МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

_____ Оксана ШОВКОПЛЯС

(підпис)

6 грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»

освітньо-професійної програми «Інформатика»

на тему: «Інформаційна технологія тестування мікросервісів в процесі розробки»

здобувача групи ІН.м-32 Кіхтенка Дмитра Євгенійовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Дмитро КІХТЕНКО

(підпис)

Керівник Тетяна ЛАВРИК

Старший викладач кафедри кібербезпеки,

к.п.н., доцент _____

(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр

зі спеціальності 122 – «Комп'ютерних наук», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-32 Кіхтенка Дмитра Євгенійовича

1. Тема роботи: «Інформаційна технологія тестування мікросервісів в процесі розробки» затверджую наказом по СумДУ від «3» грудня 2024 року наказ №1257-VI
2. Термін здачі здобувачем кваліфікаційної роботи до 6 грудня 2024 року
3. Вхідні дані до кваліфікаційної роботи_____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Моделювання та проєктування. 3) Програмна реалізація.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
6. Консультанти до проєкту (роботи) із зазначенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «22» серпня 2024 р.
Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз предметної області, аналіз сучасних підходів*		
2	*Визначення переваг і недоліків сучасних рішень, визначення завдань роботи*		
3	*Проєктування програмної реалізації*		
4	*Програмна реалізація*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 124 стор., 9 рис., 5 таблиць, 2 додатків, 23 використаних джерел.

Обґрунтування актуальності теми роботи: Розвиток інформаційних технологій у сучасному світі спричинив появу великих додатків на базі мікросервісної архітектури. Такі додатки створені з метою в найкоротші терміни задовольняти динамічні потреби користувачів. QA, DevOps інженери і особливо розробники часто стикаються з проблемою залежності мікросервісу від інших зовнішніх або внутрішніх компонентів в процесі його відлагодження чи тестування в ізольованому середовищі. Інструмент для створення фіктивних API допоможе зменшити час витрачений на розробку, підтримку та розширення програмного продукту на базі такої архітектури, а також дозволить фахівцям докладати менше зусиль під час тестування програмного продукту.

Об’єкт дослідження: Тестування мікросервісів в ізольованому середовищі.

Предмет дослідження. Інформаційна технологія тестування мікросервісів в ізольованому середовищі.

Мета роботи: Розробка інформаційної технології тестування мікросервісів в ізольованому середовищі з використанням інструменту API мокування, орієнтованого на систему gRPC.

Методи дослідження: Аналіз роботи сучасних інструментів API мокування, порівняння інструментів орієнтованих на створення макетних серверів gRPC API, методи розробки, що базуються на мові програмування Python із використанням мови шаблонізації jinja2.

Результати: Розроблено і протестовано утиліту для мокування gRPC API серверів, яка містить більше функцій мокування в порівнянні з аналогами, створено рекомендації та інструкції щодо її використання.

МІКРОСЕРВІСИ, ТЕСТУВАННЯ, МОКУВАННЯ API, GRPC API

ЗМІСТ

ВСТУП **5**

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ **6**

1.1 Мікросервісна архітектура 6

1.2 Проблематика 8

1.3 Інструменти API мокування 10

1.4 Система gRPC 15

1.5 Аналіз існуючих рішень 17

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ **20**

2.1 Архітектура додатку 20

2.2 Налаштування додатку 21

2.3 Технології та інструменти 22

3 ПРОГРАМНА РЕАЛІЗАЦІЯ **25**

3.1 Структура програми 25

3.2 Встановлення 26

3.3 Налаштування програми 27

3.4 Принцип роботи 35

3.5 Функції мокування 37

3.6 Приклад використання 43

ВИСНОВКИ **47**

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **48**

ДОДАТОК А **50**

ДОДАТОК Б **51**

ВСТУП

Обґрунтування теми вибору роботи. Вибір теми обумовлений досвідом розробки і тестування мікросервісів в ізольованому середовищі. Специфіка мікросервісної архітектури вимагає від фахівця особливих підходів до взаємодії з компонентами додатку. Вибір правильного підходу і відповідне інструментальне забезпечення допоможе ефективно вирішувати задачі розробки і тестування мікросервісів.

Актуальність. Якість програмного забезпечення та час, витрачений на його розробку і тестування, є важливим для додатків на базі мікросервісної архітектури в умовах конкуренції між постачальниками послуг галузі інформаційних технологій. Забезпечення фахівців цієї галузі функціональними інструментами дозволить зробити тестування компонентів додатку в ізольованому середовищі простішим, швидшим та якіснішим.

Об'єкт дослідження. Тестування мікросервісів в ізольованому середовищі.

Предмет дослідження. Інформаційна технологія тестування мікросервісів в ізольованому середовищі.

Новизна. Запропоновано новий інструмент gRPC API мокування, який на відміну від існуючих рішень включає в себе весь набір динамічного мокування функцій на відміну від існуючих рішень.

Структура. Дана робота складається із вступу, аналізу предметної галузі, моделювання та проєктування, програмної реалізації, списку використаних джерел та додатків.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Мікросервісна архітектура

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, який полягає у поділі програми на невеликі незалежні служби. Ці служби спілкуються через добре визначені API, що дозволяє легко розробляти та підтримувати їх автономними командами. Це робить мікросервіси найбільш масштабованим методом розробки програмного забезпечення.

Дизайн мікросервісів є прямою протилежністю монолітної архітектури. Моноліт — це одна велика кодова база, яка реалізує всі функції. Вся логіка додатку знаходиться в одному місці, і жоден компонент не може працювати окремо. Головною перевагою монолітів є те, що їх легко налаштовувати та розгортати.

Мікросервіси дозволяють компаніям утримувати команди невеликими та гнучкими. Головна ідея мікросервісів полягає в тому, щоб розкласти програму на невеликі служби, які можуть автономно розроблятися та розгортатися тісно згуртованими командами.

Існує декілька причин, чому сучасні IT-компанії використовують мікросервісну архітектуру для своїх продуктів:

1. Спрощене обслуговування і розвиток проєкту. Поки монолітний проєкт має невеликий розмір, розробники можуть швидко та легко вносити зміни, оскільки зв’язки між різними частинами моноліту прозорі. Однак у міру того, як збільшується об’єм задач, які має виконувати додаток, збільшується зв’язаність його частин і збільшується відповідно кількість команд, які працюють над єдиною кодовою базою, розвиток моноліту стає проблемним. Завелика кількість частин в одному місці робить процес обслуговування і розширення функціоналу додатку складним і повільним. В той же час мікросервісна архітектура дозволяє розподілити додаток на слабо пов’язані частини, що дозволяє командам легше обслуговувати і розвивати додаток.

2. Вимога швидкого розвитку проєкту. Часто мікросервісна архітектура використовується в проєкті, якщо додаток повинен забезпечувати швидке реагування на потреби кінцевих користувачів в умовах конкуренції. Монолітний додаток вимагає довгострокових циклів випуску нових версій, в той час як особливості мікросервісного додатку дозволяють використовувати набагато менші цикли випуску оновлень, адже фахівцям потрібно менше часу на розширення тієї чи іншої підсистеми, замість розширення усього додатку. З цієї причини мікросервісна архітектура використовуються в SaaS платформах [4].

3. Ефективне масштабування. Коли додаток монолітний і перевантажений кількістю запитів, він потребує масштабування. Монолітний додаток з усієї логікою в одному місці складно масштабувати, оскільки частини тісно пов’язані. В той час як мікросервісний додаток дає можливість масштабувати кожну частину окремо, що значно швидше і простіше.

До усіх переваг мікросервісів можна віднести:

1. Масштабованість. Якщо мікросервіс все частіше досягає свого пікового навантаження, можна швидко допрацювати його для підтримки масштабування і швидко розгорнути нові екземпляри в супровідному кластері, аби зменшити навантаження на один робочий екземпляр сервісу.

2. Висока відмовостійкість. Перевагою розподіленої системи є можливість уникнути окремих точок відмови. Мікросервіси можна розгортати в різних зонах доступності за допомогою хмарних технологій, гарантуючи, що користувачі ніколи не зазнають збою.

3. Менші команди. Завдяки мікросервісам команди фахівців можуть залишатися невеликими та згуртованими. Чим менша група працює над однією і тією ж частиною, тим менше накладних витрат на спілкування та краща співпраця. Amazon, наприклад, визначає розмір команди за правилом двох піц. Це означає, що команда має бути достатньо маленькою, щоб її можна було нагодувати двома піцами.

4. Прискорення впровадження інновацій. Мікросервіси є незалежними програмними компонентами, які можуть працювати на основі різних платформ і технологій. Слабка взаємозалежність між мікросервісами дозволяє компаніям швидше оновлювати окремі компоненти у великому додатку, на відміну від монолітного з єдиною кодовою базою.

5. Свобода вибору стеку технологій. В монолітному додатку параметри мовного та технологічного стеку закладені з самого початку. Нові розробники повинні адаптуватися до будь-якого вибору, зробленого в минулому. У мікросервісній архітектурі навпаки, кожен сервіс може використовувати стек технологій, який найбільше підходить для вирішення поставленої задачі. Таким чином, команда може вибрати найкращий інструмент для роботи на основі своїх навичок.

6. Менший цикл розробки і випуску продукту. Цикл розробки та тестування коротший, бо менші команди швидше здійснюють розробку. Кожна з команд здатна розгортати свої оновлення в будь-який час, тому мікросервіси можна оновлювати частіше.

Мікросервіси вирішують низку проблем для програмного забезпечення та компаній, що розвиваються. Але вони аж ніяк не є ідеальними [1, С. 9].

1.2 Проблематика

Фахівці, що працюють з мікросервісними додатками часто стикаються з проблемою залежності мікросервісу від інших внутрішніх або зовнішніх компонентів. В процесі розробки, розгортання чи налаштування одного незалежного компоненту необхідно перевірити коректність його роботи, але компонент часто може вимагати взаємодії з іншими частинами всього додатку. Фахівцям необхідно витрачати зусилля, час і ресурси для того, щоб розгорнути залежні частини або тимчасово видалити частину коду, яка відповідає за взаємодію з цими додатками. Це стосується QA і DevOps інженерів і особливо розробників.

Наприклад, маємо великий проєкт, який складається з 20 мікросервісів, що взаємодіють один із одним використовуючи мережеві протоколи. Кожен елемент системи розробляється окремо від інших, але він обов’язково повинен

взаємодіяти з іншими частинами усього додатку в процесі роботи. Візьмемо для прикладу один мікросервіс, який взаємодіє з п’ятьма іншими. Розробнику необхідно додати нову бізнес логіку до цього компоненту. Для адаптерів сервісу (тих частин, які виконують функцію взаємодії з іншими мікросервісами) необхідно зімітувати або розгорнути інші залежні мікросервіси для тестування і відлагодження адаптерів. У більшості випадків це вимагає доволі великої кількості часу на вивчення кодової бази адаптерів, пошук індивідуальної стратегії для заглушення цих адаптерів або це вимагає часу і обчислювальних ресурсів на повноцінне розгортання залежних сервісів. Основна проблема — це завелика кількість часу, витраченого на підготовку робочого середовища або на перевірку працездатності чи відлагодження цих компонентів. Дуже важливо витратити на процес підготовки робочого середовища тестування, а також на сам процес тестування якомога менше часу.

Разом із змінами в коді розробник зазвичай додає також і автоматизовані модульні тести, але в міру того, що мікросервісна архітектура має розвиватися швидко, покривати автоматизованими тестами всі випадки може бути недоцільним, адже це затратний за часом процес. Натомість доцільним є ручне тестування: інтеграційне, компонентне або наскрізне, для випадків, які стосуються тільки внесених в код додатку змін в умовах наближених до виробничого середовища.

Тестування додатку таким способом корисне розробникам під час ітераційної розробки у процесі відлагодження внесених змін в код додатку, а також перед доставкою змін у виробниче середовище. QA інженеру такий підхід корисний у випадку тестування змін вже доставлених у відповідне середовище, коли умови тестів важко відтворити в середовищі або коли воно з тієї чи іншої причини недоступне. DevOps інженерам таке рішення може бути корисним при відлагодженні проблем із розгортанням мікросервісу в кластері, пов’язаних із його залежністю від зовнішніх API.

Для того аби забезпечити швидке ручне тестування в умовах наближених до виробничого середовища необхідні ефективні допоміжні інструменти, які зможуть зекономити час, позбавити фахівців зайвої роботи і дозволять більше зосередити на вирішенні бізнес задач.

Найбільш поширеним протоколом за яким будується взаємодія між компонентами мікросервісної архітектури є протокол HTTP версії 1.1, а його найпоширенішим архітектурним стилем взаємодії між клієнтом і сервером є REST API. Другим за поширеністю підходом можна назвати використання системи взаємодії між клієнтом і сервером під назвою gRPC, яка використовує HTTP версії 2, а також окрему мову опису інтерфейсів. В мережі існує достатня кількість інструментів, які дають можливість створити API симуляцію для HTTP/1.1 серверів, які поєднують у собі всі можливі функції, чого не скажеш про інструментарій для системи gRPC, в міру того, що вона зустрічається доволі рідко і більшою мірою в мікросервісній архітектурі, таких утиліт існує небагато і вони можуть мати недостатньо функціоналу для тестування API [2-3].

1.3 Інструменти API мокування

Утиліти для симуляції API мікросервісів відносяться до групи так званих інструментів імітації API або API мокування (API mocking tools) і зазвичай забезпечують спосіб створення макетної версії API, налаштування її для повернення конкретних відповідей на вхідні запити та імітації різних сценаріїв і поведінки. Ці програми дозволяють розробникам тестувати свої додатки чи служби, не покладаючись на дійсний API, Вони можуть включати імітацію помилкових відповідей, затримку при обробці запитів та інші типи відповідей, які важко або неможливо відтворити за допомогою дійсного API.

Загалом можна виділити наступні ситуації, для яких API мокування може бути корисним:

1. Тестування: імітаційні API можна використовувати для перевірки функціональності програми чи служби, не покладаючись на діючий API. Це можна використовувати для перевірки стійкості та надійності програми чи служби, а

також може допомогти розробникам виявити та усунути будь-які проблеми, які можуть виникнути.

2. Розробка: імітаційні API можна використовувати для розробки та тестування програм або послуг, не чекаючи, поки стане доступним живий API. Це може бути особливо корисним, коли розробка деякої функціональності здійснюється декількома фахівцями одночасно, коли активний API ще недоступний або його ще неможливо використовувати активний API для розробки чи тестування іншого компоненту.

3. Симуляція: імітаційні API можна використовувати для моделювання різних сценаріїв і поведінки, які важко або неможливо відтворити за допомогою живого API. Це може включати імітацію відповідей з помилками, імітації часової затримки або особливі випадки відповідей від API.

4. Ізоляція: імітаційні API можна використовувати для ізоляції програми або служби, що розробляється, від інших систем, дозволяючи розробникам зосередитися на функціональності програми або служби без впливу зовнішніх факторів.

Головними перевагами які надають інструменти мокування є:

1. Прискорення розвитку додатку. Інструменти API симуляції прискорюють процес розробки і тестування додатку у випадках, коли компоненти від яких API інтерфейсів залежить додаток, що розробляється, так само знаходяться в процесі розробки. Замість того аби чекати реалізації необхідних для тестування компонентів, фахівці можуть скористатися інструментами мокування. Процес розробки може відбуватися паралельно, єдине, що потрібно – API інтерфейси за якими додатки мають взаємодіяти. Крім того, інструменти мокування оптимізовані, таким чином, щоб дати можливість швидко наповнювати макетні сервери фіктивними даними, чого немає у дійсних API серверах.

2. Низька вартість. Інструменти симуляції дозволяють позбутися дороговартісних комплексних рішень або сторонні послуг, необхідних для тестування API. До таких рішень можна віднести утримання окремого кластеру всього

мікросервісного додатку для індивідуальних потреб тестування. Крім того використання інструментів мокування замість дійсного API серверу для тестування і розробки вимагає менше часу, що само по собі дає меншу вартість на розробку додатку.

3. Точність і надійність розробки та тестування. Інструменти API мокування дозволяють створювати різноманітні сценарії поведінки API, які може бути важко або затратно відтворювати на дійсному API.

4. Прискорення співпраці фахівців. Коли фахівці працюють над розробкою різних частин одночасно завдяки інструментам мокування, помилки інтеграції виявлені одним фахівцем, будуть відомі фахівцям, що працюють з іншими частинами значно раніше [6].

Інструменти мокування можна поділити на 2 типи залежно від робочого середовища:

- загальнодоступні – ті, що працюють на віддаленому сервері;
- локальні – при використанні трафік не виходить в глобальну мережу.

Утиліти також поділяються на типи залежно від комбінації цільового протоколу і архітектурного стилю, за яким здійснюється комунікація між клієнтом і сервером. Так існують імітаційні API для REST API, GraphQL, Kafka Async API, Thrift та інші.

Інструменти мокування використовують так звані заглушки або фіктивні дані. Це вручну встановлені користувачем дані відповідей якими має відповідати сервер для імітації поведінки дійсного API. Заглушки можуть бути:

- статичні – вміст фіктивних даних у відповіді на один і той самий запит завжди однаковий і не змінюється залежно від умов. Статичні заглушки є базовою функцією будь-якого інструменту мокування;
- динамічні – вміст фіктивних даних у відповіді на один і той самий запит змінюється залежно від вмісту запиту або інших умов.

При визначенні функціональних можливостей утиліт API симуляції, були взяті до уваги лише ті, які є важливими саме для ручного тестування

мікросервісних додатків, можливості для автоматизованого тестування інструментів не були взяті уваги. Основними функціями інструментів мокування для ручного тестування є:

1. Створення фіктивного серверу для імітації поведінки дійсного серверу, так аби інші сервіси могли взаємодіяти з фіктивним, наче він дійсний.

2. Захоплення вхідних даних запитів від серверів, що взаємодіють із фіктивним для їх подальшого аналізу.

Інструменти мокування загалом можуть мати наступні можливості для імітації поведінки серверів:

1. Заглушки із статичними фіктивними даними.

2. Заглушки із динамічними фіктивними даними.

3. Імітація часової затримки при обробці запитів.

Процес тестування додатків з використанням інструментів мокування значно пришвидшується, якщо інструмент мокування має підтримку динамічних фіктивних даних. Можна виділити наступні стратегії створення динамічних фіктивних даних:

1. Шаблони – дозволяють додавати значення властивостей вхідних даних у вихідні дані без додаткової логіки обробки цих вхідних даних. Наприклад, на запит додавання користувача сервер має відповідати сутністю, яка містить той самий логін користувача, що і у запиті. Користувач інструменту сам вказує, які шаблони використати при обробці того чи іншого запиту.

2. Проксі режим – дозволяє перенаправляти окремі запити на дійсний сервер, що працює в хмарі за умов, якщо цей запит не змінює стан серверу. При цьому користувач інструменту вручну вказує адресу серверу на який необхідно перенаправити запит.

3. Скрипти-обробники – виконують певну логіку генерації відповіді використовуючи дані запиту. Користувач інструменту створює скрипти із логікою обробки самостійно.

4. Стани – дозволяють користувачу створити стани, в яких може перебувати сервер, створити умови за яких програма змінює свій стан і окрему поведінку серверу в кожному із цих станів.

Загальний алгоритм роботи із інструментом мокування виглядає наступним чином:

1. Визначення структури API сервісу, для якого необхідно створити симуляцію:

а) адреса сервера, сертифікати безпеки.

б) структура вхідних даних – запитів;

в) структура вихідних даних – відповідей.

2. Визначення набору заглушок і налаштувань для них залежно від можливостей інструменту мокування:

а) визначення типів для заглушок: статичні або динамічні;

б) визначення наповнення заглушок фіктивними даними для емуляції нормальної поведінки або специфічної для тестового випадку поведінки;

в) встановлення взаємозв’язку між запитом і фіктивними даними.

3. Реалізація визначених налаштувань фіктивного сервера і його заглушок з використанням інструменту мокування.

4. Налаштування програми, яку необхідно протестувати для роботи з фіктивним сервером замість дійсного серверу.

5. Перевірка поведінки програми, що тестується в різних сценаріях зі створеними заглушками.

Залежно від умов тестування і можливостей цільового інструменту алгоритм необхідно повторювати, оскільки коли змінюються умови тестування, змінюються і налаштування фіктивного серверу або його фіктивних даних.

Загалом чим більшу кількість функцій інструмент імітації серверів надає користувачу тим більше він гнучкості при створені тестових випадків і тим

менше часу необхідно на вирішення задачі, однак водночас важливо аби інструмент залишався простим у використанні [6, 7].

Терміни, опис, класифікація інструментів мокування і класифікація їхніх функцій визначена також на основі аналізу документації і локального тестування найпопулярніших рішень REST API мокування [5].

1.4 Система gRPC

Більшість інструментів мокування створені для архітектур на базі HTTP/1.1 і архітектурного стилю REST API. Системи взаємодії gRPC базується на HTTP/2 і тому значно відрізняється від них. gRPC має велику кількість особливостей, які можуть впливати на реалізацію утиліти, орієнтованої саме для цього підходу комунікації.

gRPC (Google Remote Procedure Call) – це надійна система віддаленого виклику процедур з відкритим кодом, яка використовується для створення масштабованих і швидких API. Основними складовими цього протоколу є:

1. Мережевий протокол HTTP версії 2.

2. Архітектурний стиль RPC (Remote Procedure Call).

3. Мова опису API контрактів (IDL – Interface Definition Language) під назвою Protocol Buffers або Protobuf.

У gRPC клієнт може безпосередньо викликати метод серверу на іншій машині, так, наче це локальний об’єкт, що полегшує створення розподілених програм і служб.

Як і в архітектурному стилі RPC, gRPC базується на ідеї визначення служби або сервісу, і визначення методів на кожній з таких служб, які можна викликати віддалено, з їхніми параметрами та типами повернення. Параметри запиту і параметри результату записані в повідомленнях. Повідомлення є одночасно тілом запиту або тілом відповіді і одночасно є сутністю з логічно об’єднаними властивостями або параметрами.

Ключовими особливостями архітектурного стилю RPC можна назвати те, що клієнт розглядає механізм взаємодії, як виклик процедур із певними вхідними

і результуючими параметрами на стороні серверу, в той час як в класичному REST принципі клієнт розглядає механізм взаємодії як операції над ресурсами або сутностями такі як створення, отримання, модифікація або видалення (CRUD).

На стороні сервера реалізований інтерфейс у вигляді служб із методами і повідомленнями і запускається процес прослуховування і обробки клієнтських викликів. На стороні клієнта існує адаптер, який так само знає про служби, методи і повідомлення, які має сервер і може з'єднуватися із сервером для виклику методів на службах із певними параметрами у вигляді повідомлень.

Служби, методи, повідомлення і їх параметри описані у чітко визначених одному або декількох контрактах взаємодії, які являє собою текстові файли. Файли контрактів описуються спеціально створеною мовою опису інтерфейсів (IDL) Protobuf із розширенням .proto. Особливістю є те, що клієнт і сервер використовують ці файли для компіляції з них вихідного коду, що інкапсулює логіку ефективною взаємодії за встановленими у Protobuf контрактами.

Важливою відмінністю системи взаємодії gRPC є підтримка таких функцій як стримінг на стороні клієнта, стримінг на стороні сервера, а також двонаправлений стримінг, який полягає в тому, що під час взаємодії як клієнт так і сервер можуть відправляти потік повідомлень замість лише одного повідомлення. Замість заголовків в gRPC для користувацьких потреб (авторизація, трасування запитів, інші службові дані) використовуються метадані, які відправляються тільки один раз за сеанс запиту. Початкові метадані відправляються клієнтом один раз перед відправкою потоку повідомлень. Кінцеві метадані відправляються сервером один раз після закінчення відправки потоку повідомлень до клієнта.

Протокол HTTP/2 кардинально відрізняється від класичного HTTP/1.1 за наступними критеріями:

1. Бінарний формат передачі даних – на відміну від HTTP/1.1, де передача даних відбувається в звичайному текстовому форматі, формат даних даних в

gRPC бінарний, що зменшує обсяг даних і пришвидшує їх пересилання по мережі.

2. Мультиплексування – декілька запитів на сервер можуть відбуватися в паралельному режимі в межах одного TCP/IP з'єднання, при чому сервер може відмінити обробку одного із запитів не впливаючи на інші запити в межах одного TCP сеансу. В той час як в протоколі HTTP/1.1 запити в межах одного TCP сеансу, могли оброблятися тільки послідовно, а відміна одного із запитів відміняла обробку всіх запитів в межах сеансу.

3. Підтримка двостороннього стримінгу – сервер може надсилати відповіді ще до того, як клієнт завершить відправку запиту, що може значно прискорювати обробку запитів.

4. Пріоритети потоків – клієнт може встановлювати пріоритет на обробку запитів в межах одного TCP/IP з'єднання, сервер отримуючи запити з пріоритетами обробляє їх в порядку від найбільшого за пріоритетом.

5. Стиснення заголовків запиту через алгоритм HPACK – при передачі даних запиту стиснення заголовків здійснюється на рівні протоколу, в той час як в HTTP/1.1 стиснення відбувається на рівні TLS/SSL, при цьому старий алгоритм стиснення DEFLATE в цій версії протоколу вразливий до CRIME-атак.

В цілому описані відмінності HTTP/2 протоколу від HTTP/1.1 призвели до підвищення швидкості передачі даних і до підвищення рівня безпеки протоколу [3, 7, 8].

1.5 Аналіз існуючих рішень

Було здійснено пошук інструментів для створення API симуляцій gRPC серверів. Серед усіх можливих рішень були відібрані тільки ті, що:

- не мають прив'язки до мови програмування або конкретної технології розробки програмного забезпечення. Це стосується бібліотек для написання Unit тестів для фреймворків певних мов програмування або частини фреймворків, які можуть запускати gRPC сервери;

• були спроєктовані з для підтримки тільки симуляції серверів gRPC API. Рішення, які вимагають додаткових зусиль для встановлення або налаштування плагінів для підтримки gRPC або комплексні рішення API мокування не розглядались;

• є повністю або частково безкоштовними, без обмежень або з обмеженнями у використанні. Платні рішення не брались до уваги.

Загалом знайдено одне рішення, яке підпадає під пошукові критерії: GripMock.

GripMock – це консольний інструмент, який дозволяє створювати API симуляції gRPC серверів з динамічними або статичними відповідями. GripMock – це макет-сервер для служб gRPC. Він використовує файли .proto для створення служб gRPC. Gripmock можна використовувати для налаштування наскрізного тестування або як фіктивний сервер на етапі розробки програмного забезпечення. Сервер реалізовано на GoLang, клієнтом може бути будь-яка мова програмування, яка підтримує gRPC.

Переваги і недоліки інструменту визначені в процесі аналізу документації, а також локального тестування з інструментом формування gRPC запитів grpcurl. На момент порівняння версія GripMock дорівнювала 2.6.17.

Переваги:

1. Підтримує мокування з використанням статичних фіктивних даних.

2. Підтримує можливість задавати динамічні фіктивні дані з використанням шаблонів.

3. Дає можливість змінювати фіктивні дані без перезапуску макетного серверу gRPC за допомогою окремого REST API серверу.

Недоліки:

1. Не підтримує можливість задавати динамічні фіктивні дані з використанням режиму проксі (перенаправлення запиту на інший сервер).

2. Не підтримує динамічне мокування з використанням станів програми.

3. Інструмент частково підтримує можливість задавати власну логіку обробки запитів: не підтримується можливість створювати користувацькі скрипти, однак є можливість створювати набори фіктивних даних для одного і того ж методу на основі гнучких правил співставлення атрибутів запиту із фіктивними даними, які мають бути відправлені клієнту.

4. Немає можливості налаштувати симуляцію затримки при обробці запиту.

5. Додаткова можливість змінювати фіктивні дані в процесі роботи інструменту дещо ускладнює його: запущена програма має додатковий сервер для обробки змін у фіктивних даних.

В цілому до найголовніших недоліків існуючого рішення можна віднести недостатній функціонал динамічного мокування: відсутність режиму проксі, підтримки мокування за допомогою станів, неможливість задати власну логіку обробки запитів, а також налаштувати затримки обробки запитів. Крім того для спрощення архітектури додатку і зменшення його розміру можна прибрати надлишковий API сервер для керування фіктивними даними [9].

Метою роботи є:

Розробка інструменту API мокування орієнтованого на систему gRPC, який включає всі основні можливості інструментів мокування, на відміну від існуючих рішень:

- мокування статичних фіктивних даних;
- мокування динамічних фіктивних даних з використанням шаблонів, перенаправлення запитів на інший сервер (проксі-режим), користувацьких скриптів і обробників стану;
- підтримка налаштування затримки при обробці запиту;
- захоплення даних запитів і відповідей для подальшого аналізу користувачем gRPC клієнта.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ

2.1 Архітектура додатку

Додаток складається з 5 основних компонентів:

1. Парсер налаштуань – частина програми, яка зчитує налаштування вказані користувачем з конфігураційного файлу і перевіряє їх коректність.

2. Налаштування – набір об’єктів, які зберігають налаштування вказані користувачем для використання іншими компонентами.

3. Компілятор Protobuf – використовує вказані користувачем файли опису інтерфейсів API серверу у форматі .proto, аналізує їх вміст і створює об’єкти опису цих для кожного файлу.

4. Об’єкти опису Protobuf – містять дані структури Protobuf файлів, зокрема наборів повідомлень, сервісів, методів, їх атрибутів та взаємозв’язків, а також визначають логіку серіалізації та десеріалізації повідомлень під час обробки запитів від клієнта до сервера і навпаки. Використовуються іншими компонентами програми.

5. Конфігуратор gRPC серверів – використовує структуру Protobuf файлів, серіалізатори і десеріалізатори повідомлень, а також налаштування від користувача для створення набору gRPC серверів з логікою обробників запитів, відповідно до налаштувань користувача.

6. Обробники запитів – інкапсулюють логіку основних завдань додатку: створення фіктивних даних відповідей, на основі даних запиту, а також логування даних запитів і відповідей.

7. gRPC сервери – процеси, які здійснюють прослуховування gRPC-запитів, кожен сервер надає API взаємодії з сервером відповідно до вказаних користувачем API інтерфейсів, приймає дані запитів, обробляє їх відповідно до встановлених користувачем налаштувань і віддає результат обробки до клієнта.

Архітектура у вигляді діаграми компонентів наведена на рис. 2.1.

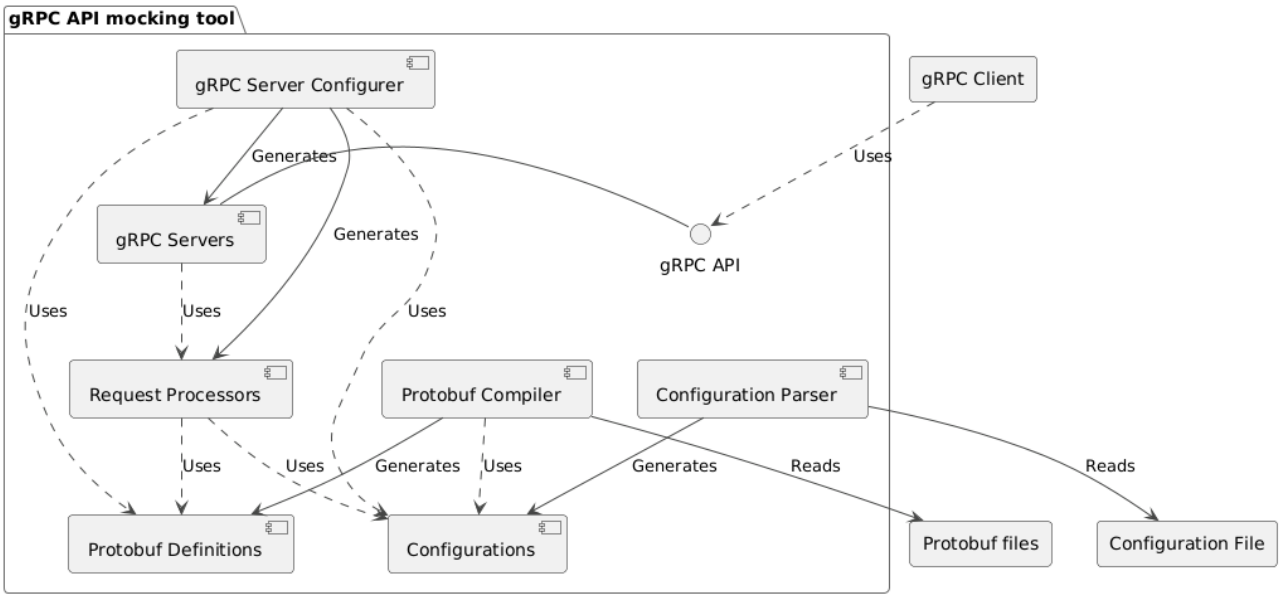

Рисунок 2.1 – Архітектура додатку

Архітектура створена за допомогою мови створення діаграм PlantUML та онлайн сервісу відображення діаграм PlantText [11, 12]. Опис діаграми мовою PlantUML наведений в додатку А.

2.2 Налаштування додатку

Налаштування gRPC API серверів здійснюється виключно за допомогою єдиного конфігураційного файлу, який прозоро відображає API структуру кожного gRPC серверу:

1. Список пар хост-порт, на яких сервер має прослуховувати запити.

2. Псевдонім серверу – коротка назва, прив’язана до кожного запиту і відповіді у логах програми.

3. Перелік Protobuf файлів, у яких записані всі служби і методи, які має обслуговувати сервер.

4. Налаштування фіктивних даних gRPC-відповіді від серверу за назвою сервісу і методу. Дані в цьому випадку можуть бути описані статично або з використанням шаблонів, кожна секція налаштування прозоро описує відповідь gRPC серверу:

1. Секція повідомлень: один або декілька повідомлень, які має відправити сервер.

2. Секція кінцевих метаданих відповіді.

3. Секція помилки з кодом помилки і повідомленням.

4. Секція налаштування проксі.

2.3 Технології та інструменти

Мовою програмування для розробки інструменту обрана мова Python, оскільки:

- мова входить в п’ятірку найпопулярніших мов програмування (Python, Java, JavaScript, C++, C# – відповідно до загального рейтингу популярності мов за PYPL на листопад 2024 року [13]), отже має більше бібліотек, документацій до них і більше відповідей на поширені запитання на форумах;
- підтримка офіційних бібліотек для роботи з gRPC: розробник системи gRPC розробляє і підтримує ці бібліотеки для мови Python [14];
- динамічна типізація мови Python дозволяє реалізувати гнучке налаштування файлу конфігурації.

Найбільш популярними і найкраще підтримуваними бібліотеками для створення серверів і клієнтів на базі gRPC є офіційні бібліотеки від розробника системи gRPC – компанії Google. Вони були обрані в якості базових бібліотек для реалізації взаємодії з gRPC [14].

В якості основної бібліотеки для реалізації можливості одночасної обробки gRPC запитів було обрано бібліотеку asyncio [15].

Для підтримки можливостей створення динамічних фіктивних даних обрана бібліотека jinja2. Вона дозволяє користувачу створювати шаблони, вміст яких обробляється механізмом візуалізації. Візуалізація приймає на вхід рядок, або текстовий файл у будь-якому форматі, що містить синтаксис мови шаблонізації схожий на елементи мови програмування Python. Цей синтаксис має:

1. Змінні.

2. Літерали.

3. Вбудовані функції (тест-функції, функції фільтри), які можуть змінювати значення змінних, а також Python методи.

4. Коментарі.

5. Керуючі конструкції: умовні оператори, цикли, макроси, фільтри, конструкції наслідування, розширення, імпортування та вбудовування інших файлів шаблонів, конструкції створення змінних і присвоювання значень змінних.

6. Оператори: логічні, математичні, оператори порівняння.

Синтаксис шаблонізації дозволяє задавати логіку генерації текстової інформації для будь-якого текстового формату. Результатом механізму візуалізації є рядок, сформований в результаті заданої шаблоном логіки [16].

Загалом для реалізації інструменту використано наступні додаткові Python бібліотеки:

• grpcio 1.67.1 – бібліотека для створення gRPC серверів та клієнтів в Python, підтримує паралельну обробку запитів за допомогою стандартних бібліотек threading – на базі багатопоточного програмування або asyncio – на базі асинхронного циклу подій. Бібліотека відповідальна за прослуховування gRPC запитів, створення TCP з'єднань, встановлення шифрованого каналу зв'язку з використанням SSL/TLS і передачі даних запиту на обробку до скомпільованого коду з Protobuf файлів [14];

• protobuf 5.28.3 – бібліотека, що містить об'єкти опису вмісту Protobuf файлів, так званими дескрипторами і надає інтерфейси для взаємодії з цими об'єктами [17].

• grpcio-tools 1.67.1 – бібліотека для взаємодії з Protobuf файлами. Класи дескриптори Protobuf файлів з цієї бібліотеки використовується в утиліті для динамічної генерації об'єктів опису структур цих файлів. Дескриптори разом із цими об'єктами використовується під час обробки повідомлень на стороні серверу [18].

• grpcio-reflection 1.67.1 – бібліотека, що містить функціонал для додавання служб рефлексії на gRPC сервері [14].

• pyyaml 6.0.2 – бібліотека-конвертер файлів і тексту у форматі YAML, використовується для читання файлу налаштувань додатку, а також для конвертування вмісту логів у YAML формат [19];

• pydantic 2.9.2 – бібліотека керування налаштуваннями проєкту і правилами перевірки допустимих значень сутностей додатку. Використовується для перевірки правильності файлу налаштування та інших сутностей програми [20];

• jinja2 3.1.4 – бібліотека для формування вмісту текстових файлів з використанням мови шаблонізації. Використовується для генерації динамічних фіктивних даних gRPC відповідей створених за допомогою цієї мови [16];

• pyinstaller 6.11.1 – бібліотека, яка виконує компіляцію вихідного коду додатку у виконувані файли, які можна запустити без середовища виконання Python. Використовується для створення виконуваних файлів інструменту [21].

Для тестування роботи програми в процесі розробки використано інструмент для формування gRPC запитів – grpcurl. Це консольний клієнт для формування і відправки gRPC запитів на gRPC сервери. Дозволяє встановлювати повідомлення для запиту на сервер, метадані, а також переглядати даних відповіді: повідомлення, метадані, заголовки, код і повідомлення у випадку помилки [22].

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Структура програми

Додаток є консольною програмою. У додатку використано об’єктно-орієнтований і функціональний підходи. Складається із наступних елементів:

- модуль logs – класи і функції, в яких реалізована логіка для конфігурації логерів програми іншими компонентами додатку, а також логіка утворення формату повідомлень логів;
- модуль config – набір класів-сутностей, що містять дані конфігураційного файлу, а також пов’язані із ними функції і методи із логікою перевірки коректності налаштувань;
- модуль protobuf – набір класів і функцій, що відповідальні за логіку компіляції файлів API специфікації Protobuf у дескриптори та об’єкти опису їхніх структур;
- модуль server – набір класів і функцій відповідальних за створення і конфігурацію gRPC серверів у додатку на основі налаштувань конфігураційного файлу, дескрипторів та об’єктів опису Protobuf;
- модуль server.processors – містить логіку обробки заглушок: логування даних запитів і відповідей, а також формування фіктивних даних відповідей і перенаправлення запитів на зовнішній сервер;
- файл utils.py – набір допоміжних функцій використовуваних всіма частинами програми;
- файл constants.py – набір констант використовуваних всіма частинами програми;
- файл templates.py – набір розширень логіки шаблонів jinja2 для реалізації обробників динамічних фіктивних даних;
- файл args.py – містить логіку роботи інтерфейсу командного рядку програми;

• файл main.py – точка входу в програму, де взаємодіють всі елементи програми;

Вихідний код додатку разом із короткою інструкцією з використання завантажено в хмарний репозиторій GitHub. Інструмент має назву cap-grpc і поширюється за ліцензією відкритого коду MIT, тому може бути використаний будь-яким фахівцем [23]. Python код програми також наведений у додатку Б.

3.2 Встановлення

Встановлення програми можна здійснити 2 способами:

1. Використання скомпільованого виконуваного файлу. Не потребує додаткових компонентів або налаштувань. Користувачу лише необхідно завантажити виконуваний файл програми із репозиторію проєкту. Виконуваний файл можна запустити використовуючи командну оболонку відповідної ОС. Виконувані файли підтримуються для платформ Linux і Windows 64-бітної розрядності.

2. Використання вихідного коду програми. Вимагає встановлення і налаштування середовища виконання, доцільно використовувати у випадку проблем сумісності із цільовою ОС при використанні першого способу. Складається з наступних кроків:

1. Завантажити вихідний код програми з хмарного репозиторію GitHub.

2. Встановити інтерпретатор мови Python версії 3.12, інтерпретатор має бути доступний із командного рядку будь яким доступним методом.

3. Встановити пакетний менеджер pip для Python 3.12, менеджер має бути доступний із командного рядку;

4. Встановити інструмент для керування залежностями та віртуальним оточенням мови Python pipenv;

5. Перейти в папку з вихідним кодом програми;

6. Встановити залежності програми, вказані у файлі Pipfile, за допомогою команди:

```
python -m pipenv install
```

7. Активувати віртуальне оточення програми із встановленими залежностями:

```
python -m pipenv shell
```

8. Після цього програму можна запустити командою з консолі:

```
python main.py
```

Коротка інструкція з описом програми, встановленням та її використанням також знаходиться у файлі README.md в репозиторії проєкту.

3.3 Налаштування програми

Основним способом налаштування програми є використання конфігураційного файлу у форматі YAML. Конфігураційний файл додатку можна вказати через окремий параметр у командному рядку.

У табл. 3.1 і 3.2 описані параметри які можна встановити за допомогою аргументів консолі та файлу налаштувань.

Таблиця 3.1 Аргументи консольного інтерфейсу

Аргумент	**Опис**	**Значення**	**За замовчуванням**
--help	Виводить у потік виводу сторінку допомоги із описом усіх аргументів програми	-	-
-c	Вказує програмі шлях до файлу конфігурації серверів	Рядок без пробілів	cap-grpc.yml
-e	Виводить у потік виводу приклад конфігураційного файлу	-	-

Файл налаштування серверів має формат YAML і складається з 3 основних секцій:

1. servers – тут налаштовується перелік серверів, що використовуватиме програма, зокрема, коротка назва серверу, перелік адрес серверів і сертифікатів для них, Protobuf файлів, які необхідно скомпілювати, вмикати чи не вмикати рефлексію на сервері. Вказується перелік заглушок запитів. Кожна заглушка

відповідає одній комбінації сервіс-метод в межах налаштування одного серверу, це вказує серверу відповідати заглушкою лише на запити спрямовані на цей сервер і метод. Кожна заглушка є об’єктом типу ключ-значення, де ключем є властивість повідомлення яким має відповідати сервер, а значення є відповідним значенням властивості;

2. general_logging_config – тут встановлюються параметри загального логування програми. Зокрема можна обрати формат логування: текстовний рядок або формат YAML, можна обрати властивості, які необхідно відобразити для текстового формату, можна встановити шаблон повідомлення. Можна встановити рівень логування, вимкнути або увімкнути вивід в потік STDOUT, а також перелік файлів, куди необхідно зберегти повідомлення логів. За замовчуванням встановлено рівень логування INFO, вивід в STDOUT і звичайний текстовий формат із логуванням властивостей повідомлення логу і рівня логування.

3. api_logging_config – тут встановлюються параметри загального логування програми. Зокрема можна обрати формат логування: текстовий рядок або формат YAML, можна обрати властивості, які необхідно відобразити для текстового формату, можна встановити шаблон повідомлення. Можна встановити рівень логування, вимкнути або увімкнути вивід в потік STDOUT, а також перелік файлів, куди необхідно зберегти повідомлення логів. За замовчуванням встановлено рівень логування INFO, вивід в STDOUT і звичайний текстовий формат із логуванням повідомлення і рівня логування.

Детальніше про секції налаштування файлу конфігурації, типи даних, їх значення за замовчуванням і обмеження в табл. 3.2.

Таблиця 3.2 Налаштування конфігураційного файлу

Елемент	Тип	За замовчуванням	Опис	Обмеження
Базовий елемент конфігураційного файлу	Config	Об’єкт із значеннями за замовчуванням	Елемент, що встановлює конфігурацію усіх серверів, їх запитів, відповідей і налаштувань логування запитів і відповідей	-
Config.servers	Список	Один елемент із значенням за замовчуванням	Секція опису списку серверів, що мають бути запущені	Як мінімум 1 елемент в списку
Елемент Config.servers	ServerConfig	-	Об’єкт налаштування серверу	-
ServerConfig.alias	Рядок	-	Коротка назва серверу	Не пустий
ServerConfig.sockets	Список	-	Список налаштувань адрес, які має прослуховувати запущений сервер	Рядок має мати формат мережевого сокету, не пустий
Елемент ServerConfig.sockets	SocketsConfig	-	Налаштування адреси, яку має прослуховувати сервер	-
SocketsConfig.socket	Рядок	-	Мережевий сокет, на якому сервер прослуховуватиме запити	Рядок повинен мати формат мережевого сокету – хост або IP адреса і порт через двокрапку, не пустий
SocketsConfig.certificates	CertificatesConfig	-	Налаштування безпечного з’єднання за адресою, якщо вказано, дані, що передаються за цією адресою будуть шифруватися за вказаними сертифікатами	-
CertificatesConfig.certificate	Рядок	-	Шлях до файлу TLS/SSL сертифікату у файловій системі	Файл має існувати у файловій системі, не пустий
CertificatesConfig.key_file	Рядок	-	Шлях до файлу ключа TLS/SSL сертифікату у файловій системі	Файл має існувати у файловій системі, не пустий

Продовження таблиці 3.2

Елемент	Тип	За замовчуванням	Опис	Обмеження
CertificatesConfig.root_certificate	Рядок	-	Шлях до кореневого CA сертифікату для перевірки клієнтських сертифікатів у файловій системі	Файл має існувати у файловій системі
ServerConfig.reflection_enabled	Булевий	Увімкнено	Вмикає або вимикає рефлексію на gRPC сервері, якщо рефлексія увімкнена, клієнти що роблять запити на цей сервер не потребують явно вказаних Protobuf файлів оскільки сервер дозволяє клієнтові отримати API контракти від серверу	Не пустий
ServerConfig.proto_files	Список	-	Список Protobuf файлів для компіляції і подальшого використання утилітою	-
Елемент ServerConfig.proto_files	Рядок	-	Відносний до директорії файлу налаштування або абсолютний шлях до файлу Protobuf у файловій системі	Файл має існувати у файловій системі, вміст файлу має відповідати формату запису Protobuf IDL. Імена файлів не повинні повторюватися, сутності у Protobuf файлах не повинні дублюватися, не пустий
ServerConfig.proto_files_base_dir	Рядок	-	Базова директорія для файлів описаних у розділі ServerConfig.proto_files, якщо програма визначила базову директорію неправильно	Директорія має існувати у файловій системі
ServerConfig.mocks	Об’єкт типу ключ-значення	-	Описує конфігурацію заглушок для серверу	-

Продовження таблиці 3.2

Елемент	Тип	За замовчуванням	Опис	Обмеження
Ключ ServerConfig.mocks	Рядок	-	Ім’я сервісу для якого у значенні буде вказане ім’я методу для заглушки. Ім’я сервісу повинно мати префікс пакету, до якого відноситься сервіс	Вказане ім’я сервісу з префіксом пакету має існувати у Protobuf файлах
Значення ServerConfig.mocks	Об’єкт типу ключ-значення	-	Описує до якого сервісу має відноситися заглушка у значенні за ключем	Не пустий
Ключ значення ServerConfig.mocks	Рядок	-	Ім’я методу для якого у значенні буде вказане тіло повідомлення для gRPC вілповіді. Ім’я методу не повинно мати префікс пакету, до якого відноситься метод	Вказане ім’я методу має існувати в межах сервісу у Protobuf файлах
Значення у значенні ServerConfig.mocks	ResponseMockConfig або рядок з мовою шаблонізації jinja2	-	Конфігурація заглушки для конкретного сервісу і методу з Protobuf файлів	Не пустий
ResponseMockConfig.messages	Об’єкт типу ключ-значення чи список таких об’єктів для серверного стримінгу або рядок з мовою шаблонізації jinja2	-	Налаштування тіла відповіді (повідомлення) у вигляді об’єкту ключ значення. Ключ має відповідати одному із властивостей відповідного повідомлення, що повертає метод у Protobuf файлі. Якщо властивості за ключем не існує, тоді встановлення властивості у відповіді буде проігноровано. Якщо значення властивості має невідповідний тип, буде використано значення за замовчуванням.	-
ResponseMockConfig.metadata	Об’єкт типу ключ-значення або рядок з мовою шаблонізації jinja2	-	Налаштування метаданих відповіді у вигляді пар ключ значення	Лише ті ідентифікатори, що вказані в секції Config.responses

Продовження таблиці 3.2

Елемент	Тип	За замовчуванням	Опис	Обмеження
Ключ ResponseMockConfig.metadata	Рядок	-	Ключ який необхідно додати в метадані	Тільки символи латинського алфавіту в нижньому регістрі, цифри, і символи нижнє підкреслення, тире і крапка, довжина від 1 до 256 символів, не пустий
Значення ResponseMockConfig.metadata	Рядок, який може включати мову шаблонізації jinja2	-	Значення які необхідно додати в метадані за ключем	Тільки латинські символи в нижньому регістрі, цифри, і символи нижнє підкреслення, тире і крапка, довжина від 1 до 8192 символів, не пустий
ResponseMockConfig.error	ErrorConfig або рядок з мовою шаблонізації jinja2	Не вказано	Об’єкт налаштування gRPC відповіді з помилкою	-
ErrorConfig.code	Ціле число або рядок з мовою шаблонізації jinja2	2, що відповідає gRPC коду UNKNOWN	Код помилки, яку необхідно повернути	Число від 1 до 16 включно, що відповідає кодам помилок gRPC
ErrorConfig.details	Рядок, який може включати мову шаблонізації jinja2	Пустий рядок	Повідомлення помилки, яку необхідно повернути	Не пустий
ResponseMockConfig.proxy	ProxyConfig	Не вказано	Конфігурація проксі режиму	-
ResponseMockConfig.seconds_delay	Ціле або дробове число або рядок з мовою шаблонізації jinja2	Не вказано	Встановлює часову затримку при обробці запиту	Більше 0

Продовження таблиці 3.2

Елемент	Тип	За замовчуванням	Опис	Обмеження
ProxyConfig.socket	Рядок або рядок з мовою шаблонізації jinja2	-	Адреса у вигляді пари хост-порт на яку сервер має перенаправити запит від клієнта	Не пустий
ProxyConfig.seconds_timeout	Ціле чи дробове число або рядок з мовою шаблонізації jinja2	10	Встановлює тайм-аут для запиту на інший сервер у секундах	Має бути більше 0
Config.general_logging_config	LoggingConfig	LoggingConfig об’єкт у якого: format_line = %(levelname)s: %(message)s, інші значення за замовчуванням	Налаштування загального логування в програмі	Не пустий
Config.api_logging_config	LoggingConfig	LoggingConfig об’єкт у якого: формат YAML, виведення в STDOUT увімкнено, властивості, які логуються в форматі YML: message, request_message, response_message, method, code, error_details, metadata, timestamp	Налаштування загального логування API запитів у програмі	Не пустий
LoggingConfig.console	Булевий	Увімкнено	Встановлює виводити повідомлення в STDOUT або ні	Не пустий

Продовження таблиці 3.2

Елемент	Тип	За замовчуванням	Опис	Обмеження
LoggingConfig.files	Список	Не вказано	Визначає, шляхи до файлів у файловій системі, куди треба зберігати повідомлення	-
Елемент LoggingConfig.files	Рядок	Не вказано	Шлях до файлу, куди треба зберігати повідомлення логів	Не пустий
LoggingConfig.level	Рядок, одне із списку значень: CRITICAL, FATAL, ERROR, WARNING, INFO, DEBUG	INFO	Визначає рівень логування	Не пустий
LoggingConfig.format	Рядок, одне із списку значень: TEXT, YAML	TEXT	Визначає формат повідомлень логів, TEXT – стандартний формат логів, YAML – повідомлення у форматі YAML	Не пустий
LoggingConfig.format_line	Рядок	%(message)s	Встановлює патерн формату повідомлень логів. Для формату TEXT дозволяє визначити, які властивості логів виводити і формат для них. Для формату YAML дозволяє визначити тільки перелік властивостей, які необхідно показувати, хоча формат і можна визначити в цьому рядку, він завжди відповідатиме формату YAML. Кожна властивість у рядку має бути вказана тільки за шаблоном: %(<ім’я властивості>)s.	-

Налаштування фіктивних даних здійснюється в секції servers.{сервер із списку}.mocks.{ім’я сервісу}.{ім’я методу}. Як видно із положення секції, кожен метод кожного налаштованого серверу має власний об’єкт налаштування фіктивних даних відповіді для запиту. Він являє собою об’єкт типу ключ-значення формату YAML кожен атрибут якого описує дані gRPC відповіді. Ця секція конфігураційного файлу може мати елементи мови шаблонізації jinja2. Процес візуалізації цієї секції спрацьовує кожного разу при обробці запиту, після чого візуалізований об’єкт перетворюється на одне або декілька повідомлень, метадані, код помилки, деталі помилки, задає налаштування проксі або затримку обробки запиту.

3.4 Принцип роботи

Програма має 3 робочі стани: ініціалізація, обробка запитів та завершення роботи.

В стані ініціалізації програма знаходиться після того, як користувач запустив програму через термінал операційної системи. Стан ініціалізації складається з наступних кроків:

1. Читання аргументів командного рядку. Аргументи перевіряються на правильність введення відповідно до табл. 3.1.

2. Читання конфігураційного файлу за вказаним шляхом. Програма перевіряє правильність встановлених налаштувань, відповідно до обмежень табл. 3.2.

3. Читання файлів опису інтерфейсів Protobuf для кожного серверу. Після зчитування додаток компілює їх у дескриптори Protobuf: для встановленого набору файлів кожного окремого серверу програма створює тимчасову директорію, в яку генерує бінарний файл з описом дескрипторів у форматі .pb. Цей файл одразу зчитується після чого тимчасова директорія разом з файлом опису видаляється. Така логіка обробки пов’язана з особливостями роботи інструменту для компіляції Protobuf файлів protoc, вбудованого в інструмент, який є частиною бібліотеки grpcio-tools. В процесі інтерпретації вміст файлів перевіряється відповідно до специфікації Protobuf.

4. Створення обробників gRPC запитів на основі налаштувань з конфігураційного файлу і об’єктів Protobuf файлів. Встановлюється відповідність між методами сервісів і обробниками для них. Програма налаштовує gRPC сервери із цими обробниками.

5. Розгортання налаштованих серверів за встановленими адресами.

Будь-яка критична помилка пов’язана із перевіркою налаштувань програми в стані ініціалізації, а також Protobuf файлів призводить до зупинки її роботи, при цьому інформація про помилку виводиться в потік STDOUT. Помилки пов’язані з компіляцією файлів опису інтерфейсів виводяться в потік STDERR.

В стані обробки запитів в програмі працюють gRPC сервери, які виконують прослуховування за встановленими адресами. Хоча запущених серверів може бути багато, інструмент являє собою один однопоточний процес, але запити при цьому оброблятися паралельно, як в межах одного серверу так і в межах усіх запущених серверів. Це досягається завдяки можливостям асинхронного циклу подій бібліотеки asyncio. Обробка запитів в цьому стані відбувається за наступним алгоритмом:

1. Сервер отримує всі метадані і повідомлення запиту і логує їх для подальшого аналізу користувачем відповідно до встановлених налаштувань логування.

2. Для обробки запиту сервер використовує об’єкт налаштування секції фіктивних даних. В момент обробки кожного запиту від серверу ця секція проходить процес візуалізації механізмом jinja2, після візуалізації ця секція використовується для формування відповіді.

3. Якщо встановлена затримка обробки запиту, програма призупиняє обробку на встановлений час: отримавши дані запиту від клієнта сервер тримає з’єднання, але нічого не відправляє клієнту.

4. Якщо налаштовано режим проксі, сервер приймає дані запиту: всі повідомлення і метадані, після чого відправляє їх за встановленою адресою іншому gRPC серверу, отримана від сервера відповідь відправляється клієнту з усіма

метаданими і повідомленнями. Всі інші атрибути об’єкту налаштування фіктивних даних ігноруються.

5. Якщо встановлені дані повідомлень, сервер відправляє ці дані клієнту.

6. Якщо встановлені метадані, сервер відправляє їх клієнту після останнього відправленого повідомлення.

7. Перед відправкою до клієнта сервер обов’язково логує всі дані відповіді: повідомлення і метадані у випадку успішної відповіді, код і деталі у випадку відповіді з помилкою відповідно до встановлених налаштувань логування.

Якщо в стані обробки запитів в програмі сталася внутрішня помилка або помилка пов’язана із обробкою запиту всі робота програми не буде перервана, при цьому деталі помилки будуть відображені у логах програми, а сервер буде відповідати кодом 2 (Unknown) і повідомленням "Mock API server internal error".

Стан завершення роботи виконує зупинку всіх розгорнутих серверів і звільняє всі ресурси: зокрема закриває відкриті в проксі режимі з’єднання з іншими серверами. Ініціювати завершення роботи інструменту стан можна послави POSIX сигнали SIGINT або SIGTERM.

3.5 Функції мокування

Усі можливі варіанти динамічного мокування налаштовуються тільки для секції опису відповідей (servers.{об’єкт серверу}.mocks.{назва сервісу}.{назва методу}) конфігураційного файлу. Вміст цієї властивості обробляється механізмом візуалізації jinja2 для формування вмісту відповіді залежно від встановленого користувачем шаблону в момент обробки відповідного запиту. Інші частини конфігураційного файлу не обробляються з використанням цього механізму і будуть проігноровані програмою, якщо вони встановлені.

Динамічне мокування з використанням шаблонів користувач може реалізувати за допомогою змінних jinja2. Під час обробки кожного запиту утиліта додає в контекст шаблону відповіді змінні, що зберігають локальні дані запиту. Ці змінні можуть бути використані для генерації вмісту відповіді з використанням jinja2 виразів. Вирази записуються за допомогою символів «{{» і «}}».

Перелік усіх змінних контексту запиту, які можна використовувати для динамічного формування вмісту показані в табл. 3.3.

Таблиця 3.3 Змінні контексту запиту

Назва змінної	**Опис**	**Вміст**
message	Містить дані першого повідомлення gRPC запиту	Відповідає вмісту першого повідомлення запиту, являє собою пари ключ значення, де у значеннях можуть бути інші об'єкти типу ключ-значення, списки, рядки, числа, булеві значення, може бути пустим, якщо повідомлення не відправлялись клієнтом
messages	Містить список усіх повідомлень gRPC запиту	Містить список об'єктів того самого типу, що і змінна message. Відповідає всім повідомленням відправленим клієнтом
metadata	Містить всі метадані запиту	Є об'єктом ключ-значення, які відповідають метаданим запиту
sockets	Містить адреси на яких працює сервер, що обробляє поточний запит	Є списком, де кожен елемент є парою хост-порт
alias	Містить псевдонім серверу, що обробляє поточний запит	Є рядком, що відповідає псевдоніму серверу або порожнім значення, якщо його не вказано користувачем
method	Містить дані методу для якого викликаний запит	Є об'єктом типу ключ значення з наступними атрибутами: • name – ім'я методу без пакету; • input_message.name – повна назва вхідних повідомлень, вказана у Protobuf; • output_message.name – повна назва вихідних повідомлень, вказана у Protobuf; • input_message.streaming – булеве значення, яке вказує чи увімкнена клієнтська потокова передача відповідно до Protobuf; • output_message.streaming – булеве значення, яке вказує чи увімкнена серверна потокова передача відповідно до Protobuf.
service	Містить дані сервісу для якого викликаний запит	Є об'єктом типу ключ-значення з наступними атрибутами: • name – ім'я сервісу без пакету • full_name – повне ім'я сервісу з префіксом пакету відповідно до Protobuf • methods – об'єкт типу ключ-значення, ключем якого є назва методу в рамках поточного сервісу, а значенням є той самий об'єкт, який описує змінна method

Використовуючи кожну змінну отримати значення атрибуту можна за допомогою символу «.». Наприклад service.full_name поверне повне ім’я сервісу для якого обробляється запит. Отримати до об’єкту в списку можна використовуючи той самий символ «.», наприклад messages.20 поверне 19 повідомлення (1 повідомлення має індекс 0). При спробі отримати доступ до значення змінної за атрибутом, якого не існує або за індексом якого немає в списку, результатом виразу буде порожній рядок.

Приклад конфігураційного файлу з використанням шаблонів показаний на рис. 3.1. Повідомлення, яке повертатиме сервер для методу SendEvent завжди матиме ті ж атрибути, що і повідомлення запиту.

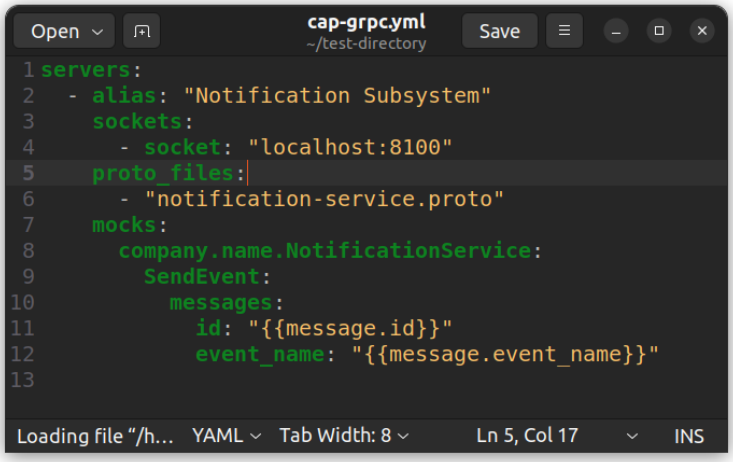

Рисунок 3.1 – приклад створення фіктивних даних за допомогою шаблонів

Динамічне мокування з використанням власної логіки обробки користувач може реалізувати за допомогою всього доступного функціоналу jinja2, а також додаткових змінних. що існують в контексті запиту (табл. 3.3). Крім цього контекст запиту містить 3 додаткових функції для розширення можливостей мокування з використанням скриптів. Функції і їх опис показані в табл. 3.4.

Таблиця 3.4 Додаткові функції для мокування з використанням скриптів

Назва	**Опис і приклади**	**Вхідні і вихідні дані**
insert	Приймає шлях до файлу, зчитує його і повертає вміст: insert('file.txt') insert('/folder/file.txt', True)	Аргументи: • абсолютний або відносний шлях до файлу • кешування, якщо встановлено, програма кешуватиме вміст файлу в оперативній пам’яті Результатом є вміст файлу
shell	Виконує команду в командній оболонці: shell('ping', '-c 5', 'google.com').code shell('jq', stdin='{"id": "29"}').stdout shell('bash', '/my-script.sh').stdout	Аргументи: • команда оболонки • список аргументів для команди • вміст STDIN потоку (встановлюється тільки за ключем stdin). Результатом є об’єкт типу ключ-значення з атрибутами: • stdout – вміст потоку STDOUT • stderr – вміст потоку STDOUT • code – код результату
relative	Перетворює відносний шлях об’єкту файлової системи в абсолютний відносно розміщення конфігураційного файлу: relative('my-file.txt')	Аргументи: • відносний шлях до файлу або директорії у файловій системі Результатом є абсолютний шлях до вказаного об’єкту

На рис. 3.2 наведений приклад створення фіктивних даних з використанням користувацьких скриптів. Сформована в прикладі відповідь завжди

міститиме 20 повідомлень, кожне матиме властивість timestamp на 1 менше ніж попереднє.

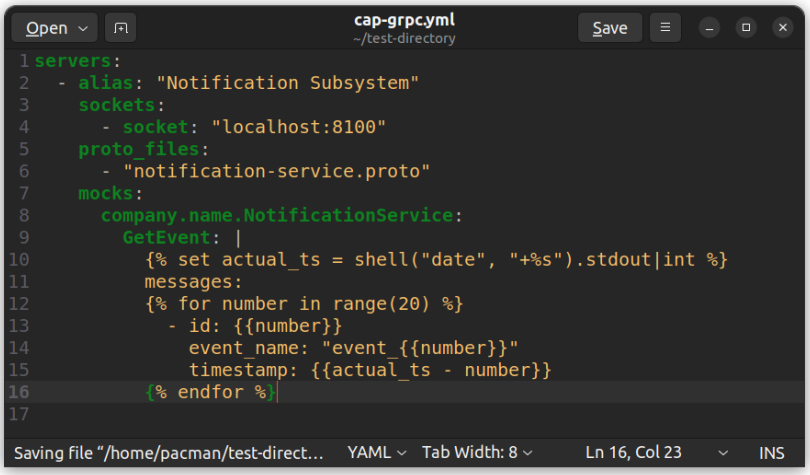

Рисунок 3.2 – приклад налаштування фіктивних даних за допомогою користувацьких скриптів

Динамічне мокування з використанням проксі режиму користувач може реалізувати за допомогою атрибутів proxy.socket та proxy.seconds_timeout, приклад такого налаштування наведений на рис. 3.3.

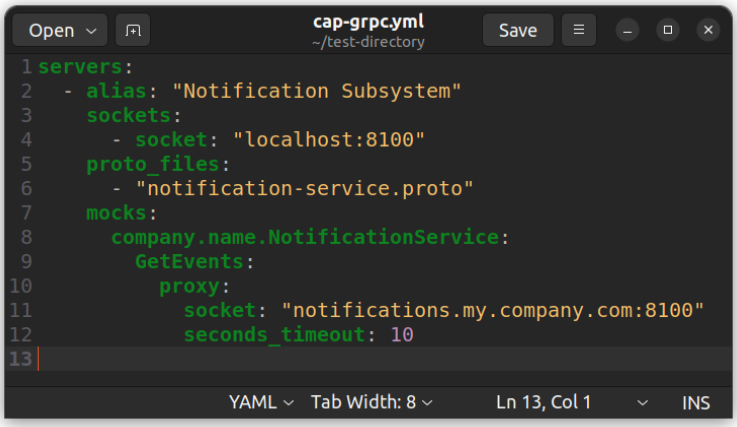

Рисунок 3.3 – приклад налаштування динамічних фіктивних даних з використанням режиму проксі

Динамічне мокування з використанням станів програми користувач може реалізувати за допомогою можливостей шаблонів і скриптів jinja2 і створених

для цього додаткових функцій роботи із станом серверу. Стан серверу є змінною, що існує в контексті кожного серверу. Це значення є спільним для всіх запитів на одному і тому ж сервері, але є різним для кожного окремого сервера встановленого у файлі налаштування. Опис функцій для роботи із станом серверу наведений в табл. 3.5.

Таблиця 3.5 Функції для роботи із станом серверу

Назва	**Опис і приклади**	**Вхідні і вихідні дані**
get_state	Повертає змінну, яка зберігає стан серверу. Якщо стан не було встановлено раніше, функція поверне стан за замовчуванням, що відповідає значенню initial.	Не приймає аргументів. Результатом є змінна, яка зберігає стан серверу
set_state	Встановлює змінну, яка зберігає стан серверу. Стан може бути змінною будь-якого типу.	Єдиним аргументом є змінна, що зберігатиме стан програми. Не повертає значень.

Приклад налаштування фіктивних даних з використанням станів показаний на рис. 3.4.

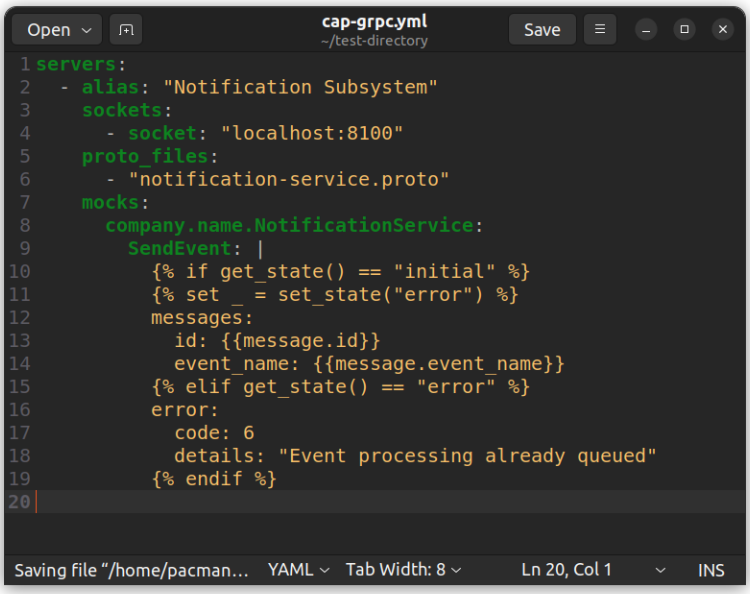

Рисунок 3.4 – Приклад налаштування фіктивних даних з використанням станів

3.6 Приклад використання

Використаємо додаток для підміни системи аутентифікації в мікросервісному додатку. Нехай необхідно протестувати зміни в кодовій базі після виправлення помилки в логіки бекенд серверу, який інкапсулює в собі сервіс управління акаунтами користувачів компанії. Були додані модульні тести і необхідно виконати контрольну перевірку роботи зміненої логіки додатку в умовах наближених до робочого середовища. Нехай сервіс має назву Partner Backend. Partner Backend в свою чергу залежить від підсистеми ідентифікації і керування доступом (Identity and Access Management), яка використовує gRPC для обробки запитів від клієнтів. Підсистема є комплексною і складається з декількох сервісів:

1. Сервер для адміністрування користувачами і їхнім доступом з gRPC API інтерфейсом.

2. Сервер авторизації з gRPC API інтерфейсом.

3. База даних, яка зберігає список користувачів, їхні ролі і доступ до сервісів.

Сервер адміністрування в свою чергу робить запит на сервер збору подій мікросервісного додатку, коли відбулися зміни даних користувачів.

Для того, щоб розгорнути Partner Backend в локальному середовищі потрібно запустити залежну підсистему Identity and Access Management, а також компоненти від яких залежить ця система. Оскільки підсистема є доволі комплексною, вона потенційно вимагає часу на те, щоб розібратися із принципами її роботи, налаштувати і розгорнути її локально. Крім того система Identity and Access Management може вимагати для своєї роботи інших залежних компонентів, які теж необхідно запустити. Разом це вимагає великої кількості часу і обчислювальних ресурсів від локального середовища тестування. Для такого завдання доцільно використати інструмент gRPC API мокування cap-grpc.

Нехай сервер адміністрування користувачами використовує API специфікацію вказану у файлі administration.proto. Вміст файлу показаний на рис. 3.5.

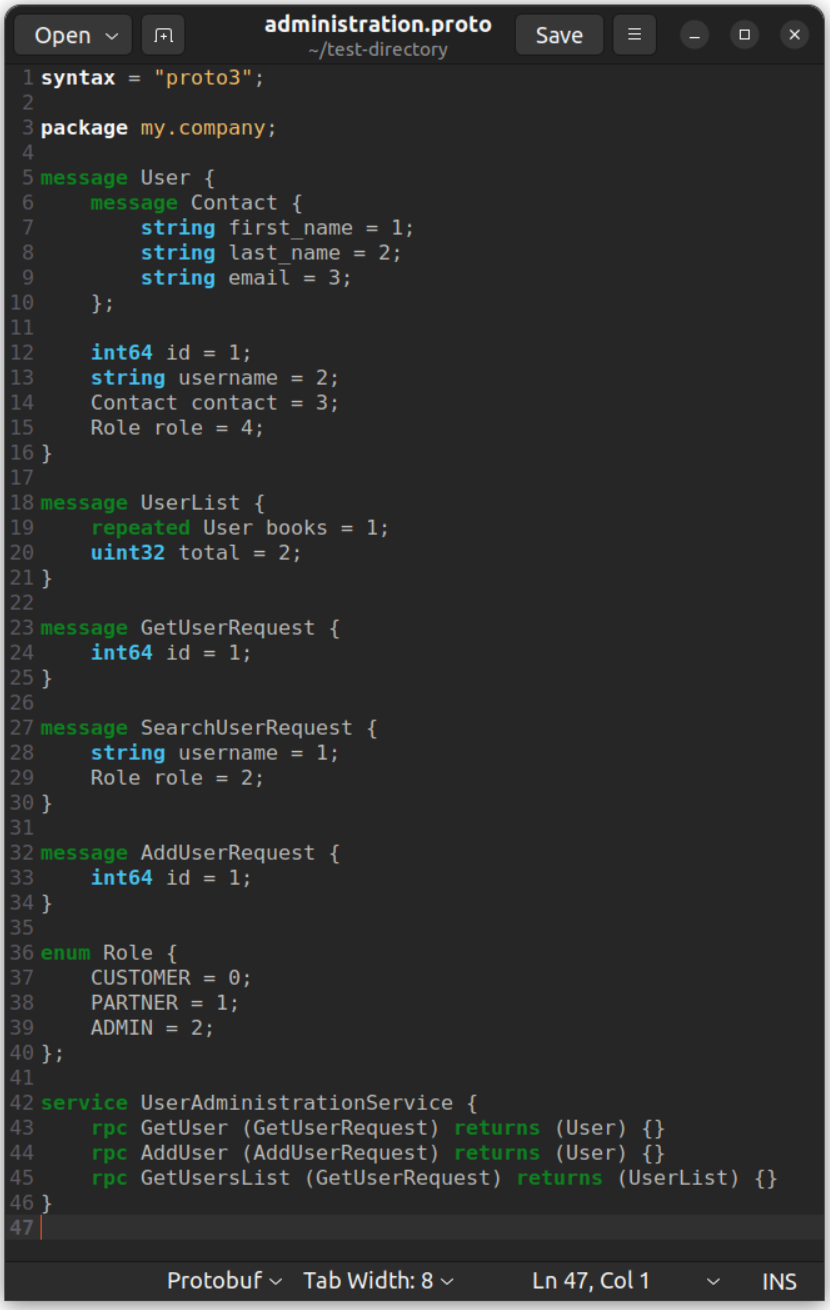

Рисунок 3.5 – Приклад Protobuf файлу API специфікації серверу

Розробник аналізує Protobuf файл і створює файл конфігурації для інструменту cap-grpc з описом фіктивного серверу адміністрування користувачами і заглушками, які повертатиме сервер для серверу Partner Backend. Конфігураційний файл показаний на рис. 3.6.

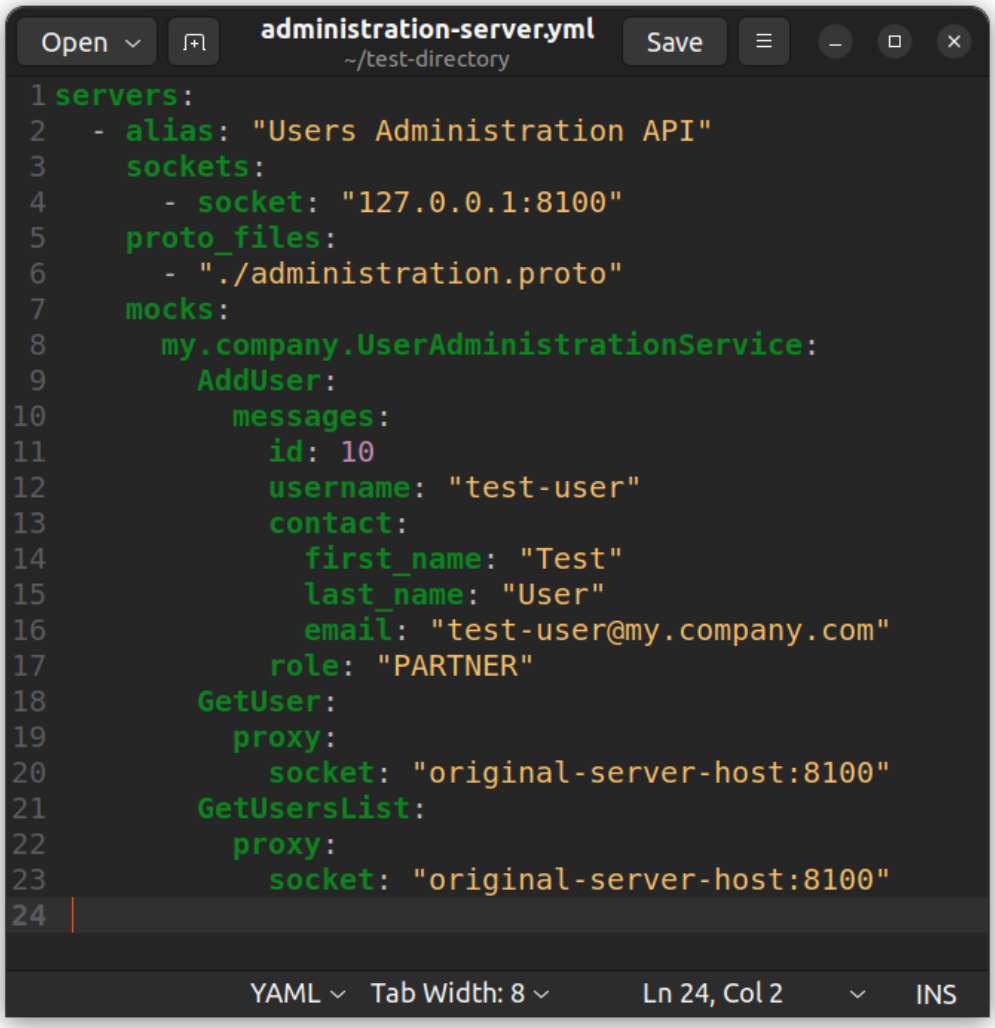

Рисунок 3.6 – Вміст конфігураційного файлу

Після створення конфігураційного файлу запустимо утиліту cap-grpc з вказівкою шляху до конфігураційного файлу. Запуск утиліти показаний на рис. 3.7.

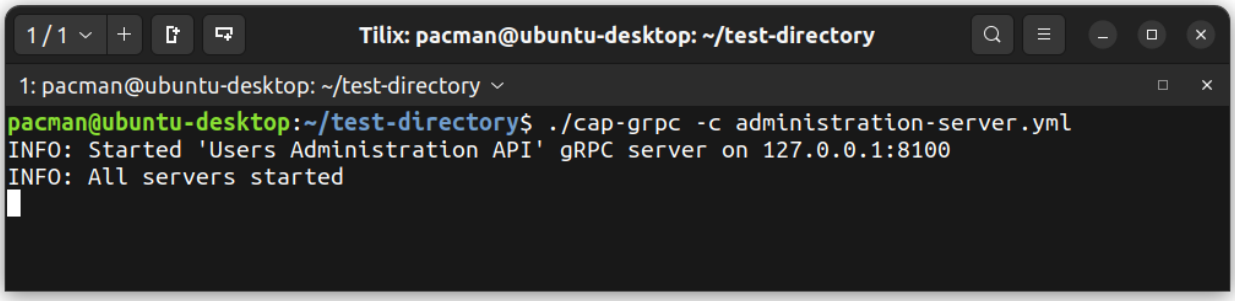

Рисунок 3.7 – Запуск утиліти cap-grpc

Перевіримо, що сервер фіктивний сервер дійсно працює: використовуючи інструмент grpcurl відправимо запит на фіктивний сервер (рис. 3.8).

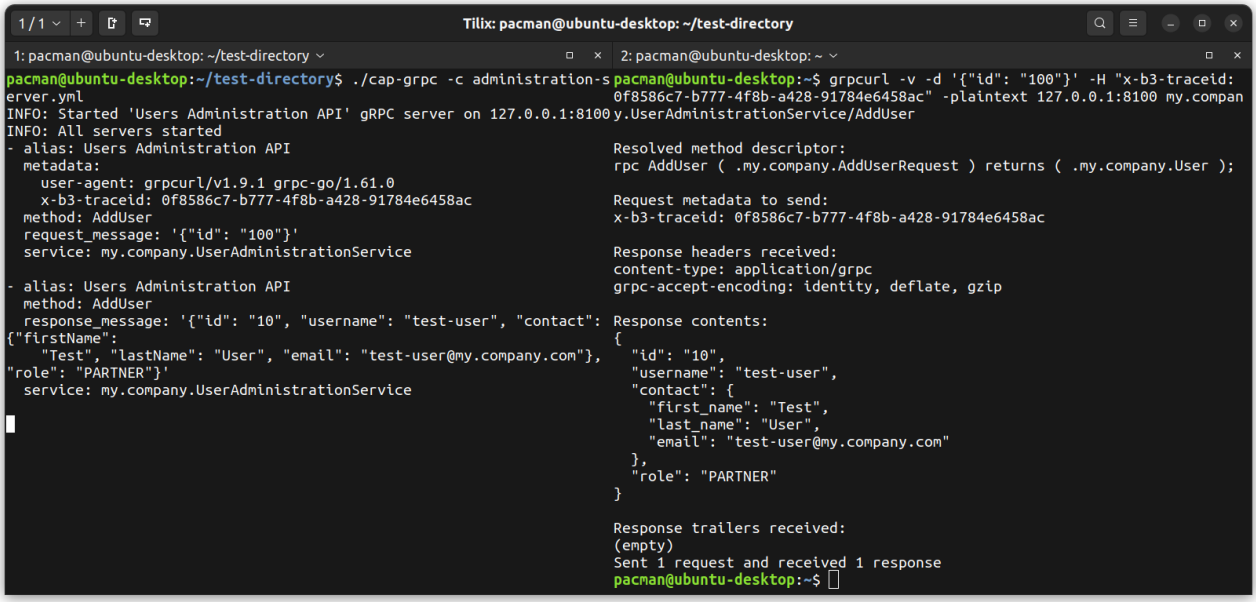

Рисунок 3.8 – Перевірка роботи фіктивного серверу з використанням grpcurl

Сервер працює відповідно до встановлених налаштувань. Тепер його можна використовувати для локального запуску, відлагодження і тестування цільового сервісу Partner Backend. Для цього необхідно лише встановити в налаштуваннях додатку адресу серверу макету замість адреси дійсного серверу Identity and Access Management.

ВИСНОВКИ

Проведений аналіз найпопулярніших рішень мокування API для архітектурного стилю REST API, визначено їх основні функції. Виявлено недоліки існуючого рішення API мокування GripMock, орієнтованого на систему клієнт-серверної взаємодії gRPC. Основним недоліком інструменту є недостатній функціонал динамічного мокування: відсутність режиму проксі, підтримки мокування за допомогою станів, неможливість задати власну логіку обробки запитів, а також налаштувати затримки обробки запитів.

Створено власний інструмент мокування, орієнтований виключно на gRPC API сервери, на мові програмування Python з підтримкою статичного мокування і всіх типів динамічного мокування: проксі режиму, використання шаблонів і користувацьких скриптів, обробки станів. В основі механізму динамічного мокування лежить механізм візуалізації текстових даних з використанням мови шаблонів jinja2. Інструмент має простий консольний інтерфейс з налаштуванням через конфігураційний файл.

Розроблена утиліта мокування дозволяє вирішити проблему залежності мікросервісу від інших компонентів додатку під час його відлагодження або тестування в ізольованому середовищі в умовах наближених до виробничого середовища. Інструмент може замінити API сервіси, від яких залежить цільовий додаток, замість їх повного або частково розгортання, зменшивши час необхідний на тестування і відлагодження, а також зменшивши витрачені ресурси.

Вихідний код розробленого інструменту разом з інструкцією встановлення та використання розміщено в хмарному репозиторії GitHub. Додаток розповсюджується за ліцензією вільного програмного забезпечення MIT, тому може бути використаний будь-який фахівцем для створення фіктивних gRPC API серверів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Transitioning from monolith to microservices handbook. Semaphore. URL: https://semaphoreci.com/wp-content/uploads/2022/09/Monolith_to_Microservices_Handbook-1.pdf (дата звернення: 05.12.2024).

2. How Microservices Communicate with Each Other. Microservices Testing Platform | Signadot. URL: https://www.signadot.com/blog/how-microservices-communicate-with-each-other (дата звернення: 05.12.2024).

3. Ozkaya M. Microservices Communications. Medium. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71 (дата звернення: 05.12.2024).

4. Petruk M. Best Practices for SaaS Development Using Microservices. Wesoftyou. URL: https://wesoftyou.com/outsourcing/best-practices-for-saas-development/ (дата звернення: 05.12.2024).

5. Kamil I. 10 Best API Mocking Tools (2024 Review). Apidog An integrated platform for API design, debugging, development, mock, and testing. URL: https://apidog.com/blog/best-api-mock-tools/ (дата звернення: 05.12.2024).

6. Bhattacharya B. What is API mocking and what are the benefits?. Tyk API Management. URL: https://tyk.io/learning-center/api-mocking/ (дата звернення: 05.12.2024).

7. Arman. The Complete Guide to API Mocking: Essentials and Top Tools - February 2024 Review. Testfully. URL: https://testfully.io/blog/mock-api/ (дата звернення: 05.12.2024).

8. Beschokov M. What is gRPC? Meaning, Architecture, Advantages. Wallarm | Integrated App and API Security Platform. URL: https://www.wallarm.com/what/the-concept-of-grpc (дата звернення: 05.12.2024).

9. Core concepts, architecture and lifecycle. gRPC. URL: https://grpc.io/docs/what-is-grpc/core-concepts/ (дата звернення: 05.12.2024).

10. Introduction. GripMock. URL: https://gripmock.org/guide/introduction/ (дата звернення: 05.12.2024).
11. PlantUML Language Reference Guide. PlantUML.com. URL: https://plantuml.com/en/guide (дата звернення: 05.12.2024).
12. PlantText UML Editor. URL: https://www.planttext.com/ (дата звернення: 05.12.2024).
13. PYPL PopularitY of Programming Language index. PYPL Index. URL: https://pypl.github.io/PYPL.html (дата звернення: 05.12.2024).
14. Documentation. gRPC. URL: https://grpc.io/docs/ (дата звернення: 05.12.2024).
15. Asynchronous I/O. Python documentation. URL: https://docs.python.org/3/library/asyncio.html (дата звернення: 05.12.2024).
16. Jinja – Jinja Documentation (3.1.x). Pallets. URL: https://jinja.palletsprojects.com/en/stable/ (дата звернення: 05.12.2024).
17. Protocol Buffer Basics: Python. Protocol Buffers Documentation. URL: https://protobuf.dev/getting-started/pythontutorial/ (дата звернення: 05.12.2024).
18. gRPC Python Tools. GitHub. URL: https://github.com/grpc/grpc/blob/master/tools/distrib/python/grpcio_tools/README.rst (дата звернення: 05.12.2024).
19. PyYAML Documentation. URL: https://pyyaml.org/wiki/PyYAMLDocumentation (дата звернення: 05.12.2024).
20. BaseModel - Pydantic. Pydantic. URL: https://docs.pydantic.dev/latest/api/base_model/ (дата звернення: 05.12.2024).
21. PyInstaller Manual – PyInstaller 6.11.1 documentation. PyInstaller. URL: https://pyinstaller.org/en/stable/ (дата звернення: 05.12.2024).
22. GitHub - fullstorydev/grpcurl: Like cURL, but for gRPC: Command-line tool for interacting with gRPC servers. GitHub. URL: https://github.com/fullstorydev/grpcurl (дата звернення: 05.12.2024).
23. Кіхтенко Д. cap-grpc. GitHub. URL: https://github.com/DmitroKihtenko/cap-grpc (дата звернення: 05.12.2024).

ДОДАТОК А

```
@startuml

!theme vibrant
rectangle "gRPC Client" as C
rectangle "Configuration File" as CF
rectangle "Protobuf files" as PF
package "gRPC API mocking tool" {
    component "Configuration Parser" as CP
    component "Configurations" as CD
    component "Protobuf Compiler" as PC
    component "Protobuf Definitions" as PD
    component "gRPC Server Configurer" as SC
    component "gRPC Servers" as S
    interface "gRPC API" as A
    component "Request Processors" as P
}

CP --> CF: Reads
CP --> CD: Generates

PC ..> CD: Uses
PC --> PF: Reads
PC --> PD: Generates

SC ..> CD: Uses
SC ..> PD: Uses
SC --> P: Generates
SC --> S: Generates
A - S

C ..> A: Uses
S ..> P: Uses
P ..> CD: Uses
P ..> PD: Uses

@enduml
```

ДОДАТОК Б

config/__init__.py:

```
from pydantic import ValidationError

from config.model import Config
from utils import get_validation_err_msg, get_exception_error


def parse_config(raw_config: dict) -> Config:
    try:
        config = Config.model_validate(raw_config)
        return config
    except ValidationError as e:
        raise IOError(
            "Config file parsing error. " +
            get_validation_err_msg(e)
        )
    except ValueError as e:
        raise IOError(
            "Config file parsing error. " +
            get_exception_error(e)
        )
```

config/model.py:

```
import logging
import sys
from enum import Enum
from typing import Any, Annotated

from grpc import StatusCode
from pydantic import (
    BaseModel, RootModel, AfterValidator, ConfigDict
)

from logs import LoggerConfig
from logs.formatters import YamlFormatter
import config.validators as v


MetadataKey = Annotated[str, AfterValidator(v.validate_grpc_meta_key)]
MetadataValue = Annotated[str, AfterValidator(v.validate_grpc_meta_value)]
```

```
GRPCErrorCode = Annotated[int, AfterValidator(
    v.validate_grpc_error_status_code
)]
FormatLine = Annotated[str, AfterValidator(v.validate_logging_keys)]


class BaseConfigModel(BaseModel):
    model_config = ConfigDict(extra="forbid")


class ErrorConfig(BaseConfigModel):
    code: GRPCErrorCode | str = StatusCode.UNKNOWN.value[0]
    details: str = ""


class ProxyConfig(BaseConfigModel):
    socket: str
    seconds_timeout: float | str | None = None


class ResponseMockConfig(BaseConfigModel):
    messages: dict[str, Any] | list[dict[str, Any]] | str = {}
    trailing_meta: str | dict[MetadataKey, MetadataValue] = {}
    error: ErrorConfig | None = None
    seconds_delay: str | float | None = None
    proxy: ProxyConfig | None = None


class GrpcMockData(RootModel):
    root: dict[str, dict[str, ResponseMockConfig | str | None]]


class CertificatesConfig(BaseConfigModel):
    certificate: str
    key_file: str
    root_certificate: str | None = None


class SocketsConfig(BaseConfigModel):
    socket: str
    certificates: CertificatesConfig | None = None
```

```
class LoggingLevel(str, Enum):
    CRITICAL = "CRITICAL"
    FATAL = "FATAL"
    ERROR = "ERROR"
    WARNING = "WARNING"
    INFO = "INFO"
    DEBUG = "DEBUG"

    def to_int_value(self) -> int:
        if self is LoggingLevel.CRITICAL:
            return logging.CRITICAL
        elif self is LoggingLevel.FATAL:
            return logging.FATAL
        elif self is LoggingLevel.ERROR:
            return logging.ERROR
        elif self is LoggingLevel.WARNING:
            return logging.WARNING
        elif self is LoggingLevel.INFO:
            return logging.INFO
        elif self is LoggingLevel.DEBUG:
            return logging.DEBUG
        else:
            return logging.FATAL


class LoggingFormat(str, Enum):
    TEXT = "text"
    YAML = "yaml"


class LoggingConfig(BaseConfigModel):
    console: bool
    files: list[str] = []
    level: LoggingLevel = LoggingLevel.INFO
    format: LoggingFormat = LoggingFormat.TEXT
    format_line: FormatLine = "%(message)s"

    def get_loggers_config(self) -> LoggerConfig:
        if self.format == LoggingFormat.TEXT:
```

```
            formatter = logging.Formatter(self.format_line)
        else:
            formatter = YamlFormatter(self.format_line)
        handlers = []
        if self.console:
            handler = logging.StreamHandler(sys.stdout)
            handler.setFormatter(formatter)
            handlers.append(handler)
        for filepath in self.files:
            handler = logging.FileHandler(filepath)
            handler.setFormatter(formatter)
            handlers.append(handler)
        return LoggerConfig(
            level=self.level.to_int_value(),
            disabled=not handlers,
            handlers=handlers,
        )


class ServerConfig(BaseConfigModel):
    alias: str
    sockets: list[SocketsConfig]
    reflection_enabled: bool = True
    proto_files: list[str] | str
    proto_files_base_dir: str | None = None
    mocks: GrpcMockData | None = None


class Config(BaseConfigModel):
    servers: list[ServerConfig]
    general_logging_config: LoggingConfig = LoggingConfig(
        console=True,
        level=LoggingLevel.INFO,
        format=LoggingFormat.TEXT,
        format_line="%(levelname)s: %(message)s",
    )
    api_logging_config: LoggingConfig = LoggingConfig(
        console=True,
        level=LoggingLevel.INFO,
        format=LoggingFormat.YAML,
```

```
            format_line="%(message)s, (request_message)s %(response_message)s
"
                        "%(method)s %(service)s %(code)s %(error_details)s"
                        "%(metadata)s %(alias)s %(timestamp)s",
        )
```

config/validators.py:

```
import constants as c


def validate_grpc_meta_key(key: str) -> str:
    if not c.RPC_HEADER_KEY_PATTERN.match(key):
        raise ValueError(
            "gRPC metadata key should contain only lowercase latin "
            "symbols, numbers and symbols -_. with length 1-256"
        )
    return key


def validate_grpc_meta_value(value: str) -> str:
    if not c.RPC_HEADER_VALUE_PATTERN.match(value):
        raise ValueError(
            "gRPC metadata value should contain only lowercase latin "
            "symbols, numbers and symbols -_. with length 1-8192"
        )
    return value


def validate_grpc_error_status_code(value: int) -> int:
    if value < 1 or value > 16:
        raise ValueError("gRPC error status code should be from 1 to 16")
    return value


def validate_logging_keys(message_format: str):
    used_keys = set(c.PY_LOGS_FORMAT_PATTERN.findall(message_format))
    invalid_keys = used_keys.difference(c.ALLOWED_LOGGING_KEYS)
    if len(invalid_keys) > 0:
        raise ValueError(
            f"Logging keys are not allowed: '{"', '".join(invalid_keys)}'.
"
            "Allowed logging keys are: "
```

```
            f"'{"', '".join(c.ALLOWED_LOGGING_KEYS)}'"
        )
    return message_format
```

logs/__init__.py:

```
import logging
from logging import Logger, Handler, PlaceHolder

from pydantic import BaseModel, ConfigDict

REQUESTS_MOCK_LOG_PREFIX = "mock_requests"


def get_logger_name(parts: list[str]) -> str:
    return ".".join(parts)


class LoggerConfig(BaseModel):
    model_config = ConfigDict(arbitrary_types_allowed=True)

    level: int | str | None = None
    disabled: bool | None = None
    handlers: list[Handler] | None = None


def configure_logger(
    logger: Logger,
    level: int | str | None = None,
    disabled: bool | None = None,
    handlers: list[Handler] | None = None,
):
    if level is not None:
        logger.setLevel(level)
    if disabled is not None:
        logger.disabled = disabled
    if handlers is not None:
        logger.handlers.clear()
        logger.handlers.extend(handlers)
    logger.propagate = False


def configure_logger_by_name(
```

```
    logger_name: str,
    level: int | str | None = None,
    disabled: bool | None = None,
    handlers: list[Handler] | None = None,
):
    logger = logging.getLogger(logger_name)
    configure_logger(logger, level, disabled, handlers)


def configure_all(
    level: int | str | None = None,
    disabled: bool | None = None,
    handlers: list[Handler] | None = None
):
    for logger in logging.root.manager.loggerDict.values():
        if not isinstance(logger, PlaceHolder):
            configure_logger(logger, level, disabled, handlers)


def configure_by_prefix(
    prefix: str,
    level: int | str | None = None,
    disabled: bool | None = None,
    handlers: list[Handler] | None = None,
):
    for logger_name, logger in logging.root.manager.loggerDict.items():
        if logger_name.startswith(prefix) and not isinstance(
            logger, PlaceHolder
        ):
            configure_logger(logger, level, disabled, handlers)
```

logs/formatters.py:

```
from datetime import datetime
import logging
from typing import Set, Iterable

import yaml

from constants import PY_LOGS_FORMAT_PATTERN


class YamlFormatter(logging.Formatter):
```

```
        def __init__(self, used_keys: Iterable | str = None, *args, **kwargs):
            logging.Formatter.__init__(self, *args, **kwargs)

            if isinstance(used_keys, str):
                used_keys = PY_LOGS_FORMAT_PATTERN.findall(used_keys)
            elif used_keys is None:
                used_keys = {"msg"}
            self.used_keys = used_keys

        @property
        def used_keys(self) -> Set[str]:
            return self.__used_keys

        @used_keys.setter
        def used_keys(self, keys: Set[str]):
            self.__used_keys = keys

        def format_fields(self, record: logging.LogRecord) -> dict:
            values = dict(record.__dict__)
            values.pop("args")
            if record.args:
                values["message"] = record.msg % record.args
                if "color_message" in values.keys():
                    values["color_message"] = values["color_message"] % rec-
ord.args
            else:
                values["message"] = record.msg

            return values

        def add_fields(self, record: dict) -> dict:
            record["timestamp"] = datetime.fromtimestamp(
                record.get("created")
            ).isoformat()
            return record

        def format(self, record: logging.LogRecord) -> str:
            record_dict = self.format_fields(record)
            record_dict = self.add_fields(record_dict)
            for key in set(record_dict.keys()):
                if key not in self.used_keys:
```

```
                record_dict.pop(key)
        result = yaml.safe_dump([record_dict])
        return result
```

protobuf/__init__.py:

```
from glob import glob, has_magic
import os

from config.model import ServerConfig
from protobuf.definitions import ProtoFilesPaths
from utils import get_relative_abs_path


def get_proto_files_paths(
    server_config: ServerConfig, config_file_dir: str,
) -> ProtoFilesPaths:
    if isinstance(server_config.proto_files, str):
        file_paths = [server_config.proto_files]
    else:
        file_paths = server_config.proto_files

    base_dir_path = server_config.proto_files_base_dir
    abs_file_paths = set()
    result_file_paths = set()

    for file_path in file_paths:
        abs_file_paths.add(get_relative_abs_path(
            config_file_dir, file_path,
        ))

    if not base_dir_path:
        base_dir_path = os.path.dirname(os.path.commonprefix(list(
            abs_file_paths
        )))
    elif not os.path.isabs(base_dir_path):
        base_dir_path = os.path.normpath(
            os.path.join(os.path.abspath(config_file_dir), base_dir_path)
        )
    for file_path in abs_file_paths:
        if has_magic(file_path):
            paths = glob(file_path, recursive=True)
            for path in paths:
```

```
                result_file_paths.add(path)
        else:
            result_file_paths.add(file_path)

    return ProtoFilesPaths(
        base_dir_abs=base_dir_path,
        proto_files_abs=result_file_paths,
    )
```

protobuf/compilers.py:

```
import tempfile

from google.protobuf import descriptor_pb2 as proto
from google.protobuf.descriptor import (
    Descriptor,
    FieldDescriptor,
    EnumDescriptor,
    EnumValueDescriptor,
    ServiceDescriptor, MethodDescriptor,
)
from google.protobuf.descriptor_pool import DescriptorPool
from grpc_tools import protoc

from constants import DESCRIPTOR_TEMP_FILENAME
from protobuf.types import ProtoType
from protobuf.definitions import (
    InMethodMessageData,
    MethodData,
    EnumField,
    MessageField,
    EnumData,
    MessageData,
    ServiceData,
    ProtoFileStructure,
    ProtoFilesPaths,
    PropertyLabel,
)
from utils import read_file_bytes, get_relative_abs_path

G_PRTBF_T_INDEX_TO_TYPE = {
    FieldDescriptor.TYPE_DOUBLE: ProtoType.DOUBLE,
    FieldDescriptor.TYPE_FLOAT: ProtoType.FLOAT,
```

```
            FieldDescriptor.TYPE_INT64: ProtoType.INT64,
            FieldDescriptor.TYPE_UINT64: ProtoType.UINT64,
            FieldDescriptor.TYPE_INT32: ProtoType.INT32,
            FieldDescriptor.TYPE_FIXED64: ProtoType.FIXED64,
            FieldDescriptor.TYPE_FIXED32: ProtoType.FIXED32,
            FieldDescriptor.TYPE_BOOL: ProtoType.BOOL,
            FieldDescriptor.TYPE_STRING: ProtoType.STRING,
            FieldDescriptor.TYPE_GROUP: ProtoType.GROUP,
            FieldDescriptor.TYPE_MESSAGE: ProtoType.MESSAGE,
            FieldDescriptor.TYPE_BYTES: ProtoType.BYTES,
            FieldDescriptor.TYPE_UINT32: ProtoType.UINT32,
            FieldDescriptor.TYPE_ENUM: ProtoType.ENUM,
            FieldDescriptor.TYPE_SFIXED32: ProtoType.SFIXED32,
            FieldDescriptor.TYPE_SFIXED64: ProtoType.SFIXED64,
            FieldDescriptor.TYPE_SINT32: ProtoType.SINT32,
            FieldDescriptor.TYPE_SINT64: ProtoType.SINT64,
        }


        def update_if_map(message_data: MessageData):
            is_map = False
            if message_data.name.endswith("Entry"):
                if len(message_data.fields) == 2:
                    if message_data.fields[0].name in {
                        "key", "value"
                    } and message_data.fields[1].name in {"key", "value"}:
                        is_map = True
            if is_map:
                message_data.is_map = True


        def generate_descriptor_pool(proto_paths: ProtoFilesPaths):
            pool = DescriptorPool()

            with tempfile.TemporaryDirectory() as dir_name:
                descriptor_abs = get_relative_abs_path(dir_name,
DESCRIPTOR_TEMP_FILENAME)
                command_code = protoc.main((
                    "",
                    f"-I{proto_paths.base_dir_abs}",
                    f"--descriptor_set_out={descriptor_abs}",
```

```
            *proto_paths.proto_files_abs,
        ))
        if command_code != 0:
            raise RuntimeError("Proto files compilation failed")

        descriptor_set = proto.FileDescriptorSet()
        data = read_file_bytes(descriptor_abs)
        descriptor_set.ParseFromString(data)
        for file_proto in descriptor_set.file:
            pool.Add(file_proto)
    return pool


class StructureParser:
    def __init__(
        self,
        pool: DescriptorPool,
        proto_paths: ProtoFilesPaths,
    ):
        self._descriptor_pool = pool
        self._proto_paths = proto_paths

    @property
    def descriptor_pool(self) -> DescriptorPool:
        return self._descriptor_pool

    @property
    def proto_paths(self) -> ProtoFilesPaths:
        return self._proto_paths

    def _parse_message(
        self,
        message_data: Descriptor,
        messages_result: dict[str, MessageData],
        enums_result: dict[str, EnumData]
    ):
        if message_data.full_name in messages_result:
            return

        fields = []
        nested_messages = []
```

```
nested_enums = []
parent_message = None

if message_data.containing_type is not None:
    parent_message = message_data.containing_type.full_name

for message_descriptor in message_data.nested_types:
    message_descriptor: Descriptor
    nested_messages.append(message_descriptor.full_name)

for enum_descriptor in message_data.enum_types:
    enum_descriptor: EnumDescriptor
    nested_enums.append(enum_descriptor.full_name)

result_message = MessageData(
    name=message_data.name,
    parent_message=parent_message,
    nested_messages=nested_messages,
    nested_enums=nested_enums,
    full_name=message_data.full_name,
    fields=[],
)
messages_result[message_data.full_name] = result_message

for field_name, field in message_data.fields_by_name.items():
    field: FieldDescriptor
    nested_message = None
    nested_enum = None
    default = None
    is_map = False

    label = PropertyLabel.OPTIONAL
    if field.label == field.LABEL_REQUIRED:
        label = PropertyLabel.REQUIRED
    elif field.label == field.LABEL_REPEATED:
        label = PropertyLabel.REPEATED

    if field.message_type is not None:
        nested_message = field.message_type.full_name
        self._parse_message(
            field.message_type, messages_result, enums_result
```

```
                )

            if field.enum_type is not None:
                nested_enum = field.enum_type.full_name
                self._parse_enum(field.enum_type, enums_result)

            if field.has_default_value:
                default = field.default_value

            if nested_message is not None:
                if nested_message in messages_result:
                    is_map = messages_result[nested_message].is_map

            fields.append(MessageField(
                message_type=nested_message,
                enum_type=nested_enum,
                name=field.name,
                simple_type=G_PRTBF_T_INDEX_TO_TYPE[field.type],
                number=field.number,
                default=default,
                is_map=is_map,
                label=label,
            ))
        result_message.fields = fields
        update_if_map(result_message)

    def _parse_enum(
        self, enum_data: EnumDescriptor, result: dict[str, EnumData]
    ):
        fields = []
        message = None
        if enum_data.containing_type is not None:
            message = enum_data.containing_type.full_name

        for value_name, value_data in enum_data.values_by_name.items():
            value_data: EnumValueDescriptor
            fields.append(EnumField(
                name=value_name,
                number=value_data.number,
            ))
        result[enum_data.full_name] = EnumData(
```

```
                    name=enum_data.name,
                    parent_message=message,
                    full_name=enum_data.full_name,
                    fields=fields,
                )

            def get_structures(self) -> dict[str, ProtoFileStructure]:
                pool = self.descriptor_pool
                result = {}

                for file_relative in self.proto_paths.get_relative_map().values():
                    try:
                        file_descriptor = pool.FindFileByName(file_relative)
                    except KeyError as e:
                        raise KeyError(f"Required component not found: {e}")
                    services_result = {}
                    messages_result = {}
                    enums_result = {}

                    for name, service_data in file_descriptor.ser-
vices_by_name.items():
                        service_data: ServiceDescriptor
                        methods_result = {}

                        for method in service_data.methods:
                            method: MethodDescriptor
                            methods_result[method.name] = MethodData(
                                name=method.name,
                                input_message=InMethodMessageData(
                                    name=method.input_type.full_name,
                                    streaming=method.client_streaming,
                                ),
                                output_message=InMethodMessageData(
                                    name=method.output_type.full_name,
                                    streaming=method.server_streaming,
                                ),
                            )
                        services_result[service_data.full_name] = ServiceData(
                            name=service_data.name,
                            full_name=service_data.full_name,
                            methods=methods_result,
```

```
                    )

                for message_data in file_descriptor.message_types_by_name.val-
ues():
                    self._parse_message(message_data, messages_result,
enums_result)

                for enum_data in file_descriptor.enum_types_by_name.values():
                    self._parse_enum(enum_data, enums_result)

                result[file_descriptor.name] = ProtoFileStructure(
                    package=file_descriptor.package or None,
                    messages=messages_result,
                    services=services_result,
                    enums=enums_result,
                )
            return result
```

protobuf/definitions.py:

```
import os
from enum import Enum

from pydantic import BaseModel

from protobuf.types import ProtoType
from utils import SimpleType


class ProtoFilesPaths(BaseModel):
    proto_files_abs: list[str]
    base_dir_abs: str

    def get_relative_map(self) -> dict[str, str]:
        result = {}
        for file_path in self.proto_files_abs:
            result[file_path] = os.path.relpath(file_path,
self.base_dir_abs)
        return result


class GeneratedData(BaseModel):
    source_files: dict[str, str]
```

```
    proto_file_to_source: dict[str, set[str]]


class InMethodMessageData(BaseModel):
    name: str
    streaming: bool = False


class MethodData(BaseModel):
    name: str
    input_message: InMethodMessageData
    output_message: InMethodMessageData


class EnumField(BaseModel):
    name: str
    number: int


class PropertyLabel(str, Enum):
    OPTIONAL = "optional"
    REPEATED = "repeated"
    REQUIRED = "required"


class MessageField(EnumField):
    message_type: str | None = None
    enum_type: str | None = None
    complex_type: str | None = None
    simple_type: ProtoType | None = None
    default: SimpleType | None = None
    is_map: bool = False
    label: PropertyLabel


class EnumData(BaseModel):
    name: str
    full_name: str
    parent_message: str | None = None
    fields: list[EnumField]
```

```
class MessageData(BaseModel):
    name: str
    full_name: str
    parent_message: str | None = None
    nested_messages: list[str] = []
    nested_enums: list[str] = []
    is_map: bool = False
    fields: list[MessageField]


class ServiceData(BaseModel):
    name: str
    full_name: str
    methods: dict[str, MethodData]


class ProtoFileStructure(BaseModel):
    package: str | None = None
    messages: dict[str, MessageData]
    services: dict[str, ServiceData]
    enums: dict[str, EnumData]
```

protobuf/types.py:

```
from ctypes import c_double, c_float, c_int64, c_uint64, c_int32, c_uint32
from enum import Enum
from typing import Callable, Any

from pydantic import BaseModel


class ProtoType(str, Enum):
    DOUBLE = "double"
    FLOAT = "float"
    INT64 = "int64"
    UINT64 = "uint32"
    INT32 = "int32"
    FIXED64 = "fixed64"
    FIXED32 = "fixed32"
    BOOL = "bool"
    STRING = "string"
    GROUP = "group"
```

```
    MESSAGE = "message"
    BYTES = "bytes"
    UINT32 = "uint32"
    ENUM = "enum"
    SFIXED32 = "sfixed32"
    SFIXED64 = "sfixed64"
    SINT32 = "sint32"
    SINT64 = "sint64"

    @classmethod
    def contains_value(cls, value):
        return value in cls._value2member_map_


class TypeData(BaseModel):
    python_type: type
    converter: Callable
    default_value: Any


GRPC_PYTHON_TYPES = {
    ProtoType.DOUBLE: TypeData(
        python_type=int,
        converter=lambda v: c_double(int(v)).value,
        default_value=0
    ),
    ProtoType.FLOAT: TypeData(
        python_type=int,
        converter=lambda v: c_float(int(v)).value,
        default_value=0
    ),
    ProtoType.INT64: TypeData(
        python_type=int,
        converter=lambda v: c_int64(int(v)).value,
        default_value=0
    ),
    ProtoType.UINT64: TypeData(
        python_type=int,
        converter=lambda v: c_uint64(int(v)).value,
        default_value=0
    ),
```

```
ProtoType.INT32: TypeData(
    python_type=int,
    converter=lambda v: c_int32(int(v)).value,
    default_value=0
),
ProtoType.FIXED64: TypeData(
    python_type=int,
    converter=lambda v: c_int64(int(v)).value,
    default_value=0
),
ProtoType.FIXED32: TypeData(
    python_type=int,
    converter=lambda v: c_int64(int(v)).value,
    default_value=0
),
ProtoType.BOOL: TypeData(
    python_type=bool,
    converter=lambda v: bool(v),
    default_value=False
),
ProtoType.STRING: TypeData(
    python_type=str,
    converter=lambda v: v if isinstance(v, str) else str(v),
    default_value=""
),
ProtoType.GROUP: None,
ProtoType.MESSAGE: None,
ProtoType.BYTES: TypeData(
    python_type=bytes,
    converter=lambda v: v if isinstance(v, bytes) else bytes(v),
    default_value=0
),
ProtoType.UINT32: TypeData(
    python_type=int,
    converter=lambda v: c_uint32(int(v)).value,
    default_value=0
),
ProtoType.ENUM: None,
ProtoType.SFIXED32: TypeData(
    python_type=int,
    converter=lambda v: c_int32(int(v)).value,
```

```
        default_value=0
    ),
    ProtoType.SFIXED64: TypeData(
        python_type=int,
        converter=lambda v: c_int64(int(v)).value,
        default_value=0
    ),
    ProtoType.SINT32: TypeData(
        python_type=int,
        converter=lambda v: c_int32(int(v)).value,
        default_value=0
    ),
    ProtoType.SINT64: TypeData(
        python_type=int,
        converter=lambda v: c_int64(int(v)).value,
        default_value=0
    ),
}

SimpleProtoType = str | float | int | bool | bytes
```

server/procesors/__init__.py:

```
import logging
from asyncio import sleep
from typing import Callable, AsyncIterator

from google.protobuf.json_format import MessageToDict
from grpc import StatusCode
from grpc._cython.cygrpc import AbortError
from grpc.aio import ServicerContext

from config.model import ServerConfig, ResponseMockConfig
from protobuf.definitions import ServiceData, MethodData
from server.helpers import ProtoObjectResolver
from server.processors.base import ProcessingMeta
from server.processors.logs import APILogProcessor
from server.processors.proxy import ProxyProcessor
from server.processors.templates import TemplateProcessor
from utils import get_exception_error
import server.processors.mock as mocks

logger = logging.getLogger(__name__)
```

```
class ResponseProcessor:
    def __init__(
        self,
        object_resolver: ProtoObjectResolver,
        server_config: ServerConfig,
        template_processor: TemplateProcessor,
        log_processor: APILogProcessor,
        proxy_processor: ProxyProcessor,
    ):
        self._object_resolver = object_resolver
        self._server_config = server_config
        self._log_processor = log_processor
        self._proxy_processor = proxy_processor
        self._template_processor = template_processor

    def generate_method_processor(
        self,
        service_data: ServiceData,
        method_data: MethodData,
    ) -> Callable | None:
        if method_data.output_message.streaming:
            mock_config = ResponseMockConfig(messages=[])
        else:
            mock_config = ResponseMockConfig(messages={})
        service_key = service_data.full_name

        if self._server_config.mocks is not None:
            retrieved = self._server_config.mocks.root.get(
                service_key, {}
            ).get(method_data.name)
            if retrieved is not None:
                mock_config = retrieved

        mock_data_func = self._template_processor.create_mock_data
        message_func = mocks.get_service_message
        metadata_func = mocks.set_trailing_metadata
        error_function = mocks.set_error_data
        log_in_message_func = self._log_processor.log_req_message
        log_out_message_func = self._log_processor.log_res_message
```

```
log_initial_meta_func = self._log_processor.log_req_initial_meta
log_trailers_func = self._log_processor.log_res_trailing_meta
log_error_func = self._log_processor.log_res_error
get_proxy = self._proxy_processor.get_proxy_function

meta = ProcessingMeta(
    object_resolver=self._object_resolver,
    server_config=self._server_config,
    service_data=service_data,
    method_data=method_data,
    mock_config=mock_config,
)

async def process_request(
    input_data: object, context: ServicerContext
) -> tuple[list[dict], list[object]]:
    request_dicts, requests = [], []
    log_initial_meta_func(context, meta)
    if isinstance(input_data, AsyncIterator):
        async for request in input_data:
            request_dict = message_func(
                meta,
                meta.method_data.input_message.name,
                MessageToDict(request),
            )[0]
            log_in_message_func(request_dict, meta)
            requests.append(request)
            request_dicts.append(request_dict)
    else:
        request_dict = message_func(
            meta,
            meta.method_data.input_message.name,
            MessageToDict(input_data),
        )[0]
        log_in_message_func(request_dict, meta)
        requests.append(input_data)
        request_dicts.append(request_dict)

    await mock_data_func(request_dicts, context, meta)
    seconds_delay = meta.mock_data.seconds_delay
    if seconds_delay is not None:
```

```
                    logger.debug(f"'{seconds_delay}' seconds delay for re-
quest")
                    await sleep(seconds_delay)

                return request_dicts, requests

            async def process_unary_response(
                input: object, context: ServicerContext
            ) -> object:
                try:
                    request_dicts, requests = await process_request(input,
context)
                    proxy_func = get_proxy(meta)
                    if proxy_func:
                        response_dict = await proxy_func(requests, context,
meta)
                    else:
                        response_dict = meta.mock_data.messages.root
                        if isinstance(meta.mock_data.messages.root, list):
                            if len(meta.mock_data.messages.root) > 0:
                                response_dict = meta.mock_data.mes-
sages.root[0]
                    metadata_func(context, meta)
                    await error_function(context, meta)
                    response_dict, response = message_func(
                        meta,
                        meta.method_data.output_message.name,
                        response_dict,
                    )
                    log_trailers_func(context, meta)
                    log_out_message_func(response_dict, context, meta)
                    return response
                except AbortError:
                    log_trailers_func(context, meta)
                    log_error_func(context, meta)
                    raise
                except Exception as e:
                    logger.error(get_exception_error(e))
                    try:
                        await context.abort(
                            StatusCode.UNKNOWN,
```

```
                        "Mock API server internal error",
                    )
                finally:
                    log_trailers_func(context, meta)
                    log_error_func(context, meta)

            async def process_stream_response(
                input: object, context: ServicerContext
            ) -> object:
                try:
                    request_dicts, requests = await process_request(input,
context)
                    proxy_func = get_proxy(meta)
                    await error_function(context, meta)
                    if proxy_func:
                        async for response_dict in proxy_func(
                            requests, context, meta
                        ):
                            response_dict, response = message_func(
                                meta,
                                meta.method_data.output_message.name,
                                response_dict,
                            )
                            log_out_message_func(response_dict, context, meta)
                            yield response
                    else:
                        if isinstance(meta.mock_data.messages.root, list):
                            for response_dict in meta.mock_data.messages.root:
                                response_dict, response = message_func(
                                    meta,
                                    meta.method_data.output_message.name,
                                    response_dict,
                                )
                                log_out_message_func(response_dict, context,
meta)
                                yield response
                        else:
                            response_dict, response = message_func(
                                meta,
                                meta.method_data.output_message.name,
                                meta.mock_data.messages.root,
```

```
                            )
                            log_out_message_func(response_dict, context, meta)
                            yield response
                    await error_function(context, meta)
                    metadata_func(context, meta)
                except AbortError:
                    log_trailers_func(context, meta)
                    log_error_func(context, meta)
                    raise
                except Exception as e:
                    code = StatusCode.UNKNOWN
                    message = "Mock API server internal error"
                    logger.error(get_exception_error(e))
                    try:
                        await context.abort(code, message)
                    finally:
                        log_trailers_func(context, meta)
                        log_error_func(context, meta)

                log_trailers_func(context, meta)

            if method_data.output_message.streaming:
                return process_stream_response
            else:
                return process_unary_response

        async def clean_resources(self):
            await self._proxy_processor.close_channels()
```

server/procesors/base.py:

```
from typing import Any

from grpc import StatusCode, ServicerContext
from pydantic import BaseModel, ConfigDict, Field, RootModel

from config.model import (
    ServerConfig, ResponseMockConfig, MetadataKey, MetadataValue, GRPCEr-
rorCode
)
from protobuf.definitions import ServiceData, MethodData
from server.helpers import ProtoObjectResolver
```

```
class MessageMock(RootModel):
    root: dict[str, Any] | list[dict[str, Any]] = {}


class MetadataMock(RootModel):
    root: dict[MetadataKey, MetadataValue] = {}


class ErrorMock(BaseModel):
    code: GRPCErrorCode | str = Field(
        StatusCode.UNKNOWN.value[0],
        description="Error response status code"
    )
    details: str = Field(
        "",
        description="Error response message"
    )


class ProxyMock(BaseModel):
    socket: str = Field(
        None,
        description="gRPC server socket for proxying requests"
    )
    seconds_timeout: float | None = Field(
        None,
        description="gRPC server proxying timeout",
        ge=0,
    )


class ResponseMock(BaseModel):
    messages: MessageMock = Field(
        MessageMock(),
        description="Response message value",
    )
    trailing_meta: MetadataMock = Field(
        MetadataMock(),
        description="Initial response metadata values",
    )
```

```
    error: ErrorMock | None = Field(
        None,
        description="Error data for mocking errors",
    )
    seconds_delay: float | None = Field(
        None,
        description="Seconds delay for request processing",
        gt=0,
    )
    proxy: ProxyMock | None = Field(
        None,
        description="Requests proxying configuration",
    )


class ProcessingMeta(BaseModel):
    model_config = ConfigDict(arbitrary_types_allowed=True)

    object_resolver: ProtoObjectResolver
    server_config: ServerConfig
    service_data: ServiceData
    method_data: MethodData
    mock_config: ResponseMockConfig | str
    mock_data: ResponseMock = ResponseMock()


def extract_invocation_metadata(context: ServicerContext) -> dict:
    metadata_dict = {}
    metadata = context.invocation_metadata()
    if metadata is not None:
        for k, v in metadata:
            if k in metadata_dict:
                if isinstance(metadata_dict[k], list):
                    metadata_dict[k].append(v)
                else:
                    metadata_dict[k] = [metadata_dict[k], v]
            else:
                metadata_dict[k] = v
    return metadata_dict
```

server/procesors/logs.py:

```
import json
```

```
    from logging import Logger, getLogger

    from grpc import StatusCode
    from grpc.aio import ServicerContext

    from logs import (
        get_logger_name, REQUESTS_MOCK_LOG_PREFIX, configure_logger, Log-
gerConfig
    )
    from protobuf.definitions import ServiceData, MethodData
    from server.processors.base import ProcessingMeta, extract_invoca-
tion_metadata


    class APILogProcessor:
        def __init__(self, loggers_conf: LoggerConfig):
            self._loggers_conf = loggers_conf
            self._loggers = {}

        def get_requests_logger(
            self,
            service_data: ServiceData,
            method_data: MethodData,
        ) -> Logger:
            logger_name = get_logger_name([
                REQUESTS_MOCK_LOG_PREFIX, service_data.full_name,
method_data.name
            ])
            logger_obj = self._loggers.get(logger_name)
            if logger_obj is None:
                logger_obj = getLogger(logger_name)
                configure_logger(logger_obj, **self._log-
gers_conf.model_dump())
                self._loggers[logger_name] = logger_obj
            return logger_obj

        def log_req_message(
            self,
            request_dict: dict,
            meta: ProcessingMeta,
        ):
```

```
        api_logger = self.get_requests_logger(
            meta.service_data, meta.method_data
        )

        extra = {
            "service": meta.service_data.full_name,
            "method": meta.method_data.name,
            "request_message": json.dumps(request_dict),
            "alias": meta.server_config.alias,
        }

        api_logger.info("Input message", extra=extra)

    def log_req_initial_meta(
        self,
        context: ServicerContext,
        meta: ProcessingMeta,
    ):
        api_logger = self.get_requests_logger(
            meta.service_data, meta.method_data
        )

        metadata_dict = extract_invocation_metadata(context)
        if metadata_dict:
            extra = {
                "service": meta.service_data.full_name,
                "method": meta.method_data.name,
                "alias": meta.server_config.alias,
                "metadata": metadata_dict
            }
            if meta.server_config.alias is not None:
                extra["alias"] = meta.server_config.alias

            api_logger.info("Invocation metadata", extra=extra)

    def log_res_message(
        self,
        response_dict: dict,
        context: ServicerContext,
        meta: ProcessingMeta,
    ):
```

```
        api_logger = self.get_requests_logger(
            meta.service_data, meta.method_data
        )

        extra = {
            "service": meta.service_data.full_name,
            "method": meta.method_data.name,
            "alias": meta.server_config.alias,
        }

        if response_dict is not None:
            extra["response_message"] = json.dumps(response_dict)
        error_details = context.details()
        if error_details:
            extra["error_details"] = error_details
        code = context.code()
        if code is not None:
            code: StatusCode
            extra["code"] = f"{code.value[0]}: {code.value[1]}"
        metadata_dict = {}
        metadata = context.trailing_metadata()
        if metadata is not None:
            for metadata_item in metadata:
                key = metadata_item[0]
                value = metadata_item[1]
                if key in metadata_dict:
                    if isinstance(metadata_dict[key], list):
                        metadata_dict[key].append(value)
                    else:
                        metadata_dict[key] = [metadata_dict[key], value]
                else:
                    metadata_dict[key] = value
        if metadata_dict:
            extra["metadata"] = metadata_dict

        api_logger.info("Output message", extra=extra)

    def log_res_error(
        self,
        context: ServicerContext,
        meta: ProcessingMeta,
```

```
):
    api_logger = self.get_requests_logger(
        meta.service_data, meta.method_data,
    )

    extra = {
        "service": meta.service_data.full_name,
        "method": meta.method_data.name,
        "alias": meta.server_config.alias,
    }

    error_details = context.details()
    if error_details is not None:
        extra["error_details"] = error_details
    code = context.code()
    if code is not None:
        code: StatusCode
        extra["code"] = f"{code.value[0]}: {code.value[1]}"

    api_logger.info("Output error", extra=extra)

def log_res_trailing_meta(
    self,
    context: ServicerContext,
    meta: ProcessingMeta,
):
    api_logger = self.get_requests_logger(
        meta.service_data, meta.method_data
    )

    metadata_dict = {}
    metadata = context.trailing_metadata()
    if metadata is not None:
        for metadata_item in metadata:
            key = metadata_item[0]
            value = metadata_item[1]
            if key in metadata_dict:
                if isinstance(metadata_dict[key], list):
                    metadata_dict[key].append(value)
                else:
                    metadata_dict[key] = [metadata_dict[key], value]
```

```
                else:
                    metadata_dict[key] = value
        if metadata_dict:
            extra = {
                "service": meta.service_data.full_name,
                "method": meta.method_data.name,
                "alias": meta.server_config.alias,
                "metadata": metadata_dict
            }
            api_logger.info("Trailing metadata", extra=extra)
```

server/procesors/mock.py:

```
from logging import getLogger
from typing import Any, Callable

from grpc.aio import ServicerContext

from server.helpers import get_grpc_status_code
from protobuf.types import ProtoType, GRPC_PYTHON_TYPES, SimpleProtoType
from protobuf.definitions import MessageField, PropertyLabel
from server.processors import ProcessingMeta

logger = getLogger(__name__)


def get_enum_value(
    meta: ProcessingMeta,
    field_data: MessageField,
    enum_name: str,
    value: str | None = None,
) -> tuple[str | None, object | None]:
    if field_data.label == PropertyLabel.OPTIONAL and value is None:
        return None, None

    enum_data = meta.object_resolver.summarized_structure.enums[
        enum_name
    ]
    enum_type = meta.object_resolver.get_enum_type(enum_data)
    if field_data.default is not None:
        value = field_data.default
    if value is not None:
        for enum_property in enum_data.fields:
```

```
                if value == enum_property.name:
                    return enum_property.name, enum_type.Value(enum_prop-
erty.name)
        return enum_type.keys()[0], enum_type.Value(enum_type.keys()[0])


    def get_simple_value(
        field_data: MessageField,
        grpc_type: ProtoType,
        value: Any,
    ) -> tuple[Any, Any]:
        if field_data.label == PropertyLabel.OPTIONAL and value is None:
            return None, None

        type_data = GRPC_PYTHON_TYPES[grpc_type]
        result = type_data.default_value
        if field_data.default is not None:
            result = field_data.default
        if value is not None:
            if type(value) is type_data.python_type or isinstance(
                value, SimpleProtoType
            ):
                try:
                    result = type_data.converter(value)
                    if result != value:
                        logger.debug(
                            f"Field '{field_data.name}' converted to "
                            f"corresponding prototype '{grpc_type.value}'"
                        )
                except Exception:
                    logger.warning(
                        f"Error converting field '{field_data.name}' to "
                        f"type '{type_data.python_type.__name__}'"
                    )
        return result, result


    def _fill_object(
        field_data: MessageField,
        value: Any,
        field_name: str | None = None,
```

```
) -> dict:
    if field_name is None:
        field_name = field_data.name
    if field_data.is_map and value is None:
        return {}
    if field_data.label == field_data.label.OPTIONAL and value is None:
        return {}
    else:
        return {field_name: value}


def _repeat_if_required(
    field_data: MessageField,
    mock_value: Any,
    inner_message_function: Callable,
    *args,
) -> tuple[list, list]:
    if field_data.label == PropertyLabel.REPEATED:
        raw_values = []
        object_values = []
        if mock_value is None:
            return [], []

        if not isinstance(mock_value, list):
            mock_value = [mock_value]
        for mock_data in mock_value:
            raw_value, object_value = inner_message_function(
                *args, mock_data
            )
            raw_values.append(raw_value)
            object_values.append(object_value)
        return raw_values, object_values
    else:
        return inner_message_function(
            *args, mock_value
        )


def get_kv_message_value(
    meta: ProcessingMeta,
    parent_field: MessageField | None,
```

```
    message_name: str,
    mock_value: Any,
) -> tuple[dict | None, dict | None]:
    if mock_value is None:
        return None, None

    message_data = meta.object_resolver.summarized_structure.messages[
        message_name
    ]

    raw_dict = {}
    objects_dict = {}
    if not isinstance(mock_value, dict):
        mock_value = {}

    key_field = message_data.fields[0]
    value_field = message_data.fields[1]

    for property_key, property_value in mock_value.items():
        raw_key, object_key = get_simple_value(
            key_field, key_field.simple_type, property_key,
        )
        if value_field.simple_type == ProtoType.MESSAGE:
            raw_value, object_value = _repeat_if_required(
                value_field,
                property_value,
                get_message_value,
                meta,
                value_field,
                value_field.message_type,
            )
        elif value_field.simple_type == ProtoType.ENUM:
            raw_value, object_value = _repeat_if_required(
                value_field,
                property_value,
                get_enum_value,
                meta,
                value_field,
                value_field.enum_type,
            )
        elif value_field.simple_type == ProtoType.GROUP:
```

```
                raw_value, object_value = get_message_value(
                    meta,
                    value_field,
                    value_field.message_type,
                    property_value,
                )
            else:
                raw_value, object_value = _repeat_if_required(
                    value_field,
                    property_value,
                    get_simple_value,
                    value_field,
                    value_field.simple_type,
                )
            raw_dict.update(_fill_object(
                key_field, raw_value, raw_key,
            ))
            objects_dict.update(_fill_object(
                key_field, object_value, object_key
            ))

    return raw_dict, objects_dict

def get_message_value(
    meta: ProcessingMeta,
    parent_field: MessageField | None,
    message_name: str,
    mock_value: Any,
) -> tuple[dict | None, object | None]:
    if (
        parent_field and
        parent_field.label == PropertyLabel.OPTIONAL and
        mock_value is None
    ):
        return None, None

    raw_dict = {}
    objects_dict = {}

    message_data = meta.object_resolver.summarized_structure.messages[
        message_name
```

```
]
message_type = meta.object_resolver.get_message_type(message_data)

if not isinstance(mock_value, dict):
    mock_value = None

for field_data in message_data.fields:
    mock_property_value = None
    if mock_value is not None:
        mock_property_value = mock_value.get(field_data.name)

    if field_data.simple_type == ProtoType.MESSAGE:
        if field_data.is_map:
            raw_value, object_value = get_kv_message_value(
                meta,
                field_data,
                field_data.message_type,
                mock_property_value,
            )
        else:
            raw_value, object_value = _repeat_if_required(
                field_data,
                mock_property_value,
                get_message_value,
                meta,
                field_data,
                field_data.message_type,
            )
    elif field_data.simple_type == ProtoType.ENUM:
        raw_value, object_value = _repeat_if_required(
            field_data,
            mock_property_value,
            get_enum_value,
            meta,
            field_data,
            field_data.enum_type,
        )
    elif field_data.simple_type == ProtoType.GROUP:
        raw_value, object_value = get_message_value(
            meta,
            field_data,
```

```
                field_data.message_type,
                mock_property_value,
            )
        else:
            raw_value, object_value = _repeat_if_required(
                field_data,
                mock_property_value,
                get_simple_value,
                field_data,
                field_data.simple_type,
            )
        raw_dict.update(_fill_object(
            field_data, raw_value
        ))
        objects_dict.update(_fill_object(
            field_data, object_value
        ))

    return raw_dict, message_type(**objects_dict)


def get_service_message(
    meta: ProcessingMeta,
    message_name: str,
    mock_value: Any,
) -> tuple[dict | None, object | None]:
    raw_value, value = get_message_value(
        meta, None, message_name, mock_value
    )
    if raw_value is None or value is None:
        raw_value, value = {}, {}
    return raw_value, value


async def set_error_data(
    context: ServicerContext,
    meta: ProcessingMeta,
):
    if meta.mock_data.error is None:
        return
```

```
    await context.abort(
        get_grpc_status_code(meta.mock_data.error.code),
        meta.mock_data.error.details,
    )


def set_trailing_metadata(
    context: ServicerContext,
    meta: ProcessingMeta,
):
    metadata_list = []
    for key, value in meta.mock_data.trailing_meta.root.items():
        if isinstance(value, list):
            for value_item in value:
                metadata_list.append((key, value_item))
        else:
            metadata_list.append((key, value))
    context.set_trailing_metadata(metadata_list)
```

server/procesors/proxy.py:

```
import asyncio
import logging
from typing import Callable

from google.protobuf.json_format import MessageToDict
from grpc import ServicerContext
from grpc.aio import AioRpcError, insecure_channel

from server.processors import ProcessingMeta
from utils import get_exception_error

logger = logging.getLogger(__name__)


class ProxyProcessor:
    def __init__(self):
        self._channels_dict = {}
        self._methods_dict = {}

    def _get_proxy_methods(self, meta: ProcessingMeta) -> callable:
        proxy_config = meta.mock_config.proxy
        service_data = meta.service_data
```

```
method_data = meta.method_data

if proxy_config.socket not in self._channels_dict:
    channel = insecure_channel(proxy_config.socket)
    self._channels_dict[proxy_config.socket] = channel
else:
    channel = self._channels_dict[proxy_config.socket]

if service_data.full_name not in self._methods_dict:
    self._methods_dict[service_data.full_name] = {}
if method_data.name not in self._methods_dict[
    service_data.full_name
]:
    in_type = meta.object_resolver.get_message_type(
        meta.object_resolver.summarized_structure.messages[
            method_data.input_message.name
        ]
    )
    out_type = meta.object_resolver.get_message_type(
        meta.object_resolver.summarized_structure.messages[
            method_data.output_message.name
        ]
    )
    if method_data.input_message.streaming:
        if method_data.output_message.streaming:
            processor = channel.stream_stream
        else:
            processor = channel.stream_unary
    else:
        if method_data.output_message.streaming:
            processor = channel.unary_stream
        else:
            processor = channel.unary_unary

    method = processor(
        f"/{service_data.full_name}/{method_data.name}",
        request_serializer=in_type.SerializeToString,
        response_deserializer=out_type.FromString,
        _registered_method=True
    )
```

```
                    self._methods_dict[
                        service_data.full_name
                    ][method_data.name] = method
                return self._methods_dict[ser-
vice_data.full_name][method_data.name]

            async def _process_unary_proxying(
                self,
                requests: list[object],
                context: ServicerContext,
                meta: ProcessingMeta,
            ) -> dict | None:
                try:
                    method_func = self._get_proxy_methods(meta)

                    metadata_list = []
                    metadata = context.invocation_metadata()
                    if metadata is not None:
                        for k, v in metadata:
                            metadata_list.append((k, v))

                    if meta.method_data.input_message.streaming:
                        async def requests_generator():
                            for item in requests:
                                yield item
                        request_obj = requests_generator
                    else:
                        if len(requests) == 0:
                            logger.error("Proxying request internal error")
                            return None
                        request_obj = requests[0]

                    timeout = None
                    if meta.mock_data.proxy.seconds_timeout is not None:
                        timeout = meta.mock_data.proxy.seconds_timeout

                    response = await method_func(
                        request_obj, metadata=metadata_list, timeout=timeout,
                    )
                    return MessageToDict(response)
                except AioRpcError as e:
```

```
                context.set_trailing_metadata(e.trailing_metadata())
                await context.abort(
                    e.code(),
                    e.details(),
                )
            except Exception as e:
                logger.error(
                    f"Proxying request internal error. {get_exception_er-
ror(e)}"
                )

        async def _process_stream_proxying(
            self,
            requests: list[object],
            context: ServicerContext,
            meta: ProcessingMeta,
        ):
            try:
                method_func = self._get_proxy_methods(meta)

                metadata_list = []
                metadata = context.invocation_metadata()
                if metadata is not None:
                    for k, v in metadata:
                        metadata_list.append((k, v))

                if meta.method_data.input_message.streaming:
                    request_obj = requests
                else:
                    if len(requests) == 0:
                        return
                    request_obj = requests[0]

                timeout = None
                if meta.mock_data.proxy.seconds_timeout is not None:
                    timeout = meta.mock_data.proxy.seconds_timeout

                async for response in method_func(
                    request_obj, metadata=metadata_list, timeout=timeout
                ):
                    yield MessageToDict(response)
```

```
                except AioRpcError as e:
                    context.set_trailing_metadata(e.trailing_metadata())
                    await context.abort(
                        e.code(),
                        e.details(),
                    )
                except Exception as e:
                    logger.error(
                        f"Proxying request internal error. {get_exception_er-
ror(e)}"
                    )

            def get_proxy_function(
                self, meta: ProcessingMeta
            ) -> Callable | None:
                if meta.mock_data.proxy is None:
                    return None

                if meta.method_data.output_message.streaming:
                    return self._process_stream_proxying
                else:
                    return self._process_unary_proxying

            async def close_channels(self):
                await asyncio.gather(
                    *[channel.close() for channel in self._channels_dict.values()]
                )
```

server/procesors/templates.py:

```
from logging import getLogger
from typing import TypeVar, Type, Any

from grpc import StatusCode
from grpc.aio import ServicerContext
from jinja2 import Environment
from pydantic import BaseModel, ValidationError, RootModel
from yaml import YAMLError

import constants as c
from templates import AccessibleVariable
from config.model import ResponseMockConfig, ErrorConfig, ProxyConfig
from server.processors import ProcessingMeta
```

```
import server.processors.base as base
import utils

logger = getLogger(__name__)


ModelType = TypeVar("ModelType", bound=BaseModel)


async def render_simple_type(
    env: Environment,
    simple_type: Type[utils.SimpleType],
    value: utils.SimpleType,
) -> utils.SimpleType:
    rendered = await env.from_string(str(value)).render_async()
    try:
        return simple_type(rendered)
    except Exception:
        logger.error("Error parsing rendered data to required type")
        return value


async def render_list(env: Environment, values: list) -> list:
    result = []
    for item in values:
        if isinstance(item, list):
            result.append(await render_list(env, item))
        elif isinstance(item, dict):
            result.append(await render_dict(env, item))
        elif isinstance(item, str):
            result.append(await env.from_string(item).render_async())
        else:
            result.append(item)
    return result


async def render_dict(env: Environment, values: dict) -> dict:
    result = {}
    for key, value in values.items():
        if isinstance(value, list):
            result[key] = await render_list(env, value)
```

```
        elif isinstance(value, dict):
            result[key] = await render_dict(env, value)
        elif isinstance(value, str):
            result[key] = await env.from_string(value).render_async()
        else:
            result[key] = value
    return result


def create_model(
    entity_type: Type[ModelType], *args, **kwargs
) -> ModelType | None:
    try:
        return entity_type(*args, **kwargs)
    except ValidationError as e:
        logger.error(utils.get_msg_from_parts(
            "Invalid mock data format", utils.get_validation_err_msg(e))
        )
    return None


async def render_model_from_str(
    env: Environment, entity_type: Type[ModelType], value: str
) -> ModelType | None:
    rendered = await env.from_string(value).render_async()
    parsed = None
    try:
        parsed = utils.parse_from_yaml(rendered.encode())
    except YAMLError as e:
        logger.error(utils.get_msg_from_parts(
            "Error parsing YML of mock data", utils.get_yml_err_msg(e))
        )
    if parsed is None:
        return None
    if issubclass(entity_type, RootModel):
        return create_model(entity_type, root=parsed)
    elif isinstance(parsed, dict):
        return create_model(entity_type, **parsed)
    else:
        return create_model(entity_type, parsed)
```

```
    async def render_model(
        env: Environment, entity_type: Type[ModelType], value: dict | list
    ) -> ModelType | None:
        if isinstance(value, list):
            rendered = await render_list(env, value)
        else:
            rendered = await render_dict(env, value)
        try:
            return entity_type.model_validate(rendered)
        except YAMLError as e:
            logger.error(utils.get_msg_from_parts(
                "Error parsing YML of mock data", utils.get_yml_err_msg(e))
            )
        except ValidationError as e:
            logger.error(utils.get_msg_from_parts(
                "Invalid mock data format", utils.get_validation_err_msg(e))
            )
        return None


    class TemplateProcessor:
        def __init__(self, environment: Environment):
            self._env = environment
            self._state = c.TEMP_INITIAL_STATE

        async def render_error_config(self, error_config: ErrorConfig):
            code = StatusCode.UNKNOWN.value[0]
            if error_config.code is not None:
                code = await render_simple_type(self._env, int, error_con-
fig.code)

            details = await render_simple_type(
                self._env, int, error_config.details
            )
            return create_model(ErrorConfig, code=code, details=details)

        async def render_proxy_config(
            self, proxy_config: ProxyConfig
        ) -> base.ProxyMock:
```

```
                socket = await render_simple_type(self._env, str, proxy_con-
fig.socket)

                seconds_timeout = None
                if isinstance(proxy_config.seconds_timeout, float):
                    seconds_timeout = proxy_config.seconds_timeout
                elif isinstance(proxy_config.seconds_timeout, str):
                    seconds_timeout = await render_simple_type(
                        self._env, str, proxy_config.socket
                    )
                return create_model(
                    base.ProxyMock, socket=socket, seconds_timeout=seconds_timeout
                )

            async def render_mock_config(
                self, mock_config: ResponseMockConfig | str
            ) -> base.ResponseMock:
                if isinstance(mock_config, str):
                    return await render_model_from_str(
                        self._env, base.ResponseMock, mock_config
                    ) or base.ResponseMock()

                message = None
                if isinstance(mock_config.messages, str):
                    message = await render_model_from_str(
                        self._env, base.MessageMock, mock_config.messages
                    )
                elif isinstance(mock_config.messages, dict | list):
                    message = await render_model(
                        self._env, base.MessageMock, mock_config.messages
                    )
                if not message:
                    message = base.MessageMock()

                metadata = None
                if isinstance(mock_config.trailing_meta, str):
                    metadata = await render_model_from_str(
                        self._env, base.MessageMock, mock_config.trailing_meta
                    )
                elif isinstance(mock_config.trailing_meta, dict):
                    metadata = await render_model(
```

```
                self._env, base.MetadataMock, mock_config.trailing_meta
            )
        if not metadata:
            metadata = base.MetadataMock()

        error = None
        if isinstance(mock_config.error, ErrorConfig):
            error = await self.render_error_config(mock_config.error)
        elif isinstance(mock_config.error, str):
            error = await render_model_from_str(
                self._env, base.ErrorMock, mock_config.error
            )

        seconds_delay = mock_config.seconds_delay
        if seconds_delay is not None:
            seconds_delay = await render_simple_type(
                self._env, float, mock_config.seconds_delay
            )

        proxy = None
        if isinstance(mock_config.proxy, ProxyConfig):
            proxy = await self.render_proxy_config(mock_config.proxy)
        elif isinstance(mock_config.proxy, str):
            proxy = await render_model_from_str(
                self._env, base.ProxyMock, mock_config.proxy
            )

        return create_model(
            base.ResponseMock,
            messages=message,
            trailing_meta=metadata,
            error=error,
            seconds_delay=seconds_delay,
            proxy=proxy,
        ) or base.ResponseMock()

    def _set_state(self, value: Any):
        self._state = value

    def _get_state(self) -> Any:
        return self._state
```

```
        def _fill_environment(
            self,
            requests: list[dict],
            context: ServicerContext,
            meta: ProcessingMeta,
        ):
            self._env.globals[c.TEMP_SOCKETS_KEY] = AccessibleVariable([
                socket_data.socket for socket_data in meta.server_config.sock-
ets
            ])
            self._env.globals[c.TEMP_ALIAS_KEY] = meta.server_config.alias
            self._env.globals[c.TEMP_SERVICE_KEY] = AccessibleVariable(
                meta.service_data.model_dump()
            )
            self._env.globals[c.TEMP_METHOD_KEY] = AccessibleVariable(
                meta.method_data.model_dump()
            )
            self._env.globals[c.TEMP_METADATA_KEY] = AccessibleVariable(
                base.extract_invocation_metadata(context)
            )
            self._env.globals[c.TEMP_MESSAGES_KEY] = AccessibleVariable(re-
quests)
            if len(requests) > 0:
                self._env.globals[c.TEMP_MESSAGE_KEY] = AccessibleVariable(
                    requests[0]
                )
            else:
                self._env.globals[c.TEMP_MESSAGE_KEY] = None

            self._env.globals[c.TEMP_SET_STATE_KEY] = self._set_state
            self._env.globals[c.TEMP_GET_STATE_KEY] = self._get_state

        async def create_mock_data(
            self,
            requests: list[dict],
            context: ServicerContext,
            meta: ProcessingMeta,
        ):
            self._fill_environment(requests, context, meta)
            response_mock = await self.render_mock_config(meta.mock_config)
```

```
        meta.mock_data = response_mock
```

server/__init__.py:

```
import logging
from asyncio import AbstractEventLoop

from grpc.aio import Server

from logs import LoggerConfig
from config.model import ServerConfig
from protobuf import get_proto_files_paths
from protobuf.compilers import StructureParser, generate_descriptor_pool
from server.configurers import GRPCServerConfigurer
from server.helpers import ProtoObjectResolver
from server.processors import (
    APILogProcessor, ResponseProcessor, TemplateProcessor
)
from server.processors.proxy import ProxyProcessor
from templates import create_base_environment

logger = logging.getLogger(__name__)


def create_server(
    server_config: ServerConfig,
    config_file_dir: str,
    loop: AbstractEventLoop,
    api_loggers_config: LoggerConfig,
) -> tuple[Server, GRPCServerConfigurer]:
    proto_paths = get_proto_files_paths(server_config, config_file_dir)

    pool = generate_descriptor_pool(proto_paths)
    structures = StructureParser(pool, proto_paths).get_structures()

    logger.debug("Proto files parsing successful")

    object_resolver = ProtoObjectResolver(structures, pool)

    configurer = GRPCServerConfigurer(
        object_resolver,
        ResponseProcessor(
            object_resolver,
```

```
            server_config,
            TemplateProcessor(create_base_environment(config_file_dir)),
            APILogProcessor(api_loggers_config),
            ProxyProcessor(),
        ),
        server_config,
    )
    server = configurer.build_server(config_file_dir, loop)

    logger.debug("Servers build successful")

    return server, configurer
```

server/configurers.py:

```
import asyncio
from logging import getLogger

import grpc
from google.protobuf import descriptor_pool
from google.protobuf.descriptor import FieldDescriptor
from google.protobuf.message_factory import MessageFactory
from grpc import RpcMethodHandler
from grpc.aio import Server
from grpc_reflection.v1alpha import reflection

from protobuf.types import ProtoType
from server.helpers import ProtoObjectResolver
from config.model import ServerConfig
from server.processors import ResponseProcessor
from protobuf.definitions import ServiceData, ProtoFileStructure
from utils import read_file, get_relative_abs_path


logger = getLogger(__name__)


PROTO_TYPE_INTERNAL_DATA = {
    ProtoType.DOUBLE: FieldDescriptor.TYPE_DOUBLE,
    ProtoType.FLOAT: FieldDescriptor.TYPE_FLOAT,
    ProtoType.INT64: FieldDescriptor.TYPE_INT64,
    ProtoType.UINT64: FieldDescriptor.TYPE_UINT64,
    ProtoType.INT32: FieldDescriptor.TYPE_INT32,
```

```
            ProtoType.FIXED64: FieldDescriptor.TYPE_FIXED64,
            ProtoType.FIXED32: FieldDescriptor.TYPE_FIXED32,
            ProtoType.BOOL: FieldDescriptor.TYPE_BOOL,
            ProtoType.STRING: FieldDescriptor.TYPE_STRING,
            ProtoType.GROUP: FieldDescriptor.TYPE_GROUP,
            ProtoType.MESSAGE: FieldDescriptor.TYPE_MESSAGE,
            ProtoType.BYTES: FieldDescriptor.TYPE_BYTES,
            ProtoType.UINT32: FieldDescriptor.TYPE_UINT32,
            ProtoType.ENUM: FieldDescriptor.TYPE_ENUM,
            ProtoType.SFIXED32: FieldDescriptor.TYPE_SFIXED32,
            ProtoType.SFIXED64: FieldDescriptor.TYPE_SFIXED64,
            ProtoType.SINT32: FieldDescriptor.TYPE_SINT32,
            ProtoType.SINT64: FieldDescriptor.TYPE_SINT64,
        }


        def check_methods(
            structure: ProtoFileStructure, server_config: ServerConfig,
        ):
            if server_config.mocks is None:
                return

            unknown_services = set()
            unknown_methods = {}
            for service_name, methods in server_config.mocks.root.items():
                if service_name not in structure.services:
                    unknown_services.add(service_name)
                else:
                    for method_name in methods.keys():
                        if method_name not in structure.services[ser-
vice_name].methods:
                            if service_name not in unknown_methods:
                                unknown_methods[service_name] = set()
                            unknown_methods[service_name].add(method_name)

            if len(unknown_services) > 0:
                logger.warning(
                    f"Services were not described in Protobuf file/s for server "
                    f"'{server_config.alias}': '{', '.join(unknown_services)}'"
                )
```

```
    if len(unknown_methods) > 0:
        methods_strings = []
        for service_name, methods in unknown_methods.items():
            methods_strings.append(
                f"'{service_name}' -> '{"', '".join(methods)}'"
            )
        logger.warning(
            f"Methods were not described in Protobuf file/s for server "
            f"'{server_config.alias}':\n{"\n".join(methods_strings)}"
        )


class GRPCServerConfigurer:
    def __init__(
        self,
        object_resolver: ProtoObjectResolver,
        response_processor: ResponseProcessor,
        server_config: ServerConfig,
    ):
        self._obj_resolver = object_resolver
        self._response_processor = response_processor
        self._server_config = server_config
        self._pool = descriptor_pool.Default()
        self._factory = MessageFactory(self._pool)

    @property
    def object_resolver(self) -> ProtoObjectResolver:
        return self._obj_resolver

    @object_resolver.setter
    def object_resolver(self, object_resolver: ProtoObjectResolver):
        self._obj_resolver = object_resolver

    @property
    def server_config(self) -> ServerConfig:
        return self._server_config

    @server_config.setter
    def server_config(self, server_config: ServerConfig):
        self._server_config = server_config
```

```
        @property
        def response_processor(self) -> ResponseProcessor:
            return self._response_processor

        @response_processor.setter
        def response_processor(self, mock_processor: ResponseProcessor):
            self._response_processor = mock_processor

        def _create_rpc_method_handlers(
            self, service_data: ServiceData,
        ) -> dict[str, RpcMethodHandler]:
            rpc_method_handlers = {}
            for method_data in service_data.methods.values():
                method_func = self._response_processor.generate_method_proces-
sor(
                    service_data,
                    method_data,
                )
                if method_func is None:
                    logger.warning(
                        f"Error creating method '{method_data.name}' in ser-
vice "
                        f"'{service_data.full_name}'"
                    )

                in_data = self.object_resolver.summarized_structure.messages[
                    method_data.input_message.name
                ]
                out_data = self.object_resolver.summarized_structure.messages[
                    method_data.output_message.name
                ]
                in_type = self.object_resolver.get_message_type(in_data)
                out_type = self.object_resolver.get_message_type(out_data)
                if method_data.input_message.streaming:
                    if method_data.output_message.streaming:
                        handler_creator = grpc.stream_stream_rpc_method_han-
dler
                    else:
                        handler_creator = grpc.stream_unary_rpc_method_handler
                else:
                    if method_data.output_message.streaming:
```

```
                handler_creator = grpc.unary_stream_rpc_method_handler
            else:
                handler_creator = grpc.unary_unary_rpc_method_handler

        rpc_method_handlers[
            method_data.name
        ] = handler_creator(
            method_func,
            request_deserializer=in_type.FromString,
            response_serializer=out_type.SerializeToString,
        )
    return rpc_method_handlers

def build_server(
    self,
    config_file_dir: str,
    loop: asyncio.AbstractEventLoop | None = None,
) -> Server:
    check_methods(
        self.object_resolver.summarized_structure,
        self.server_config,
    )

    resolver = self._obj_resolver
    server = grpc.aio.server()

    for socket_data in self.server_config.sockets:
        if socket_data.certificates is None:
            server.add_insecure_port(socket_data.socket)
        else:
            cert_config = socket_data.certificates
            cert_data = read_file(get_relative_abs_path(
                config_file_dir, cert_config.certificate,
            ))
            key_data = read_file(get_relative_abs_path(
                config_file_dir, cert_config.certificate,
            ))
            root_cert_data = None
            if cert_config.root_certificate is not None:
                root_cert_data = read_file(get_relative_abs_path(
                    config_file_dir, cert_config.root_certificate,
```

```
                    ))
                credentials = grpc.ssl_server_credentials(
                    [(key_data, cert_data)],
                    root_cert_data,
                    root_cert_data is not None,
                )
                server.add_secure_port(socket_data.socket, credentials)

        for service_data in resolver.summarized_structure.services.val-
ues():
            method_handlers = self._create_rpc_method_handlers(ser-
vice_data)
            services_handler = grpc.method_handlers_generic_handler(
                service_data.full_name, method_handlers)
            server.add_generic_rpc_handlers((services_handler,))
            server.add_registered_method_handlers(
                service_data.full_name, method_handlers
            )

        if self.server_config.reflection_enabled:
            reflection_services = [
                reflection.SERVICE_NAME,
                *resolver.summarized_structure.services.keys(),
            ]
            reflection.enable_server_reflection(
                reflection_services,
                server,
                self._obj_resolver.get_descriptor_pool(),
            )

        if loop is not None:
            server._loop = loop

        return server
```

server/helpers.py:

```
import copy
import logging
from typing import Type

from google.protobuf.descriptor import FieldDescriptor
from google.protobuf.descriptor_pool import DescriptorPool
```

```
from google.protobuf.internal.enum_type_wrapper import EnumTypeWrapper
from google.protobuf.message import Message
from google.protobuf.message_factory import GetMessageClass
from grpc import StatusCode

from config.model import GRPCErrorCode
from protobuf.definitions import (
    ProtoFileStructure,
    MessageData,
    EnumData,
)
from protobuf.types import ProtoType

logger = logging.getLogger(__name__)


def get_grpc_status_code(value: GRPCErrorCode) -> StatusCode:
    for enum_value in StatusCode:
        if enum_value.value[0] == value:
            return enum_value
    return StatusCode.UNKNOWN


PROTO_TYPE_INTERNAL_DATA = {
    ProtoType.DOUBLE: FieldDescriptor.TYPE_DOUBLE,
    ProtoType.FLOAT: FieldDescriptor.TYPE_FLOAT,
    ProtoType.INT64: FieldDescriptor.TYPE_INT64,
    ProtoType.UINT64: FieldDescriptor.TYPE_UINT64,
    ProtoType.INT32: FieldDescriptor.TYPE_INT32,
    ProtoType.FIXED64: FieldDescriptor.TYPE_FIXED64,
    ProtoType.FIXED32: FieldDescriptor.TYPE_FIXED32,
    ProtoType.BOOL: FieldDescriptor.TYPE_BOOL,
    ProtoType.STRING: FieldDescriptor.TYPE_STRING,
    ProtoType.GROUP: FieldDescriptor.TYPE_GROUP,
    ProtoType.MESSAGE: FieldDescriptor.TYPE_MESSAGE,
    ProtoType.BYTES: FieldDescriptor.TYPE_BYTES,
    ProtoType.UINT32: FieldDescriptor.TYPE_UINT32,
    ProtoType.ENUM: FieldDescriptor.TYPE_ENUM,
    ProtoType.SFIXED32: FieldDescriptor.TYPE_SFIXED32,
    ProtoType.SFIXED64: FieldDescriptor.TYPE_SFIXED64,
    ProtoType.SINT32: FieldDescriptor.TYPE_SINT32,
```

```
    ProtoType.SINT64: FieldDescriptor.TYPE_SINT64,
}


class ProtoObjectResolver:
    def __init__(
        self,
        proto_structures: dict[str, ProtoFileStructure],
        descriptor_pool: DescriptorPool,
    ):
        self._structures = proto_structures
        self._summarized_structure = self._summarize_proto_structure()

        self._descriptor_pool = descriptor_pool
        self._message_types = self._create_messages_types()
        self._enum_types = self._create_enum_types()

    def _summarize_proto_structure(self) -> ProtoFileStructure | None:
        result = None
        for structure in self._structures.values():
            if result is None:
                result = copy.deepcopy(structure)
            else:
                result.services.update(structure.services)
                result.messages.update(structure.messages)
                result.enums.update(structure.enums)
        return result

    @property
    def pool(self) -> DescriptorPool:
        return self._descriptor_pool

    @property
    def summarized_structure(self) -> ProtoFileStructure:
        return self._summarized_structure

    @property
    def structures(self) -> dict[str, ProtoFileStructure]:
        return self._structures

    def _create_messages_types(self) -> dict[str, Type[Message]]:
```

```
            result = {}
            for message_data in self.summarized_structure.messages.values():
                key = message_data.full_name
                result[key] = GetMessageClass(
                    self._descriptor_pool.FindMessageTypeByName(key)
                )
            return result

        def _create_enum_types(self) -> dict[str, EnumTypeWrapper]:
            result = {}
            for enum_data in self.summarized_structure.enums.values():
                key = enum_data.full_name
                result[key] = EnumTypeWrapper(
                    self._descriptor_pool.FindEnumTypeByName(key)
                )
            return result

        def get_descriptor_pool(self) -> DescriptorPool:
            return self._descriptor_pool

        def get_enum_type(self, enum_data: EnumData) -> EnumTypeWrapper:
            enum_type = self._enum_types.get(enum_data.full_name)
            if enum_type is None:
                message = (
                    f"Error processing enum type '{enum_data.full_name}': "
                    f"object descriptor not found"
                )

                logger.error(message)
                raise KeyError(message)
            return enum_type

        def get_message_type(self, message_data: MessageData) -> Type[Mes-
sage]:
            descriptor = self._message_types.get(message_data.full_name)
            if descriptor is None:
                message = (
                    f"Error processing message type '{mes-
sage_data.full_name}': "
                    f"object descriptor not found"
                )
```

```
            logger.error(message)
            raise KeyError(message)
        return descriptor
```

args.py:

```
from argparse import Namespace, ArgumentParser
from sys import exit


def get_args() -> Namespace:
    arg_parser = ArgumentParser(
        prog="cap-grpc",
        description="gRPC API mocking tool")
    arg_parser.add_argument(
        "-c",
        default="cap-grpc.yml",
        metavar="config file",
        type=str,
        help="configuration .yml file path")
    arg_parser.add_argument(
        "-e",
        default=None,
        action='store_true',
        help="print configuration file examples and exit"
    )
    parsed = arg_parser.parse_args()

    if parsed.e:
        print(
"""
YAML config file example:

servers:
  - alias: 'Book API'
    sockets:
      - socket: 'localhost:8100'
      - socket: 'my.domain:8201'
        certificates:
          certificate: "certificate.pem"
          key_file: "certificate.key"
          root_certificate: "/certs/root-cert.crt"
```

```
    reflection_enabled: true
    proto_files:
      - "types/*.proto"
      - "*.proto"
    proto_files_base_dir: "./"
    mocks:
      com.book.BookService:
        GetBook:
          messages:
            id: "{{message.id}}"
            name: "Expanded and Revised: Names of the Damned"
            type: "ENCYCLOPEDIA"
            author:
              first_name: "Michael"
              last_name: "Belanger"
          trailing_meta:
            custom_metadata: metadata
        AddBook:
          error:
            code: 16
            details: "Unauthorized. Credentials required"
        GetBooksList:
          proxy:
            socket: "original-book-service-host:8100"
            seconds_timeout: 10
api_logging_config:
  console: false
  files:
    - "logs/program.logs"
  format: "text"
  format_line: "%(levelname)s: %(message)s"
  level: "INFO"
general_logging_config:
  console: true
  files:
    - "logs/api.logs"
  format: "yaml"
  format_line: |
    %(message)s (request_message)s %(response_message)s
    %(method)s %(service)s %(code)s %(error_details)s
    %(metadata)s %(alias)s %(timestamp)s
```

```
  level: "DEBUG"
"""
        )
        exit(0)
    return parsed


args: Namespace = get_args()
```

constants.py:

```
import re

PY_LOGS_FORMAT_PATTERN = re.compile(r"%\(([^)]+)\)s")
ALLOWED_LOGGING_KEYS = {
    "alias", "created", "exc_info", "exc_text", "filename", "code", "mes-
sage",
    "funcName", "levelname", "levelno", "lineno", "metadata", "method",
    "module", "msecs", "msg", "name", "pathname", "error_details", "pro-
cess",
    "processName", "relativeCreated", "service", "stack_info", "taskName",
    "thread", "threadName", "timestamp", "request_message", "response_mes-
sage",
}

RPC_HEADER_KEY_PATTERN = re.compile(r"^[a-z0-9-_.]{1,256}$")
RPC_HEADER_VALUE_PATTERN = re.compile(r"^[a-z0-9-_.]{0,8192}$")

DESCRIPTOR_TEMP_FILENAME = "descriptor.pb"

TEMP_BASE_DIR_KEY = "directory"
TEMP_FILES_CACHE_KEY = "files_cache"

TEMP_RELATIVE_KEY = "relative"
TEMP_INSERT_KEY = "insert"
TEMP_SCRIPT_KEY = "shell"
TEMP_SET_STATE_KEY = "set_state"
TEMP_GET_STATE_KEY = "get_state"
TEMP_INITIAL_STATE = "initial"
TEMP_SOCKETS_KEY = "sockets"
TEMP_ALIAS_KEY = "alias"
TEMP_SERVICE_KEY = "service"
TEMP_METHOD_KEY = "method"
```

```
TEMP_METADATA_KEY = "metadata"
TEMP_MESSAGES_KEY = "messages"
TEMP_MESSAGE_KEY = "message"
```

main.py:

```
from args import args
import asyncio
import logging
import os
import sys
from signal import SIGINT, SIGTERM, signal

from grpc.aio import Server

from logs import configure_all
from server import create_server
from utils import get_exception_error, read_file_bytes, parse_from_yaml
from config import parse_config
from server.configurers import GRPCServerConfigurer

logger = logging.getLogger(__name__)


def set_default_logging_config():
    os.environ["GRPC_VERBOSITY"] = "NONE"

    handler = logging.StreamHandler(sys.stdout)
    handler.setFormatter(logging.Formatter("%(message)s"))
    configure_all(logging.INFO, False, [handler])


async def stop_grpc_server(
    server_data: tuple[Server, GRPCServerConfigurer],
):
    await server_data[0].stop(None)

    server_config = server_data[1].server_config
    alias = ""
    if server_config.alias is not None:
        alias = f"'{server_config.alias}'"
    sockets_str = ", ".join([v.socket for v in server_config.sockets])
    logger.info(f"Stopped {alias} gRPC server on {sockets_str}")
```

```
async def start_grpc_server(
    server_data: tuple[Server, GRPCServerConfigurer]
):
    await server_data[0].start()

    server_config = server_data[1].server_config
    alias = ""
    if server_config.alias is not None:
        alias = f"'{server_config.alias}'"
    sockets_str = ", ".join([v.socket for v in server_config.sockets])
    logger.info(f"Started {alias} gRPC server on {sockets_str}")


async def start_grpc_servers(
    servers: list[tuple[Server, GRPCServerConfigurer]],
):
    await asyncio.gather(
        *[start_grpc_server(server_data) for server_data in servers]
    )
    logger.info("All servers started")


async def shutdown_grpc_servers(
    servers: list[tuple[Server, GRPCServerConfigurer]]
):
    await asyncio.gather(
        *[stop_grpc_server(server_data) for server_data in servers]
    )
    logger.info("All servers stopped")

    close_resources_functions = []
    for server_data in servers:
        close_resources_functions.append(
            server_data[1].response_processor.clean_resources()
        )
    await asyncio.gather(*close_resources_functions)


async def wait_for_servers_termination(
```

```
        servers: list[tuple[Server, GRPCServerConfigurer]],
    ):
        await asyncio.gather(
            *[server_data[0].wait_for_termination() for server_data in serv-
ers]
        )


    async def run_servers(
        servers: list[tuple[Server, GRPCServerConfigurer]]
    ):
        await start_grpc_servers(servers)

        loop = asyncio.get_event_loop()

        def grace_shutdown(*args):
            asyncio.run_coroutine_threadsafe(shutdown_grpc_servers(servers),
loop)

        signal(SIGINT, grace_shutdown)
        signal(SIGTERM, grace_shutdown)

        await wait_for_servers_termination(servers)


    def main():
        try:
            set_default_logging_config()

            config = parse_config(
                parse_from_yaml(read_file_bytes(args.c))
            )
            config_file_dir = os.path.dirname(args.c)

            configure_all(
                **config.general_logging_config.get_loggers_con-
fig().model_dump()
            )

            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)
```

```
        servers_data = []
        for server_config in config.servers:
            server_data = create_server(
                server_config,
                config_file_dir,
                loop,
                config.api_logging_config.get_loggers_config(),
            )
            servers_data.append(server_data)

        loop.run_until_complete(run_servers(servers_data))
    except SystemExit:
        pass
    except KeyboardInterrupt:
        logger.critical("Interrupted")
    except Exception as e:
        logger.critical(get_exception_error(e))


if __name__ == "__main__":
    main()
```

templates.py:

```
import asyncio
import json
import os
from asyncio.subprocess import create_subprocess_exec
from copy import deepcopy
from logging import getLogger
from typing import Callable

from jinja2 import BaseLoader, Environment, pass_context
from jinja2.runtime import Context

import constants as c
from utils import read_file, get_relative_abs_path, get_exception_error

logger = getLogger(__name__)


class AccessibleVariable:
```

```
        def __init__(self, obj: dict | list):
            self._raw_obj = obj
            self._obj = deepcopy(obj)
            self._is_dict = isinstance(self._obj, dict)
            if self._is_dict:
                for k in self._obj.keys():
                    if isinstance(self._obj[k], list | dict):
                        self._obj[k] = AccessibleVariable(self._obj[k])
            elif isinstance(self._obj, list):
                for index in range(len(self._obj)):
                    if isinstance(self._obj[index], list | dict):
                        self._obj[index] = AccessibleVariable(self._obj[in-
dex])
            else:
                self._raw_obj = {}
                self._obj = {}

        def __getattr__(self, key):
            if self._is_dict:
                return self._obj.get(key)
            try:
                key = int(key)
            except KeyError:
                return None
            if len(self._obj) > key:
                return self._obj[key]
            return None

        def __getitem__(self, key):
            if self._is_dict:
                return self._obj.get(str(key))
            if len(self._obj) > key:
                return self._obj[key]
            return None

        def __iter__(self):
            if self._is_dict:
                return iter(self._obj.items())
            else:
                return iter(self._obj)
```

```
    def keys(self):
        if self._is_dict:
            return list(self._obj.keys())
        else:
            return [index for index in range(len(self._obj))]

    def values(self):
        if self._is_dict:
            return list(self._obj.values())
        else:
            return self._obj

    def items(self):
        if self._is_dict:
            return list(self._obj.items())
        else:
            return [(k, self._obj[k]) for k in range(len(self._obj))]

    def __str__(self):
        return json.dumps(self._raw_obj)


class AnyPathFSLoader(BaseLoader):
    def __init__(self, base_dir: str):
        self._base_dir = base_dir

    @property
    def base_dir(self) -> str:
        return self._base_dir

    @base_dir.setter
    def base_dir(self, base_dir: str):
        self._base_dir = base_dir

    def get_source(
        self, environment: Environment, template: str
    ) -> tuple[str, str | None, Callable[[], bool] | None]:
        source = ""
        try:
            if os.path.isabs(template):
                source = read_file(template)
```

```
            else:
                source = read_file(os.path.join(self.base_dir, template))
        except IOError:
            pass
        return source, template, lambda: True


@pass_context
def get_file_content(
    context: Context,
    path: str,
    encoding: str | None = None,
    use_cache: bool = True,
) -> str | None:
    base_dir = context.get(c.TEMP_BASE_DIR_KEY, "")
    files = context.get(c.TEMP_FILES_CACHE_KEY, None)
    if files is None and use_cache:
        logger.error("Error using files cache")
        use_cache = False
    try:
        file_path = get_relative_abs_path(base_dir, path)
        if use_cache:
            if file_path not in files:
                file_data = read_file(file_path, encoding)
                files[file_path] = file_data
            return files[file_path]
        else:
            return read_file(file_path, encoding)
    except IOError:
        return None


@pass_context
def get_relative_path(context: Context, file_name: str) -> str | None:
    base_dir = context.get(c.TEMP_BASE_DIR_KEY, "")
    try:
        return get_relative_abs_path(base_dir, file_name)
    except Exception as e:
        logger.error(get_exception_error(e))
```

```
async def run_shell_script(
    program: str, *args, stdin: str | None = None
) -> AccessibleVariable | None:
    try:
        process = await create_subprocess_exec(
            program,
            *args,
            stdin=asyncio.subprocess.PIPE,
            stdout=asyncio.subprocess.PIPE,
            stderr=asyncio.subprocess.PIPE
        )
        stdin_bytes = None
        if stdin is not None:
            stdin_bytes = stdin.encode()
        stdout, stderr = await process.communicate(input=stdin_bytes)
        return AccessibleVariable({
            "code": process.returncode,
            "stdout": stdout.decode(),
            "stderr": stderr.decode(),
        })
    except Exception as e:
        logger.error(get_exception_error(e))
        return None


def create_base_environment(base_dir: str) -> Environment:
    result = Environment(
        loader=AnyPathFSLoader(base_dir),
        enable_async=True,
    )
    result.globals[c.TEMP_BASE_DIR_KEY] = base_dir
    result.globals[c.TEMP_FILES_CACHE_KEY] = {}
    result.globals[c.TEMP_RELATIVE_KEY] = get_relative_path
    result.globals[c.TEMP_INSERT_KEY] = get_file_content
    result.globals[c.TEMP_SCRIPT_KEY] = run_shell_script

    return result
```

utils.py:

```
import logging
import os
```

```
import yaml
from pydantic import ValidationError
from yaml import YAMLError

logger = logging.getLogger(__name__)


SimpleType = int | str | float | bool


def get_relative_abs_path(base_dir: str, file_path: str):
    if not os.path.isabs(file_path):
        return os.path.normpath(
            os.path.join(os.path.abspath(base_dir), file_path)
        )
    return file_path


def get_msg_from_parts(*parts, default: str | None = None) -> str:
    result = ". ".join(parts)
    if not result:
        if not default:
            result = "Unexpected internal error"
        else:
            result = default
    return result


def get_yml_err_msg(err: YAMLError) -> str:
    return str(err)


def get_validation_err_msg(err: ValidationError) -> str:
    error_details = err.errors()
    parts = []
    for error_detail in error_details:
        result_error = ""
        message = error_detail.get("msg")
        location = None

        location_data = error_detail.get("loc")
```

```
            if location_data:
                location = " -> ".join([str(v) for v in location_data])
            if location and message:
                result_error = f"{location}: {message}"
            elif location:
                result_error = f"Unknown error at {location}"
            elif message:
                result_error = message
            if result_error:
                parts.append(result_error)
        return get_msg_from_parts(*parts, default="Unknown validation error")


    def get_io_err_msg(exception: Exception) -> str:
        parts = []
        for argument in exception.args:
            if argument is not None and not isinstance(argument, int):
                parts.append(str(argument))
        return get_msg_from_parts(*parts)


    def get_unknown_err_msg(exception: Exception) -> str:
        parts = []
        for argument in exception.args:
            if argument is not None:
                parts.append(str(argument))
        return get_msg_from_parts(*parts)


    def get_exception_error(exception: Exception) -> str:
        if isinstance(exception, IOError):
            return get_io_err_msg(exception)
        elif isinstance(exception, YAMLError):
            return get_yml_err_msg(exception)
        else:
            return get_unknown_err_msg(exception)


    def read_file_bytes(filepath: str) -> bytes:
        try:
            with open(filepath, "rb") as file:
```

```
            return file.read()
    except IOError as e:
        error_data = get_exception_error(e)
        message = f"File '{filepath}' reading error"
        if error_data:
            message = f"{message}. {error_data}"
        raise IOError(message)


def read_file(filepath: str, encoding: str | None = None) -> str:
    try:
        with open(filepath, "r", encoding=encoding) as file:
            return file.read()
    except IOError as e:
        error_data = get_exception_error(e)
        message = f"File '{filepath}' reading error"
        log_message = message
        if error_data:
            log_message = f"{message}. {error_data}"

        logger.error(log_message)

        raise IOError(message)


def parse_from_yaml(value: bytes) -> dict:
    try:
        return yaml.safe_load(value)
    except Exception as e:
        raise IOError(
            "YAML parsing error. " + get_exception_error(e)
        )
```