BAB III
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Selama program kerja magang di PT Traveloka Indonesia, penulis bekerja pada tim *backend payment out* di divisi *financial services* sebagai *backend engineer intern*. Divisi *financial services* memiliki berbagai *domain* lain seperti *backend payment in*, *backend Traveloka pay*, *QA*, *frontend*, legal, dan sebagainya. Penulis berada dibawah koordinasi manajer *payment engineer* Anton Wibowo, diatur oleh manajer proyek Aga Sebastian, dan dibimbing oleh *backend engineer payment out* Naufal Pamungkas. Tim *payment out* memiliki fungsi untuk mendesain, mengembangkan dan bertanggung jawab atas sistem teknologi dan operasional yang mengatur uang keluar perusahaan seperti *refund*, transfer ke vendor, dan sebagainya. Tim *payment out* berkolaborasi dengan banyak tim lain seperti *Treassury*, *Corporate IT*, *frontend*, data, dan tim lainnya.

Payment out menjadi salah satu kunci penting dalam berjalannya perusahaan, karena dalam 1 hari banyak sekali transaksi perusahaan dengan supplier dan vendor untuk membayar seperti hotel, bus, dan maskapai dan juga proses *refund* ke pelanggan jika pelanggan membatalkan pemesanan atau kelebihan bayar. Terdapat beberapa skenario rumit seperti yang ada pada proses *refund*, misal karena tujuan rekening pelanggan tidak dapat menerima jumlah uang yang akan di-*refund* sehingga harus membagi jumlah *refund* ke jenis rekening pelanggan yang lain. Dan semua ini harus dikendalikan secara automasi untuk mendukung skalabilitas perusahaan.

Penulis diberikan tugas untuk membuat beberapa fitur penting selama melakukan kerja magang, yaitu *tooling*, *reporting*, *tracking*, sentralisasi penggunaan data *microservices*, dan migrasi sistem transfer yang menjadi salah satu fungsi utama domain *payment out*. Seluruh proyek ini memiliki tujuan untuk mengoptimalisasi kualitas keseluruhan sistem dan keamanan dengan menggunakan Bahasa Java, *Framework* Spring, Komunikasi RPC dan HTTP API, dan Terraform AWS. Selama mengerjakan proyek yang diberikan, penulis juga berkolaborasi dengan engineer lainnya dan memberikan ide dan solusi yang berguna untuk

keberhasilan proyek. Penulis juga diberikan kebebasan dan diberi kepercayaan untuk membuat berbagai dokumentasi pada platform Confluence.

Tim *payment out* berkoordinasi menggunakan metode Scrum dengan bantuan aplikasi Atlassian seperti Jira, Confluence, dan Slack. Scrum adalah kerangka kerja untuk mengelola dan menyelesaikan proyek yang kompleks. Scrum dirancang untuk membantu tim bekerja dengan cepat dan efisien, namun tetap fleksibel dan responsif terhadap perubahan. Setiap *development life cycle* terdapat beberapa pertemuan :

1. *Daily standup* : diskusi singkat tentang apa yang sudah dilakukan kemarin dan apa yang akan dikerjakan hari ini dan juga membahas sesuatu yang sulit didiskusikan melalui pesan teks.
2. *Backlog grooming* : pertemuan untuk *planning*, me-*review*, dan membuat tiket tugas yang akan dikerjakan pada sprint berikutnya
3. *Retrospective session* : pertemuan untuk membahas perkembangan tim dan apa yang sudah berjalan dengan baik, apa yang perlu diperbaiki, dan apa yang harus dilakukan ke depannya.
4. *Sprint review* : pertemuan dimana setiap engineer akan membagikan cerita dan konteks tentang apa yang dikerjakan dalam 1 sprint ini dapat digunakan untuk membagikan permasalah apa yang dihadapi saat mengerjakan proyek dan apa solusinya.
5. *Sprint planning* : pertemuan untuk memulai sprint dalam 2 minggu ke depan dengan task yang sudah ditentukan pada *backlog grooming* sebelumnya dan memberi tahu tim lain tentang fokus tim dalam 2 minggu kedepan.

Project manajer dan manajer *engineer* akan mendapatkan dan membuat *task* untuk para *engineer* tim tersebut. Lalu pada pertemuan *backlog grooming*, dilakukan perencanakan untuk fokus sprint berikutnya. *Engineer*, manajer *engineer*, project manajer, dan tim terkait akan membahas konteks *task* yang akan dikerjakan seperti detail proyek yang dimaksud, urgensi *task*, menentukan PIC dan pengembang *task*, dan juga menentukan *story point* yang menjadi patokan lama pengerjaan untuk *task* tersebut. Lama pengerjaan suatu *task* akan ditentukan secara musyawarah dan kesepakatan tim. Setiap *engineer* akan memberikan

estimasi pengerjaan *task* tersebut dan akan diambil suara terbanyak, hal ini dilakukan agar lama pengerjaan suatu *task* optimal dan sesuai karena diambil dari berbagai pihak. Semua *task* yang sudah dibahas secara detail akan dirangkum pada aplikasi Jira sebagai alat *monitoring* Scrum dan diumumkan ke *stakeholder* melalui aplikasi Slack.

3.2 Tugas dan Uraian dalam Kerja Magang

3.2.1 Tugas Kerja Magang

Program kerja magang di Traveloka dilaksanakan dengan menggunakan pendekatan *experiential learning*, dimana fokus pembelajaran berpusat pada keterlibatan dan kontribusi langsung dalam pengalaman kerja nyata (*real-world problem*) yang akan membantu dalam meningkatkan pemahaman dan keterampilan sesuai latar belakang dan penempatannya. Secara umum, prinsip *experiental learning* di Traveloka meliputi perpaduan antara *on-the-job experience learning*, *class room training*, dan *learning from others/mentor*.

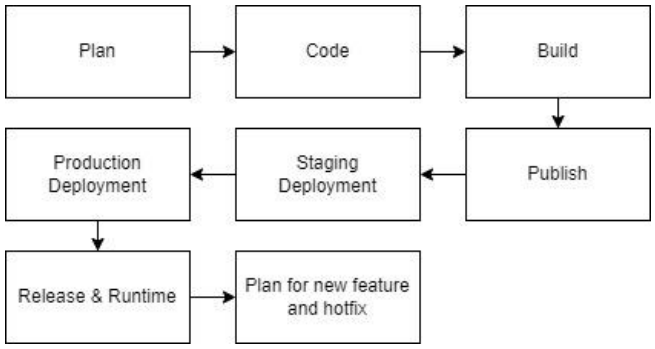
Setiap tugas mencakup kompetensi sebagai berikut :

Tabel 3.1 Tabel Kompetensi Program Kerja Magang Traveloka

Kompetensi	Definisi
Communication & Collaboration	Dapat menyampaikan informasi dan ide-ide secara efektif melalui berbagai kanal komunikasi ke tim terkait.
Accountability	Dapat memprioritaskan pekerjaan berdasarkan tingkat urgensi proyek dan bertanggung jawab atas pekerjaan yang telah diberikan untuk dilakukan dengan benar.
Practical Innovation	Insiatif dan berani untuk mengeksplorasi berbagai hal dan menyampaikan ide untuk mendapatkan solusi yang inovatif.
Drive for result	Menyelesaikan proyek dengan baik pada rentang waktu yang sudah direncanakan.

Programming	Memahami kebutuhan tugas dan menerjemahkan ke dalam program sesuai dengan *business logic* yang diharapkan.
Testing	Membuat *unit test*, menjalankan *local* dan *staging* test, dan membuktikan bahwa program berjalan sesuai dengan baik.
Software Engineering Life Cycle	Dapat menyesuaikan dengan alur kerja dan berkolaborasi secara professional.
Subversioning (Git)	Dapat menggunakan git dan Github untuk berkolaborasi dan berkontribusi pada repositori bersama.
Debugging	Dapat melakukan analisa, menemukan penyebab permasalahan, dan memperbaiki permasalahan secara tepat dan cepat.
Database	Dapat menggunakan RDBMS dan membuat *query database* yang sesuai dan optimal.
Writing Techincal Documentation	Dapat membuat dokumentasi secara tepat, jelas, dan mudah dimengerti oleh tim.

Alur kerja pada proses pengembangan program diilustrasikan sebagai berikut:



Gambar 3.1 Ilustrasi Alur kerja pada proses pengembangan program

Dimana terdapat beberapa faktor penting pada alur kerja ini, awal proyek harus direncanakan secara matang dan didiskusikan dengan tim. Lalu harus membuat dokumen RFC sebagai dasar proses pengembangan proyek, lalu terdapat beberapa tahap testing yaitu *unit test*, *local test*, dan *staging test* dimana meminimalisir ditemukan bug dan ketidaksesuaian pada *business logic* pada program yang dikembangkan.

Sebagai *backend engineer*, penulis bertanggung jawab atas pembuat program servis sesuai *business logic* yang diminta, melakukan perencanaan penyimpanan data pada *database*, *debugging* servis jika ada permasalahan, *testing*, *deployment*, dan mendukung tim lain jika menggunakan domain *payment out*. Penulis mendapatkan pekerjaan yang bervariasi dan mencakup lebih dari 1 domain *payment out*.

Berikut merupakan linimasa penulis dalam melakukan proyek proyek program kerja magang :

Tabel 3.2 Linimasa Tugas Kerja Magang

Proyek	Agustus				September				Oktober				November				Desember			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2																				
3																				
4																				
5																				
6																				
7																				

Dari linimasa tugas kerja magang penulis, terdapat penjabaran tentang proyek yang dikerjakan penulis, yaitu :

1. Onboarding

Pada tahap ini, penulis diberi kesempatan untuk mempelajari teknologi dan budaya kerja yang digunakan di Traveloka. Penulis juga mempersiapkan alat, *software*, dan akun untuk mempermudah pengerjaan proyek. Penulis mempelajari teknologi yang digunakan melalui secara insiatif membuat proyek kecil untuk membuat API server dengan Java Spring Boot dan mengerjakan *backend bootcamp*.

2. Tooling

Proyek ini adalah membuat suatu fitur baru pada *microservice* agar dapat mengubah data yang penting dengan aman dan tervalidasi, mengurangi *human error*, menghindari kesalahan pergantian data, dan mengurangi akses *database* secara langsung. Sebelumnya, engineer melakukan pergantian data dengan *database query* lansung.

3. Reporting

Proyek ini adalah pengembangan pada fungsi *reporting payment out*. Setiap transaksi pada *payment out* harus dilakukan *reporting* kepada tim yang berkaitan. *Reporting* diharuskan mengirimkan data yang valid, format yang benar, dan sesuai dengan permintaan tim lain untuk kebutuhan *business needs* lebih lanjut. Pada proyek ini penulis menambahkan suatu data yang penting dalam proses *reporting* SNS dan menyimpan data tersebut ke dalam *database*.

4. Migrasi API transfer

Proyek ini adalah melakukan migrasi suatu bank transfer API ke versi yang lebih baru. Dimana suatu perusahaan harus mengikuti legalitas hukum pemerintahan, oleh karena itu Traveloka mengikuti migrasi pengubahan API Transfer menggunakan SNAP. SNAP adalah Strandar API Pembayaran Nasional, dimana seluruh transaksi perbankan yang menggunakan komunikasi API dapat digeneralisir dan miliki standar tertentu. Tak hanya melakukan migrasi namun penulis juga membuat suatu *microservice* baru untuk tempat seluruh integrasi pihak eksternal berada.

5. Tracking

Proyek ini memiliki tujuan untuk mengirimkan data ke *database* di BigQuery agar transaksi pada sistem *payment out* dapat ditelusuri dan dianalisis lebih lanjut oleh tim data. Adanya pembaharuan pada sistem *tracking* membuat diharuskan untuk melakukan migrasi dari sistem *tracking* versi lama.

6. Sentralisasi penggunaan data *microservice*

Proyek ini membuat suatu jalur komunikasi antar microservice agar dapat membagi data sehingga dapat mensentralisasi penggunaan data. Hal ini dilakukan karena sebelumnya 2 *microservice* menggunakan *database* yang berbeda namun merupakan data yang sama. Hal ini menimbulkan permasalahan ketika ada data yang perlu diganti sehingga perlu mengganti di 2 *database*.

7. Job handover dan offboarding

Tahap ini penulis melaporkan kepada mentor tentang proyek proyek yang sudah dikembangkan. Penulis berhasil menyelesaikan semua proyek dengan baik sesuai waktu yang direncanakan. Penulis mengembalikan seluruh aset perusahaan yang digunakan untuk menyelesaikan proyek program kerja magang.

Adapun detail tugas yang dilakukan oleh penulis sebagai berikut :

Tabel 3.3 Tugas yang dilakukan oleh penulis

Minggu	Tugas yang Dilakukan
1 - 2	1. *Onboarding* dan pengenalan dengan tim dan budaya kerja bersama mentor dan manajer. 2. Menyiapkan alat, *software*, dan akun untuk mengerjakan proyek. 3. Mempelajari *code convention* dan dokumentasi kode yang digunakan tim. 4. Mengerjakan *Backend bootcamp*. 5. Mempelajari teknologi yang akan digunakan seperti Java Spring Boot, JOOQ, Lombok, RPC, *CRUD Building Block*, dan lain lain dengan membuat proyek kecil API.

	6. Presentasi kepada mentor dan manajer tentang yang sudah dipelajari.
3 - 4	1. Menghadiri pertemuan harian dan mingguan tim. 2. Perencanaan dan pengembangan proyek *tooling*, seperti membuat servis baru, membuat komponen dan servlet RPC, membuat schema CRUD *database*, mengimplementasikan pada kedua *microservice*, dan lain lain. 3. Membuat dokumen RFC. 4. Melakukan *pull request*, Publishing JAR, local dan staging test, *fixing* dan *debugging* proyek. 5. Mempelajari dan menggunakan Terraform, AWS Codepipeline, EC2, dan Cloudwatch. 6. Pertemuan dengan mentor dan manajer.
5 - 6	1. Menghadiri pertemuan harian dan mingguan tim. 2. Perencanaan dan pengembangan proyek *reporting* seperti membuat servis baru, mengubah sistem flow, membaut komponen dan servlet RPC dan Feign HTTP Client, memodifikasi kelas accessor *database*, dan lain lain. 3. Melakukan modifikasi pada infrastruktur AWS dengan Terraform. 4. Melakukan *pull request*, *local* dan *staging test*, *fixing* dan *debugging* proyek.. 5. Melakukan koordinasi dengan tim lain. 6. Membuat dokumentasi atas proyek yang sudah dibuat. 7. Pertemuan dengan mentor dan manajer.
7	1. Menghadiri pertemuan harian dan mingguan tim. 2. Melakukan perencanaan, penelitian, pembuatan dokumentasi, dan diskusi dengan tim untuk menyampaikan hasil penelitian. 3. Pertemuan dengan mentor. 4. Mempelajari alur sistem *payment out.*

8 – 9	1. Menghadiri pertemuan harian dan mingguan tim. 2. Perencanaan dan pengembangan proyek migrasi API transfer. 3. Melakukan perencanaan infrastruktur dan pembuatan kontrak API. 4. Melakukan komunikasi dengan pihak bank. 5. Melakukan diskusi dengan tim tentang proyek yang sedang dihadapi. 6. Intern's day.
10 – 11	1. Menghadiri pertemuan harian dan mingguan tim. 2. Pengembangan proyek migrasi API transfer seperti membuat *endpoint* dengan OpenAPI, HTTP Client dengan feign, transfer handler, *error handling*, dan lain lain. 3. Melakukan testing pada transfer API. 4. Diskusi dengan tim tentang proyek yang dihadapi.
12 – 13	1. Menghadiri pertemuan harian dan mingguan tim. 2. Melakukan perencanaan dan pengembangan proyek migrasi API transfer dan keamanan *microservice*. 3. Membantu masalah administrasi proyek migrasi API transfer seperti menentukan IP static public untuk *whitelisting* dan *generate* public dan private key menggunakan OpenSSL. 4. Melakukan *database query* dan memodifikasi infrastruktur menggunakan Terraform. 5. Melakukan *pull request*, *Publishing* JAR, *local* dan *staging test*, *fixing* dan *debugging* proyek. 6. Diskusi dengan tim untuk proyek yang dihadapi.
14	1. Menghadiri pertemuan harian dan mingguan tim. 2. Melakukan perencanaan dan pengembangan proyek migrasi API transfer yang terdapat fitur baru. 3. Melakukan perencanaan dan pengembangan proyek *tracking*. 4. Membuat konfigurasi dan konvensi baru.

	5. Melakukan *local* dan *staging test*, *fixing* dan *debugging* proyek. 6. Membuat dokumentasi RFC untuk proyek yang dibuat.
15	1. Menghadiri pertemuan harian dan mingguan tim. 2. Melakukan perencanaan dan pengembangan proyek *tracking*. 3. Melakukan *pull request*, *Publishing* JAR, *local* dan *staging test*, *fixing* dan *debugging* proyek *tracking*. 4. Berkoordinasi dengan tim Data. 5. Melakukan validasi data *tracking* dengan BigQuery.
16 – 17	1. Menghadiri pertemuan harian dan mingguan tim. 2. UAT proyek migrasi API transfer dengan bank. 3. Melakukan perencanaan dan pengembangan proyek sentralisasi penggunaan data microservices. 4. Pertemuan dengan mentor dan manajer. 5. Melakukan *pull request*, *Publishing* JAR, *local* dan *staging test*, *fixing* dan *debugging* proyek sentralisasi penggunaan data *microservices*. 6. *Winning presentation training*.
18 – 19	1. Menghadiri pertemuan harian dan mingguan tim. 2. *End to end testing* proyek migrasi API transfer dan pengembangan untuk penambahan fitur dan optimalisasi. 3. Presentasi akhir program kerja magang di perusahaan. 4. Melakukan *Hotfix* pada proyek *tracking*.
20	1. Menghadiri pertemuan harian dan mingguan tim. 2. *Offboarding* dan *handover job*. 3. Membuat dokumentasi teknis dan RFC.

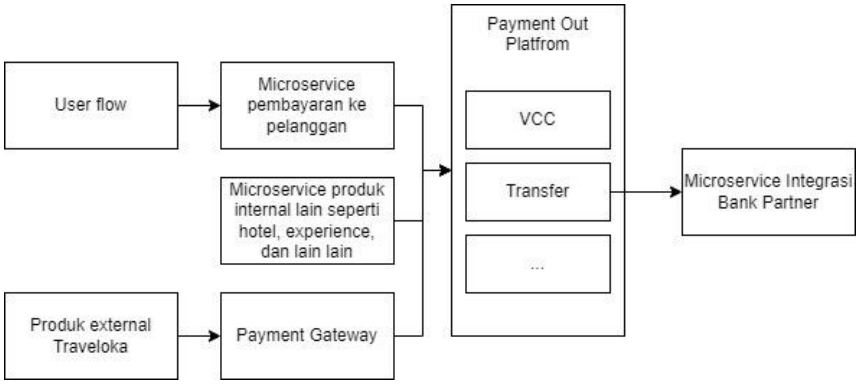
3.2.2 Uraian Pelaksanaan Kerja Magang

Uraian dan gambaran proyek yang dijelaskan dibawah hanya merupakan ilustrasi yang dibuat penulis agar pembaca mendapatkan gambaran hasil proyek

yang dilakukan oleh penulis. Hal ini guna untuk menjaga kerahasiaan data internal perusahaan dan *Intellectual Property* dari PT Traveloka Indonesia.

Sistem uang keluar dapat digunakan oleh konsumen untuk *refund* pemesanan yang dibatalkan atau diganti melalui aplikasi dan web, tim AP/AR dan *Treassury* untuk mentransfer uang ke vendor melalui *web dashboard* internal perusahaan, *microservice* lain melalui pemanggilan API menggunakan RPC atau HTTP API. Selain melakukan transaksi uang keluar, *payment out* juga bertugas untuk melaporkan setiap transaksi ke Tim *Corptech IT* dan Data.

Sistem *payment out* memiliki dasar *microservice* yang terpisah dan independen. Beberapa *microservices* di payment out saling berhubungan menggunakan protokol RPC untuk komunikasinya. Untuk lebih jelasnya, dapat melihat ilustrasi pada gambar 3.2.



Gambar 3.2 Struktur Sistem Payment Out Traveloka

Pada saat *onboarding* penulis mencari tahu mengapa sistem *payment out* didesain dengan beberapa *microservices* melalui membaca dokumen RFC atau dokumen Single Source of Truth (SSOT) yang terdapat pada portal Confluence terkait sistem payment out dan dokumen lainnya. Dan juga penulis menanyakan beberapa pertanyaan kepada mentor dan manajer terkait fungsi dan tujuan desain arsitektur microservice tersebut.

Setiap sistem *payment out* memiliki inti tugas dan tujuannya masing masing untuk menangani satu business core yang berkaitan, oleh karena itu setiap *microservice* memiliki beberapa servis yang berbeda namun berkaitan. Pembuatan beberapa *microservices* tidak hanya untuk skalabilitas dan *clean architecture* agar

memudahkan menambah fitur baru namun juga untuk menangkal permasalahan legal. Karena setiap *instance microservice* memiliki kredensial untuk dapat menangani masalah keuangan secara sesuai dengan peraturan legalitas. Dibalik beberapa hal tersebut juga terdapat maksud lain yang perlu dipertimbangkan.

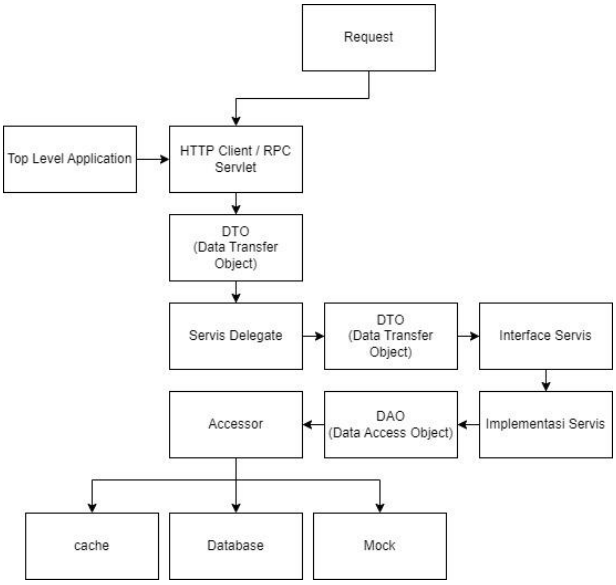
Setiap tahap awal perencanaan pengembangan proyek, penulis juga akan mencari dokumen terkait dengan *microservice* tersebut untuk memahami proyek secara lengkap, walaupun terkadang pada dokumentasi dan anggota tim yang baru bergabung tidak mengetahui beberapa alasan pembuatan komponen *microservice* tersebut. Memahami fungsi dan tujuan *microservices* sangat penting untuk menentukan arsitektur program yang akan dibuat seperti *API Contract*, penggunaan *library*, SDK, dan infrastruktur, dan lain lain.

Sistem automasi keuangan diharapkan memiliki kemampuan untuk skalabilitas, akurasi dan validasi yang tinggi, sesuai dengan *business logic* yang diharapkan, dan tahan terhadap kondisi apa saja. Dengan persyaratan tersebut, penulis diajarkan untuk membuat program sesuai dengan *clean architecture*, *defensive code*, *readability*, *best practices*, memanfaatkan paradigma *object oriented programming*, dan menggunakan protokol komunikasi antar *microservices* yang sesuai dan aman untuk menciptakan program yang reliabel dan performa tinggi.

Berikut merupakan beberapa ilmu yang didapatkan oleh penulis :

1. *Clean architecture*

Clean Architecture adalah metode untuk mendesain program dengan memisahkan komponen menjadi lapisan-lapisan independen yang saling berhubungan. Pemisahan komponen membuat program lebih mudah untuk mempertahankan sistem yang sudah ada dan menambahkan fitur baru, karena memungkinkan untuk mengembangkan di lapisan dalam sistem tanpa perlu mengkhawatirkan lapisan luar sistem. Terdapat DTO dan DAO sebagai objek yang memisahkan kedua servis, servis *delegate* yang berfungsi untuk mengirimkan ke servis lain. Untuk ilustrasi struktur *clean architecture* yang penulis lakukan dapat dilihat pada gambar 3.3.



Gambar 3.3 Struktur Clean Architecture

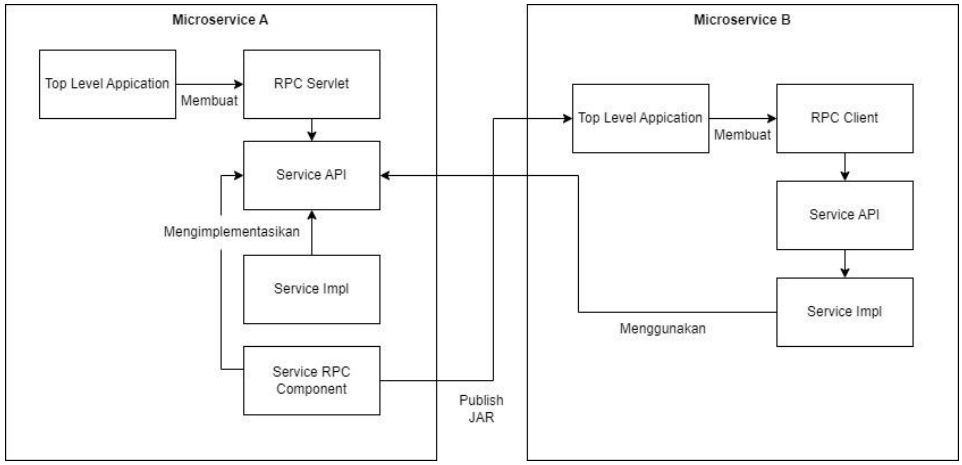
Penjelasan dari desain arsitektur diatas adalah pada dasarnya dibagi menjadi 4 bagian yaitu driver, interface, application input dan output, dan inti servis untuk menjalankan business process.

Penjelasan bagiannnya adalah :

1. Driver : Semua input dan output microservice akan dihandle oleh driver yang di-construct di top level application component dan accessor seperti HTTP Client, RPC servlet, dan koneksi database (JOOQ).
2. Interface : Yaitu servis delegate sebagai routing ke service tujuan yang memikiki business process yang dimaksud.
3. Application input dan output : antara driver, interface, dan service akan memiliki object model yang dinamik untuk menghubungkan beberapa service yang berbeda beda fungsinya. Model yang dinamik dapat dicapai dengan membuat variabel memiliki kapabilitas untuk nullable atau kosong.
4. Service business process : adalah servis memiliki instruksi dari business process yang diharapkan.

2. Komunikasi RPC

JSON RPC adalah protokol yang ringan untuk mengirim *request* yang menggunakan anotasi JSON. RPC adalah protokol sederhana yang memungkinkan klien mengirim permintaan ke server, dan server mengirim respons kembali ke klien. JSON RPC menggunakan *pengkodean* yang sederhana dan ringan, sedangkan HTTP API dapat menggunakan berbagai pengkodean dan memiliki struktur yang lebih fleksibel. Pada komunikasi RPC penulis juga menggunakan metode Interface Description Language (IDL), dimana menggunakan interface untuk mendefinisikan metode, data, parameter, dan lain lain untuk diterjemahkan menjadi anotasi RPC. Untuk lebih jelasnya dapat melihat struktur RPC pada gambar 3.4.



Gambar 3.4 Struktur RPC

3. Pemrograman berorientasi objek

Dalam pembuatan proyek penulis memanfaatkan pilar pilar pemrograman berorientasi object yang memungkinkan kita untuk membuat kode yang fleksibel, modular, dan *reusable*, yaitu :

1. *Inheritance* : Memakai kembali object atau library yang sudah dibuat oleh engineer lain. Penulis juga memanfaatkan overloading dan overriding untuk menghindari kerusakan pada alur program dalam merubah metode.
2. *Encapsulation* : mendeklarasikan objek dengan tipe *final private* variable untuk masalah keamanan dan menghindari akses objek *public.*

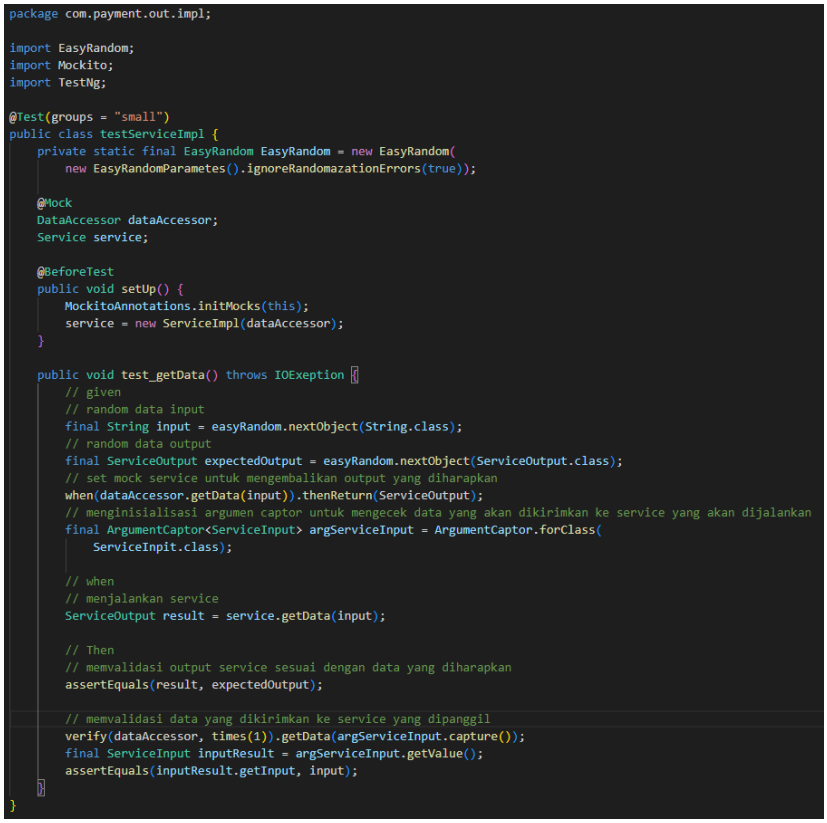
3. *Abstraction* : Membuat servis dengan teknik *code to interface* atau bisa disebut juga sebagai *defensive coding* untuk menutupi implementasi servis utamanya.
4. *Polymorphism* : Memakai interface untuk menjalankan instruksi dan data.

Setiap servis yang dibuat terdiri dari 2 *package* yaitu *package* "API" yang berisi *interface* dan model yang akan digunakan pada servis tersebut dan "IMPL" yang berisi kelas yang mengimplemetasikan *interface* API servis tersebut dan unit test dari implementasi tersebut.

Objek dan kelas dideklarasikan menggunakan *library* Lombok dengan konfigurasi yang berbeda beda. Jika suatu Objek atau kelas tidak memerlukan pemberian *argument* dapat menggunakan konfigurasi *NoArgsConstructor* dengan akses level *private* sehingga objek dideklarasikan secara *private*, dan jika membutuhkan *argument* dapat menggunakan konfigurasi *RequiredArgsConstructor* atau *AllArgsConstructor*. Library Lombok membantu pembuatan *builder* dan *constructor* dalam suatu objek.

4. Unit Test

Pada saat program kerja magang, penulis mendapatkan banyak pengalaman untuk membuat *unit test* yang baik dan benar. Penulis menggunakan *library* Junit, EasyRandom, dan Mockito untuk membuat *unit test*. Pada saat membuat *unit test* harus memikirkan segala skenario yang akan program jalani dan harus pintar untuk membuat data yang sesuai dengan skenario tersebut. Seperti membuat *Mock* data *accessor* yang sesuai SQL query, dan lain lain. Penulis juga menggunakan Mockito *argument captor*, *argument captor* memiliki fungsi untuk mendapatkan parameter yang diberikan pada suatu servis yang sudah di-*mock* sehingga dapat memvalidasi input dan output suatu servis pada *unit test*.



```
package com.payment.out.impl;

import EasyRandom;
import Mockito;
import TestNg;

@Test(groups = "small")
public class testServiceImpl {
    private static final EasyRandom EasyRandom = new EasyRandom(
        new EasyRandomParametes().ignoreRandomazationErrors(true));

    @Mock
    DataAccessor dataAccessor;
    Service service;

    @BeforeTest
    public void setUp() {
        MockitoAnnotations.initMocks(this);
        service = new ServiceImpl(dataAccessor);
    }

    public void test_getData() throws IOExeption {
        // given
        // random data input
        final String input = easyRandom.nextObject(String.class);
        // random data output
        final ServiceOutput expectedOutput = easyRandom.nextObject(ServiceOutput.class);
        // set mock service untuk mengembalikan output yang diharapkan
        when(dataAccessor.getData(input)).thenReturn(ServiceOutput);
        // menginisialisasi argumen captor untuk mengecek data yang akan dikirimkan ke service yang akan dijalankan
        final ArgumentCaptor<ServiceInput> argServiceInput = ArgumentCaptor.forClass(
            ServiceInpit.class);

        // when
        // menjalankan service
        ServiceOutput result = service.getData(input);

        // Then
        // memvalidasi output service sesuai dengan data yang diharapkan
        assertEquals(result, expectedOutput);

        // memvalidasi data yang dikirimkan ke service yang dipanggil
        verify(dataAccessor, times(1)).getData(argServiceInput.capture());
        final ServiceInput inputResult = argServiceInput.getValue();
        assertEquals(inputResult.getInput, input);
    }
}
```

Gambar 3.5 Contoh Pembuatan Unit Test dan Argumen Captor

Setiap *pull request* yang dibuat, akan secara otomatis Sonarlint menganalisis kode yang diganti. Sonarlint akan menentukan seberapa besar *covarage unit test* yang sudah dibuat melalui dari banyaknya cabang program yang sudah dilewati pada setiap alur program (dari input sampai output) termasuk kondisi if dan switch case. Misal pada logic if, terdapat 2 akhir yang berbeda yaitu true atau false. Hal ini akan dianggap sebagai 2 skenario yang berbeda. Untuk 2 skenario tersebut harus dilakukan pada *unit test* yang berbeda.

Selagi menjalankan skenario yang berbeda harus juga divalidasi untuk input dan outputnya yang sesuai dengan business process yang diharapkan. Oleh karena itu harus pintar untuk memvalidasi suatu unit test agar tidak hanya sekedar menjalankan alur agar tidak terdeteksi error atau crash program.

Pada proses *unit test* juga bisa menjadi cara untuk mendeteksi suatu servis *coupled tied* atau terlalu berhubungan dengan servis yang digunakan sehingga tidak *clean architecture* dan akan sulit jika servis tersebut dipakai oleh servis lainnya

yang berbeda desain. Hal ini dapat didapatkan ketika kita tidak bisa mengatur jalan suatu *test* sehingga tidak bisa dilakukan *mock* suatu servis atau komponen.

5. Optimalisasi Looping

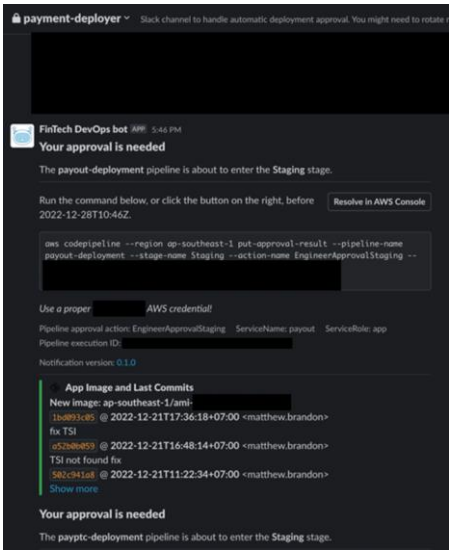
Penulis juga belajar untuk mengoptimalisasi *looping* dengan paradigma "*fail fast, fail early*". Metode ini memiliki tujuan untuk membuat validasi data dengan *if* diawal fungsi untuk menghindari menjalankan instruksi yang tidak perlu dijalankan. Hal ini merupakan hal kecil namun sangat berdampak pada skalabilitas sistem. Contohnya seperti berikut :

```java
package com.channel.integration.bank.service;

import BankAccessTokenService;

@NoArgsConstructor
public class bankAccessTokenImpl implements bankAccessTokenService {

    @Override
    public accessTokenOutput generateAccessToken(accessTokenInput input){
        // validasi jika input tidak sesuai dengan yang diharapkan
        if (accessTokenInput == null) {
            throw new Exception("Data Not Valid");
        }

        // lakukan instruksi utama
    }
}
```

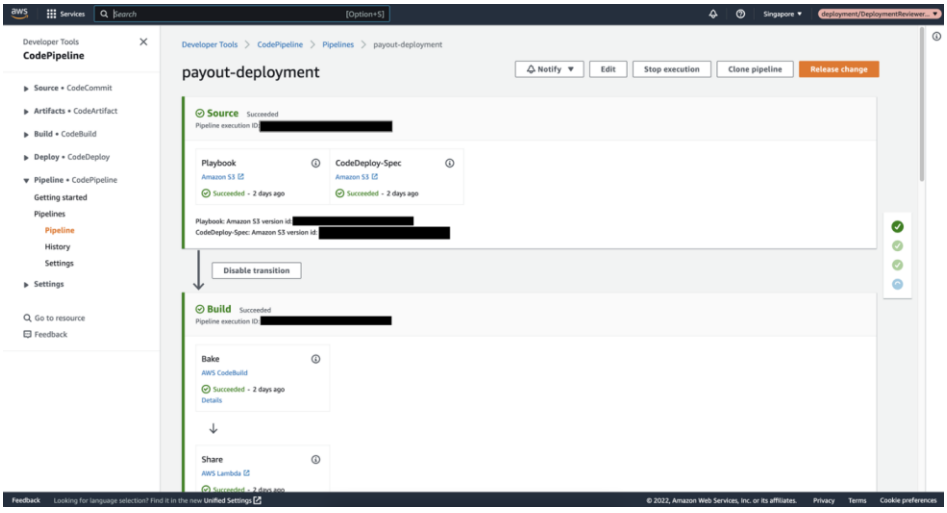
Gambar 3.6 Contoh aplikasi "*fail fast, fail early*"

Selama program kerja magang, penulis sudah membuat sebanyak 38 *pull request* atau kontribusi ke dalam Github repositori perusahaan di berbagai domain *payment*. Penulis memanfaatkan berbagai fitur pada Git seperti *cherry-pick*, *rebase*, *push* dan *pull*, *resolve conflict*, dan lain lain.

Setiap *branch* pada repositori proyek memiliki konvensi nama masing masing. Untuk proses *deployment*, menggunakan konvensi nama branch dengan *prefix* /test/ untuk lingkup *staging* dan /release/ untuk lingkup *production*. Jika dilakukan *push* pada salah satu *branch test* atau *release*, maka CodePipeline akan memulai untuk proses *deployment*, dimulai dari proses *baking* AMI, *share* hasil *image*, pembuatan *instance*, dan *load balance* ke *instance* yang baru secara *automasi*. Semua proses *deployment* dapat dimonitor melalui *dashboard* AWS dan juga aplikasi Slack untuk notifikasi pembaharuan proses *deployment*.



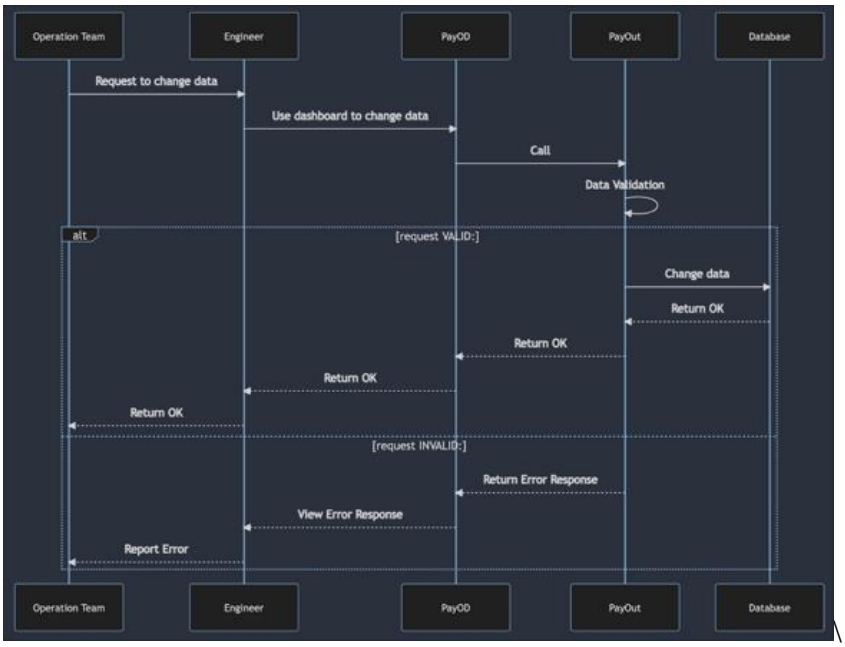
Gambar 3.7 Tampilan Notifikasi Approval Deployment



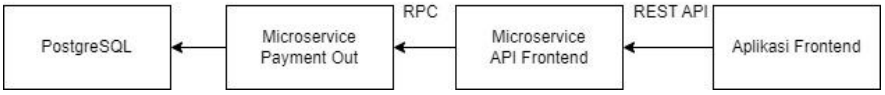
Gambar 3.8 Tampilan CodePipeline saat proses deployment

3.2.2.1 Tooling

Penulis menggunakan SDK fungsi CRUD (*Create*, *Read*, *Update*, *Delete*) yang sudah dibuat oleh engineer lain di Traveloka. Penulis menggunakan SDK ini karena mempermudah proses pengembangan *backend* dan juga tim *frontend* karena API yang dipanggil sudah memiliki *schema field* dan data yang standar dan sesuai pada struktur *frontend*. Berikut merupakan ilustrasi dari alur proyek tooling pada gambar 3.9.



Gambar 3.9 Ilustrasi alur kerja setelah implementasi *tooling*



Gambar 3.10 Arsitektur sistem proyek tooling

1. Pengembangan

Proyek dimulai dengan membuat servis, objek, dan *schema data* baru untuk fungsi CRUD. Saat membuat *shema* untuk *entity* data, penulis juga menambahkan fungsi otorisasi agar fungsi ini hanya dapat diakses oleh *role* tertentu.

Lalu membuat komponen RPC dan RPC servlet kemudian menginisiasi pada *top level application* dengan konfigurasi RPC yang sudah ditentukan seperti *endpoint*. Setelah RPC server sudah siap digunakan, penulis melakukan *publish* JAR untuk digunakan pada *microservice* API *frontend*. Penulis juga melakukan pengembangan pada *microservice* API *frontend* agar mengubah HTTP API *request* menjadi *request* RPC ke *microservice* utama. Servis juga dibuatkan *unit test* untuk mengetes apakah fungsi yang dibuat sudah sesuai dengan yang diharapkan.

Servis akan menggunakan kelas *accessor* pada database postgresSQL. Penulis membuat kelas *accessor* dengan menggunakan RDBMS JOOQ pada database PostgreSQL. Salah satunya penulis membuat *query* untuk mendapatkan data dengan *query select* dan parameter *limit* dan *skip* untuk fitur paginasi.

Setiap data memiliki beberapa kelas *accessor* dengan sumber data yang berbeda, misal untuk data yang dikerjakan oleh penulis memiliki 3 kelas *accessor* yaitu PostgreSQL, *cached* data, dan *Mock*. Ketiganya saling berhubungan untuk kondisi masing masing, untuk *Mock accessor* pada kondisi *testing*, *cached accessor* data jika data sudah pernah dipanggil, dan PostgreSQL *accessor* jika mengambil data langsung pada database. *Mock accessor* adalah *accessor* untuk mendapatkan data sesuai dengan konfigurasi skenario testing untuk keperluan *unit test*. Penulis membuat *Mock accessor* dengan mengkonversikan SQL query dengan data yang di random.

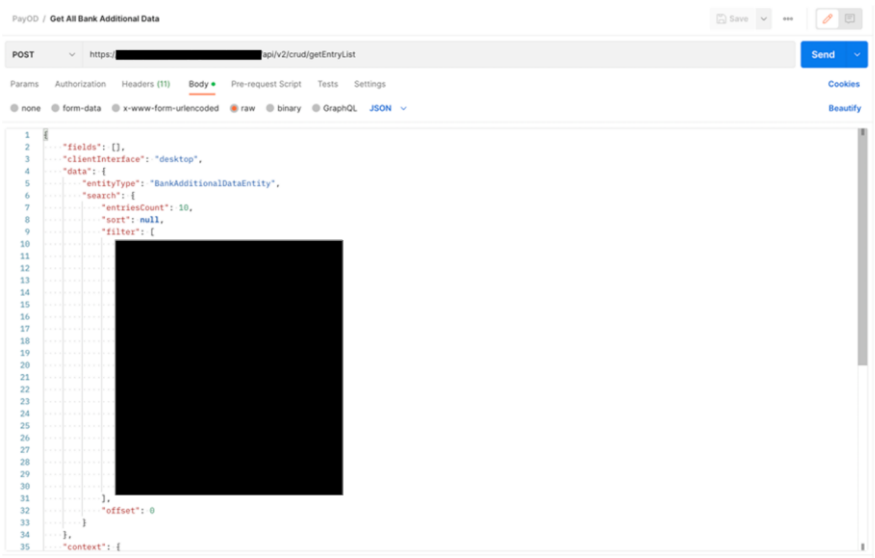
```
@Override
public MockResult[] execute(MockExecuteContext ctx) thr SQLException {
    // menggunakan dialect PostgresSQL
    DSLContext dslContext = DSL.using(SQLDialect.POSTGRES);
    MockResult [] mock =  new MockResult[1];

    String sql = ctx.sql();
    Object[] bindings = ctx.bindings();

    if ((sql.toUpperCase().startsWith("SELECT"))) {
        Result<AdditionalData> result = dslContext.newResult(ADDITIONAL_DATA);
        // melakukan pembuatan data random
        result.add(generateRandomData());
        mock[0] = new MockResult(1, result)l
    }
    return mock;
}
```

Gambar 3.11 Contoh Mock Accessor pada PostgresSQL

Pada akhir proyek, penulis juga membuat dokumentasi step step penggunaan servis ini berserta RPC dan HTTP API payload yang harus dikirimkan.

2. Hasil

Setelah melakukan proses pengembangan program, penulis berhasil membuat fitur tooling yang dapat melihat, mengubah, menambah, dan menghapus data dengan melalui HTTP API. Terdapat fitur validasi yang membuat keamanan data meningkat. Berikut merupakan hasil uji coba yang dilakukan penulis.

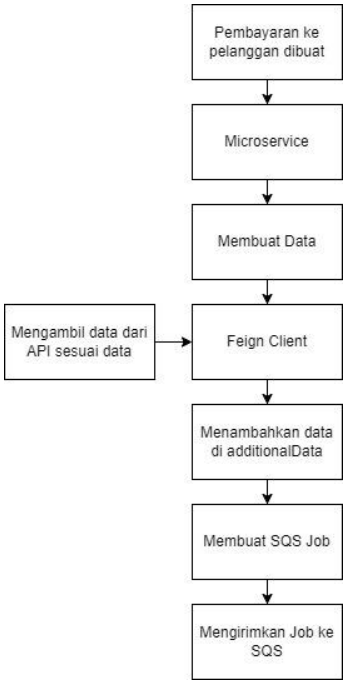


Gambar 3.12 Bentuk input dari HTTP API

Gambar 3.13 Tampilan saat memanggil RPC tooling dari instance bastion

3.2.2.2 Reporting

Pada proyek ini penulis harus mendapatkan data baru yang didapatkan melalui *request* HTTP API terhadap *microservice* tim lain dengan mengirimkan data yang diperlukan seperti tanggal transaksi dan lain lain. *Reporting* ini menggunakan teknologi SQS sehingga data dikirimkan dalam bentuk *message* ke AWS SNS. Berikut merupakan alur proyek reporting pada gambar 3.14.



Gambar 3.14 Alur proyek *reporting*

1. Pengembangan

Pada proses pengembangan tugas penulis adalah :

1. Melakukan perencanaan terhadap pengambilan data yang dibutuhkan untuk mendapatkan data target, tempat untuk menyimpan data target yang sudah didapatkan, dan cara mengambil data target saat melakukan *reporting*.
2. Membuat suatu servis yang menggunakan Feign HTTP Client untuk mendapatkan data target dari *microservice* tim lain.
3. Memodifikasi alur sistem saat pembuatan data utama agar ditambahkan data target
4. Memodifikasi kelas *accessor* penyimpanan data
5. Memodifikasi SQS *reporting* agar mengirimkan data target
6. Mengatur Terraform agar kedua *microservices* dapat saling berkomunikasi

Penulis dengan bantuan mentor memutuskan untuk tidak menambahkan field data pada database, namun menyimpan data pada *field additionalData* yang sudah disiapkan oleh engineer sebelumnya untuk penyimpanan data yang baru. *Field additionalData* berbentuk *HashMap* sehingga dapat diisi beberapa data sekaligus.

Penulis memodifikasi kelas *accessor* pada postgreSQL dengan menambahkan fungsi baru dengan nama yang sama namun berbeda parameter, ini disebut dengan *overloading*. Hal ini karena agar tidak merusak servis lain yang menggunakan fungsi kelas *accessor* yang sama.

Karena untuk mendapatkan data target membutuhkan data tanggal transaksi yang dimana pada alur sistem sebelumnya tidak ditemukan data tanggal transaksi maka penulis melakukan modifikasi pada alur sistem agar dapat mengirimkan data tanggal transaksi dari data itu di buat dengan menambah 1 variabel *hashmap* pada objek DTO servis yang digunakan.

Salah satu pelajaran yang didapatkan oleh penulis adalah sistem *return early*, dimana jika data sudah ditemukan atau data yang dibutuhkan untuk mendapatkan data target tidak ada maka tidak akan melakukan pengambilan data, penyimpanan data, dan reporting. Hal ini cukup remeh namun dapat berdampak besar pada skalabilitas.

Pada proyek kali ini penulis pertama kali menggunakan *unit test* dengan *argument captor*, dimana *argument captor* digunakan untuk mengambil nilai dari suatu variabel yang dikirimkan pada servis yang di *mock*. Dengan ini *unit test* dapat memvalidasi *input* dan *output* servis yang dibuat, misal untuk memvalidasi nilai yang dikirimkan pada *request* API saat permintaan data ke *microservice* lain.

Konfigurasi Terraform diperlukan untuk memperbolehkan kedua *microservice* berkomunikasi. Namun karena kedua *microservice* ini berada di 2 buah VPC yang berbeda, maka digunakan NAT IP sebagai *security rule*. NAT IP adalah IP *static* yang gunakan pada *application load balancer* sehingga semua *instance* yang terhubung pada *load balancer* ini akan menggunakan NAT IP ini.

Terraform adalah Infrastruktur as Code (IaC) dimana memungkinkan untuk menentukan dan mengelola sumber daya infrastruktur, seperti *instance*, *load balancer*, VPC, dan lain lain dalam bahasa konfigurasi tingkat tinggi, yang dikenal sebagai Bahasa Konfigurasi HashiCorp (HCL) dan aman.

2. Hasil

Setelah melakukan proses pengembangan, penulis berhasil mendapatkan hasil yang diinginkan. Dimana data dapat diambil, disimpan, dan dilaporkan.

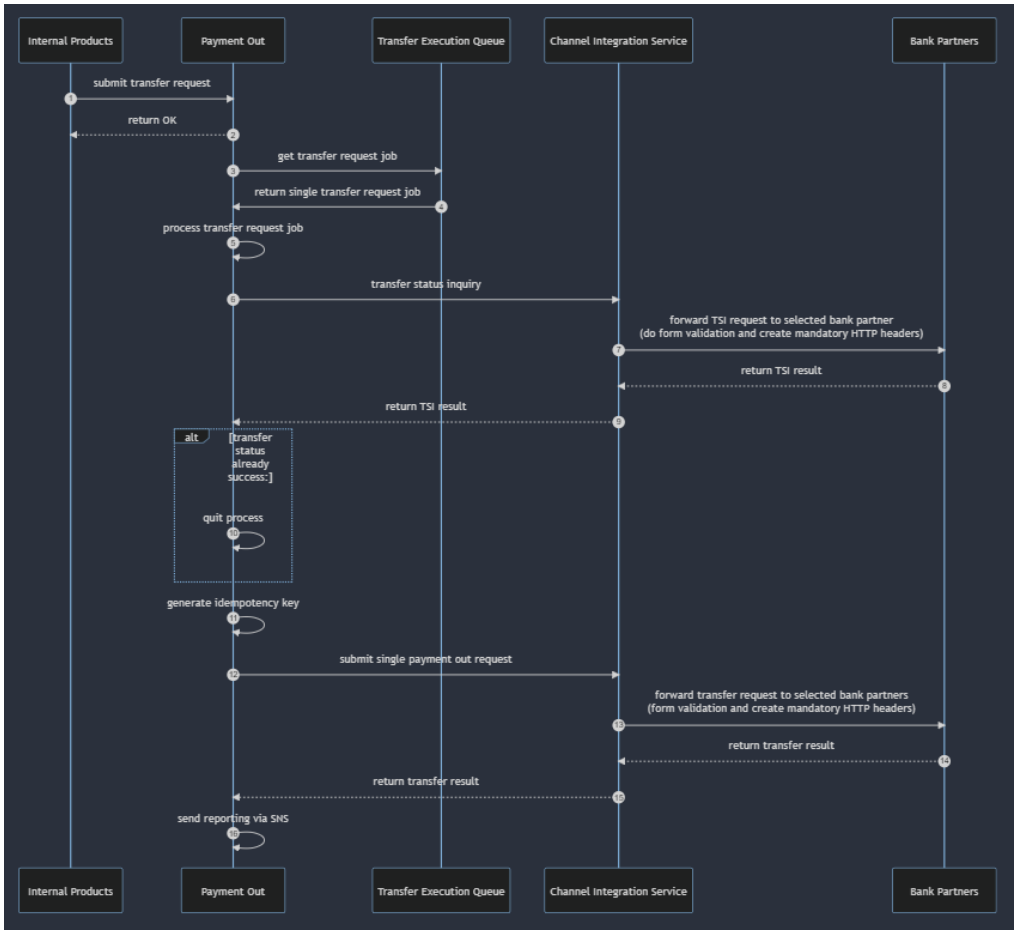
Testing ditempuh dengan memicu servis yang melakukan *reporting* tersebut dengan mengirim komunikasi RPC. Dan penulis menambahkan fungsi *logging* pada kode di lingkup *staging* sehingga dapat dimonitor melalui Cloudwatch

Pada akhir proyek, penulis berkoordinasi dengan tim yang berkaitan untuk mengkonfirmasi bahwa data sudah diterima dengan baik.

3.2.2.3 Pembaharuan Sistem Transfer Bank

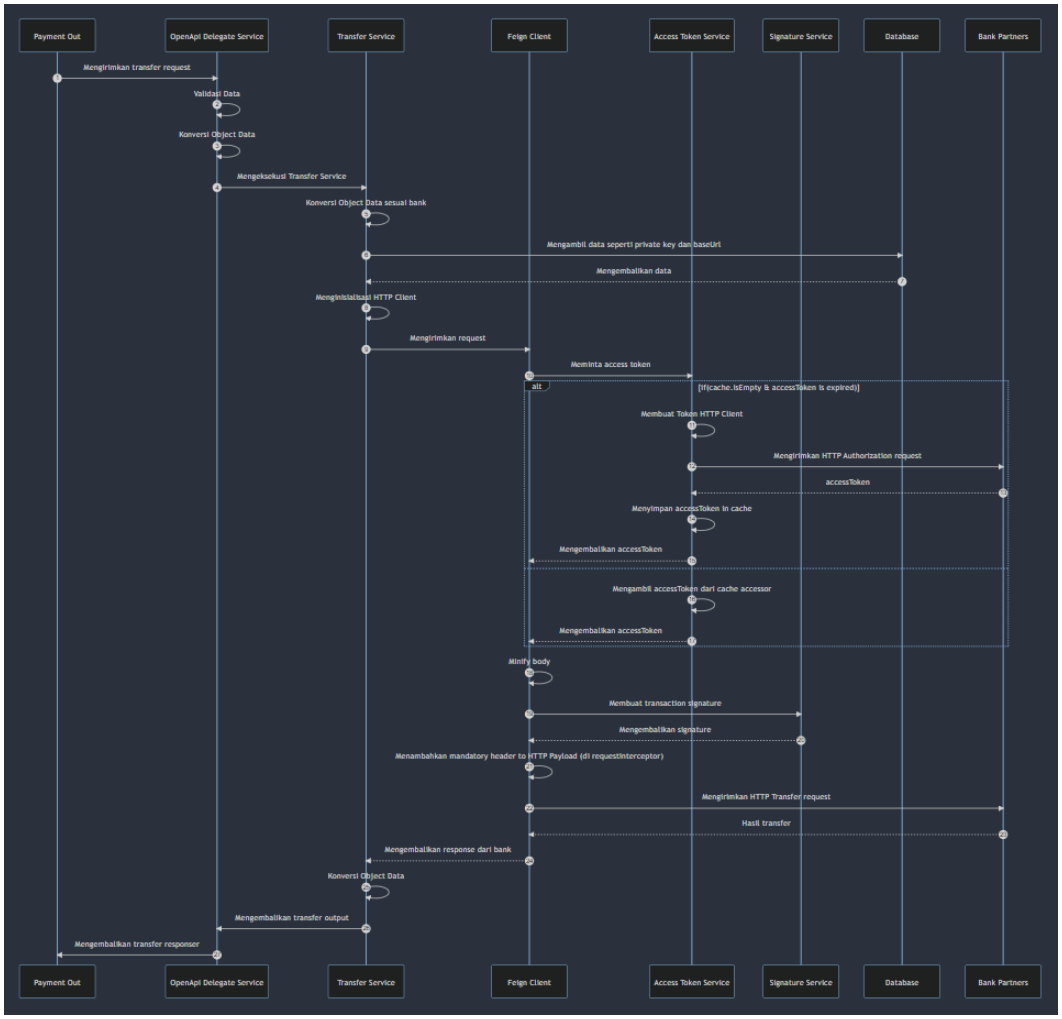
Pada proyek ini penulis berkontribusi pada *microservice* baru yaitu suatu *microservice* yang bernama Channel Integration, servis ini digunakan sebagai *microservice* untuk integrasi API dengan pihak eksternal seperti bank dan sebagainya untuk kebutuhan transfer dan *virtual account*. Dikarenakan sebelumnya alur transfer tidak menggunakan microservice ini, maka akan ada beberapa modifikasi penanganan error dan pembuatan generalisasi DTO yang membuat proses pengembangan penulis menjadi lebih lama dari pada proyek lainnya.

Untuk flow transfer yang akan diubah sebagai berikut :



Gambar 3.15 Struktur alur proyek transfer

Untuk Alur kerja *microservice channel integration* sebagai berikut:



Gambar 3.16 Struktur alur *microservice channel integration*

Pada *microservice* channel integration tidak ada *business logic*, namun terdapat data validasi, penambahan HTTP *header* yang wajib, dan *signature service*. Untuk SNAP menggunakan keamanan *access token* dan juga *signature* yang dienkripsi dengan *public* dan *private key*.

1. Pengembangan

Ada beberapa tugas yang perlu dikerjakan oleh penulis dalam proyek ini :

1. Melakukan perencanaan dan pembuatan API Contract dimana merupakan generalisir dari beberapa request bank sekaligus. (Perlu diingat bahwa walaupun

bank sudah menggunakan SNAP yang terstandarisasi namun implementasi dari API bank bisa saja berbeda beda)

2. Melakukan perencanaan dan pembuatan *centralize error response* dan *error handling*, error harus ditanyani secara tepat dan ini merupakan tantangan karena walaupun error codenya sama namun setiap bank bisa memiliki perbedaan.

3. Pengembangan Feign HTTP Client API untuk mengirim request ke bank sesuai dengan API Contract pada technical documentation bank. Akan menggunakan *request interceptor* untuk memodifikasi *request* agar sesuai dengan persyaratan bank.

4. Mengubah alur sistem transfer menjadi 1 transfer handler yang tergeneralisir.

5. Melakukan UAT bersama dengan bank partner dan membantu permasalahan administrasi bersama manajer proyek.

6. Melakukan uji coba *end to end transfer flow*

Penulis melakukan koordinasi dengan bank partner untuk migrasi API yang sudah ada dan melakukan UAT yang sesuai dengan persyaratan pihak bank.

Penulis membuat *API contract* sesuai kebutuhan data, dimana API contract ini mencakup beberapa bank metode transfer dan bank sekaligus.

Tabel 3.4 Tabel Contoh Endpoint *API Contract* transfer

Endpoint	Fungsi
/payment/out/transfer	Melakukan transfer
/payment/out/transfer-status	Mendapatkan transfer status
/payment/out/inquiry-accout	Mendapatkan data account yang dituju
/payment/out/account-balance	Mendapatkan detail saldo akun

Tabel 3.5 Tabel Contoh Field *API Contract* transfer

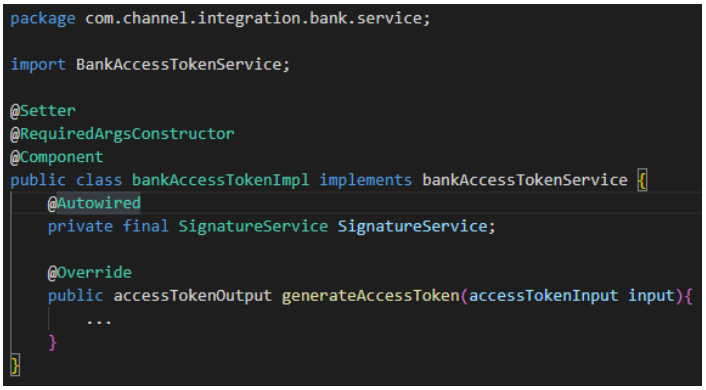
Field	Type
sourceAccountNo	String (15)
beneficiaryAccountNo	String (10)
feeType	String (5)

Tak hanya membuat standar untuk *endpoint dan field data* transfer, namun penulis juga membuat kode setiap metode transfer untuk membedakan bank yang dituju dan metode transfer yang digunakan. Penulis mendokumentasikan *API contract* ini pada dokumen RFC dan juga diimplementasikan pada OpenAPI.

Berikut merupakan beberapa inti pembelajar saat penulis mengerjakan proyek ini :

1. Spring Boot dan autowired

Pada proyek kali ini penulis menggunakan *framework* Spring Boot sehingga terdapat fitur autowired dari Spring Boot. Untuk menggunakan fitur autowired, suatu kelas harus menggunakan anotasi autowired seperti @component, @service, dan lain lain untuk membuat suatu *bean*. Autowired adalah fitur dari spring boot untuk memberi tahu Java Runtime bahwa saat men-*construct* suatu servis harus terlebih dahulu meng-*construct bean* yang sudah didaftarkan, dimana autowired akan memanggil *setter* pada servis yang dituju. Hal ini mempermudah developer dalam menghubungkan suatu servis ke *top level application*. Dimana pada *framework* spring, kita harus mendeklarasikan dan mengirimkan komponen yang dibutuhkan servis secara manual dari kelas *top level application*. *Top level application* yang dimaksud disini adalah kelas yang akan dirun oleh komputer.
Library Lombok juga digunakan disini sebagai *helper* untuk *constructor* objek dan kelas.



Gambar 3.17 Contoh kode untuk Autowired

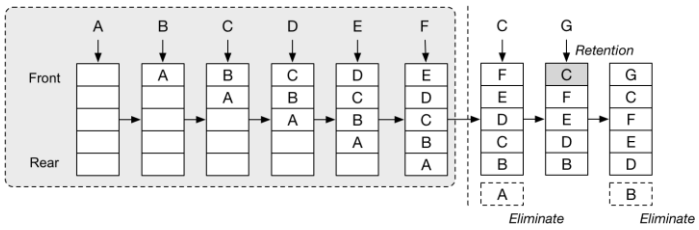
Jika dilihat pada contoh diatas, jika servis bank access token di-*construct* maka secara otomatis servis signature juga akan ikut di-*construct*. Dan tidak diperlukan lagi untuk mendeklarasikan pada *top level application.*

2. Access Token Cache

Setiap *request* transaksi ke bank membutuhkan access token yang digunakan untuk keperluan signature dan juga *authorization header request*. Agar tidak perlu meminta access token berkali kali, digunakanlah LRU cache. LRU cache adalah tempat penyimpanan objek pada RAM yang berbentuk linkedList. LRU cache dideklarasikan sepanjang 10 tempat access token karena servis dapat dijalankan pada *thread processor* yang berbeda. Pada LRU cache disimpan access token dan waktu dibuat dalam milis, sehingga dapat mengecek apakah access token tersebut sudah expired atau belum. Untuk buffer time diset sebesar 10% dari expired access token sesuai dokumentasi bank, misal accessToken yang berfungsi selama 15 menit maka buffer time sebesar 2 menit, jika access token melebihi 2 menit maka servis akan merequest access token yang baru. LRU Cache adalah algoritma penyimpanan yang akan menghapus data yang paling tidak sering dipakai. Untuk lebih jelasnya tentang LRU Cache dapat melihat ilustrasi gambar 3.18.

```java
package com.channel.integration.bank.service;

import BankAccessTokenService;

@Slf4j
@RequiredArgsConstructor
public class bankAccessTokenImpl implements bankAccessTokenService {
    private static final int SECRET_CACHE_SIZE = 10;

    // buffer time = 10 % dari 15 menit = 2 menit
    static final long BUFFER_TIME = Timeunit.MINUTES.toMilis(2);

    private Cache<String, String> cache = new LRUCache<>(SECRET_CACHE_SIZE);

    @Override
    public String generateAccessToken(String input) {
        try {
            accessToken = cache.get(input);
            if(accessToken == null || isAccessTokenExpired(accessToken)) {
                // melakukan request ke bank
                accessToken = refreshAndStoreAccessToken(input);
            }
        } catch (Exception e) {
            // logging
            log.error("Gagal mendapatkan access token dari cache", e)
            // melakukan request ke bank
            accessToken = refreshAndStoreAccessToken(input);
        }
        return accessToken;
    }
}
```

Gambar 3.18 Contoh kode untuk LRU Cache



Gambar 3.19 Ilustrasi LRU Cache

3. OpenAPI

OpenAPI digunakan sebagai pembuatan API Contract dan mempermudah konfigurasi API dalam suatu service. Penulis membuat spesifikasi OpenAPI sesuai dengan API Contract yang sudah disetujui dan berformat file .YAML. Lalu spesifikasi OpenAPI akan digenerate menggunakan gradle dan secara otomatis akan terbuat servis delegatenya dan model request dan responsenya. Model ini nantinya akan dipublish oleh penulis untuk digunakan di microservice payment out untuk memanggil service openAPI ini. Servis delegate ini akan menjadi pintu pertama kita kita memanggil OpenAPI ini, namun untuk service transfer, penulis

tidak mengimplementasikan pada service delegate ini namun membuat service dibawah delegate dengan dilengkapi DTO untuk meningkatkan clean architecture. Untuk lebih jelasnya dapat melihat gambar 3.16 yang berisi struktur alur microservice ini.

4. Feign

Feign adalah library HTTP client yang digunakan oleh penulis untuk mengirim request kepada bank sesuai API Contract pada technical documentation bank. Dalam implementasinya, penulis membuat beberapa komponen sebagai berikut :

1. Suatu kelas interface yang berisi fungsi dan endpoint API, dimana interface ini akan digunakan oleh feignBuilder untuk membuat HTTP Clientnya.

```java
package com.channel.integration.bank.api;

import feign;

@Headers({
    "Authorization: {accesssToken}"
    "Content-Type: application/json"
})
public interface feignTransferClient {
    @Nonnull
    @RequestLine("POST /transfer-intrabank")
    TransferResponse transferToBank(
        @Nonnull TransferRequest request,
        @Nonnull @Param("accessToken") String accessToken
    ) throws ApiResponseException
}
```

Gambar 3.20 Contoh kode untuk Interface Feign

Pada interface ini terdapat endpoint, jenis HTTP yang akan dikirim, header, dan juga model request dan response.

2. Request dan Response model sebagai DTO

Penulis membuat object DTO menggunakan library Lombok sebagai construtornya.

3. Error decoder dan adapter

Untuk menangkap error dari response bank, kita perlu membuat suatu object enum yang berisi seluruh error code dari bank dan nantinya akan lakukan mapping

error code untuk generalisasi error code atau untuk keperluan error handling. Error decoder menggunakan gson decoder untuk mendapatkan value dari anotasi json.

4. Response mapper

Penulis membuat suatu fungsi mapper dengan bantuan library mapStruct. Dimana dibuat suatu interface yang berisikan fungsi dan anotasi mapStruct untuk mengkonversikan object dan nantinya akan diimplementasikan oleh library tersebut secara otomatis untuk dibuatkan objek buildernya. Dalam fungsi konversi objek dapat dilakukan juga suatu fungsi logic jika diinginkan, seperti ingin mengkalkulasikan 2 data untuk menghasilkan 1 konversi data. Response mapper merupakan salah satu komponen yang penting karena suatu error harus ditangani secara tepat. Misal untuk penanganan error seperti berikut :

Tabel 3.6 Tabel Contoh penanganan error

Error Code	Error Message	Reason	Handling
40000	Bad Request	INVALID_DATA	Need to check
40302	Melebihi jumlah transfer	EXCEED_LIMIT_AMOUNT	Need to check

5. Request Interceptor

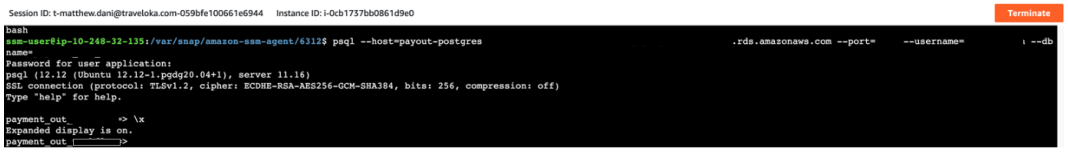
```java
package com.channel.integration.bank.service;

import feign;

@Slf4j
@RequiredArgsConstructor
public class requestInterceptorTransfer implements FeignRequestInterceptor {
    @Override
    public void apply(final RequestTemplate template) {
        try {
            // mengambil data dari paramater store dan integration data
            IntegrationData data = getData();

            // melakukan logic untuk membuat data
            String signature = signatureService.createSignature(data);

            // menambahkan header yang dibutuhkan
            template.header("Signature", signature);
            template.header("Origin", BankConstant.ORIGIN_VALUE);
        } catch (Exception e) {
            log.error("Gagal menambahkan header", e);
        }
    }
}
```

Gambar 3.21 Contoh kode untuk Request Interceptor

Request interceptor memiliki fungsi penting dalam microservice channel integration, karena fungsinya adalah membuat suatu request memenuhi syarat request API bank. Dimana akan dilakukan modifikasi request seperti penambahan header, pengurangan header, melakukan minify body request, dan lain lain. Salah satunya adalah menjalankan service signature pada fungsi ini.

5. Microservice Channel Integration Data

Untuk melakukan service, microservice channel integration data memerlukan beberapa data. Oleh karena itu penulis juga melakukan query ke database postgres untuk menambahkan data untuk integration data seperti endpoint, port, proxy endpoint, timeout time, dan key parameter store. Untuk data yang lebih confidential seperti private key, client id, client secret, dan sebagainya akan disimpan di AWS parameter store.



Gambar 3.22 Tampilan saat mengakses AWS RDS Database PostgreSQL dari *instance bastion*

Dengan AWS Parameter Store (dan juga penyimpanan database postgreSQL), *value* yang penting dapat diganti secara dinamik selagi program berjalan. Hal ini karena program akan meminta ke AWS Parameter Store secara berulang kali jika dibutuhkan (*caching* sudah di-*handle* oleh AWS SDK). Disisi lain jika menggunakan file .env (jika pada Java Spring disebut dengan file .config) maka akan dibaca satu kali pada saat suatu servis di-*construct* (di *top level application component*) dan disimpan pada variabel dengan parameter final. Oleh karena itu tidak bisa dilakukan pergantian *value* secara dinamik saat program berjalan jika menggunakan file .config.

Dan juga jika menggunakan AWS Parameter Store akan dilakukan otorisasi *machine to machine* dengan role yang terikat pada *instance microservice*

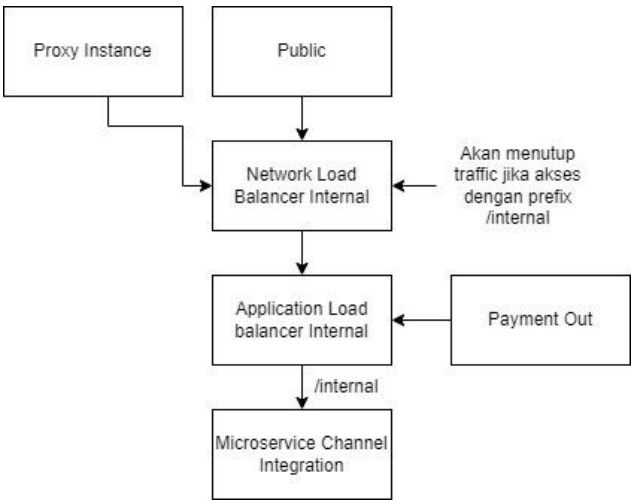
itu sendiri, sehingga keamanannya sangat terjamin. Bahkan untuk mengganti dan melihat *value* AWS Parameter Store di *web dashboard* AWS membutuhkan AWS *role* khusus.

6. Administrasi

Untuk melakukan *request* ke bank transfer, harus dilakukan pendaftaran ip address instance *microservice* yang digunakan. Hal ini dapat dilakukan menggunakan *instance proxy* atau IP NAT *application load balancer* yang memiliki AWS elasticIP atau public static IP. Keduanya memiliki kelebihan dan kekurangannya masing masing. Jika menggunakan proxy instance maka bisa saja proxy tersebut mengalami gangguan yang akan berakibat fatal, namun jika menggunakan IP NAT dari *application load balancer*, maka seluruh instance yang berada di 1 VPC akan bisa menggunakan layanan ini karena IP addressnya sama.
Penulis juga membuat public key dan private key untuk digunakan sebagai pembuatan dan verifikasi signature untuk keamanan transaksi. Penulis menggunakan tool OpenSSL untuk membuat public dan private key. Spesifikasi key yang dibutuhkan seperti X509, RSA, PKCS8, dan 2048 bit.

7. Keamanan

Untuk keamanan OpenAPI microservice channel integration menggunakan keamanan dari load balancer. Network load balancer dikonfigurasi untuk tidak melewatkan traffic yang berasal dari public ke prefix domain yang sudah ditentukan. Misal untuk prefix /internal/ akan diblock dari traffic public, maka domain yang menggunakan prefix ini misal /internal/transfer/ akan tidak akan bisa diakses oleh eksternal VPC. Hal ini dapat diatur melalui terraform untuk mengatur infrastruktur AWS.



Gambar 3.23 Arsitektur keamanan *microservice channel integration*

8. Mengoptimalkan penanganan error pengambilan access token

Penulis melakukan optimalisasi secara inisiatif, yaitu ketika access token tidak berhasil didapatkan yang dikarenakan masalah koneksi atau sesuatu yang tidak diinginkan, maka tranfer request tidak akan dikirimkan. Hal ini akan membuat sistem berjalan efisien secara keseluruhan kapan pun.

Solusinya penulis membuat suatu ApiException baru yang diberi nama AccessTokenApiException dan service access token dipanggil di servis yang lebih tinggi dari service request interceptor, dimana sebelumnnya di panggil di request interceptor. Dimana servis yang menerima exception tersebut akan membawa ke topLevelComponent. Penulis berhasil untuk membatalkan request jika tidak ada access token yang didapatkan.

6. UAT / User Acceptance testing

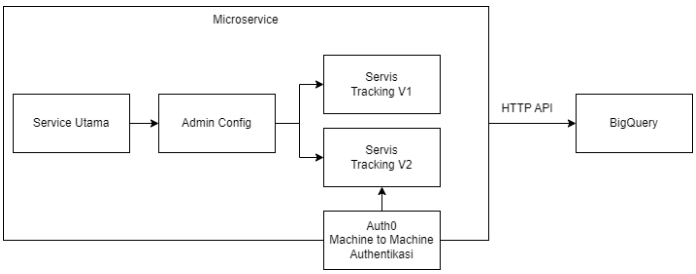
UAT dilakukan untuk menjalankan program dengan scenario yang tidak diinginkan. Misal untuk scenario koneksi gagal, data yang dikirimkan salah, tidak mendapatkan access token, dan sebagainya. Dengan scenario yang gagal, program diharapkan tetap dapat menghasilkan hasil yang diekspektasikan. Penulis melakukan UAT API dengan berkoordinasi dengan pihak bank untuk menyesuaikan kondisi perilaku API bank.

2. Hasil

Setelah melakukan proses pengembangan dan UAT yang panjang, penulis berhasil menyelesaikan proyek ini dengan hasil *end to end testing* semua berjalan dengan baik.

3.2.2.4 Tracking

Tracking akan dijalankan ketika ada beberapa skenario seperti terjadi permintaan transfer, koreksi data, dan lain lain. Berikut merupakan arsitektur proyek tracking pada gambar 3.24.



Gambar 3.24 Arsitektur proyek *tracking*

1. Pengembangan

Selama pengembangan proyek ini, penulis mengerjakan tugas:

1. Migrasi menggunakan SDK *tracking* V2 dan meminta tim data melakukan *publish schema* tracking V2
2. Membuat *admin config*
3. *Monitoring* pada Google Big Query

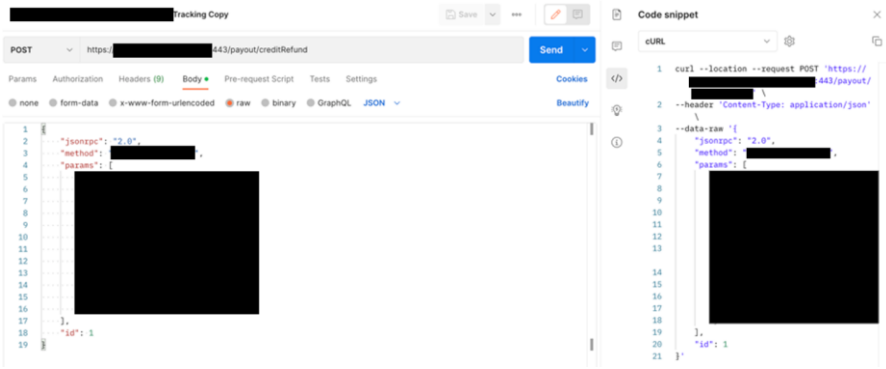
Awal pengerjaan penulis harus berkoordinasi dengan tim data dan membuat tiket untuk merilis *schema* baru V2 untuk data *data tracking* yang akan dimigrasi. Untuk mengirimkan *tracking* V2 menggunakan SDK yang sudah dibuat oleh engineer Traveloka.

Servis SDK *tracking* menggunakan auth0 *machine to machine authentication* dan juga HTTP Client, oleh karena itu harus terlebih dahulu di deklarasikan pada *top level component*. *Admin config* digunakan untuk mengatur nyala atau matinya servis *tracking* V2, dimana jika mati akan menggunakan servis *tracking* V1. Admin

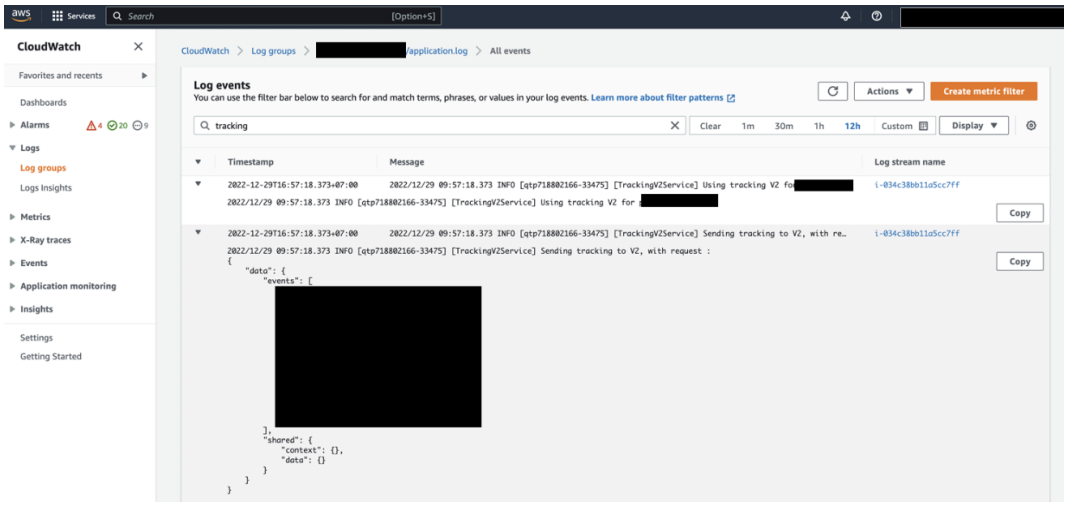
config merupakan data yang disimpan di mongoDB dan dapat dikonfigurasi melakui dashboard internal perusahaan. Admin config didapatkan secara berulang kali ketika *tracking* akan dikirimkan sehingga pengaturan nyala atau mati akan dinamis.

2. Hasil

Pada akhir pengerjaan, penulis melakukan *test staging* dengan menambahkan beberapa fungsi *logging*. Lalu memicu fungsi dengan mengirim RPC dan memonitor melalui AWS Cloudwatch dan BigQuery pada Google Cloud Plafrom. Setelah semua data diterima dengan baik oleh database BigQuery, penulis mengkonfirmasi pada PIC tim data untuk payment. Hasilnya semua data diterima dengan baik menggunakan *tracking* V2



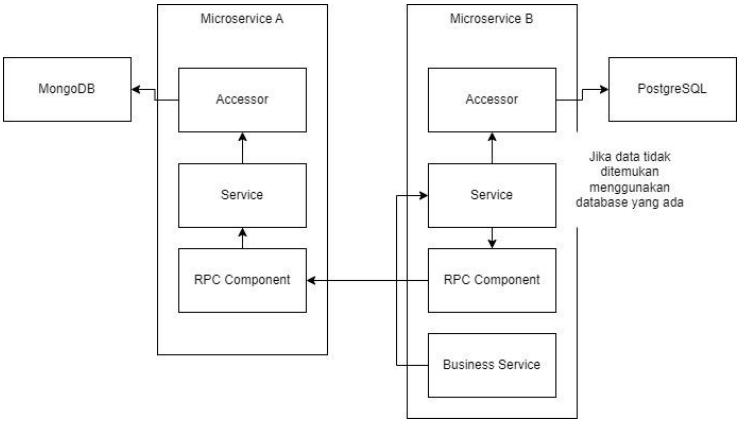
Gambar 3.25 Tahap pertama uji coba *tracking* dengan mengirimkan RPC



Gambar 3.26 Tahap kedua dengan melihat log tracking service pada AWS Cloudwatch

Gambar 3.27 Tahap ketiga dengan memvalidasi data yang ada di Google BigQuery

3.2.2.5 Sentralisasi Penggunaan Data



Gambar 3.28 Arsitektur proyek sentralisasi penggunaan data *microservices*

Microservice B secara normal akan meminta data ke *microservice* A melalui protokol RPC, dan jika data tidak ditemukan maka *microservice* B akan menggunakan kelas accessor data miliknya.

1. Pengembangan

Tugas penulis :

1. Membuat servis yang menggunakan *accessor* dari mongoDB
2. Membuat RPC servlet dan komponen RPC
3. Deklarasi RPC servlet pada *top level application* sesuai konfigurasi seperti menentukan *endpoint* RPC
4. Mempublish komponen RPC untuk digunakan pada microservice A
5. Menggunakan komponen RPC pada microservice A dan membuat config baru

6. Membuat *unit test* pada kedua *microservice*

7. Memperbolehkan komunikasi antar 2 *microservice* menggunakan terraform

Terraform pada kedua *microservice* B dan *microservice* A menggunakan security group id agar memperbolehkan komunikasi antar 2 *instance microservices*. Tak hanya menambahkan security group id namun juga harus menambahkan objek microservice baru yang berisi konfigurasi *firewall* seperti port yang diperbolehkan, tipe *firewall*, *protocol*, dan lain lain di *image* terraform *microservice* tujuan dan mempublish image yang baru tersebut.

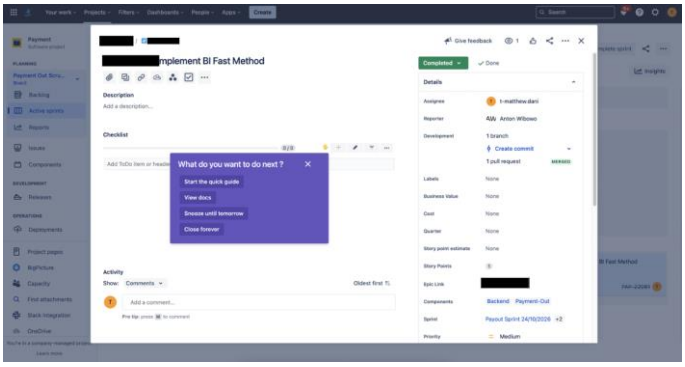
```
resource "aws_security_group_rule" "ingress_[redacted]-lbint" {
  count                    = var.[redacted] != "" ? 1 : 0
  from_port                = "443"
  to_port                  = "443"
  protocol                 = "tcp"
  security_group_id        = aws_security_group.[redacted]-lbint.id
  source_security_group_id = var.[redacted]
  type                     = "ingress"
  description              = "ingress from [redacted]-lbint"
}
```

Gambar 3.29 Contoh konfigurasi Terraform untuk konfigurasi *firewall*

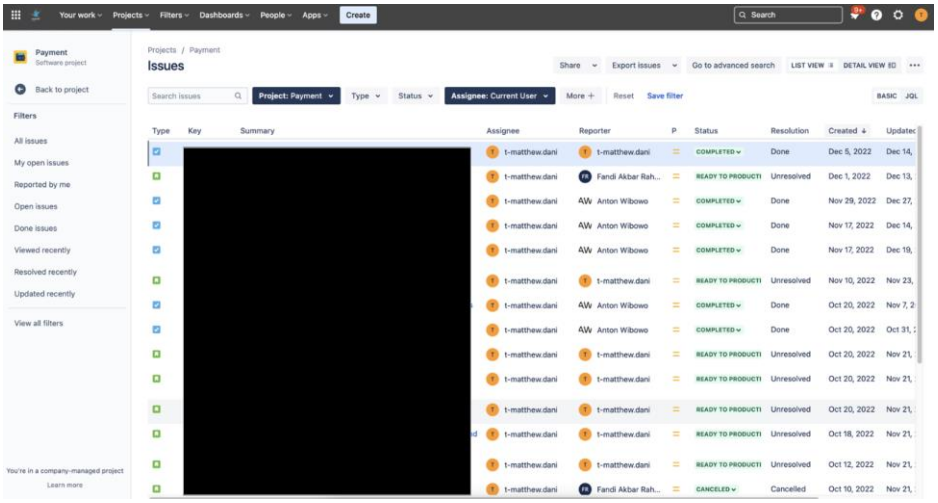
2. Hasil

Penulis melakukan uji coba program dan hasilnya semua berjalan baik dan microservice B menggunakan data dari microservice A.

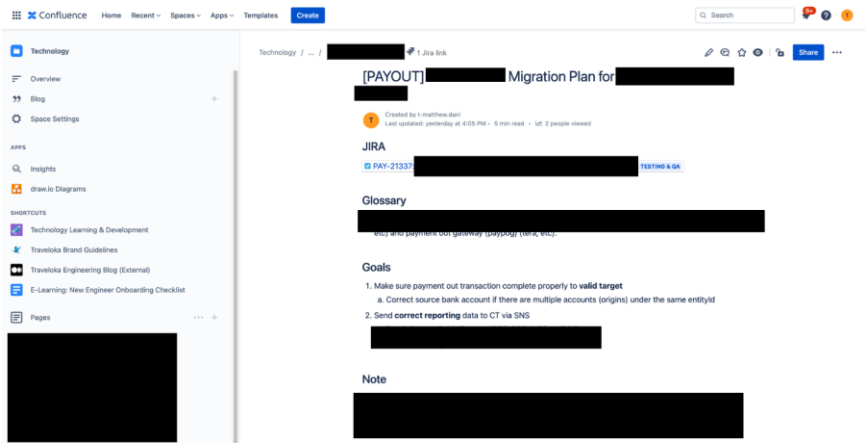
Setiap proyek yang diberikan kepada penulis, didokumentasikan melalui aplikasi Jira dan juga Confluence. Berikut merupakan tampilan dari aplikasi yang digunakan oleh penulis.



Gambar 3.30 Tampilan aplikasi Jira untuk dokumentasi tugas



Gambar 3.31 Tampilan aplikasi Jira



Gambar 3.32 Tampilan aplikasi Confluence untuk menyimpan dokumentasi

3.2.3 Kendala dan Solusi yang Diberikan

Ada beberapa kendala yang dialami oleh penulis ketika mengembangkan proyek proyek program kerja magang di PT Traveloka Indonesia, salah satunya adalah kurangnya pengetahuan penulis untuk menggunakan suatu teknologi sehingga menghambat proses pengembangan, analisis permasalahan, dan perbaikan masalah.

Solusi yang penulis tempuh adalah pertama kali mencari informasi dari dokumentasi perusahaan dan sumber eksternal untuk memahami teknologi yang ada, dan jika tetap tidak menemukan solusi, penulis berdiskusi dengan tim untuk

menanyakan konvensi dan cara untuk menggunakan teknologi tersebut. Misalnya cara untuk memicu suatu servis agar berjalan, pertama kali penulis mencari dokumentasi tentang servis tersebut lalu jika belum menemukan penulis akan bertanya pada tim terkait.

Pada awalnya penulis merasa sulit untuk mengikuti *code convention* yang digunakan dalam 1 domain *microservice*, karena setiap domain bisa saja konvensinya berbeda beda. Misalnya penggunaan *library* dan *depedency* dalam beberapa *microservices* bisa saja berbeda beda. Dan pergantian setiap *depedency* bisa saja berpengaruh besar pada *microservice* tersebut. Solusi dari hal ini adalah mendiskusikan terlebih dahulu dengan tim terkait mengenai proyek yang akan dibuat.

Penulis juga merasakan untuk sulit integrasi pada *business process*, karena setiap pihak eksternal dan *business process* membutuhkan penanganan yang berbeda dan ketika mengeneralisasi suatu integrasi diperlukan ketelitian dan koordinasi yang besar untuk menggabungkan beberapa integrasi tersebut.