КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

КАФЕДРА ТЕОРІЇ ТА ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ

Кваліфікаційна робота

за спеціальністю 122 Комп'ютерні науки

на тему:

Валідація обміну даними при клієнт-серверній взаємодії

Виконав студент 2 курсу магістратури

Попович Андрій Дмитрович

(підпис)

Науковий керівник:
доцент
Кузенко В.Ф.

_____

(підпис)

Засвідчую, що в цій кваліфікаційній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування.
«27» квітня 2021 р., протокол No 10
Завідувач кафедри

М. С. Нікітченко _____

(підпис)

Київ – 2021

Реферат

Обсяг роботи: 40 сторінок, 13 ілюстрацій, 6 джерел посилань.

ВАЛІДАЦІЯ, CLIENT-SIDE, SERVER-SIDE, SSOT, DRY, KISS, BDUF, MIDDLEWARE , RXJS, JSON SCHEMA

Об’єктом дослідження є сучасні підходи до реалізації структур валідації даних та їх обміну на клієнтських та серверних частинах.

Метою роботи було зробити аналіз основних підходів до валідації даних на клієнтській та серверній частинах, виділити основні недоліки кожного з підходів, виділити основні ситуації, при яких доцільно використовувати певні підходи та оптимізувати їх, базуючись на компромісах між DRY та KISS.

Методами та інструментами дослідження були вивчення та узагальнення відомостей про сучасні підходи до валідації даних, порівняння даних методів і дедукції(розгляд підходів в загальних випадках та більш уніфікованих) у сукупності з аналізом передових технологій від розробників зі світовим ім'ям і дискусій щодо оптимальних підходів з більш досвідченими розробниками.

Результатами роботи є реалізація відповідної структури валідацій даних з використанням ssot та перевірка доцільності використання даного підходу у певних ситуаціях.

Зміст

Вступ

Оцінка сучасного стану об’єкта дослідження.

На даних момент всі основні підходи до валідацій у певній мірі ігнорують принцип DRY, що у ситуаціях швидкого розвитку системи ускладнює процес супроводу і підтримки якості системи. Що у свою чергу погіршує можливість синхронізування валідацій, або навмисне зменшення якості валідації на певній стороні.

Мета й завдання роботи.

Метою роботи було зробити аналіз основних підходів до валідації даних на клієнтській та серверній частинах, виділити основні недоліки кожного з підходів, виділити основні ситуації, при яких доцільно використовувати певні підходи та оптимізувати їх, базуючись на компромісах між DRY та KISS.

Об’єкт і методи дослідження

Об’єктом дослідження є сучасні підходи до реалізації структур валідації даних та їх обміну на клієнтських та серверних частинах

Методами та інструментами дослідження були вивчення та узагальнення відомостей про сучасні підходи до валідації даних, порівняння даних методів і дедукції(розгляд підходів в загальних випадках та більш уніфікованих) у сукупності з аналізом передових технологій від розробників зі світовим ім'ям і дискусій щодо оптимальних підходів з більш досвідченими розробниками.

РОЗДІЛ 1. Клієнт-серверна архітектура[1]

1.1 Клієнт-серверна архітектура

Архітектура клієнт-сервер є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережевих застосунків і передбачає взаємодію та обмін даними між ними. Вона передбачає такі основні компоненти:

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами.

1.1.1 Обов'язки та взаємодія

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером. Логічно можна відокремити три рівні операцій:

- рівень представлення даних, який по суті являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
- прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;
- рівень управління даними, який забезпечує зберігання даних та доступ до них.

1.1.2 Трирівнева архітектура

Трирівнева клієнт-серверна архітектура, яка почала розвиватися з середини 90-х років, передбачає відділення прикладного рівня від управління даними. Відокремлюється окремий програмний рівень, на якому зосереджується прикладна логіка застосунку. Програми проміжного рівня можуть функціонувати під управлінням спеціальних серверів застосунків, але запуск

таких програм може здійснюватися і під управлінням звичайного веб-сервера. А саме управління даними здійснюється сервером даних.

Для роботи з системою користувач використовує стандартне програмне забезпечення — звичайний браузер. Це позбавляє його необхідності завантажувати та інсталювати спеціальні програми, якщо сам застосунок не є відповідно мобільним, або десктопним. Але користувачеві слід надати в розпорядження інтерфейс, який дозволяв би йому взаємодіяти з системою і формувати запити до неї. Форми, що визначають цей інтерфейс, розміщуються на веб-сторінках та завантажуються разом з ними.

Користувач формує свій запит та пересилає його до сервера, який здійснює обробку. При необхідності сервер викликає серверні програмні модулі, які забезпечують обробку запиту і в разі потреби звертаються до сервера даних. Сервер даних здійснює операції з даними, що зберігаються в системі та складають її інформаційну основу. Зокрема, він може здійснити вибірку з інформаційної бази відповідно до запиту та передати її модулю проміжного рівня для подальшої обробки. Дані, з якими працює сервер даних, найчастіше організовані як реляційна база даних.

Найчастіше веб-сервер і серверні модулі проміжного рівня розміщуються на одному комп'ютері, хоч і являють собою окремі і логічно незалежні програмні модулі.

РОЗДІЛ 2. Валідація даних[2]

В інформатиці валідації даних - це процес забезпечення того, що дані, котрі пройшли відповідні перевірки, є правильними та корисними. Він використовує процедури, які часто називають "правилами перевірки", “перевірками обмежень" або "процедурами перевірки", які перевіряють правильність, значимість та безпеку даних, що вводяться в систему. Правила можуть бути реалізовані за допомогою автоматизованих засобів словника даних або шляхом включення явної логіки перевірки прикладних програм комп'ютера.

2.1 Огляд

Перевірка даних призначена для надання певних чітко визначених гарантій відповідності та узгодженості даних у програмі або автоматизованій системі. Правила перевірки даних можуть бути визначені та розробленііз використанням різних методологій, і можуть бути розгорнуті в різних контекстах. Для їх реалізації можна використовувати декларативні правила цілісності даних або ділові правила на основі процедур.

Валідація даних не обов’язково гарантують їх коректність, і помилки введення даних, такі як орфографічні помилки, можуть бути прийняті як валідні.

2.1.1 Типи валідацій

Валіація дозволених символів

Перевіряє, щоб переконатися, що в полі присутні лише очікувані символи. Наприклад, числове поле може містити лише цифри 0–9, десяткову крапку і, можливо, знак мінуса або коми. Текстове поле, наприклад особисте ім'я, може заборонити символи, які використовуються для розмітки. Для адреси електронної пошти може знадобитися принайні один знак @ та різні інші

структурні деталі. Регулярні вирази можуть бути ефективними способами здійснення таких перевірок.

Валідація пропущених значень

Дані відправлені окремими полями мають відповідати деякому сумарному значенню.

Валідація кардинальності

Перевіряє, що запис містить дійсну кількість пов’язаних записів. Наприклад, якщо запис контакту класифікується як "клієнт", він повинен мати принаймні одне пов'язане з ним замовлення (кардинальність> 0). Цей тип правила може ускладнюватися додатковими умовами. Наприклад, якщо запис працівника у базі даних заробітної плати класифікується як "колишній працівник", то він не повинен мати жодних пов'язаних виплат зарплати після дати розірвання контракту/звільнення (кардинальність = 0).

Валідація узгодженості

Перевіряє поля, щоб переконатися, що дані в полях узгоджені, наприклад, якщо “термін дії” минув, то статус має бути “не активний”.

Міжсистемна валідація узгодженості

Порівнює дані в різних системах, щоб забезпечити їх узгодженість. Системи можуть представляти одні й ті самі дані по-різному, у цьому випадку порівняння вимагає трансформації (наприклад, одна система може зберігати ім'я клієнта в одному полі Name як 'Doe, John Q', тоді як інша використовує firstName ‘John’, lastName ‘Doe’ і middleName ‘Quality’).

Валідація типів даних

Перевіряє відповідність вводу допустимим даним. Наприклад, поле введення, яке приймає лише числові дані, має відхилити !number.

Валідація наявностей файлів

Перевіряє наявність файлу із зазначеним іменем. Ця перевірка є важливою для програм, які використовують обробку файлів.

Валідація формату даних

Перевіряє, чи дані мають вказаний формат (шаблон), наприклад, дати мають бути у форматі YYYY-MM-DD. Для цього виду перевірки можуть використовуватися регулярні вирази.

Валідація “наявності”

Перевіряє наявність даних, наприклад, від клієнтів може знадобитися існуюча електронна адреса.

Валідація діапазону

Перевіряє, чи дані перебувають у визначеному діапазоні значень, наприклад, ймовірність повинна бути від 0 до 1.

Довідкова цілісність

Значення в двох таблицях реляційних баз даних можуть бути пов'язані за допомогою зовнішнього ключа та первинного ключа. Якщо значення в полі зовнішнього ключа не обмежуються внутрішніми механізмами, тоді їх слід перевірити, щоб переконатися, що посилальна таблиця завжди посилається на рядок у вказаній таблиці.

Валідація унікальності даних

Перевіряє, що кожне значення є унікальним. Це доцільно може бути застосовано до таких полів, як наприклад ID, CompanyName тощо.

2.1.2 Види валідацій

Оцінюючи основні типи валідації даних, можна зробити узагальнення щодо різних видів перевірки відповідно до їх обсягу, складності та мети.

- Валідація типу даних;
- Валідація діапазонів та обмежень;
- Валідація коду та перехресних посилань;
- Структурована валідація
- Валідація узгодженості

Валідація типів даних

Найпростіша валідація типів даних підтверджує, що окремі символи, введені користувачем, відповідають очікуваним символам одного або декількох відомих примітивних типів даних, як визначено мовою програмування або механізмом зберігання та пошуку даних.

Наприклад, integer поле може вимагати щоб дані, були представлені лише символами від 0 до 9 та деякими спецсимволами.

Валідація діапазонів та обмежень

Валідація діапазону та обмежень може перевіряти вхідні дані на узгодженість з мінімальним / максимальним діапазоном або узгодженість з тестом для оцінки послідовності символів, наприклад тестів за рахунок регулярних виразів(regex). Або наприклад, деяке значення має бути цілим додатнім числом, дата прибуття повинна бути пізніше відправлення, а пароль має відповідати мінімальній довжині та містити спецсимволи декількох категорій.

Валідація коду та перехресних посилань

Перевірка коду та перехресних посилань включає операції з перевірки відповідності даних одному або декільком можливо зовнішнім правилам, вимогам чи колекціям, що стосуються певної організації, контексту чи набору основних припущень. Ці додаткові обмеження можуть включати в себе перехресне посилання на надані дані з відомою пошуковою таблицею або інформаційною службою каталогів, такою як LDAP.

Наприклад, наданий користувачем код країни може знадобитися для ідентифікації поточного геополітичного регіону.

LDAP - (англ. Lightweight Directory Access Protocol — Полегшений протокол доступу до директорій / каталогів) — мережевий протокол прикладного рівня для надсилання запитів та модифікації даних служби каталогів через TCP/IP. LDAP є відкритим, комерційно-нейтральним, (англ. vendor-neutral), промисловим стандартним протоколом. LDAP розроблений IETF як полегшений варіант розробленого ITU-T протоколу DAP.

Структурована валідація

Структурована валідація дозволяє поєднувати інші види перевірки, разом із більш складною обробкою. Така складна обробка може включати перевірку умовних обмежень для цілого складного об'єкта даних або набору операцій процесу в системі.

Валідація узгодженості

Перевірка узгодженості забезпечує логічність даних. Наприклад, даті вильоту літака може бути заборонено передувати даті його прильоту до його наступного пункту призначення.

2.1.3 Post-validation actions

Дія “примусового виконання”

Дія “примусового виконання”, як правило, відхиляє запит на введення даних і вимагає від вхідного суб'єкта внести зміни, які приведуть дані у відповідність правилам валідації. Він також добре працює у ситуаціях обміну файлами, де введення файлу може бути відхилено, а набір повідомлень надіслано назад до користувача, чому файл було відхилено.

Інша форма примусового виконання передбачає автоматичну зміну даних та збереження відповідної версії замість початкової версії. Це найбільш підходить для косметичних змін. Недоречним використанням даних методів може бути у ситуаціях, коли вони призводить до втрати ділової інформації. Наприклад, збереження скороченого коментаря, якщо довжина введених даних перевищує очікувану. Зазвичай це не є good practice, оскільки це може призвести до втрати значних даних.

Консультативні дії

Консультативні дії, як правило, дозволяють вводити дані без змін, але надсилають повідомлення вихідному актору із зазначенням тих проблем перевірки, які мали місце. Це найбільш підходить для неінтерактивної системи, для систем, де зміна не є критичною для бізнесу, для етапів очищення існуючих даних та для етапів перевірки процесу введення.

Дії верифікації

Дії верифікації - це особливі випадки консультативних дій. У цьому випадку стороні клієнту пропонується підтвердити, що ці дані є тим, що вони дійсно хотіли б ввести, при припущенні про протилежне. Тут крок перевірки пропонує альтернативний варіант (наприклад, перевірка поштової адреси повертає інший спосіб форматування цієї адреси або взагалі пропонує іншу адресу). У цьому випадку користувачеві надається можливість прийняти рекомендацію або зберегти початкову версію.

Логування валідацій

Навіть у випадках, коли валідація даних не виявила жодних проблем, може бути важливим ведення журналу перевірок, які були проведені, та їх результатів. Це корисно для виявлення будь-яких відсутніх перевірок даних та для покращення вже існуючої валідації

2.1.4 Валідація та безпека

Помилки або упущення в процесі валідації даних можуть призвести до пошкодження даних або вразливості безпеки. Валідація даних перевіряє, чи дані придатні для своєї мети, дійсні та безпечні перед їх обробкою.

2.2 Стандартні та нестандартні підходи до валідації

Найбільш поширеними варіантами валідації обміну даними є підходи з написанням валідаційних схем відокремлено на клієнтській частині та на сервері, або ж виключно на сервері, нехтуючи валідацією на клієнті. Варіант з валідацією лише на клієнті з досить очевидних причин не використовується, бо вона легко обходиться, якщо це може знадобитися. Це називається bypassing. Таким чином на сервер можуть потрапити невалідні дані.

Основною проблемою першого варіанту є те, що більша частина інформації дублюється в двох місцях і при великих обсягах коду при розробці виникає проблема пов'язана із синхронізацією валідацій. Також даний підхід нехтує одним з основних принципів розробки DRY.

Проблемою другого варіанту є те, що користувач може дізнатися про те, що введені ним дані є некоректними лише після того, як сервер обробить відправлений запит і відповідно поверне помилки.

Також є варіант з відправкою валідаційних запитів на сервер при кожному оновленні клієнтом даних. І цей варіант би був найкращим з усіх, бо вся валідація б зберігалася і використовувалася лише в одному місці, і була гарантовано актуальною. А найголовнішим мінусом, який повністю виключає сенс використання даного підходу - це потенційно надмірні навантаження на сервер, котрі в даний час є невиправданими на користь інших підходів.

Після детального аналізу проблем можна прийти до досить очевидних рішеннь, а саме наявність ssot (single source of truth) в якому зберігатимуться загальні аспекти для клієнтської і серверної валідації, це позбавляє нас проблеми контролю за актуальністю двох окремих схем, як при першому згаданому варіанті.

2.2.1 Принципи розробки та пошук компромісу між DRY та KISS[4]

Хорошому програмісту необхідно вміти поєднувати свої навички зі здоровим глуздом. Вся справа в прагматизмі і вмінні вибору кращого рішення для вирішення певної проблеми. Коли ви стикаєтеся з проблемою при розробці ПЗ, ви можете скористатися базовими принципами, які допоможуть у виборі найбільш правильного підходу.

Так-як наявність ssot вирішує певні проблеми, це в той же самий час це вимагає наявності певного функціоналу для коректного функціонування системи. Клієнт і сервер в найчастіше пишуться на різних мовах, тому основним способом зберігання загальних валідацій можуть бути описи валідаційних схем у форматі json. Що в свою чергу зобов'язує на наявність парсерів і різних методів для комбінування загальних схем з унікальними лише для певної сторони.

Намагаючись дотримуватися адекватного балансу при нехтуванні певною мірою різними принципами, можна домогтися потенційного зменшення кількості багів при розвитку системи, позбутися від більшості проблем підтримки коду в подібних ситуаціях і інших схожих аспектів.

YAGNI

You Are not Gonna Need It / Вам це не знадобиться

Цей принцип простий і очевидний, але йому далеко не всі слідують. Якщо ви пишете код, то будьте впевнені, що він вам знадобиться. Немає сенсу писати код, якщо ви лише думаєте, що він можливо стане в нагоді пізніше.

DRY

Do not Repeat Yourself / Не повторюйтеся

Ця концепція була вперше сформульована в книзі Енді Ханта і Дейва Томаса «Програміст-прагматик: шлях від підмайстра до майстра».

Ідея обертається навколо єдиного джерела правди (single source of truth - SSOT).

У проектуванні і теорії інформаційних систем єдине джерело істини (SSOT) - це практика структурування інформаційних моделей і схеми даних, яка має на увазі, що всі фрагменти даних обробляються (або редагуються) тільки в одному місці... SSOT надають достовірні, актуальні і придатні до використання дані.

Використання SSOT дозволить створити більш зрозумілу кодову базу.

Дублювання коду - марна трата часу і ресурсів. Вам доведеться підтримувати одну і ту ж логіку і тестувати код відразу в двох місцях, причому якщо ви зміните код в одному місці, його потрібно буде змінити і в іншому.

У більшості випадків дублювання коду відбувається через незнання системи. Перш ніж що-небудь писати, проявіть прагматизм: озирніться. Можливо цей функціонал вже десь реалізований. Можливо, ця бізнес-логіка існує в іншому місці. Повторне використання коду - завжди розумне рішення.

KISS

Keep It Simple, Stupid / Будь простіше

Цей принцип був розроблений ВМС США в 1960 році. Цей принцип говорить, що прості системи будуть працювати краще і надійніше.

Стосовно до розробки ПЗ він означає наступне - не вигадуйте до задачі більш складного рішення, ніж їй потрібно.

Іноді найрозумніше рішення виявляється і найпростішим. Написання продуктивного, ефективного та простого коду - це доцільно.

Одна з найпоширеніших помилок нашого часу - використання нових інструментів виключно через те, що вони нові. Розробників слід мотивувати використовувати новітні технології не тому, що вони нові, а тому що вони підходять для роботи.

Big Design Up Front

Глобальне проектування перш за все

Цей підхід до розробки програмного забезпечення дуже важливий, і його часто ігнорують. Перш ніж переходити до реалізації, треба переконатися, що все добре продумано.

Найчастіше продумування рішень рятувало нас від проблем при розробці... Внесення змін до специфікації займало годину або дві. Якби ми вносили ці зміни в код, на це йшли б тижні. Я навіть не можу висловити, наскільки сильно я вірю у важливість проектування перед реалізацією, хоча адепти екстремального програмування піддали цю практику анафемі. Я економив час і

робив свої продукти краще, використовуючи BDUF, і я пишаюся цим фактом, щоб там не говорили фанатики екстремального програмування. Вони просто помиляються, інакше сказати не можу.

- Джоел Спольскі

Іноді в усуненні потенційних недоліків і процесах розробки архітектури повинні бути задіяні й всі частини команди. Чим раніше всі все обговорять, тим якіснішим і очевиднішим буде процес розробки системи для всіх.

Дуже поширений контраргумент полягає в тому, що вартість вирішення проблем найчастіше нижче вартості часу планування. Чим з меншою кількістю помилок зіткнеться користувач, тим краще буде його досвід. У вас може не бути іншого шансу впоратися з проблемами “за меншими втратами”.

Я програмую. Я будую речі.

Бритва Оккама

Бритва Оккама - методологічний принцип, в стислому вигляді гласить: «Не слід множити сутності без необхідності» (або «Не слід залучати нові сутності без крайньої на те необхідності»).

Таким чином, грунтуючись на BDUF можна постаратися знайти компроміс між DRY та KISS при виборі підходів до валідації системи. Потенційно, якщо загального коду(валідації) планується мало - можна знехтувати DRY, бо складність реалізації взаємодії з ssot і відповідних парсерів буде вище, ніж можна з цього виграти, якщо ж коду очікується дуже багато і валідація буде досить складна, то дублювання всього в кінцевому підсумку з розвитком

проекту породжує хаос і ускладнює можливість якісного внесення будь-яких змін.

2.3 Валідація при наявності ssot

2.3.1 JSON schema

Передмова

Хотілось привернути більше уваги до переваг опису передаваємих JSON повідомлень схемою JSON Schema. Незважаючи на те, що «на вході» розробка REST API без будь-якої JSON-схеми завжди простіше і швидше, але з ростом системи, її відсутність так чи інакше призводить до подорожчання супроводу і підтримки системи. Також будь-яка попередня проробка структури повідомлень сприяє більш якісної організації обміну повідомленнями, без зайвого дублювання при обміні даними і загальними правилами їх обробки.

JavaScript Object Notation (JSON)

Мова розмітки JSON задає обмежений набір типів даних. Для пари { "ключ": значення} для «ключа» завжди використовують тип string, для «значення» застосовні типи: string, number, object (тип JSON), array, boolean (true або false) і null.

Синтаксис JSON є підмножиною синтаксису JavaScript, де:

- Дані записуються у вигляді пар { "ключ": значення }.
- Дані розділяються комами.
- У фігурних дужках записуються об'єкти.
- У квадратних дужках записуються масиви.
- Найменування «ключів» чутливо до регістру.

JSON Schema[3]

Оскільки схема json написана в форматі JSON, вона підтримує всі типи JSON плюс доповнення: тип integer, який є підтипом типу number. Сама схема є JSON-об'єктом і призначена для опису даних в форматі JSON.

І тепер найважливіше - правила, що використовуються в схемі для завдання обмежень і структурування JSON-повідомлень.

JSON Schema дозволяє:

- Обмежити тип даних для елементів документа JSON.
- Залежно від типу валідації даних, також можуть бути застосовні додаткові правила - «keywords», починаючи з кореня схеми документа і спускаючись до їх дочірнім елементам.

Деякі «keywords» є чисто описовими, як наприклад: «title», «description» та ін., Які просто описують призначення схеми. Інші призначені для ідентифікації документа: «$ schema». Це ключове слово використовується для вказівки бажаної версії схеми. Значення цього ключового слова має бути рядком, що представляє URI

«Keywords» для опису схеми

"$schema"
"$id"
"examples"
"comment"
"title"
"description"

Загальні «Validation keywords», незалежні від типу даних елемента

"enum"

"const"

"type"

Ключові слова, що поєднують перевірки для різних частин схеми

"allOf"

"oneOf"

"anyOf"

Keywords, залежні від того типу даних, з яким вони використовуються

"type": "string"

pattern

contentEncoding

contentMediaType

minLength

maxLength

"type": "number" или "type": "integer"

maximum

minimum

exclusiveMaximum

exclusiveMinimum

multipleOf

"type": "object"

minProperties
maxProperties
properties
required
dependencies
propertyNames
patternProperties
additionalProperties

- "type": "array"

items
uniqueItems
additionalItems
maxItems
minItems
contains

- "type": "boolean"

Ключові слова для накладення умов на перевірки

"if-then-else"
"not"

Ключові слова, що забезпечують посилання і зв'язність схем

"$ref" та "definitions"
"$id" та "$ref"

Замість висновку

Що ж застосування JSON Schema в результаті може дати?

1. Дозволить розробляти сервіси, опрацьовуючи формати і склад даних з «доробком» на майбутній розвиток системи.
2. Може спростити життя розробникам і поліпшити код з валідації JSON-повідомлень. Іншими словами, це спрощення підтримки і інтеграції ПО.
3. Застосувати перевірку документів в документо-орієнтованих, об'єктно-орієнтованих БД.
4. JSON Schema може бути застосована при реалізації DTO
5. JSON-Schema може допомогти заощадити час на тестуванні та документуванні API.
6. Гнучку валідацію при генерації JSON Schema в run-time зі значеннями в «enum», одержуваними на етапі виконання програми.
7. Спрощення підтримки зворотної сумісності API.

2.3.2 Server-Side валідація використовуючи Typescript, NodeJS, Middleware та SSOT придставленний JSON Schema

Однією з переваг використання стандартної схеми JSON є те, що ви можна використовувати одну і ту ж визначену схему з різними мовами. Крім того, з такими редакторами, як код WebStorm, ви можете отримати дуже хороший інструмент для автозаповнення вашого JSON

Ще одна перевага полягає в тому, що ми можемо автоматично генерувати з них typescript інтерфейси

2.3.2.1 Яким чином JSON Schema використовується для валідації

Проста відповідь полягає в компіляції схеми у функцію, яка виконує перевірку. Хороша річ полягає в тому, що у більшості випадків ви зможете знайти майже будь-яке рішення, яке ви можете придумати, на NPM. Отже, вам не потрібно застосовувати власне рішення. І поєднуючи дані перевірки з перевірками, котрі можуть реалізуватися виключно лише за допомогою сервера, можна гарантувати безпеку та надійність роботи з даними.

2.3.2.2 Які основні валідатори наразі існують і котрі доцільно вибрати?

AJV vs Express validator vs Joi vs Hapi vs Validator.js. Особисто я вважаю найкращим рішення AJV, оскільки він швидкий, легко налаштовується, надійний і використовує стандартну схему JSON. Його популярність є ще однією корисною причиною його використання, оскільки ви маєте більше шансів на вирішення проблем набагато швидше, якщо і коли вони виникають. AJV компілює визначену схему JSON у функцію і внутрішньо майже не використовує ітерації

2.3.2.3 Як працює AJV

Бібліотека бере схему JSON і компілює її у функцію javascript (за допомогою конструктора функцій), яка реалізує всі визначені обмеження зі схем. Також можна зробити кілька важливих перевірок у Mongoose Schema. Однак існують обмеження щодо того, наскільки ми можемо перевірити їх.

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://www.example.org/user.json",
  "title": "user schema",
  "type": "object",
  "required": ["username", "name", "email", "password"],
  "properties": {
    "username": {
      "$id": "#username",
      "type": "string",
      "minLength": 3,
      "uniqueItems": true,
      "maxLength": 15
    },
    "name": {
      "type": "string",
      "minLength": 3,
      "maxItems": 100
    },
    "email": {
      "$id": "#email",
      "type": "string",
      "format": "email",
      "maxLength": 100
    },
    "age": {
      "type": "number",
      "minimum": 18,
      "maxItems": 150
    },
    "password": {
      "type": "string",
      "minLength": 7
    }
  },
  "additionalProperties": false
}
```

2.3.2.4 Організація схем JSON та спрощення їх імпортування задля покращень autocomplete

Перший метод: зберігання їх у форматі json

Особисто я вважаю за краще створювати окремі папки для своїх схем JSON. За допомогою наведеної структури всі схеми можна легко імпортувати у файл "schemas.ts" та повторно експортувати звідти.

```
// modes/schemas/schemas.ts
import userSchema from './users-schema.json';
import coordinateSchema from './location-schema.json';

const allSchemas = {
userSchema,
coordinateSchema
}

const schemas  = [...Object.values(allSchemas)]

export default {
...allSchemas
};
```

Це дозволяє легко імпортувати будь-яку схему з будь-якої точки кодової бази з хорошим автозаповненням.

Другий метод: використовування ts файлів

Перевага цього перед першим полягає в тому, що ми не тільки отримуємо кращі інструменти а й миттєву перевірку.

Тут замість використання файлів json ми тепер використовуємо файли ts, які можуть забезпечити кращу миттєву перевірку наших схем, а також можуть бути налаштовані. Нам також потрібно визначити нашу власну схему JSON, щоб зробити необхідним "title". Це буде потрібно для створення назв інтерфейсів. Ще одна причина для цього - якщо знадобиться змінити тип схеми JSON, після впровадження нових drafts (скажімо, draft-08).

2.3.2.5 Створення middleware для валідації request body

Визначена нижче функція є функцією вищого порядку, яка приймає схему JSON і надалі повертає middleware, котре виконує перевірку. За допомогою Ajv можливо скомпілювати схему у функцію та перевірити, чи є вона дійсною. Також, як видно, виключається метод GET, оскільки дані не повинні надходити через нього.

```
export const validateRequestBodyBySchema = (JSONSchema: CustomJSONSchema) => (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const ajv = new Ajv({ allErrors: true });
  const validate = ajv.compile(JSONSchema);
  const isValid = validate(req.body);

  if (!isValid && req.method !== 'GET') {
    logger.error(validate.errors);
    return res.status(400).json(validate.errors);
  }
  next();      You, 22 days ago • Feature: Middlewware to validate request body
};
```

2.3.2.6 Використання валідаційного middleware в контролерах

В Express, ви зазвичай можете використовувати middleware двома поширеними способами.

З використанням “use ”методу вашого маршрутизатора або програми. Це заафектить кожен контролер нижче визначеної валідації. Завдяки цьому будь-які недійсні дані не зможуть потрапити до жодного з наших контролерів через тіло запиту.

В якості альтернативи middleware може передаватися окремим контролерам із відповідною схемою як параметром.

Тіло запиту буде перевірено перед усім іншим, і якщо щось є невалідним, клієнту мають надсилатися помилки. Це можна використовувати безпосередньо на стороні клієнта. А ще краще, ми можемо поділитися однією і тією ж валідацією на клієнтській стороні, як було обговорено в самому початку другого розділу, та виконати деяку обробку помилок у реальному часі із повідомленнями про помилки, наданими AJV. Також можна запровадити власні повідомлення про помилки.

2.3.3 Client-side валідація з використанням ssot(JSON Schema), Typescript, React, Redux

2.3.3.1 Приклади проектів в яких застосовувався даний підхід

Даних підхід на практиці довелося використовувати в двох проектах, що в свою чергу полегшило розробку і мінімізувала потенційну кількість багів.

Першим проектом був проект, що стосується глобальної релокації з трьома клієнтськими частинами і загальним сервером. І так як при роботі над різними клієнтськими частинами періодично чередувалися команди, то треба було гарантувати, що зміни на сервері зроблені однією командою будуть актуальні на всіх етапах розробки і для іншої команди, яка працює над іншою частиною системи, але взаємодіє з аналогічними даними через загальні ендпоінти.

Другим проектом є проект в банківській сфері і це той самий випадок, коли будь-яких неточностей повинно бути мінімальна кількість. А значить, що чим менше потенційних можливостей виникнення розбіжностей - тим якісніше і швидше розробляється функціонал.

2.3.3.2 Приклади реалізації

```
const converter = (schema: JSONSchema7, config?: transformer.Config): Yup.ObjectSchema<object> | undefined => {
    config && transformer.setConfiguration(config);
    const normalizedSchema = normalize(schema);

    return transformer.default(normalizedSchema);
};

export type Config = transformer.Config;
export default converter;
```

Converter - використовується для парсингу та трансформації json schema у схему, котру можна використовувати у відповідних компонентах задля валідації полів.

В аргументи він приймає json схему і конфігурацію. У transformer записується конфігурація і функція повертає transformer з записаною нормалізованою схемою

```
export const normalize = (schema: JSONSchema7): JSONSchema7 => {
    const normalizer = flow([
        removeEmptyObjects,
        transformRefs,
        applyPaths,
        applyIfTypes
    ]);

    return normalizer(schema);
};
```

Функція нормалізації json schema. Використовує lodash функцію flow і видаляє випадкові порожні об'єкти, трансформує референси тощо. І повертає нормалізовану схему, що отримується, як аргумент на вході

```
export const transformRefs = (schema: JSONSchema7): JSONSchema7 => {
    const replaceRefs = (result: JSONSchema7, value: any, key: string) => {
        const hasRef = get(value, "$ref");
        const replaced = hasRef
            ? getDefinitionItem(schema, get(value, "$ref"))
            : isPlainObject(value)
                ? replaceAllRefs(value)
                : value;
        result[key] = replaced;
    };
    const replaceAllRefs = (schema: JSONSchema7): JSONSchema7 =>
        transform(schema, replaceRefs);
    return isPlainObject(schema) ? replaceAllRefs(schema) : schema;
};

export const applyIfTypes = (schema: JSONSchema7): JSONSchema7 => {
    const addType = (schema: JSONSchema7): JSONSchema7 =>
        transform(schema, (result: JSONSchema7, value: any, key: string) => {
            if (key === "if" && !isEmpty(value)) {
                const properties = get(schema, "properties");
                const ifProperties = get(value, "properties");
                const ifSchema = ifProperties && getObjectHead(ifProperties);
                if (ifSchema) {
                    const [ifSchemaKey, ifSchemaValue] = ifSchema;
                    const type =
                        ifSchemaKey &&
                        !has(ifProperties, [ifSchemaKey, "type"]) &&
                        has(properties, ifSchemaKey) &&
                        get(properties, [ifSchemaKey, "type"]);
                    value = type
                        ? {
                            ...value,
                            properties: {
                                ...ifProperties,
                                [ifSchemaKey]: { ...ifSchemaValue, type }
                            }
                        }
                        : value;
                }
            }
            result[key] = isPlainObject(value) ? addType(value) : value;
        });
    return isPlainObject(schema) ? addType(schema) : schema;
};
```

Деякі з функцій, що використовуються для нормалізації. А саме функція для трансформації референсів, які є keyword "$ ref" в json schema. І обробка if значень, які можуть бути записані в properties.

```
export const useSchema = () => {
    const { schema, isLoading } = useSelector < (({ validation }) => validation.userValidation);

    const [onPageLoadGetSchema] = useSchemaLoadHandler(schemaLoadAction);

    return [schema, isLoading, onPageLoadGetSchema];
};
```

Основний React hook, що повертає Json Schema з Redux сховища та функцію на завантаження самої схеми з сервера. Даний хук використовується в react компонентах і дозволяє отримати схему, отриману від сервера і безпосередньо екшен для завантаження даної схеми. Відповідно при завантаженні сторінки може бути викликаний екшен на завантаження схеми і після її отримання вона вже може бути передана в converter.

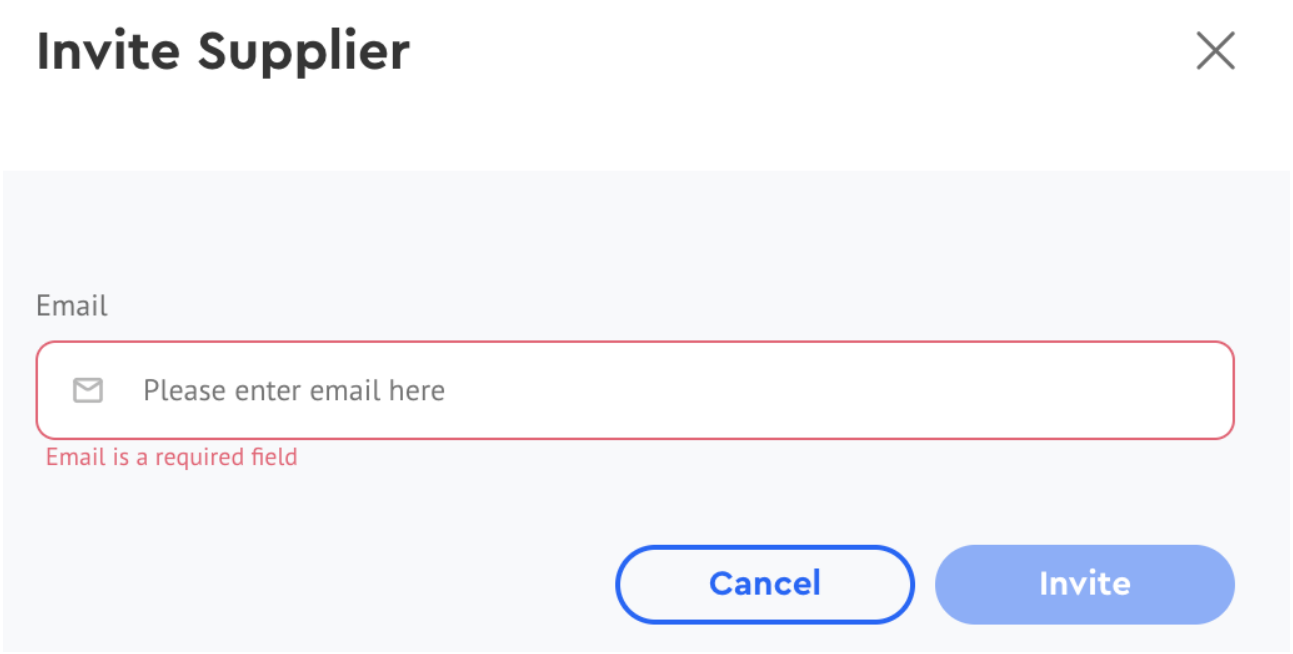
```
const [schema, isLoading, onPageLoadGetSchema] = useSchema();

useEffect( effect: () => onPageLoadGetSchema(),  deps: []);

const convertedJsonSchema = converter(schema);

const partnerUserValidationSchema = object()
    .concat(convertedJsonSchema)
    .shape( fields: {
        phoneNumber: string().label( label: 'Mobile').phoneNumber().required(),
        role: string().label( label: 'Role').oneOf(Object.values(RoleEnum)).required()
    });
```

Виклик функції на отримання схеми при першому рендері сторінки, конвертування схеми та приєднання її до схеми з іншими перевірками

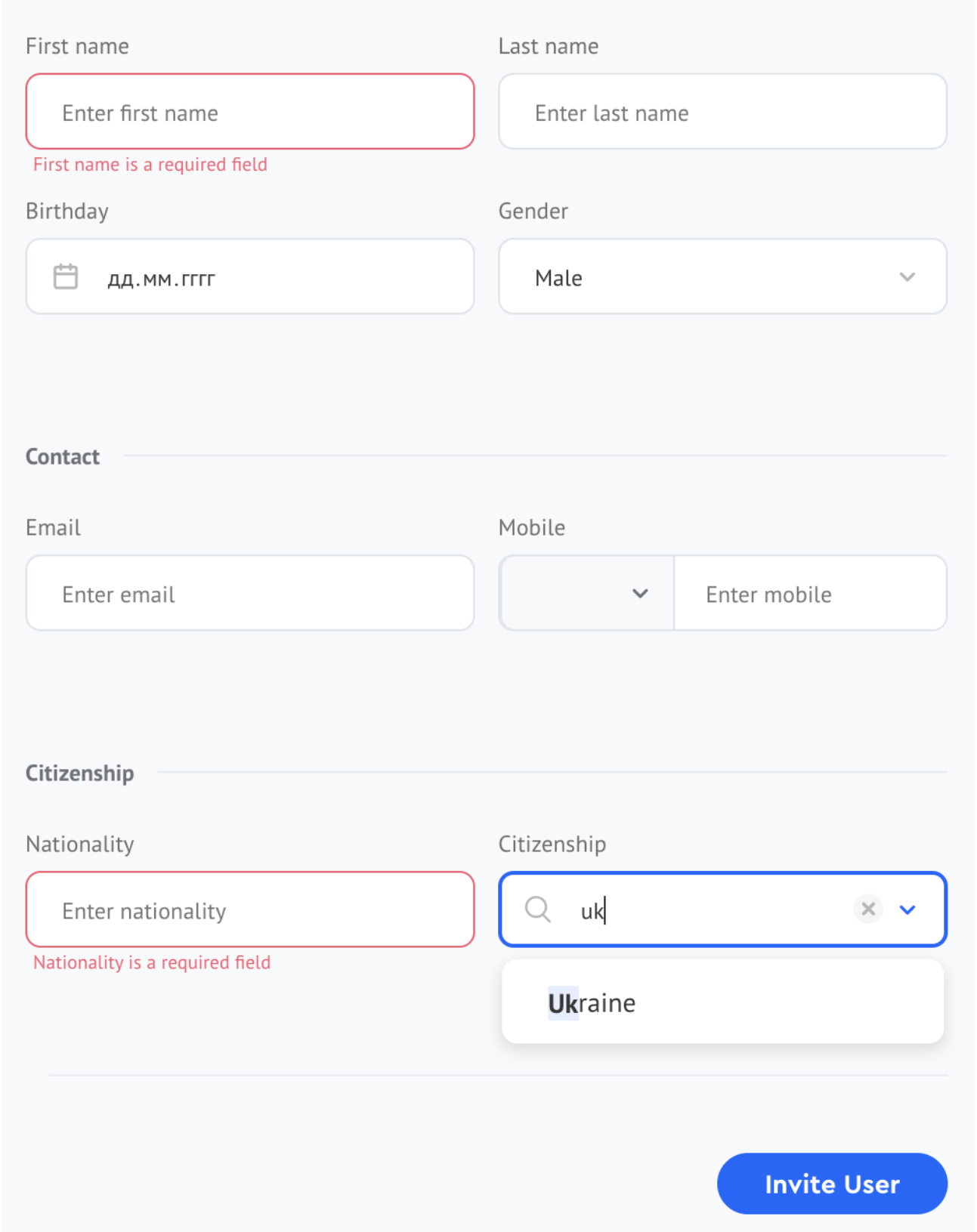
Приклад валідації на сервері отриманих даних у потоці з використанням RxJS. В даному випадку функція приймає потік даних ($ source), обробляє дані, проводить необхідні перевірки і повертає Observable в разі, якщо всі дії були коректними, або ж помилку, яка перериває потік і повертає інформацію про помилку на клієнтську частину. Або ж можна не переривати відразу потік, а ще й додатково обробити помилки, форматувати їх тощо., бо все залежить від необхідного функціоналу для конкретних випадків.

```
const PersonalSearchValidationSchema = object().shape( fields: {
  firstName: string().label( label: 'FirstName').when(
    keys: ['surname', 'postalCode'],  builder: {
     is: (surname, postalCode) =>
      isDefined(surname) && surname.length > 0 && (
       !isDefined(postalCode) || (isDefined(postalCode) && postalCode.length === 0)
      ),
     then: string().required()
    }
  ),
  firstNameBeginsWith: boolean().label( label: 'First Name Begins With'),
  postalCode: string().label( label: 'PostalCode').when(
    keys: ['surname', 'firstName'],  builder: {
     is: (surname, firstName) =>
      isDefined(surname) && surname.length > 0 && (
       !isDefined(firstName) || (isDefined(firstName) && firstName.length === 0)
      ),
     then: string().required()
    }
  ),
  //TODO: upd oneOf
  provinceState: string().label( label: 'Province State').oneOf( arrayOfValues: []),
  sin: string().label( label: 'Sin').when(
    keys: 'surname',  builder: {
     is: surname => isDefined(surname) && surname.length > 0,
     then: string().length( limit: 0)
    }
  ),
  //a way to combine multiples whens...or use them as nested https://github.com/jquense/yup/issues/335#issuecomment-619362597
  surname: string().when( keys: 'sin',  builder: {
    is: sin => isDefined(sin) && sin.length > 0,
    then: string().length( limit: 0),
    otherwise: string().when(  keys: ['dobYear', 'postalCode' , 'provinceState'],  builder: {
      //TODO: add all constants in specific file somewhere
      is: (dobYear, postalCode, provinceState) =>
       (isDefined(dobYear) && dobYear !== 'yyyy') ||
       (isDefined(postalCode) && postalCode.length > 0) ||
       (isDefined(provinceState) && provinceState.length > 0),
      then: string().required()
      }
    )
  }),
  surnameBeginsWith: boolean().label( label: 'Surname Begins With'),
  dobDay: string().label( label: 'Day of Birthday').matches( regex: /^(dd.+|(?!dd).*)$/),
  //TODO: upd oneOf with array of month
  dobMonth: string().label( label: 'Month of Birthday').oneOf( arrayOfValues: []),
  dobYear: string().label( label: 'Year of Birthday').matches( regex: /^(yyyy.+|(?!yyyy).*)$/),
```

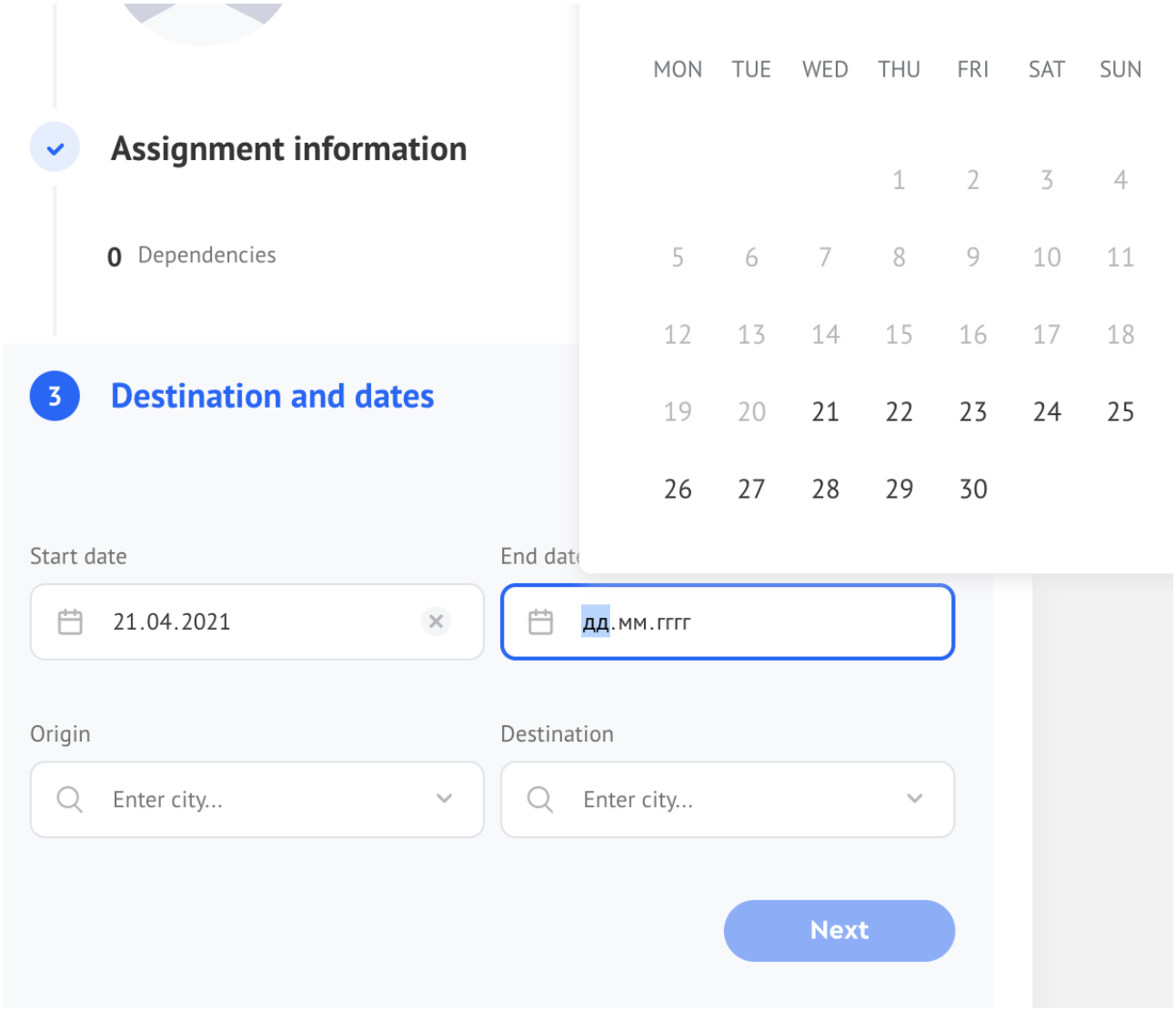
Приклад більш складної валідації на клієнті з використанням залежностей між полями. Тут визначається які дані повинно мати певне поле і додаткові умови, які описуються за допомогою .when і умов всередині даної функції, а саме is, then, otherwise. В when першим аргументом передаються ключі інших полів, значення яких будуть впливати на валідацію даного поля, яка обчислюється в builder, який описується другим аргументом .Самих методів для валідації полів в рази більше і це лише окремий приклад.



Приклад того, що задля того щоб запросити постачальника до системи, повинно бути корректно введено його email адресу, та це поле є обов’язковим. Тобто допоки ця умова не буде виконаною, валідація на клієнті не дозволить відправити запит на сервер.



Приклад валідації, коли користувач відразу бачить, яка умова не була виконана щодо певного поля, якщо він його намагався вводити.



Приклад того, що не у всіх випадках валідація потрібна на клієнті. В даному випадку користувачеві не дозволяється вибрати дату закінчення події раніше її початку. Таким чином, користувач не матиме змоги відправити невалідний запит.

Висновки:

В процесі написання кваліфікаційної роботи був проведений аналіз основних підходів до валідації даних на клієнтській та серверній частинах, виділено основні недоліки кожного з підходів, описано оптимізований підхід, котрий базується на компромісах між DRY та KISS , та який був описаний в цій роботі.

Даний підхід був перевірений на реальних проектах, що знаходяться в процесі розробки на певних етапах.

Доцільним застосування даного підходу є у випадках розробки будь-якої систем з використанням клієнт-серверної архітектури, у котрих валідації даних має приділятися особлива увага, а саме там, де важлива гарантія синхронізації валідацій на клієнтській і серверних частинах, для систем у котрих функціонал, що стосується валідації налічує тисячі і більше рядків, задля того, щоб на усунення потенційних проблем виділялося в рази менше часу, ніж при використанні інших підходів.

Використані джерела інформації

1. https://en.wikipedia.org/wiki/Client-server_model [Електронне джерело]
2. https://en.wikipedia.org/wiki/Verification_and_validation [Електронне джерело]
3. https://json-schema.org/ [Електронне джерело]
4. https://techrocks.ru/2019/02/10/best-software-engineering-principles [Електронне джерело]
5. https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat08255 [Електронне джерело]
6. https://www.researchgate.net/publication/347625018_Data_Validation [Електронне джерело]