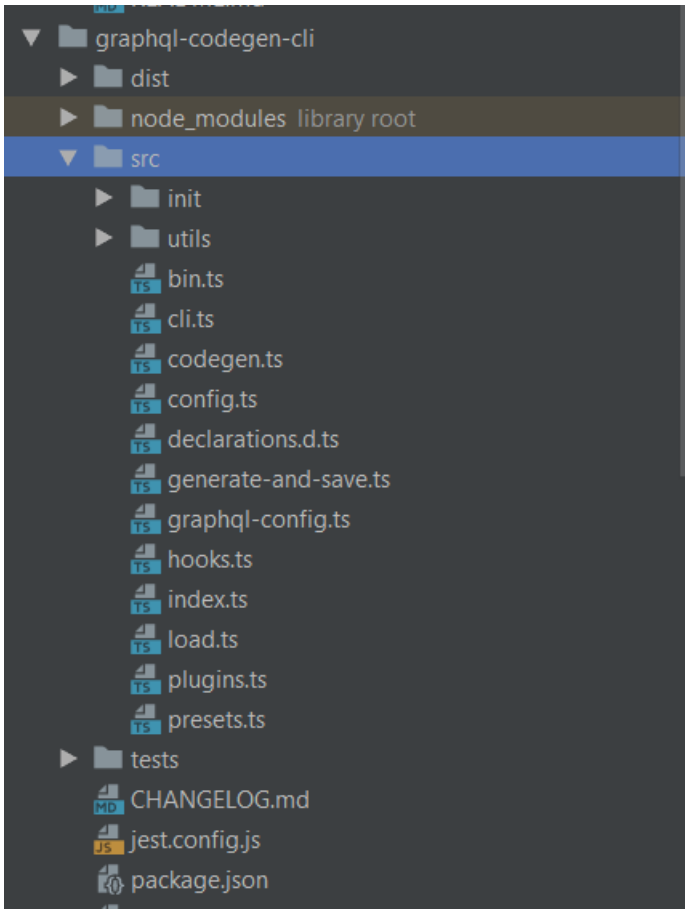

Рис. 3.2. Структура GraphQL CLI

Також існує спеціалізований плагін, створенний системою, який контролює порядок виклику інших плагінів, які можуть додатково змінювати формат данних – execute-plugin. Важливим є те, що кожен крок , а саме плагін, якій змінює дані, має реалізовувати певний інтерфейс, для того щоб бути впевненими, що плагін на наступному кроці буде знати які саме дані він отримає, я що має віддати на виході із функції.

Дуже важливою частиною системи є CLI, за допомогою якого можна спілкуватися із системою, та виконувати ряд команд, а саме:

1. config – визначає файл конфігурації

2. watch – перевизначає поле watch у конфигураційному файлі, та дозволяє отримувати новий сгенерований файл, без необхідності запускати команду сбірки

3. silent – дозволяє не відображати помилки при генерації

4. errors-only – показувати лише помилки при генерації

5. require – конфігурації, які необхідно запровадити усім плагінам, які використовуються при генерації файлу.

6. overwrite – створення нового файлу, заміть перезаписування вже існуючого.

Повний список підтримуваних параметрів, які ви можна визначити у файлі конфігурації:

1. schema (обов’язково) - URL-адреса до вашої кінцевої точки GraphQL, локальний шлях до файлу .graphql, шаблон глобуса до ваших файлів схеми GraphQL або файл JavaScript, який експортує схему для генерації коду. Це також може бути масив, який визначає кілька схем, з яких генерується код. Детальніше про підтримувані формати можна прочитати тут.

2. документи - масив шляхів або шаблонів глобусів для файлів, які експортують документи GraphQL за допомогою тегу gql або простого рядка; наприклад: ./src/**/*.graphql. Ви також можете надати ці параметри рядку замість масиву, якщо ви маєте справу з одним документом. Детальніше про підтримувані формати можна прочитати тут.

3. generate (обов’язково) - Карта, де ключ представляє вихідний шлях для згенерованого коду, а значення представляє набір параметрів, які мають значення для цього конкретного файлу. Нижче наведені можливі варіанти, які можна вказати:

4. generates.plugins (обов’язково) - Список плагінів, які слід використовувати під час створення файлу. Шаблони також розглядаються як плагіни, і їх можна вказати в цьому розділі. Повний список підтримуваних

плагінів можна знайти тут. Ви також можете вказати на спеціальний плагін у локальному файлі (див. Користувацькі плагіни).

5. generates.schema - Те саме, що і коренева схема, але застосовується лише до певного вихідного файлу.

6. generates.documents - Те саме, що і кореневі документи, але застосовується лише до певного вихідного файлу.

7. generates.config - Те саме, що і коренева конфігурація, але застосовується лише до певного вихідного файлу.

8. generates.overwrite - Те саме, що і перезапис root, але застосовується лише до певного вихідного файлу.

9. require - Шлях до файлу, який визначає власні обробники Node.JS require () для власних розширень файлів. Це важливо, якщо генератор коду повинен переглядати файли, для яких потрібні інші файли, у непідтримуваному форматі (за замовчуванням). Дивіться більше інформації. Зверніть увагу, що значення, вказані у вашому файлі .yml, завантажуватимуться після завантаження файлу .yml.

10. config - Параметри, які ми хотіли б надати вказаним плагінам. Параметри можуть відрізнятися залежно від того, які плагіни ви вказали. Для отримання додаткової інформації прочитайте документацію до цього плагіна. Детальніше про те, як передати конфігурацію плагінам, ви можете прочитати тут.

11. overwrite - прапор для перезапису файлів, якщо вони вже існують під час генерації коду (істинно за замовчуванням).

12. watch -прапор, який запускає кодоген, коли змінюються зазначені схеми GraphQL. Ви можете вказати логічне значення, щоб увімкнути / вимкнути його, або вказати масив шаблонів глобусів, щоб додати власні файли до годинника.

13. silent - прапор для придушення помилок друку, коли вони виникають.

14. errorsOnly - прапор для придушення друку будь-чого, крім помилок.

15. hooks - Вказує сценарії, які запускатимуться, коли події відбуваються в ядрі кодегену. Детальніше про гачки життєвого циклу ви можете прочитати тут. Ви можете вказати це у своїй кореневій конфігурації або в кожному вихідному файлі.

16. pluginLoader - Якщо ви використовуєте програмний API у середовищі браузера, ви можете замінити цю конфігурацію, щоб завантажувати свої плагіни способом, відмінним від необхідного.

17. pluckConfig - Дозволяє замінити конфігурацію для graphql-tag-pluck, інструменту, який витягує ваші операції GraphQL з файлів коду.

18. pluckConfig.modules - масив із {name: рядок, ідентифікатор: рядок}, який буде використовуватися для відстеження використання та імпортування gql. Використовуйте це, якщо файли коду імпортують gql з іншої бібліотеки або у вас є власний тег gql. ідентифікатор - це названий експорт, тому не вказуйте його, якщо функція тегу імпортується за замовчуванням.

19. pluckConfig.gqlMagicComment - Налаштовує чарівні коментарі GraphQL для пошуку. Типовим значенням є / * GraphQL * /).

20. pluckConfig.globalGqlIdentifierName - замінює ім'я ідентифікатора імені GraphQL за замовчуванням.

3.2. Тестування системи

Jest - це чудова платформа для тестування JavaScript, орієнтована на простоту, яка дозволяє протестувати будь-який аспект системи для генерування коду.

Кожен плагін, а також ядро системи повинно покриватися jest тестами, а саме:

1. Виклик кожного метода має бути перевіреним. Необхідно передавати дані, згідно задекларованого інтерфейсу метода, та порівнювати дані, які поверне цей метод з очікуваним результатом.

2. Необхідно створювати mock файли, за допомогою яких можна бути впевненим у даних, за допомогою яких ми тестуємо.

3. У випадку, якщо під час проходження тестів хоча б один не є успішним, або якщо є помилки під час транспіляції коду, то дана версія проекту не є валідною.

```
const { resolve } = require('path');
const { pathsToModuleNameMapper } = require('ts-jest/utils');

const ROOT_DIR = __dirname;
const TSCONFIG = resolve(ROOT_DIR, 'tsconfig.json');
const tsconfig = require(TSCONFIG);
const CI = !!process.env.CI;

module.exports = ({ dirname, projectMode = true }) => {
  const pkg = require(resolve(dirname, 'package.json'));

  return {
    ...(CI || !projectMode ? {} : { displayName: pkg.name.replace('@graphql-codegen/', '') }),
    transform: { '^.+\\.tsx?$': 'ts-jest' },
    testEnvironment: 'node',
    rootDir: dirname,
    globals: {
      'ts-jest': {
        diagnostics: false,
        tsconfig: 'tsconfig.json',
      },
    },
    restoreMocks: true,
    reporters: ['default'],
    modulePathIgnorePatterns: ['dist'],
    moduleNameMapper: pathsToModuleNameMapper(tsconfig.compilerOptions.paths, { prefix: `${ROOT_DIR}/` }),
    cacheDirectory: resolve(ROOT_DIR, `${CI ? '' : 'node_modules/'}.cache/jest`),
    setupFiles: [`${ROOT_DIR}/dev-test/setup.js`],
    collectCoverage: false,
  };
};
```

Рис. 3.3. Конфігураційний файл jest

Дуже важливо, щоб розробка була швидкою. Відомо, що считування файлів є повільною операцією, саме тому був зроблений mock модуля файлової системи node js, виключно для тестування.

Рис. 3.4. Mock файлової системи nodejs

Для тестування асинхронного коду є декілька опцій. Найпоширенішим асинхронним шаблоном є зворотні дзвінки.

Наприклад, скажімо, у вас є функція fetchData (зворотний виклик), яка отримує деякі дані та викликає зворотний виклик (дані), коли вона завершена. Ви хочете перевірити, що ці повернуті дані є рядком "дані-1".

За замовчуванням тести Jest завершуються, коли вони досягають кінця свого виконання. Проблема в тому, що тест завершиться, як тільки fetchData завершиться, перш ніж коли-небудь викликати зворотний виклик. Існує альтернативна форма тесту, яка це виправляє. Замість того, щоб поміщати тест у функцію з порожнім аргументом, використовуйте один аргумент, який називається done. Jest зачекає, поки не буде викликаний зворотний виклик, перш ніж закінчити тест.

Якщо ніколи не викликати done (), тест не вдасться (з помилкою тайм-ауту), саме цього ви і хочете. Оператор очікування не вдається, він видає помилку, і done () не викликається. Якщо ми хочемо побачити в тестовому журналі, чому він не вдався, нам слід обернути очікуваний блок спроб і передати помилку в блоці catch готовому. В іншому випадку ми отримуємо непрозору помилку тайм-ауту, яка не показує, яке значення було отримано очікуванням (дані).

Jest можна використовувати в проектах, які використовують webpack для управління активами, стилями та компіляцією. webpack пропонує деякі

унікальні завдання порівняно з іншими інструментами, оскільки він інтегрується безпосередньо з вашим додатком, дозволяючи керувати таблицями стилів, ресурсами, такими як зображення та шрифти, а також розширюваною екосистемою мов та інструментів компіляції в JavaScript.

Власні функції таймера (тобто setTimeout, setInterval, clearTimeout, clearInterval) є менш ніж ідеальними для середовища тестування, оскільки вони залежать від реального часу. Jest може поміняти таймери функціями, які дозволяють контролювати хід часу.

Інша можливість використовувати jest.advanceTimersByTime (msToRun). Коли викликається цей API, усі таймери просуваються на мілісекунди msToRun. Будуть виконані всі очікувані "макрозадачі", які були поставлені в чергу через setTimeout () або setInterval () і будуть виконані протягом цього періоду часу. Крім того, якщо ці макрозадачі планують нові макрозадачі, які будуть виконуватися за той самий часовий проміжок, вони будуть виконуватися до тих пір, поки в черзі не залишиться більше макрозавдань, які слід запускати протягом msToRun мілісекунд.

3.3. Технології необхідні для побудови проекту

Для збірки фреймворку використовується технологія bob.

Bob - інструмент побудови на основі конвенцій для проектів node.js. Боб пропонує набір завдань, пов’язаних зі збіркою, які працюють на різних платформах і прості у використанні, дотримуючись кількох домовленостей. Він працює з нульовою конфігурацією та дозволяє мінімальну настройку, коли ви не хочете використовувати тип певного завдання за замовчуванням. Він встановлює лише інструменти за замовчуванням, тоді як альтернативні інструменти встановлюватимуться ліниво за необхідності. Він не має плагінів. Він використовує різні інструменти CLI та налаштовує їх використання у файлах конфігурації завдань.

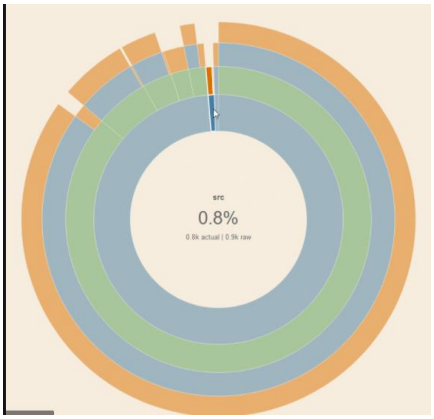

Рис. 3.5. Перетворення вхідних файлов за допомогою bob

Спілкування між системою та Bob js відбувається за допомогою задач, які необхідно ставити:

1. clean – видалення сгенерованних файлів
2. complexity – встановлення складності перевірки js файлів
3. coverage – перевірка покриття коду тестами
4. dep – встановлення залежностей системи
5. depgraph – генерує модуль залежностей у вигляді графа
6. doc – генерує документацію коду
7. lint – запускає лінтер коду
8. nuke – завершення усіх процесів проекту
9. package – створення пакету
10. publish – публікація пакету
11. restart – перезапускає процеси
12. rmdep – показує карту запущених процесів
13. stop – поставити на паузу процеси
14. test – запуск тестів системи
15. updep – обновити усі залежності
16. versionup – підняти версію проекта

Рис. 3.6. Конфігураційний файл tsconfig.json

Також, щоб кодова база складалася із коду, з єдиним стилем, використовується eslint. ESLint статистично аналізує ваш код для швидкого пошуку проблем. ESLint вбудований у більшість текстових редакторів, і ви можете запустити ESLint як частину конвеєра безперервної інтеграції. Багато проблем, які знаходить ESLint, можна виправити автоматично. Виправлення ESLint відповідають синтаксису, тому ви не будете відчувати помилок, спричинених традиційними алгоритмами пошуку та заміни. Попередньо обробіть код, використовуйте власні парсери та напишіть власні правила, які працюють поряд із вбудованими правилами ESLint. Ви можете налаштувати ESLint так, щоб він працював саме так, як вам потрібно для вашого проекту.

Рис. 3.7. Конфігураційний файл eslint

ВИСНОВОК ДО РОЗДІЛУ 3

Система кодогенерації складається с ядра програми, та плагінів. Ядро відповідає за наступні кроки:

1. Загрузка файла із схемою (через http або локальний файл)
2. Перетворення схеми у формат, зручний для обробки
3. Створення об’єкту с відповідними типами
4. Створення файлу с генерованими типами та методами виклику
5. Виклик плагінів на кожному кроці з відповідним контекстом виклику

Система плагінів використовується для модіфікації формату даних, які передаються на кожному кроці, важливим є те, щоб кожен плагін відповідам інтерфейсу даних, на тому кроцу, де він використовується.

Для спілкування між програмою та користувачем використовується CLI, який надає декілька опціональних прапорів для створення більш точної команди запиту, для генерації файлів з типами та методами для отрмання даних з API.

РОЗДІЛ 4

ГЕНЕРАЦІЯ КОДУ ЗА ДОПОМОГОЮ СТВОРЕННОЇ СИСТЕМИ

4.1. Написання серверної частини

Для написання серверної части, будемо використовувати GraphQL Apollo-Server.

Створення файлу, який описує моделі нашої БД.

```
import { ApolloServer } from "apollo-server";
import * as bookModel from "./models/book";
import * as characterModel from "./models/character";
import * as houseModel from "./models/house";
import * as tvSeriesModel from "./models/tv-series";
import { resolvers } from "./resolvers";
import { schema } from "./schema";

export interface Context {
  models: {
    character: typeof characterModel;
    house: typeof houseModel;
    tvSeries: typeof tvSeriesModel;
    book: typeof bookModel;
  };
}

const context: Context = {
  models: {
    character: characterModel,
    house: houseModel,
    tvSeries: tvSeriesModel,
    book: bookModel
  }
};

const server = new ApolloServer({
  typeDefs: schema,
  resolvers,
  context
});

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

Рис. 4.1. Опис моделей сервера

Кафедра КІТ (47)

Створення опис GraphQL схем, які будуть безпосередньо зв’язані зі створенними типами.

```
export const schema = gql`
  type Query {
    getCharacters(sortDirection: SortDirection): [Character!]!
    getCharacter(characterId: ID!): Character
    getHouses(sortDirection: SortDirection): [House!]!
    getHouse(houseId: ID!): House
  }

  type Character {
    id: ID!
    name: String!
    culture: String
    titles: [String!]
    aliases: [String!]
    born: String
    died: String
    father: Character
    mother: Character
    spouse: Character
    children: [Character!]
    allegiances: [House!]
    appearedIn: [TvSeason!]!
    isAlive: Boolean!
    playedBy: String
    books: [Book!]
  }
```

Рис. 4.2. Опис загального типу Query і типу Character

Також необхідно типізувати кожен тип, який буде використовуватися у нашій БД. Після чого необхідно написати функції резолвери, які будуть робити запити до БД, і віддавати необхідні дані клієнту. Важливим є те, що б у клієнта нашого серверу була можливість запрошувати поля які йому необхідні – самостійно, але, при цьому сервер має контролювати , щоб клієнт не отримав ті дані, до яких у нього немає доступу. За цю роль у GraphQL можуть відповідати guards (захисники). При спробі отримати ті дані, до яких у клієнта не має доступу, у відповіді клієнт отримає відповідне повідомлення, але якщо необхідні права доступу є – клієнт отримає саме ті дані, які він запитує.

```
  type TvSeason {
    id: ID!
    startDate: String!
    endDate: String!
    name: String!
    characters: [Character!]!
  }

  type House {
    id: ID!
    name: String!
    titles: [String!]
    members: [Character!]!
    slogan: String
    overlord: Character
    currentLord: Character
    founder: Character
    ancestralWeapons: [String!]
    coatOfArms: String
    seats: [String!]
  }

  type Book {
    id: ID!
    name: String!
    releaseDate: String!
  }

  enum SortDirection {
    ASC
    DESC
  }
`;
```

Рис. 4.3. Опис Типів книга, сортування

Ми повинні визначити властивості нашого об'єкта-розв'язувача, щоб TypeScript знав, чого очікувати (Запит, Символ тощо). Для кожного з цих властивостей (функцій резольвера) ми повинні вручну визначити визначення типів для всіх параметрів.

```
const express = require('express')
const {
  GraphQLObjectType,
  GraphQLSchema,
  GraphQLString
} = require('graphql');
```

```
const {
  getGraphQLParameters,
  processRequest,
  renderGraphiQL,
  shouldRenderGraphiQL
} = require('graphql-helix');

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      hello: {
        type: GraphQLString,
        resolve: () => 'Hello world!',
      },
    },
  }),
});

const app = express();

app.use(express.json());

app.use('/graphql', async (req, res) => {
  const request = {
    body: req.body,
    headers: req.headers,
    method: req.method,
    query: req.query,
  };

  if (shouldRenderGraphiQL(request)) {
    res.send(renderGraphiQL());
  } else {
    const { operationName, query, variables } =
getGraphQLParameters(request);

    const result = await processRequest({
      operationName,
      query,
      variables,
      request,
      schema,
    });

    if (result.type === 'RESPONSE') {
      result.headers.forEach(({ name, value }) => res.setHeader(name,
value));
      res.status(result.status);
      res.json(result.payload);
    } else {
    // graphql-helix also supports subscriptions and incremental delivery
(i.e. @defer and @stream directives)
    // out of the box. See the repo for more complete examples that also
implement those features.
    }
  }
```

```
});

app.listen(4000, () =>
  console.log('Now browse to http://localhost:4000/graphql');
)
```

З використанням php:

```
<?php
declare(strict_types=1);
require_once '/path/to/vendor/autoload.php';

use SilerDiactoros;
use SilerGraphql;
use SilerHttp;

$typeDefs = file_get_contents(__DIR__.'/schema.graphql');
$resolvers = [
    'Query' => [
        'hello' => 'world',
    ],
];
$schema = Graphqlschema($typeDefs, $resolvers);

echo "Server running at http://127.0.0.1:8080";

Httpserver(Graphqlpsr7($schema), function (Throwable $err) {
    var_dump($err);
    return Diactorosjson([
        'error'   => true,
        'message' => $err->getMessage(),
    ]);
})()->run();
class ProductController
{
    /**
     * @Query()
     */
    public function product(string $id): Product
    {
        // Some code that looks for a product and returns it.
    }
}

/**
 * @Type()
 */
class Product
{
    /**
     * @Field()
     */
    public function getName(): string
    {
        return $this->name;
    }
    // ...
```

```
}
```

З використанням python:

```
pip install graphene
import graphene

class Query(graphene.ObjectType):
  hello = graphene.String(name=graphene.String(default_value="World"))

  def resolve_hello(self, info, name):
    return 'Hello ' + name

schema = graphene.Schema(query=Query)
result = schema.execute('{ hello }')
print(result.data['hello']) # "Hello World"
from ariadne import ObjectType, QueryType, gql, make_executable_schema
from ariadne.asgi import GraphQL
# Define types using Schema Definition Language
(https://graphql.org/learn/schema/)
# Wrapping string in gql function provides validation and better error
traceback
type_defs = gql("""
    type Query {
        hello: String!
    }
""")
# Bind resolver functions to Query's fields using QueryType
query_type = QueryType()
# Resolvers are simple python functions
@query_type.field("hello")
def resolve_hello(*_):
    return "Hello world!"
# Create executable GraphQL schema
schema = make_executable_schema(type_defs, query_type)
# Create an ASGI app using the schema, running in debug mode
app = GraphQL(schema, debug=True)
query {
    movies (directorName: "Vince Gilligan", first: 3) {
        edges {
            node {
                id
                name
                rating
                releaseDate
            }
        },
        pageInfo {
            endCursor
            startCursor
            hasNextPage
            hasPreviousPage
        }
    }
}

     from sgqlc.types.relay import Node, Connection
```

```
from sgqlc.types.datetime import Date
from sgqlc.types import String, Float, Type, Field, list_of
class MovieNode(Node):
    name = String
    rating = Float
    release_date = Date
class MovieEdge(Type):
    node = Field(MovieNode)
class MovieConnection(Connection):

from sgqlc.types.relay import Node, Connection
from sgqlc.types.datetime import Date
from sgqlc.types import Int, String, Float, Type, Field, list_of
class MovieNode(Node):
    name = String
    rating = Float
    release_date = Date
class MovieEdge(Type):
    node = Field(MovieNode)
class MovieConnection(Connection):
    edges = list_of(MovieEdge)
class Query(Type):
    movies = Field(
        MovieConnection,
        args={
            'director_name': String,
            'first': Int
        }
    )
from sgqlc.operation import Operation
from sgqlc.endpoint.http import HTTPEndpoint
from .schema import Query
query = Operation(Query)
query.movies(director_name='Vince Gilligan', first=3)

from sgqlc.operation import Operation
from sgqlc.endpoint.http import HTTPEndpoint
from .schema import Query
query = Operation(Query)
query.movies(director_name='Vince Gilligan', first=3)
query.movies.edges()  # Attaches all fields (along with nodes and edges)
```

У резолверах перший аргумент, який зазвичай називають параметром “root” або “parent”, має інший тип, залежно від того, над яким засобом вирішення проблеми ми працюємо. У цьому прикладі ми бачимо будь-які символи, TvSeries та House, залежно від того, з яким батьківським типом ми працюємо. Другий аргумент містить аргументи запиту GraphQL, які різні для кожного розподільника. У цьому прикладі деякі є будь-якими, інші беруть sortDirection, а деякі приймають ідентифікатор. Третій аргумент, контекст, на щастя однаковий для кожного вирішувача.

Express.js- це внутрішня веб-програма для Node.js, випущена як безкоштовне програмне забезпечення з відкритим кодом під ліцензією MIT. Він

призначений для створення веб-додатків та API. Його називали фактичним стандартним серверним фреймворком для Node.js.

Оригінальний автор, TJ Holowaychuk, описав його як сервер, натхненний Sinatra, що означає, що він відносно мінімальний з багатьма функціями, доступними як плагіни. Express - це внутрішній компонент стеку MEAN, разом із програмним забезпеченням для баз даних MongoDB та інтерфейсною структурою AngularJS.

4.2. Написання клієнтської частини

Клієнтська частина буде реалізована за допомогою create-react-app і Apoll-client для комунікації із сервером. У прикладі у інтерфейсі буде відображатися список акторів, що грали роль у фільми.

```
const Detail: React.FC<{
  character: CharacterDetail;
  select: (characterId: number) => void;
}> = ({ character, select }) => {
  return (
    <>
      <h2>{character.name}</h2>
      {character.allegiances && character.allegiances.length > 0 && (
        <div>
          <strong>Loyal to</strong>:{" "}
          {character.allegiances.map(allegiance => allegiance.name).join(", ")}
        </div>
      )}
      {renderItem("Culture", character.culture)}
      {renderItem("Played by", character.playedBy)}
      {renderListItem("Titles", character.titles)}
      {renderListItem("Aliases", character.aliases)}
      {renderItem("Born", character.born)}
      {renderItem("Died", character.died)}
      {renderItem("Culture", character.culture)}
      {renderCharacter(select, "Father", character.father)}
      {renderCharacter(select, "Mother", character.mother)}
      {renderCharacter(select, "Spouse", character.spouse)}
      {character.children && character.children.length > 0 && (
        <div>
          <strong>Children</strong>:{" "}
          {character.children.map(child => (
            <>
              <a href="#" onClick={() => select(parseInt(child.id))}>
                {child.name}
              </a>{" "}
            </>
          ))}
        </div>
      )}
      {renderListItem(
        "TV Seasons",
        character.appearedIn ? character.appearedIn.map(x => x.name) : []
      )}
      {renderListItem(
        "Books",
        character.books ? character.books.map(x => x.name) : []
      )}
    </>
  );
};
```

Рис. 4.4. Реалізція відображання інтерфейсу актора

Також, необхідно створити файли налаштування, які необхіді для системи кодогенерації, а саме, необхідно повідомити про папку, де треба створити файл, передати опції щодо сгенерованих методів, для виклику методів серверу, а також опції щодо налаштування Typescript.

```
schema: ./server/schema.ts
generates:
  server/gen-types.ts:
    config:
      defaultMapper: any
      contextType: ./#Context
    plugins:
      - typescript
      - typescript-resolvers
  ./client/src/gen-types.tsx:
    documents: ./client/src/queries/*.tsx
    plugins:
      - add: /* eslint-disable */
      - typescript
      - typescript-operations
      - typescript-react-apollo
```

Рис. 4.5. Налаштування клієнтської частини

Методи, необхідні для відображення інтерфейсу.

1. GQLHooks.Fragments.Actor.useQueryById(/* params */) to query by id
2. GQLHooks.Fragments.Actor.useQueryByIdLazy(/* params */) Hook to query by id - lazy execution
3. GQLHooks.Fragments.Actor.useQueryObjects(/* params */) Hook to query one or multiple entities
4. GQLHooks.Fragments.Actor.useQueryObjectsLazy(/* params */) Hook to query one or multiple entities - lazy execution
5. GQLHooks.Fragments.Actor.useSubscribeToById(/* params */) Hook to subscribe by id
6. GQLHooks.Fragments.Actor.useSubscribeToObjects(/* params */) Hook to subscribe to one or multiple entities

7. GQLHooks.Fragments.Actor.useInsert(/* params */) Hook to insert single entity

8. GQLHooks.Fragments.Actor.useInsertWithOnConflict(/* params */) Hook to insert with strongly types onconflict options

9. GQLHooks.Fragments.Actor.useInsertObjects(/* params */) Hook to insert one or multiple entities

10. GQLHooks.Fragments.Actor.useUpdateById(/* params */) Hook to update by id

11. GQLHooks.Fragments.Actor.useUpdateObjects(/* params */) Hook to update one or multiple entities

12. GQLHooks.Models.Actor.useRemoveById(/* params */) Hook to delete entity by id

13. GQLHooks.Models.Actor.useRemoveObjects(/* params */) Hook to delete one or multiple entities

Для написання клієнтської частини коду використовується React. React - це відкрита, інтерфейсна бібліотека JavaScript для побудови користувальницьких інтерфейсів або компонентів інтерфейсу. Він підтримується Facebook та спільнотою окремих розробників та компаній. React можна використовувати як основу при розробці односторінкових або мобільних додатків. Однак React займається лише наданням даних у DOM, і тому створення React-програм зазвичай вимагає використання додаткових бібліотек для управління станом та маршрутизації. Redux та React Router є відповідними прикладами таких бібліотек.

Код реагування складається з сутностей, які називаються компонентами. Компоненти можуть бути відтворені до певного елемента в DOM за допомогою бібліотеки React DOM. Під час візуалізації компонента можна передавати значення, які відомі як "реквізит"

Ще однією помітною особливістю є використання віртуальної об'єктної моделі документа або віртуальної DOM. React створює кеш-структуру даних в пам’яті, обчислює отримані різниці, а потім ефективно оновлює

відображуваний DOM браузера. Цей процес називається звіренням. Це дозволяє програмісту писати код так, ніби вся сторінка відображається при кожній зміні, тоді як бібліотеки React відображають лише підкомпоненти, які насправді змінюються. Цей вибірковий візуалізація забезпечує значне підвищення продуктивності. Це економить зусилля з перерахунку стилю CSS, макета сторінки та рендеринга для всієї сторінки.

Методи життєвого циклу використовують форму підключення, що дозволяє виконувати код у задані точки протягом життя компонента.

1. shouldComponentUpdate дозволяє розробнику запобігти непотрібному повторному рендерингу компонента, повертаючи значення false, якщо візуалізація не потрібна.

2. componentDidMount викликається, як тільки компонент "змонтований" (компонент був створений в інтерфейсі користувача, часто шляхом асоціювання його з DOM-вузлом). Це зазвичай використовується для запуску завантаження даних з віддаленого джерела через API.

3. componentWillUnmount викликається безпосередньо перед тим, як компонент буде зруйнований або "демонтований". Це зазвичай використовується для очищення залежних від ресурсу компонентних компонентів, які не будуть просто видалені при демонтуванні компонента (наприклад, видалення будь-яких екземплярів setInterval (), які пов'язані з компонентом, або "eventListener", встановленого на " документ "через наявність компонента)

4. візуалізація є найважливішим методом життєвого циклу і єдиним необхідним у будь-якому компоненті. Зазвичай він викликається кожного разу, коли стан компонента оновлюється, що повинно відображатися в інтерфейсі користувача.

JSX, або JavaScript XML, є розширенням синтаксису мови JavaScript. Подібно до зовнішнього вигляду HTML, JSX забезпечує спосіб структурування візуалізації компонентів за допомогою синтаксису, знайомого багатьом розробникам. Компоненти React, як правило, пишуться

за допомогою JSX, хоча їх не обов’язково (компоненти також можуть бути написані на чистому JavaScript). JSX подібний до іншого синтаксису розширення, створеного Facebook для PHP під назвою XHP.

Хуки - це функції, які дозволяють розробникам "підключатись" до стану реагування та функцій життєвого циклу з функціональних компонентів. Вони роблять код читабельним і легко зрозумілим. Гачки не працюють всередині класів - вони дозволяють використовувати React без занять.

React надає кілька вбудованих гачків, таких як useState, useContext, useReducer та useEffect, щоб назвати декілька. Усі вони вказані в Довідковому матеріалі API Hooks. useState та useEffect, які найчастіше використовуються, призначені для управління станами та побічними ефектами відповідно у React Components.

Для підтримки концепції React про односпрямований потік даних (який може бути протиставлений двонаправленому потоку AngularJS), архітектура Flux представляє альтернативу популярній архітектурі модель-view-контролер. Flux має дії, які надсилаються через центральний диспетчер до магазину, а зміни в магазині передаються назад до подання. При використанні з React це розповсюдження здійснюється завдяки властивостям компонентів.

Потік можна вважати варіантом моделі спостерігача. Компонент React під архітектурою Flux не повинен безпосередньо змінювати будь-який переданий йому реквізит, але повинен передавати функції зворотного виклику, які створюють дії, які відправляються диспетчером для модифікації сховища. Дія - це об'єкт, відповідальність якого полягає в описі того, що відбулося: наприклад, дія, що описує одного користувача, "що слідує" за іншим, може містити ідентифікатор користувача, цільовий ідентифікатор користувача та тип USER_FOLLOWED_ANOTHER_USER. Магазини, які можна сприймати як моделі, можуть змінити себе у відповідь на дії, отримані від диспетчера.

Ця закономірність іноді виражається як "властивості стікають вниз, дії течуть вгору". З моменту його створення було створено багато реалізацій Flux, мабуть, найвідомішим є Redux, який має єдиний магазин, який часто називають єдиним джерелом істини.

4.3. Генерація файлу із типами та методами для отримання даних

Після створення клієнтської та серверної частини а також налашлування конфігураційних параметрів для кодогенератора є можливість перевірити його роботу.

Після запуску команди graphql-codegen у папці src/generated з’явився файл types.ts із наступним змістом.

```
/* eslint-disable */

export type Maybe<T> = T | null;
/** All built-in and custom scalars, mapped to their actual values */
export type Scalars = {
  ID: string;
  String: string;
  Boolean: boolean;
  Int: number;
  Float: number;
};

export type Book = {
  id: Scalars["ID"];
  name: Scalars["String"];
  releaseDate: Scalars["String"];
};

export type Character = {
  id: Scalars["ID"];
  name: Scalars["String"];
  culture?: Maybe<Scalars["String"]>;
  titles?: Maybe<Array<Scalars["String"]>>;
  aliases?: Maybe<Array<Scalars["String"]>>;
  born?: Maybe<Scalars["String"]>;
  died?: Maybe<Scalars["String"]>;
  father?: Maybe<Character>;
  mother?: Maybe<Character>;
  spouse?: Maybe<Character>;
  children?: Maybe<Array<Character>>;
  allegiances?: Maybe<Array<House>>;
  appearedIn: Array<TvSeason>;
  isAlive: Scalars["Boolean"];
  playedBy?: Maybe<Scalars["String"]>;
  books?: Maybe<Array<Book>>;
};

export type House = {
  id: Scalars["ID"];
  name: Scalars["String"];
```

```
      titles?: Maybe<Array<Scalars["String"]>>;
      members: Array<Character>;
      slogan?: Maybe<Scalars["String"]>;
      overlord?: Maybe<Character>;
      currentLord?: Maybe<Character>;
      founder?: Maybe<Character>;
      ancestralWeapons?: Maybe<Array<Scalars["String"]>>;
      coatOfArms?: Maybe<Scalars["String"]>;
      seats?: Maybe<Array<Scalars["String"]>>;
    };

    export type Query = {
      getCharacters: Array<Character>;
      getCharacter?: Maybe<Character>;
      getHouses: Array<House>;
      getHouse?: Maybe<House>;
    };

    export type QueryGetCharactersArgs = {
      sortDirection?: Maybe<SortDirection>;
    };

    export type QueryGetCharacterArgs = {
      characterId: Scalars["ID"];
    };

    export type QueryGetHousesArgs = {
      sortDirection?: Maybe<SortDirection>;
    };

    export type QueryGetHouseArgs = {
      houseId: Scalars["ID"];
    };

    export enum SortDirection {
      Asc = "ASC",
      Desc = "DESC"
    }

    export type TvSeason = {
      id: Scalars["ID"];
      startDate: Scalars["String"];
      endDate: Scalars["String"];
      name: Scalars["String"];
      characters: Array<Character>;
    };
    export type CharacterDetailQueryVariables = {
      id: Scalars["ID"];
    };

    export type CharacterDetailQuery = { __typename?: "Query" } & {
      getCharacter: Maybe<
        { __typename?: "Character" } & CharacterDetailFieldsFragment
      >;
    };

    export type CharacterDetailFieldsFragment = { __typename?: "Character" }
& Pick<
      Character,
      | "name"
      | "playedBy"
      | "culture"
      | "titles"
      | "aliases"
```

```
      | "born"
      | "died"
      | "isAlive"
    > & {
        allegiances: Maybe<Array<{ __typename?: "House" } & Pick<House,
"name">>>;
        father: Maybe<
          { __typename?: "Character" } & Pick<Character, "id" | "name">
        >;
        mother: Maybe<
          { __typename?: "Character" } & Pick<Character, "id" | "name">
        >;
        spouse: Maybe<
          { __typename?: "Character" } & Pick<Character, "id" | "name">
        >;
        children: Maybe<
          Array<{ __typename?: "Character" } & Pick<Character, "id" |
"name">>
        >;
        appearedIn: Array<{ __typename?: "TvSeason" } & Pick<TvSeason,
"name">>;
        books: Maybe<Array<{ __typename?: "Book" } & Pick<Book, "id" |
"name">>>;
      };

    export type CharacterListQueryVariables = {};

    export type CharacterListQuery = { __typename?: "Query" } & {
      getCharacters: Array<{ __typename?: "Character" } &
CharacterInfoFragment>;
    };

    export type CharacterInfoFragment = { __typename?: "Character" } & Pick<
      Character,
      "id" | "name" | "playedBy" | "culture" | "isAlive"
    > & {
        allegiances: Maybe<Array<{ __typename?: "House" } & Pick<House,
"name">>>;
      };


    export const CharacterListComponent = (
      props: Omit<
        Omit<
          ReactApollo.QueryProps<CharacterListQuery,
CharacterListQueryVariables>,
          "query"
    import {
      CreateFilmLocationMutation,
      CreateFilmLocationMutationVariables,
      CreateFantasyMutation,
      CreateFantasyMutationVariables,
      CreatePointMutation,
      CreatePointMutationVariables,
      CreditCardQuery,
      CreditCardQueryVariables,
      DestroyFilmMutation,
      DestroyFilmMutationVariables,
      LocationPartsFragmentDoc,
      DeliveryAccountsQuery,
      DeliveryAccountsQueryVariables,
      ActorAccountsQuery,
      ActorAccountsQueryVariables,
      FilmDataDocument,
```

```
        FilmDataQuery,
        FilmDataQueryVariables,
        FilmDetailsQuery,
        FilmDetailsQueryVariables,
        FilmEditFieldsQuery,
        FilmEditFieldsQueryVariables,
        FilmOverviewQuery,
        FilmOverviewQueryVariables,
        FilmQueryQuery,
        FilmQueryQueryVariables,
        FilmSelectQuery,
        FilmSelectQueryVariables,
        FilmsLocationsQuery,
        FilmsLocationsQueryVariables,
        FilmsQuery,
        FilmsQueryVariables,
        FilmStatusLogsQuery,
        FilmStatusLogsQueryVariables,
        FilmsPointListQuery,
        FilmsPointListQueryVariables,
        FilmPointsQuery,
        FilmPointsQueryVariables,
        PaymentsQuery,
        PaymentsQueryVariables,
        FantasysQuery,
        FantasysQueryVariables,
        UndeleteFilmMutation,
        UndeleteFilmMutationVariables,
        UpdateFilmMutation,
        UpdateFilmMutationVariables,
        UpdateFantasyMutation,
        UpdateFantasyMutationVariables,
        PointCloseActorsFragmentDoc,
        PointCommentPartsFragmentDoc,
        PointDetailsQuery,
        PointDetailsQueryVariables,
        PointDistancePartsFragmentDoc,
        PointEventPartsFragmentDoc,
        PointPartsFragmentDoc,
        PointQuery,
        PointQueryVariables,
        PointRecurrencePartsFragmentDoc,
        PointRouteQuery, PointRouteQueryVariables
    } from "./types";

    /**
     * __useFilmDataQuery__
     *
     * To run a query within a React component, call `useFilmDataQuery` and
pass it any options that fit your needs.
     * When your component renders, `useFilmDataQuery` returns an object from
Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useFilmDataQuery({
     *   variables: {
     *      id: // value for 'id'
     *   },
     * });
```

```
 */
export function useFilmDataQuery(baseOptions?:
Apollo.QueryHookOptions<FilmDataQuery, FilmDataQueryVariables>) {
  return Apollo.useQuery<FilmDataQuery,
FilmDataQueryVariables>(FilmDataDocument, baseOptions);
}
export function useFilmDataLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<FilmDataQuery, FilmDataQueryVariables>) {
  return Apollo.useLazyQuery<FilmDataQuery,
FilmDataQueryVariables>(FilmDataDocument, baseOptions);
}
export type FilmDataQueryHookResult = ReturnType<typeof
useFilmDataQuery>;
export type FilmDataLazyQueryHookResult = ReturnType<typeof
useFilmDataLazyQuery>;
export type FilmDataQueryResult = Apollo.QueryResult<FilmDataQuery,
FilmDataQueryVariables>;
export const CreateFilmLocationDocument = gql`
    mutation createFilmLocation($input: LocationInput!) {
  createLocation(input: $input) {
    data {
      id
    }
  }
}
    `;
export type CreateFilmLocationMutationFn =
Apollo.MutationFunction<CreateFilmLocationMutation,
CreateFilmLocationMutationVariables>;

/**
 * __useCreateFilmLocationMutation__
 *
 * To run a mutation, you first call `useCreateFilmLocationMutation`
within a React component and pass it any options that fit your needs.
 * When your component renders, `useCreateFilmLocationMutation` returns a
tuple that includes:
 * - A mutate function that you can call at any time to execute the
mutation
 * - An object with fields that represent the current status of the
mutation's execution
 *
 * @param baseOptions options that will be passed into the mutation,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options-2;
 *
 * @example
 * const [createFilmLocationMutation, { data, loading, error }] =
useCreateFilmLocationMutation({
 *   variables: {
 *      input: // value for 'input'
 *   },
 * });
 */
export function useCreateFilmLocationMutation(baseOptions?:
Apollo.MutationHookOptions<CreateFilmLocationMutation,
CreateFilmLocationMutationVariables>) {
  return Apollo.useMutation<CreateFilmLocationMutation,
CreateFilmLocationMutationVariables>(CreateFilmLocationDocument,
baseOptions);
}
export type CreateFilmLocationMutationHookResult = ReturnType<typeof
useCreateFilmLocationMutation>;
```

```
    export type CreateFilmLocationMutationResult =
Apollo.MutationResult<CreateFilmLocationMutation>;
    export type CreateFilmLocationMutationOptions =
Apollo.BaseMutationOptions<CreateFilmLocationMutation,
CreateFilmLocationMutationVariables>;
    export const DeliveryAccountsDocument = gql`
        query DeliveryAccounts($filter: [AccountFilter]) {
      accounts(filter: $filter) {
        data {
          id
          fullName
          avatar {
            data {
              thumbUrl
            }
          }
        }
      }
    }
        `;

    /**
     * __useDeliveryAccountsQuery__
     *
     * To run a query within a React component, call
`useDeliveryAccountsQuery` and pass it any options that fit your needs.
     * When your component renders, `useDeliveryAccountsQuery` returns an
object from Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useDeliveryAccountsQuery({
     *   variables: {
     *      filter: // value for 'filter'
     *   },
     * });
     */
    export function useDeliveryAccountsQuery(baseOptions?:
Apollo.QueryHookOptions<DeliveryAccountsQuery,
DeliveryAccountsQueryVariables>) {
      return Apollo.useQuery<DeliveryAccountsQuery,
DeliveryAccountsQueryVariables>(DeliveryAccountsDocument, baseOptions);
    }
    export function useDeliveryAccountsLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<DeliveryAccountsQuery,
DeliveryAccountsQueryVariables>) {
      return Apollo.useLazyQuery<DeliveryAccountsQuery,
DeliveryAccountsQueryVariables>(DeliveryAccountsDocument, baseOptions);
    }
    export type DeliveryAccountsQueryHookResult = ReturnType<typeof
useDeliveryAccountsQuery>;
    export type DeliveryAccountsLazyQueryHookResult = ReturnType<typeof
useDeliveryAccountsLazyQuery>;
    export type DeliveryAccountsQueryResult =
Apollo.QueryResult<DeliveryAccountsQuery, DeliveryAccountsQueryVariables>;
    export const FilmSelectDocument = gql`
        query FilmSelect($filter: [FilmFilter]) {
      Films(filter: $filter) {
        data {
          id
```

```
          fullName
          birthDate
          avatar {
            data {
              thumbUrl
            }
          }
        }
      }
    }
        `;

    /**
     * __useFilmSelectQuery__
     *
     * To run a query within a React component, call `useFilmSelectQuery` and
pass it any options that fit your needs.
     * When your component renders, `useFilmSelectQuery` returns an object
from Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useFilmSelectQuery({
     *   variables: {
     *      filter: // value for 'filter'
     *   },
     * });
     */
    export function useFilmSelectQuery(baseOptions?:
Apollo.QueryHookOptions<FilmSelectQuery, FilmSelectQueryVariables>) {
      return Apollo.useQuery<FilmSelectQuery,
FilmSelectQueryVariables>(FilmSelectDocument, baseOptions);
    }
    export function useFilmSelectLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<FilmSelectQuery, FilmSelectQueryVariables>) {
      return Apollo.useLazyQuery<FilmSelectQuery,
FilmSelectQueryVariables>(FilmSelectDocument, baseOptions);
    }
    export type FilmSelectQueryHookResult = ReturnType<typeof
useFilmSelectQuery>;
    export type FilmSelectLazyQueryHookResult = ReturnType<typeof
useFilmSelectLazyQuery>;
    export type FilmSelectQueryResult = Apollo.QueryResult<FilmSelectQuery,
FilmSelectQueryVariables>;
    export const FilmsDocument = gql`
        query Films($filter: [FilmFilter], $pagination: PaginationInput,
$sorting: FilmSorting) {
      Films(filter: $filter, pagination: $pagination, sorting: $sorting) {
        data {
          id
          fullName
          phoneNumber
          relationship
          birthDate
          gender
          insertedAt
          status
          account {
            data {
              id
```

```
              fullName
            }
          }
        }
        pagination {
          afterCursor
          beforeCursor
          limit
          totalCount
        }
      }
    }
        `;

    /**
     * __useFilmsQuery__
     *
     * To run a query within a React component, call `useFilmsQuery` and pass
it any options that fit your needs.
     * When your component renders, `useFilmsQuery` returns an object from
Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useFilmsQuery({
     *   variables: {
     *      filter: // value for 'filter'
     *      pagination: // value for 'pagination'
     *      sorting: // value for 'sorting'
     *   },
     * });
     */
    export function useFilmsQuery(baseOptions?:
Apollo.QueryHookOptions<FilmsQuery, FilmsQueryVariables>) {
      return Apollo.useQuery<FilmsQuery, FilmsQueryVariables>(FilmsDocument,
baseOptions);
    }
    export function useFilmsLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<FilmsQuery, FilmsQueryVariables>) {
      return Apollo.useLazyQuery<FilmsQuery,
FilmsQueryVariables>(FilmsDocument, baseOptions);
    }
    export type FilmsQueryHookResult = ReturnType<typeof useFilmsQuery>;
    export type FilmsLazyQueryHookResult = ReturnType<typeof
useFilmsLazyQuery>;
    export type FilmsQueryResult = Apollo.QueryResult<FilmsQuery,
FilmsQueryVariables>;
    export const FilmOverviewDocument = gql`
        query FilmOverview($id: ID!) {
      Film(id: $id) {
        data {
          id
          fullName
          avatar {
            data {
              thumbUrl
            }
          }
          phoneNumber
          language
```

```
          gender
          email
          birthDate
          account {
            data {
              id
              fullName
            }
          }
          relationship
          status
        }
      }
    }
        `;

    /**
     * __useFilmOverviewQuery__
     *
     * To run a query within a React component, call `useFilmOverviewQuery`
and pass it any options that fit your needs.
     * When your component renders, `useFilmOverviewQuery` returns an object
from Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useFilmOverviewQuery({
     *   variables: {
     *      id: // value for 'id'
     *   },
     * });
     */
    export function useFilmOverviewQuery(baseOptions?:
Apollo.QueryHookOptions<FilmOverviewQuery, FilmOverviewQueryVariables>) {
      return Apollo.useQuery<FilmOverviewQuery,
FilmOverviewQueryVariables>(FilmOverviewDocument, baseOptions);
    }
    export function useFilmOverviewLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<FilmOverviewQuery, FilmOverviewQueryVariables>) {
      return Apollo.useLazyQuery<FilmOverviewQuery,
FilmOverviewQueryVariables>(FilmOverviewDocument, baseOptions);
    }
    export type FilmOverviewQueryHookResult = ReturnType<typeof
useFilmOverviewQuery>;
    export type FilmOverviewLazyQueryHookResult = ReturnType<typeof
useFilmOverviewLazyQuery>;
    export type FilmOverviewQueryResult =
Apollo.QueryResult<FilmOverviewQuery, FilmOverviewQueryVariables>;
    export const FilmDetailsDocument = gql`
        query FilmDetails($id: ID!) {
      Film(id: $id) {
        data {
          id
          description
          preferredContactMethod
          stillDrives
          warnings {
            data
          }
          status
```

```
          requirements {
            data
          }
          insertedAt
          updatedAt
          enrolledBy {
            data {
              id
              fullName
            }
          }
          genderPreference
          languagePreference
          vehiclePreference
          favoriteActors {
            data {
              id
              fullName
            }
          }
          blockedActors {
            data {
              id
              fullName
            }
          }
          biographyQuestions {
            question
            answer
          }
        }
      }
    }
        `;

    /**
     * __useFilmDetailsQuery__
     *
     * To run a query within a React component, call `useFilmDetailsQuery`
and pass it any options that fit your needs.
     * When your component renders, `useFilmDetailsQuery` returns an object
from Apollo Client that contains loading, error, and data properties
     * you can use to render your UI.
     *
     * @param baseOptions options that will be passed into the query,
supported options are listed on:
https://www.apollographql.com/docs/react/api/react-hooks/#options;
     *
     * @example
     * const { data, loading, error } = useFilmDetailsQuery({
     *   variables: {
     *      id: // value for 'id'
     *   },
     * });
     */
    export function useFilmDetailsQuery(baseOptions?:
Apollo.QueryHookOptions<FilmDetailsQuery, FilmDetailsQueryVariables>) {
      return Apollo.useQuery<FilmDetailsQuery,
FilmDetailsQueryVariables>(FilmDetailsDocument, baseOptions);
    }
    export function useFilmDetailsLazyQuery(baseOptions?:
Apollo.LazyQueryHookOptions<FilmDetailsQuery, FilmDetailsQueryVariables>) {
      return Apollo.useLazyQuery<FilmDetailsQuery,
FilmDetailsQueryVariables>(FilmDetailsDocument, baseOptions);
```

```
    }
    export type FilmDetailsQueryHookResult = ReturnType<typeof
useFilmDetailsQuery>;
    export type FilmDetailsLazyQueryHookResult = ReturnType<typeof
useFilmDetailsLazyQuery>;
    export type FilmDetailsQueryResult = Apollo.QueryResult<FilmDetailsQuery,
FilmDetailsQueryVariables>;
    export const FilmPointsDocument = gql`
        query FilmPoints($filter: [PointFilter], $pagination:
```

Що свідчить про правильне налаштування конфігурацій кодогенератора, а також, що він корректно працію, т.к в цьому файлі є описання усіх типів, що були створенні на стороні сервера, а також методи, для отримання даних із БД.

ВИСНОВОК ДО РОЗДІЛУ 4

У даному розділі була створена сервера частина, з використанням GraphQL та ApolloServer, де були створені описи типів моделей, що використовуються у БД а також їх схеми.

У клієнтській частині був створений інтерфейс, з використанням React і Apollo-Client. Для прикладу була обрана тематика акторів. Кожен актор має назву ролі, її значення у фільмі, список інших акторів, що мають до нього відношення і.т.д.

На клієнтській і серверній частині були створені конфігураційні файли, які є необхідними для створенної системи кодогенерації.

Після запуску CLI команди, доступ до якої надає система, був створений файл, який містів у собі систему типів, яка була використана при створені GraphQL схем, а також методи, за допомогою яких можна звертатися до серверу, для отримання даних.

ВИСНОВКИ

В результаті даної роботи був проведений аналіз та детальний розбір технологій, за допомогою яких є можливість створювати сгенеровані файли, а саме:

1. JavaScript – мова програмування, яка використовується для реалізації усієї логіки кодогенератора.

2. TypeScript – мова програмування, яка була розроблена компанією Microsoft для створення веб-додатків. Ця мова компілюється в JavaScript на етапі розробці додатку, тому швидкодія додатку ніяк не зміниться. Ця мова надає значні переваги розробнику, дає те, що є у інших типізованих мовах, але немає в JavaScript, а саме: можливість використовувати типи (string, number, Boolean, object, Array, динамічні типи, а також можливість створювати власні типи та інтерфейси), можливість декларувати модулі, інтерфейси, класи та інші.

GraphQL має ряд особливостей проектування:

1. Ієрархічність: Більшість розробок продуктів сьогодні включає створення та маніпулювання ієрархіями поглядів.

2. Орієнтований на продукт: GraphQL не виправдовується вимогами переглядів та інтерфейсних інженерів, які їх пишуть.

3. Надійне введення: Кожен сервер GraphQL визначає специфічну для типу програми систему. Запити виконуються в контексті системи цього типу.

Був створений серверний та клієнтський додаток с використанням мови програмування Typescript, мови запитів GraphQL, а також бібліотек React, Apoll-Server, Apollo-client для поліпшення коммунікація меж клієнтом та сервером і покращенням зв’язку між БД та схемами GraphQL.

Після налаштування конфігураційного файлу у серверному та клієнтському додатках, був отриманий доступ до CLI кодогенератора. Після

запущення команди graphql-generate був створений файл, із автогенерованими типами та методами для звернення до API.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. Документація JavaScript,Typescript [Електронний ресурс]. – 2020. – режим доступу: http://www.w3.org/ (04.06.2020)
2. Документація веб стандартів [Електронний ресурс]. – 2020. – режим доступу: http://web-standards.ru/ (04.06.2020)
3. Документація по JavaScript [Електронний ресурс]. – 2020. – режим доступу: https://javascript.com/ (дата звернення 04.06.2020)
4. Документація по JavaScript (стандарт ES6) [Електронний ресурс]. – 2020. – режим доступу: https://learn.javascript.com/. (дата звернення 04.06.2020)
5. Документація по fast-ethernet [Електронний ресурс]. – 2014. – режим доступу: http://www.ixbt.com/comm/tech-fast-ethernet.html (дата звернення 04.06.2020)

ДОДАТОК А

```
import { DetailedError, Types, isComplexPluginOutput, federationSpec } from
'@graphql-codegen/plugin-helpers';
import { visit, parse, DefinitionNode, Kind, print, NameNode } from 'graphql';
import { executePlugin } from './execute-plugin';
import {
  checkValidationErrors,
  validateGraphQlDocuments,
  printSchemaWithDirectives,
  Source,
} from '@graphql-tools/utils';

import { mergeSchemas } from '@graphql-tools/merge';

export async function codegen(options: Types.GenerateOptions): Promise<string>
{
  const documents = options.documents || [];

  if (documents.length > 0 && !options.skipDocumentsValidation) {
    validateDuplicateDocuments(documents);
  }

  const pluginPackages = Object.keys(options.pluginMap).map(key =>
options.pluginMap[key]);

  if (!options.schemaAst) {
    options.schemaAst = mergeSchemas({
      schemas: [],
```

```
    typeDefs: [options.schema],
```



```
   convertExtensions: true,
    assumeValid: true,
    assumeValidSDL: true,
    ...options.config,
   });
  }

  // merged schema with parts added by plugins
  let schemaChanged = false;
  let schemaAst = pluginPackages.reduce((schemaAst, plugin) => {
   const addToSchema =
    typeof plugin.addToSchema === 'function' ?
plugin.addToSchema(options.config) : plugin.addToSchema;

   if (!addToSchema) {
    return schemaAst;
   }

   return mergeSchemas({
    schemas: [schemaAst],
    typeDefs: [addToSchema],
   });
  }, options.schemaAst);

  const federationInConfig = pickFlag('federation', options.config);
  const isFederation = prioritize(federationInConfig, false);
```

```
 if (
```



```
  isFederation &&
    !schemaAst.getDirective('external') &&
    !schemaAst.getDirective('requires') &&
    !schemaAst.getDirective('provides') &&
    !schemaAst.getDirective('key')
  ) {
    schemaChanged = true;
    schemaAst = mergeSchemas({
      schemas: [schemaAst],
      typeDefs: [federationSpec],
      convertExtensions: true,
      assumeValid: true,
      assumeValidSDL: true,
    });
  }

  if (schemaChanged) {
    options.schema = parse(printSchemaWithDirectives(schemaAst));
  }

  const skipDocumentValidation =
    typeof options.config === 'object' && !Array.isArray(options.config) &&
options.config.skipDocumentsValidation;

  if (options.schemaAst && documents.length > 0 && !skipDocumentValidation)
{
    const extraFragments: { importFrom: string; node: DefinitionNode }[] =
```

```
 options.config && (options.config as any).externalFragments ? (options.config as
any).externalFragments : [];
   const errors = await validateGraphQlDocuments(options.schemaAst, [
     ...documents,
     ...extraFragments.map(f => ({
       location: f.importFrom,
       document: { kind: Kind.DOCUMENT, definitions: [f.node] },
     })),
   ]);
   checkValidationErrors(errors);
 }

 const prepend: Set<string> = new Set<string>();
 const append: Set<string> = new Set<string>();

 const output = await Promise.all(
   options.plugins.map(async plugin => {
     const name = Object.keys(plugin)[0];
     const pluginPackage = options.pluginMap[name];
     const pluginConfig = plugin[name] || {};

     const execConfig =
       typeof pluginConfig !== 'object'
         ? pluginConfig
         : {
             ...options.config,
             ...pluginConfig,
```

```
      };
```



```
const result = await executePlugin(
    {
      name,
      config: execConfig,
      parentConfig: options.config,
      schema: options.schema,
      schemaAst,
      documents: options.documents,
      outputFilename: options.filename,
      allPlugins: options.plugins,
      skipDocumentsValidation: options.skipDocumentsValidation,
      pluginContext: options.pluginContext,
    },
    pluginPackage
  );

  if (typeof result === 'string') {
    return result || '';
  } else if (isComplexPluginOutput(result)) {
    if (result.append && result.append.length > 0) {
      for (const item of result.append) {
        if (item) {
          append.add(item);
        }
      }
    }
```

```
      if (result.prepend && result.prepend.length > 0) {
```



```
    for (const item of result.prepend) {
          if (item) {
            prepend.add(item);
          }
        }
      }
      return result.content || '';
    }

    return '';
  })
 );

 return [...sortPrependValues(Array.from(prepend.values())), ...output,
...Array.from(append.values())]
   .filter(Boolean)
   .join('\n');
}

function resolveCompareValue(a: string) {
 if (a.startsWith('/*') || a.startsWith('//') || a.startsWith(' *') || a.startsWith(' */') ||
a.startsWith('*/')) {
   return 0;
 } else if (a.startsWith('package')) {
   return 1;
 } else if (a.startsWith('import')) {
   return 2;
```

```
  } else {
```



```
    return 3;
  }
}

export function sortPrependValues(values: string[]): string[] {
  return values.sort((a: string, b: string) => {
    const aV = resolveCompareValue(a);
    const bV = resolveCompareValue(b);

    if (aV < bV) {
      return -1;
    }
    if (aV > bV) {
      return 1;
    }

    return 0;
  });
}

function validateDuplicateDocuments(files: Types.DocumentFile[]) {
  // duplicated names
  const definitionMap: {
    [kind: string]: {
      [name: string]: {
        paths: Set<string>;
        contents: Set<string>;
```

```
    };
```



```
  };
} = {};

function addDefinition(
  file: Source,
  node: DefinitionNode & { name?: NameNode },
  deduplicatedDefinitions: Set<DefinitionNode>
) {
  if (typeof node.name !== 'undefined') {
    if (!definitionMap[node.kind]) {
      definitionMap[node.kind] = {};
    }
    if (!definitionMap[node.kind][node.name.value]) {
      definitionMap[node.kind][node.name.value] = {
        paths: new Set(),
        contents: new Set(),
      };
    }

    const definitionKindMap = definitionMap[node.kind];

    const length = definitionKindMap[node.name.value].contents.size;
    definitionKindMap[node.name.value].paths.add(file.location);
    definitionKindMap[node.name.value].contents.add(print(node));
    if (length === definitionKindMap[node.name.value].contents.size) {
      return null;
    }
```

```
    }
```



```
    return deduplicatedDefinitions.add(node);
  }

  files.forEach(file => {
    const deduplicatedDefinitions = new Set<DefinitionNode>();
    visit(file.document, {
      OperationDefinition(node) {
        addDefinition(file, node, deduplicatedDefinitions);
      },
      FragmentDefinition(node) {
        addDefinition(file, node, deduplicatedDefinitions);
      },
    });
    (file.document as any).definitions = Array.from(deduplicatedDefinitions);
  });

  const kinds = Object.keys(definitionMap);

  kinds.forEach(kind => {
    const definitionKindMap = definitionMap[kind];
    const names = Object.keys(definitionKindMap);
    if (names.length) {
      const duplicated = names.filter(name =>
definitionKindMap[name].contents.size > 1);

      if (!duplicated.length) {
        return;
```

```
    }
```



```
    const list = duplicated
      .map(name =>
        `
      * ${name} found in:
        ${[...definitionKindMap[name].paths]
          .map(filepath => {
            return `
            - ${filepath}
          `.trimRight();
          })
          .join('')}
    `.trimRight()
      )
      .join('');

    const definitionKindName = kind.replace('Definition', '').toLowerCase();
    throw new DetailedError(
      `Not all ${definitionKindName}s have an unique name: ${duplicated.join(',
')}`,
      `
        Not all ${definitionKindName}s have an unique name
        ${list}
      `
    );
  }
 });
}
```

```
function isObjectMap(obj: any): obj is Types.PluginConfig<any> {
  return obj && typeof obj === 'object' && !Array.isArray(obj);
}

function prioritize<T>(...values: T[]): T {
  const picked = values.find(val => typeof val === 'boolean');

  if (typeof picked !== 'boolean') {
    return values[values.length - 1];
  }

  return picked;
}

function pickFlag(flag: string, config: Types.PluginConfig): boolean | undefined {
  return isObjectMap(config) ? (config as any)[flag] : undefined;
}
import { DetailedError, Types, CodegenPlugin } from '@graphql-codegen/plugin-
helpers';
import { DocumentNode, GraphQLSchema, buildASTSchema } from 'graphql';

export interface ExecutePluginOptions {
  name: string;
  config: Types.PluginConfig;
  parentConfig: Types.PluginConfig;
  schema: DocumentNode;
  schemaAst?: GraphQLSchema;
```

```
  documents: Types.DocumentFile[];
```



```
 outputFilename: string;
  allPlugins: Types.ConfiguredPlugin[];
  skipDocumentsValidation?: boolean;
  pluginContext?: { [key: string]: any };
}

export async function executePlugin(options: ExecutePluginOptions, plugin:
CodegenPlugin): Promise<Types.PluginOutput> {
  if (!plugin || !plugin.plugin || typeof plugin.plugin !== 'function') {
    throw new DetailedError(
      `Invalid Custom Plugin "${options.name}"`,
      `
        Plugin ${options.name} does not export a valid JS object with "plugin"
function.

        Make sure your custom plugin is written in the following form:

        module.exports = {
          plugin: (schema, documents, config) => {
            return 'my-custom-plugin-content';
          },
        };
        `
    );
  }
```

```
 const outputSchema: GraphQLSchema = options.schemaAst ||
buildASTSchema(options.schema, options.config as any);
```

Продовження додатку А

```
 const documents = options.documents || [];
 const pluginContext = options.pluginContext || {};

 if (plugin.validate && typeof plugin.validate === 'function') {
  try {
   // FIXME: Sync validate signature with plugin signature
   await plugin.validate(
    outputSchema,
    documents,
    options.config,
    options.outputFilename,
    options.allPlugins,
    pluginContext
   );
  } catch (e) {
   throw new DetailedError(
    `Plugin "${options.name}" validation failed:`,
    `
      ${e.message}
    `
   );
  }
 }

 return Promise.resolve(
  plugin.plugin(
```

```
    outputSchema,
    documents,
```



```
  typeof options.config === 'object' ? { ...options.config } : options.config,
    {
      outputFile: options.outputFilename,
      allPlugins: options.allPlugins,
      pluginContext,
    }
  )
 );
}

import { loadConfig, GraphQLExtensionDeclaration, GraphQLConfig } from
'graphql-config';
import { ApolloEngineLoader } from '@graphql-tools/apollo-engine-loader';
import { CodeFileLoader } from '@graphql-tools/code-file-loader';
import { GitLoader } from '@graphql-tools/git-loader';
import { GithubLoader } from '@graphql-tools/github-loader';
import { PrismaLoader } from '@graphql-tools/prisma-loader';

export const CodegenExtension: GraphQLExtensionDeclaration = (api: any) => {
 // Schema
 api.loaders.schema.register(new CodeFileLoader());
 api.loaders.schema.register(new GitLoader());
 api.loaders.schema.register(new GithubLoader());
 api.loaders.schema.register(new ApolloEngineLoader());
 api.loaders.schema.register(new PrismaLoader());
 // Documents
```

```
  api.loaders.documents.register(new CodeFileLoader());
  api.loaders.documents.register(new GitLoader());
```



```
  api.loaders.documents.register(new GithubLoader());

  return {
    name: 'codegen',
  };
};

export async function findAndLoadGraphQLConfig(filepath?: string):
Promise<GraphQLConfig | void> {
  const config = await loadConfig({
    filepath,
    rootDir: process.cwd(),
    extensions: [CodegenExtension],
    throwOnEmpty: false,
    throwOnMissing: false,
  });

  if (isGraphQLConfig(config)) {
    return config;
  }
}

// Kamil: user might load a config that is not GraphQL Config
//        so we need to check if it's a regular config or not
function isGraphQLConfig(config: GraphQLConfig): config is GraphQLConfig {
  if (!config) {
```

```
  return false;
 }
```



```
 try {
  return config.getDefault().hasExtension('codegen');
 } catch (e) {}

 try {
  for (const projectName in config.projects) {
   if (config.projects.hasOwnProperty(projectName)) {
    const project = config.projects[projectName];

    if (project.hasExtension('codegen')) {
     return true;
    }
   }
  }
 } catch (e) {}

 return false;
}
import { defineCommand } from '@graphql-cli/common';
import {
 CodegenExtension,
 CodegenContext,
 generate,
 updateContextWithCliFlags,
 buildOptions,
 YamlCliFlags,
```

```
} from '@graphql-codegen/cli';
```

Продовження додатку А

```
export default defineCommand<{}, YamlCliFlags>(api => {
  return {
    command: 'codegen',
    builder(yargs) {
      return yargs.options(buildOptions() as any) as any;
    },
    async handler(args) {
      const config = await api.useConfig({
        rootDir: args.config || process.cwd(),
        extensions: [CodegenExtension],
      });

      // Create Codegen Context with our loaded GraphQL Config
      const codegenContext = new CodegenContext({
        graphqlConfig: config,
      });
      // This will update Codegen Context with the options provided in CLI
arguments
      updateContextWithCliFlags(codegenContext, args);

      await generate(codegenContext);
    },
  };
});
import { DetailedError, Types, CodegenPlugin } from '@graphql-codegen/plugin-
helpers';
```

```
import { DocumentNode, GraphQLSchema, buildASTSchema } from 'graphql';
```



```
export interface ExecutePluginOptions {
  name: string;
  config: Types.PluginConfig;
  parentConfig: Types.PluginConfig;
  schema: DocumentNode;
  schemaAst?: GraphQLSchema;
  documents: Types.DocumentFile[];
  outputFilename: string;
  allPlugins: Types.ConfiguredPlugin[];
  skipDocumentsValidation?: boolean;
  pluginContext?: { [key: string]: any };
}

export async function executePlugin(options: ExecutePluginOptions, plugin:
CodegenPlugin): Promise<Types.PluginOutput> {
  if (!plugin || !plugin.plugin || typeof plugin.plugin !== 'function') {
    throw new DetailedError(
      `Invalid Custom Plugin "${options.name}"`,
      `
        Plugin ${options.name} does not export a valid JS object with "plugin"
function.

        Make sure your custom plugin is written in the following form:

        module.exports = {
          plugin: (schema, documents, config) => {
            return 'my-custom-plugin-content';
```

```
        },
      };
```



```
      `
  );
 }

 const outputSchema: GraphQLSchema = options.schemaAst ||
buildASTSchema(options.schema, options.config as any);
 const documents = options.documents || [];
 const pluginContext = options.pluginContext || {};

 if (plugin.validate && typeof plugin.validate === 'function') {
  try {
   // FIXME: Sync validate signature with plugin signature
   await plugin.validate(
    outputSchema,
    documents,
    options.config,
    options.outputFilename,
    options.allPlugins,
    pluginContext
   );
  } catch (e) {
   throw new DetailedError(
    `Plugin "${options.name}" validation failed:`,
    `
      ${e.message}
    `
   );
```

```
  }
 }
```



```
 return Promise.resolve(
  plugin.plugin(
   outputSchema,
   documents,
   typeof options.config === 'object' ? { ...options.config } : options.config,
   {
    outputFile: options.outputFilename,
    allPlugins: options.allPlugins,
    pluginContext,
   }
  )
 );
}
import { isBrowser, isNode } from './is-browser';

export function cliError(err: any, exitOnError = true) {
 let msg: string;

 if (err instanceof Error) {
  msg = err.message || err.toString();
 } else if (typeof err === 'string') {
  msg = err;
 } else {
  msg = JSON.stringify(err);
 }
```

```
  // eslint-disable-next-line no-console
  console.error(msg);
```



```
  if (exitOnError && isNode) {
    process.exit(1);
  } else if (exitOnError && isBrowser) {
    throw err;
  }
}
import { getLogger } from './logger';

let queue: Array<{
  message: string;
  meta?: any[];
}> = [];

export function debugLog(message: string, ...meta: any[]) {
  if (!process.env.GQL_CODEGEN_NODEBUG && process.env.DEBUG !==
undefined) {
    queue.push({
      message,
      meta,
    });
  }
}

export function printLogs() {
  if (!process.env.GQL_CODEGEN_NODEBUG && process.env.DEBUG !==
undefined) {
```

```
    queue.forEach(log => {
      getLogger().info(log.message, ...log.meta);
```



```
 });
    resetLogs();
  }
}

export function resetLogs() {
  queue = [];
}
import { writeFileSync, statSync, readFileSync, unlink } from 'fs';

export function writeSync(filepath: string, content: string) {
  return writeFileSync(filepath, content);
}

export function readSync(filepath: string) {
  return readFileSync(filepath, 'utf-8');
}

export function fileExists(filePath: string): boolean {
  try {
    return statSync(filePath).isFile();
  } catch (err) {
    return false;
  }
}

export function unlinkFile(filePath: string, cb?: (err?: Error) => any): void {
```

```
  unlink(filePath, cb);
}
```

Продовження додатку А

```
declare let window: any, process: any;

const isBrowser = typeof window !== 'undefined' && typeof window.document
!== 'undefined';

const isNode = typeof process !== 'undefined' && process.versions != null &&
process.versions.node != null;

export { isBrowser, isNode };
import chalk from 'chalk';
import indentString from 'indent-string';
import logSymbol from 'log-symbols';
import ansiEscapes from 'ansi-escapes';
import wrapAnsi from 'wrap-ansi';
import { stripIndent } from 'common-tags';
import { ListrTask } from 'listr';
import { DetailedError, isDetailedError } from '@graphql-codegen/plugin-helpers';
import { Source } from 'graphql';
import { debugLog, printLogs } from './debugging';

const UpdateRenderer = require('listr-update-renderer');

export class Renderer {
  private updateRenderer: any;

  constructor(tasks: ListrTask, options: any) {
```

```
    this.updateRenderer = new UpdateRenderer(tasks, options);
  }
```

Продовження додатку А

```
render() {
    return this.updateRenderer.render();
  }

  end(
    err: Error & {
      errors?: (Error | DetailedError)[];
      details?: string;
    }
  ) {
    this.updateRenderer.end(err);

    if (typeof err === 'undefined') {
      logUpdate.clear();
      return;
    }

    // persist the output
    logUpdate.done();

    // show errors
    if (err) {
      const errorCount = err.errors ? err.errors.length : 0;

      if (errorCount > 0) {
```

```
      const count = indentString(chalk.red.bold(`Found ${errorCount}
error${errorCount > 1 ? 's' : ''}`), 1);
      const details = err.errors
        .map(error => {
```

Продовження додатку А

```
    debugLog(`[CLI] Exited with an error`, error);

        return { msg: isDetailedError(error) ? error.details : null, rawError: error };
      })
      .map(({ msg, rawError }, i) => {
        const source: string | Source | undefined = (err.errors[i] as any).source;

        msg = msg ? chalk.gray(indentString(stripIndent(`${msg}`), 4)) : null;
        const stack = rawError.stack ?
chalk.gray(indentString(stripIndent(rawError.stack), 4)) : null;

        if (source) {
          const sourceOfError = typeof source === 'string' ? source : source.name;
          const title = indentString(`${logSymbol.error} ${sourceOfError}`, 2);

          return [title, msg, stack, stack].filter(Boolean).join('\n');
        }

        return [msg, stack].filter(Boolean).join('\n');
      })
      .join('\n\n');
    logUpdate.emit(['', count, details, ''].join('\n\n'));
  } else {
    const details = err.details ? err.details : '';
```

```
      logUpdate.emit(`${chalk.red.bold(`${indentString(err.message,
2)}`)}\n${details}\n${chalk.grey(err.stack)}`);
```

Продовження додатку А

```
    }
   }

   logUpdate.done();

   printLogs();
  }
}

const render = tasks => {
  for (const task of tasks) {
   task.subscribe(
    event => {
     if (event.type === 'SUBTASKS') {
      render(task.subtasks);
      return;
     }

     if (event.type === 'DATA') {
      logUpdate.emit(chalk.dim(`${event.data}`));
     }
     logUpdate.done();
    },
    err => {
     logUpdate.emit(err);
     logUpdate.done();
```

```
      }
    );
```



```
 }
};

export class ErrorRenderer {
  private tasks: any;

  constructor(tasks, _options) {
    this.tasks = tasks;
  }

  render() {
    render(this.tasks);
  }

  static get nonTTY() {
    return true;
  }

  end() {}
}

class LogUpdate {
  private stream = process.stdout;
  // state
  private previousLineCount = 0;
  private previousOutput = '';
```

```
private previousWidth = this.getWidth();
```



```
emit(...args: string[]) {
  let output = args.join(' ') + '\n';
  const width = this.getWidth();

  if (output === this.previousOutput && this.previousWidth === width) {
    return;
  }

  this.previousOutput = output;
  this.previousWidth = width;

  output = wrapAnsi(output, width, {
    trim: false,
    hard: true,
    wordWrap: false,
  });

  this.stream.write(ansiEscapes.eraseLines(this.previousLineCount) + output);
  this.previousLineCount = output.split('\n').length;
}

clear() {
  this.stream.write(ansiEscapes.eraseLines(this.previousLineCount));
  this.previousOutput = '';
  this.previousWidth = this.getWidth();
  this.previousLineCount = 0;
```

```
  }
```



```
  done() {
    this.previousOutput = '';
    this.previousWidth = this.getWidth();
    this.previousLineCount = 0;
  }

  private getWidth() {
    const { columns } = this.stream;

    if (!columns) {
      return 80;
    }

    return columns;
  }
}

const logUpdate = new LogUpdate();
import { executeCodegen } from '../codegen';
import { Types, normalizeInstanceOrArray, normalizeOutputParam } from
'@graphql-codegen/plugin-helpers';

import isGlob from 'is-glob';
import debounce from 'debounce';
import logSymbols from 'log-symbols';
import { debugLog } from './debugging';
```

```
import { getLogger } from './logger';
import { join } from 'path';
```



```
import { FSWatcher } from 'chokidar';
import { lifecycleHooks } from '../hooks';
import { loadContext, CodegenContext } from '../config';
import { isValidPath } from '@graphql-tools/utils';

function log(msg: string) {
  // double spaces to inline the message with Listr
  getLogger().info(`  ${msg}`);
}

function emitWatching() {
  log(`${logSymbols.info} Watching for changes...`);
}

export const createWatcher = (
  initalContext: CodegenContext,
  onNext: (result: Types.FileOutput[]) => Promise<Types.FileOutput[]>
): Promise<void> => {
  debugLog(`[Watcher] Starting watcher...`);
  let config: Types.Config & { configFilePath?: string } =
initalContext.getConfig();
  const files: string[] = [initalContext.filepath].filter(a => a);
  const documents =
normalizeInstanceOrArray<Types.OperationDocument>(config.documents);
  const schemas = normalizeInstanceOrArray<Types.Schema>(config.schema);
```

```
  // Add schemas and documents from "generates"
  Object.keys(config.generates)
```



```
   .map(filename => normalizeOutputParam(config.generates[filename]))
    .forEach(conf => {
      schemas.push(...normalizeInstanceOrArray<Types.Schema>(conf.schema));

documents.push(...normalizeInstanceOrArray<Types.OperationDocument>(conf.d
ocuments));
    });

  if (documents) {
    documents.forEach(doc => {
      if (typeof doc === 'string') {
        files.push(doc);
      } else {
        files.push(...Object.keys(doc));
      }
    });
  }

  schemas.forEach((schema: string) => {
    if (isGlob(schema) || isValidPath(schema)) {
      files.push(schema);
    }
  });

  if (typeof config.watch !== 'boolean') {
    files.push(...normalizeInstanceOrArray<string>(config.watch));
```

```
}
```



```
let watcher: FSWatcher;

const runWatcher = async () => {
  const chokidar = await import('chokidar');
  let isShutdown = false;

  const debouncedExec = debounce(() => {
    if (!isShutdown) {
      executeCodegen(initalContext)
        .then(onNext, () => Promise.resolve())
        .then(() => emitWatching());
    }
  }, 100);
  emitWatching();

  const ignored: string[] = [];
  Object.keys(config.generates)
    .map(filename => ({ filename, config:
normalizeOutputParam(config.generates[filename]) }))
    .forEach(entry => {
      if (entry.config.preset) {
        const extension = entry.config.presetConfig &&
entry.config.presetConfig.extension;
        if (extension) {
          ignored.push(join(entry.filename, '**', '*' + extension));
        }
```

```
    } else {
      ignored.push(entry.filename);
```



```
 }
 });

watcher = chokidar.watch(files, {
  persistent: true,
  ignoreInitial: true,
  followSymlinks: true,
  cwd: process.cwd(),
  disableGlobbing: false,
  usePolling: config.watchConfig?.usePolling,
  interval: config.watchConfig?.interval,
  depth: 99,
  awaitWriteFinish: true,
  ignorePermissionErrors: false,
  atomic: true,
  ignored,
});

debugLog(`[Watcher] Started`);

const shutdown = () => {
  isShutdown = true;
  debugLog(`[Watcher] Shutting down`);
  log(`Shutting down watch...`);
  watcher.close();
  lifecycleHooks(config.hooks).beforeDone();
```

```
};
```



```
// it doesn't matter what has changed, need to run whole process anyway
  watcher.on('all', async (eventName, path) => {
    lifecycleHooks(config.hooks).onWatchTriggered(eventName, path);
    debugLog(`[Watcher] triggered due to a file ${eventName} event: ${path}`);
    const fullPath = join(process.cwd(), path);
    delete require.cache[fullPath];

    if (eventName === 'change' && config.configFilePath && fullPath ===
config.configFilePath) {
      log(`${logSymbols.info} Config file has changed, reloading...`);
      const context = await loadContext(config.configFilePath);

      const newParsedConfig: Types.Config & { configFilePath?: string } =
context.getConfig();
      newParsedConfig.watch = config.watch;
      newParsedConfig.silent = config.silent;
      newParsedConfig.overwrite = config.overwrite;
      newParsedConfig.configFilePath = config.configFilePath;
      config = newParsedConfig;
    }

    debouncedExec();
  });

  process.once('SIGINT', shutdown);
  process.once('SIGTERM', shutdown);
```

```
};
```



```
  // the promise never resolves to keep process running
  return new Promise<void>((resolve, reject) => {
    executeCodegen(initalContext)
      .then(onNext, () => Promise.resolve())
      .then(runWatcher)
      .catch(err => {
        watcher.close();
        reject(err);
      });
  });
};
```