НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.4

До захисту допущено:
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем» зі спеціальності 121 «Інженерія програмного забезпечення» на тему: «Метод комплексної валідації даних»

Виконав:
студент VI курсу, групи ІМ-32мп
Олег ДУБИНСЬКИЙ _____

Керівник:
доц. каф. ОТ, к.т.н., доц.,
Артем ВОЛОКИТА _____

Консультант з нормоконтролю:
проф. каф. ОТ, д.т.н., професор
Валерій ЖАБІН _____

Рецензент:
_____
_____ _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2024 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет/ННІ *Інформатики та обчислювальної технік*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність, спеціалізація *121 «Інженерія програмного забезпечення»*
(код і назва)

Освітньо-професійна програма *Інженерія програмного забезпечення комп'ютерних систем*
(код і назва)

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

Михайло НОВОТАРСЬКИЙ
(підпис) (ініціали, прізвище)

«___»_____20__ р.

ЗАВДАННЯ

на магістерьку дисертацію студенту

Дубинському Олегу Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Метод комплексної валідації даних

Науковий керівник дисертації Волокита Артем Миколайович, доцент кафедри ОТ, кандидат технічних наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 08 » листопада 2024 р. № 5016-С

2. Строк подання студентом дисертації _____

3. Об'єкт дослідження Процес комплексної валідації даних.

4. Предмет дослідження Методи та процеси комплексної валідації даних, який можна адаптувати під потреби користувачів

5. Перелік завдань, які потрібно розробити вивчення існуючих підходів валідації даних та їх недоліків, розробка підходу для комплексної валідації та бібліотеки, що його реалізує, перевірка ефективності підходу та бібліотеки на практичних прикладах

6. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Жабін В. І. проф. каф. ОТ, д.т.н., професор		
Розділ 1	Волокита А. М. доц. каф. ОТ, к.т.н., доц.		
Розділ 2	Волокита А. М. доц. каф. ОТ, к.т.н., доц.		
Розділ 3	Волокита А. М. доц. каф. ОТ, к.т.н., доц.		
Розділ 4	Волокита А. М. доц. каф. ОТ, к.т.н., доц.		

7. Дата видачі завдання 02.09.2024

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1	Затвердження теми роботи	02.09.2024-07.09.2024	
2	Огляд літератури	07.09.2024-17.09.2024	
3	Аналіз існуючих рішень	17.09.2024-25.10.2024	
4	Розробка підходів для реалізації методу комплексної валідації даних	25.10.2024-5.11.2024	
5	Проєктування підходу	5.11.2024-14.11.2024	
6	Програмна реалізація підходу	14.11.2024-20.11.2024	
7	Тестування підходу та виправлення помилок	20.11.2024-24.11.2024	
8	Розробка стартап-проєкту	24.11.2024-28.11.2024	
9	Оформлення магістерької дисертації	28.10.2024-29.11.2024	
10	Захист	19.12.24	

Студент _____ Олег ДУБИНСЬКИЙ
(підпис) (ім'я, прізвище)

Науковий керівник дисертації _____ Артем ВОЛОКИТА
(підпис) (ім'я, прізвище)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Метод комплексної валідації даних

студентом: Дубинським Олегом Ігоровичем

Робота складається із вступу та чотирьох розділів. Загальний обсяг роботи: 115 аркуші основного тексту, 41 ілюстрацій, 33 таблиці. При підготовці використовувалася література із 20 різних джерел.

Актуальність теми: В контексті сучасних програмних систем, важливість ефективної та надійної валідації даних є ключовою. Розвиток технологій вимагає складніших та більш гнучких підходів до перевірки вхідних даних, особливо у великих і розподілених системах. Кожна КС унікальна і потребує, щоб підхід до валідації даних в ній підстроювався під її потреби. Результуючий підхід та сама бібліотека може бути використана в широкому спектрі сфер, адже будь-де потрібна валідація даних, ці сфери включають: військову, медичну, фінансову, логістичну, освітню, енергетичну сфери, у сільському господарстві, державному управлінні.

Мета роботи - аналіз та вдосконалення підходів валідації даних через розробку нового підходу та його імплементація в бібліотеці під назвою Fields для мови програмування Scala, яка забезпечує комплексну валідацію даних з можливістю адаптування під потреби користувача.

Об’єкт дослідження – процес комплексної валідації даних.

Завдання дослідження:

1. Вивчення існуючих підходів валідації даних та їх недоліків.
2. Розробка підходу для комплексної валідації та бібліотеки, що його реалізує.
3. Перевірка ефективності підходу та бібліотеки на практичних прикладах.

Предмет дослідження – підхід комплексної валідації даних, який можна адаптувати під потреби користувачів.

Методи дослідження: аналіз літератури, практична реалізація бібліотеки та її тестування.

Практичне значення одержаних результатів роботи полягає в покращенні процесів валідації в програмному забезпеченні, збільшення гнучкості при виборі валідаційних структур даних, систем ефектів та власне структури самих помилок. Розроблено новий підхід до валідації, що надає гнучкий DSL та базується на тайпкласах з можливістю конфігурувати середовище виконання, валідаційну структуру та тип помилки, забезпечуючи більшу інформативність та гнучкість у порівнянні з традиційними методами.

Основні задачі дослідження випливають з його мети і полягають у наступному:

1. Аналіз існуючих бібліотек та фреймворків для валідації даних.
2. Визначити переваги та недоліки аналогів.
3. Розробити концепцію, яка включала б переваги аналогів та не мала б їх недоліків.
4. Розробити підхід та бібліотеку для комплексної валідації даних.
5. Покрити бібліотеку тестами для полегшення його майбутньої підтримки.
6. Створити документацію та домашню сторніку для бібліотеки
7. Розмістити бібліотеку на GitHub та зарелізити в глобальний artifactory для забезпечення можливості його публічного використання
8. Написати статтю про бібліотеку для демонстрації її можливостей широкому загалу.
9. Відображення результатів підходу, аналіз переваг та недоліків та опис можливих покращен розробленого програмного забезпечення та шляхи його розвитку.

Особистий внесок здобувача полягає в розробці концепції бібліотеки для комплексної валідації, її імплементації, дослідженні наявних підходів з аналізом їх переваг та недоліків врахуванні цих аспектів при розробці бібліотеки, тестування та аналіз отриманих результатів.

Ключові слова. SCALA, ПРОГРАМУВАННЯ, БІБЛІОТЕКА, ВАЛІДАЦІЯ, КОМПЛЕКСНА ВАЛІДАЦІЯ

ABSTRACT

for a master's thesis

implemented on topic: "Method for complex data validation"

by student: Dubynskyi Oleh Igorovych

The thesisconsists of an introduction and four chapters. Total volume of the work: 115 pages of the main text, 41 illustrations, 33 tables. Literature from 20 different sources was used in the preparation.

Relevance of the topic: In the context of modern software systems, the importance of efficient and reliable data validation is key. The development of technologies requires more complex and flexible approaches to input data validation, especially in large and distributed systems. Every CS is unique and requires method for data validation to adapt to its needs. The resulting approach and the library itself can be used in a wide range of areas, as data validation is required wherever data is needed, and these areas include: military, medical, financial, logistics, education, energy, agriculture, and public administration.

Research Objective: The purpose of the study is to analyse and improve data validation methods by developing a new method and its implementation as a library with name "Fields" for the Scala programming language that provides comprehensive and efficient data validation.

Research Object: the processes of complex data validation.

Research objectives:

1. Study of existing data validation methods and their shortcomings.
2. To develop a method for complex data validation and a library that implements it.
3. Testing the effectiveness of the method and library on practical examples.

Research Subject: method for complex data validation that can be adapted to consumer needs.

Research methods: literature analysis, practical implementation of the library and its testing.

The practical significance of the research results is to improve software validation processes, increase flexibility in choosing validation data structures, effect systems, and the structure of the errors themselves. Developed new approach to validation that provides flexible DSL and is based on typeclasses, with a ability to configure runtime, validation structure and error type, providing more information and flexibility compared to traditional methods.

The main research tasks of the study stem from its purpose and are as follows:

1. Analysis of existing libraries and frameworks for data validation.
2. Identify the advantages and disadvantages of analogues.
3. Develop a concept that includes the advantages of analogues and does not have their disadvantages.
4. Develop a method and library for complex data validation.
5. Cover the library with tests to facilitate its future support.
6. Create documentation and homepage for the library
7. Place the library on GitHub and release it to the global artifactory to enable its public use
8. Write an article about the library to demonstrate its capabilities to the general public.
9. Display the results of the approach, analyse the advantages and disadvantages and describe possible improvements to the developed software and ways to develop it.

The researcher's personal contribution is to develop the concept of a library for complex validation, its implementation, research of existing approaches with an analysis of their advantages and disadvantages, taking into account these aspects in the development of the framework, testing and analysis of the results.

Keywords. SCALA, PROGRAMMING, LIBRARY, VALIDATION, COMPLEX VALIDATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API – Application Programming Interface

БД – База Даних

DSL – Domain-specific language

КС – Комп’ютерна Система

ПЗ - Програмне Забезпечення

JSR - Java Specification Requests

I18n - Internationalization

CI/CD - Continuous Integration and Continuous Delivery

JSON - JavaScript Object Notation

IDE - Integrated Development Environment

VCS - Version Control System

AST - Abstract Syntax Tree

DTO - Data Transfer Object

POC - Proof of Concept

GC - Garbage Collection

JMH - Java Microbenchmark Harness

UX – User Experience

ВСТУП

В будь-якій системі чи застосунку завжди є потреба в валідації даних, які надає користувач чи які приходять з сторонніх систем і цей етап обробки даних є невід'ємною частиною будь-якого процесу обробки даних. Якщо невірні чи некоректні дані мали б змогу потрапити в КС безперешкодно, то це призводило до чималої кількості помилок та багів, які напряму впливають на роботоспроможність застосунку, а й відповідно на прибуток компанії, що цей застосунок розробляє. На щастя наразі поняття того що валідація даних потрібна – це аксіома і є чимало способів це робити.

Зазвичай всю логіку валідації доводиться писати власноруч враховуючи всі потреби до формату даних та сутностей в КС та до специфіки помилок в системі, також часто певні етапи валідації потребують виконання не тільки синхронного коду, а й асинхронного, потрібно взаємодіяти з сторонніми КС, БД, файловою системою і т.д, а також в залежності від бізнес логіки потрібно або перевіряти всі валідаційні правила та акумулювати помилки, або виводити перше правило, що не пройшло і не витрачати зайві обчислювальні ресурси на решту валідаційних правил.

Саме тому через наявність такого різноманіття потреб при валідації даних є доцільним створити бібліотеку, яка б давала можливість все це конфігурувати і писати валідацію в єдиному стилі без зайвих додаткових саморобних надстройок, а де це потрібно надавати можливість для кастомізації під потреби конкретної КС.

Метою магістерської роботи є покращення та спрощення процесів валідації в КС за рахунок створення методу для комплексної валідації даних та його реалізації у вигляді бібліотеки для мови програмування Scala.

Ця бібліотека надаватиме можливість обирати з яким середовищем виконання працювати(синхронне, асинхронне, користувацьке), з якою структурою валідації(fail-fast, accumulate, будь-які інші структури за бажанням) та даватиме свободу вибору типу помилки.

При створені архітектури бібліотеки буде враховано можливість інтегрування з іншими бібліотеками, які надають ті чи інші середовища виконання та валідаційні структури. Додатковою цілю буде виводити місце або поле щодо якого було застосовано те чи інше валідаційне правило в разі виникнення помилки.

В результаті створення бібліотеки не мало важливим буде опублікувати її в публічний репозиторій для можливості використання іншими розробниками, створити публічний сайт з документацією бібліотеки та документацію всіх класів та функцій бібліотеки.

Для досягнення мети дослідження поставлено і вирішено такі завдання:

− дослідження і аналіз підходів до валідації даниї;

− дослідження і аналіз існуючих рішень для валідації даних;

− розробка самої бібліотеки

− розробка сайту з документацією бібліотеки

− публікація бібліотеки в артифакторів для загального доступу

− написання онлайн статті на ресурсі Medium

− розробка стартап-проєкту.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Загальні відомості про валідацію

Валідація даних – це процес перевірки точності, правильності та відповідності даних перед їх використанням, це критичний аспект забезпечення якості даних у будь-якій КС, оскільки невірні чи неповні дані можуть призводити до помилок в роботі системи. Основна мета валідації – це забезпечення того, щоб всі дані відповідали заданим унікальним вимогам, визначеним для конкретної КС.

Серед основних варіантів валідації полів варто визначити такі види перевірок:

- Коректність - дані мають бути коректними відповідно до визначених правил. Наприклад, номер телефону має містити тільки цифри, а адреса електронної пошти повинна включати символ "@".
- Повнота - усі необхідні дані повинні бути заповнені. Якщо певне поле є обов'язковим, воно не повинно залишатися порожнім.
- Послідовність - дані мають бути логічно узгодженими між собою. Наприклад, дата народження не може бути більшою за поточну дату.
- Унікальність - деякі дані повинні бути унікальними. Наприклад, ідентифікаційний номер користувача не може повторюватися.
- Форматування - дані повинні відповідати певному формату. Наприклад, дата повинна бути в форматі "ДД/ММ/РРРР".

Всі ці перевірки також можуть ускладнюватись в залежності від різних варіацій валідації:

- Синтаксична валідація - перевіряє, чи відповідають дані певному формату або структурі. Наприклад, перевірка формату номера телефону чи дати.

- Семантична валідація - перевіряє логічну правильність даних. Наприклад, вік користувача повинен бути більшим за нуль, або дата початку події не може бути пізнішою за дату завершення.
- Перехресна валідація - включає перевірку взаємозалежності між різними полями даних. Наприклад, якщо вказано країну, то регіон повинен бути дійсним для цієї країни.

В цілому процес реалізації валідації поділяється на такі етапи:

1. Збір вимог - спершу визначаються вимоги до даних, які включають правила та обмеження, що будуть застосовані під час валідації.
2. Визначення правил валідації - створюються конкретні правила, яким повинні відповідати дані. Це можуть бути умови на значення, формат, діапазони або навіть взаємозв'язки між полями.
3. Реалізація валідаційних механізмів - на цьому етапі валідаційні правила реалізуються у вигляді коду, що перевіряє дані перед їх використанням в системі.
4. Тестування і перевірка - після реалізації проводиться тестування правил валідації для перевірки її ефективності та правильності.

В практичному плані валідація даних є важливим етапом у багатьох галузях. Наприклад, у фінансовій сфері дані про транзакції повинні бути точними і повними, щоб уникнути помилок або шахрайства. У медичній сфері невірні дані можуть вплинути на точність діагнозу або лікування пацієнта. У комерції валідація даних допомагає запобігти введенню некоректних адрес доставки чи контактної інформації, що забезпечує належну роботу служби обслуговування клієнтів.

На практиці використовуються різноманітні інструменти для валідації даних, починаючи від саморобних рішень чи простих бібліотек для валідації форм у веб-розробці до складних систем для автоматичного аналізу великих обсягів даних. В кожній мові програмування є свої рішення для валідації даних, десь це встроєно в мову, десь є фреймворки чи бібліотеки і можна обирати чим

користуватись, в подальших розділах буде розглянуто існуючі рішення для мови програмування Scala [3].

1.2 Огляд існуючих інструментів для валідації

Валідація даних це дуже важливий аспект будь-якого застосунку саме тому наразі вже існує велика кількість бібліотек та фреймворків для валідації даних в розрізі мови програмування Scala. Завдяки своєму інноваційному компілятору та парадигмам яких притримуються розробники мови, існує чимало вбудованих інструментів для метапрограмування, що дозволяє авторам бібліотек створювати будь-які надстройки над мовою задля досягнення бажаними ними результатів. Завдяки цьому в Scala світі існує чимало бібліотек для валідації даних кожна з яких по своєму особлива. Серед найпопулярніших можна виокремити такі рішення:

- Dupin
- Accord
- Octopus
- Scala Either
- Cats Validated

Тому саме їх і буде розглянуто далі та проаналізовано їх плюси та мінуси.

1.2.1 Dupin

Dupin - це мінімальна, ідіоматична, настроювана бібліотека для валідації в мові програмування Scala [4].

Серед цікавих властивостей бібліотеки можна виокремити:

- Прозорий та компонований підхід до валідації
- Можливість обирати тип валідаційної помилки
- Можливість обрати середовище виконання всередині валідатора (Future, IO, синхронне, тощо...)
- Наявність можливості опису парсингу, окрім валідації

- Вона побудована поверх бібліотеки Cats та слідує її парадигмам та підходам, що робить бібліотеку привабливою для проектів з Cats
- Наявна підтримка Scala 3, Scala JS або Scala Native

Дана бібліотека побудована навколо двох тайпкласів:

- Validator[F[_],E,A] – конструкція валідатора для типу A, представляє собою функцію з аргументом A та результатом F[ValidatedNec[E, Unit]]
- Parser[F[_], E, A, B] - є синтаксичним аналізатором з типу A в B, представляє собою функцію з аргументом A і результатом F[IorNec[E, B]]

Бібліотека опублікована в Maven Central та розмішена в системі контролю версій GitHub за посиланням https://github.com/yakivy/dupin, в бібліотеки немає окремого сайту з документацією, проте її README файл включає певний обсяг базової документації, якої буде достатньо, щоб почати нею користуватись.

API бібліотеки полягає в тому, щоб оголошувати для кожного валідованого типу тайпклас Validator або Parser якщо це парсинг. Бібліотека надає ряд базових конструкцій для створення валідаційних правил та їх компонування в більш складні валідаційні конструкції (рис. 1.1).

```
import cats._
import dupin.basic.all._

//validator for simple type or value class
implicit val nameValidator: BasicValidator[Name] = BasicValidator
    .root[Name](_.value.nonEmpty, c => s"${c.path} should be non empty")

//idiomatic validator for complex type
implicit val memberValidator: BasicValidator[Member] =
    nameValidator.comapP[Member](_.name) combine
    BasicValidator.root[Int](
        a => a > 18 && a < 40,
        c => s"${c.path} should be between 18 and 40"
    ).comapP[Member](_.age)

//same validator but with combination helpers for better type resolving
val alternativeMemberValidator: BasicValidator[Member] = BasicValidator
    .success[Member]
    .combineP(_.name)(nameValidator)
    .combinePR(_.age)(a => a > 18 && a < 40, c => s"${c.path} should be between 18 and 40")

//derived validator
implicit val teamValidator: BasicValidator[Team] = BasicValidator
    .derive[Team]
    .combineR(_.members.size <= 8, _ => "team should be fed with two pizzas!")

//two stage validator
val failingTeamValidator: BasicValidator[Team] = teamValidator
    .andThen(BasicValidator.failure[Team](_ => "validation error after heavy computations"))
```

Рис. 1.1. Приклад опису валідації за допомогою Dupin

Після оголошення валідатора бібліотека надає синтаксис для його застосування: можна перевірити чи валідний той чи інший об’єкт, наявна інформація про поле в якому виникла помилка, а також інформація про те яка саме це була помилка.

Бібліотека дозволяє використовувати власні типи помилки та працювати з системами ефектів. Наявні способи створення власних базових конструкцій для валідації.

З мінусів можна навести те що немає стандартного набору валідаційних правил та помилок всі умови і помилки треба створювати власноруч також користувач не може обрати валідаційну структуру вона зафіксована як Cats ValidatedNec відповідно немає підтримки fail-fast валідації. Також немає змоги створювати багаторівневі перевірки вкладених полів в яких тип колекції чи мапи і не можна робити перехресні валідації, відповідно комплексна валідація теж відсутня.

1.2.2 Accord

Accord – бібліотека з унікальним підхідом до валідації, за рахунок якого валідація пишеться майже, що англійською мовою. У порівнянні з валідацією JSR 303 та Scalaz вона має на меті забезпечити наступне [6]:

- Компонованість: Оскільки JSR 303 базується на анотаціях, через що правила валідації не можна компонувати. На відміну від JSR 303 правила валідації Accord легко компонуються.
- Простоту: Accord надає надзвичайно простий підхід для визначення правил валідації, використовуючи макроси, які автоматично запам’ятовують шлях полів, що валідуються (рис. 1.2.).
- Самодостатність: Accord базується на макросах, але є повністю автономним, а отже, з залежностей він має лише стандартну бібліотеку та модуль рефлексії Scala.

- Інтегрованість: Окрім надання власної бібліотеки DSL та бібліотеки відповідності, Accord розроблений для легкої інтеграції з різноманітними інструментами екосистеми Scala, і надає готову підтримку Scala.js, а також модулі інтеграції для Spring Validation, Specs2 та ScalaTest.

```
import com.wix.accord.dsl._    // Import the validator DSL

case class Person( firstName: String, lastName: String )
case class Classroom( teacher: Person, students: Seq[ Person ] )

implicit val personValidator = validator[ Person ] { p =>
  p.firstName is notEmpty                    // The expression be
  p.lastName as "last name" is notEmpty      // You can also expl
}

implicit val classValidator = validator[ Classroom ] { c =>
  c.teacher is valid          // Implicitly relies on personValidat
  c.students.each is valid
  c.students have size > 0
}
```

Рис. 1.2. Приклад опису валідації за допомогою Accord

Бібліотека опублікована в Maven Central та розмішена в системі контролю версій GitHub за посиланням https://github.com/wix-incubator/accord, в бібліотеки є окремий сайту з документацією та детальною інформацією про неї його можна знайти за посиланням https://wix.github.io/accord/. Там є багато прикладів використання, а також доступних API.

Існує можливість оголошення власних комбінаторів та велика кількість вбудованих базових валідаційних правил. Тип помилки в цій бібліотеці строго закріплений до певного типу визначеного бібліотекою та включає повідомлення помилки та шлях помилки.

З недоліків можна виокремити те що бібліотеку перестали підтримувати і вона недоступна для версій Scala старше 2.13, а також занадто складні макроси всередині імплементації, що робить розширення бібліотеки досить складним. Також має всі ті ж мінуси що в dupin на додачу до цього не можна обирати середовище виконання, доступна лише синхронна валідація.

1.2.3 Octopus

Octopus - це бібліотека Scala для валідації без шаблонного коду, як її позиціонує автор бібліотеки [5].

Вона визначає абстракцію Validator[T], що надає композитний DSL для визначення правил валідації для визначених користувачем типів і може автоматично створювати валідатори для дата класів, кортежів, закритих ієрархій класів і різних стандартних типів Scala шляхом використання існуючих визначень.

Бібліотека опублікована в Maven Central та розмішена в системі контролю версій GitHub за посиланням https://github.com/krzemin/octopus, в бібліотеки немає окремого сайту з документацією, проте існуючого README документу достатньо, щоб використовувати бібліотеку на базовому рівні, для детального вивчення доведеться власноруч вивчати код бібліотеки.

API бібліотеки полягає в тому, щоб оголошувати для кожного валідованого типу тайпкласс Validator, використовуючи наявні базові конструкції або написавши власну логіку перевірки. Також доступні методи для об'єднання в більш складні валідаційні правила (рис. 1.3.).

Оголошений валідатор має синтаксис для його застосування, перевірки чи правильність, можна переглядати інформацію про місце виникненя помилки, а також є інформація про саму помилку.

```
import octopus.dsl._

implicit val userIdValidator: Validator[UserId] = Validator[UserId]
  .rule(_.id > 0, "must be positive number")

implicit val emailValidator: Validator[Email] = Validator[Email]
  .rule(_.address.nonEmpty, "must not be empty")
  .rule(_.address.contains("@"), "must contain @")
  .rule(_.address.split('@').last.contains("."), "must contain . after @")

implicit val potalCodeValidator: Validator[PostalCode] = Validator[PostalCode]
  .ruleVC((_: String).length == 5, "must be of length 5")
  .ruleVC((_: String).forall(_.isDigit), "must contain only digits")

implicit val addressValidator: Validator[Address] = Validator
  .derived[Address] // derives default validator for Address
  .rule(_.city, (_: String).nonEmpty, "must not be empty")
  .rule(_.street, (_: String).nonEmpty, "must not be empty")
```

Рис. 1.3. Приклад опису валідації за допомогою Octopus

Наявна інтеграція з Cats Validated та можливість обирати середовище виконання, які підтримує Cats. Наявний DSL доволі приємний в використанні, проте не дає можливість використовувати сторонні структури для валідації, та помилки строго зашиті до типу строки.

Одним з найбільших плюсів, є простота синхронних правил валідації, за рахунок чого для базових потреб бібліотека є дуже швидкою.

Наслідує свої недоліки з dupin, оскільки має аналогічну архітектуру.

1.2.4 Scala Either

В стандартній бібліотеці Scala є вбудована структура даних під назвою Either, яка має низькорівневий API для написання fail-fast валідації. Вона не здатна акумулювати помилки, що виникають, проте якщо існує потреба саме в валідації до першої помилки цей підхід може бути дуже ефективним.

Наявний повний контроль над типом помилки та ряд допоміжних методів для створення правил валідації, проте немає вбудованих базових валідаційних правил, їх доведеться реалізовувати самому з нуля (рис. 1.4).

```
def validateLastName(lastName: String): Either[DomainValidation, String] =
   Either.cond(
     lastName.matches("^[a-zA-Z]+$"),
     lastName,
     LastNameHasSpecialCharacters
   )

def validateAge(age: Int): Either[DomainValidation, Int] =
   Either.cond(
     age >= 18 && age <= 75,
     age,
     AgeIsInvalid
   )

 def validateForm(username: String, password: String, firstName: String, lastName: String,
age: Int): Either[DomainValidation, RegistrationData] = {

   for {
     validatedUserName <- validateUserName(username)
     validatedPassword <- validatePassword(password)
     validatedFirstName <- validateFirstName(firstName)
     validatedLastName <- validateLastName(lastName)
     validatedAge <- validateAge(age)
   } yield RegistrationData(validatedUserName, validatedPassword, validatedFirstName, valid
atedLastName, validatedAge)
 }

}
```

Рис. 1.4. Приклад опису валідації за допомогою Either

Стандартна документація Scala дає можливість досить швидко розібратись з API Either тому це великий плюс.

Незважаючи на все важко уявити Either в якості повноцінної бібліотеки для валідації через його низькорівневість та потребу в написанні додаткового коду специфічного для вашого застосунку, але за рахунок цього валідація написана на ньому буде неймовірно швидкою і без зайвих алокацій. Також Either представляє собою чималу цінність в якості структури даних для fail-fast валідації, що можна використати при розробці бібліотеки.

1.2.5 Cats Validated

Cats - це бібліотека, яка надає абстракції для функціонального програмування мовою програмування Scala.

Scala підтримує як об'єктно-орієнтоване, так і функціональне програмування, і це відображено в гібридному підході стандартної бібліотеки. Cats прагне надати абстракції для функціонального програмування [1], які є фундаментальними конструкціми для побудови застосунків, ці конструкції є бінарно сумісними, модульними, доступними та ефективними.

Ширша мета Cats полягає у створенні основи для екосистеми чистих, типізованих бібліотек для підтримки функціонального програмування у Scala додатках. Незважаючи на складні поняття теорії категорій покладені в основу бібліотеки, розробникам не потрібно знати нічого про теорію категорій, щоб використовувати Cats.

В цій бібліотеці присутня абстракція під назвою Validated, яка позиціонується як інструмент для опису валідації. Вона заточена під акумулятивну валідацію, тобто валідований об’єкт перевіряється на відповідність усім правилам перед тим як віддати перелік помилок, що виникають. В загальному випадку зазвичай треба саме перелічити усі помилки, наприклад якби при реєстрації на сайті при нажаті кнопки зареєструватись користувачеві вказувало на помилки по одній, це призвело б до жахливого

досвіду користувача UX, коли він з разу в раз міняє дані, щоб виправити помилку і отримує нову, замість того, щоб виправити все за один раз.

Саме для цих цілей і існує Validated, API якого дуже схожий на вбудований в мову тип Either проте він акумулює помилки, а не зупиняється на першій, що зустрічається.

Так виглядає структура оголошення цієї астракції (рис. 1.5).

```
sealed abstract class Validated[+E, +A] extends Product with Serializable
final case class Valid[+A](a: A) extends Validated[Nothing, A]
final case class Invalid[+E](e: E) extends Validated[E, Nothing]
```

Рис. 1.5. Оголошення Validated

Виглядає просто, тому що це доволі низькорівневий підхід, для проведення валідації користувачу доведеться власноруч писати свої правила валідації, без вбудованих базових примітивів. Проте в наявності є ряд допоміжних методів для спрощення створення цих конструкцій. Також плюсом є те що в нас є повний контроль над типом помилки та валідаційною структурою (рис. 1.6). А ще можна реалізувати парсинг на основі цієї структури.

```
sealed trait FormValidatorNec {

  type ValidationResult[A] = ValidatedNec[DomainValidation, A]

  private def validateUserName(userName: String): ValidationResult[String] =
    if (userName.matches("^[a-zA-Z0-9]+$")) userName.validNec else UsernameHasSpecialCharact
ers.invalidNec

  private def validatePassword(password: String): ValidationResult[String] =
    if (password.matches("(?=^.{10,}$)((?=.*\\d)|(?=.*\\W+))(?![.\\n])(?=.*[A-Z])(?=.*[a-
z]).*$")) password.validNec
    else PasswordDoesNotMeetCriteria.invalidNec

  private def validateFirstName(firstName: String): ValidationResult[String] =
    if (firstName.matches("^[a-zA-Z]+$")) firstName.validNec else FirstNameHasSpecialCharact
ers.invalidNec

  private def validateLastName(lastName: String): ValidationResult[String] =
    if (lastName.matches("^[a-zA-Z]+$")) lastName.validNec else LastNameHasSpecialCharacter
s.invalidNec

  private def validateAge(age: Int): ValidationResult[Int] =
    if (age >= 18 && age <= 75) age.validNec else AgeIsInvalid.invalidNec

  def validateForm(username: String, password: String, firstName: String, lastName: String,
age: Int): ValidationResult[RegistrationData] = {
    (validateUserName(username),
    validatePassword(password),
    validateFirstName(firstName),
    validateLastName(lastName),
    validateAge(age)).mapN(RegistrationData)
  }

}
```

Рис. 1.6. Приклад опису валідації за допомогою Cats Validated

Як видно з рисунку для валідації форми було окремо створено методи для валідації кожного з полів, всі перевірки створення помилок та результатів роблять вручну.

1.2.6 Висновки по існуючим рішенням для валідації

Оглянувши існуючі рішення стає зрозуміло, що Scala дуже популярна мова програмування і вона має міцну спільноту, яка за роки існування мови створила чимало різноманітних бібліотек для вирішення ряду задач, включаючи і задачі зі сфери валідації даних. Як бачимо з результатів огляду є чимало різноманітних рішень, кожне має свої переваги та недоліки, проте всі вони в чомусь схожі між собою і таке відчуття, що кожен свого часу надихався один одним в процесі розробки. Саме тому оскільки цілюю є створити якісну бібліотеку для комлексної валідації даних виникає потреба проаналізувати всі нюанси та властивості цих бібліотек та порівняти їх між собою. А також проаналізувати методи та підходи, які вони використовують та як це впливає на їх використання.

1.3 Аналіз та порівняння існуючих інструментів для валідації

Оглянувши існуючі рішення для валідації на мові програмування Scala уже прослідковуються тенденції та властивості існуючих рішень. В цьому розділі буде доцільно проаналізувати всі ці властивості, їх вплив на рішення та оглянути як ці властивості виглядають в кожному з аналогів.

Далі розглянемо ряд властивостей бібліотек для валідації, умовно їх можна поділити на загальну та технічну категорії.

1.3.1 Загальні властивості бібліотек валідації

Перший важливий аспект для кожної з бібліотек це наявність сайту. Якщо є сайт та він наповнений інформацією про рішення, як його використовувати, детальною документацією, прикладами та різноманітними нюансами, що

можуть виникнути при використанні інструменту - це дуже спростить її використання та позитивно діятиме на її використання. Тобто більша кількість людей буде використовувати її.

Навіть якщо сайту немає в будь-якому разі обов'язково кожна бібліотека має включати документацію з прикладами використання та детальним описом всього, що може цікавити користувачів бібліотеки. Відсутність документації унеможливлює нормальне використання бібліотеки та негативно впливає на її популярність.

Також немало важливо є те що бібліотека має бути актуальною та підтримуваною, тобто автори бібліотеки не мають закидувати її та мають продовжувати, вносити нові правки, підтримувати її в актуальному стані та випускати нові версії, щоб йти нога в ногу з новими функціями доступними в мові програмування Scala.

Для бібліотеки дуже важливим аспектом є її спільнота – це люди які її використовують та контриб'ютять у її розвиток. Доповідать про знайдені ними недоліки та генерують нові ідеї для покращення ПЗ. Саме тому дуже важливо, щоб бібліотека була open-source і була розмішена на популярному ресурсі контролю версій, як от наприклад Github.

Якщо розробник бібліотеки бажає, щоб у всіх був доступ до неї, невід'ємною частиною буде розмістити зкомпільовані модулі бібліотеки в глобально доступному artifactory як от наприклад Maven Central чи Sonatype, щоб кожен розробник мав змогу в декілька кліків інсталювати бібліотеку та почати працювати з нею.

Як уже згадувалось дуже важливо, щоб у бібліотеки було міцне комюніті, тому важливим аспектом також є наявність каналів комунікації всередині нього, а також наявність статей про бібліотеку, що могло б привернути увагу широкого загалу до неї.

1.3.2 Технічні властивості бібліотек валідації

Незважаючи на те що дуже важливим аспектом будь-якої бібліотеки є всі загальні властивості перелічені раніше, найважливішим при виборі бібліотеки є її технічна складова. Бібліотеки для валідації мають чимало складних технічних аспектів, які буде розглянуто далі.

Будь-яка зручна валідація починається з наявністі вбудованих базових валідаційний конструкцій в бібліотеці. Навіщо дублювати в сотні місць базову логіку перевірки максимального та мінімального значень числа, розмірності строки чи перевірки відповідності регулярному виразу і так далі. Всі такі конструкції обов'язково мають бути частиною бібліотеки і користувач не повинен їх визначати сам.

Валідація це дуже гнучка річ, в кожній системі потрібно перевіряти свої нюанси і необов'язково вистачить вбудованих конструкцій бібліотеки. Саме тому потрібно передбачати можливість створення нових примітивних валідаційних конструкцій. Це має бути зручно, просто та бути ефективним способом композиції валідаційної логіки.

В реальному світі валідація не обмежується синхронними перевірками даних на відповідність певним примітивним правилами, дуже часто доводиться перевіряти такі умови, які потребують виконання асинхронних дій, як от наприклад перевірка, що ім'я користувача не зайняте потребує виконати операцію в базі даних для визначення валідності зробленого запиту. Саме тому в бібліотеці має бути присутня можливість вибору та роботи з різноманітними системами ефектів. Системи ефектів це ряд різноманітних типів ефектів, що абстрагують виконання програми за певними правилами. В Scala зазвичай їх реалізують за допомогою інструменту під назвою Monad, яка являє собою тайпклас/інтерфейс, що має методи map, flatMap, unit, bind та інші за допомогою можна конструювати програму незважаючи на те як саме виконується той чи інший ефект. Завдяки цьому можна легко перетворити синхронну програму на асинхронну чи заміняти логіку виконання на будь яку бажану іншу.

Можливість вибору та роботи з валідаційними структурами даних є дуже важливим аспектом бібліотеки для валідації даних тому що в залежності від специфіки валідації може виникати потреба використовувати ту чи іншу структуру даних, вона впливає на те як саме комбінуються та виконуються валідаційні правила. В залежності від неї валідація може відбуватись до першої помилки(fail-fast) чи агрегувати всі можливі помилки, які помічає політика валідації(accumulate). Також користувач може захотіти створити свою структуру даних, саме тому це має бути конфігурованою опцією в бібліотеці.

Немало важливим є можливість вибору власного типу помилки, яка буде повертатись в якості інформації про те чому об’єкт є невалідним. Завдяки цьому валідацію можна заточити під потреби користувача і це відкриває можливість інтернаціоналізації застосунку(i18n), покращити обробку помилок за рахунок того що під кожну помилку можна створити свій тип і детально обробляти ці помилки, покращити інформативність помилок для користувачів та спростити протокол взаємодії з клієнтами вашого застосунку. Без можливості кастомізувати тип помилки було б неможливо врахувати потреби усіх користувачів бібліотеки.

Можливість кастомізувати помилки, які видають базові валідаційні конструкції також потрібно враховувати, адже який сенс в можливості вибру власних типів помилок для своїх правил, якщо встроєні правила все ще повертають помилки, які не відповідають формату вашого застосунку. Саме тому важливо надати можливість редагувати помилки, які надсилаються стандартними валідаційними структурами бібліотеки і це повинно бути досить просто зробити.

Інформація про джерело виникнення помилки – це невід’ємна частина будь якої валідації, адже як по іншому визначити для якого поля виникла та чи інша помилка. Кожна бібліотека для валідації, обов’язково має реалізувати механізм визначення виникнення помилки та надавати користувачу цю інформацію у разі виникнення помилки. Це має добре працювати не тільки для

визначеннь верхнього рівня “username”, а й для глибоко-вкладених визначень по типу “copyTo.users[24].details.username”.

Якість стандартного DSL для опису валідаційних правил дуже сильно впливає на просту, ефективність та популярність бібліотеки, це має бути зрозумілий синтаксис, який легко читати, розуміти та розширювати. Немає бути ніяких магічних дій, незрозумілих пересічному користувачу. Валідаційні правила мають бути якомога ближче до людської мови, щоб можна було легко та швидко прочитати цей DSL та визначити, що ж валідується в тому чи іншому випадку.

Наявність fail-fast валідації також дуже важливий аспект бібліотек валідації даних, адже в багатьох випадках потрібно не виконувати всі валідаційні правила, а дійти до першого невірного та повернути інформацію про це користувачу. Наприклад якщо ім’я користувача має невірну довживу немає сенсу робити якісь інші дорогі валідації, які можуть робити side-ефекти по типу звернень в базу даних.

З цього випливає інший аспект, а саме наявність механізму short-circuit. Суть цього механізму полягає в тому, щоб не запускати обчислення виразів, якщо це не потрібно, багато бібліотек для валідації мають fail-fast підхід, але не враховують short-ciruit механізм і незважаючи на те що помилка в результаті повертається лише одна, все ж обчислення насправді запускаються, що не є очевидним для кінцевого користувача, але рано чи пізно він з цим стикнеться і навряд чи подякує розробникам цих бібліотек.

Незважаючи на частоту використання fail-fast валідації, набагато частіше застосовується accumulate валідації. Який сенс змушувати користувача чи суміжну систему, виправляти помилки по одній, поки не будуть виправлені всі помилки, це збільшує навантаження на систему та додає непотрібні додаткові кроки, саме тому набагато ефективнішим буде зібрати всі помилки та повідомити про них користувачу, тоді він зможе виправити їх всі махом та послати новий запит з уже валідними даними.

Якщо йде мова про якісну бібліотеку то варто згадати про мінімальну кількість залежностей в ній. Чим більше залежностей має бібліотека тим складніше її підтримувати та використовувати користувачам, саме тому дуже важливо, щоб core функціонал був написаний взагалі без сторонніх залежностей, але це не означає, що не має потреби інтегруватись з іншими бібліотеками, це можна робити у вигляді окремих модулів, що підключаються за бажанням.

Інший пункт це інтегрованість з іншими популярними бібліотеками для блага користувачів. У випадку коли core функціонал написаний якісно, створити окремі модулі для інтеграції з іншими бібліотеками є досить зручним, більше того, якщо архітектура бібліотеки вдала навіть самі користувачі бібліотеки зможуть інтегрувати її з власними бібліотеками, що збільшує ефективність розробленої бібліотеки експоненційно.

Простота написання комплексних валідаційних правил – це один з найважливіших аспектів для валідації в реальних бізнес застосунках, зазвичай в реальному світі валідація може бути настільки складною, що можна зійти з розуму намагаючих розібратися в тому що ж там відбувається, саме тому треба не тільки мати гарний DSL в бібліотеці, а й передбачити, щоб з ростом валідаційних політик код можна було читати так само легко як і раніше.

І останнє, але не найменш важливе – це можливість генерації документації щодо політики валідації. Було б дуже зручно мати можливість написати валідаційну логіку і автоматично згенерувати інформацію про те що ж там валідується та надати її клієнту в якості документації. Саме тому було б чудово спробувати реалізувати такий функціонал в валідаційній бібліотеці або хоча б покласти основи.

1.3.3 Порівняння властивостей аналогів

На основі визначених властивостей, варто провести порівняльний аналіз існуючих рішень та визначити їх сильні та слабкі сторони. Після цього варто

буде зробити висновки, щодо того що обов’язково треба буде імлементувати та що можна зробити нове чого немає у аналогічних рішень.

Таблиця 1.1

Порівняння бібліотек

Бібліотека / Властивість	Accord	Dupin	Octopus	Either	Validated
Сайт	+	-	-	-	+
Документація	+	+	+	+	+
Підтримка	-	+	+	+	+
Канали зв’язку	-	-	-	+	+
Спільнота	-	-	-	+	+
Open-source	+	+	+	-/+	+
Статті	+	-	-	+	+
Доступність	+	+	+	+	+
Базові конструкції	+	-	-	-	-
Власні конструкції	+/-	+	+	+	+
Вибір середовища виконання	-	+	+	+/-	+/-
Вибір структури даних	-	-	-	-	-
Вибір помилки	-	+	+	+	+
Шлях виникнення помилки	+	+	+	+/-	+/-
DSL	+	-	-	-	-
fail-fast	-	-	-	+	+
short-circuit	N\A	N\A	N\A	+	+
accumulate	+	+	+	-	+
Відсутність залежностей	-	-	-	+	+/-
Комплексні валідації	-	-	-	+	+
Перехресна валідація	-	-	-	-	-

Як видно з таблиці ідеального рішення серед аналогів немає, тому оптимальним варіантом виглядає розглянути всі підходи і обрати вдалі рішення з кожного, саме тому варто буде детально проаналізувати, як вони працюють в процесі розробки нашого методу та його програмної реалізації - бібліотеки.

Варто відзначиьи що найбільш проблематичними властивостями для всіх бібліотек є перехресна валідація, комплексні валідації, відсутність залежностей, проблеми з fail-fast/short-circuit, слабкий DSL та відсутність базових конструкцій.

В результаті аналізу існуючих рішень вийшло виділити ряд властивостей, які має мати бібліотека для комплексної валідації даних, було проведено порівняльний аналіз всіх бібліотек згідно цих властивостей та визначено на архітектуру яких бібліотек варто буде звернути уваги при реалізації власного рішення. Також завдяки аналізу всіх властивостей стає зрозуміло, які речі обов’яково повині будуть імплементовані в бібліотеці, оскільки поставлено задачу досягти успіху і створити оптимальну бібліотеку для валідації комплесних даних.

1.4 Постановка задачі

Проаналізувавши рішення аналоги можна сформулювати мету та задачу дослідження.

Мета: Розробити бібліотеку для комплексної валідації даних на мові програмування Scala, яка буде легко інтегруватися у різні проекти, підтримуючи принцип fail-fast, short-circuit та накопичувальної (accumulate) валідації. Бібліотека повинна мати можливість налаштування та розширення для різноманітних сценаріїв валідації без зовнішніх залежностей.

Вимоги до бібліотеки:

- Базові конструкції - підтримка простих валідацій, таких як перевірка на null, діапазони, відповідність типам, розміри колекції і т.д.
- Власні конструкції - можливість створення власних валідаційних конструкцій користувачами.
- Комплексні валідації - можливість перевіряти складні взаємозв'язки між полями.
- Перехресна валідація - перевірка взаємозв'язку між різними полями або наборами дании.
- Вибір помилки - можливість налаштувати, які саме помилки мають бути виведені в результаті невдалої валідації.
- Шлях виникнення помилки - фіксація та відображення місця (шляху) виникнення помилки для подальшого відстеження.
- fail-fast - зупинення валідації при першій помилці.
- short-circuit - продовження роботи середовища виконання до моменту виявлення помилок.
- Accumulate - накопичення всіх помилок для подальшого виведення їх списком.
- DSL - надання зручної мови для оголошення валідаційних правил, що дозволить описувати валідації в природному для Scala стилі.

- Вибір середовища виконання - гнучкість у виборі середовища для запуску валідацій – як для синхронного, так і для асинхронного виконання.
- Вибір структури даних - можливість вибору оптимальних структур даних для валідації залежно від потреб конкретного проекту.
- Відсутність залежностей - бібліотека не повинна залежати від сторонніх бібліотек, що забезпечить легкість інтеграції.
- Документація - докладна документація, що містить приклади використання, налаштування та розширення функціоналу.
- Приклади - наявність широкого спектру прикладів застосування бібліотеки.
- Бенчмарки - достовірні бенчмарки бібліотеки, що демонструють її ефективні сторони в порівнянні з аналогами
- Open-source - бібліотека має бути відкрита для спільноти та мати можливість підтримки та розвитку.
- Канали зв'язку та спільнота - наявність активної спільноти та канали для швидкої підтримки.
- Доступність - бібліотека повинна бути доступною для широкого кола розробників, із можливістю легкої інтеграції в існуючі проекти.
- Статті та ресурси - наявність додаткових матеріалів, таких як статті, туторіали, що допоможуть новим користувачам швидко освоїти бібліотеку.
- Unit тести - бібліотека має бути покрита Unit тестами, для організації CI/CD та в якості додаткових прикладів для користувачів

ВИСНОВКИ ДО РОЗДІЛУ 1

Проаналізовано різноманітні існуючі рішення для валідації даних, які включають такі фреймворки та бібліотеки як: dupin, accord, octopus, Cats Validated та Scala Either. Було детально розглянуто, що кожна з них представляє та проведено аналіз переваг та недоліків кожної з них.

В розділі було описано, що існують як високорівневі інструменти з зручним DSL API для зрозумілого для людини опису валідацій, так само як і низькорівненві конструкції для опису валідацій з нуля з повним контролем над всім, що відбувається в процесі валідації.

Було розглянуто вплив спільноти на якість, популярність та життєздатність бібліотек, а також було визначено чимало технічних аспектів, про які кінцевому користувачу не потрібно задумуватись, але вони обов’язково повинні бути враховані розробниками бібліотеки.

Тож на основі проведеного аналізу було виявлено ряд властивостей та ознак, які потрібно буде якісно та ефективно врахувати при розробці нашої власної бібліотеки та методу для валідації комплексних даних. Таким чином, актуальною є задача розробки ПЗ для реалізації методу комплексної валідації даних у вигляді бібліотеки для мови програмування Scala на основі визначених вимог.

РОЗДІЛ 2
ПРОЕКТУВАННЯ БІБЛІОТЕКИ ДЛЯ КОМПЛЕКСНОЇ ВАЛІДАЦІЇ ДАНИХ

2.1 Підхід до валідаціх даних та супутні визначення

Валідація даних – це процес підтвердження, що певні данні відповідають очікуванням КС, при цьому валідація даних відбувається в декілька етапів оскільки спочатку дані надані в тому чи іншому вигляді потрібно зчитати згідно заданого формату і перетворити в сутності, які відображають цю модель у вигляді об’єкту в випадку з мовою програмування Scala цей об’єкт є типізованим відповідно спершу при розробці створюється клас, який уособлює модель даних, наприклад можемо створити сутність для обробки реєстрації користувача на якомусь сайті (рис. 2.1.)

```
case class RegisterRequest(
    username: String,
    email: String,
    password: String,
    passwordRepeat: String,
    age: Int,
)
```

Рис. 2.1. Декларація класу форми реєстрації

Припустимо, що цей запит буде отримувати сервер нашої КС і дані будуть приходити в JSON форматі, тоді формально першим етапом валідації буде власне парсинг JSON даних та перетворення їх на екземпляр класу, для цього вже існує купа готових рішень, які автоматично генерують серіалізатори та десеріалізатори для довільних структур даних, тому не будемо зациклювати увагу на цьому. Після отримання екземпляру класу можна бути впевненими лише в тому, що поля моделі мають правильні типи даних, проте це ще не означає, що дані дійсно валідні, ось саме тут і з’являється потреба у формальній валідації, що дані відповідають вимогам системи, насправді можна вважати, що валідація напряму залежить від бізнес логіки і вона скоріш за все дуже специфічна. В поточному прикладі все доволі просто, припустімо, що для початку потрібно провести базову валідацію, таку як:

- Ім’я користувача має бути певної довжини
- Пошта повина бути в правильному форматі
- Пароль має бути певної довжини включати певні види символів
- Повтор пароля має дорівнювати першому його вводу
- Користувач має бути повнолітнім та вік має бути в рамках допустимого

Здавалось би в такому простому запиті криється така кількість вимог до даних і це навіть не все, що нам варто було б перевірити, крім цих базових перевірок, які можна зробити досить швидко і просто, нам би варто було перевірити, що такого користувача в нашій КС ще не було, можливо варто перевірити, що такий емейл взагалі існує, тут варто звернути увагу на те, що щоб виконати ці перевірки потрібно скоріш за все виконувати ресурсоємкі операції – сходити в БД або в іншу систему і т.д. Ці операції об’єднує одна спільна властивість вони зазвичай асинхронні за природою, тому що не можна дозволити процесору простоювати в очікуванні ресурсоємної операції, саме тому такі операції виділяються в окрему асинхронну задачу і ось уже валідація в застосунку має включати не тільки прості синхроні кроки, а й доволі комплексні складні.

Що ж припустімо попередні кроки вже зробили тепер питання полягає в тому що коли валідуються дані чи є потреба щоб при першій же невдалій спробі було повернуто користувачу відповідь про те, що якесь з полів неправильне і не перевіряти дані далі чи потрібно перевірити все що можливо і віддати користувачеві інформацію про всі невірно заповнені дані – це напряму впливає на те яку потрібно використати валідаційну структуру.

Ось вже складено набір правил валідації, обрано валідаційну структуру і лишився один останній нюанс – який саме вигляд та формат має бути у помилки валідації, що виникає. Можливо в одній системі потрібно локалізовані помилки, можливо десь достатньо коду помилки, або звичайного тексту помилки, десь буде бажання автоматично відображати інформацію про помилки саме на тих

полях, які не відповідають вимогам. Знову виникає дилема, тому що кожна система унікальна і має свої вимоги та архітектуру.

З цього випливає, що процес валідації доволі складний і що буде досить важко задовольнити всіх користувачів бібліотеки, проте все ж це можливо реалізувати. На щастя в розрізі мови програмування Scala вийшло знайти рішення, яке задовольнило б усіх користувачів бібліотеки, проте тут не обійшлось без використання великої кількості підходів та архітектурних рішень, які буде розглянуто в подальших розділах.

2.2 Вибір технологій бібліотеки fields

Для написання бібліотеки було обрано мову програмування Scala [3], це доволі просунута мова в плані функціональності, яка в ній є і наразі найновіша версія це Scala 3, проте велика кількість проектів досі використовує Scala 2, адже ще не всі встигли чи знайшли ресурси для переходу на відносно новий мажорний реліз саме тому автори бібліотек заради того, щоб більше людей могли скористатись їх бібліотекою використовують підхід кросс-компіляції, коли одні і ті ж source файли компілюються різними версіями мови програмування, а в місцях де можуть виникати розбіжності створюють окремі модулі специфічні для тої чи іншої версії мови, таким чином публікуються окремі артефакти для кожного несумісного релізу і ця бібліотека не виключення [11].

В якості редактору коду використовується редактор IntelliJ IDEA, який чудово інтегрований з мовою програмування Scala та усіма супутніми інструментами розробки.

В якості системи збірки проекту буде використовуватись SBT або Simple Built Tool [2] – цей збірник дає змогу описувати структуру проекту і його властивостей на тій же Scala і за рахунок цього можна дописувати найдивовижніші речі, які тільки можуть знадобитись, є велика купа плагінів і додатків і доволі потужний встроєний функціонал.

Серед плагінів, які буде використовувати бібліотека:

- sbt-ide-settings – плагін автоматично підтягує значення з sbt в IDE користувача
- sbt-unidoc – плагін генерації scaladoc/Javadoc для проекту
- sbt-ci-release – плагін, що відповідає за реліз стратегію проекту
- sbt-scalafmt – плагін, що відповідає за форматування проекту
- sbt-mdoc – плагін, що дає змогу створювати статичний сайт з документацією проекту
- sbt-buildinfo – плагін, що автоматично генерує інформацію з sbt для використання в момент виконання коду, наприклад версія додатку, назва і т.д
- sbt-header – плагін для генерації заголовку soure файлів таких як copyright та ліцензування
- sbt-tpolecat – плагін, що вмикає набір сталих та налагоджених часом опцій компілятора, що вважається ознакою якісного коду
- sbt-api-mappings – плагін, що дозволяє scaladoc/Javadoc зсилатись на зовніші залежності
- sbt-github-actions – плагін, що переносить опис автоматизації проекту в sbt

В рамках бібліотеки основна задача щоб core модуль не мав залежностей крім самої Scala тому серед бібліотек в нас тільки MUnit - бібліотека для написання тестів. З інших бібліотек варто виділити zio та cats, які знаходяться в своїх окремих модулях, ці бібліотеки реалізують свої асинхронні середовища виконання, а в випадку cats там ще є свої тайп класи для середовищ виконання за рахунок цього бібліотека охоплює величезний спектр середовищ виконання.

Також є залежність на бібліотеку circe для парсингу JSON, адже є окремий модуль для інтеграції з нею і встроюванням валідації на етапі парсингу JSON-у сутності.

В якості систему контролю версій VCS використовується Github, він же нам дає можливість ведення обліку задач, багів, запитів на створення

функціональності та реалізацію CI/CD підходу в бібліотеці, коли достатньо зробити коміт для того, щоб бібліотека автоматично зарелізилась і відбулися всі потрібні операції. Документація та Scaladoc/Javadoc також зберігаються на github pages і автоматично публікуються з кожним новим релізом.

Документація написана за рахунок комбінації MDoc [14] плагіну та фреймворку для написання документації Docusaurus [17], що базується на фреймворці для розробки клієнтських застосунків React [16]. За рахунок MDoc можна описувати приклади коду, які автоматично компілюються і наглядно демонструють роботу бібліотеки користувачеві. Також використовується бібліотека Algolia для реалізації пошуку по документації, вона створює індекс всіх сторінок, заголовків, тексту і дає можливість шукати по ключовим словам сторінки в рамках сайту документації.

2.3 Проектування, архітектура та підходи бібліотеки fields

В рамках наступних розділів потрібно визначитись з рядом підходів для вирішення поставлених задач, оскільки задачі непрості, то і рішення доведеться шукати елегантні та максимально використовувати всі можливості мови програмування задля досягнення мети.

2.3.1 Фундаментальні визначення бібліотеки fields

Перш за все нам треба визначити клас відповідальний за збереження інформації про місце виникнення помилки FieldPath по суті він буде містити колекцію частинок поля FieldPart в якому окрмі назви поля, міститься інформація про тип шляху: поле, ключ хешмапи чи індекс масиву, тому буде доцільно описати його як інтерфейс з трьома реалізаціями, щоб мати змогу в майбутньому робити специфічні перетворення над шляхом, також оскільки FieldPath це просто колекція хотілося б уникнути додаткових алокацій тому скористаємось функціональністю мови opaque типами. Врешті потрібно об’єднати зберігання

шляху з значенням, яке валідується для цього створено класс Field. На основі цих тверджень реалізація буде виглядати так (рис 2.2).

```
sealed abstract class FieldPart(val name: String)
object FieldPart {
  case class Key(value: String)  extends FieldPart(value)
  case class Path(value: String) extends FieldPart(value)
  case class Index(value: Int)   extends FieldPart(value.toString)
}
final case class FieldPath(parts: List[FieldPart]) extends AnyVal
final case class Field[+P](path: FieldPath, value: P)
```

Рис. 2.2. Фундаментальні визначення бібліотеки

Це лише базове визначення цих класів оскільки ще доведеться створити чимало допоміжних методів для зручності користувача, які буде розглянуто згодом. Проте вже на цьому етапі можна декларувати поля разом з їх значеннями хоча поки і не дуже зручно, проте це буде покращено згодом (рис. 2.3.).

```
Field(FieldPath(FieldPart.Path("a"), FieldPart.Index(0), FieldPart.Key("asd")), 1)
```

Рис. 2.3. Оголошення поля

Як видно з рисунку оголошені таким чином поля згодом можна аналізувати, адже шлях поля це не просто текст, а повноцінний опис типів валідованих полів.

2.3.2 Взаємодія бібліотеки з сторонніми модулями

Розглянувши приклад з пункту 2.1 приклад стає зрозуміло, що потрібно певним чином мати змогу нашій бібліотеці взаємодіяти з середовищами виконання, валідаційними структурами чи помилками про які нам невідомо нічого, тим паче треба передбачити можливість, щоб нашу бібліотеку можна було використовувати з уже існуючими рішенням в кодову базу яких не має змоги втручатись. Що ж перше, що приходить в голову це скористатись властивостями поліморфізму і оголосити базовий клас, який описує все що можна робити з валідаційною структурою, середовищем виконання чи помилкою, проте навіть якщо це і спрацює якщо якимось чином домовитись з авторами інших КС віднаслідуватись від інтерфейсів розробленої бібліотеки, що малоймовірно, це не дуже зручно, бо не можна адаптувати існуючі рішення.

Проте можна було б скористатись шаблоном проектування адаптер для рішення цієї проблеми і реалізувати імплементації наших інтерфейсів, недоліком цього було б те що на кожну операцію довелось би створювати екземпляри класів адаптеру ну і те що це виглядало б доволі громіздко в коді.

Для валідаційної бібліотеки дуже важливо мати простий і зрозумілий синтаксис в якому не потрібно думати занадто багато про те які треба адаптери імплементувати і як це правильно робити, на щастя в мові програмування Scala є досить багато функціоналу, який дає можливість дуже зручно працювати з таким явищем як type class. Type class - це конструкція системи типів мови програмування, що дає змогу реалізувати ad-hoc поліморфізм. Для того щоб оголосити type class в мові програмування скала достатньо оголосити інтерфейс – він оголошується за допомогою ключового слова trait також в цьому інтерфейсі потрібно потрібно оголосити generic type parameter який уособлює собою можливість того, що імплементувати цей інтерфейс можна для будь чого, порівняльний приклад реалізації поліморфізму через наслідування та за допомогою type class можна на прикладі серіалізації (рис. 2.4)

```
// Interface
trait Serializable { def serialize: String }
case class Person(name : String, age : Int) extends Serializable {
  override def serialize = s"Person($name, $age)"
}

val person = Person(“Pavel”, 25)
person.serialize
//Type class
trait Serializer[T] { def process(t: T) : String }
def serialize[T](t : T)(implicit s : Serializer[T]) = s.process(t)

implicit object PersonSerializer extends Serializer {
  def process(p : Person) = s"Person($name, $age)"
}

serialize(person)
```

Рис. 2.4. Порівняння class та type class реалізацій поліморфізму

Як видно з прикладу оголошується type class Serializer і маємо дженерік тип T і оголошуємо метод process який приймає як аргумент екземпляр типу T і повертає строку, нижче приклад реалізації тайпкласу та його використання, в цьому прикладі використовується ще одна властивість мови програмування Scala а саме implicit або неявні параметри.

Неявні параметри дуже широко використовуються в мові програмування Scala тому дуже важливо пояснити їх суть перш ніж заглиблюватись в деталі реалізації бібліотеки, це працює таким чином, що можна оголошувати неявні параметри певного типу та очікувати неявні параметри в методах, аргументах функцій, класів та багато де ще, на основні неявних класів в скалі реалізовано приведення типів і є змога створювати свої, також можна робити extension методи, тобто оголошувати якісь методи для певних об'єктів, навіть для тих що уже скомпілювані таким чином можна створювати зручні DSL та підвищувати якість коду та зручність його перегляду.

Тобто якщо повернутись до попереднього прикладу можна реалізувати extension метод для type class-у(рис. 2.5.) і код буде виглядати ідентично до прикладу з наслідуванням, таким чином буде об'єднано всі плюси обох підходів.

```
implicit class SerializerOps[T](private val t: T) extends AnyVal {
  def serialize(implicit s: Serializer[T]): String = s.process(t)
}
```

Рис. 2.5. Приклад extension методу для тайп класу

Важливий момент завдяки компілятору виклики до методу serialize не матимуть накладного розходу ініціалізації класу SerializerOps – компілятор достатньо розумний, щоб зрозуміти, що цього не треба робити і достатньо заінлайнити виклики extension методів.

2.3.3 Абстрагування середовища виконання та валідаційної структури

Тепер визначивши спосіб, яким чином бібліотека буде взаємодіяти зі стороніми КС, наступним кроком потрібно розібратись як взаємодіяти і працювати з різними середовищами виконання та валідаційними структурами. Потрібна якась абстракція, за допомогою якої будуть будуватись різноманітні валідаційні правила при цьому не придаючи уваги, яка буде реальна імплементація.

2.3.3.1 Абстракція Effect для сердовища виконання

Середовище виконання [12] може бути не тільки синхронне виконання команд, а й асинхронне чи мемоізоване, чи ліниве, чи кешоване, можливо там є трасування дій або якесь логування чи контекст користувача, тобто варіантів середовища виконання може бути дуже багато адже всі КС не схожі один на одного і нам потрібно забезпечити потреби у всіх цих випадках, щоб задовольнити всіх користувачів бібліотеки, перше, що спадає на думку це перевикористати те що було розглянуто в попередньому розділі, а саме тайп класи, можна створити тайп клас, який би описував всі можливі дії середовища виконання і на цьому побудувати нашу бібліотеку – і це справді вдала ідея, проте тут є певні нюанси, адже в випадку з середовищем виконання нас цікавить не тільки саме середовище надалі його будемо позначати як F, а й результат виконання цього середовища, на щастя для цього випадку в мові програмування Scala є два рішення, які можна застосувати в цьому випадку.

Спершу потрібно якимось чином описати тип вищого порядку, можна провести аналогію з коробкою, що має якийсь вміст, незалежно від того, що лежить в коробці вона поводиться завжди однаково, аналогічно має поводитись і F для того щоб це зробити можна скористатись таким поняттям як Tagless Final [7] тому створюється тайп класс, який матиме таке визначення (рис. 2.6):

```
trait Effect[F[_]] {}
```

Рис. 2.6. Приклад Tagless Final тайп класу

Проте, що ж має робити цей інтерфейс, які потрібно зробити дії для того, щоб працювати з різними видами середовищ виконання, відповідь на це – це монади. Якщо слідувати класичному визначенню з теорії категорій, то монада – це лише моноїд в категорії ендофункторів, проте це визначення досить складно осягнути лише з цього твердження, якщо у людини немає глибоких знань в теорії категорій, на щастя це можна описати набагато протіше, якщо поглянути з практичної точки зору. Монада – це певна структура, що об’єднує фрагменти нашого коду, а точніше функції, і огортає результат їх виконання в ту ж монаду

з певними додатковими властивостями тієї чи іншої монади. Зазвичай монади визначають набір базових методів поверх яких можна будувати більш складні будь якої складності, зазвичай це метод, щоб огортати будь-яке значення в монаду, та метод, що дає змогу об'єднувати функції результат яких повертає монаду. Простими словами за допомогою монади можна абстрагувати середовище виконання і користувач сам зможе обрати, яким воно буде, які в нього будуть властивості і за рахунок цього в бібліотеці не треба буде піклуватись про те що в результаті обере користувач, будь що буде працювати так як треба. Ключовими методами монади є:

- pure - огортання будь-чого в монаду
- suspend – ліниве огортання будь-чого в монаду
- defer – ліниве огортання монади в монаду
- flatMap – перетворення вмісту монади за допомогою функції, що повертає іншу монаду
- map - перетворення вмісту монади за допомогою функції, що повертає нове значення

То ж визначення середовища виконання може виглядати так (рис 2.7)

```
trait Effect[F[_]] {
  def pure[A](a: A): F[A]
  def suspend[A](a: => A): F[A]
  def defer[A](a: => F[A]): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

Рис. 2.7. Визначення тайп классу, що описує середовище виконання

Маючи такий тайп клас є змога працювати з будь-яким середовищем виконання і будувати валідаційні правила в бібліотеці на основі цих п'яти базових методів.

2.3.4 Абстракція Validated для валідаційної структури

Валідаційна структура чимось схожа на середовище виконання – це така ж коробка щодо якої нам цікаво, що в ній лежить та як ця коробка працює, те що було досліджено в попередньому пункті достатньо для реалізації тайп класу для

валідаційної структуру, по аналогії з середовищем виконанням буде використано типи вищого порядку, tagless final та монадами.

Валідаційну структуру надалі буде позначена як V[_]можна описати тайп класом Validated (рис. 2.8.) який має основні методи:

- strategy – валідаційна стратегія, яка може бути або накомичувальна або fail-fast тобто яка при першій помилці припиняє обробку подальших правил, це поле додається, щоб у подальшому при роботі з валідаційними правилами правильно працювати з валідаційною структурою в рамках середовища виконання
- valid – валідаційна структура з успішним результатом виконання
- invalid – валідаційна структура з помилкою виконання
- isValid – перевірити чи валідаційна структура валідна
- and – об’єднати дві валідаційні структури логічним I

Решту методів можна вивести з цих чотирьох і їх всі буде розглянуто в подальших розділах роботи.

```
sealed trait ValidatedStrategy
case object AccumulateStrategy extends ValidatedStrategy
case object FailFastStrategy extends ValidatedStrategy

trait Validated[V[_]] {
  def strategy: ValidatedStrategy
  def valid[E]: V[E]
  def invalid[E](e: E): V[E]
  def isValid[E](v: V[E]): Boolean
  def and[E](va: V[E], vb: V[E]): V[E]
}
```

Рис. 2.8. Визначення тайп классу для валідаційної структури

Для зручності користувача в бібліотеці вже є вбудовані реалізації Accumulate та FailFast валідаційних структур, які дуже сильно оптимізовані для найкращого перфомансу та мають всі потрібні методи, які можуть знадобитись при валідації даних.

2.3.5 Теговані типи та композиція розроблених абстракцій Rule

Врешті-решт було розроблено підхід для роботи з середовищем виконання F[_] та валідаційної структури даних V[_] і тепер їх можна об’єднати разом для

створення базової абстракції для опису валідаційних правил Rule. Насправді все доволі просто валідаційне правиле це по суті композиція F[_], V[_] та типу помилки E, яку обрав користувач бібліотеки тобто по суті правило являє собою F[V[E]] і на цьому можна було б уже і зупинитись проте якщо лишити все як є виникають певні нюанси.

Оскільки на верхньому рівні знаходиться середовище виконання то виникає питання як в коді розрізняти валідаційну логіку від іншого коду застосунку, можуть виникати неприємні ситуації, де користувач буде плутати бізнес логіку з валідаційною, або робитиме щось небажане з валідаційними правилами, що призводитиме до незручних моментів і потенційних багів в КС. Саме тому слід певним чином приховати той факт, що валідаційне правило це саме композиція інших абстракцій і щоб коли користувач бібліотеки виконував валідацію, отримував певний бібліотечний тип Rule.

Один з варіантів зробити це - можна створити допоміжний клас в який огортати наші абстракції. Проте тоді на кожен раз коли буде виконуватись валідація виникатимуть зайві алокації об’єктів чого хотілось би уникнути адже валідація буде дуже часто виконуватись і в ідеальному варінті це має бути швидко і не використовувати занадто багато ресурсів КС, на щастя мова програмування Scala має дуже потужну систему типів і має все потрібне для створення opaque типів, тобто типів, що на етапі компіляції виглядають ніби це окремий тип, проте в runtime це буде той самий тип, що приховується, таким чином можна реалізувати інкапсуляцію логіки і при цьому не буде накладних розходів при виконанні коду.

Для реалізації цього можна скористатись підходом під назвою теговані типи, суть полягає в тому щоб створити інтерфейс тегу та ним протегувати/підмішати до оголошення нового типу за рахунок цього на етапі компіляції користувач буде бачити тип даних Rule і за потреби зможе конвертувати туди і в зворотньому напрямку валідаційні правила, що додасть додатковий шар інкапсуляції і спростить взаємодію з бібліотекою (рис. 2.9.)

```
object Rule {
  trait Tag[+F[_], +V[_], +E] extends Any
  type Base[+F[_], +V[_], +E] = Any { type __Rule__ }
  type Type[+F[_], +V[_], +E] <: Base[F, V, E] with Tag[F, V, E]

  def wrap[F[_], V[_], E](effect: F[V[E]]): Rule[F, V, E] =
    effect.asInstanceOf[Rule[F, V, E]]

  def unwrap[F[_], V[_], E](rule: Rule[F, V, E]): F[V[E]] =
    rule.asInstanceOf[F[V[E]]]
}
```

Рис. 2.9. Тегований тип для абстракції Rule

Виглядає це досить магічно, але насправді для всього є пояснення: створено тег та базовий тип який є екземпляром типу Any, щоб не виникло зайвих алокацій після чого оголошується власне тип правила як базовий тип, що наслідує оголошений тег, таким чином на етапі компіляції буде збережено всю інформацію про те якого типу F[_], V[_] та E. Додатково оголошено методи wrap та unwrap для перетворення в правила та назад, робиться це за рахунок приведення типів, проте в рантаймі нічого з цього не буде виконуватись адже компілятор достатньо розумний, щоб просто в усіх місцях де було Rule[F, V, E] підставити F[V[E]] під час виконання коду.

Крім того що було створено сам клас Rule де згодом буде оголошено ряд допоміжних методів для роботи з ним тобто його DSL проте важливо наголосити на двох фундаментальних визначенях Rule - це методи об’єднання правил and та or, вони мають працювати аналогічно логічним операторів для булевих виразів, проте тут є один цікавий аспект метод and має працювати по-різному для різних типів валідаційних структур, а саме для accumulate стратегії він буде виконувати всі вирази середовища виконання та об’єднувати валідаційні структури, проте для fail-fast стратегії потрібно спочатку запустити перший вираз та переконатись можливо результат його виконання вже неправильним в такому разі можна не продовжувати виконання та зупинитись на цьому (рис. 2.10)

```
def and[F[_], V[_], E](ra: Rule[F, V, E], rb: Rule[F, V, E])(implicit : Effect[F], V: Validated[V]): Rule[F, V, E] = Rule {
  V.strategy match {
    case AccumulateStrategy => F.flatMap(ra.unwrap)(aa => F.map(rb.unwrap)(bb => V.and(aa, bb)))
    case FailFastStrategy   => F.flatMap(ra.unwrap)(aa => if (V.isInvalid(aa)) F.pure(aa) else rb.unwrap)
  }
}
def or[F[_], V[_], E](ra: Rule[F, V, E], rb: Rule[F, V, E])(implicit F: Effect[F], V: Validated[V]): Rule[F, V, E] = Rule {
  F.flatMap(ra.unwrap) { aa =>
    if (V.isValid(aa)) F.pure(aa)
    else F.map(rb.unwrap)(bb => V.or(bb, aa))
  }
}
```

Рис. 2.10. Реалазція and/or методів класу Rule

На цьому прикладі коду гарно видно, що на етапі компіляції компілятор не дає використати правило як абстракцію середовища виконання та навпаки і потрібно вручну загортати та розгортати правило для цього.

2.3.6 Метапрограмування та макроси для зручності створення Field

Один з викликів покладених в основу бібліотеки – це вміти розрізняти місце або поле для якого виникає та чи інша помилка, це можна було б вирішити прямолінійно, якщо примушувати користувача бібліотеки самостійно для кожного валідаційного правила вказувати назву поля, яке перевіряється і це працювало б, проте було б не дуже зручно і в результаті бібліотека мала б не дуже гарний DSL. Хотілося б мати таке рішення, щоб назва поля автоматично підтягувалась при валідації і можна навіть помітити цікавий момент, що коли описується валідація того чи іншого поля моделі зазвичай потрібно звернутись по назві вкладеного поля в моделі, щоб дістати його значення, тобто як мінімум, щоб доступатись до вкладених значень уже потрібно вказати ім’я і було б чудово, якби можна було б певним чином доступатись до тексту назв полів цього звернення і з нього будувати шлях валідованого поля.

На щастя мова програмування Scala славиться своїм значним вкладом в розвиток та використання підходу метапрограмування та своїми макросами. Метапрограмування – це процес в програмуванні пов’язаний з створенням програмного коду на етапі компіляції, який генерується згідно побажань розробника на основі метаданих про наявні методи, дані, поля, назви класів і так далі, аналізуючи програмний код генерується новий код на етапі компіляції, що

дозволяє вирішувати широкий спектр проблем чи рутинних задач без кодогенерації. Все відбувається безшовно прямо на етапі компіляції без втручань в процес збірки проекту чи в компілятор. На відміну від рефлексії, яка працює на етапі виконання коду і аналізує метадані на льоту метапрограмування працює на етапі компіляції і за рахунок цього немає накладних розходів під час виконання програми плюсом до цього можна створювати синтаксичні надстройки поверх вже існуючих речей в мові програмування за рахунок цього можна реалізувати дуже зручний DSL для бібліотеки.

Написання макросів було досить довгий період часу в Scala нішовою галузю оскільки було погано задокументовано і щоб розібратись в усіх цих тонкощах потрібно було бути як кажуть в спільноті – «чарівником» проте це змінилось в новій мажорній версії Scala 3 і ситуація значно покращилась, люди почали активно впроваджувати метапрограмування не лише в бібліотеках, а й бізнес коді своїх продуктів. Оскільки Scala 2 та Scala 3 досі активно використовуються доведеться написати макрос двічі в форматі обох релізів.

В розділі 2.1.1 було створено клас Field та визначено фундаментальні класи бібліотеки наразі ними не дуже зручно користуватись адже користувачеві потрібно вручну задавати назву полів, є потреба це покращити. Тому буде створено макрос для Scala 3, а макрос для Scala 2 можна буде подивитись в лістингу коду, для повноти. Для зручності було оголошено 3 методи один на самому полі з назвою sub і два на класі Field з назвами sub та from. В усіх цих методах передається функція selector або сам вираз в якому доступаємось до вкладених полів поля, відрізнятимуться вони лише тим, що sub на екземплярі поля буде додавати до уже існуючого шляу поля вкладений шлях виразу, а Field.sub буде діставати весь шлях крім першого виразу, якщо ж треба включити і перший вираз selector-у то можна скористатись методом Field.from. Для написання макросів потрібно оголосити методи як inline і всі аргументи, функцій, AST дерева яких будуть аналізуватись також. Далі в силу вступає власне написання самого макросу. В розпоряджені розробників макросів є splicing та quoting механізми, як перетворюють AST дерева виразів в реальний

код та назад, перетворюючи вирази в AST дерева є можливість аналізувати властивості коду, що передав користувач і робити потрібні перетворення. В розглянутому випадку потрібно пройтись по AST дереву і рекурсивно прямувати вглиб нього поки зустрічаються вирази, що являються собою звернення до полів екземпляру, в разі якщо зустрічається щось не підходяще можна видавати помилку компіляції з детальними інструкціями для користувача, щоб він знав як користуватись бібліотекою і покращити його UX.

Оскільки в усіх трьох методах закладена майже аналогічна логіка є сенс створити допоміжний клас SelectorMacro, який буде включати логіку розбору синтаксичного дерева виразу (рис. 2.11)

```
class SelectorMacro[Q <: Quotes](using val q: Q) {
  import q.reflect._
  given FieldPartToExpr: ToExpr[FieldPart] with {
    def apply(part: FieldPart)(using Quotes) =
      part match {
        case Key(value)   => '{ Key(${ Expr(value) }) }
        case Path(value)  => '{ Path(${ Expr(value) }) }
        case Index(value) => '{ Index(${ Expr(value) }) }
      }
  }
  def selectorPath(term: Term, includeIdent: Boolean, title: String): Expr[List[FieldPart]] = {
    @tailrec
    def go(term: Term, acc: List[FieldPart] = Nil): List[FieldPart] =
      term match {
        case Ident(name)           => if (includeIdent) Path(name) :: acc else acc
        case Select(This(_), name) => if (includeIdent) Path(name) :: acc else acc
        case Select(rest, name)    => go(rest, Path(name.toString) :: acc)
        case Inlined(_, _, rest)   => go(rest, acc)
        case Apply(Select(rest, "apply"), List(Literal(IntConstant(index))))  => go(rest, Index(index) :: acc)
        case Apply(Select(rest, "apply"), List(Literal(StringConstant(key)))) => go(rest, Key(key) :: acc)
        case unmatched => report.errorAndAbort(FieldMacroMessage.selectorErrorMessage(title))
      }

    Expr(go(term))
  }
}
```

Рис. 2.11. Утилітарний клас SelectorMacro

Як видно з фрагменту коду аналізується синтаксичне дерево та витягаються назви полів до яких звертаються в раніше оголошений клас FieldPart. Далі буде розглянуто лише реалізацію для Field.from та Field.sub адже всі 3 доволі схожі між собою (рис. 2.12)

```
object FieldMacro {
  def fromImpl[P: Type](value: Expr[P], includeIdent: Boolean, title: String)(using q: Quotes) = {
    import quotes.reflect._
    // report.info(value.asTerm.show(using Printer.TreeStructure))
    val parts = new SelectorMacro[q.type].selectorPath(value.asTerm, includeIdent, title)
    '{
      Field(
        path = FieldPath(${ parts }),
        value = $value,
      )
    }
  }
}

extension (field: Field.type) {
  inline def from[V](inline value: V): Field[V] = ${
    FieldMacro.fromImpl[V]('value, includeIdent = true, title = "Field.from")
  }

  inline def sub[V](inline value: V): Field[V] = ${
    FieldMacro.fromImpl[V]('value, includeIdent = false, title = "Field.sub")
  }
}
```

Рис. 2.12. Реалізація Field.from та Field.sub

З рисунку видно що в обох методах перевикористовується допоміжний клас SelectorMacro, який аналізує AST дерево виразу і дає шлях викликів, який згодом використовується для створення екземпляру поля з вказаним шляхом і розширюється клас Field додатковими методами, які викликають написаний макрос, за рахунок цього тепер можна дуже легко конструювати поля (рис. 2.13.).

```
case class Request(name: String)
val request = Request("Ann")
Field.from(request.name) // -> request.name
Field.sub(request.name) // -> name
Field(request).sub(  .name) // -> name
```

Рис. 2.13. Використання написаних макросів для створення Field

З рисунку видно, що тепер коритувач одразу описує і поле, яке буде валідуватись і одразу ж автоматично отримується його назва за допогою макросу.

2.3.7 Створення екземплярів помилок для встроєних валідаційних правил

Гарна бібліотека для валідації має включати чималий список встроєних правил, проте викликає дилема адже в попередніх розділах було абстраговано тип помилки, якщо про помилку нічого не відомо, то як можна створювати екземпляри помилок для стандартних перевірок, наприклад коли перевіряється, що розмір колекції не має бути більше за якесь значення, для цього можна

перевикористати раніше описаний підхід тайп класів і для кожного такого випадку буде створено свій, зараз же буде розглянуто лише приклад з максимальним розміром (рис. 2.14)

```
trait FailWithMaxSize[E, +P] { def maxSize[PP >: P](size: Int)(field: Field[PP]): E }
```

Рис. 2.14. Тайпклас для створення помилки перевірки максимального розміру колекції

Як видно з рисунку оголошено інтерфейс з одним методом в якому є все потрібне для створення помилки, а саме максимальний розмір та саме поле, що перевіряється.

За рахунок такого підходу згодом можна буде де це потрібно неявним чином просити імплементацію цього інтерфейсу, який користувач реалізує сам, або якщо він використовує встроєні типи помилок, то вони вже будуть реалізовані. Інший великий плюс такого підходу, що якщо користувач ніколи не буде користуватись певними видами стандартних методів перевірки йому не доведеться реалізовувати ті чи інші інтерфейси, адже вони атомарні і їх потрібно реалізовувати лише, якщо користуєшся відповідними методами бібліотеки.

2.3.8 Встроєні валідаційні правила, політика валідації та DSL

Вже пройдено чималий шлях в розробці бібліотеки і ось нарешті прийшов час об’єднати всі раніше придумані підходи та абстракції. Перш за все для зручності користувачів було створено абстракцію ValidationPolicy (рис. 2.15). Вона являє собою функцію, яка приймає як аргумент поле типу P та виконує ряд валідаційних правил Rule за рахунок цієї абстракції можна декларувати правила та одразу інкапсулювати логіку валідації до конкретних типів даних.

```
trait ValidationPolicy[P, F[_], V[_], E] extends (Field[P] => Rule[F, V, E]) { self =>
  def validate(field: Field[P]): Rule[F, V, E]

  def apply(field: Field[P]): Rule[F, V, E] = validate(field)
}
```

Рис. 2.15. Валідаційна політика

Нарешті справа дійшла до створення встроєних валідаційних правил в яких використовуються всі раніше оголошені абстракції. Синтаксис для встроєних правил побудований навколо функціоналу extension методів, який вже був розглянутий в попередніх розділах, тут визначаються generic типи F[_], V[_] та E для того, щоб можна було використовувати встроєну валідацію незалежно від їх типів. На прикладі правил для булевих значень буде найпростіше зрозуміти принцип дії (рис. 2.16)

```
final class BooleanFieldOps[F[_], V[_], E](private val field: Field[Boolean]) extends AnyVal {
  def isTrue(implicit F: Effect[F], V: Validated[V], FW: FailWithCompare[E, Boolean]): Rule[F, V, E] =
    field.ensure(_ == true, _.failEqual(true))

  def isFalse(implicit F: Effect[F], V: Validated[V], FW: FailWithCompare[E, Boolean]): Rule[F, V, E] =
    field.ensure(_ == false, _.failEqual(false))
}
```

Рис. 2.16. Встроєний синтаксис для булевих значень

На прикладі методі isTrue та isFalse видно, що неявно запрошуються реалізації тайп класів Effect, Validated та FailWith*, які в подальшому використовуються для того, щоб реалізувати правила, використовується синтаксис ensure в якому передається умова, коли дані вірні та помилка, яку потрібно видати в разі якщо умова не виконається, аналогічним чином написані і решта встроєних валідаційних правил.

Тепер лишилось оголосити ValidationDsl (рис. 2.17.) модуль з усіма generic параметрами та неявно затянути тайп класи Effect та Validated, щоб користувачеві потрібно було лише раз їх імпортувати, а далі вони автоматично підтягувались під час валідації, клас Dsl наслідується від усіх класів зі встроєними правилами, щоб вони автоматично потрапляли в scope користувача і він міг користуватись ними. Також на цьому класі оголошено ряд скорочень для методів стосовно ValidationPolicy та Rule адже вони в базові імплементації включають аж 3 generic параметри F, V та E за рахунок цього користувачеві не треба буде кожен раз перелічувати їх всі адже вони вже будуть запам’ятовані в рамках DSL модуляю.

```
class ValidationDsl[F[_], V[_], E](implicit
    val F: Effect[F],
    val V: Validated[V],
) extends ModuleAllSyntax[F, V, E]
```

Рис. 2.17. Модуль з усім валідаційним DSL

Як видно з рисунку ValidationDsl об’єднує в собі розглянути в попередніх розділах рішення і є точкою входу для написання валідації

2.3.9 Інтеграція з сторонніми бібліотеками на прикладі circe

Немало важливо розглянути приклад інтеграції бібліотеки з стороньою, щоб продемонструвати, що при розробці бібліотеки валідації комплексних даних дійсно було прийнято вірні архітектурні рішення і її можна легко інтегрувати з іншими програмними модулями. Для прикладу було обрано бібліотеку для роботи з JSON – circe. Вона дає змогу генерувати серіалізатори та десеріалізатори сутностей за допомогою макросів, в певних випадках може виникнути потреба перенести момент виконання валідації ще під час парсингу JSON і це доцільно оскільки circe видає помилки разом з місцем їх виникнення тому можна валідувати сутності за допомогою розробленої бібліотеки і конвертувати в формат помилок circe. Оголошується метод, який буде брати існуючий десериалізатор та додаватиме до нього етап валідації (рис. 2.18)

```
def policyDecoder[T, F[_]: RunSync, V[_]: Validated: HasErrors, E: HasFieldPath](
policy: ValidationPolicy[T, F, V, E], decoder: Decoder[T]) =
  decoder.flatMap { entity =>
    val result = RunSync[F].run(policy.validate(Field(entity)).effect)
    if (result.isValid) Decoder.const(entity)
    else
      new Decoder[T] {
        private val errors: List[E] = result.errors
        private val failures       =
          errors.map(e =>
            DecodingFailure(
              message = e.toString,
              ops = HasFieldPath[E].fieldPath(e).parts.reverse.map {
                case FieldPart.Path(value)  => CursorOp.DownField(value)
                case FieldPart.Index(value) => CursorOp.DownN(value)
              },
            )
          )
        final def apply(c: HCursor): Result[T] = Left(failures.head)
        final override def decodeAccumulating(c: HCursor): AccumulatingResult[T] =
          cats.data.Validated.invalid(NonEmptyList.fromListUnsafe(failures))
      }
}
```

Рис. 2.18. Інтеграція з бібліотекою circe

Як видно з фрагменту коду було розширено бібліотечний тип Decoder, щоб валідувати дані одразу після їх парсингу з JSON за рахунок прийнятих рішень з легкістю вийшло віддавати помилки валідації в форматі з яким працює circe.

2.4 Складові бібліотеки

Бібліотека складається з таких компонент:

- sbt модулі
    - fields-core – core модуль бібліотеки містить всі базові визначення бібліотеки, не має жодної залежності та гарно покритий unit тестами
    - fields-zio – модул містить адаптер/інтеграцію з широко відомою бібліотекою zio для асинхронного програмування [8]
    - fields-cats – модуль містить адаптер/інтеграцію з не менш відомою бібліотекою cats [9], що включає в себе широкий спектр тайп класів, які використовує чимало проектів та бібліотек, що робить інтеграцію бібліотеки з ними гладкою і простою для користувачів. Також це дає змогу інтегруватись з бібліотекою cats-effect [10], яка реалізує IO – абстракцію для асинхронного програмування, аналог ZIO
    - fields-docs – модуль містить компонування scaladoc, документації та сайту документації в одне ціле рішення.
- тека docs – містить документацією бібліотеки в форматі Markdown
- тека website - містить логіку сайту бібліотеки, який агрегує в собі Scaladoc, Javadoc та документацію.

Детальну діаграму класів, компонентів та прецендентів бібліотеки можна переглянути в додатках А, Б та В.

ВИСНОВКИ ДО РОЗДІЛУ 2

В рамках цього розділу було знайдено і обрано ряд підходів при проектування бібліотеки. Зпроектовано і закладено фундаментальні визначення бібліотеки, знайдено архітектурні рішення задля того, щоб задовольнити вимоги поставлені перед бібліотекою. Варто виокремити влучність застосування цілого спектру різноманітних підходів задля досягнення оптимальних рішень в кожному аспекті валідації комплексних даних.

Серед переваг бібліотеки варто звернути увагу на унікальний підхід для досягнення найбільшої швидкодії бібліотеки без зайвих алокацій пам’яті, зручний набір встроєних валідаційних правил і значний спектр застосування та можливостей розширення бібліотеки. Вона здатна підстроюватись під потреби користувача і дуже легко інтегрується з сторонніми бібліотеками за рахунок прийнятих архітектурних рішень.

Найбільшою відмінністю від конкурентів, окрім виправлення недоліків конкурентів та реалізації більшості їх переваг, варто зазначити наявність перехресної валідації, підтримку комплексної валідації даних, можливість обирати валідаційну структуру та реалізація підтримки fail-fast/short-circuit підходів.

Було визначено стек технологій для розробки бібліотеки, її циклу життєдіяльності, документації та підтримки. На основі описаних підходів і результатів проектування було закладено основи для реалізації самої бібліотеки.

Результуюча бібліотека є універсальним інструментом, який можна застовувати будь-де, наприклад у військовій сфері для аналізу даних розвідки для прийняття стратегічних рішень, перевірки достовірності координат і даних геолокації, контролю за правильністю параметрів озброєння та спорядження. У фінансовій для перевірка фінансових транзакцій і попередження шахрайства, валідації даних у системах обліку і аудиту, аналіз кредитоспроможності клієнтів.

РОЗДІЛ 3
ОЦІНЮВАННЯ БІБЛІОТЕКИ ДЛЯ КОМПЛЕКСНОЇ ВАЛІДАЦІЇ ДАНИХ

3.1 Інструкція користувача

Для початку роботи з бібліотекою користувачеві потрібно або створити новий Scala проект або мати вже існуючий. Він заходить на GitHub бібліотеки та починає ознайомлення з нею [18]. Оскільки артефакт бібліотеки розміщений в піблічному артифакторі Sonatype, який автоматично публікується з певним інтервалом в Maven Central артифакторі, який за замовчуванням уже підключений в більшість популярних інструментів для менеджменту залежностей та проектів на JVM платформі, таких як Maven, Gradle, sbt та інші , то достатньо буде затягнути собі в проект потрібні модулі бібліотеки.

Будемо вважати, що користувач використовує sbt та в такому разі він заходить на головну сторінку сайту бібліотеки https://jap-company.github.io/fields/ та бачить всередині інструкцію як почати роботу з нею після чого додає в свій build.sbt файл залежності на всі доступні модулі бібліотеки fields (рис. 3.1)

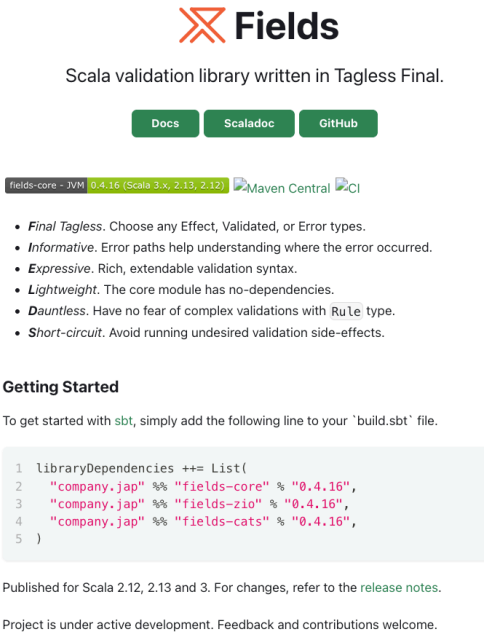


Рис. 3.1. Початок роботи з бібліотекою

Як видно з цього фрагменту коду для кожної залежності вказується організація бібліотеки, яка є інверсією домену бібліотеки, який був завчасно придбаний та налаштований в Sonatype, щоб підвердити право власності над ним, далі слідує префікс назви бібліотеки та власне назва модуля. Завершує все тут версія бібліотеки якою користувач хоче скористатися.

Для початку розглянемо, що потрібно зробити, щоб писати валідацію, користувач переходить на сторінку /docs/overview/ та бачить огляд того як може виглядати базова валідація, якщо у користувача немає бажання нічого кастомізувати і він згоден користуватись синхронним середовищем виконанням, бібліотечною реалізацією помилки, що включає місце її виникнення та повідомлення англійсько мовою причини виникнення помилки. Для такого випадку вже є передвизначені модулі DSL, з уже визначеними тайпкласами для всієї машинерії бібліотеки DefaultFailFastDsl та DefaultAccumulateDsl, які відрізняються між собою лише валідаційною структурою, таким чином користувачеві потрібно імпортувати вміст пакетів jap.fields та jap.fields. DefaultAccumulateDsl або jap.fields.DefaultFailFastDsl.

Для початку йому не потрібно вдаватись в потробиці та нюанси роботи з бібліотекою проте коли прийде час він зможе ознайомитись детальніше з кожною сутністю на відповідних сторінках:

- Валідаційним DSL - /docs/validation-module/
- Валідаційною структурою - /docs/validated/
- Середовищем виконання - /docs/effect/
- Встроєними типами помилок - /docs/validation-error/
- Абстракцією для створення помилок стандартних валідацій - /docs/fail-with/
- Класом валідованого поля Field - /docs/field/
- Класом опису шляху поля - /docs/fieldpath/
- Типом даних Rule - /docs/rule/
- Валідаційною політикою - /docs/validation-policy/

- Інтеграцією з circe - /docs/interop-circe/

Тепер потрібно створити сутності, яку буде валідувати користувач або він вже має готові сутності – це просто набір DTO класів з описом їх вмісту, вони можуть бути пласкі або включати вкладені сутності це неважливо адже всі з цих випадків підтримуються бібліотекою.

Після оголошення моделі можна перейти власне до валідації в базовому випадку пересічному користувачеві не потрібно робити свої унікальні валідації і йому буде достатньо встроєних валідаційних правил, тому він спочатку ознайомлюється з усіма правилами бібліотеки (див. табл. 3.1) на сторінці /docs/validation-syntax/

Таблиця 3.1

Перелік синтаксичних конструкцій бібліотеки

Синтаксис	Опис
Загальний	
when	Виконує валідацію тільки якщо виконується умова
unless	Виконує валідацію тільки якщо умова не виконується
when	Виконує валідацію тільки якщо виконується умова для значення поля
whenF	Виконує валідацію тільки якщо не виконується умова значення поля (асинхронно)
whenType	Виконує правило для вказаних підтипів, інакше є вірним
ensure	Перевіряє умову і повертає помилку, якщо вона не виконується
ensureF	Перевіряє умову асинхронно і повертає помилку, якщо умова не виконується
assert	Перевіряє умову і повертає помилку, якщо умова не виконується
assertF	Перевіряє умову асинхронно і повертає помилку, якщо умова не виконується
check	Повертає результат застосування функції до поля
checkF	Повертає результат застосування асинхронної функції до поля

Таблиця 3.1 (продовження)

Перелік синтаксичних конструкцій бібліотеки

===	Перевіряє на рівність поля
equalTo	Перевіряє на рівність поля
!==	Перевіряє на нерівність поля
notEqualTo	Перевіряє на нерівність поля
in	Значення поля міститься в переліку
all	Комбінує всі валідації за допомогою логічним AND
any	Комбінує всі валідації за допомогою логічного OR
validate	Виконує валідацію за допомогою політики
validateEither	Виконує валідацію за допомогою політики і повертає або перелік помилок або значення поля
Для булевих значень	
isTrue	Значення поля дорівнює true
isFalse	Значення поля дорівнює false
Для колекцій	
isEmpty	Колекція порожня
nonEmpty	Колекція не порожня
minSize	Розмір колекції не менший за min
maxSize	Розмір колекції не більший за max
each	Застосовує check до кожного елемента колекції
eachWithIndex	Застосовує check до кожного елемента колекції з індексом
any	Застосовує check до кожного елемента колекції, будь-який повинен пройти
anyWithIndex	Застосовує check до кожного елемента колекції з індексом, будь-який повинен пройти
isDistinctBy	Колекція має унікальні елементи за властивістю by
isDistinct	Колекція має унікальні елементи

Таблиця 3.1 (продовження)

Перелік синтаксичних конструкцій бібліотеки

Для хеш-мап	
isEmpty	Мапа порожня
nonEmpty	Мапа не порожня
minSize	Розмір мапи не менший за min
maxSize	Розмір мапи не більший за max
each	Застосовує валідацію до кожного елемента мапи
eachKey	Застосовує валідацію до кожного ключа мапи
eachValue	Застосовує валідацію до кожного значення мапи
any	Застосовує валідацію до кожного елемента мапи, будь-який повинен пройти
anyKey	Застосовує валідацію до кожного ключа мапи, будь-який повинен пройти
anyValue	Застосовує валідацію до кожного значення мапи, будь-який повинен пройти
isDistinctBy	Мапа має унікальні елементи за властивістю by
isDistinct	Мапа має унікальні елементи
Для опціональних значень	
isSome	Поле не пусте
isDefined	Поле не пусте
isNone	Поле пусте
isEmpty	Поле пусте
some	Застосовує валідацію до не пустого поля
Для порівнюваних значень	
>=	Значення поля більше або дорівнює порівнюваному
gte	Значення поля більше або дорівнює порівнюваному
>	Значення поля більше порівнюваного
gt	Значення поля більше порівнюваного
<=	Значення поля менше або дорівнює порівнюваному

Таблиця 3.1 (продовження)

Перелік синтаксичних конструкцій бібліотеки

lte	Значення поля менше або дорівнює порівнюваному
<	Значення поля менше порівнюваного
lt	Значення поля менше порівнюваного
isBetween	Значення поля знаходиться в визначених рамках
Для строк	
minSize	Мінімальний розмір рядка дорівнює min
maxSize	Максимальний розмір рядка дорівнює max
matches	Рядок відповідає Java регулярному виразу
matchesRegex	Рядок відповідає Scala регулярному виразу
isEnum	Рядок є частиною Scala Enumeration
isJEnum	Рядок є частиною Java Enum

З представленої таблиці видно, що базові валідаційні конструкції бібліотеки покривають широкий спектр потреб користувачів, і для середньостатистичного користувача буде достатньо DSL для строк, булевих значень, числових/порівнюваних значень, колекцій, мап, опціональних значень та загального DSL, яких дає змогу будувати свої визначення досить просто.

3.1.1 Огляд простого випадку валідації даних

Після ознайомлення з прикладом використання та доступними встроєними валідаційними правилами користувач врешті пише свою першу валідаційну політику (рис. 3.2) та запускає валідацію на невірних даних, щоб побачити, як генеруються помилки разом з їх місцем виникнення

```
import jap.fields._
import jap.fields.DefaultAccumulateVM._

case class User(username: String, password: String, passwordRepeat: Option[String])
case class UserFeatures(standsWithUkraine: Boolean)
case class Request(user: User, features: UserFeatures)
object Request {
  implicit val policy: Policy[Request] =
    Policy
      .builder[Request]
      .subRule(_.user.username)(_.nonBlank, _.minSize(4))
      .subRule(_.user.password)(_.nonBlank, _.minSize(8), _.maxSize(30))
      .subRule(_.user.password, _.user.passwordRepeat)((p, pr) => pr.some(_ === p))
      .rule { request =>
        val standsWithUkraineF = request.sub(_.features.standsWithUkraine)
        standsWithUkraineF.ensure(_ == true, _.failMessage("Please support Ukraine"))
      }
      .build
}

val request  = Request(User("Ann", "1234", Some("")), UserFeatures(false))

Field(request).validate
// Invalid(
//   errors = List(
//     MinSize(path = "user.username, size = 4),
//     MinSize(path = user.password"), size = 8),
//     Equal(path = "user.passwordRepeat", compared = ".user.password"),
//     Message(path = "features.standsWithUkraine", error = " Please support Ukraine ")
//   )
// )
```

Рис. 3.2. Приклад базового використання бібліотеки

На розглянутому прикладі видно, що достатньо двох імпортів модулів з бібліотеки, щоб написати базову валідацію моделі користувача і це виглядає досить природньо і зрозуміло з першого погляду, що валідується.

3.1.2 Огляд випадку комплексної валідації даних

Вже було розглянуто простий випадок валідації даних тепер варто розглянути випадок, коли мова йде про більш досвідченого користувачача, який має бажання все налаштувати під себе і використовує бібліотеку по-максимуму.

В цій емульованій ситуації користувач - це розробник платформи Medium, що вирішив скористатись бібліотекою валідації fields, його зацікавила можливість скористатись готовою бібліотекою для валідації та замінити існуюче самописне рішення компанії тому він вирішує спробувати написати Proof of concept або POC для валідації сутності створення запису в блогі Medium.

Є ряд існуючих вимог до цієї валідації:

- Середовищем виконання слугує абстракція для асинхронного виконання з бібліотеки zio – Task

- В коді компанії використовується уже існуюча акамулююча валідаційна структура MediumErrors, яку потрібно перевикористати
- В компанії прийнято при виникненні помилки валідації передавати клієнтському застосунку місце виникнення помилки разом з числовим кодом помилки – це реалізовано класом MediumErrorCode
- При валідації потрібно виконувати асинхронні виклики до бази даних.

Маємо чималий ряд вимог і тепер спробуємо розглянути кроки розробника Medium. Для початку для існуючого типу помилки розробник реалізує тайпклас FailWith та визначає коди для всіх встроєних помилок (рис 3.3).

```
case class MediumErrorCode(path: FieldPath, code: Int)
object MediumErrorCode {
  def apply(code: Int)(path: FieldPath): MediumErrorCode = MediumErrorCode(path, code)
}

trait CanFailMediumErrorCode {
  implicit val FailWith: FailWith[MediumErrorCode, Nothing] = FailWithMediumErrorCode
}

object FailWithMediumErrorCode extends FailWith.Base[MediumErrorCode] {
  def empty[P](field: Field[P]): MediumErrorCode = MediumErrorCode(field, 1)
  def minSize[P](size: Int)(field: Field[P]): MediumErrorCode = MediumErrorCode(field, 2)
  def invalid[P](field: Field[P]): MediumErrorCode = MediumErrorCode(field, 3)
  def nonEmpty[P](field: Field[P]): MediumErrorCode = MediumErrorCode(field, 4)
  def maxSize[P](size: Int)(field: Field[P]): MediumErrorCode = MediumErrorCode(field, 5)
  def oneOf[P](variants: Seq[P])(field: Field[P]): MediumErrorCode = MediumErrorCode(field, 7)
  def message[P](error: String, message: Option[String])(field: Field[P]): MediumErrorCode = MediumErrorCode(field, 1)
  def compare[P](operation: CompareOperation, compared: String)(field: Field[P]): MediumErrorCode = MediumErrorCode(field, 8)
}
```

Рис. 3.3. Реалізація FailWith для помилки застосунку

Після чого йому потрібно реалізувати тайпклас Validated для їх валідаційної структури (рис. 3.4)

```
case class MediumErrors[E](errors: List[E]) extends Throwable with NoStackTrace {
  override def toString = errors.mkString("ERRORS:\n\t", "\n\t", "\n")
}
object MediumErrors {
  implicit object MediumErrorsValidated extends AccumulateLike[MediumErrors] {
    def valid[E]: MediumErrors[E]                                  = MediumErrors(Nil)
    def invalid[E](e: E): MediumErrors[E]                          = MediumErrors(e :: Nil)
    def and[E](va: MediumErrors[E], vb: MediumErrors[E]): MediumErrors[E] = MediumErrors(va.errors ++ vb.errors)
    def isValid[E](v: MediumErrors[E]): Boolean                    = v.errors.isEmpty
  }
}
```

Рис. 3.4. Реалізація акамуляційного Validated тайпкласу

Оскільки бібліотека вже має модуль fields-zio то Effect тайпклас реалізовувати немає потреби розробник просто переходить до етапу оголошення валідаційного DSL в якому використовує задані умовами середовище виконання zio.Task,

валідаційну структуру MediumErrors та тип помилки MediumErrorCode та на додачу до цього реалізує свій метод, щоб створювати помилки з специфічними кодами помилок та свій замінює поведінку порівняння полів з типом PostDate, щоб видавати унікальний код для нього (рис 3.5)

```
object MediumValidationDsl extends ValidationDsl[Task, MediumErrors, MediumErrorCode] with CanFailMediumErrorCode
{
 def failCode[P](code: Int)(field: Field[P]) =
   V.invalid(MediumErrorCode(field.path, code))

 implicit object FailPostDateWithErrorCode extends FailWithCompare[MediumErrorCode, PostDate] {
  override def compare[P](operation: CompareOperation, compared: String)(field: Field[P]): MediumErrorCode =
    MediumErrorCode(field.path, 123)
 }
}
```

Рис. 3.5. Реалізація валідаційного DSL

Тепер для вже існуючої моделі запиту створення посту MediumPost розробник спочатку описує прості валідаційні правила, для цього він імпортує все з створеного ним модулю DSL та оголошує валідаційну політику (рис. 3.6)

```
import MediumValidationDsl._

case class PostId(value: Int)          extends AnyVal
case class UserId(value: Int)          extends AnyVal
case class PostDate(value: LocalDateTime) extends AnyVal
object PostDate {
 implicit val policy: Policy[PostDate]    = _ > PostDate(LocalDateTime.now)
 implicit val ordering: Ordering[PostDate] = Ordering.by(_.value)
}
sealed trait ParagraphContent
case class Link(value: String) extends ParagraphContent
case class Text(value: String) extends ParagraphContent

case class MediumPost(
   postId: PostId,
   title: String,
   description: Option[String],
   tags: List[String],
   creationDate: PostDate,
   updateDate: PostDate,
   authorId: UserId,
   paragraphs: Map[String, ParagraphContent],
)
object MediumPost {
 implicit val policy: Policy[MediumPost] =
   Policy
    .builder[MediumPost]
    .fieldRule(_.sub(_.postId).map(_.value))(_ > 0, _ !== Int.MaxValue)
    .fieldRule(_.sub(_.authorId).map(_.value))(_ > 0)
    .subRule(_.title)(_.minSize(6), _.maxSize(255))
    .subRule(_.description)(_.some(_.nonBlank))
    .subRule(_.tags)(_.nonEmpty, _.maxSize(12), _.each(_.all(_.minSize(3), _.notEqualTo("JAVA"))))
    .subRule(_.creationDate, _.updateDate)((c, u) => c < u, (c, u) => c.validate && u.validate)
    .subRule(_.paragraphs)(_.eachKey(_.nonEmpty), _.eachValue(validateParagraphContent))
    .build

 def validateParagraphContent(content: Field[ParagraphContent]): MRule = {
   content.whenType[Link](_.map(_.value).all(_.minSize(10), _.maxSize(60))) &&
   content.whenType[Text](_.map(_.value).all(_.minSize(4), _.maxSize(50)))
 }
}
```

Рис. 3.6. Базова валідація MediumPost

Варто відзначити, що модель цієї сутності доволі складна і для вкладених полів таких як PostDate розробник оголосив окрему політику, щоб можна було її перевикористовувати. Він користуються широким спектром валідаційного DSL для того щоб створити доволі комплексну логіку валідації, проте завдяки зручному синтаксису ця логіка доволі легко читається і розуміється.

Тепер залишився лише пункт щодо взаємодії з базою даних, в коді Medium при валідації створення блогу перевіряється, що ідентифікатор автора існує та що блогу посту з таким айді немає і заголовок статті теж унікальний. Тому розробник створює PostValidationService в якого є залежності на PostRepo та UserRepo, які взаємодіють з базою даних і він врешті реалізує фінальну валідаційну політику при цьому викликає базову політику в парі з взаємодією з БД (рис. 3.7)

```
case class PostValidationService(postRepo: PostRepo, userRepo: UserRepo) {
  implicit val policy: Policy[MediumPost] =
    Policy
      .builder[MediumPost]
      .rule(MediumPost.policy)
      .subRule(_.authorId)(_.ensureF(userRepo.userExists, failCode(33)))
      .subRule(_.postId)(_.ensureF(postRepo.postDoesNotExist, failCode(44)))
      .subRule(_.title)(_.ensureF(postRepo.titleExists(_).negate, failCode(55)))
      .build
}
```

Рис. 3.7. Результуюча валідаційна політика MediumPost

Таким чином розробник компанії Medium успішно реалізував POC використання бібліотеки комплексної валідації даних і компанія перейшла до більш маштабного рефакторингу, щоб повністю перейти на бібліотеку fields.

Звісно ж всі збіги в цій історії випадкові і вона повністю вигадана, щоб продемонструвати, як міг би скористатись бібліотекою пересічний розробник якоїсь компанії та які кроки йому довелось би зробити для того, щоб скористатись по максимум можливостями бібліотеки.

3.2 Огляд інфраструктури та поріг входу у використання fields

В попередньому розділі було розглянуто два приклади використання бібліотеки та детально описано всі потрібні кроки для цих випадків, але ці приклади написані власне автором бібліотеки звісно, що він розуміє що і як в ній

влаштовано. На мою думку найважливіший аспект бібліотеки після самої її реалізації – це її інфраструктура і спільнота тому давайте розглянемо в цьому розділі наскільки вдала вона в бібліотеки для комплексної валідації fields, наскільки зручно буде продовжувати її розробку, якісно оцінимо все за допомогою опитування інших Scala розробників.

3.2.1 Встроєна в код документація - Scaladoc

Дуже важливо при написанні коду документувати деталі реалізації, принципи роботи тих чи інших методів, описувати суть параметрів методів, результат виконання, розповідати нюанси та наводити приклади використання і це стає можливим завдяки Scaladoc [15].

Для того, щоб написати документацію для методу, поля чи змінної достатньо використати спеціальний синтаксис (рис 3.8)

```
/** Returns subfield using `selector` function to extract value and as path
  *
  * Example
  * {{{
  * scala> val request = Request(User("ann"))
  * scala> val field = Field.from(request)
  * val field: jap.fields.Field[Request] = request:Request(User(ann))
  * scala> field.sub(_.user.name)
  * val res1: jap.fields.Field[String] = request.user.name:ann
  * }}}
  */
inline def sub[S](inline selector: P ⇒ S): Field[S] = ${ FieldMacro.subImpl[P, S]('field, 'selector) }
```

Рис. 3.8. Приклад Scaladoc з бібліотеки

Всі декларації в бібліотеці описані аналогічним чином і мають опис того що вони роблять, де це потрібно є приклади використання з описом того що відбувається, це дає можливість користувачу бібліотеки переглядати документацію двома способами – прямо з редактору коду та в згенерованому HTML вигляді з зручним пошуком та приємним користувацьким інтерфейсом.

У першому випадку достатньо просто навести курсором на метод, який цікавить користувача, щоб побачити інформацію про нього (рис 3.9)

Рис. 3.9. Перегляд Scaladoc з IDE

Інший варіант, як користувач може переглядати документацію – це з коду послідувати у визначення методу і IDE перейде до коду бібліотеки де він зможу побачити аналогічний опис (рис. 3.10)

Рис. 3.10. Перегляд Scaladoc при огляді коду залежноситей бібліотеки

Іншим місцем, де можна зручно переглянути документацію є власне сайт бібліотеки з головної сторінки можна перейду в заголовку сторінки за посиланням документації і користувач потрапить на сторінку перегляду Scaladoc у автоматично згенерованому клієнтському застосунку з готовою навігацією по класам і деклараціям бібліотеки (рис. 3.11)

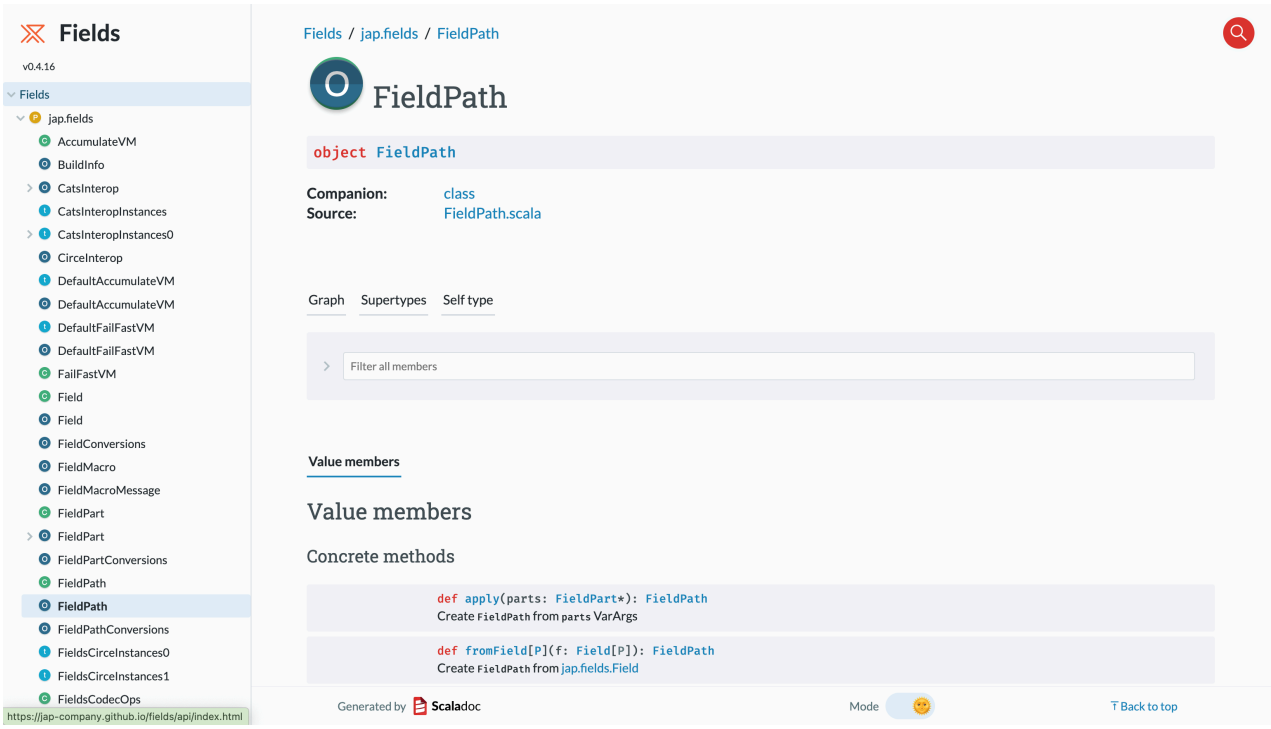


Рис. 3.11. Перегляд Scaladoc при огляді коду залежноситей бібліотеки

Також на цій же сторінці наявний пошук в якому можна шукати не тільки за символьними назвами а й за допомогою їх сигнатур, достатньо ввести тип аргументів та результат виконання, щоб знайти потрібний метод (рис. 3.12)

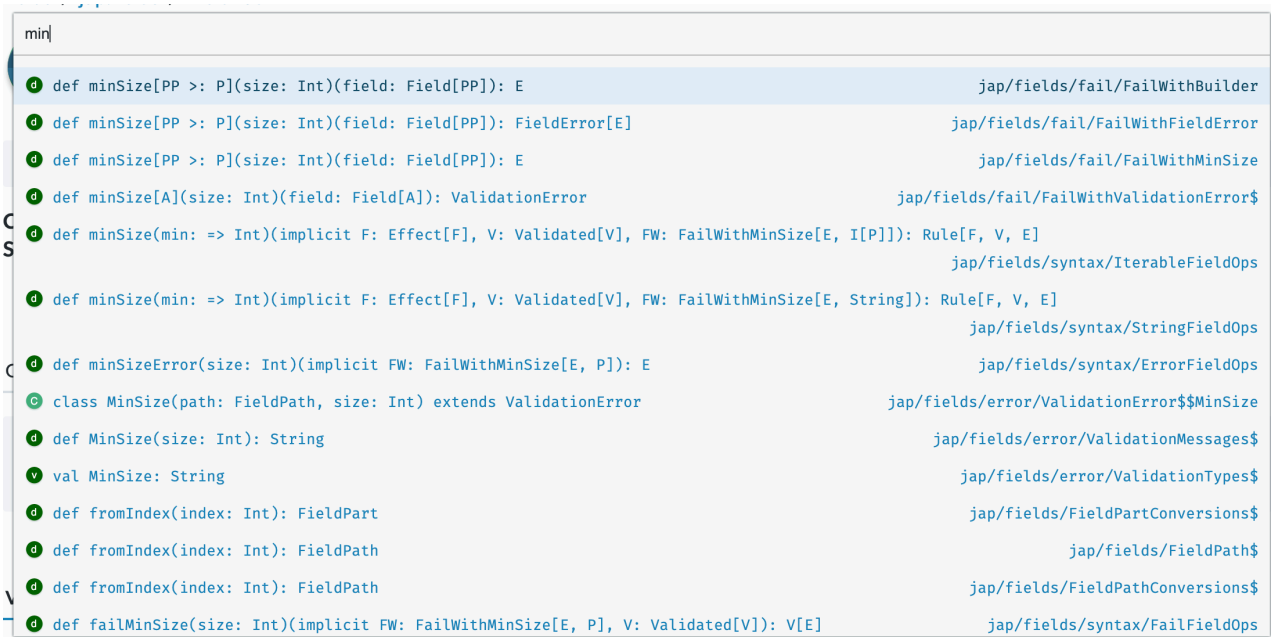


Рис. 3.12. Перегляд Scaladoc при огляді коду залежноситей бібліотеки

Як видно з рисунку ввівши хоча б фрагмент назви метода з’являється підказка щодо наявних реалізацій і є змога перейти проглянути її деталі.

3.2.2 Динамічна документація – Mdoc та сайт бібліотеки

При огляді структури проекту було згадано теку docs з документацією проекту, вона написана в форматі Markdown з певними покращеннями, покращення полягають в тому, що фрагменти коду написані в документації автоматично компілюються та запускаються – це забезпечує перевірку, того що наведені фрагменти коду правильні як синтаксично так і логічно, а також робить підтримку документації в актуальному стані простішою, адже якщо приклади з документації будуть неактуальними то проект не зкомпілюється.

За результатами виконання піля кожної стрічки фрагменту коду автоматично виводиться результат її виконання для наглядності перегляду користувачем (рис 3.13)

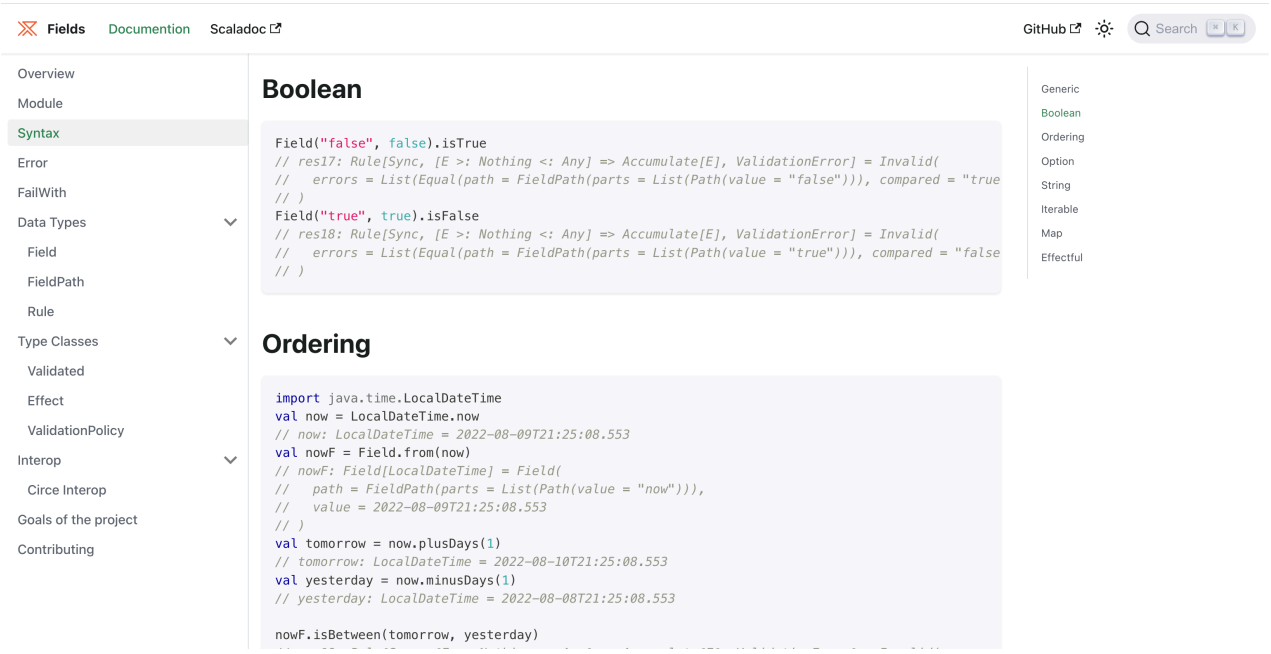


Рис. 3.13. Перегляд MDoc документації бібліотеки

З рисунку видно, що написана в форматі MDoc документація та приклади компілюється аналогічно всьому іншому коду і демонструється результат виконання дій, що створює інтерактивний досвід прочитання документації.

3.2.3 Покриття Unit тестами

При написанні КС не можна обійтись без тестування продукту. Тести це не тільки перевірка правильності, що написаний код зараз працює вірно – це також інструмент для інкрементальної розробки, спосіб впевнетись, що зробивши певні зміни в продукті немає регресії і користувачі залишуться задоволені, немає багів та все працює як треба. Інша користь від тестів – це наглядний приклад користування бібліотекою та тих речей, які можна робити. Розробникам часто доводиться витрачати години у пошуках вирішення їх специфічних проблем і якщо бібліотека має тести розробник з великою вірогідністю знайде аналогічне рішення в них.

Бібліотека fields не виключення і кожен її модуль покритий Unit тестами (рис. 3.14), де перевіряється її функціональність включно з роботою макросів та перевіркою всіх заявлених її властивостей.

```
class MacrosSuite extends munit.FunSuite {  Olejk4 +1
  test("Field.from") {
    val data = TestData()
    assertEquals(Field.from(data), Field(FieldPath.parse("data"), data))
    assertEquals(Field.from(data.nested), Field(FieldPath.parse("data.nested"), data.nested))
    assertEquals(
      Field.from(data.nested.deep),
      Field(FieldPath.parse("data.nested.deep"), data.nested.deep),
    )
  }

  test("Field.sub") {
    val data = TestData()
    assertEquals(Field.sub(data), Field(FieldPath.Root, data))
    assertEquals(Field.sub(data.nested), Field(FieldPath.fromPath("nested"), data.nested))
    assertEquals(
      Field.sub(data.nested.deep),
      Field(FieldPath.parse("nested.deep"), data.nested.deep),
    )
  }
```

Рис. 3.14. Unit тести бібліотеки

З цього рисунку видно приклад перевірки роботи макросу, що дістає назву полів з селектор функції, що дістає назви, за рахунок чого в разі виникнення регресії в ПЗ тести це виявлять.

3.3 Функціональне порівняння fields з аналогами

Тепер коли бібліотека вже реалізована прийшов час порівняти її функцінальність з бібліотеками аналогами (таблиця. 3.2) категорії порівняння ті ж самі що були в першому розділі, оскільки ці критерії були завчасно визначені, то в результаті імплементації бібліотеки вийшло виконати кожен з критеріїв, що є чудовим показником, який говорить, що в результаті створено повноцінний продукт з своєю екосистемою та інфраструктурою, який може стати в нагоді широкому спектру користувачів, кожен з яких може знайти те що він шукає при цьому матиме нагоду легко ознайомитись з документацією, Unit тестами, Scaladoc, прикладами коду, бенчмарками та рештою переваг бібліотеки fields.

Таблиця 3.2

Функціональне порівнянніія бібліотек

Бібліотека / Властивість	Fields	Accord	Dupin	Octopus	Either	Validated
Сайт	+	+	-	-	-	+
Документація	+	+	+	+	+	+
Підтримка	+	-	+	+	+	+
Канали зв’язку	+	-	-	-	+	+
Спільнота	+	-	-	-	+	+
Open-source	+	+	+	+	-/+	+
Статті	+	+	-	-	+	+
Доступність	+	+	+	+	+	+
Базові конструкції	+	+	-	-	-	-
Власні конструкції	+	+/-	+	+	+	+
Вибір середовища виконання	+	-	+	+	+/-	+/-
Вибір структури даних	+	-	-	-	-	-
Вибір помилки	+	-	+	+	+	+
Шлях виникнення помилки	+	+	+	+	+/-	+/-
DSL	+	+	-	-	-	-
fail-fast	+	-	-	-	+	+
short-circuit	+	N\A	N\A	N\A	+	+

Таблиця 3.2 (продовження)

Функціональне порівнянніія бібліотек

accumulate	+	+	+	+	-	+
Відсутність залежностей	+	-	-	-	+	+/-
Комплексні валідації	+	-	-	-	+	+
Перехресна валідація	+	-	-	-	+	+

В наведеній таблиці гарно видно, що в порівняння з бібліотеками аналогами вийшло вирішити ряд проблемних місць, що існували, і було враховано всі поставлені раніше вимоги.

3.4 Дослідження швидкодії fields у порівнянні з аналогами

Коли розробник обирає бібліотеку він керується певним набором критеріїв одним з ключових критеріїв для будь-якої КС це саме швидкодія, в залежності від потреб розробника по цьому критерії він може вирішити чи готовий він пожертвувати швидкодією заради якихось інших аспектів бібліотеки, важливо розуміти, що чим складніше бібліотека чим зручніше синтаксис тим більше вона відповідно і високорівнева. Високорівневі речі завжди будуть повільніше за низькорівневі, можна провести аналогію з написанням одного і того ж алгоритму на Python, Java, C++, C та асемблері достатньо самих тільки назв мов програмування, щоб зрозуміти, що буде працювати швидше, проте з цього ж випливає розуміння, що і писати той чи інший алгоритм буде складніше, адже чим нижче рівень тим більше буде якихось речей, які треба робити власноруч.

Валідація даних в цьому випадку не виключення і коли буде порівнюватись швидкодія потрібно враховувати нюанси та принципи робіт механізмів.

Власне тепер було визначено, що швидкодія це лише одна з метрик, можна перейти власне до заміру швидкодії нашої бібліотеки та бібліотек аналогів. Для

цього буде використано фреймворк для мікро бенчмарків JMH [13], він дозволяє конфігурувати певні сценаріції валідації та заміряти швидкість виконання певного коду, з певною кількістю кроків для прогріву та певною кількістю власне ітерацій заміру швидкодії. Є змога сконфігурувати JVM спільними для всіх бенчмарків параметрами, щоб всі були в однакових умовах, також крім метрики швидкодії JMH надає інформацію про алокацію пам'яті як сумарну так і кількість алокацій в секунду, інформація про кількість збірок сміття – GC та його сумарний час і кожна з цих метрик є важливою при виборі бібліотеки, адже в кожної КС свої потреби.

По результатам виконання бенчмарків створюється таблиця з результатами, а також JSON файл з повною інформацією про кожний прохід бенчмарка, який можна візуалізувати на сайті https://jmh.morethan.io/ проте для зручності було реалізовано власну візуалізацію у вигляді bar chart-ів на основі бібліотеки recharts.js переглянути їх можна прямо в браузері за посиланням https://jap-company.github.io/fields/benchmarks.

Тобто було обрано певний набір валідаційних правил і кожне з них реалізували на порівнюваних бібліотеках, при цьому дані, які валідуються було поділено на 3 категорії: повністю вірні, повнітю невірні та випадкові. Тепер можна перейти до аналізу отриманих результатів і почати з Throughput, що являє собою кілкість операцій в секунду, результати можна побачити на представлених графіках (рис 3.15, рис 3.16, рис. 3.17).

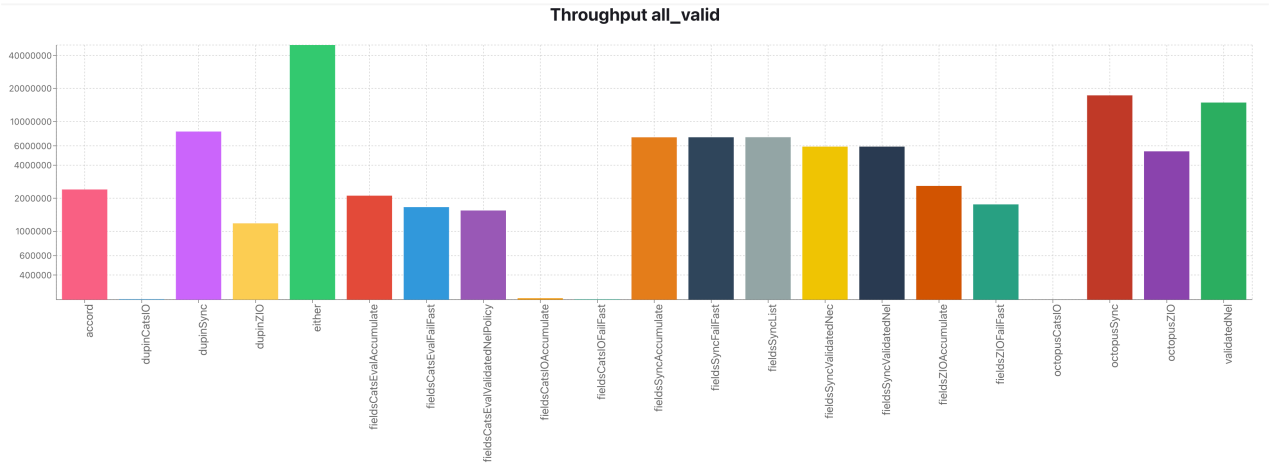


Рис. 3.15. Швидкодія для повністю вірних даних

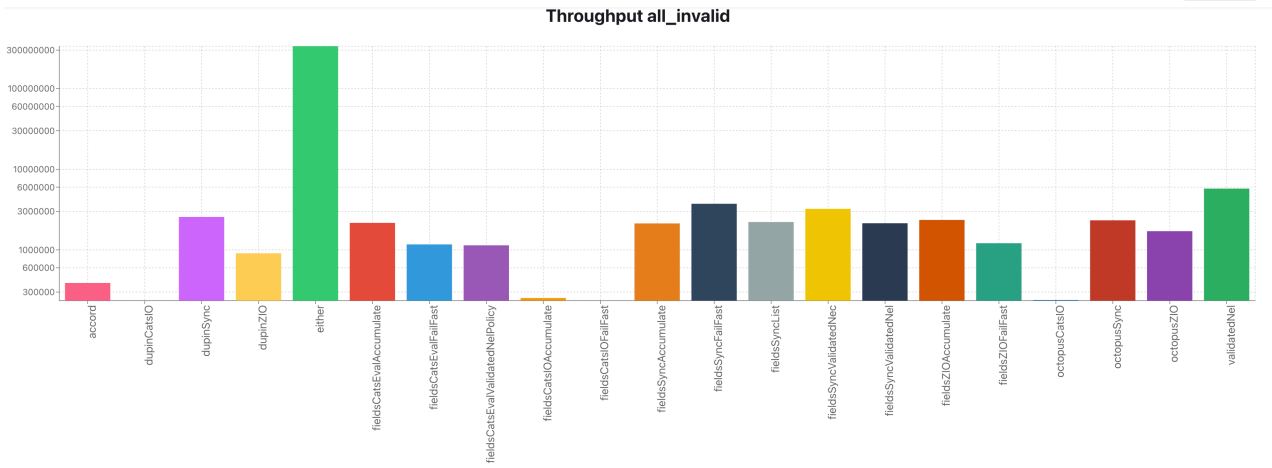


Рис. 3.16. Швидкодія для повністю невірних даних

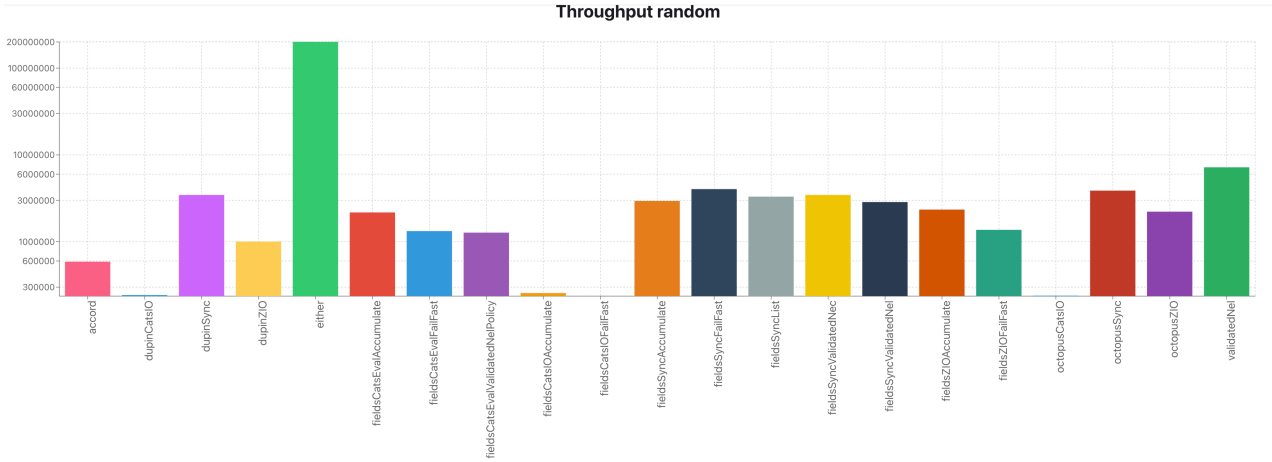


Рис. 3.17. Швидкодія для випадкових даних

На всіх графіках виділяється Either як вже зазначалось він є найнизькорівневішим з поміж усіх тому це не дивно, що він найшвидший.

ValidatedNel теж показав себе досить добре, проте гірше, бо він робить складніші трансформації і варто враховувати, що він акумулює помилки на відміну від Either, який працює по принципу fail-fast.

Непогано себе показав accord, але серед бібліотек синхронна валідація найшвидша у octopus і воно і не дивно, адже якщо переглянути вихідний код стає зрозуміло, що там майже немає ніякої складної логіки, проте з цього ж випливають недоліки бібліотеки, які вже згадувались раніше.

В плані синхронної валідації бібліотека fields показала себе гідно і для більшості валідаційних структур результати схожі. Dupin показав себе аналогічно fields.

Окремо варто зазначити cats та zio бенчмарки бібліотек вони на відміну від синхронних валідацій виглядають дуже повільними, проте так воно виглядає лише в рамках одиночних запусків валідації, якби це був повноцінний застосунок вони показали б себе набагато краще за синхронні варіанти, варто враховувати, що для того, щоб асинхронні бібліотеки розкрили себе на повну їм треба певний час попрацювати на прогрітому сервері, тому їх варто порівнювати лише між собою, а не з синхронними версіями.

Варто звернути увагу на Eval варіанти вони мають гірший перфоманс ніж синхронні версії, проте за їх використання кількість алокацій пам’яті менша адже відпрацьовує механізм short-circuit і у випадку fail-fast валідації виконуються лише ті правила, які потрібно виконувати.

В цілому видно тенденцію, що всі підходи мають меншу швидкодію чим більше невірних даних зустрічається, адже це потребує додаткових алокацій для повернення помилок та шляхів, які ведуть до них.

В заключення можна зробити висновок, що бібліотека fields має широкий спектр застосування, адже дає можливість обирати валідаційну структуру, середовище виконання та тип помилки за рахунок цього користувач може налаштувати її під свої потреби та досягти бажаної швидкодії при цьому не сильно втративши в інших аспектах роботи з бібліотекою.

ВИСНОВКИ ДО РОЗДІЛУ 3

В цьому розділі було оглянуто простий та складний приклади використання бібліотеки в залежності від потреб користувача, структура проекту та інфраструктури, документація та Scaladoc проекту, покриття unit тестами, доступні встроєні валідаційні правила, розглянуто, де і як користувач може ознайомитись з тими чи іншими принципами бібліотеки.

На додачу до цього було досліджено та порівняно створену бібліотеку з аналогами в результаті чого виявилось, що вона включає в себе всі аспекти поставленої до неї задачі та успішно поєднує їх всі в собі, що робить її дуже універсальної, щоб кожен розробник незалежно від його потреб міг нею зкористатися.

Було розроблено та проаналізовано бенчмарки бібліотеки та аналогів і досліджено швидкодію та алокацію пам’яті при різних конфігураціях даних, середовищ виконання, валідаційних структур та типів помилок, при цьому було розглянуто прицнипи роботи цих підходів та як вони впливають на результати бенчмарків.

РОЗДІЛ 4
РОЗРОБКА СТАРТАП-ПРОЄКТУ

4.1 Інформаційна карта проєкту

Розпочнемо з розробки інформаційної картки проєкту для короткого ознайомлення з ним.

Таблиця 4.1

Інформаційна карта проєкту

1. Назва проекту	Fields — бібліотека для комплексної валідації даних, що пропонує налаштовуваний DSL для гнучкої адаптації під різні потреби валідації.
2. Автор проєкту	Олег Дубинський
3. Коротка анотація	Fields — це бібліотека, розроблена для мови програмування Scala, яка дозволяє ефективно виконувати комплексну валідацію даних з урахуванням індивідуальних потреб системи. Вона використовує унікальний підхід з тайпкласами та підтримує налаштування середовища виконання, структури валідації та типу помилок. Fields підходить для великих та розподілених систем, де потреба в якісній валідації є критичною. Додаткові платні модулі можуть надавати розширені функції, такі як автоматичне створення валідаційних схем, консультації та підтримка з інтеграції бібліотеки.
4. Термін реалізації проекту	6 місяців
5. Необхідні ресурси	Фінансові ресурси
	2 Scala розробники — 70 000 грн./міс. (загалом 140 000 грн./міс.) 1 технічний письменник для документації — 30 000 грн/міс Маркетолог для просування бібліотеки на GitHub та інших ресурсах — 40 000 грн./міс. Проєктний менеджер – 60 000 грн./міс. Оренда домену в рік – 1120 грн. Оформлення торгової марки – 8000 грн. Оформлення авторського права – 4000грн. Всього: 1 633 120 грн.

Таблиця 4.1 (продовження)

Інформаційна карта проєкту

	Матеріальні ресурси
	Ліцензія на Intellij IDEA – 4185 грн. (2 шт.) Ноутбук Macbook Pro 16 M3 – 76 878 грн. (2 шт.) Ноутбук MacBook Air 13 M1 – 28 208 грн. (3 шт.) Всього: 246 750 грн.
	Інтелектуальні ресурси
	Наукові статті, документація, професійна література — безкоштовно
	Загальні витрати на всі ресурси: 1 879 870 грн
Опис проблеми	Існуючі бібліотеки для валідації даних мають низку обмежень, таких як складність розширення, обмежена підтримка асинхронних операцій та відсутність гнучких конфігурацій для комплексних валідацій. Fields пропонує вирішення цих проблем, надаючи універсальний підхід до валідації даних, що підходить для різних потреб.
Головні цілі та завдання	Головна ціль — розробити бібліотеку, яка стане стандартом для валідації даних в Scala. Завдання: • Реалізувати підтримку конфігурації бібліотеки під потреби користувача: середовище виконання, валідаційна структура, тип помилки • Забезпечити інтеграцію з популярними фреймворками та бібліотеками. • Розробити багатий ідеоматичний DSL, що завоює серця користувачів • Отримати 100 активних користувачів протягом першого року.
Очікувані результати	Бібліотека Fields стане важливим інструментом у розробці на Scala, дозволяючи зменшити витрати на підтримку систем та підвищити якість обробки даних.

4.2 Формування команди стартапу

Тепер перейдемо до створення команди для виконання цього стартап-проєкту. Команда складатиметься з 5 людей, в ній немає тестувальників тому що кінцевий продукт - це саме бібліотека для тестування якої потрібно вміти

програмувати на Scala тому замість тестувальників розробники покривають весь функціонал різноманітними тестами.

Також можна помітити, що в команді немає Front-end розробників адже весь UI генерується автоматично за допомогою фреймворку Docusaurus, тобто Scala розробники здатні підтримувати його самі.

Важливимими ролями також є технічний письменник, який у тісній співпраці з розробниками, буде підтримувати документацію впорядку та маркетолог, що буде вміло просувати бібліотеку на тих ресурсах, де зазвичай бувають Scala розробники та вестиме акаунти соц. мереж проєкту.

Але найважливішою людиною в будь-якому стартап-проєкті є проєктний менеджер на плечах якого лежить налагодження всіх процесів всередині команди та її взаємодія між собою

Таблиця 4.2

Склад команди стартапу

№	Роль	Зона відповідальності	Кількість
1	Scala розробник (Full-stack)	Розробка та підтримка основного функціоналу бібліотеки, розробка та оптимізація алгоритмів валідації. Написання Unit, Integration тестів та синтетичних тестів.	2
2	Технічний письменник	Підготовка документації для користувачів, створення API-документації, підготовка навчальних матеріалів.	1
3	Маркетолог	Розробка та реалізація стратегії просування бібліотеки на ринку, взаємодія з користувачами, підтримка на соціальних платформах.	1
4	Проєктний менеджер	Координація роботи команди, контроль за виконанням завдань, планування релізів і комунікація з клієнтами.	1

Тепер визначимо важливість наявних фаторів для реалізації стартап-проєкту:

Таблиця 4.3

Важливість факторів реалізації інноваційного проєкту

№	Фактор	Важливість
1	Створення та просування ідеї	9
2	Формування бізнес-плану	7
3	Компетентність команди	9
4	Співпраця та комунікація	8
5	Технічна якість продукту	10
6	Виконання завдань у межах термінів	8
7	Розробка документації	6
8	Просування та маркетинг продукту	7

Далі проаналізуємо важливість цих факторів для кожного з співробітників у вигляді таблиці та наскільки сильно залежить кожен пункт від конкретного працівника

Таблиця 4.4

Важливість факторів реалізації інноваційного проєкту для співробітників

Фактор	Scala розробники	Технічний письменник	Маркетолог	Проєктний менеджер	
Створення та просування ідеї	20	10	40	80	
Формування бізнес-плану	5	5	20	80	
Компетентність	90	60	60	65	
Співпраця та комунікація	15	30	20	100	
Виконання обов’язків	90	30	20	80	
Технічна якість продукту	100	80	10	20	
Просування та маркетинг продукту	5	5	100	30	
Разом	325	20	270	455	1 270
У відсотках	25.6%	7.3%	21.25%	35.82%	100%

Як бачимо з аналізу всі члени команди мають більш-менш рівний внесок, окрім проєктного менеджера, адже на ньому найбільше відповідальності за продукт і в нього найширший діапазон обов’язків

4.3 Формулювання оновної ідеї стартап проєкту

Основна ідея проєкту Fields полягає у створенні інноваційної бібліотеки для комплексної валідації даних, що надає можливість гнучкого налаштування процесів валідації відповідно до потреб будь-якої КС. Fields забезпечує розробників інструментами для точного та надійного контролю якості даних через унікальний підхід, що базується на використанні тайпкласів, гнучкого Domain-Specific Language (DSL) та можливості конфігурування середовища виконання, структури валідації та типу помилок.

Бібліотека спроектована з акцентом на високу продуктивність і здатність працювати як у синхронному, так і в асинхронному режимах. Це дозволяє легко адаптувати її для розподілених систем та забезпечити максимальну точність перевірки даних в умовах великого навантаження. Ключовим аспектом підходу бібліотеки є саме можливість комплесної валідації даних, яка передбачує різноманітну взаємодію між полями, які валідуються.

Крім основного функціоналу, проєкт Fields передбачає:

- Генерація валідаційних схем, яка надаватиметься як окремий платний модуль та можливість генерувати документацію по цим схемам, а також генерація валідації на клієнтській стороні
- Платна підтримка та консультації з питань інтеграції та використання бібліотеки.
- Консультаційні послуги щодо налаштування валідаційних процесів для великих проєктів.

Ця ідея має на меті задовольнити попит на універсальну, налаштовувану та легку у використанні систему валідації даних, яка підійде як для невеликих проєктів, так і для великих корпоративних систем.

Таблиця 4.5

Плюси та мінуси стартап-проєкту Fields

№	Плюси	Мінуси
1	Гнучкий підхід до валідації даних завдяки використанню тайпкласів та налаштовуваного DSL	Складність реалізації та налаштування інфраструктури для комплексної валідації
2	Підтримка різних середовищ виконання, валідаційних структур, типів помилок, що підвищує універсальність використання	Високі вимоги до кваліфікації розробників, особливо для підтримки DSL
3	Можливість розширення бібліотеки через додаткові платні модулі	Конкуренція з іншими бібліотеками валідації, що можуть мати більше функцій
4	Платні послуги підтримки та консалтингу, що створюють додаткові джерела доходу	Витрати на маркетинг та просування продукту серед спільноти Scala
5	Висока точність та надійність валідації, що забезпечує якість даних	Необхідність постійного оновлення та підтримки для збереження актуальності
6	Прозора інтеграція з іншими бібліотеками та фреймворками на Scala	Можливі проблеми сумісності з новими версіями Scala або сторонніх бібліотек
7	Підтримка комплесної валідації даних, що дозволяє бізнес логіку різної складності	Потреба у великій кількості тестування для забезпечення стабільності
8	Відкрите програмне забезпечення з документацією, доступне для широкого загалу	Ризик недостатньої залученості користувачів або недостатньої популярності

Згідно описаних плюсів та мінусів плюси переважають, а мінуси радше виглядають, як виклики, як потрібно буде побороти на шляху реалізації проєкту.

Для успішної реалізації проєкту варто проаналізувати всі технічні аспекти продукту та порівняти з аналогами, якщо в третьому розділі було проаналізовано саме технічні складові, то тут більше цікавить функціональна різниця fields та продуктів аналогів.

Таблиця 4.6

Техніко-економічні характеристики стартапу Fields та продуктів-аналогів

№	Характеристика	Fields	Аналоги (Dupin, Accord, Octopus, Cats Validated)
1	Гнучкий DSL для валідації даних	Так	Частково
2	Підтримка синхронної та асинхронної валідації	Так	Частково (не всі аналоги підтримують асинхронну валідацію)
3	Можливість адаптації під різні валідаційні структури (fail-fast, accumulate)	Так	Обмежена
4	Типізація та можливість використання тайпкласів	Так	Лише деякі аналоги підтримують тайпкласи
5	Простота інтеграції з іншими Scala бібліотеками	Так	Частково
6	Можливість автоматичного створення валідаційних схем	Доступно в платному модулі	Відсутня
7	Рівень технічної підтримки	Open-source трекер питань та проблем + можлива платна підтримка і консультації для компаній	Обмежена або відсутня
8	Відкритість коду (Open-source)	Так	Так

Таблиця 4.6 (продовження)

Техніко-економічні характеристики стартапу Fields та продуктів-аналогів

№	Характеристика	Fields	Аналоги (Dupin, Accord, Octopus, Cats Validated)
9	Наявність активної документації та прикладів	Так	Частково (деякі аналоги мають базову документацію)
10	Рентабельність	Потенційно висока завдяки платним модулям і підтримці та потенційним спонсорам бібліотеки	Залежить від кількості користувачів
11	Цільовий ринок	Scala-розробники в IT-компаніях, що працюють з великими даними та складними системами	Scala-розробники
12	Залежність від версій Scala	Можлива адаптація під нові версії	Деякі аналоги не підтримуються на нових версіях
13	Функціонал	Високий, включає розширені можливості валідації	В основному базові функції
14	Унікальні конкурентні переваги	Можливість розширення через платні модулі, розширений DSL, адаптивність до специфічних вимог	Відсутні або частково реалізовані
15	Ступінь залученості користувачів	Потенційно високий за рахунок додаткових функцій та підтримки	Частково високий, залежно від популярності

Згідно таблиці видно, що бібліотека fields є гарним конкурентом для продуктів аналогів, що є позитивним показником для реалізації стартап проєкту.

4.4 Аналіз ринкових можливостей

Таблиця 4.7

Попередня характеристика потенційного ринку стартап-проєкту

№	Показники стану ринку	Характеристика
1	Кількість головних гравців	Обмежена кількість конкурентів - 3 основних
2	Загальний обсяг ринку	Значний, зростання попиту на інструменти для якісної валідації в умовах великих та розподілених систем
	Загальний обсяг продаж від реклами, грн./ум.од.	
3	Динаміка ринку	Ринок зростає завдяки потребі в якісному управлінні даними, масштабованості та інтеграції в розподілених системах
4	Наявність обмежень для входу	Низькі бар'єри для входу з точки зору технології; потреба у високій якості та підтримці бібліотеки для зростання популярності
5	Специфічні вимоги до стандартизації	Відсутні особливі вимоги до сертифікації; бібліотека має відповідати стандартам функціонального програмування для Scala
6	Середня норма рентабельності в галузі	(613 370/2 745 213) * 100 = 22.34% ра рік

Тепер розглянемо характеристику потенційних клієнтів стартап-проєкту fields:

Таблиця 4.8

Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних цільових груп клієнтів	Вимоги до споживачів товару
1	Комплексна валідація даних у великих та розподілених системах	IT-компанії, що працюють з великими даними, розподіленими системами та фінансовими застосунками	Великі компанії шукають стабільні рішення з підтримкою, стартапи – економічно вигідні опції	Гнучкість налаштування та підтримка високої продуктивності бібліотеки
2	Забезпечення відповідності даних правилам для мінімізації помилок	Розробники на Scala, які працюють з великими обсягами даних або критичними до помилок системами	Розробники, що займаються великими проєктами, потребують розширених можливостей для інтеграції	Проста інтеграція, доступна документація, можливість адаптації під проєкт
3	Оптимізація процесу валідації даних	Компанії, які шукають готові рішення для валідації в комплексних системах	Більші компанії шукають підтримку та консультації, менші потребують базових рішень	Доступність платних модулів, які зменшують витрати на налаштування
4	Автоматизація перевірки відповідності даних правилам	Компанії, що використовують скриптові або великі аналітичні системи	Компанії з інтенсивними процесами обробки даних потребують автоматизації	Підтримка fail-fast і accumulate режимів для адаптивної роботи
5	Дотримання стандартів валідації та підтримка масштабованості системи	Розробники складних систем для підприємств, IT-консультанти	Консультанти та системні інтегратори потребують можливості гнучкої адаптації бібліотеки	Підтримка останніх стандартів, наявність платної підтримки та консультацій

Далі оглянемо фактори загроз для стартапу fields:

Таблиця 4.9

Фактори загроз

№	Фактор загрози	Зміст загрози	Можлива реакція компанії
1	Поява нових конкурентів з кращими можливостями	Конкуренти можуть запропонувати інноваційні функції або більше гнучкості, переманюючи клієнтів	Постійне оновлення продукту, додавання нових функцій і вдосконалення існуючих
2	Швидкий розвиток технологій	Ринок швидко змінюється, і нові стандарти чи технології можуть зробити Fields застарілим	Постійний моніторинг технологічних трендів та своєчасне оновлення продукту
3	Високі вимоги до якості та стабільності від користувачів	Клієнти очікують стабільності і надійності, особливо для великих проектів, і можуть відмовитися при виявленні проблем	Впровадження ретельного тестування, автоматизація CI/CD для швидкого виправлення помилок
4	Недостатньо ефективна маркетингова кампанія	Низький рівень обізнаності про продукт може знизити його конкурентоспроможність на ринку	Інвестиції в цифровий маркетинг, активна комунікація з спільнотою Scala
5	Можливі проблеми з сумісністю з новими версіями Scala	Застарілі бібліотеки або проблеми з сумісністю з новими версіями мови можуть відлякати користувачів	Пріоритетна підтримка нових версій Scala, регулярні оновлення та тестування
6	Складність для користувачів без глибоких технічних знань	Складна конфігурація та налаштування бібліотеки можуть бути викликом для менш досвідчених розробників	Створення докладної документації, покрокових інструкцій, проведення вебінарів

Таблиця 4.9 (продовження)

Фактори загроз

№	Фактор загрози	Зміст загрози	Можлива реакція компанії
7	Недостатня залученість користувачів та низька активність спільноти	Відсутність зворотного зв'язку і пропозицій від користувачів може ускладнити розвиток продукту	Активне залучення користувачів, програми лояльності та підтримка open-source спільноти
8	Можливість копіювання функціоналу конкурентами	Конкуренти можуть швидко запровадити аналогічний функціонал, що зменшить унікальність Fields	Постійна інновація, акцент на унікальних можливостях та швидкість оновлень

Таблиця 4.10

Фактори можливостей

№	Фактор можливості	Зміст можливості	Можлива реакція компанії
1	Постійне зростання потреби у комплексній валідації даних	Збільшення попиту на інструменти для управління якістю даних в умовах великих і розподілених систем	Просування Fields як рішення для великомасштабних і розподілених систем
2	Розширення функціоналу через платні модулі	Можливість отримувати додатковий дохід за рахунок платних модулів, таких як генерація валідаційних схем або автоматичні налаштування	Створення додаткових модулів, таких як генерація схем, адаптивна валідація
3	Зростання популярності мови Scala	Популяризація Scala у великих компаніях та фінансових установах відкриває нові ринки для бібліотеки	Розширення маркетингової кампанії на компанії, що впроваджують Scala
4	Вихід на міжнародний ринок	Можливість залучення користувачів з інших країн, особливо в Європі та США	Розробка документації англійською мовою, участь у міжнародних конференціях
5	Співпраця з великими IT-компаніями	Можливість укладання партнерських угод для інтеграції Fields у масштабні проекти	Встановлення партнерських зв’язків з великими IT-компаніями, що використовують Scala

Таблиця 4.10 (продовження)

Фактори можливостей

№	Фактор можливості	Зміст можливості	Можлива реакція компанії
6	Підтримка open-source спільноти	Підвищення лояльності користувачів та їхнього зворотного зв'язку завдяки open-source моделі	Створення активної спільноти, організація воркшопів і підтримка на GitHub
7	Поширення комплексних систем управління даними	Зростаючий інтерес до комплексних систем управління даними вимагає якісних рішень для валідації даних	Акцент на маркетингових матеріалах, що показують переваги для таких систем
8	Інновації у сфері функціонального програмування	Розвиток функціонального програмування сприяє популяризації складних інструментів валідації	Інтеграція з новітніми функціональними фреймворками та підтримка стандартів

Таблиця 4.11

Ступеневий аналіз конкуренції на ринку

Конкурентне середовище	Прояв конкуренції	Вплив на проєкт	Можливі покращення
Монополістичне	Ринок рішень для валідації даних на Scala має обмежену кількість конкурентів	Відносно легкий вихід на ринок, але потрібно переконати користувачів у перевагах	Встановлення унікальних функцій, що відрізняють Fields від аналогів
Глобальне	Конкуренти розкидані по всьому світу, популярні у США, Європі	Необхідність розширення ринку, вихід на міжнародний рівень	Створення документації англійською мовою, просування на глобальних платформах
Внутрішньо-галузеве	Конкуренція в основному в галузі IT, серед компаній, що використовують Scala	Ринок обмежений компаніями, що працюють з даними та використовують Scala	Просування Fields як спеціалізованого рішення для IT та великих даних

Таблиця 4.11 (продовження)

Ступеневий аналіз конкуренції на ринку

Конкурентне середовище	Прояв конкуренції	Вплив на проєкт	Можливі покращення
Товарно-родове	Конкуренція з іншими бібліотеками для валідації даних на Scala	Необхідність інтегрування та покращення функцій, схожих на існуючі аналоги	Вдосконалення існуючих функцій, додавання унікальних модулів і підтримки
Нецінове	Конкуренція базується на якості та функціональності, а не на ціні	Користувачі обирають рішення на основі можливостей та зручності	Покращення якості, швидкості та простоти використання, забезпечення документації
Немарочне	Бренд менш важливий, ніж функціональність	Важливо надавати продукт високої якості, який вирішує конкретні потреби	Фокус на демонстрації переваг та надійності Fields

Таблиця 4.12

Аналіз конкуренції в галузі згідно М. Портера

Складові аналізу	Опис конкуренції	Вплив на проєкт	Можливі покращення
Прямі конкуренти в галузі	Існує кілька популярних бібліотек для валідації даних на Scala, зокрема Dupin, Accord, Octopus, Cats Validated	Потрібно відрізнятися від прямих конкурентів, забезпечуючи унікальні функції та платні модулі	Розробка функцій, що не мають аналогів у конкурентів, активне просування та підвищення зручності інтеграції
Потенційні конкуренти	Можливий вихід на ринок нових гравців, які запропонують вдосконалені або дешевші рішення	Ризик втрати частки ринку при виході нових конкурентів	Постійне вдосконалення функціоналу, створення лояльної спільноти користувачів, оновлення продукту відповідно до нових вимог ринку

Таблиця 4.12 (продовження)

Аналіз конкуренції в галузі згідно М. Портера

Складові аналізу	Опис конкуренції	Вплив на прокєт	Можливі покращення
Постачальники	Основні «постачальники» - це розробники бібліотек, які забезпечують технічну підтримку	Важлива роль кваліфікованих розробників для підтримки та вдосконалення продукту	Забезпечення мотивації команди розробників, залучення професіоналів для розвитку бібліотеки та її підтримки
Клієнти	Клієнти — це IT-компанії та розробники, які використовують Scala, зокрема для складних систем	Високий рівень очікувань щодо якості та підтримки продукту	Налагодження якісної підтримки, активна комунікація з клієнтами, забезпечення детальної документації та зручності інтеграції
Товари-замінники	Замінники — інші бібліотеки для валідації даних, написані для інших мов, або власні розробки компаній	Деякі компанії можуть розробляти власні рішення або обирати продукти на інших мовах	Акцент на унікальних можливостях бібліотеки Fields для Scala, постійне вдосконалення та інтеграція з популярними фреймворками

Таблиця 4.13

Фактори конкурентоспроможності

№	Фактор	Опис	Можливі покращення
1	Унікальний функціонал	Fields пропонує розширені можливості для налаштування та гнучкість у валідації, зокрема синхронну та асинхронну валідацію, вибір структури помилок та режимів (fail-fast, accumulate)	Постійна розробка нових функцій, які забезпечують додаткову вартість для користувачів
2	Додаткові платні модулі	Розширення функціоналу за допомогою платних модулів, таких як генерація валідаційних схем, підтримка та консалтинг, що додає гнучкості та дозволяє отримувати додатковий дохід	Пропонування додаткових модулів на основі потреб користувачів і ринку

Таблиця 4.13 (продовження)

Фактори конкурентоспроможності

№	Фактор	Опис	Можливі покращення
3	Висока якість та надійність	Забезпечення стабільної та якісної роботи продукту, що важливо для великих проєктів та розподілених систем	Впровадження строгого тестування та автоматизації процесів, регулярні оновлення
4	Простота інтеграції та детальна документація	Легка інтеграція з іншими бібліотеками на Scala та зрозуміла документація для швидкого впровадження	Постійне вдосконалення документації, створення прикладів використання
5	Підтримка та технічна допомога	Можливість отримати консультації та підтримку в інтеграції Fields в масштабні проєкти	Пропонування планів підтримки, регулярні вебінари та консультації для клієнтів
6	Активна спільнота користувачів	Наявність open-source підходу та взаємодія зі спільнотою сприяє розвитку продукту та його популяризації	Залучення користувачів через open-source платформи, підтримка GitHub, організація воркшопів
7	Адаптивність до нових версій Scala	Продукт підтримує нові версії Scala, що важливо для стабільної роботи у різних проєктах та збереження актуальності	Оперативне тестування та підтримка продукту на нових версіях мови Scala
8	Розширення ринку через міжнародне просування	Вихід на міжнародний ринок (США, Європа), зокрема для IT-компаній, що використовують Scala у великих системах	Створення англомовної документації, просування на міжнародних IT-конференціях та спільнотах

Таблиця 4.14

Порівняльний аналіз сильних та слабких сторін

№ п/п	Фактор конкурентоспроможності	Бали (1-20)	Рейтинг товарів-замінників у порівнянні з Fields (-3 до +3)
1	Унікальний функціонал (гнучкість, розширені можливості налаштування валідації)	19	-3
2	Додаткові платні модулі	17	-2
3	Висока якість та надійність	18	-2
4	Простота інтеграції та детальна документація	16	-1
5	Підтримка та технічна допомога	15	-2
6	Активна спільнота користувачів	14	-1
7	Адаптивність до нових версій Scala	15	-2
8	Розширення ринку через міжнародне просування	13	-1

Таблиця 4.15

SWOT-аналіз стартап-проєкту

Характеристика	Опис
Сильні сторони (Strengths)	- Гнучкий та розширений функціонал: Fields пропонує унікальні можливості налаштування валідації, які включають синхронний і асинхронний режими, fail-fast та accumulate валідацію, що робить його привабливим для великих та складних проєктів. - Додаткові платні модулі: Можливість монетизації через додаткові модулі, такі як генерація валідаційних схем і платні консультації. - Підтримка спільноти: Проєкт має open-source модель, яка дозволяє залучати активну спільноту, забезпечуючи швидкий розвиток та зворотний зв’язок від користувачів. - Адаптивність до нових версій: Швидка підтримка нових версій Scala дозволяє залишатися актуальним для користувачів, що використовують останні технології.

Таблиця 4.15 (продовження)

SWOT-аналіз стартап-проєкту

Характеристика	Опис
Слабкі сторони (Weaknesses)	- Високі технічні вимоги до реалізації: Розробка гнучкої та надійної бібліотеки вимагає високої кваліфікації розробників, що може вплинути на витрати та складність реалізації. - Висока залежність від мови Scala: Fields обмежений ринком користувачів, які використовують Scala, що звужує потенційну аудиторію порівняно з багатоплатформеними рішеннями. - Необхідність у постійному оновленні та підтримці: Для збереження конкурентоспроможності потрібні постійні оновлення, а це вимагає ресурсів та витрат на підтримку. - Потреба у сильній маркетинговій стратегії: Щоб конкурувати з уже відомими бібліотеками, необхідні інвестиції в маркетинг і просування продукту.
Можливості (Opportunities)	- Зростання попиту на якісну валідацію даних: Ринок інструментів для валідації даних активно зростає, особливо в умовах великих розподілених систем, де важлива надійність. - Вихід на міжнародний ринок: Популярність Scala зростає, особливо серед IT-компаній у США та Європі, що відкриває можливість для міжнародного розширення. - Співпраця з великими IT-компаніями: Укладання партнерських угод для інтеграції Fields у масштабні проєкти, що збільшить прибуток і репутацію продукту. - Підтримка open-source спільноти: Можливість розширити аудиторію та підвищити популярність продукту завдяки підтримці активної спільноти, що забезпечує зворотний зв'язок та ідеї для вдосконалення.
Загрози (Threats)	- Вихід нових конкурентів: Нові гравці можуть вийти на ринок з аналогічними або вдосконаленими функціями, що створює загрозу втрати частки ринку. - Швидкий розвиток технологій: Постійні зміни технологій можуть зробити існуючі рішення застарілими, якщо не вдасться вчасно оновити продукт. - Проблеми з сумісністю з новими версіями Scala: Відсутність підтримки нових версій мови або фреймворків може знизити популярність продукту. - Можливість копіювання функціоналу: Конкуренти можуть швидко впровадити аналогічний функціонал, зменшуючи унікальність Fields . - Високі вимоги до стабільності та якості: Користувачі очікують стабільності та високої якості продукту, і будь-які помилки можуть призвести до втрати довіри.

Таблиця 4.16

Альтернативи ринкового впровадження стартап-проєкту

Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
Альтернатива 1: Самостійне впровадження на ринок через open-source платформу – Розміщення Fields на GitHub як open-source проєкту. – Залучення спільноти через розробку документації та прикладів. – Підтримка користувачів через форуми, блоги та технічні статті. – Просування через соціальні мережі та форуми, що орієнтовані на Scala-розробників.	Висока (доступні ресурси, open-source спільнота)	Короткострокова (3-6 місяців)
Альтернатива 2: Співпраця з великими компаніями для інтеграції та просування – Налагодження партнерських зв'язків з IT-компаніями, які використовують Scala. – Інтеграція Fields у масштабні проєкти для забезпечення стабільного потоку клієнтів. – Надання платної підтримки та консультацій для інтеграції в корпоративні системи. – Публікація кейсів успішного впровадження для залучення нових клієнтів.	Середня (потрібен час на побудову партнерств)	Середньострокова (6-12 місяців)
Альтернатива 3: Створення комерційної версії з додатковими платними модулями – Розробка платних модулів (генерація схем, автоматичне налаштування). – Впровадження ліцензійної моделі для додаткових функцій і платної технічної підтримки. – Створення публічного сайту з документацією та онлайн підтримкою. – Просування комерційної версії через вебінари, статті, кейси та огляди.	Висока (потреба в ресурсах для розробки)	Середньо- та довгострокова (9-18 місяців)

Таблиці 4.16 (продовження)

Альтернативи ринкового впровадження стартап-проєкту

Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
Альтернатива 4: Вихід на міжнародний ринок з маркетинговою кампанією – Переклад документації на англійську мову. – Проведення маркетингової кампанії на міжнародних платформах (Reddit, Stack Overflow, Twitter). – Участь у міжнародних конференціях і заходах, орієнтованих на Scala-розробників. – Публікація оглядів і кейсів використання у відомих міжнародних IT-виданнях.	Середня (необхідні ресурси для маркетингу)	Довгострокова (12-24 місяців)

З усіх варіантів найбільш вдалим виглядає перший оскільки він майже не залежить від зовнішніх чинників та дає можливсть в подальшому розвинути рішення в сторону решти варіантів ринкових впроваджень

4.5 Розробка ринкової стратегії проєкту

Таблиця 4.17

Вибір цільових груп потенційних споживачів

Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
Великі IT-компанії, що використовують Scala для розробки розподілених систем - Розробники, що працюють із великими обсягами даних і складними системами, для яких важлива стабільна і масштабована валідація. - Цінують якісну підтримку, адаптивність, можливість швидкого налаштування	Висока	Високий	Висока	Середня

Таблиця 4.17 (продовження)

Вибір цільових груп потенційних споживачів

1	2	3	4	5
Стартапи та середні компанії, які активно використовують Scala - Переважно невеликі та середні IT-компанії, що використовують сучасні технології для розробки своїх продуктів - Відкриті до інноваційних рішень з гнучким налаштуванням, але потребують економічно доступних варіантів	Середня	Середній	Низька	Висока
Розробники, що займаються фрілансом та невеликими проектами на Scala - Незалежні розробники або малі компанії, що працюють з проєктами на Scala і потребують ефективних інструментів для валідації - Обирають рішення з доступною документацією та легкою інтеграцією, мінімально залежні від підтримки	Висока	Середній	Низька	Висока
IT-консультанти та інтегратори, що працюють над рішеннями для великих компаній - Працюють із комплексними проектами для корпоративного сектору, де важлива гнучка валідація даних - Цінують можливість налаштування, розширюваність та стабільну підтримку	Висока	Високий	Висока	Середня

Раціонально буде обрати в першу чергу першу та другу цільові групи, адже завоювавши прихильність компаній за рахунок кращої якості продукту решта цільових груп автоматично підхоплять продукт і почнуть ним користуватись

Таблиця 4.18

Визначення базової стратегії проєкту

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Спільне впровадження через open-source платформу та партнерство з великими компаніями - Розміщення Fields на GitHub для залучення широкої аудиторії через open-source підхід - Побудова партнерських відносин із великими IT-компаніями для інтеграції та стабільного доходу від платних модулів та підтримки	Нішове охоплення ринку з фокусом на Scala-розробників - Орієнтація на великі компанії, стартапи та фрілансерів, які використовують Scala для роботи з даними	- Гнучкий та потужний функціонал: гнучкість налаштування валідації, підтримка fail-fast і accumulate режимів, що робить продукт універсальним для різних застосувань - Додаткові платні модулі: пропонування платних функцій, таких як генерація валідаційних схем, для компаній, що потребують розширених можливостей - Open-source підтримка: наявність GitHub-репозиторію та активної документації для залучення спільноти, що знижує вартість впровадження - Партнерські зв'язки з великими компаніями: інтеграція у масштабні проєкти забезпечує стабільний потік клієнтів та додаткові можливості для монетизації	Диверсифікований розвиток - Впровадження open-source підходу для охоплення широкої аудиторії та зниження бар'єру для початкового використання - Розробка платних модулів та консультаційної підтримки для великих компаній, які потребують більш комплексного підходу - Розширення присутності на міжнародному ринку, розвиток документації та підтримка нових версій Scala для підвищення залученості користувачів

Таблиця 4.19

Визначення стратегії конкурентної поведінки

Питання	Відповідь
Чи є проєкт першопрохідцем на ринку?	Ні, ринок має кілька відомих бібліотек для валідації даних на Scala, як-от Dupin, Accord, Octopus.
Чи буде компанія шукати нових споживачів чи забирати існуючих у конкурентів?	Компанія прагне як залучити нових користувачів (Scala-розробників, що працюють із великими системами), так і переманити клієнтів у конкурентів за рахунок кращої гнучкості та надійної підтримки.
Чи буде компанія копіювати основні риси товару конкурента, і які?	Основні риси конкурентів не будуть копіюватися. Fields планує виділитися завдяки розширеному функціоналу (синхронна та асинхронна валідація, fail-fast та accumulate режими) та підтримці платних модулів для комплексної валідації.
Стратегія конкурентної поведінки	Стратегія диференціації та розширення - Fields виділятиметься за рахунок унікальних можливостей налаштування, розширеного функціоналу та додаткових платних модулів. - Компанія зосередиться на залученні нових користувачів, які ще не користувалися інструментами для валідації на Scala, а також на переманюванні існуючих клієнтів конкурентів, пропонуючи вигідніші можливості підтримки, консультаційні послуги та гнучкий функціонал. - Стратегія передбачає також активну взаємодію зі спільнотою розробників на GitHub і соціальних платформах, що дозволить створити лояльну аудиторію та забезпечити зворотний зв’язок для подальшого вдосконалення продукту.

Таблиця 4.20

Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Цільова аудиторія (розробники на Scala, IT-компанії, що працюють з великими системами, стартапи) очікує гнучкості, стабільності, легкості інтеграції та надійної технічної підтримки. Вони шукають продукт, який забезпечує комплексну валідацію даних, можливість налаштування під специфічні вимоги та додаткову підтримку для швидкого впровадження в проєкт.
Базова стратегія розвитку	Диверсифікований розвиток через open-source підхід та платні модулі Проєкт планує охопити широку аудиторію за рахунок доступності open-source версії, а також залучити компанії, що потребують додаткових функцій і технічної підтримки через платні модулі та консультації. Це поєднання дозволяє зберегти доступність продукту і водночас забезпечити стабільний дохід.
Ключові конкурентоспроможні позиції власного стартап-проєкту	- Гнучкість і функціональність: Fields пропонує можливість адаптації валідації під конкретні потреби завдяки fail-fast та accumulate режимам, а також синхронній і асинхронній роботі. - Додаткові платні модулі та консультаційна підтримка: Опція розширення функціоналу через платні модулі (генерація схем, підтримка) для клієнтів, що мають специфічні вимоги. - Open-source підтримка та активна спільнота: Відкритий код і доступність на GitHub залучають спільноту, що сприяє швидкому зворотному зв’язку та покращенню продукту.
Вибір асоціацій, які мають сформувати комплексну позицію власного проєкту	1. Надійність — продукт забезпечує стабільну і надійну валідацію даних, що критично для великих і розподілених систем. 2. Гнучкість — можливість налаштування і розширення під специфічні потреби клієнтів завдяки додатковим платним модулям та різним режимам валідації. 3. Доступність — відкритий код і доступність для широкого кола користувачів через GitHub, що знижує бар'єри для впровадження та сприяє створенню активної спільноти.

4.6 Маркетингова програма стартап-проєкту

Таблиця 4.21

Визначення ключових переваг концепції потенційного товару

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, які можна створити)
Гнучка та надійна валідація даних у великих і розподілених системах	Можливість налаштування валідації під специфічні вимоги, вибір між fail-fast та accumulate режимами, підтримка синхронної та асинхронної валідації	Унікальна гнучкість: Більшість конкурентів не мають таких можливостей налаштування і підтримки різних режимів, що робить Fields універсальним рішенням для різних систем.
Додаткові функції для складних валідаційних сценаріїв	Розширення функціоналу через платні модулі, такі як генерація схем та автоматичне налаштування валідації	Модульність та розширюваність: Можливість додавати платні модулі дозволяє компаніям використовувати продукт саме в тій конфігурації, яка найкраще підходить для їхніх потреб.
Якісна документація та підтримка	Детальна документація, доступна на GitHub, та можливість отримати технічну підтримку або консультації	Висока якість підтримки: Більшість конкурентів не пропонують гнучкої підтримки та консультацій, що робить Fields більш привабливим для великих компаній і проєктів.
Відкритий доступ для швидкого впровадження та розширення спільноти	Open-source модель на GitHub, що дає можливість безкоштовного доступу та внеску від спільноти розробників	Відкритість та активна спільнота: Наявність open-source підходу сприяє зростанню спільноти користувачів і швидкому впровадженню, що стимулює покращення та підтримку продукту.
Стабільність та актуальність продукту на нових версіях Scala	Постійна підтримка нових версій Scala, що забезпечує стабільну роботу продукту для сучасних проєктів	Адаптивність до змін: Швидке оновлення продукту для підтримки нових версій Scala робить Fields актуальним і надійним вибором для розробників, які працюють з останніми технологіями.

Таблиця 4.22

Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Бібліотека для гнучкої та комплексної валідації даних на мові Scala		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Гнучкість налаштування	Нм	Тх
	Надійність	Нм	Е
	Підтримка accumulate та fail-fast	Нм	Тх
	Інформативні помилки з джерелом їх виникнення	Нм	Е
	Відсутність зайвої складності в інтеграції	Нм	Ор
	Підтримка різних валідаційних структур	Нм	Тх
	Підтримка різних середовищ виконання	Нм	Тх
	Конфігуруваня типу помилки	Нм	Тх
	Відсутність залежностей та підтримка різних версій Scala	Нм	Тл
	Генерація валідаційних схем	Нм	Тх
	Платні модулі для розширення функціоналу	Нм	Тх
	Open source бібліотека	Нм	Ор
	Якість: програмне забезпечення має відповідати стандарту ISO/IEC 25010		
	Пакування: бібліотека з відкритим кодом на GitHub		
	Марка: Fields		
III. Товар із підкріпленням	До продажу: стабільність, доступність на GitHub		
	Після продажу: технічна підтримка, консультації		
Захист від копіювання: за рахунок відкритого коду, ліцензії, патенту, активної спільноти, інноваційного функціоналу та додаткових платних модулів			

Таблиця 4.23

Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Великі IT-компанії, що використовують Scala - Прагнуть стабільних, надійних рішень для комплексної валідації даних у великих системах. Віддають перевагу продуктам, що надають технічну підтримку та платні додаткові функції.	Професійні конференції, спеціалізовані IT-видання, LinkedIn, вебінари	Надійність, підтримка, гнучкість налаштування	Донести, що Fields є надійним і гнучким рішенням, яке легко інтегрується у великі системи та пропонує технічну підтримку	“Fields – стабільність і надійність для вашого проекту”: Акцент на надійності продукту, що витримує великі навантаження та пропонує професійну підтримку
Стартапи та середні компанії на Scala - Шукають економічно вигідні рішення з широкими можливостями налаштування, щоб підтримувати швидке зростання. Готові використовувати open-source інструменти та звертають увагу на документацію та легкість інтеграції.	GitHub, форуми для розробників (Stack Overflow, Reddit), Medium, соціальні мережі (Twitter, LinkedIn)	Гнучкість, доступність, open-source	Залучити увагу компаній, які прагнуть швидкої та зручної інтеграції за рахунок open-source доступу	“Fields – копмлексна валідація даних у вашому розпорядженні”: Підкреслення доступності на GitHub, простоти інтеграції та гнучкості налаштування

Таблиця 4.23 (продовження)

Концепція маркетингових комунікацій

1	2	3	4	5
Фрілансери та розробники невеликих проєктів на Scala - Цінують простоту використання та безкоштовний доступ. Шукають інструменти з активною спільнотою та зручними прикладами.	GitHub, YouTube (для відеоуроків), форуми для розробників, Medium	Доступність, спільнота, простота інтеграції	Залучити нових користувачів до спільноти та підкреслити, що Fields легко використовувати у невеликих проєктах	“Fields – безкоштовний інструмент для вашого проекту”: Акцент на доступності, підтримці спільноти та простоті використання
IT-консультанти та інтегратори для корпоративного сектору - Цінують модульність та можливість консультаційної підтримки для впровадження в корпоративні проєкти.	Професійні конференції, LinkedIn, технічні вебінари, блоги, спеціалізовані форуми	Модульність, технічна підтримка, розширені можливості	Донести, що Fields підтримує складні корпоративні рішення завдяки модульності та професійній підтримці	“Fields – модульність і підтримка для складних проектів”: Підкреслення наявності платних модулів і консультаційної підтримки для корпоративних клієнтів

4.7 Виробничий план

Таблиця 4.24

Календарний план-графік реалізації стартап-проєкту

№ п/п	Етапи реалізації	0-ий рік, 1-ий кв.	0-ий рік, 2-ий кв.	0-ий рік, 3-ій кв.	0-ий рік, 4-ий кв.	1-ий рік, 1-ий кв.	1-ий рік, 2-ий кв.	1-ий рік, 3-ій кв.	1-ий рік, 4-ий кв.	2-ий рік	3-ій рік
1	Дослідження ринку та аналіз конкурентів	●	●								
2	Розробка концепції та базової архітектури продукту	●	●								
3	Розробка основних функціональних модулів продукту		●	●	●						
4	Тестування та відлагодження основної версії продукту			●	●						
5	Запуск open-source версії на GitHub				●						
6	Розробка платних модулів (генерація схем, підтримка)					●	●				
7	Розробка документації та прикладів використання				●	●					

Таблиця 4.24 (продовження)

Календарний план-графік реалізації стартап-проєкту

1	2	3	4	5	6	7	8	9	10	11	12
8	Запуск маркетингової кампанії для просування на GitHub					●					
9	Налагодження партнерств із великими компаніями						●	●			
10	Запуск платних модулів та монетизація							●			
11	Розширення маркетингової кампанії на міжнародний ринок								●	●	
12	Підтримка та оновлення продукту, адаптація під нові версії Scala									●	●
13	Розвиток спільноти користувачів та підтримка на GitHub					●	●	●	●	●	●

Таблиця 4.25

Планова вартість матеріальних активів

№ п/п	Вид обладнання (устаткування, пристрою)	Тип (модель)	Виробник обладнання (устаткування, пристрою)	Термін постачання	Вартість за 1 шт., грн.	Кількість, шт.
1	Ноутбук	MacBook Air 13” M1	Apple	7 днів	28 208	3
2	Ноутбук	MacBook Pro 16” M3	Apple	7 днів	76 878	2
3	Програмне забезпечення	Intellij IDEA	Jetbrains	1 день	4 185	5
				Всього	259 305	

Таблиця 4.26

Планова вартість нематеріальних активів

№ п/п	Вид активів	Активи, що можуть бути віднесені до даного типу	Вартість, грн.
1	Авторське право та суміжні з ним права	Право власності на код програмного забезпечення, що стосується продукту	4 000
2	Право інтелектуальної власності	Торгова марка Fields	8 000
		Всього	12 000

Таблиця 4.27

Плановий обсяг виробництва продукції стартап-проєкту

Вид продукції	Одиниця виміру	Обсяг виробництва за період		
		1-й рік	2-й рік	3-й рік
Програмне забезпечення	Підписка	135	1337	5690

Таблиця 4.28

Планова потреба та витрати на персонал

№	Категорія персоналу	Чисельність	Заробітна плата, грн/міс.	Відрахування на соціальні заходи, грн/міс.	Виплати на оплату праці за період, грн.		
					1-й рік	2-й рік	3-й рік
1	Scala розробник	2	70 000	5 062	900 744	900 744	900 744
2	Маркетолог	1	40 000	3 562	522 744	522 744	522 744
3	Технічний	1	30 000	3 062	396 744	396 744	396 744
4	Проектний менеджер	1	60 000	4 562	774 744	774 744	774 744
Разом		5	260 000	16 248	3 495 720	3 495 720	3 495 720

Соціальні заходи рахуються, як ЄСВ + 5% від зарплати, так як усі співробітники компанії працюють, як ФОПи. Мінімальний внесок ЄСВ на момент написання роботи становить 1562,00 грн.

Таблиця 4.29

Загальні початкові витрати проєкту

№ п/п	Стаття витрат	Обсяги витрат в 0-ий рік, грн.
1	Проведення науково-дослідницьких робіт	1 300
2	Розробка ТЗ і ТЕО	-
3	Робоче проєктування та тестування	Входить у витрати персоналу
4	Витрати на придбання обладнання	259 305
5	Витрати на інфраструктуру	1120
6	Витрати на придбання нематеріальних активів	12 000
7	Витрати на передвиробничі маркетингові дослідження і створення збутної мережі	-
8	Витрати, пов’язані з діяльністю персоналу	3 495 720
	Разом	3 769 445

Таблиця 4.30

Планові загальногосподарські витрати

№ п/п	Стаття витрат	Витрати за період, тис. грн.		
		1-й рік	2-й рік	3-й рік
1	Витрати на інфраструктуру	1 120	2 120	4 532
2	Витрати на персонал (на відрядження, соціальні заходи тощо)	3 495 720	3 495 720	3 495 720
3	Витрати на просування та рекламу	46 000	65 000	75 000
	Разом	3 542 840	3 562 840	3 575 252

Згідно проведених оцінок та підрахунків для того щоб перетворити open-source бібліотеку fields е повноцінний продукт – стартап доведеться витратити чимало ресурсів, проте згідно проведеного аналізу, з великою вірогідністю прибуток від реалізації проєкту перекриє витрати за рахунок чого реалізація стартапу вважається успішною та доцільною.

ВИСНОВКИ ДО РОЗДІЛУ 4

У цьому розділі було розроблено стартап-проєкт бібліотеки для валідації даних на мові Scala під назвою Fields та проведено його маркетинговий аналіз. Було створено інформаційну картку проєкту, сформовано опис необхідної команди проєкту, сформульовано основні ідеї, проаналізовано ринкові можливості, розроблено ринкову стратегію, маркетингову програму та сформовано виробничий план.

В ході аналізу було визначено необхідність у команді з 5 людей для реалізації проєкту. Серед запропонованих альтернатив впровадження стартап-проєкту було обрано наступну: запуск проєкту у форматі open-source на платформі GitHub, розробка платних модулів для комерційного використання та налагодження партнерських відносин з великими компаніями. Цільовими групами проєкту були визначені великі IT-компанії, стартапи та фрілансери, які працюють з великими обсягами даних та потребують гнучкої валідації. Стратегією охоплення ринку було обрано стратегію нішевого маркетингу, базовою стратегією було обрано стратегію диференціації, а стратегію конкурентної поведінки також було визначено як стратегію диференціації.

Було визначено ключові характеристики, які вигідно вирізнятимуть Fields серед конкурентів і стимулюватимуть зацікавленість у продукті. Серед них:

- Гнучка настройка валідації даних із підтримкою fail-fast та accumulate режимів;
- Конфігурованість середовища виконання, валідаційної структури та типу помилки
- Можливість використання додаткових платних модулів, таких як генерація схем для валідації;
- Доступність open-source версії для широкого кола розробників.

Обчислено, що початкові витрати проєкту в нульовий рік становитимуть 3 769 445 грн, в 1-ий рік - 3 562 840 грн, в 2-ий рік - 3 562 840 грн, а в 3-ій рік - 3 575 252 грн.

ВИСНОВКИ

Магістерська дисертація присвячена розробці методу валідації комплексних даних та його програмній реалізації у вигляді бібліотеки для мови програмування Scala.

В рамках виконання роботи було оглянуто рішення-аналоги, прицнипи їх роботи, переваги та недоліки. Розглянувши детально всі властивості аналогів було виділено головні риси, які має включати в себе рішення створене в рамках цієї роботи з врахуванням усіх нюансів та застосуванням великої кількостій різноманітних підходів задля досягнення поставлених цілей.

Було оглянуто ряд існуючих підходів, які було адаптовано під проблематику методу валідації комплексних даних, такі як: тайпкласи, теговані типи, метапрограмування та макроси, розробка DSL, кроскомпіляція, Tagless Final та інші.

Всі рішення було детально розписано і аргументовано чому та навіщо, для перевірки коректності роботи бібліотека була покрита тестами та було створено синтетичні тести на JMH для порівняння швидкодії бібліотеки.

Унікальний метод до валідації даних дає можливість гнучко конфігурувати бібліотеку під потреби користувача на етапі компіляції, зменшує ризики помилитись при написанні коду та дає можливість писати комплесну валідаційну логіку у простому та лаконічному вигляді за допомогою потужного DSL. При цьому користувач може модифікувати типи помилки під свої потреби та навіть повертати помилки з інформацією про місце її виникнення.

В рамках розробки бібліотеки для реалізації методу, також подбали про зручність її використання та швидкість вивчення за рахунок створення сайту з документацією, Scaladoc, було надано широкий спектр різноманітних прикладів, написано статтю на зарубіжний ресурси Medium та пост на Scala сторінці на ресурсі Reddit. Де спільнота радо прийняла бібліотеку та відзначила це гарним рейтингом на github сторінці бібліотеки, туди долучились навіть автори бібліотек конкурентів, що говорить про визнання ними нашого методу та самої бібліотеки.

Створений метод комплексної валідації даних за рахунок усієї проробленої роботи є чудовим вибором для написання валідації різної складності і може бути застосованим на реальних підприємствах та різноманітних проєктах включаючи проекти військової сфери, медицини, логістики, фінансів, освіти, сільського господарства, енергетики, екології, державного управління.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Functional Programming in Scala / P. Chiusano, R. Bjarnason. – Washington, Manning Publications, 2014. – 320 p
2. sbt [Електронний ресурс] – Режим доступу до ресурсу: https://www.scala-sbt.org/
3. Scala [Електронний ресурс] – Режим доступу до ресурсу: https://www.scala-lang.org/
4. Dupin [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/yakivy/dupin
5. Octopus [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/krzemin/octopus
6. Accord [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/wix-incubator/accord
7. Tagless Final [Електронний ресурс] – Режим доступу до ресурсу: https://degoes.net/articles/tagless-horror
8. ZIO [Електронний ресурс] – Режим доступу до ресурсу: https://zio.dev/
9. Cats [Електронний ресурс] – Режим доступу до ресурсу: https://typelevel.org/cats/
10. Cats Effect [Електронний ресурс] – Режим доступу до ресурсу: https://typelevel.org/cats-effect/
11. Programming Scala, 2nd Edition / D. Wampler, A. Payne – O'Reilly Media, 2014 – 583 p
12. Learning Concurrent Programming in Scala / A. Prokopec – Packt Publishing, 2017. – 368 p
13. JMH [Електронний ресурс] – Режим доступу до ресурсу: https://openjdk.org/projects/code-tools/jmh/
14. mdoc [Електронний ресурс] – Режим доступу до ресурсу: https://scalameta.org/mdoc/

15. Scaladoc [Електронний ресурс] – Режим доступу до ресурсу: https://docs.scala-lang.org/style/scaladoc.html
16. React [Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/
17. Docusaurus [Електронний ресурс] – Режим доступу до ресурсу: https://docusaurus.io/
18. Fields [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/jap-company/fields
19. Medium: Fields — Scala Validation Library [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@oleh.dubynskiy/fields-scala-validation-library-86ac818cd704
20. Reddit: Fields — Scala Validation Library Library [Електронний ресурс] – Режим доступу до ресурсу: https://www.reddit.com/r/scala/comments/vm3a7n/fields_scala_validation_library/

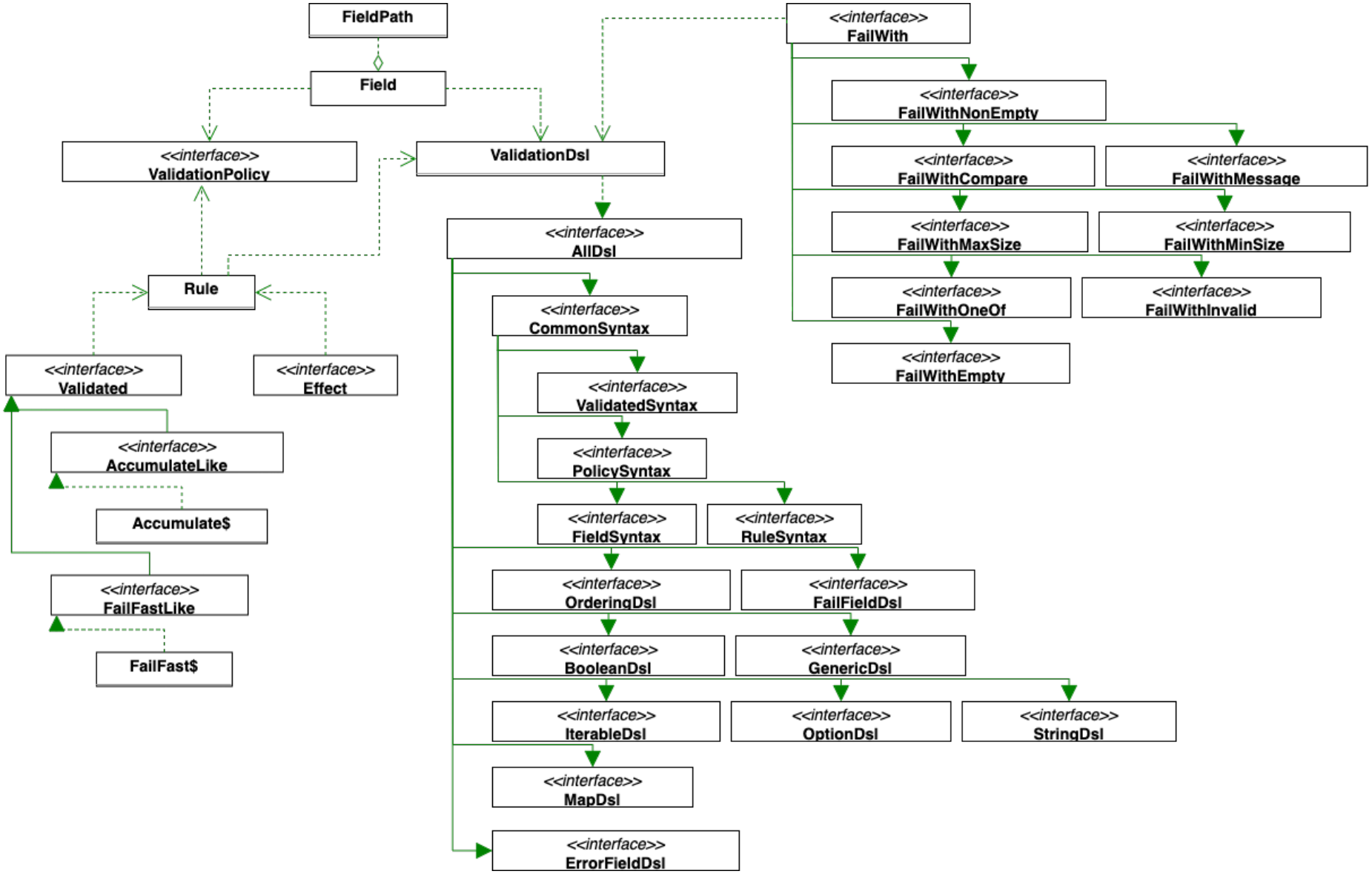
ДОДАТОК А

Бібліотека для комплексної валідації даних

Діграма класів бібліотеки

Аркушів 1

Київ – 2024


FieldPath
Field
<<interface>>
ValidationPolicy
ValidationDsl
<<interface>>
FailWith
<<interface>>
FailWithNonEmpty
<<interface>>
FailWithCompare
<<interface>>
FailWithMessage
<<interface>>
FailWithMaxSize
<<interface>>
FailWithMinSize
<<interface>>
FailWithOneOf
<<interface>>
FailWithInvalid
<<interface>>
FailWithEmpty
<<interface>>
AllDsl
Rule
<<interface>>
CommonSyntax
<<interface>>
Validated
<<interface>>
Effect
<<interface>>
ValidatedSyntax
<<interface>>
AccumulateLike
<<interface>>
PolicySyntax
Accumulate$
<<interface>>
FieldSyntax
<<interface>>
RuleSyntax
<<interface>>
FailFastLike
<<interface>>
OrderingDsl
<<interface>>
FailFieldDsl
<<interface>>
BooleanDsl
<<interface>>
GenericDsl
FailFast$
<<interface>>
IterableDsl
<<interface>>
OptionDsl
<<interface>>
StringDsl
<<interface>>
MapDsl
<<interface>>
ErrorFieldDsl

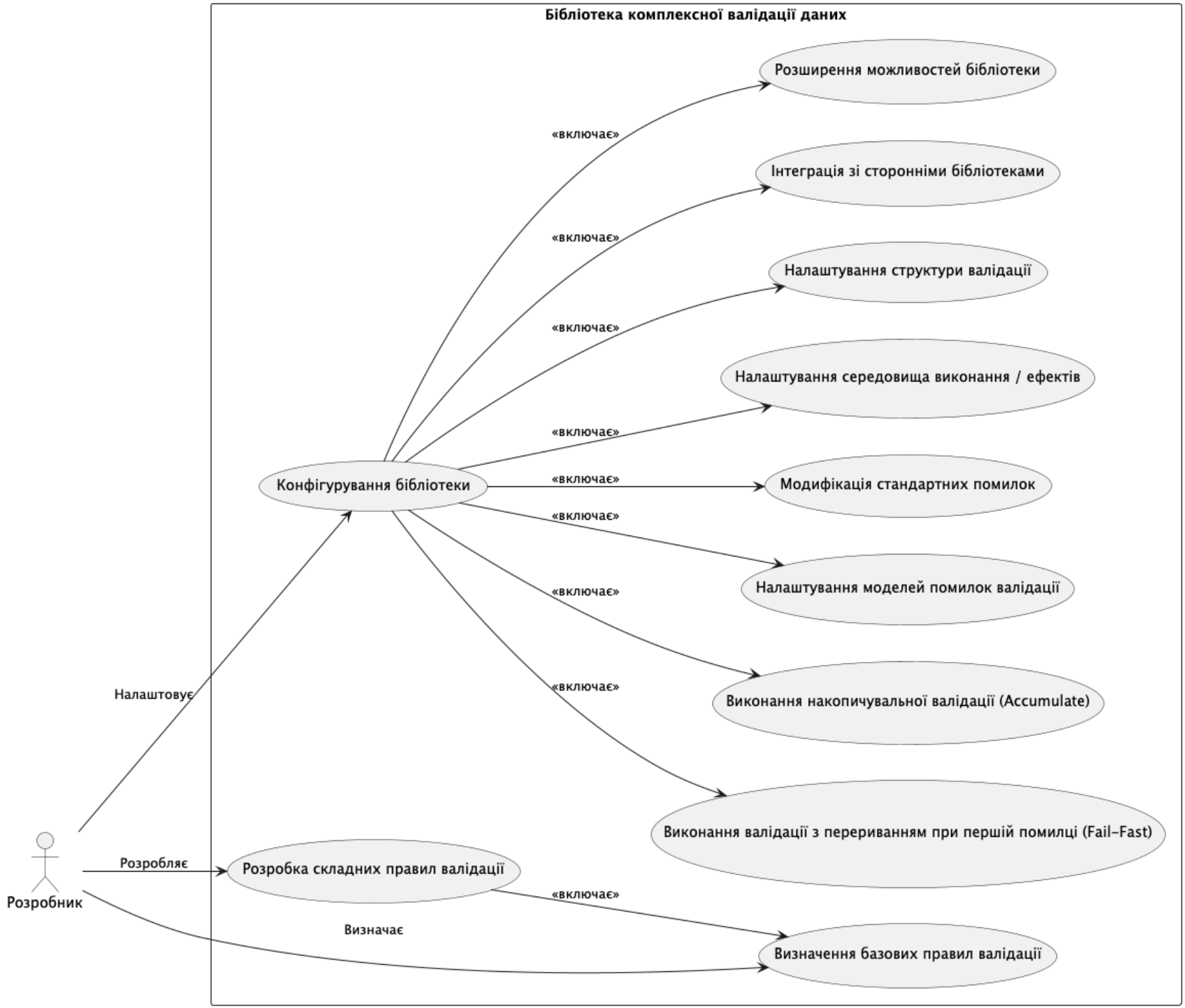
ДОДАТОК Б

Бібліотека для комплексної валідації даних

Діграма прецендетів бібліотеки

Аркушів 1

Київ – 2024


Бібліотека комплексної валідації даних
Розширення можливостей бібліотеки
Інтеграція зі сторонніми бібліотеками
Налаштування структури валідації
Налаштування середовища виконання / ефектів
Конфігурування бібліотеки
Модифікація стандартних помилок
Налаштування моделей помилок валідації
Виконання накопичувальної валідації (Accumulate)
Виконання валідації з перериванням при першій помилці (Fail-Fast)
«включає»
«включає»
«включає»
«включає»
«включає»
«включає»
«включає»
«включає»
Налаштовує
Розробник
Розробляє
Розробка складних правил валідації
Визначає
Визначення базових правил валідації

ДОДАТОК В

Бібліотека для комплексної валідації даних

Лістинг бібліотеки

Аркушів 20

Київ – 2024

Field.scala

```
package jap.fields

/** `Field` is heart of the library and contains [[jap.fields.FieldPath]] and its value */
final case class Field[+P](path: FieldPath, value: P) {

  /** Returns [[jap.fields.FieldPath.name]] of `path` */
  def name = path.name

  /** Returns [[jap.fields.FieldPath.full]] of `path` */
  def fullPath = path.full

  /** Creates new `Field` with provided `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](subPath: FieldPath, value: S): Field[S] = Field(path ++ subPath, value)

  /** Creates new `Field` with provided `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](part: FieldPart, value: S): Field[S] = Field(path.down(part), value)

  /** Creates new `Field` with provided `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](subPath: String, value: S): Field[S] = Field(path.down(subPath), value)

  /** Creates new `Field` with provided `value` and `index`. Appends `subIindex` to this field `path` */
  def downN[S](subIndex: Int, value: S): Field[S] = Field(path.down(subIndex), value)

  /** Creates new `Field` with selected `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](subPath: String, selector: P => S): Field[S] = down(subPath, selector(value))

  /** Creates new `Field` with provided `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](part: FieldPart, selector: P => S): Field[S] = down(part, selector(value))

  /** Creates new `Field` with provided `value` and `subPath`. Appends `subPath` to this field `path` */
  def down[S](subPath: FieldPath, selector: P => S): Field[S] = down(subPath, selector(value))

  /** Creates new `Field` with selected `value` and `subPath`. Appends `subPath` to this field `path` */
  def downN[S](subIndex: Int, selector: P => S): Field[S] = downN(subIndex, selector(value))

  /** Renames this `Field`. Changes last `path` part aka name. */
  def named(name: String): Field[P] = withPath(path.named(name))

  /** Change `path` */
  def withPath(newPath: FieldPath): Field[P] = copy(path = newPath)

  /** Change `value` */
  def withValue[V](newValue: V): Field[V] = copy(value = newValue)

  /** Maps `value` */
  def map[B](f: P => B): Field[B] = withValue(f(value))

  /** Maps `path` */
  def mapPath(f: FieldPath => FieldPath): Field[P] = withPath(f(this.path))

  /** Gets first tuple element of `value`. Given `P` is Tuple */
  def first[P1, P2](implicit ev: P <:< (P1, P2)): Field[P1] = map(ev(_)._1)

  /** Gets second tuple element of `value`. Given `P` is Tuple */
  def second[P1, P2](implicit ev: P <:< (P1, P2)): Field[P2] = map(ev(_)._2)

  /** Turns Option on `value` into Option on `jap.fields.Field`. Given `P` i `Option[V]` */
  def option[V](implicit ev: P <:< Option[V]): Option[Field[V]] = ev(value).map(withValue)

  override def toString = fullPath + ":" + value
}

object Field {

  /** Create `Field` with given `value` and [[FieldPath.Root]] `path` */
  def apply[P](value: P): Field[P] = new Field(FieldPath.Root, value)
}

object FieldConversions {

  /** Converts Field[A] to Field[B] using `conversion` function */
  implicit def fieldConversion[A, B](field: Field[A])(implicit conversion: A => B): Field[B] = field.map(conversion)

  /** Converts `field` into its value */
  implicit def fieldToValue[A](field: Field[A]): A = field.value
}
```

FieldPath.scala

```
package jap.fields

/** `FieldPath` contains path parts of the Field. */
final case class FieldPath(parts: List[FieldPart]) extends AnyVal {

  /** Returns list of parts names */
  def names: List[String] = parts.map(_.name)

  /** Is current path root. */
  def isRoot: Boolean = parts.isEmpty

  /** Full name of the path of this path. For root path this will be "." */
  def full: String =
    parts.foldLeft(FieldPath.RootName) {
      case (FieldPath.RootName, FieldPart.Path(path)) => FieldPath.RootName + path
      case (acc, FieldPart.Path(path))                => acc + '.' + path
      case (acc, FieldPart.Key(path))                 => acc + '[' + path + ']'
      case (acc, FieldPart.Index(index))              => acc + '[' + index + ']'
    }

  /** Name of the path is the last part of path. */
  def name: String = parts.lastOption.map(_.name).getOrElse(FieldPath.RootName)

  /** Changes name of this path */
  def named(name: String): FieldPath = FieldPath(parts.dropRight(1) :+ FieldPart.Path(name))

  /** Append other `FieldPath` to current path */
  def ++(path: FieldPath): FieldPath = FieldPath(parts ++ path.parts)

  /** Append other path part to current path */
  def +(path: String): FieldPath = down(path)

  /** Append other index part to current path */
  def +(index: Int): FieldPath = down(index)

  /** Append other part to current path */
  def +(part: FieldPart): FieldPath = down(part)

  /** Append other part to current path */
  def down(part: FieldPart): FieldPath = FieldPath(parts :+ part)

  /** Append other path part to current path */
  def down(path: String): FieldPath = FieldPath(parts :+ FieldPart.Path(path))

  /** Append other index part to current path */
  def down(index: Int): FieldPath = FieldPath(parts :+ FieldPart.Index(index))

  /** Shows `FieldPath.full` */
  override def toString: String = full
}

object FieldPath {

  /** Name of the FieldPath that has empty `parts` */
  val RootName = "."

  /** Root FieldPath */
  val Root = FieldPath()

  /** Create `FieldPath` from Path `parts` */
  def fromPaths(parts: List[String]): FieldPath = FieldPath(parts.map(FieldPart.Path(_)))

  /** Create `FieldPath` from Path `parts` */
  def fromPaths(parts: String*): FieldPath = fromPaths(parts.toList)

  /** Create `FieldPath` from `parts` VarArgs */
  def apply(parts: FieldPart*): FieldPath = FieldPath(parts.toList)

  /** Create `FieldPath` from `String` */
  def fromPath(path: String): FieldPath = FieldPath(FieldPart.Path(path))

  /** Create `FieldPath` from `Int` */
  def fromIndex(index: Int): FieldPath = FieldPath(FieldPart.Index(index))

  /** Create `FieldPath` from [[jap.fields.Field]] */
  def fromField[P](f: Field[P]): FieldPath = f.path
```

```
  val IndexRegex = "\\[(\\d+)\\]".r

  /** Parse `FieldPath` from dot-separated `path` string */
  def parse(path: String) = FieldPath(
    path
      .split('.')
      .flatMap { part =>
        if (part.isEmpty) Nil
        else
          FieldPart.Path(part.takeWhile(_ != '[')) +: IndexRegex
            .findAllMatchIn(part)
            .map(i => FieldPart.Index(i.group(1).toInt))
            .toList
      }
      .toList
  )
}

sealed abstract class FieldPart(val name: String)
object FieldPart {
  case class Key(value: String)  extends FieldPart(value)
  case class Path(value: String) extends FieldPart(value)
  case class Index(value: Int)   extends FieldPart(value.toString)
}

object FieldPartConversions {

  /** Conversion from `String` to [[jap.fields.FieldPart.Path]] */
  implicit def fromPath(path: String): FieldPart = FieldPart.Path(path)

  /** Conversion from `String` to [[jap.fields.FieldPart.Index]] */
  implicit def fromIndex(index: Int): FieldPart = FieldPart.Index(index)
}

object FieldPathConversions {

  /** Conversion for `FieldPath.fromPath` */
  implicit def fromPath(path: String): FieldPath = FieldPath.fromPath(path)

  /** Conversion for `FieldPath.fromIndex` */
  implicit def fromIndex(index: Int): FieldPath = FieldPath.fromIndex(index)

  /** Conversion for `FieldPath.apply` */
  implicit def fromParts(parts: List[FieldPart]): FieldPath = FieldPath(parts)

  /** Conversion for `FieldPath.fromPaths` */
  implicit def fromPaths(parts: List[String]): FieldPath = FieldPath.fromPaths(parts)

  /** Conversion for `FieldPath.fromField` */
  implicit def fromField[P](f: Field[P]): FieldPath = FieldPath.fromField(f)
}
```

Rule.scala

```
package jap.fields

object Rule {
  // ----TAGGED---- //
  trait Tag[+F[_], +V[_], +E] extends Any
  type Base[+F[_], +V[_], +E] = Any { type __Rule__ }
  type Type[+F[_], +V[_], +E] <: Base[F, V, E] with Tag[F, V, E]

  /** Same as [[Rule.wrap]] */
  @inline def apply[F[_], V[_], E](effect: F[V[E]]): Rule[F, V, E] = wrap(effect)

  /** Wraps `rule` into tagged type */
  @inline def wrap[F[_], V[_], E](effect: F[V[E]]): Rule[F, V, E] = effect.asInstanceOf[Rule[F, V, E]]

  /** Unwraps `rule` from tagged type */
  @inline def unwrap[F[_], V[_], E](rule: Rule[F, V, E]): F[V[E]] = rule.asInstanceOf[F[V[E]]]
  // ----TAGGED---- //

  implicit final class RuleExtensions[F[_], V[_], E](private val rule: Rule[F, V, E]) extends AnyVal {

    /** Unwraps `rule` to its actual type */
    @inline def unwrap: F[V[E]] = Rule.unwrap(rule)

    /** Alias for [[unwrap]] */
    @inline def effect: F[V[E]] = Rule.unwrap(rule)
```

```
    /** Combines `rule` and result of running `f` on `rule` using [[Rule.and]]. This is tricky syntax for usage with
      * for-comprehension. Check example in [[flatMap]]
      */
    @inline def map(f: V[E] => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
      R.and(rule, R.modify(rule)(f))

    /** Combines `rule` and result of running `f` on `rule` using [[Rule.and]]. This is tricky syntax for usage with
      * for-comprehension yield should always return valid Validated if you want this to work correctly.
      *
      * {{{
      * scala> import jap.fields._
      * scala> import jap.fields.DefaultAccumulateDsl._
      * scala> val intF = Field(4)
      * val intF: jap.fields.Field[Int] = root:4
      * scala> for {
      *      | _ <- intF > 4
      *      | _ <- intF < 4
      *      | _ <- intF !== 4
      *      | } yield V.valid
      * val res0:
      *   jap.fields.Rule[[A] =>> A, jap.fields.typeclass.Validated.Accumulate,
      *     jap.fields.ValidationError
      *   ] = Invalid(List(root -> must be greater than 4, root -> must be less than 4, root -> must not be equal to 4))
      * }}}
      */
    @inline def flatMap(f: V[E] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
      R.and(rule, R.modifyM(rule)(f))

    @inline def mapK[FF[_]](f: F[V[E]] => FF[V[E]]): Rule[FF, V, E] = Rule(f(effect))
  }
}
```

ValidationDsl.scala

```
package jap.fields

import scala.annotation.implicitNotFound

import syntax._
import typeclass._
import data._
import error._

/** FailFast [[jap.fields.ValidationDsl]] */
abstract class FailFastDsl[F[_]: Effect, E] extends ValidationDsl[F, FailFast, E]

/** Accumulating [[jap.fields.ValidationDsl]] */
abstract class AccumulateDsl[F[_]: Effect, E] extends ValidationDsl[F, Accumulate, E]

/** Default ValidationDsl where:
  *   - Effect is [[jap.fields.typeclass.Effect.Sync]]
  *   - Validated is Accumulate
  *   - Error is ValidationError
  */
trait DefaultAccumulateDsl  extends AccumulateDsl[Effect.Sync, ValidationError]
object DefaultAccumulateDsl extends DefaultAccumulateDsl

/** Default ValidationDsl where:
  *   - Effect is [[jap.fields.typeclass.Effect.Sync]]
  *   - Validated is FailFast
  *   - Error is ValidationError
  */
trait DefaultFailFastDsl  extends FailFastDsl[Effect.Sync, ValidationError]
object DefaultFailFastDsl extends DefaultFailFastDsl

/** God object that provides all validation syntax for choosen Effect - F[_], Validated - V[_] and Error - E Requires
  * user to provide implicit instances of Effect and Validated typeclasses for choosen F[_] and V[_].
  */
@implicitNotFound("ValidationDsl[${F}, ${V}, ${E}] not found")
class ValidationDsl[F[_], V[_], E](implicit
    val F: Effect[F],
    val V: Validated[V],
) extends AllDsl[F, V, E] { module =>
  implicit val R: RuleOps[F, V, E] = RuleOps.instance(F, V)

  /** [[jap.fields.ValidationPolicy]] alias. Infers F, V, E */
  type Policy[P] = ValidationPolicy[P, F, V, E]
```

```
  // /** [[jap.fields.ValidationSchema]] alias. Infers F, V, E */
  // type Schema[P] = ValidationSchema[P, F, V, E]

  /** [[jap.fields.ValidationPolicy]] alias. Infers F, V, E */
  type MPolicy[P] = Policy[P]
  val MPolicy = Policy

  /** [[jap.fields.ValidationPolicyBuilder]] alias. Infers F, V, E */
  type PolicyBuilder[P] = ValidationPolicyBuilder[P, F, V, E]

  /** [[jap.fields.ValidationPolicyBuilder]] alias. Infers F, V, E */
  type MPolicyBuilder[P] = PolicyBuilder[P]

  object Policy {

    /** See [[ValidationPolicy.builder]] */
    def builder[P]: PolicyBuilder[P] = ValidationPolicy.builder
  }

  final type MRule = Rule[F, V, E]
}
```

ValidationPolicy.scala

```
package jap.fields

import typeclass.*

/** Typeclass that defines how to validate given field */
trait ValidationPolicy[P, F[_], V[_], E] extends (Field[P] => Rule[F, V, E]) { self =>
  def validate(value: P): Rule[F, V, E] = validate(Field(value))

  def validate(field: Field[P]): Rule[F, V, E]

  def apply(field: Field[P]): Rule[F, V, E] = validate(field)

  /** Validates `field` using Policy and returns it as Either.Left if invalid and Either.Right if valid */
  def validateEither(field: Field[P])(implicit R: RuleOps[F, V, E], E: HasErrors[V]): F[Either[List[E], P]] =
    R.F.map(validate(field).effect) { v =>
      if (R.V.isValid(v)) Right(field.value)
      else Left(E.errors(v))
    }

  def and(other: ValidationPolicy[P, F, V, E])(implicit R: RuleOps[F, V, E]): ValidationPolicy[P, F, V, E] =
    field => R.and(self(field), other(field))

  def or(other: ValidationPolicy[P, F, V, E])(implicit R: RuleOps[F, V, E]): ValidationPolicy[P, F, V, E] =
    field => R.or(self(field), other(field))
}

object ValidationPolicy {
  def builder[P, F[_], V[_], E](implicit R: RuleOps[F, V, E]): ValidationPolicyBuilder[P, F, V, E] =
    ValidationPolicyBuilder(_ => R.valid)
}

/** Builder class for [[jap.fields.ValidationPolicy]]. [[jap.fields.ValidationDsl]] implicit should be in scope */
case class ValidationPolicyBuilder[P, F[_], V[_], E](policy: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]) {
  type FnRule[A]       = Field[A] => Rule[F, V, E]
  type FnRule2[A, B]   = (Field[A], Field[B]) => Rule[F, V, E]
  type FnRule3[A, B, C] = (Field[A], Field[B], Field[C]) => Rule[F, V, E]

  def combineRuleFn[T](empty: FnRule[T], rules: List[FnRule[T]]): FnRule[T] =
    FoldUtil.fold[FnRule[T]](rules, empty, (r1, r2) => field => R.and(r1(field), r2(field)))

  def combineRuleFn2[A, B](empty: FnRule2[A, B], rules: List[FnRule2[A, B]]): FnRule2[A, B] =
    FoldUtil.fold[FnRule2[A, B]](rules, empty, (r1, r2) => (a, b) => R.and(r1(a, b), r2(a, b)))

  def combineRuleFn3[A, B, C](empty: FnRule3[A, B, C], rules: List[FnRule3[A, B, C]]): FnRule3[A, B, C] =
    FoldUtil.fold[FnRule3[A, B, C]](rules, empty, (r1, r2) => (a, b, c) => R.and(r1(a, b, c), r2(a, b, c)))

  /** Adds new rule to builder */
  def rule(rule: FnRule[P]*): ValidationPolicyBuilder[P, F, V, E] =
    copy(policy = combineRuleFn(policy, policy :: rule.toList))

  /** Adds new sub field rule to builder. First extracts sub field using `sub` function and then applies all `rules` to
    * it
    */
  def fieldRule[S](sub: Field[P] => Field[S])(rules: FnRule[S]*) = {
    val subRule = combineRuleFn[S](_ => R.valid, rules.toList)
```

```
  rule(f => subRule(sub(f)))
 }

 /** Same as `fieldRule` but for 2 sub fields */
 def fieldRule[A, B](
     sub1: Field[P] => Field[A],
     sub2: Field[P] => Field[B],
 )(rules: FnRule2[A, B]*) = {
  val subRule = combineRuleFn2[A, B]((_, _) => R.valid, rules.toList)
  rule(f => subRule(sub1(f), sub2(f)))
 }

 /** Same as `fieldRule` but for 3 sub fields */
 def fieldRule[A, B, C](
     sub1: Field[P] => Field[A],
     sub2: Field[P] => Field[B],
     sub3: Field[P] => Field[C],
 )(rules: FnRule3[A, B, C]*) = {
  val subRule = combineRuleFn3[A, B, C]((_, _, _) => R.valid, rules.toList)
  rule(f => subRule(sub1(f), sub2(f), sub3(f)))
 }

 /** Applies all validaiton rules to [[jap.fields.Field]]#value */
 def build: ValidationPolicy[P, F, V, E] = policy(_)
}
```

RuleOps.scala

```
package jap.fields

import typeclass._

trait RuleOps[F[_], V[_], E] {
 type R = Rule[F, V, E]

 implicit val F: Effect[F]
 implicit val V: Validated[V]

 /** Lazily converts an `V[E]` to `Rule[F, VR, E]` */
 @inline def pure(validated: => V[E]): R = Rule(F.suspend(validated))

 /** Lazily converts an `F[V[E]]` to a `Rule[F, VR, E]` */
 @inline def effect(effect: => F[V[E]]): R = Rule(F.defer(effect))

 /** Defers Rule */
 @inline def defer(rule: => R): R = Rule(F.defer(rule.unwrap))

 /** Flattens Effect with Rule */
 @inline def flatten(rule: => F[R]): R = effect(F.flatMap(rule)(_.effect))

 /** Converts `E` to a `R` */
 @inline def invalid(error: => E): R = pure(V.invalid(error))

 /** Returns always valid `R` */
 lazy val valid: R = pure(V.valid)

 /** Combines two `R`'s using logical AND. Short-circuits if `Validated.strategy` is
   * [[jap.fields.typeclass.FailFastStrategy]]
   */
 lazy val and: (R, R) => R = V.strategy match {
  case AccumulateStrategy =>
   (ra: R, rb: R) => Rule(F.flatMap(ra.unwrap)(aa => F.map(rb.unwrap)(bb => V.and(aa, bb))))

  case FailFastStrategy =>
   (ra: R, rb: R) => Rule(F.flatMap(ra.unwrap)(aa => if (V.isInvalid(aa)) F.pure(aa) else rb.unwrap))
 }

 /** Combines two `R`'s using logical OR. Short-circuits if first rule is valid */
 @inline def or(ra: R, rb: R): R =
  Rule {
   F.flatMap(ra.unwrap) { aa =>
    if (V.isValid(aa)) F.pure(aa)
    else F.map(rb.unwrap)(bb => V.or(bb, aa))
   }
  }

 /** Applies `rule` only when `test` pass */
 @inline def when(test: => Boolean)(rule: => R): R =
  defer(if (test) rule else valid)
```

```
  /** Applies `rule` only when `test` pass */
  @inline def whenF(test: => F[Boolean])(rule: => R): R =
    effect(F.flatMap(F.defer(test))(if (_) rule.unwrap else valid.unwrap))

  /** Ensures that if `test` pass else returns provided `V[E]` */
  @inline def ensure(v: => V[E])(test: => Boolean): R =
    pure(if (test) V.valid else v)

  /** Ensures that if `test` pass else returns provided `V[E]` */
  @inline def ensureF(v: => V[E])(test: => F[Boolean]): R =
    effect(F.map(F.defer(test))(if (_) V.valid else v))

  /** Asserts that if `test` pass else returns provided `E` */
  @inline def assert(e: => E)(test: => Boolean): R = ensure(V.invalid(e))(test)

  /** Asserts that if `test` pass else returns provided `E` */
  @inline def assertF(e: => E)(test: => F[Boolean]): R = ensureF(V.invalid(e))(test)

  /** Combines all rules using AND */
  @inline def andAll(rules: List[R]): R =
    FoldUtil.fold[R](rules, valid, and)

  /** Combines all rules using OR */
  @inline def orAll(rules: List[R]): R =
    FoldUtil.fold[R](rules, valid, or)

  /** Modifies `rule` Validated value using `f` */
  @inline def modify(rule: R)(f: V[E] => V[E]): R =
    Rule(F.map(rule.effect)(f))

  /** Modifies `rule` Validated value using `f` */
  @inline def modifyM(rule: R)(f: V[E] => R): R =
    Rule(F.flatMap(rule.effect)(f.andThen(_.unwrap)))
}

object RuleOpsConversions {
  implicit def validatedFromRuleOps[F[_], V[_], E](implicit R: RuleOps[F, V, E]): Validated[V] = R.V
  implicit def effectFromRuleOps[F[_], V[_], E](implicit R: RuleOps[F, V, E]): Effect[F] = R.F
}
object RuleOps {

  implicit def instance[F[_], V[_], E](implicit F0: Effect[F], V0: Validated[V]): RuleOps[F, V, E] =
    new RuleOps[F, V, E] {
      implicit val F: Effect[F] = F0
      implicit val V: Validated[V] = V0
    }
}
```

Validated.scala

```
package jap.fields
package typeclass

import data.{Accumulate, FailFast}

/** Base trait for [[jap.fields.typeclass.Validated]] that accumulate errors */
trait AccumulateLike[V[_]] extends Validated[V] { val strategy: ValidatedStrategy = AccumulateStrategy }

/** Base trait for [[jap.fields.typeclass.Validated]] that fail-fast */
trait FailFastLike[V[_]] extends Validated[V] { val strategy: ValidatedStrategy = FailFastStrategy }

/** Typeclass that represents Validated */
trait Validated[V[_]] {

  /** Returns this Strategy */
  def strategy: ValidatedStrategy

  /** Returns valid V[E] */
  def valid[E]: V[E]

  /** Returns invalid V[E] with given error */
  def invalid[E](e: E): V[E]

  /** Checks if `v` is valid */
  def isValid[E](v: V[E]): Boolean

  /** Combines `a` and `b` using AND */
  def and[E](va: V[E], vb: V[E]): V[E]
```

```
  /** Combines all errors in single result */
  def invalidAll[E](list: List[E]): V[E] = traverse(list)(invalid)

  /** Combines all errors in single result */
  def invalidAll[E](list: E*): V[E] = traverse(list.toList)(invalid)

  /** Checks if `v` is invalid */
  def isInvalid[E](v: V[E]): Boolean = !isValid(v)

  /** Combines `a` and `b` using OR */
  def or[E](va: V[E], vb: V[E]): V[E] = if (isValid(va) || isValid(vb)) valid else and(va, vb)

  /** Returns `error` if `v` is invalid */
  def asError[E](v: V[E])(error: E): V[E] = whenInvalid[E](v)(_ => invalid(error))

  /** Returns `invalid` if `v` is invalid */
  def asInvalid[E](v: V[E])(invalid: V[E]): V[E] = whenInvalid[E](v)(_ => invalid)

  /** Returns `o` if `a` is valid else returns `a` */
  def whenValid[E](va: V[E])(vb: => V[E]): V[E] = if (isValid(va)) vb else va

  /** Returns `f` applied to `a` if `a` is invalid else returns `a` */
  def whenInvalid[E](v: V[E])(f: V[E] => V[E]): V[E] = if (isInvalid(v)) f(v) else v

  /** Returns `v` if `cond` is true else returns valid */
  def when[E](cond: Boolean)(v: => V[E]): V[E] = if (cond) v else valid[E]

  /** Returns `v` if `cond` is false else returns valid */
  def unless[E](cond: Boolean)(v: => V[E]): V[E] = if (cond) valid[E] else v

  /** `Validated.andAll` alias */
  def sequence[E](list: List[V[E]]): V[E] = andAll[E](list)

  /** Combiness all `list` using AND */
  def andAll[E](list: List[V[E]]): V[E] = FoldUtil.fold[V[E]](list, valid[E], and[E])

  /** Combiness all `list` using OR */
  def orAll[E](list: List[V[E]]): V[E] = FoldUtil.fold[V[E]](list, valid[E], or[E])

  /** Traverse over `list` applying `f` to each element and combining all using `Validated.sequence` */
  def traverse[A, E](list: List[A])(f: A => V[E]): V[E] = sequence(list.map(f))

  /** Traverse over `list` applying `f` to each element and combining all using `Validated.sequence` */
  def traverse[A, E](list: A*)(f: A => V[E]): V[E] = sequence(list.toList.map(f))
}

object Validated {

  /** Returns [[jap.fields.typeclass.Validated]] instance for given V */
  def apply[V[_]](implicit V: Validated[V]): Validated[V] = V

  implicit object ListValidated extends AccumulateLike[List] {
    def valid[E]: List[E]                       = Nil
    def invalid[E](e: E): List[E]               = List(e)
    def and[E](va: List[E], vb: List[E]): List[E] = va ::: vb
    def isValid[E](v: List[E]): Boolean         = v.isEmpty
  }

  implicit object EitherUnitValidated extends FailFastLike[Either[_, Unit]] {
    def valid[E]: Either[E, Unit]                                   = Right(())
    def invalid[E](e: E): Either[E, Unit]                           = Left(e)
    def and[E](va: Either[E, Unit], vb: Either[E, Unit]): Either[E, Unit] = va.flatMap(_ => vb)
    def isValid[E](v: Either[E, Unit]): Boolean                     = v.isRight
  }

  implicit object OptionValidated extends FailFastLike[Option] {
    def valid[E]: Option[E]                       = None
    def invalid[E](e: E): Option[E]               = Some(e)
    def and[E](va: Option[E], vb: Option[E]): Option[E] = va.orElse(vb)
    def isValid[E](v: Option[E]): Boolean         = v.isEmpty
  }

  implicit val accumulateValidated: Validated[Accumulate] = Accumulate
  implicit val failFastValidated: Validated[FailFast] = FailFast
}

/** Will change behaviour of combining Validated's depending on strategy */
```

```
sealed trait ValidatedStrategy

/** This strategy will accumulate all errors that occur */
case object AccumulateStrategy extends ValidatedStrategy

/** When first error occur validation will short-circuit and will not execute other validations */
case object FailFastStrategy extends ValidatedStrategy
```

Effect.scala

```
package jap.fields
package typeclass

import scala.concurrent._

/** [[jap.fields.typeclass.Effect]] is a typeclass that provides ValidationDsl with Monad/Defer capabilities. */
trait Effect[F[_]] {

  /** Lifts value into Effect */
  def pure[A](a: A): F[A]

  /** Lazily lifts value into Effect */
  def suspend[A](a: => A): F[A]

  /** Makes effect lazy */
  def defer[A](a: => F[A]): F[A]

  /** FlatMap one effect into another using `f` function */
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

  /** Map effect into another using `f` function */
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

object Effect {

  def apply[F[_]](implicit lm: Effect[F]): Effect[F] = lm

  /** Sync [[jap.fields.typeclass.Effect]]. Short-circuit wont work with it. */
  type Sync[A] = A
  implicit object SyncInstance extends Effect[Sync] {
    def pure[A](a: A): A                 = a
    def flatMap[A, B](fa: A)(f: A => B): B = f(fa)
    def map[A, B](fa: A)(f: A => B): B     = f(fa)
    def suspend[A](a: => A): A           = a
    def defer[A](a: => A): A             = a
  }

  object future {

    /** Requires implicit [[scala.concurrent.ExecutionContext]] in scope */
    implicit def toFutureInstance(implicit ec: ExecutionContext): FutureInstance = new FutureInstance

    /** Future [[jap.fields.typeclass.Effect]]. Sadly Future is not lazy so short-circuit wont work with it, too.
      */
    class FutureInstance(implicit ec: ExecutionContext) extends Effect[Future] {
      def pure[A](a: A): Future[A]                                = Future.successful(a)
      def flatMap[A, B](fa: Future[A])(f: A => Future[B]): Future[B] = fa.flatMap(f)
      def map[A, B](fa: Future[A])(f: A => B): Future[B]          = fa.map(f)
      def defer[A](a: => Future[A]): Future[A]                    = a
      def suspend[A](a: => A): Future[A]                          = Future(a)
    }
  }
}
```

FailWith.scala

```
package jap.fields
package fail

import error.{ValidationMessages as M, ValidationTypes as T}

/** Aggregates all possible FailWith* typeclasses shorthand if you need all of them. If are free to implemented only
  * those you will use. Best practise will be to but implicit instance of this into companion object of your error.
  */
trait FailWith[E, +P]
    extends FailWithMessage[E, P]
    with FailWithCompare[E, P]
    with FailWithInvalid[E, P]
    with FailWithEmpty[E, P]
```

```
    with FailWithNonEmpty[E, P]
    with FailWithMinSize[E, P]
    with FailWithMaxSize[E, P]
    with FailWithOneOf[E, P] { self =>
  trait Mixin {
    implicit val failWith: FailWith[E, P] = self
  }
}

object FailWith {
  type Base[E] = FailWith[E, Nothing]

  trait Builder[E, +P] extends FailWith[E, P] {
    def build[PP >: P](field: Field[PP], errorType: => String, errorMessage: => Option[String]): E

    def invalid[PP >: P](field: Field[PP]): E            = build[PP](field, T.Invalid, Some(M.Invalid))
    def empty[PP >: P](field: Field[PP]): E              = build[PP](field, T.Empty, Some(M.Empty))
    def nonEmpty[PP >: P](field: Field[PP]): E           = build[PP](field, T.NonEmpty, Some(M.NonEmpty))
    def minSize[PP >: P](size: Int)(field: Field[PP]): E = build[PP](field, T.MinSize, Some(M.MinSize(size)))
    def maxSize[PP >: P](size: Int)(field: Field[PP]): E = build[PP](field, T.MaxSize, Some(M.MaxSize(size)))
    def message[PP >: P](error: String, message: Option[String])(field: Field[PP]): E = build[PP](field, error, message)
    def oneOf[PP >: P](variants: Seq[PP])(field: Field[PP]): E                        =
      build[PP](field, T.OneOf, Some(M.OneOf(variants.map(_.toString))))
    def compare[PP >: P](operation: CompareOperation, compared: String)(field: Field[PP]): E =
      build[PP](field, operation.constraint, Some(operation.message(compared)))
  }

  case class Mapped[E, F, +P](mapping: (FieldPath, E) => F)(implicit fw: FailWith[E, P]) extends FailWith[F, P] {
    def invalid[PP >: P](field: Field[PP]): F                 = mapping(field.path, fw.invalid[PP](field))
    def empty[PP >: P](field: Field[PP]): F                   = mapping(field.path, fw.empty[PP](field))
    def nonEmpty[PP >: P](field: Field[PP]): F                = mapping(field.path, fw.nonEmpty[PP](field))
    def minSize[PP >: P](size: Int)(field: Field[PP]): F      = mapping(field.path, fw.minSize[PP](size)(field))
    def maxSize[PP >: P](size: Int)(field: Field[PP]): F      = mapping(field.path, fw.maxSize[PP](size)(field))
    def oneOf[PP >: P](variants: Seq[PP])(field: Field[PP]): F =
      mapping(field.path, fw.oneOf[PP](variants)(field))
    def message[PP >: P](error: String, message: Option[String])(field: Field[PP]): F       =
      mapping(field.path, fw.message[PP](error, message)(field))
    def compare[PP >: P](operation: CompareOperation, compared: String)(field: Field[PP]): F =
      mapping(field.path, fw.compare[PP](operation, compared)(field))
  }

  implicit val errorMessage: Base[String] = new FailWith.Builder[String, Nothing] {
    def build[P](field: Field[P], errorType: => String, errorMessage: => Option[String]): String =
      errorMessage.getOrElse(errorType)
  }

  implicit val fieldPathAndErrorMessage: Base[(FieldPath, String)] = FailWith.Mapped((path, error: String) => (path, error))
  implicit val pathAndErrorMessage: Base[(String, String)] = FailWith.Mapped((path, error: String) => (path.full, error))

  val errorType: Base[String] = new FailWith.Builder[String, Nothing] {
    def build[P](field: Field[P], errorType: => String, errorMessage: => Option[String]): String =
      errorType
  }
}
```

BooleanSyntax.scala

```
package jap.fields
package syntax

import GenericSyntax._
import FailFieldSyntax._
import fail._
import RuleOpsConversions._

object BooleanSyntax extends BooleanSyntax
trait BooleanSyntax {
  implicit final def toBooleanFieldOps[F[_], V[_], E](field: Field[Boolean]): BooleanFieldOps[F, V, E] =
    new BooleanFieldOps(field)
}

trait BooleanDsl[F[_], V[_], E] {
  implicit final def toBooleanFieldOps(field: Field[Boolean]): BooleanFieldOps[F, V, E] =
    new BooleanFieldOps(field)
}

final class BooleanFieldOps[F[_], V[_], E](private val field: Field[Boolean]) extends AnyVal {

  /** Validates [[jap.fields.Field]]#value is `true` */
```

```
  def isTrue(implicit R: RuleOps[F, V, E], FW: FailWithCompare[E, Boolean]): Rule[F, V, E] =
    field.ensure(_ == true, _.failEqual(true))

  /** Validates [[jap.fields.Field]]#value is `false` */
  def isFalse(implicit R: RuleOps[F, V, E], FW: FailWithCompare[E, Boolean]): Rule[F, V, E] =
    field.ensure(_ == false, _.failEqual(false))
}
```

GenericSyntax.scala

```
package jap.fields
package syntax

import scala.reflect.ClassTag

import typeclass._
import fail._

trait GenericDsl[F[_], V[_], E] {
  implicit final def toFieldOps[P](field: Field[P]): FieldOps[P, F, V, E] =
    new FieldOps(field)
}

object GenericSyntax extends GenericSyntax
trait GenericSyntax {
  implicit final def toFieldOps[F[_], V[_], E, P](field: Field[P]): FieldOps[P, F, V, E] =
    new FieldOps(field)
}

final class FieldOps[P, F[_], V[_], E](private val field: Field[P]) extends AnyVal {

  /** Runs validation only if true */
  def when(test: => Boolean)(f: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.when(test)(f(field))

  /** Runs validation only if false */
  def unless(test: => Boolean)(f: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.when(!test)(f(field))

  /** See [[Rule.when]] */
  def when(test: P => Boolean)(rule: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.when(test(field.value))(rule(field))

  /** See [[Rule.whenF]] */
  def whenF(test: P => F[Boolean])(rule: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.whenF(test(field.value))(rule(field))

  /** Runs rule for subtype `PP` else is valid */
  def whenType[PP <: P](rule: Field[PP] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      CT: ClassTag[PP],
  ): Rule[F, V, E] =
    R.defer {
      field.value match {
        case _: PP => rule(field.asInstanceOf[Field[PP]])
        case _     => R.valid
      }
    }

  /** See [[Rule.ensure]] */
  @inline def ensure(test: P => Boolean, error: Field[P] => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.ensure(error(field))(test(field.value))

  /** See [[Rule.ensureF]] */
  @inline def ensureF(test: P => F[Boolean], error: Field[P] => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.ensureF(error(field))(test(field.value))

  /** Like [[Rule.ensure]] but for explicit error */
  @inline def assert(test: P => Boolean, error: Field[P] => E)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.assert(error(field))(test(field.value))

  /** Like [[Rule.ensureF]] but for explicit error */
  @inline def assertF(test: P => F[Boolean], error: Field[P] => E)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.assertF(error(field))(test(field.value))

  /** Returns Suspended Outcome of applying `f` to `field` */
  @inline def check(f: Field[P] => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.pure(f(field))

  /** Returns Defered Outcome of applying `f` to `field` */
```

```
  @inline def checkF(f: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.defer(f(field))

  /** Alias for [[equalTo]] */
  def ===[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    equalTo[C](compared)

  /** Validates that [[jap.fields.Field]]#value is equal to `compared` */
  def equalTo[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    assert(_ == C.value(compared), FW.equal[P, C](compared))

  /** Alias for [[notEqualTo]] */
  def !==[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    notEqualTo[C](compared)

  /** Validates that [[jap.fields.Field]]#value is not equal to `compared` */
  def notEqualTo[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    assert(_ != C.value(compared), FW.notEqual[P, C](compared))

  /** Validates that [[jap.fields.Field]]#value is contained by `seq` */
  def in(seq: => Seq[P])(implicit R: RuleOps[F, V, E], FW: FailWithOneOf[E, P]): Rule[F, V, E] =
    assert(seq.contains, FW.oneOf(seq))

  /** Combines all validations using AND */
  def all(f: Field[P] => Rule[F, V, E]*)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.andAll(f.map(_.apply(field)).toList)

  /** Combines all validations using OR */
  def any(f: Field[P] => Rule[F, V, E]*)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.orAll(f.map(_.apply(field)).toList)

  /** Validates [[jap.fields.Field]] using implicit [[ValidationPolicy]] */
  def validate(implicit P: ValidationPolicy[P, F, V, E]): Rule[F, V, E] = P.validate(field)

  /** Validates [[jap.fields.Field]] using implicit [[ValidationPolicy]] */
  def validateEither(implicit
      R: RuleOps[F, V, E],
      P: ValidationPolicy[P, F, V, E],
      E: HasErrors[V],
  ): F[Either[List[E], P]] =
    P.validateEither(field)
}
```

IterableSyntax.scala

```
package jap.fields
package syntax

import scala.annotation.nowarn

import GenericSyntax._
import fail._
import RuleOpsConversions._
import FailFieldSyntax._

trait IterableDsl[F[_], V[_], E] {
  implicit final def toIterableFieldOps[P, C[X] <: Iterable[X]](
      field: Field[C[P]]
  ): IterableFieldOps[P, C, F, V, E] =
    new IterableFieldOps(field)
}

object IterableSyntax extends IterableSyntax
trait IterableSyntax {
```

```
  implicit final def toIterableFieldOps[F[_], V[_], E, P, C[X] <: Iterable[X]](
      field: Field[C[P]]
  ): IterableFieldOps[P, C, F, V, E] = new IterableFieldOps(field)
}

final class IterableFieldOps[P, C[X] <: Iterable[X], F[_], V[_], E](private val field: Field[C[P]]) extends AnyVal {

  /** Checks that collection is empty */
  def isEmpty(implicit R: RuleOps[F, V, E], FW: FailWithEmpty[E, C[P]]): Rule[F, V, E] =
    field.ensure(_.isEmpty, _.failEmpty)

  /** Checks that collection is not empty */
  def nonEmpty(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, C[P]]): Rule[F, V, E] =
    field.ensure(_.nonEmpty, _.failNonEmpty)

  /** Checks that collection minimum size is `min` */
  def minSize(min: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMinSize[E, C[P]]): Rule[F, V, E] =
    field.ensure(_.size >= min, _.failMinSize(min))

  /** Checks that collection maximum size is `max` */
  def maxSize(max: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMaxSize[E, C[P]]): Rule[F, V, E] =
    field.ensure(_.size <= max, _.failMaxSize(max))

  /** Applies `check` to each collection element */
  def each(check: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    eachWithIndex((f, _) => check(f))

  /** Applies `check` to each collection element */
  def eachWithIndex(check: (Field[P], Int) => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.andAll(field.value.zipWithIndex.map { case (p, i) => check(field.downN(i, p), i) }.toList)

  /** Applies `check` to each collection element, any should succeed */
  def any(check: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    anyWithIndex((f, _) => check(f))

  /** Applies `check` to each collection element, any should succeed */
  def anyWithIndex(check: (Field[P], Int) => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.orAll(field.value.zipWithIndex.map { case (p, i) => check(field.downN(i, p), i) }.toList)

  /** Verifies that collection has distinct elements using `by` property and fails duplicated items using `fail` */
  @nowarn
  def isDistinctBy[K](by: P => K, fail: Field[P] => V[E])(implicit R: RuleOps[F, V, E]) = {
    val distinct = field.value.groupBy(by).mapValues(_.size == 1)
    each(item => R.ensure(fail(item))(distinct.getOrElse(by(item.value), false)))
  }

  /** Verifies that collection has distinct elements and fails duplicated items using `fail` */
  def isDistinct(fail: Field[P] => V[E])(implicit R: RuleOps[F, V, E]) = isDistinctBy(identity, fail)
}
```

MapSyntax.scala

```
package jap.fields
package syntax

import typeclass._
import GenericSyntax._
import fail._
import scala.annotation.nowarn
import RuleOpsConversions._
import FailFieldSyntax._


trait MapDsl[F[_], V[_], E] {
  implicit final def toMapFieldOps[K, P](field: Field[Map[K, P]]): MapFieldOps[K, P, F, V, E] =
    new MapFieldOps(field)
}

object MapSyntax extends MapSyntax
trait MapSyntax {
  implicit final def toMapFieldOps[F[_], V[_], E, K, P](field: Field[Map[K, P]]): MapFieldOps[K, P, F, V, E] =
    new MapFieldOps(field)
}

final class MapFieldOps[K, P, F[_], V[_], E](private val field: Field[Map[K, P]]) extends AnyVal {

  /** Checks that collection is empty */
  def isEmpty(implicit R: RuleOps[F, V, E], FW: FailWithEmpty[E, Map[K, P]]): Rule[F, V, E] =
    field.ensure(_.isEmpty, _.failEmpty)
```

```
  /** Checks that collection is not empty */
  def nonEmpty(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, Map[K, P]]): Rule[F, V, E] =
    field.ensure(_.nonEmpty, _.failNonEmpty)

  /** Checks that collection minimum size is `min` */
  def minSize(min: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMinSize[E, Map[K, P]]): Rule[F, V, E] =
    field.ensure(_.size >= min, _.failMinSize(min))

  /** Checks that collection maximum size is `max` */
  def maxSize(max: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMaxSize[E, Map[K, P]]): Rule[F, V, E] =
    field.ensure(_.size <= max, _.failMaxSize(max))

  /** Applies `check` to each Map element, each should succeed */
  def each(f: Field[(K, P)] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    R.andAll(field.value.zipWithIndex.map { case (entry @ (key, _), index) =>
      f(field.down(TP.toPart(key, index), entry))
    }.toList)

  /** Applies `check` to each Map key, each should succeed */
  def eachKey(check: Field[K] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    each(e => check(e.first))

  /** Applies `check` to each Map value, each should succeed */
  def eachValue(check: Field[P] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    each(e => check(e.second))

  /** Applies `check` to each Map element, any should succeed */
  def any(check: Field[(K, P)] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    R.orAll(field.value.zipWithIndex.map { case (entry @ (key, _), index) =>
      check(field.down(TP.toPart(key, index), entry))
    }.toList)

  /** Applies `check` to each Map key, any should succeed */
  def anyKey(check: Field[K] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    any(e => check(e.first))

  /** Applies `check` to each Map value, any should succeed */
  def anyValue(check: Field[P] => Rule[F, V, E])(implicit
      R: RuleOps[F, V, E],
      TP: MapKeyToPart[K],
  ): Rule[F, V, E] =
    any(e => check(e.second))

  /** Verifies that collection has distinct elements using `by` property and fails duplicated items using `fail` */
  @nowarn
  def isDistinctBy[S](by: ((K, P)) => S, fail: Field[(K, P)] => V[E])(implicit R: RuleOps[F, V, E]) = {
    val distinct = field.value.groupBy(by).mapValues(_.size == 1)
    each(item => R.ensure(fail(item))(distinct.getOrElse(by(item.value), false)))
  }

  /** Verifies that collection has distinct elements and fails duplicated items using `fail` */
  def isDistinct(fail: Field[(K, P)] => V[E])(implicit R: RuleOps[F, V, E]) = isDistinctBy(identity, fail)
}
```

OptionSyntax.scala

```
package jap.fields
package syntax

import GenericSyntax._
import fail._
import RuleOpsConversions._
import FailFieldSyntax._
```

```
trait OptionDsl[F[_], V[_], E] {
  implicit final def toOptionFieldOps[P](field: Field[Option[P]]): OptionFieldOps[P, F, V, E] =
    new OptionFieldOps(field)

  /** Unpacks `rule` from `Option` if `None` returns valid */
  def someOrValid(rule: => Option[Rule[F, V, E]])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    OptionSyntax.someOrValid(rule)
}

object OptionSyntax extends OptionSyntax
trait OptionSyntax {
  implicit final def toOptionFieldOps[F[_], V[_], E, P](field: Field[Option[P]]): OptionFieldOps[P, F, V, E] =
    new OptionFieldOps(field)

  /** Unpacks `rule` from `Option` if `None` returns valid */
  def someOrValid[F[_], V[_], E](rule: => Option[Rule[F, V, E]])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.defer(rule.getOrElse(R.valid))
}

final class OptionFieldOps[P, F[_], V[_], E](private val field: Field[Option[P]]) extends AnyVal {

  /** Alias for [[isDefined]] */
  def isSome(implicit R: RuleOps[F, V, E], FW: FailWithEmpty[E, Option[P]]): Rule[F, V, E] =
    isDefined

  /** Validates that [[jap.fields.Field]]#value is [[scala.Some]] */
  def isDefined(implicit R: RuleOps[F, V, E], FW: FailWithEmpty[E, Option[P]]): Rule[F, V, E] =
    field.ensure(_.isDefined, _.failEmpty)

  /** Alias for [[isEmpty]] */
  def isNone(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, Option[P]]): Rule[F, V, E] =
    isEmpty

  /** Validates that [[jap.fields.Field]]#value is [[scala.None]] */
  def isEmpty(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, Option[P]]): Rule[F, V, E] =
    field.ensure(_.isEmpty, _.failNonEmpty)

  /** Appies `check` to [[jap.fields.Field]]#value if it is [[scala.Some]] or returns valid */
  def some(check: Field[P] => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.defer(field.option.fold[Rule[F, V, E]](R.valid)(check))
}
```

StringSyntax.scala

```
package jap.fields
package syntax

import jap.fields.error.{ValidationMessages => M}
import jap.fields.error.{ValidationTypes => T}

import scala.util.Try
import scala.util.matching.Regex

import GenericSyntax._
import fail._
import RuleOpsConversions._
import FailFieldSyntax._

trait StringDsl[F[_], V[_], E] {
  implicit final def toStringFieldOps(field: Field[String]): StringFieldOps[F, V, E] = new StringFieldOps(field)
}

object StringSyntax extends StringSyntax
trait StringSyntax {
  implicit final def toStringFieldOps[F[_], V[_], E](field: Field[String]): StringFieldOps[F, V, E] =
    new StringFieldOps(field)
}

final class StringFieldOps[F[_], V[_], E](private val field: Field[String]) extends AnyVal {

  /** Validates that [[jap.fields.Field]]#value starts with `value` */
  def startsWith(
      value: => String
  )(implicit R: RuleOps[F, V, E], FW: FailWithMessage[E, String]): Rule[F, V, E] =
    field.ensure(_.startsWith(value), _.failMessage(T.StringStartsWith, M.StringStartsWith(value)))

  /** Validates that [[jap.fields.Field]]#value ends with `value` */
  def endsWith(
      value: => String
  )(implicit R: RuleOps[F, V, E], FW: FailWithMessage[E, String]): Rule[F, V, E] =
```

```
    field.ensure(_.endsWith(value), _.failMessage(T.StringEndsWith, M.StringEndsWith(value)))

  /** Validates that [[jap.fields.Field]]#value is not empty */
  def nonEmpty(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, String]): Rule[F, V, E] =
    field.ensure(_.nonEmpty, _.failNonEmpty)

  /** Validates that [[jap.fields.Field]]#value is not blank */
  def nonBlank(implicit R: RuleOps[F, V, E], FW: FailWithNonEmpty[E, String]): Rule[F, V, E] =
    field.ensure(_.nonEmpty, _.failNonEmpty)

  /** Validates that [[jap.fields.Field]]#value is blank */
  def blank(implicit R: RuleOps[F, V, E], FW: FailWithEmpty[E, String]): Rule[F, V, E] =
    field.ensure(_.isEmpty, _.failEmpty)

  /** Validates that [[jap.fields.Field]]#value minimum size is `min` */
  def minSize(min: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMinSize[E, String]): Rule[F, V, E] =
    field.ensure(_.size >= min, _.failMinSize(min))

  /** Validates that [[jap.fields.Field]]#value maximum size is `max` */
  def maxSize(max: => Int)(implicit R: RuleOps[F, V, E], FW: FailWithMaxSize[E, String]): Rule[F, V, E] =
    field.ensure(_.size <= max, _.failMaxSize(max))

  /** Validates that [[jap.fields.Field]]#value matches Regexp */
  def matches(r: => String)(implicit R: RuleOps[F, V, E], FW: FailWithMessage[E, String]): Rule[F, V, E] =
    field.ensure(_.matches(r), _.failMessage(T.StringMatch, M.StringMatch(r)))

  /** Validates that [[jap.fields.Field]]#value is matches [[scala.util.matching.Regex]] */
  def matchesRegex(r: => Regex)(implicit R: RuleOps[F, V, E], FW: FailWithMessage[E, String]): Rule[F, V, E] =
    matches(r.regex)

  /** Validates that [[jap.fields.Field]]#value is part of [[scala.Enumeration]] */
  def isEnum(e: Enumeration)(implicit R: RuleOps[F, V, E], FW: FailWithOneOf[E, String]): Rule[F, V, E] =
    field.ensure(
      v => Try(e.withName(v)).toOption.isDefined,
      _.failOneOf(e.values.map(_.toString).toList),
    )

  /** Validates that [[jap.fields.Field]]#value is part of Java Enum */
  def isJEnum[T <: Enum[T]](
      values: Array[T]
  )(implicit R: RuleOps[F, V, E], FW: FailWithOneOf[E, String]): Rule[F, V, E] =
    field.ensure(values.map(_.name()).contains, _.failOneOf(values.toSeq.map(_.toString)))

}
```

FailFieldOps.scala

```
package jap.fields
package syntax

import typeclass._
import fail._
import error.FieldError


object FailFieldSyntax extends FailFieldSyntax
trait FailFieldSyntax {
  implicit final def toFailFieldOps[P, V[_], E](field: Field[P]): syntax.FailFieldOps[P, V, E] =
    new syntax.FailFieldOps(field)
}

trait FailFieldDsl[F[_], V[_], E] {
  implicit final def toFailFieldOps[P](field: Field[P]): syntax.FailFieldOps[P, V, E] =
    new syntax.FailFieldOps(field)
}

final class FailFieldOps[P, V[_], E](private val field: Field[P]) extends AnyVal {

  /** Use field path to construct error */
  def failPath(error: FieldPath => E)(implicit V: Validated[V]): V[E] = V.invalid(error(field.path))

  /** Just pathrought error */
  def fail(error: E)(implicit V: Validated[V]): V[E] = V.invalid(error)

  /** Useful when your error is wrapped in FieldError */
  def failField[EE](error: EE)(implicit ev: FieldError[EE] =:= E, V: Validated[V]): V[E] =
    V.invalid(ev(FieldError(field.path, error)))

  /** Returns InvalidError using [[jap.fields.fail.FailWithInvalid]] typeclass */
  def failInvalid(implicit FW: FailWithInvalid[E, P], V: Validated[V]): V[E] = V.invalid(FW.invalid(field))
```

```
  /** Returns EmptyError using [[jap.fields.fail.FailWithEmpty]] typeclass */
  def failEmpty(implicit FW: FailWithEmpty[E, P], V: Validated[V]): V[E] = V.invalid(FW.empty(field))

  /** Returns NonEmptyError using [[jap.fields.fail.FailWithNonEmpty]] typeclass */
  def failNonEmpty(implicit FW: FailWithNonEmpty[E, P], V: Validated[V]): V[E] = V.invalid(FW.nonEmpty(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failGreater[C](c: C)(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.greater(c)(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failGreaterEqual[C](
      c: C
  )(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.greaterEqual(c)(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failLess[C](c: C)(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.less(c)(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failLessEqual[C](c: C)(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.lessEqual(c)(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failEqual[C](c: C)(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.equal(c)(field))

  /** Returns CompareError using [[jap.fields.fail.FailWithCompare]] typeclass */
  def failNotEqual[C](c: C)(implicit FW: FailWithCompare[E, P], C: FieldCompare[P, C], V: Validated[V]): V[E] =
    V.invalid(FW.notEqual(c)(field))

  /** Returns MinSizeError using [[jap.fields.fail.FailWithMinSize]] typeclass */
  def failMinSize(size: Int)(implicit FW: FailWithMinSize[E, P], V: Validated[V]): V[E] =
    V.invalid(FW.minSize(size)(field))

  /** Returns MaxSizeError using [[jap.fields.fail.FailWithMaxSize]] typeclass */
  def failMaxSize(size: Int)(implicit FW: FailWithMaxSize[E, P], V: Validated[V]): V[E] =
    V.invalid(FW.maxSize(size)(field))

  /** Returns OneOfError using [[jap.fields.fail.FailWithOneOf]] typeclass */
  def failOneOf[PP >: P](variants: Seq[PP])(implicit FW: FailWithOneOf[E, P], V: Validated[V]): V[E] =
    V.invalid(FW.oneOf(variants)(field))

  /** Returns MessageError using [[jap.fields.fail.FailWithMessage]] typeclass */
  def failMessage(error: String, description: Option[String] = None)(implicit
      FW: FailWithMessage[E, P],
      V: Validated[V],
  ): V[E] =
    V.invalid(FW.message(error, description)(field))

  /** Returns MessageError using [[jap.fields.fail.FailWithMessage]] typeclass */
  def failMessage(error: String, description: String)(implicit
      FW: FailWithMessage[E, P],
      V: Validated[V],
  ): V[E] =
    V.invalid(FW.message(error, Some(description))(field))
}
```

OrderingSyntax.scala

```
package jap.fields
package syntax

import GenericSyntax._
import typeclass._
import fail._
import RuleOpsConversions._
import FailFieldSyntax._


trait OrderingDsl[F[_], V[_], E] {
  implicit final def toOrderingFieldOps[P](field: Field[P]): OrderingFieldOps[P, F, V, E] =
    new OrderingFieldOps(field)
}
object OrderingSyntax extends OrderingSyntax
trait OrderingSyntax                    {
  implicit final def toOrderingFieldOps[F[_], V[_], E, P](field: Field[P]): OrderingFieldOps[P, F, V, E] =
    new OrderingFieldOps(field)
```

```
}
final class OrderingFieldOps[P, F[_], V[_], E](private val field: Field[P]) extends AnyVal {

  /** Validates that [[jap.fields.Field]]#value is greater or equal to `compared` */
  def >=[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] = gte[C](compared)

  /** Validates that [[jap.fields.Field]]#value is greater or equal to `compared` */
  def gte[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    field.ensure(O.gteq(_, C.value(compared)), _.failGreaterEqual(compared))

  /** Validates that [[jap.fields.Field]]#value is greater than `compared` */
  def >[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] = gt[C](compared)

  /** Validates that [[jap.fields.Field]]#value is greater than `compared` */
  def gt[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    field.ensure(O.gt(_, C.value(compared)), _.failGreater(compared))

  /** Validates that [[jap.fields.Field]]#value is less or equal to `compared` */
  def <=[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] = lte(compared)

  /** Validates that [[jap.fields.Field]]#value is less or equal to `compared` */
  def lte[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    field.ensure(O.lteq(_, C.value(compared)), _.failLessEqual(compared))

  /** Validates that [[jap.fields.Field]]#value is less than `compared` */
  def <[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] = lt[C](compared)

  /** Validates that [[jap.fields.Field]]#value is less than `compared` */
  def lt[C](compared: => C)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      C: FieldCompare[P, C],
  ): Rule[F, V, E] =
    field.ensure(O.lt(_, C.value(compared)), _.failLess(compared))

  /** Validates that [[jap.fields.Field]]#value is  greaterEqual than `from` and lessEqual `to` */
  def isBetween[FROM, TO](from: => FROM, to: => TO)(implicit
      R: RuleOps[F, V, E],
      O: Ordering[P],
      FW: FailWithCompare[E, P],
      CF: FieldCompare[P, FROM],
      CT: FieldCompare[P, TO],
```

```
  ): Rule[F, V, E] =
    R.and(gte[FROM](from), lte[TO](to))
}
```

RuleSyntax.scala

```
package jap.fields
package syntax

import typeclass._

object RuleSyntax extends RuleSyntax
trait RuleSyntax {
  implicit final def toRuleValidationOps[F[_], V[_], E](rule: Rule[F, V, E]): RuleValidationOps[F, V, E] =
    new RuleValidationOps[F, V, E](rule)

  implicit final def toRuleSequenceOps[F[_], V[_], E](rules: Iterable[Rule[F, V, E]]): RuleSequenceOps[F, V, E] =
    new RuleSequenceOps(rules)
}

final class RuleSequenceOps[F[_], V[_], E](private val rules: Iterable[Rule[F, V, E]]) extends AnyVal {

  /** See [[Rule.andAll]] */
  def combineAll(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.andAll(rules.toList)

  /** See [[Rule.andAll]] */
  def andAll(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.andAll(rules.toList)

  /** See [[Rule.orAll]] */
  def orAll(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.orAll(rules.toList)
}

final class RuleValidationOps[F[_], V[_], E](private val rule: Rule[F, V, E]) extends AnyVal {

  /** Same as [[jap.fields.typeclass.Validated.isInvalid]] but effectful */
  def isInvalid(implicit R: RuleOps[F, V, E]): F[Boolean] =
    R.F.map(rule.effect)(R.V.isInvalid)

  /** Same as [[jap.fields.typeclass.Validated.isValid]] but effectful */
  def isValid(implicit R: RuleOps[F, V, E]): F[Boolean] =
    R.F.map(rule.effect)(R.V.isValid)

  /** Same as [[jap.fields.typeclass.HasErrors.errors]] but effectful */
  def errors(implicit F: Effect[F], E: HasErrors[V]): F[List[E]] =
    F.map(rule.effect)(E.errors[E])

  /** Same as [[jap.fields.typeclass.Validated.whenValid]] but effectful */
  def whenValid(b: => Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.modifyM(rule)(vr => if (R.V.isValid(vr)) b else rule)

  /** Same as [[jap.fields.typeclass.Validated.whenValid]] but effectful */
  def whenInvalid(f: V[E] => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.modify(rule)(vr => if (R.V.isInvalid(vr)) f(vr) else vr)

  /** Same as [[jap.fields.typeclass.Validated.asError]] but effectful */
  def asError(error: => E)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.modify(rule)(R.V.asError(_)(error))

  /** Same as [[jap.fields.typeclass.Validated.asError]] but effectful */
  def asInvalid(invalid: => V[E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.modify(rule)(R.V.asInvalid(_)(invalid))

  /** Same as [[jap.fields.typeclass.Validated.when]] but effectful */
  def when(cond: => Boolean)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.defer(if (cond) rule else R.valid)

  /** Same as [[jap.fields.typeclass.Validated.unless]] but effectful */
  def unless(cond: => Boolean)(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.defer(if (cond) R.valid else rule)

  /** See [[Rule.or]] */
  def or(b: Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.or(rule, b)

  /** See [[Rule.or]] */
  def ||(b: Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.or(rule, b)
```

```
  /** See [[Rule.and]] */
  def and(b: Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.and(rule, b)

  /** See [[Rule.and]] */
  def &&(b: Rule[F, V, E])(implicit R: RuleOps[F, V, E]): Rule[F, V, E] =
    R.and(rule, b)
}
```