BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Organisasi

Pelaksanaan kerja magang dilakukan pada kelompok *Human Capital System* dengan posisi *IT Developer Intern* yang memiliki tugas untuk membantu pengembangan sistem kepegawaian yang dimiliki oleh PT Bank Negara Indonesia Tbk. Kegiatan magang berlangsung di bawah bimbingan dan pengawasan Arief Firmansyah selaku mentor yang juga merupakan asisten manager di kelompok *Human Capital System*. Mentor atau pembimbing lapangan memiliki tugas untuk membimbing, mengawasi, serta memberikan informasi dan arahan terkait dengan tugas yang harus dikerjakan. Fokus utama untuk tugas yang diberikan pada saat kegiatan magang adalah implementasi *microservices* yang menggunakan *framework* Spring Boot. Selain itu, terdapat juga pengembangan aplikasi microservices berbasis web dengan *framework* Spring MVC serta tampilan dengan menggunakan HTML, CSS, dan Bootstrap.

Pemberian tugas atau penugasan dilakukan oleh Arief Firmansyah sebagai pembimbing lapangan. Untuk setiap tugas yang diberikan, terdapat seorang *senior developer* yang ditugaskan untuk membantu dan memberi arahan apabila peserta magang mengalami kesulitan. Setiap dua hari sekali diadakan *checkpoint* untuk melaporkan hasil pekerjaan yang sudah dilakukan serta hal-hal yang perlu dilakukan ke depannya. Pada hari Jumat di setiap minggu, juga diadakan *weekly meeting* untuk membahas *progress* dari proyek yang sedang dikerjakan dalam kelompok *Human Capital System* serta mengingatkan target-target yang harus dicapai. Karena sebagian karyawan masih bekerja secara *hybrid*, maka koordinasi di dalam kelompok dilakukan dengan menggunakan media daring seperti Zoom Meeting dan Whatsapp.

3.2 Tugas yang Dilakukan

Selama pelaksanaan praktik kerja di PT Bank Negara Indonesia Tbk, terdapat beberapa tugas yang dilakukan dengan berfokus pada implementasi *microservices* menggunakan Spring Boot dan pengembangan *backend* Spring MVC. Tugas pertama yang dilakukan sekaligus menjadi tugas akhir dari pelatihan pra-magang adalah membuat sebuah API dengan arsitektur *microservices* untuk

platform CoCreate. Fitur-fitur yang dikembangkan pada CoCreate platform antara lain sign in, registrasi, melakukan *posting*, dan memberikan komentar. CoCreate platform juga menerapkan komunikasi asinkron antar *services* menggunakan Kafka.

Selain melakukan perancangan dan pembuatan aplikasi *backend*, terdapat juga tugas untuk melakukan peningkatan (*enhancement*) fitur pada sistem yang sudah digunakan. *Enhancement* yang dilakukan yaitu, menambahkan fitur *logging* pada aplikasi Self Service, Mobile API Portal, Car Ownership Program, serta menambahkan CRUD (*Create-Read-Update-Delete*) pada aplikasi rekrutment eksternal dan IHCS (*Integrated Human Capital System*). Terakhir, untuk memberikan gambaran terkait dengan arsitektur *microservices* yang digunakan oleh PT Bank Negara Indonesia, terdapat tugas khusus untuk mengimplementasikan *microservices* dengan menggunakan Spring Boot berdasarkan modul yang diberikan.

3.3 Uraian Pelaksanaan Magang

Tugas-tugas yang dikerjakan selama pelaksanaan magang, dikelompokkan ke dalam 20 minggu kerja. Berikut merupakan uraian dari pekerjaan yang dilakukan selama kegiatan magang berlangsung seperti yang dapat dilihat pada Tabel 3.1.

Tabel 3.1. Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang

Minggu Ke -	Pekerjaan yang dilakukan
1	*Onboarding* peserta magang kampus merdeka, pengenalan area kerja pada divisi DGL PT Bank Negara Indonesia Tbk, serta finalisasi tugas akhir *bootcamp* (pelathian pra-magang).
2	Pemberian tugas untuk membuat sebuah *log* khusus pada Spring Boot yang dapat disimpan dalam sebuah *file* untuk keperluan *debugging* dan analisis.
3	Melakukan pengujian *load testing* untuk *project logging* dengan menggunakan JMeter serta perubahan versi *project* ke 2.0.0.
4	Melanjutkan pengujian *project* dengan menambahkan beberapa *abnormal case*, menambahkan *method* DELETE dan PATCH ke dalam API agar dapat melakukan *delete* dan *update* data.

5	Melakukan perubahan *service registry* dari Eureka ke Spring Boot Admin serta melakukan pengujian terhadap perubahan yang sudah dilakukan.
6	Memperbaiki konfigurasi Spring Boot Admin agar dapat menampilkan isi dari *logfile*. Menambahkan fitur notifikasi via email dengan menggunakan SMTP *server*.
7	Implementasi Spring Boot *logging* ke dalam *project* Self Service khususnya untuk proses *checkin* dan *checkout* pegawai.
8	Perubahan metode notifikasi Spring Boot Admin menggunakan Telegram dikarenakan SMTP *server* tidak dapat berjalan dengan baik. Mempelajari implementasi *logging* untuk *project* berbasis Spring MVC.
9	Implementasi *logging* ke dalam *project* Mobile API Portal untuk *endpoint login*.
10	Melakukan perbaikan terhadap *error* dan *bug* serta konfigurasi *logging* agar kompatibel dengan Spring MVC.
11	Melakukan *slicing UI design* ke dalam HTML CSS dan Bootstrap 5 untuk halaman *application history* pada website recruitment eksternal.
12	Membuat tampilan halaman website *application history* menjadi reponsif serta melakukan perbaikan komponen agar sesuai dengan desain yang diajukan.
13	Mempelajari modul ”Master Microservices with Spring Boot and Spring Cloud”.
14	Mengimplementasikan *project microservices* yang terdapat dalam modul pembelajaran.
15	Membuat laporan, *deploy* ke *remote server*, serta presentasi hasil implementasi *microservices* menggunakan Spring Boot.
16	Melakukan dokumentasi *testing internal* untuk *project* Self Service dan Mobile API Portal.
17	Mengimplementasikan Spring Boot *logging* pada *project* Car Ownership Program.
18	*Enhancement* aplikasi rekrutment eksternal dari tampilan yang lama ke tampilan baru.

19	Promosi *project logging* Self Service dan Mobile API Portal.
20	Membuat CRUD *backend* dan *frontend* untuk *project* SK digital.

Pada minggu pertama, pelaksanaan magang belum sepenuhnya dimulai dikarenakan seluruh peserta diwajibkan terlebih dahulu untuk melaporkan hasil akhir dari tugas pelatihan pra-magang. Tugas akhir dari pelatihan berupa sebuah aplikasi *backend* yang dapat digunakan untuk menampilkan karya dan inovasi dari para pegawai. Terdapat beberapa MVP atau fitur-fitur minimum yang harus terdapat di dalam aplikasi diantaranya *user* dapat mendaftar pada aplikasi, melakukan *posting* konten, dan memberi komentar. Agar perbandingan antara arsitektur *microservices* dan *monolithic* dapat terlihat dengan jelas, aplikasi diwajibkan untuk menggunakan dua buah *database* serta komunikasi secara asinkron menggunakan Kafka. Selain mempresentasikan hasil dari tugas akhir pelatihan, pada minggu pertama juga terdapat orientasi program magang. Program orientasi berisi pengenalan area kerja yang akan ditempati, tugas dari setiap kelompok pada area kerja, dan peraturan yang harus dipatuhi selama kegiatan magang berlangsung.

Pada minggu kedua, diadakan pertemuan dengan pembimbing lapangan dan beberapa anggota di kelompok *Human Capital System* (HCY) untuk mendiskusikan terkait dengan *project* yang harus dikerjakan. Terdapat sebuah *project* permintaan dari divisi Operasional Teknologi Informasi (OTI) untuk melakukan penambahan fitur *logging* yang dapat menyaring seluruh *request* yang masuk ke dalam API. Fitur *logging* nantinya akan diterapkan secara bertahap ke seluruh *project* yang berada di bawah kelompok HCY sebagai bentuk *monitoring* terhadap seluruh permintaan dan respon dari dan menuju *server*. Harapannya adalah jika terjadi anomali atau kegagalan sistem dapat segera dideteksi dengan melakukan visualisasi melalui aplikasi bernama Kibana.

Pelaksanaan proyek dimulai dengan membuat sebuah API sederhana berbasis Spring Boot yang dapat melakukan *get* dan *post* data. Konsep pembuatan API diibaratkan seperti sistem registrasi di mana user dapat melakukan penambahan data dan menampilkan seluruh data yang terdapat dalam *database*. Setelah API selesai dan berjalan dengan baik, kemudian ditambahkan sebuah package *filter* yang berfungsi untuk menyaring setiap *request* yang masuk. Fitur *logging* kemudian ditambahkan di dalam *filter* yang akan menampilkan informasi dari *request* yang masuk seperti *request body*, URL, *host name*, dll. Selain ditampilkan

melalui *console*, setiap *log* juga akan tercatat dalam sebuah *file* agar dapat diolah menggunakan Kibana.

Pada minggu ketiga, purwarupa atau *prototype* dari implementasi fitur *logging* sudah selesai dan dapat dilanjutkan ke tahap uji coba. Namun, sebelum pengujian berlangsung terdapat perubahan terhadap versi *project* agar sesuai dengan yang digunakan oleh BNI. Kegiatan pengujian dilakukan dengan melakukan uji beban (*load testing*) pada API yang telah dibuat. Pengujian ini bertujuan untuk memastikan bahwa fitur *logging* yang ditambahkan tidak berpengaruh secara signifikan terhadap performa sistem secara keseluruhan.

Hasil pengujian yang sudah dilakukan sebelumnya kemudian menjadi bahan evaluasi dan perbaikan pada *project* yang dilakukan pada minggu keempat. Terdapat penambahan metode untuk menghapus dan meperbarui data agar skema pengujian bisa lebih beragam. Selain itu, pada saat melakukan pengujian juga diusulkan untuk menambahkan beban *request* yang masuk dari 100 ke 1000 *request* secara bersamaan.

Dalam konsep *microservices*, terdapat *service registry* yang berfungsi untuk mempublikasikan setiap *service* yang terdaftar. Sebelumnya, *service registry* yang digunakan adalah Eureka, namun atas permintaan untuk pengembangan akhirnya pada minggu kelima disepakati *service registry* yang digunakan menjadi Spring Boot Admin. Terdapat beberapa kelebihan dari Spring Boot Admin, yaitu dapat melihat *health status* dari setiap *service* yang terdaftar, melihat *logfile*, serta mengirim notifikasi jika terdapat perubahan status pada *service*.

Pada minggu keenam dilakukan implementasi fitur Spring Boot Admin, yaitu *monitoring logfile* dan notifikasi via email dengan menggunakan SMTP *server*. Spring Boot Admin dapat membaca isi dari *logfile* dan menampilkannya di *dashboard* sehingga tidak perlu lagi mencari *logfile* yang tersimpan di direktori. Selain itu, Spring Boot Admin juga dapat mengirimkan notifikasi melalui email, Telegram, Hipchat, dan Slack jika terdapat perubahan status pada *service*.

Setelah implementasi fitur *logging* dan Spring Boot Admin berjalan dengan baik, pada minggu ketujuh dilakukan implementasi pada sistem eksisting, yaitu aplikasi Self Service. Fitur *logging* tidak diterapkan pada seluruh *endpoint* aplikasi. Hanya *endpoint* untuk *checkin* dan *checkout* pegawai yang akan menampilkan *log*. Mekanisme pembatasan *endpoint* dilakukan dengan menggunakan *file* JSON yang di dalamnya terdapat daftar *endpoint* yang dapat menampilkan *log*.

Pada minggu kedelapan, terdapat masalah ketika akan melakukan pengujian notifikasi Spring Boot Admin. Setelah ditelusuri, masalah disebabkan karena

SMPT *server* tidak dapat berjalan pada *localhost*. Oleh karena itu, dilakukan perubahan metode notifikasi dengan menggunakan Telegram. Notifikasi melalui telegram memanfaatkan Telegram bot untuk mengirimkan pesan yang dapat didaftarkan pada konfigurasi Spring Boot Admin. Selain menerapkan fitur *logging* pada Spring Boot, terdapat permintaan juga untuk menerapkan fitur ini pada project Mobile API Portal yang berbasis Spring MVC. Agar proses implementasi berjalan dengan lancar, maka terlebih dahulu dilakukan penelusuran bagaimana cara mengimplementasikan fitur *logging* pada Spring MVC.

Pada minggu kesembilan, implementasi *logging* pada *project* Mobile API Portal mulai dilakukan dengan terlebih dahulu melakukan *clone project* pada gitlab. Saat *project* dijalankan ternyata terdapat kendala, yaitu *request* tidak dapat diterima oleh API. Setelah dilakukan *debugging* dan penelusuran ditemukan bahwa masalah terdapat pada proses otorisasi. Ternyata *endpoint* yang dituju tidak terdapat pada daftar *endpoint* yang diizinkan sehingga semua *request* yang masuk akan langsung ditolak.

Berbeda dengan fitur *logging* pada Spring Boot yang menggunakan *filter*, pada Spring MVC digunakan *interceptor* untuk dapat menyaring *request* yang masuk. Secara konsep, penerapan *logging* pada Spring Boot dan Spring MVC sebenarnya tidak jauh berbeda namun, terdapat kendala yang disebabkan karena *request body* yang masuk tidak diteruskan ke *controller*. Oleh karena itu, dibutuhkan *request* dan *response body advice* agar *request* dan *response body* tidak tertahan di *interceptor*. Selain itu, *interceptor* harus didaftarkan dengan menggunakan *class* konfigurasi agar dapat berjalan dengan baik pada Spring MVC.

Selepas menyelesaikan tugas *logging*, pada minggu kesebelas dan kedua belas terdapat penugasan untuk mengerjakan UI/UX untuk *project enhancement* rekrutmen eksternal. Tahap pertama yang dikerjakan, yaitu melakukan *slicing design UI* yang telah disediakan ke dalam format HTML serta menggunakan format tampilan CSS dan Bootstrap. Tampilan yang dibuat adalah untuk halaman *application history* dimana *user* dapat melihat proses rekrutmen yang sudah dan akan dijalani. Setelah tampilan terbentuk, langkah selanjutnya adalah membuat tampilan menjadi responsif agar dapat menyesuaikan ukuran layar di semua jenis perangkat.

Pada minggu ketiga belas hingga kelima belas, sebagai pembekalan sebelum mengerjakan *project* selanjutnya, diberikan sebuah modul pembelajaran terkait dengan konsep arsitektur *microservices* yang digunakan di BNI. Karena terdapat beberapa bagian dari aplikasi yang bersifat *confidential* maka dibuat sebuah mockup

agar dapat menggambarkan alur dari *microservices*. *Mockup* dibuat menyerupai aplikasi yang dapat digunakan untuk melakukan pertukaran mata uang. Terdapat dua *service* di dalamnya, yaitu *service* konversi dan *service* kalkulasi. Setelah implementasi selesai, hasilnya kemudian didokumentasikan dan dipresentasikan kepada pembimbing agar dapat mengetahui sejauh mana pemahaman terkait konsep *microservices*.

Sebelum melakukan promosi atau menaikkan aplikasi ke *server*, terlebih dahulu harus dilakukan tahap pengujian terhadap fungsi fungsi di dalamnya. Proses ini dilakukan pada minggu keenam belas yang dimulai dengan melakukan dokumentasi pada project *logging*. Dokumentasi berisi tentang fitur yang ditambahkan, alur dari fitur tersebut, serta kondisi normal ketika program dapat dijalankan dengan baik.

Pada minggu ketujuh belas ternyata terdapat satu project tambahan yang belum diimplementasikan fitur *logging*, yaitu Car Ownership Program. Aplikasi ini memiliki fungsi untuk pengajuan program kepemilikan kendaraan pribadi oleh karyawan. Adapun *endpoint* yang perlu untuk ditampilkan dalam *log* adalah *save*, *approver*, dan *eligible*. Proses implementasi yang dilakukan tidak menemui kendala berarti dikarenakan kerangka fitur *logging* sudah terbentuk sehingga dapat dengan mudah digunakan di berbagai *project*.

Project rekrutmen eksternal yang sudah dimulai pada minggu kesepuluh dilanjutkan pengerjaannya ke bagian *backend* pada minggu kedelapan belas. Bagian *backend* dari aplikasi rekrutmen eksternal menggunakan Spring MVC dengan *database* Oracle. Pembaruan yang dilakukan pada aplikasi, yaitu menambahkan operasi CRUD untuk data-data yang sebelumnya bersifat statis. Beberapa diantaranya seperti judul dan deskripsi halaman, informasi kontak, testimoni, dll.

Pada minggu kesembilan belas dilaksanakan UAT atau *User Acceptance Test* untuk *project logging* pada Self Service dan Mobile API Portal. Pelaksanaan UAT merupakan tahap terakhir yang dilakukan sebelum sebuah *project* siap dipublikasikan ke *server*. Selama proses UAT berlangsung dilakukan pengujian terhadap format *log*, proses atau cara kerja fitur *logging*, dan kesesuaian dengan ketentuan yang diminta oleh *user*. Setelah lulus, maka *project* dapat dipublikasikan atau dipromosikan ke *server*. Pelaksanaan promosi dilakukan pada malam hari ketika aplikasi sudah jarang digunakan agar tidak mengganggu aktivitas para karyawan.

Tugas terakhir dalam program magang ini adalah penambahan fitur SK Digital pada aplikasi IHCS (*Integrated Human Capital System*). Implementasi

tugas hampir sama dengan *project* rekrutmen eksternal yang menggunakan Spring MVC. Namun, dikarenakan keterbatasan waktu, tugas kemudian diambil alih oleh karyawan senior di kelompok HCY.

Proses pelaksanaan praktik kerja magang akan diuraikan berdasarkan tugas yang dikerjakan, yaitu *Logging Enhancement*, *Microservices Currency Exchange*, dan *Rekrutmen Eksternal*.

3.3.1 Logging Enhancement

A. Ketentuan Format

Kegiatan *enhancement* pada aplikasi dilatarbelakangi kebutuhan untuk dapat melakukan *monitoring* pada aktivitas yang sering dilakukan seperti *login*, absensi, dll. Maka, dibutuhkan fitur *logging* yang dapat menangkap informasi yang dibutuhkan untuk mengoperasikan, memecahkan masalah dan melaporkan kinerja platform IHCS (*Integrated Human Capital System*), DigiHC dan komponen penyusunnya. Catatan *log* dapat dilihat dan diolah secara langsung oleh pengguna dan sistem, diindeks dan dimuat ke dalam penyimpanan data, dan digunakan untuk menghitung metrik serta menghasilkan laporan. Berdasarkan hasil diskusi dengan divisi Operasi Teknologi Informasi (OTI), disetujui format *log* yang digunakan seperti Gambar 3.1.

```
[INFO] 2022-02-10 00:20:31,415 simpanan.Connection jsonPost - 464AE8C93C3C7AB282A4CABE3410202D
ID : 20220210000157323448
STEP : Api Create Account
URL : http://soa.dp:44071/eform/simpanan
HTTP_RESP : 200
HOST : eformsimpanan.service.bni.co.id
REQ : {
  "channel": "EFORM",
  "tellerId": "",
  "accountNum": "",
  "branch": "008",
  "accountType": "2003",
  "subCat": "1001",
  "othersOpenAccReason": "Orangtua",
  "othersSrcOfFund": "Orangtua",
  "idNum": "1405050212990002",
  "idType": "0001",
}
RES : {"refNum":"EFOR851315518","channel":"EFORM","cifNum":""
,"accountNum":"","customerName":"AGGER PRABOWO","idNum":"1405050212990002"
,"idType":"0001","status":"FAILED","errorCode":"9018",
"errorMessage":"(SOA) FAILED TO VALIDATE IDENTITY (POB)"}
RESPONSETIME : 2250
```


Gambar 3.1. Format *Log*

Keseragaman format *log* dibutuhkan untuk mempermudah proses pengolahan data. *Log* yang tersimpan di dalam *file* nantinya akan diolah

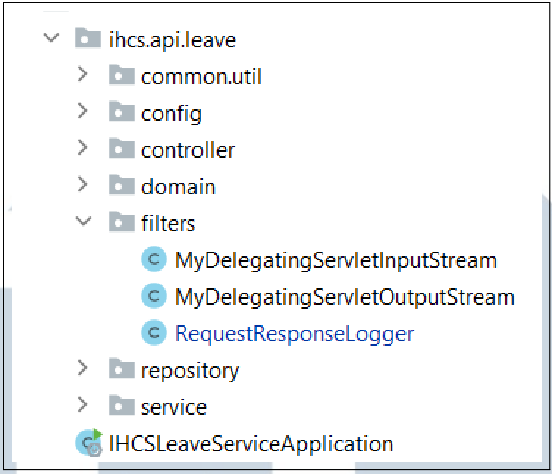
menggunakan aplikasi Kibana untuk divisualisasikan dalam bentuk histogram, grafik, dan diagram. Adapun informasi yang harus terdapat di dalam log, yaitu :

1. ID : Identitas unik yang menjadi pembeda antara *log* satu dengan yang lainnya.
2. STEP : Deskripsi singkat tentang *request* yang diminta.
3. URL : Alamat untuk mengakses atau melakukan *request*.
4. HTTP_RESP : Berisi status dari *request* yang dikirimkan apakah berhasil atau gagal.
5. HOST : Nama dari *host* atau penyedia layanan untuk melakukan *request*.
6. REQ : Data atau informasi yang dikirimkan.
7. RES : Nilai kembalian atau respon dari *server*.
8. RESPONSETIME : Waktu yang dibutuhkan untuk memproses *request* hingga selesai.

B. Hasil Implementasi

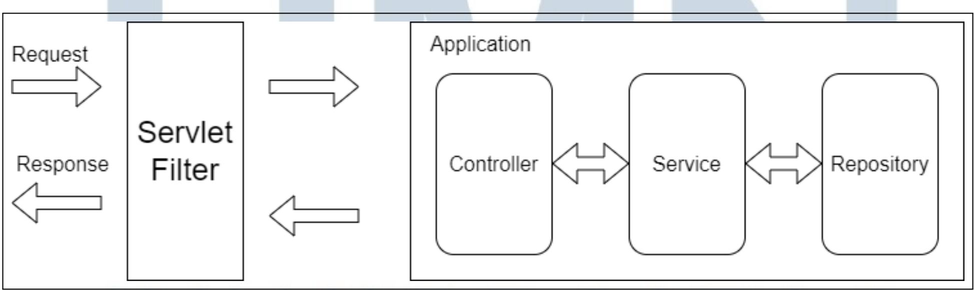
Filter adalah objek yang digunakan untuk menyaring *request* dan respons HTTP dari aplikasi.[5] Terdapat dua operasi yang dapat dijalankan di dalam *filter*, yaitu menyaring informasi *request* sebelum diteruskan ke *controller* dan menyaring informasi respons sebelum diteruskan ke *client*. Kemampuan dari *filter* tersebut kemudian digunakan untuk dapat menampilkan informasi dari *request* dan respons ke dalam *log*. Implementasi *filter* di dalam *project* sangat sederhana dan hanya membutuhkan satu buah *package* khusus. Contoh struktur *project* setelah penambahan *filter* dapat dilihat pada Gambar 3.2.

UNIVERSITAS MULTIMEDIA NUSANTARA



Gambar 3.2. Struktur *Project*

Posisi *filter* di dalam *project* Spring Boot memiliki letak yang terpisah dengan aplikasi, yaitu tepat sebelum *controller*. Ketika terdapat *request* yang masuk, *filter* akan menyaring informasi *request* dan meneruskannya ke dalam *controller*. Setelah itu, *controller* akan memetakan *request* ke fungsi tertentu dan meneruskannya ke *service* untuk menerapkan *business logic*. Terakhir, terdapat *repository* yang berfungsi untuk melakukan operasi CRUD *database* melalui DAO (*Data Access Object*). Hasil akhir berupa respons dari aplikasi, kemudian dikembalikan dan ditangkap kembali oleh *filter* sebelum diteruskan ke *client*. Ilustrasi alur kerja *filter* dapat dilihat pada Gambar 3.3.



Gambar 3.3. Ilustrasi Alur Kerja *Filter*

Terdapat tiga buah *method* utama dari *object filter*, yaitu *init*, *doFilter*, dan *destroy*. *Method init* akan dipanggil saat pertama kali instansiasi *bean* sedangkan *destroy* akan dipanggil sebelum bean dihilangkan dari *container*. Pada implementasi *logging* ini digunakan *method doFilter* yang akan terpanggil secara langsung ketika terdapat *request* yang masuk. Oleh karena itu, hanya *log* yang memiliki *step* atau terdaftar di dalam *file* JSON yang dapat ditampilkan. *Method* utama *filter* dapat dilihat pada Gambar 3.4.

```java
Anthony Rafael
@Override
public void init(FilterConfig filterConfig) throws ServletException {

}

Anthony Rafael *
@Override
public void doFilter(ServletRequest servletRequest,
                     ServletResponse servletResponse,
                     FilterChain filterChain)
        throws IOException, ServletException {

}

Anthony Rafael
@Override
public void destroy() {

}
```

Gambar 3.4. *Method* Utama *Filter*

Pada *method doFilter*, terdapat tiga buah parameter *object*, yaitu *servlet request*, *servlet response*, dan *filter chain*. Untuk mengambil data atau informasi terkait dengan *request* dan respons dapat digunakan *method* yang telah disediakan oleh *servlet request* dan *servlet response*. Nilai dari id, diambil dari tanggal dan waktu diterimanya *request*. Kemudian untuk *response time* diambil dari jarak antara waktu pertama kali *request* diterima oleh *filter* dengan waktu respons diterima oleh *filter*. Selain dua data tersebut, nilainya dapat diambil menggunakan *method* dari *servlet request* dan *servlet response*. Implementasi pengambilan data *request* dan respons dapat dilihat seperti pada Gambar 3.5.

UNIVERSITAS MULTIMEDIA NUSANTARA

```java
long start = System.currentTimeMillis();
CustomHttpRequestWrapper requestWrapper = new CustomHttpRequestWrapper((HttpServletRequest) servletRequest);

SimpleDateFormat f = new SimpleDateFormat( pattern: "yyyy/MM/dd HH:mm:ss:SSS");
String date = f.format(new Date());
UUID uuid = UUID.randomUUID();

String method = requestWrapper.getMethod();
String uri = requestWrapper.getRequestURI();
String step = null;

JSONParser jsonParser = new JSONParser();

String url = String.valueOf(requestWrapper.getRequestURL());
String serverName = requestWrapper.getServerName();
String reqBody = new String(requestWrapper.getByteArray());

CustomHttpResponseWrapper responseWrapper = new CustomHttpResponseWrapper((HttpServletResponse) servletResponse);

filterChain.doFilter(requestWrapper, responseWrapper);

long end = System.currentTimeMillis() - start;

int status = responseWrapper.getStatus();

String resBody = new String(responseWrapper.getBaos().toByteArray());
```

Gambar 3.5. Pengambilan Data

Terdapat sebuah masalah ketika *request* dan *response body* dibaca di dalam *filter*, yaitu *request* tidak akan diteruskan ke dalam *controller* dan respons tidak diteruskan ke *client*. Hal tersebut dapat terjadi dikarenakan *request* dan respons dianggap sudah terkonsumsi di dalam *filter*. Untuk mencegah agar hal tersebut tidak terjadi, maka dapat digunakan *wrapper* untuk membungkus *request* dan *respone body*. Implementasi *request* dan *response wrapper* dapat dilihat pada Gambar 3.6 dan Gambar 3.7.

```
2 usages   Anthony Rafael
private class CustomHttpRequestWrapper extends HttpServletRequestWrapper {
    3 usages
    private byte[] byteArray;

    1 usage   Anthony Rafael
    public CustomHttpRequestWrapper(HttpServletRequest request) {
        super(request);
        try {
            byteArray = IOUtils.toByteArray(request.getInputStream());
        } catch (IOException e) {
            throw new RuntimeException("Issue while reading request stream");
        }
    }

    Anthony Rafael
    @Override
    public ServletInputStream getInputStream() throws IOException {
        return new MyDelegatingServletInputStream(new ByteArrayInputStream(byteArray));
    }

    1 usage   Anthony Rafael
    public byte[] getByteArray() { return byteArray; }
}
```

Gambar 3.6. *Request Wrapper*

```
private class CustomHttpResponseWrapper extends HttpServletResponseWrapper {
    2 usages
    private ByteArrayOutputStream baos = new ByteArrayOutputStream();

    2 usages
    private PrintStream printStream = new PrintStream(baos);

    1 usage   Anthony Rafael
    public ByteArrayOutputStream getBaos() { return baos; }

    1 usage   Anthony Rafael
    public CustomHttpResponseWrapper(HttpServletResponse response) { super(response); }

    Anthony Rafael
    @Override
    public ServletOutputStream getOutputStream() throws IOException {
        return new MyDelegatingServletOutputStream(new TeeOutputStream(super.getOutputStream(), printStream));
    }

    Anthony Rafael
    @Override
    public PrintWriter getWriter() throws IOException {
        return new PrintWriter(new TeeOutputStream(super.getOutputStream(), printStream));
    }
}
```

M U L T I M E D I A

Gambar 3.7. *Response Wrapper*

N U S A N T A R A

Penentuan *step* untuk setiap *request* akan diambil berdasarkan *endpoint* yang dituju. Daftar dari setiap *endpoint* dan deskripsinya tersimpan di dalam sebuah *file* berformat JSON seperti pada Gambar 3.8. Oleh karena itu, data dari *file* JSON harus dimuat ke dalam sebuah *object* menggunakan JSON *parser* dan dikonversikan

ke dalam bentuk JSON *object*. Selanjutnya, dilakukan iterasi atau perulangan dengan menggunakan *iterator* dan *for looping* untuk mencocokkan antara *key* dari JSON dengan URI atau *endpoint* yang didapatkan dari *request*. Jika terdapat kecocokkan antara *key* dengan URI maka variabel *step* yang awalnya bernilai *null* didefinisikan kembali dengan nilai dari *key*. Potongan kode untuk penentuan step secara dinamis dapat dilihat pada Gambar 3.9.

step.json

C: > json > step.json > ...

```json
{
  "GET": {
    "/cop/api/selfService/request/check-eligible": "Check Eligible"
  },
  "POST": {
    "/cop/api/selfService/request/save": "Save",
    "/cop/api/selfService/request/approver" : "Approver",
    "/selfservice/api/absen/check-in-check-out" : "Checkin-Checkout"
  },
  "DELETE": {
    "/user": "Delete"
  },
  "PATCH": {
    "/user": "Update"
  }
}
```

Gambar 3.8. *File Step* JSON

```java
try {
    Object obj = jsonParser.parse(new FileReader( fileName: "/home/ihcs/services/logging/json/step.json"));

    JSONObject jsonObject = (JSONObject) obj;

    JSONObject step1 = (JSONObject) jsonObject.get(method);
    Iterator<String> keys = step1.keySet().iterator();


    if (step1.size() > 0) {
        for (Iterator iterator = step1.keySet().iterator(); iterator.hasNext(); ) {
            String key = (String) iterator.next();
            if(key != null && !key.isEmpty()) {
                if (uri.contains(key)) {
                    step = (String) step1.get(key);
                }
            }
        }
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ParseException e) {
    e.printStackTrace();
} catch(NullPointerException e) {
    e.printStackTrace();
}
```

Gambar 3.9. Mekanisme *Step* Dinamis

Data-data yang sudah dikumpulkan, kemudian akan ditampilkan dengan menggunakan *logger* dengan *level info*. *Log level* atau tingkat kepentingan log, adalah status yang memberi tahu betapa pentingnya pesan *log* yang diberikan. Hal ini sangat memudahkan ketika ingin melakukan pelacakan terhadap *log* yang ditampilkan. Beberapa *log level* yang sering digunakan, yaitu *INFO*, *ERROR*, *WARN*, *DEBUG*, dan *TRACE*. Dengan mempertimbangkan tujuan dari *log* untuk memberikan informasi terkait dengan *request* yang masuk, maka dapat digunakan *level info* untuk tingkat kepentingannya. Sebelum menampilkan *log*, terdapat satu tahap lagi yang harus dilakukan, yaitu validasi. Tidak semua *request* yang masuk akan ditampilkan ke dalam *log*, karena hal tersebut dapat memakan *resources* yang cukup banyak. Potongan kode untuk menampilkan log dapat dilihat pada Gambar 3.10.

```
if(step!=null && !step.isEmpty()) {
    log.info("[INFO] {} - {} \n" +
            "ID : {}\n" +
            "STEP : {}\n" +
            "URL: {}\n" +
            "HTTP_RESP : {}\n" +
            "HOST : {}\n" +
            "REQ : {}\n" +
            "RES : {}\n" +
            "RESPONSETIME : {}", date, uuid, getCurrentlyDateTime(), step, url, status, serverName, reqBody, resBody, end);
}
```

Gambar 3.10. Potongan Kode *Logger*

Logback adalah salah satu kerangka kerja *logging* yang paling banyak digunakan di Java dan merupakan pengganti dari pendahulunya, Log4j.[6] Logback menawarkan implementasi yang lebih cepat, menyediakan lebih banyak opsi untuk konfigurasi, dan lebih banyak fleksibilitas dalam mengarsipkan *file log* lama. Agar format *log* yang dihasilkan sesuai dengan ketentuan maka perlu dilakukan konfigurasi untuk logback dengan menggunakan *file* xml. Beberapa atribut yang dapat dikonfigurasi melalui *file* logback.xml ini, yaitu *logging pattern*, lokasi penyimpanan *log file*, serta aturan *rolling file*. Untuk aturan *rolling file* digunakan *time-based policy* yang akan melakukan perubahan *file* setiap hari. Konfigurasi logback pada *file* xml dapat dilihat pada Gambar 3.11.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property name="LOG_PATTERN" value="%m%n" />
  <property name="APP_LOG_ROOT" value="/home/ihcs/services/logging/log/"/>
  <property resource="application.properties" />
  <timestamp key="timestamp" datePattern="dd-MM-yyyy" />

  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
      <encoder>
          <pattern>${LOG_PATTERN}</pattern>
      </encoder>
  </appender>

  <appender name="applicationLog" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${APP_LOG_ROOT}/logfile-${server.address}-${server.port}.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>${APP_LOG_ROOT}/logfile-${server.address}-${server.port}-%d{dd-MM-yyyy}.log</fileNamePattern>
    </rollingPolicy>
    <encoder>
        <pattern>${LOG_PATTERN}</pattern>
    </encoder>
  </appender>

  <logger name="com.ihcs.api.leave.filters" level="INFO">
    <appender-ref ref="applicationLog" />
    <appender-ref ref="console" />
  </logger>
</configuration>
```

Gambar 3.11. Konfigurasi Logback

Saat menjalankan aplikasi, biasanya terdapat beberapa *log* berisi informasi yang tidak dibutuhkan. Untuk menghindari informasi tersebut tercampur ke dalam *log* maka perlu dilakukan pembatasan untuk *logging level* yang ditampilkan. Pada tingkat *root* aplikasi hanya perlu ditampilkan *log* yang memiliki tingkat kepentingan *error*. Namun, saat mengatur *root logging level* menjadi *error*, secara otomatis seluruh kelas yang ada di aplikasi hanya dapat menampilkan *log* yang memiliki *level error*. Oleh karena itu, perlu ditambahkan pengaturan khusus untuk kelas ResponseLogger agar dapat menampilkan *log*. Konfigurasi serta hasil keluaran *log* dapat dilihat pada Gambar 3.12 dan Gambar 3.13.

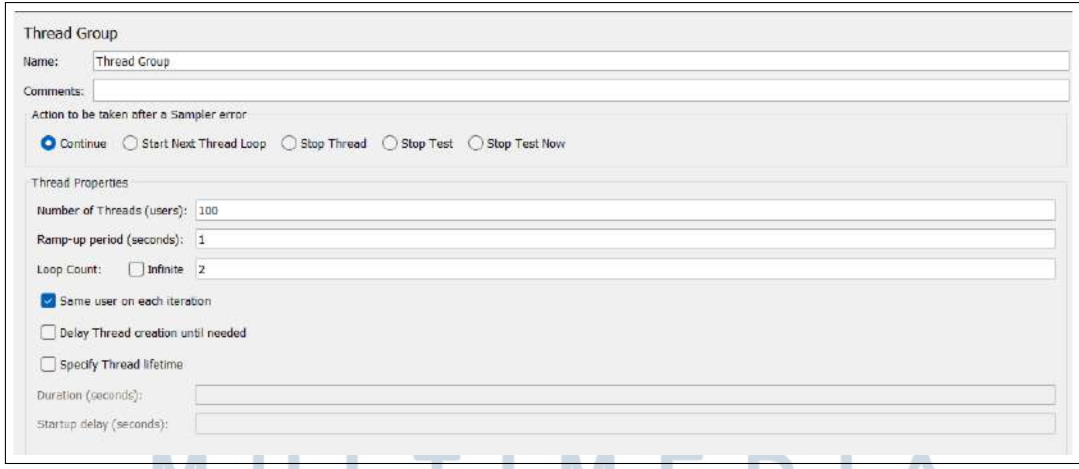
```
##Log
logging.level.root=ERROR
logging.level.com.ihcs.api.leave.filters.RequestResponseLogger=INFO
```

Gambar 3.12. Konfigurasi *Logging Level*

```
logfile-localhost-8672 - Notepad
File   Edit   View

[INFO] 2022/06/16 10:07:53:986 - 59de20c2-5124-4538-9d54-398639350c56
ID : 20220616100754
STEP : Checkin-Checkout
URL: http://localhost:8672/selfservice/api/absen/check-in-check-out
HTTP_RESP : 200
HOST : localhost
REQ : {
 "actionType": "checkout",
 "attPolygonId": 2564,
 "mobileLatitude" : -7.8016521,
 "mobileLongitude": 110.3645683,
 "timezoneCode": "WIB"
}
RES : {"data":null,"status":false,"message":"Gagal Check Out, Anda dapat checkout setelah jam [18:00]","code":402}
RESPONSETIME : 357
```

Gambar 3.13. Contoh *Log file*

Setelah fitur *logging* dapat berjalan dengan baik, diperlukan *testing internal* berupa *load testing* untuk menguji performa aplikasi ketika ditambahkan fitur *logging*. Aplikasi yang digunakan untuk *load testing* adalah JMeter. Untuk melakukan *load testing*, dibutuhkan sebuah *thread group* dan *sampler*. *Thread group* merupakan kumpulan dari *thread* yang merepresentasikan satu buah *user*. Jumlah *thread* dapat didefinisikan pada bagian *number of threads*. Persyaratan minimum jumlah *user* yang diujicobakan untuk *load testing* pada BNI adalah sebanyak 100 *threads* dengan perulangan sebanyak dua kali. Konfigurasi thread group dapat dilihat pada Gambar 3.14.

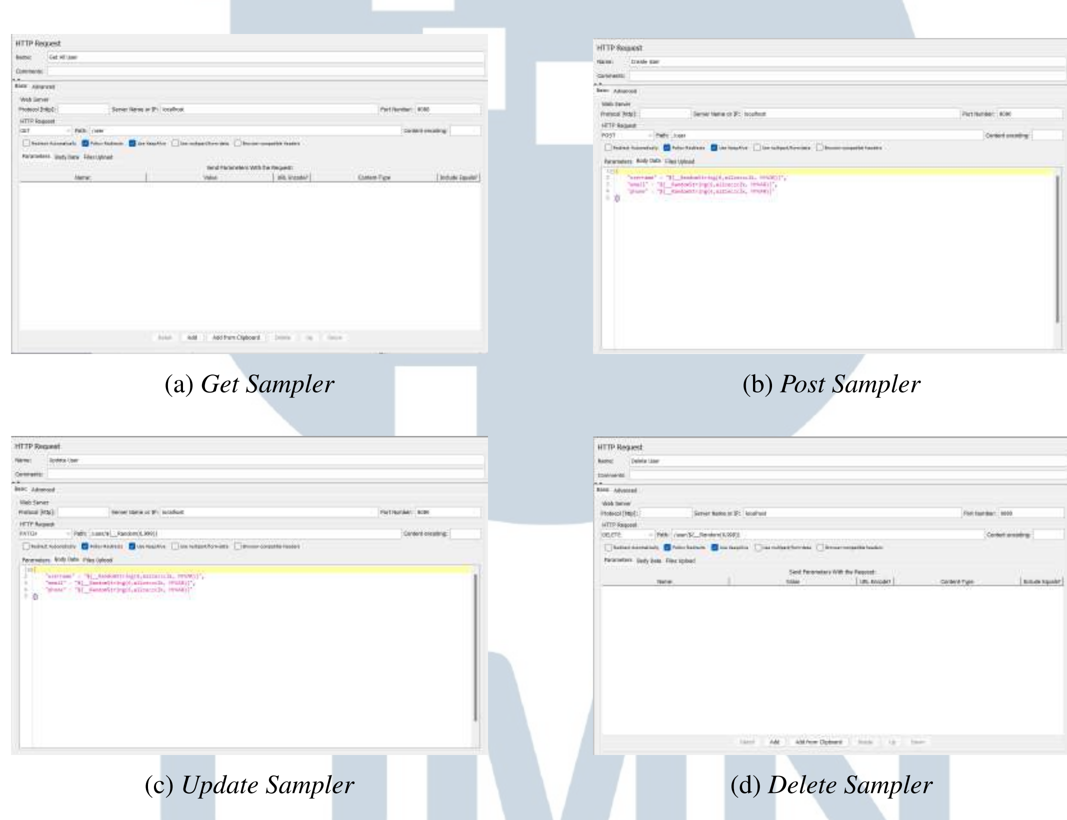


Gambar 3.14. Konfigurasi *Thread Group*

MULTIMEDIA
NUSANTARA

Setiap *thread group* harus memiliki *sampler* yang berfungsi untuk mengirimkan *request* serta mendefinisikan tipe dari *request* yang dikirim. Pada *load testing* ini tipe *request* yang dikirimkan adalah HTTP request. Terdapat empat *method* yang diujikan dalam *load testing*, yaitu *Get*, *Post*, *Update* (*Put*), dan

Delete. Adapun informasi yang perlu disediakan agar *request* bisa sampai ke tujuan, antara lain nama *server*, *port*, *method*, dan *path*. Selain itu, untuk *method* yang membutuhkan *input* atau masukkan diperlukan konfigurasi untuk jenis data yang dikirimkan serta nilai nya. JMeter menyediakan fungsi untuk menghasilkan data acak yang dapat digunakan sebagai *input* dengan menggunakan metode *Random* untuk data *number* dan *Random String* untuk data *string*. Konfigurasi untuk *sampler* selengkapnya dapat dilihat pada Gambar 3.15.



(a) *Get Sampler*

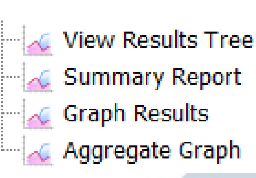
(b) *Post Sampler*

(c) *Update Sampler*

(d) *Delete Sampler*

Gambar 3.15. HTTP *Request Sampler*

Hasil dari pengujian dapat ditampilkan dengan cara menambahkan *listener*. *Listener* adalah komponen yang dapat menunjukkan hasil dari sampel. Terdapat beberapa jenis hasil yang dapat ditampilkan seperti *tree*, tabel, grafik atau dapat ditulis ke *file log*. Pada pengujian fitur *logging* ini digunakan 4 jenis hasil, yaitu *tree*, tabel (*report*), *graph*, dan *aggregate graph*. Namun, untuk kebutuhan perhitungan yang lebih jelas digunakan bentuk tabel atau *report* sebagai acuan utama dalam menentukan apakah sesuai dengan standar minimal yang diharapkan. Hasil pengujian dapat dilihat pada Gambar 3.16



(a) Ragam Hasil Pengujian

(b) Grafik Hasil Pengujian

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get All User	200	1549	237	4678	765.58	0.00%	25.45825	6581.58	2.98	264729.1
Create User	200	923	9	3383	701.7	12.00%	29.80626	7.03	7.39	241.5
Update User	200	534	11	2183	473.84	79.50%	31.19152	7.16	7.89	234.9
Delete User	200	495	11	1741	463.27	82.00%	45.78755	9.71	9.39	217.2
TOTAL	800	875.25	67	2996.25	601.0975	43%	33.06	1,651.37	6.91	66,355.68

(c) Tabel Hasil Pengujian

Gambar 3.16. Hasil Pengujian

Berikut merupakan komponen hasil pengujian:

1. *Samples* : Jumlah HTTP *request* yang dijalankan. Misalnya, jumlah *user* 100 dan *loop* 2 maka jumlah sampel secara keseluruhan adalah 100 x 2 = 200.
2. *Average* : Waktu respons rata-rata untuk setiap HTTP *request* (dalam satuan ms).
3. *Min & Max* : Waktu respons tercepat dan terlama untuk setiap HTTP *request* (dalam satuan ms).
4. *Std. Dev* : Berapa banyak kasus yang ditemukan yang menyimpang dari nilai rata-rata waktu respons. Semakin kecil nilainya maka semakin konsisten.
5. *Error* : Berapa banyak *error* yang terjadi selama pengiriman *request* (contohnya 404 *not found*).
6. *Throughput* : Jumlah permintaan per unit waktu (detik) yang dikirimkan selama pengujian.
7. *Received & Send* : Berapa KiloBytes per detik yang diterima oleh *server* dan *client*.
8. *Avg. Bytes* : Rata rata jumlah respons (dalam bytes).

Tidak semua komponen hasil pengujian digunakan untuk menentukan apakah aplikasi layak untuk dilajutkan ke tahap *delivery*. Komponen yang digunakan sebagai bahan pertimbangan utama, yaitu *average response time* dan *std. dev*. Kedua komponen tersebut kemudian dibandingkan dengan kondisi

sebelum ditambahkan fitur *logging*. Semakin kecil perbedaan atau interval nya maka akan semakin baik juga. Berdasarkan hasil diskusi dan juga pertimbangan hasil pengujian, disepakati bahwa fitur *logging* sudah sesuai dengan ketentuan awal dan dapat dilanjutkan ke tahap UAT (*User Acceptance Test*) dan *delivery*.

Selain menambahkan fitur *logging*, terdapat *enhancement* untuk menambahkan Spring Boot Admin. Spring Boot Admin adalah aplikasi web, yang digunakan untuk mengelola dan memantau aplikasi Spring Boot. Setiap aplikasi akan dianggap sebagai klien dan harus mendaftar ke *server* admin. Fitur utama dari Spring Boot Admin adalah untuk melakukan *monitoring* terhadap setiap *service* yang didaftarkan. Terdapat beberapa alasan dipilihnya Spring Boot Admin, yaitu:

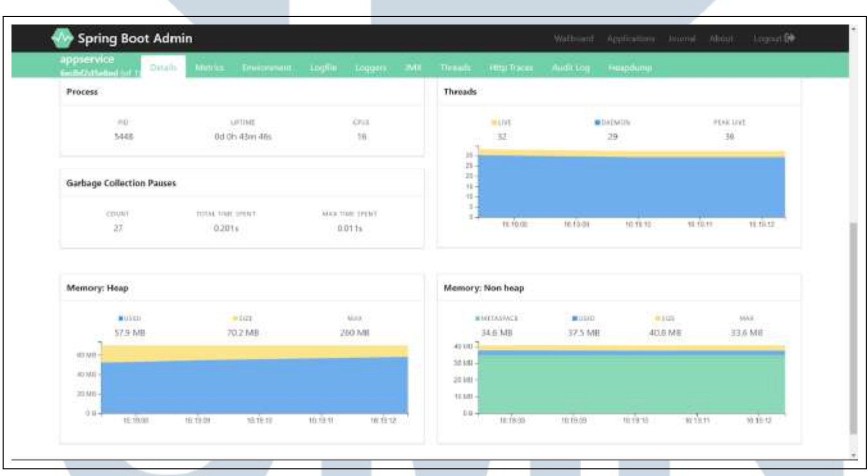
1. Dapat Menampilkan *cache metrics* yang mengukur performa *disk caching*, yaitu *average read* dan *write latency*, *hit rate*, *insertion rate*, dll.
2. Dapat Menampilkan dan mengunduh *logfile* yang tersimpan di dalam *local system*.
3. Dapat melihat Spring Boot Configuration Properties yang terdapat pada *file* application.properties.
4. Dapat mengubah dan mengatur *log level* dengan mudah melalui *interface* yang telah disediakan.
5. Dapat Menampilkan atau melihat *http-endpoints*.
6. Dapat memberi notifikasi ketika terdapat perubahan status melalui email, Slack, Hipchat, dll.

Proses pembuatan Spring Boot Admin terbilang cukup mudah dan sederhana. Langkah yang harus dilakukan adalah membuat sebuah project Spring Boot, kemudian ditambahkan sebuah *dependency* Spring Boot Admin seperti Gambar 3.17. Alamat dan *port* dari Spring Boot Admin dapat dilakukan konfigurasi melalui *file application properties*. Terakhir, setiap *service* atau aplikasi hanya perlu mendaftar dengan cara menambahkan alamat dan *port* dari Spring Boot Admin.

```xml
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-dependencies</artifactId>
            <version>${spring-boot-admin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Gambar 3.17. *Dependency* Spring Boot Admin

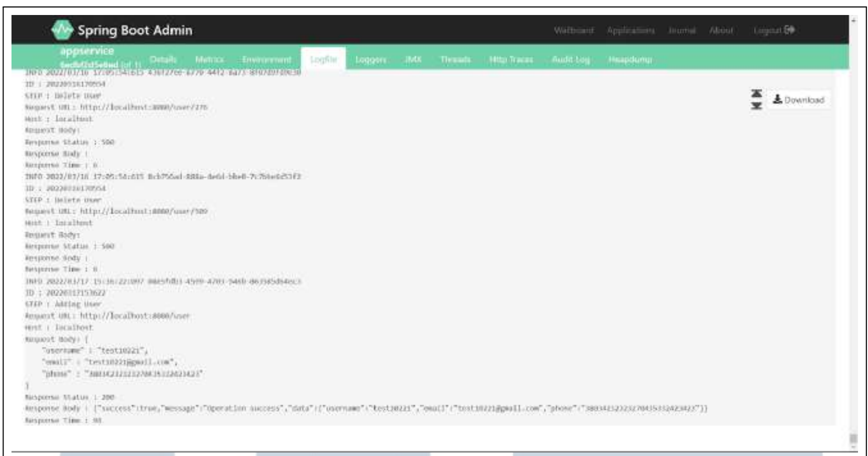
Hasil implementasi Spring Boot Admin dapat dilihat pada Gambar 3.18. Saat aplikasi berjalan, Spring Boot Admin dapat melakukan *monitoring* penggunaan sumber daya seperti *threads*, *memory heap*, dan *memory non heap*. Indikator-indikator tersebut sangat diperlukan oleh divisi operasional sehingga jika terjadi anomali pada aplikasi dapat segera dideteksi.



Gambar 3.18. Spring Boot Admin *Dashboard*

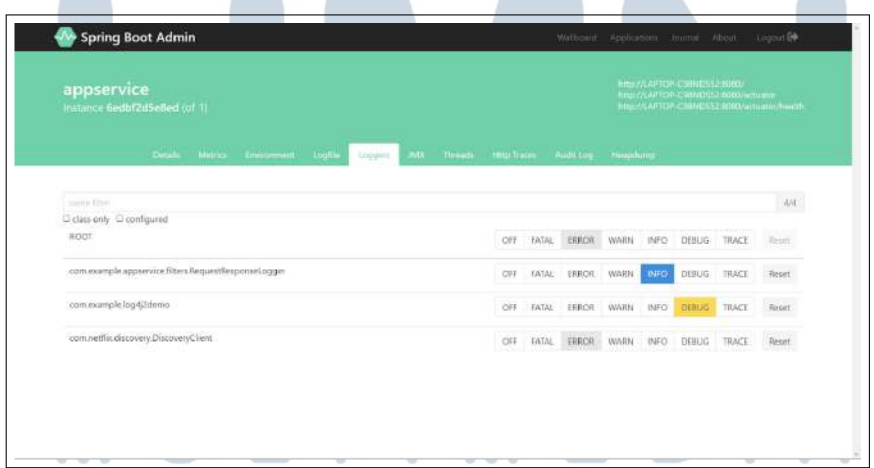
Salah satu keunggulan Spring Boot Admin adalah dapat membaca dan menampilkan *logfile*. Hal ini tentu sangat memudahkan ketika melakukan *monitoring* karena dapat mempersingkat waktu untuk mencari letak dari *logfile*. Lokasi dari *logfile* hanya perlu didefinisikan pada aplikasi agar Spring Boot Admin dapat membacanya. Selanjutnya, isi dari *logfile* akan ditampilkan seperti pada Gambar 3.19

UNIVERSITAS MULTIMEDIA NUSANTARA



Gambar 3.19. *Log Monitoring*

Jika pada bagian sebelumnya untuk mengubah *log level* harus dilakukan pada *file application properties*, dengan menggunakan Spring Boot Admin perubahan *log level* dapat dilakukan secara langsung bahkan ketika aplikasi sedang berjalan. Spring Boot Admin akan secara otomatis menampilkan *list* dari *class* yang dapat menampilkan *log*. Perubahan terhadap *log level* pada Spring Boot Admin sifatnya hanya sementara. Artinya, jika aplikasi dimatikan kemudian dijalankan kembali *log level* akan tetap mengikuti konfigurasi dari *application properties*. Tampilan Spring Boot Admin untuk mengatur *log level* dapat dilihat pada Gambar 3.20.



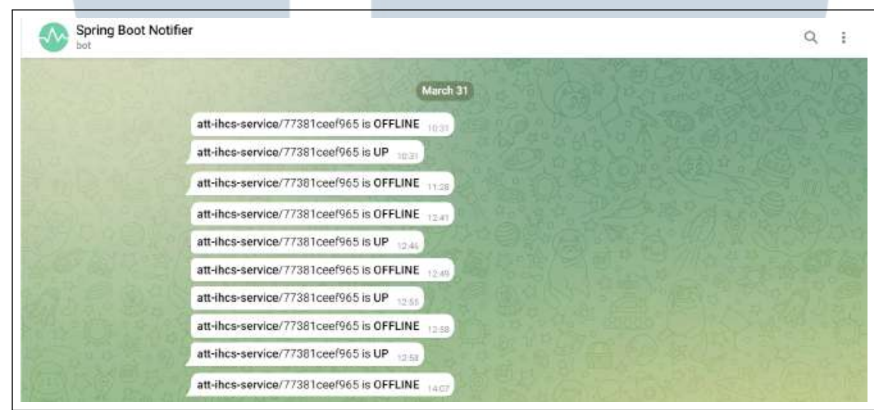
Gambar 3.20. Konfigurasi *Log Level*

Fitur terakhir yang disyaratkan untuk ditambahkan dalam Spring Boot Admin adalah notifikasi melalui Telegram. Pada saat tahap pengembangan, notifikasi awalnya akan dikirim melalui email. Namun, dikarenakan terdapat masalah pada SMPT *server* saat pengujian, maka dilakukan perubahan dengan

menggunakan Telegram. Langkah awal dibutuhkan sebuah Telegram bot untuk dapat mengirimkan pesan secara otomatis. Pada bagian Spring Boot Admin perlu ditambahkan *token* dan *chat id* dari Telegram bot yang telah dibuat seperti pada Gambar 3.21. Spring Boot Admin akan mengirimkan notifikasi secara otomatis ke Telegram jika terdapat perubahan *health status* pada aplikasi yang terdaftar. Contoh notifikasi *health status* pada Telegram dapat dilihat pada Gambar 3.22

```
#Telegram
spring.boot.admin.notify.telegram.auth-token=5112040962:AAFjUxDngJ0G1k9ABVXRaE4BwdP9kig3xbw
spring.boot.admin.notify.telegram.chat-id=1243079539
```

Gambar 3.21. Konfigurasi Notifikasi Telegram



Gambar 3.22. Notifikasi Telegram

Fitur *logging* tidak hanya diterapkan pada aplikasi berbasis Spring Boot tapi juga pada aplikasi berbasis Spring MVC, yaitu Mobile API Portal. Secara umum, tidak terdapat perbedaan signifikan antara penerapan *logging* pada Spring Boot dan Spring MVC. Perbedaan hanya sebatas istilah atau penyebutan misalnya, pada Spring Boot untuk menangkap *request* dan *response* menggunakan *filter* sedangkan pada Spring MVC menggunakan *interceptor*. Namun, ada hal yang perlu dilakukan pada Spring MVC sebelum *interceptor* dapat bekerja, yaitu meregistrasinya pada kelas *configurer* seperti pada Gambar 3.23.

```
@Configuration
public class MyConfig implements WebMvcConfigurer {
        @Override
        public void addInterceptors(InterceptorRegistry registry){
                registry.addInterceptor(new LoggerInterceptor());
        }
```

Gambar 3.23. Konfigurasi Spring MVC

3.3.2 Microservices Currency Exchange

A. Konsep Dasar

Microservices merupakan sebuah konsep arsitektur dimana aplikasi dikembangkan ke dalam *service* atau layanan yang lebih kecil.[7] Setiap *service* berdiri sendiri dan dapat saling berkomunikasi satu sama lain. Keuntungan dari penggunakan *microservices*, antara lain dapat beradaptasi dengan mudah terhadap teknologi baru karena setiap *service* dapat menggunakan teknologi yang berbeda, kemudahan dalam meningkatkan dan mengurangi kapasitas (dynamic scaling), serta waktu perilisan yang cepat karena antar *service* tidak saling bergantung. Adapun tantangan ketika menggunakan *microservices* dan solusinya, yaitu:

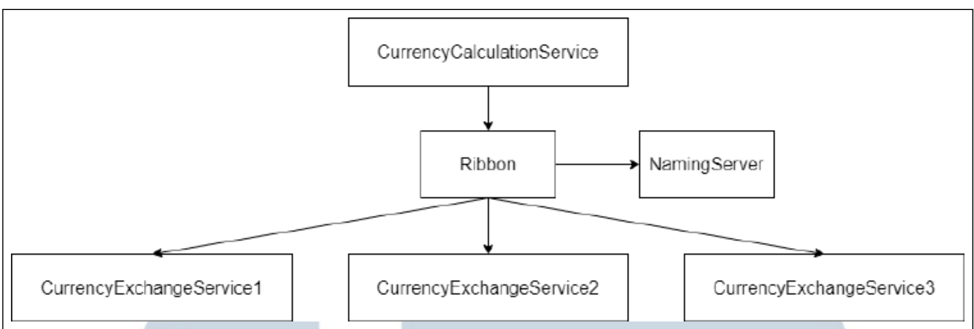
1. Setiap *service* dalam *microservices* dapat memiliki berbagai *instance* yang terdapat dalam berbagai *environment*. Kondisi ini tentu memerlukan berbagai konfigurasi dan menjaga nya agar dapat berjalan dengan baik. Solusi yang dapat dilakukan adalah dengan menggunakan Spring Cloud Config Server.

2. Memerlukan kemampuan untuk dapat membuat *instance* baru secara dinamis dan mendistribusikan beban ke dalam *instance* yang baru. Solusinya adalah dengan menggunakan *Naming Server* (Eureka) dan *Load Balancer* (Ribbon).

3. Perlunya sebuah visibilitas yang jelas tentang apa yang sedang terjadi pada setiap *service*. Misalnya, perlu untuk mengetahui *service* mana yang sedang *down* atau kehabisan ruang dalam tempat penyimpanan. Solusinya adalah dengan menggunakan Zipkin dan Netflix API Gateway

4. Cara untuk mencegah agar sebuah *service* tidak menyebabkan seluruh aplikasi *down*, cara untuk membuat sebuah *fault tolerance* ke dalam *microservices*. Solusinya adalah dengan menggunakan Hystrix *fault tolerance*.

Terdapat tiga buah komponen solusi Spring Boot yang akan diimplementasikan, yaitu Eureka *Naming Server*, Ribbon, dan *API Gateway*. Untuk mengimplementasikan ketiga komponen tersebut dibuat sebuah aplikasi yang memiliki fungsi untuk melakukan pertukaran antar mata uang. Terdapat dua service utama di dalamnya, yaitu *currency exchange service* dan *currency converter service*. Arsitektur untuk aplikasi ini dapat dilihat pada Gambar 3.24.



Gambar 3.24. Arsitektur *Microservices Currency Exchange*

B. Hasil Implementasi

Currency exchange service merupakan *service* yang memiliki fungsi untuk menampilkan nilai tukar antar mata uang. *Service* ini dapat memberikan informasi terkait dengan nilai tukar mata uang untuk dilakukan perhitungan oleh *currency conversion service*. Beberapa data yang dimuat di dalam *currency exchange service*, antara lain asal mata uang, tujuan mata uang, dan nilai tukar antara kedua mata uang. Seluruh data di muat dalam *database* dan direpresentasikan di dalam *domain model*. *Domain model* dilambangkan dengan anotasi @Entity dan harus memiliki sebuah *primary key*, *field*, *constructor*, serta *getter/setter*. Implementasi *domain model* dapat dilihat seperti Gambar 3.25.

```
@Entity
public class ExchangeValue {

    @Id
    private Long id;

    @Column(name="currency_from")
    private String from;

    @Column(name="currency_to")
    private String to;

    private BigDecimal conversionMultiple;
    private int port;

    public ExchangeValue() {

    }


    public ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {
        super();
        this.id = id;
        this.from = from;
        this.to = to;
        this.conversionMultiple = conversionMultiple;
    }

    public Long getId() {
        return id;
    }
```

Gambar 3.25. Potongan Kode *Currency Exchange Model*

Setelah *domain model* dibuat, langkah selanjutnya adalah menambahkan *repository*. *Repository interface* merupakan sebuah kelas abstrak yang memiliki fungsi untuk melakukan operasi CRUD pada *database*. Pada implementasi *currency exchange service* menggunakan jenis JPA Repository. Untuk membuat *repository*, dapat menambahkan *library* JpaRepository yang terdiri dari implementasi berbagai fungsi, metode, dan tipe data dependen. Setelah *interface* dibuat, maka fungsi seperti "save()", "count()", "info()", "findAll()", "sort()" dan lainnya daapat digunakan untuk melakukan *query* data atau manipulasi data yang diperlukan. Implementasi JPA *repository* dapat dilihat pada Gambar 3.26.

```
public interface ExchangeValueRepository extends
        JpaRepository<ExchangeValue, Long>{
    ExchangeValue findByFromAndTo(String from, String to);
}
```

Gambar 3.26. Potongan Kode *Currency Exchange Repository*

Pada konsep arsitektur *microservices* umumnya terdapat pemisahan antara *application logic* dan *business logic*. Namun, pada implementasi aplikasi *currency exchange* semuanya disatukan di dalam *controller* dikarenakan *endpoint* yang sedikit. *Controller* pada Spring Boot ditandai dengan anotasi @RestController. Terdapat satu buah fungsi pada *controller currency exchange* yang memiliki tugas untuk memetakan *request* yang masuk ditandai dengan anotasi @GetMapping. Anotasi mapping tersebut sudah termasuk dengan *request method* sehingga tidak perlu didefinisikan lagi. Untuk operasi yang berhubungan dengan *database* dapat memanggil *object repository* beserta dengan method nya. Potongan kode *controller* serta output dari *currency exchange service* dapat dilihat pada Gambar 3.27 dan 3.28.

UNIVERSITAS MULTIMEDIA NUSANTARA

```java
@RestController
public class CurrencyExchangeController {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private Environment environment;

    @Autowired
    private ExchangeValueRepository repository;

    @GetMapping("/currency-exchange/from/{from}/to/{to}")
    public ExchangeValue retrieveExchangeValue
        (@PathVariable String from, @PathVariable String to){

        ExchangeValue exchangeValue =
                repository.findByFromAndTo(from, to);

        exchangeValue.setPort(
                Integer.parseInt(environment.getProperty("local.server.port")));

        logger.info("{}", exchangeValue);

        return exchangeValue;
    }
}
```

Gambar 3.27. Potongan Kode *Currency Exchange Controller*

```
localhost:8000/currency-exchange/from/EUR/to/INR

{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "port": 8000
}
```

Gambar 3.28. Contoh *Output Currency Exchange*

Currency conversion service adalah *consumer* dari *currency exchange service*. Untuk melakukan proses perhitungan di dalamnya, *currency conversion service* akan memanggil *currency exchange service* untuk mendapatkan informasi terkait nilai tukar mata uang. Pada *currency conversion service* tidak menggunakan *database* untuk penyimpanan data sehingga *domain model* tidak diperlukan. Sebagai gantinya, untuk dapat menyimpan dan menampilkan data digunakan *bean*. *Bean* memiliki fungsi seperti *class* pada OOP yang di dalamnya terdapat *constructor* serta *getter/setter method*. Potongan kode implementasi *currency converrsion service bean* dapat dilihat pada Gambar 3.29.

```java
public class CurrencyConversionBean {
    private Long id;
    private String from;
    private String to;
    private BigDecimal conversionMultiple;
    private BigDecimal quantity;
    private BigDecimal totalCalculatedAmount;
    private int port;

    public CurrencyConversionBean() {

    }

    public CurrencyConversionBean(Long id, String from, String to, BigDecimal conversionMultiple, BigDecimal quantity,
            BigDecimal totalCalculatedAmount, int port) {
        super();
        this.id = id;
        this.from = from;
        this.to = to;
        this.conversionMultiple = conversionMultiple;
        this.quantity = quantity;
        this.totalCalculatedAmount = totalCalculatedAmount;
        this.port = port;
    }
```

Gambar 3.29. Potongan Kode *Currency Conversion Bean*

Pada bagian *controller*, *currency conversion service* tidak mengambil data secara langsung ke *database* melainkan memanggil *currency exchange service* untuk mendapatkan data nilai tukar mata uang. Terdapat dua cara untuk mendapatkan data dari *currency exchange service*, yaitu menggunakan *response entity* dan *proxy*. Penggunaan *proxy* memiliki keunggulan dari segi efisiensi karena untuk memanggil *service* tidak perlu menulis kode berulang kali, hanya perlu memanggil *proxy* saja. Implementasi kedua cara pemanggilan service serta *output* dari *currency conversion* dapat dilihat pada Gambar 3.30 dan Gambar 3.31.

```java
@RestController
public class CurrencyConversionController {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private CurrencyExchangeServiceProxy proxy;

    @GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")
    public CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to,
            @PathVariable BigDecimal quantity) {

        // Feign - Problem 1
        Map<String, String> uriVariables = new HashMap<>();
        uriVariables.put("from", from);
        uriVariables.put("to", to);

        ResponseEntity<CurrencyConversionBean> responseEntity = new RestTemplate().getForEntity(
                "http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.class,
                uriVariables);

        CurrencyConversionBean response = responseEntity.getBody();

        return new CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity,
                quantity.multiply(response.getConversionMultiple()), response.getPort());
    }

    @GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}")
    public CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to,
            @PathVariable BigDecimal quantity) {

        CurrencyConversionBean response = proxy.retrieveExchangeValue(from, to);

        logger.info("{}", response);

        return new CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity,
                quantity.multiply(response.getConversionMultiple()), response.getPort());
    }

}
```

Gambar 3.30. Potongan Kode *Currency Conversion Controller*

```
localhost:8100/currency-converter/from/USD/to/INR/quantity/10

{
    "id": 10001,
    "from": "USD",
    "to": "INR",
    "conversionMultiple": 65,
    "quantity": 10,
    "totalCalculatedAmount": 650,
    "port": 8000
}
```

Gambar 3.31. Contoh *Output Currency Conversion*

Pada *currency conversion service* juga ditambahkan Ribbon sebagai *load balancer*. Hal ini memungkinkan setiap panggilan dari *currency conversion service* akan terdistribusi secara otomatis ke dalam setiap *instance currency exchange service* yang aktif. Cara menambahkan Ribbon ke dalam aplikasi adalah dengan menambahkan *dependency* pada *file* pom.xml seperti Gambar 3.32. Untuk versi Spring Boot terbaru, dapat menggunakan Spring Cloud Load Balancer sebagai pengganti Ribbon.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Gambar 3.32. Ribbon *Dependency*

Eureka *server* adalah aplikasi yang menyimpan informasi untuk semua *service* dalam arsitektur *microservices*.[8] Ketika sebuah *service* ingin melakukan panggilan ke *service* lain, *service* tersebut perlu mengetahui alamat dan *port* dari *service* yang dituju. Cara konvensional yang dapat dilakukan adalah dengan mendaftarkan semua alamat dari *service* lain melalui *file application properties*. Hal tersebut tentunya tidak efektif meningat dalam sebuah perusahaan bisa terdapat banyak sekali *service*. Dengan menggunakan Eureka *server*, sebuah *service* hanya perlu bertanya untuk dapat mengetahui alamat dan *port service* lain. Implementasi Eureka *server* dapat dilakukan dengan membuat sebuah aplikasi Spring Boot dan menambahkan *dependency* Eureka *server* seperti Gambar 3.33.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Gambar 3.33. Eureka *Server Dependency*

Komponen terakhir yang terdapat dalam arsitektur *microservices* adalah API *gateway*. Perubahan *traffic* baik itu dari atau menuju API terkadang menyebabkan permasalahan tanpa adanya pemberitahuan terlebih dahulu. Oleh karena itu, dibutuhkan sebuah sistem yang dapat cepat beradaptasi dengan situasi tersebut. API *gateway* menjadi jawaban dari permasalahan tersebut dengan kemampuan untuk menangani semua *request* yang masuk serta melakukan *dynamic routing*. Implementasi API *gateway* sama seperti Eureka *server* yang hanya perlu menambahkan *dependency* ke dalam *file* pom.xml. *Dependency* untuk menambahkan Zuul Gateway dalam aplikasi dapat dilihat pada Gambar 3.34.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

Gambar 3.34. Zuul *Gateway Dependency*

Hal yang tak kalah penting dalam penerapan *microservices* adalah standarisasi *port*. Standarisasi *port* dibutuhkan karena setiap *service* dapat memiliki lebih dari satu *instance* sehingga perlu ada pola yang sama antar *instance* di sebuah *service*. Dalam penerapan standarisasi *port*, terdapat dua buah *port* yang secara khusus hanya digunakan untuk API *gateway* dan *naming server*, yaitu *port* 8765 dan 8761. Contoh dari penerapan standarisasi *port* dapat dilihat seperti Gambar 3.35.

Application	Port
Limits Service	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Service	8000, 8001, 8002, ..
Currency Conversion Service	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
Netflix Zuul API Gateway Server	8765
Zipkin Distributed Tracing Server	9411

Gambar 3.35. Standarisasi *Port*

3.3.3 Rekrutmen Eksternal

A. Cakupan Pengerjaan

Aplikasi rekrutmen eksternal merupakan sebuah aplikasi yang memiliki fungsi untuk menampilkan informasi dan mengajukan lamaran terkait dengan lowongan kerja yang sedang dibuka di PT Bank Negara Indonesia. Selain itu, pengguna juga dapat memantau perkembangan dari proses rekrutmen yang sedang diikuti. Penamaan rekrutmen eksternal memiliki arti *platform* atau aplikasi ini ditujukkan hanya untuk pelamar dari luar perusahaan. Sedangkan terdapat aplikasi lain, yaitu rekrutmen internal yang digunakan khusus untuk orang yang sudah bekerja di dalam perusahaan, misalnya karyawan kontrak atau *outsource*.

Proses *enhancement* aplikasi rekrutmen eksternal dibagi ke dalam dua tahap. Untuk tahap pertama dikerjakan pada Q1 dan Q2 tahun 2022, sedangkan untuk tahap kedua dikerjakan pada Q3 tahun 2022. Cakupan pengerjaan untuk *enhancement* tahap pertama, antara lain pembaruan tampilan lama ke tampilan baru, penambahan CRUD agar informasi yang ditampilkan lebih dinamis, dan perbaikan alur rekrutmen pada aplikasi. Sedangkan untuk tahap kedua akan dilakukan pembuatan tampilan responsif dan integrasi dengan aplikasi rekrutmen internal.

Pembagian kerja dilakukan di dalam tim beranggotakan 6 orang yang difokuskan untuk tugas *enhancement* ini. Khusus untuk kegiatan magang, terdapat dua tugas yang diberikan, yaitu *slicing* UI/UX dengan mengggunakan HTML dan CSS serta penambahan *backend* dan tampilan untuk CRUD *Content Management System* (CMS). Pada tugas *slicing* UI/UX bagian yang dikerjakan hanya untuk halaman *application history*, sedangkan untuk *backend* CRUD bagian yang dikerjakan adalah konten halaman *sign up*, *job posting*, dan testimoni.

B. Hasil Implementasi

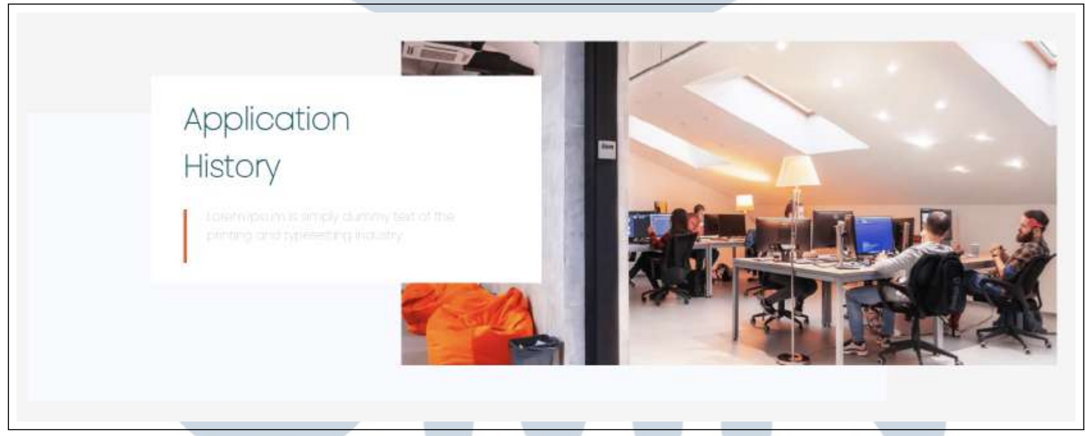
Halaman *application history* memiliki fungsi untuk melacak perkembangan dari tahap rekrutmen yang sedang diikuti oleh pelamar. Pada bagian atas dari *application history* terdapat *hero section* atau *headline* yang terdiri dari 3 elemen yang memiliki posisi saling bertumpukan. Ketiga elemen tersebut, dibagi ke dalam dua bagian div. Bagian div pertama berisi elemen kotak abu abu yang letaknya di paling belakang serta gambar di atasnya, sedangkan untuk div kedua berisi kotak yang menampung judul halaman dan deskripsi.

Penyusunan tampilan *hero section* agar sesuai dengan desain, menggunakan

property positon dari CSS. Terdapat dua jenis *position* yang digunakan, yaitu *relative* dan *absolute*. Nilai *absolute* digunakan agar suatu elemen bisa dipindahkan dan membuat elemen mengambang. Perbedaan dengan *property relative*, pada *property absolute layer* di bawahnya akan bergeser seolah olah kedua elemen saling tumpang tindih. Potongan kode dan tampilan dari implementasi *hero section* dapat dilihat seperti pada Gambar 3.36 dan Gambar 3.37.

```
<div class="Rectangle">
    <img src="assets/img/img.png" srcset="assets/img/img@2x.png 2x, assets/img/img@3x.png 3x" class="img_">
</div>
<div class="Rectangle_in">
    <span class="app_history_title">
        Application History
    </span>
    <div class="inline">
        <div class="rectangle_bar"></div>
        <span class="Lorem-Ipsum-is-simpl">
            Lorem Ipsum is simply dummy text of the printing and typesetting industry.
        </span>
    </div>
</div>
```

Gambar 3.36. Potongan Kode HTML *Hero Section*



Gambar 3.37. Tampilan *Hero Section*

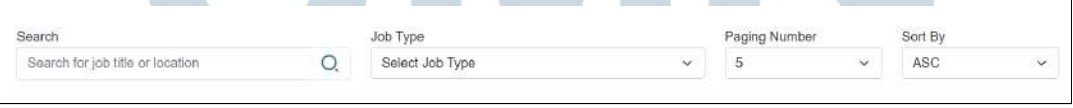
Komponen berikutnya yang terdapat di dalam halaman *application history* adalah *search bar* dan *filter*. Fungsi dari komponen ini adalah untuk mencari *history* dari lamaran melakukan *filter* berdasarkan tipe pekerjaan jumlah data untuk setiap halaman dan *sorting* berdasarkan nama. Masing masing fungsi, merupakan elemen *input* yang berada di dalam sebuah form. Untuk *search bar* menggunakan *input* dengan jenis *text* sedangkan yang lainnya menggunakan *input* dengan jenis *form select*. Agar setiap elemen berada dalam satu baris yang sama, digunakan fitur *grid system* dari Bootstrap dengan menggunakan row dan col. Potongan kode

UNIVERSITAS
MULTIMEDIA
NUSANTARA

dan tampilan dari komponen *search* dan *filter* dapat dilihat pada Gambar 3.38 dan Gambar 3.39.

```html
<li class="list-group-item">
    <form class="row align-items-center">
        <div class="col-4">
            <label for="autoSizingInput">Search</label>
            <div class="input-group">
                <input type="text" class="form-control border-end-0" placeholder="Search for job title or location">
                <div class="input-group-append">
                    <button class="btn btn-outline-secondary bg-white border-start-0 border ms-n5" type="button">
                        <i><img src="assets/img/ic-search.png" alt=""></i>
                    </button>
                  <!-- <span class="input-group-text" id="basic-addon2"></span> -->
                </div>
            </div>
        </div>
        <div class="col">
            <label for="autoSizingSelect">Job Type</label>
            <select class="form-select" id="autoSizingSelect">
                    <option selected>Select Job Type</option>
                    <option value="1">One</option>
                    <option value="2">Two</option>
                    <option value="3">Three</option>
            </select>
        </div>
        <div class="col-2">
            <label for="autoSizingSelect">Paging Number</label>
            <select class="form-select" id="autoSizingSelect">
                <option selected>5</option>
                <option value="1">1</option>
                <option value="2">2</option>
                <option value="3">T3</option>
            </select>
        </div>
        <div class="col-2">
            <label for="autoSizingSelect">Sort By</label>
            <select class="form-select" id="autoSizingSelect">
                    <option selected>ASC</option>
                    <option value="1">DESC</option>
            </select>
        </div>
    </form>
</li>
```

Gambar 3.38. Potongan Kode HTML *Search* dan *Filter*

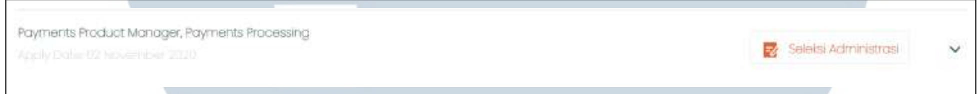


Gambar 3.39. Tampilan *Search* dan *Filter*

Tepat di bawah komponen *search* dan *filter*, terdapat kumpulan dari *history* lamaran yang pernah atau sedang diajukan oleh pengguna. Informasi yang ditampilkan pada komponen ini adalah nama pekerjaan, tanggal pengajuan lamaran, dan status lamaran. Pengaturan tampilan masih sama seperti *search* dan *filter*, yaitu menggunakan *grid system* dari Bootstrap. Potongan kode dan hasil implementasi untuk tampilan *history* lamaran dapat dilihat pada Gambar 3.40 dan Gambar 3.41.

```html
<li class="list-group-item">
    <div class="row">
        <div class="col">
            <h6 class="heading">
                Payments Product Manager, Payments Processing
            </h6>
            <span class="apply-date">
                Apply Date: 02 November 2020
            </span>
        </div>
        <div class="col">
            <div class="status-rectangle">
                <i><img src="assets/img/ic-seleksi.png"></i><span class="status"> Seleksi Administrasi</span>
            </div>
        </div>
        <div class="col-1" style="width: 50px;">
            <a data-bs-toggle="collapse" href="#collapseArea" role="button" aria-expanded="false" aria-controls="collapseArea">
                <i data-feather="chevron-down" class="ex-button"></i>
            </a>
        </div>
    </div>
```

Gambar 3.40. Potongan Kode HTML *Card History*



Gambar 3.41. Tampilan *Card History*

Pada setiap daftar *history* lamaran, terdapat tombol untuk membuka detail status dan nilai dari setiap tahap lamaran. Komponen detail status menggunakan *dropdown* atau *class collapse* yang sudah disediakan oleh Bootstrap. Terdapat dua macam tampilan untuk detail status ini, yaitu tampilan dengan warna oranye jika tahap rekrutmen dinyatakan lulus dan tampilan dengan warna abu abu jika tahap rekrutmen tidak lulus atau belum dilaksanakan. Masing masing tahapan dibungkus dengan komponen *card* yang didalamnya juga menggunakan *grid system*. Potongan kode dan tampilan dari *drowpdown card* dapat dilihat pada Gambar 3.42 dan Gambar 3.43.

```html
<div class="collapse" id="collapseArea" style="margin-top: 20px ;">
    <div class="card card-body">
        <div class="row">
            <div class="col-1">
                <img src="assets/img/ic-disable-seleksi-administrasi.png" class="disable">
            </div>
            <div class="col">
                <h6 class="heading">
                    Seleksi Administrasi
                </h6>
                <span class="apply-date">
                    02 November 2020
                </span>
            </div>
            <div class="col">
                <div class="score-rectangle-disable">
                    <span class="score-disable">0/100</span>
                </div>
            </div>
        </div>
    </div>
```

Gambar 3.42. Potongan Kode HTML *Dropdown Card*

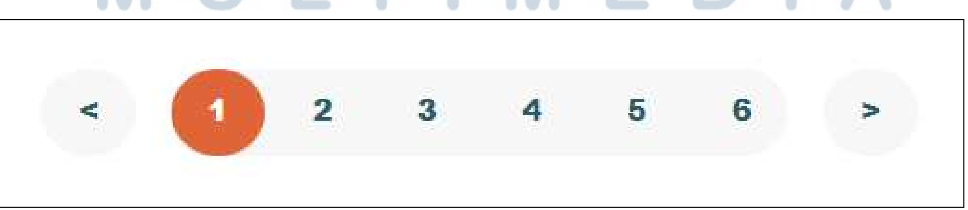
Gambar 3.43. Tampilan *Dropdown Card*

Komponen terakhir yang terdapat pada halaman *application history* adalah *pagination*. *Pagination* adalah sebuah proses untuk membagi data atau informasi yang ditampilkan ke dalam beberapa halaman. Pada halaman *application history* ini informasi yang dibagi adalah daftar *history* lamaran. Setiap elemen di dalam komponen *pagination* dibungkus dengan sebuah tag nav. Karena perpindahan halaman menggunakan *link*, maka digunakan elemen *anchor* sebagai tombol navigasi. Hasil implementasi *pagination* dapat dilihat pada Gambar 3.45.

```
<li style="list-style: none;">
    <nav class="pagination-container">
        <div class="pagination">
                <a class="pagination-newer" href="#"><</a>
                <span class="pagination-inner">
                    <a href="#">1</a>
                    <a class="pagination-active" href="#">2</a>
                    <a href="#">3</a>
                    <a href="#">4</a>
                    <a href="#">5</a>
                    <a href="#">6</a>
                </span>
                <a class="pagination-older" href="#">></a>
        </div>
    </nav>
</li>
```

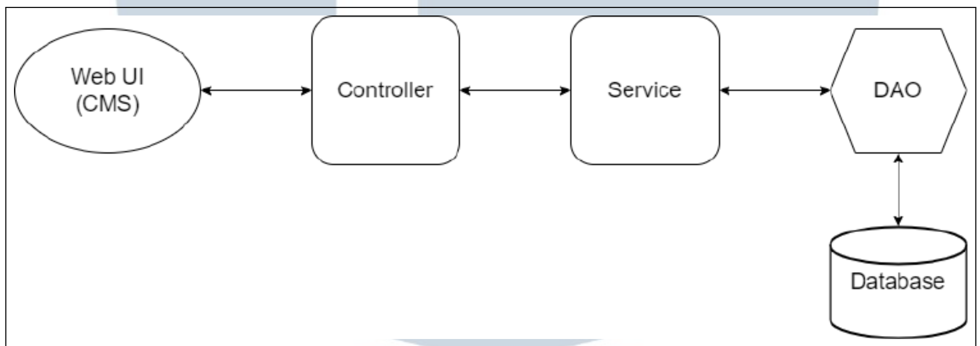
UNIVERSITAS MULTIMEDIA

Gambar 3.44. Potongan Kode HTML *Pagination*



Gambar 3.45. Tampilan *Pagination*

Bagian kedua dari penugasan *enhancement* aplikasi rekrutmen eksternal adalah melakukan penambahan CRUD untuk konten yang akan ditampilkan. Komponen yang terdapat pada backend, antara lain *controller*, *service*, dan DAO (*Data Access Object*). Pengguna dapat melakukan operasi CRUD melalui aplikasi web, kemudian *request* akan dikirimkan ke *controller*. Dari *controller*, *request* yang masuk akan dipetakan dan diteruskan ke *service* sesuai dengan *method* dan *path* yang dituju. Seluruh proses bisnis dari *request* yang masuk akan diolah di dalam *service*. Sedangkan jika *service* membutuhkan operasi data dari dan ke *database*, seluruh prosesnya akan ditangani oleh DAO (*Data Access Object*). Ilustrasi alur *backend* dari CRUD aplikasi rekrutmen eksternal dapat dilihat pada Gambar 3.46



Gambar 3.46. Alur *Backend* CMS

Model domain adalah representasi dari tipe penyimpanan data yang dibutuhkan oleh logika bisnis.[9] Di dalamnya terdapat berbagai objek *domain* (entitas), atribut, peran, relasi di antaranya, serta *constraint* atau batasan. Pembuatan *domain* ditandai dengan anotasi @Entity serta nama tabel yang terdapat dalam *database*. Setiap *domain* atau entitas harus memiliki *primary key* yang unik sebagai identitas. Untuk mendefinisikan *primary key*, dapat menggunakan anotasi @Id. Selain itu terdapat juga anotasi @Column yang menandakan *field* dalam tabel. Terdapat beberapa elemen atau *constraint* yang dapat ditambahkan untuk setiap *column* dalam table, seperti *name*, *length*, *nullable*, dan *unique*. Contoh implementasi dari *domain* dan tabel di *database* dapat dilihat pada Gambar 3.47 dan Gambar 3.48.

```java
@Entity
@Table(name = "[illegible]")
@Where(clause = "DELETED_STATUS = 0")
@XmlRootElement
public class AppMstOurEnvironment extends TrackedEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    2 usages
    @Id
    @TableGenerator(
            name = '[illegible]',
            table = '[illegible]',
            pkColumnName = "seq_name",
            valueColumnName = "seq_value",
            allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "[illegible]")
    @Basic(optional = false)
    @NotNull
    @Column(name = "OUR_ENVIRONMENT_ID")
    private BigDecimal ourEnvironmentId;
```

Gambar 3.47. Potongan Kode *Domain*

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFA...	COLUMN_ID	COMMENTS
1	OUR_ENVIRONMENT_ID	NUMBER(19,0)	No	(null)	1	(null)
2	TITLE	VARCHAR2(255 BYTE)	Yes	(null)	2	(null)
3	DESCRIPTION	VARCHAR2(2000 BYTE)	Yes	(null)	3	(null)
4	REVIEWER_NAME	VARCHAR2(255 BYTE)	Yes	(null)	4	(null)
5	REVIEWER_JOB	VARCHAR2(255 BYTE)	Yes	(null)	5	(null)
6	FILE_NAME	VARCHAR2(255 BYTE)	Yes	(null)	6	(null)
7	FILE_PATH	VARCHAR2(255 BYTE)	Yes	(null)	7	(null)
8	FILE_GENERATE	VARCHAR2(255 BYTE)	Yes	(null)	8	(null)
9	STATUS	NUMBER(19,0)	Yes	(null)	9	(null)
10	USER_CREATED	NUMBER(19,0)	Yes	(null)	10	(null)
11	DATE_CREATED	DATE	Yes	(null)	11	(null)
12	USER_UPDATED	NUMBER(19,0)	Yes	(null)	12	(null)
13	DATE_UPDATED	DATE	Yes	(null)	13	(null)
14	USER_DELETED	NUMBER(19,0)	Yes	(null)	14	(null)
15	DATE_DELETED	DATE	Yes	(null)	15	(null)
16	DELETED_STATUS	NUMBER(19,0)	Yes	(null)	16	(null)
17	FILE_ID	NUMBER(19,0)	Yes	(null)	17	(null)

Gambar 3.48. Contoh Tabel *Database*

Semua request yang diajukan dari *client* atau aplikasi web akan masuk ke dalam controller. Di dalam *controller* terdapat logika aplikasi yang akan memetakan setiap *request* berdasarkan *endpoint* yang dituju. Jika terdapat kecocokan, maka *controller* akan meneruskan *request* ke *service*. Pada aplikasi rekrutmen eksternal terdapat empat *endpoint* yang merepresentasikan operasi

UNIVERSITAS MULTIMEDIA NUSANTARA

CRUD, yaitu *getAll*, *getById*, *create*, *update*, dan *delete*. Potongan kode controller dapat dilihat pada Gambar 3.49.

```
@RestController
@RequestMapping("our-environment/")
public class OurEnvironmentAPI {
    final static Logger logger = Logger.getLogger(OurEnvironmentAPI.class.getName());

    6 usages
    @Autowired
    OurEnvironmentService ourEnvironmentService;

    Anthony Rafael
    @RequestMapping(value = "pagging-all", method = RequestMethod.POST)
    public Map<String,Object> ourEnvironmentPaggingAll(@RequestBody Map<String, Object> params){
        return ourEnvironmentService.ourEnvironmentServicePaggingAll(params);
    }

    Anthony Rafael
    @RequestMapping(value = "all", method = RequestMethod.POST)
    public Map<String,Object> ourEnvironmentAll(){
        return ourEnvironmentService.ourEnvironmentServiceAll();
    }
```

Gambar 3.49. Potongan Kode *Controller*

DAO atau *Data Access Object* merupakan sebuah konsep desain aplikasi yang digunakan sebagai perantara dengan *database*. Seluruh operasi CRUD dari *database* dapat dilakukan melalui DAO. Penggunaan DAO membuat aplikasi lebih terstruktur dikarenakan program dikelompokkan berdasarkan fungsinya. Terdapat dua bagian DAO dalam implementasi *project* rekrutmen eksternal, yaitu DAO dan DAOImpl. DAO menggunakan *interface* yang mendefinisikan kelas abstrak untuk melakukan operasi CRUD pada suatu *object*. Pada contoh Gambar 3.50 *object* yang dimaksud adalah AppMstOurEnvironment yang sudah didefinisikan pada bagian *domain*. Sedangkan DAOImpl sesuai namanya merupakan implementasi dari kelas abstrak DAO. Contoh kelas implementasi DAO dapat dilihat pada Gambar 3.51.

```
public interface OurEnvironmentDao extends IGenericDao<AppMstOurEnvironment, BigDecimal>{

    /rawtypes/
    public Class getFieldType(String fieldName);

}
```

Gambar 3.50. Potongan Kode *Data Access Object*

```
@Repository
public class OurEnvironmentDaoImpl extends AbstractHibernateDao<AppMstOurEnvironment, BigDecimal> implements OurEnvironmentDao {

    1 usage
    private static final Logger LOGGER = Logger.getLogger(PortalJoinDaoImpl.class);

    /rawtypes/
    @Override
    public Class getFieldType(String fieldName) {
        try {
            Field field = AppMstPortalJoin.class.getDeclaredField(fieldName);
            if (field != null) {
                return field.getType();
            }
        } catch (NoSuchFieldException | SecurityException ex) {
            LOGGER.error(ex);
        }
        return null;
    }

}
```

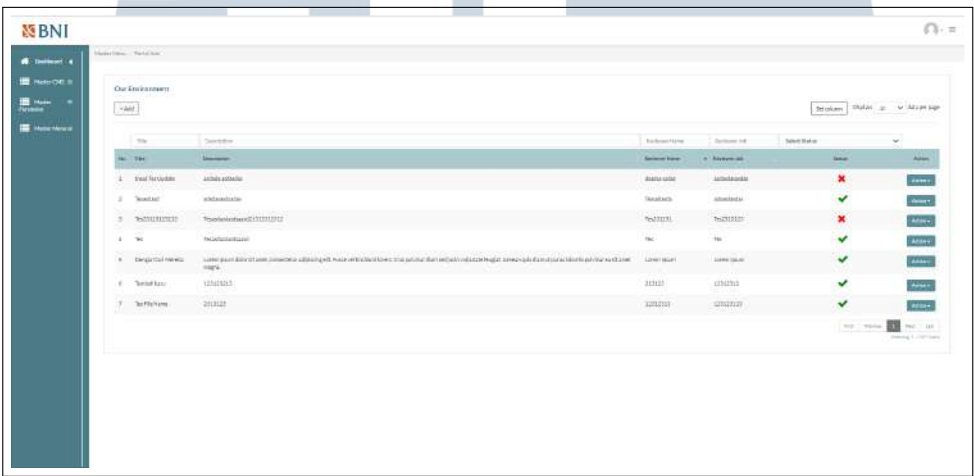
Gambar 3.51. Potongan Kode DAO Impl

Komponen terakhir yang terdapat pada *backend* aplikasi rekrutmen eksternal adalah *service*. *Service* merupakan sebuah *class* yang di dalamnya berisi logika bisnis suatu aplikasi.[10] Logika bisnis tersebut diimplementasikan pada fungsi sesuai dengan kebutuhan. Misalnya untuk aplikasi rekrutmen eksternal dibutuhkan fungsi CRUD maka terdapat empat fungsi untuk melakukan pembacaan, penambahan, pembaruan, dan penghapusan data. Setiap operasi yang berhubungan dengan *database* memerlukan operator DAO. Oleh karena itu, pada bagian awal *service* harus didefinisikan sebuah *object* DAO. Contoh potongan kode implementasi dari *service* dapat dilihat pada Gambar 3.52.

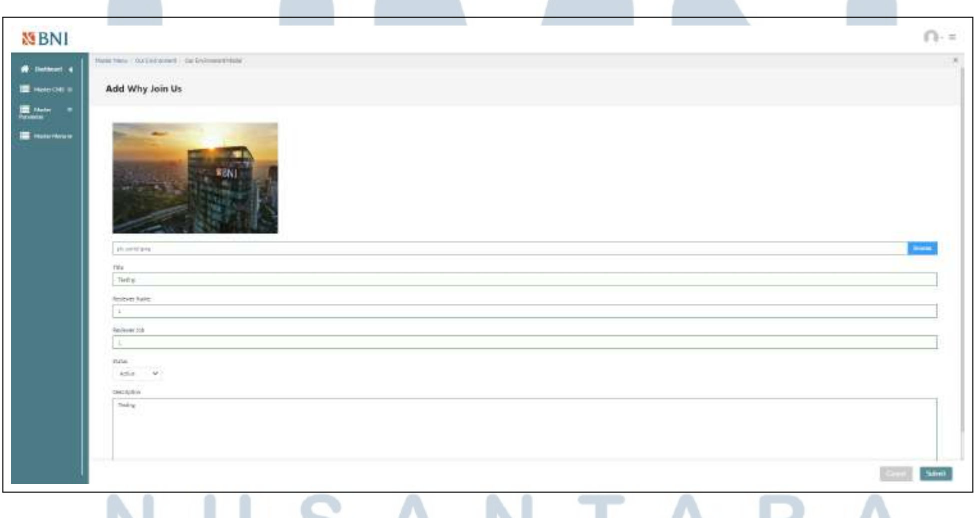
```
@Component
public class OurEnvironmentService implements Serializable {
    private static final long serialVersionUID = 1L;
    2 usages
    private static final Logger LOGGER = Logger.getLogger(OurEnvironmentService.class);
    2 usages
    private static final Short SHORT_ZERO = Short.valueOf( s: "0");
    1 usage
    private static final Short SHORT_ONE = Short.valueOf( s: "1");

    10 usages
    @Autowired
    OurEnvironmentDao ourEnvironmentDao;

    1 usage   new *
    public Map<String,Object>  ourEnvironmentServiceAll(){
        Map<String,Object> response = new HashMap<~>();
        try {
            response = fillSuccessResponse();
            response.put("data", ourEnvironmentDao.findAllActive());
        }catch (Exception e) {
            e.printStackTrace();
            response = fillErrorResponse();
        }
        return response;
    }
```

Gambar 3.52. Potongan Kode *Service*

Bagian *frontend* dari sistem CRUD aplikasi rekrutmen eksternal disebut dengan CMS (*Content Management System*). Tampilan *frontend* dibuat dengan menggunakan JSP (*Java Server Page*) yang merupakan teknologi untuk membuat tampilan HTML menjadi dinamis. Terdapat dua bagian tampilan JSP pada CMS, yaitu *page* dan *partial page*. *Page* merupakan tampilan utama dari halaman CMS sedangkan *partial page* berisi modal yang akan menampilkan tabel atau form sesuai dengan menu yang dipilih. Contoh tampilan dari halaman CMS rekrutmen eksternal dapat dilihat pada Gambar 3.53 dan Gambar 3.54.



Gambar 3.53. Contoh Tampilan CMS



Gambar 3.54. Contoh Tampilan Form

3.4 Kendala dan Solusi yang Ditemukan

Kendala yang dihadapi selama melaksanakan kegiatan magang di PT Bank Negara Indonesia Tbk, yaitu :

1. Ketika melakukan *clone* suatu *project* terkadang IDE (*Integrated Development Environment*) tidak dapat melakukan *import* maven secara sempurna. Hal ini seringkali terjadi khususnya ketika menggunakan Intellij sebagai IDE.

2. Skala aplikasi yang terlalu besar ditambah waktu untuk mempelajari arsitektur yang cukup singkat menyebabkan cukup sulit untuk memahami alur secara keseluruhan.

3. Peserta magang tidak diberikan akses secara penuh untuk dapat mengakses *repository internal* dan jaringan intranet.

Solusi dari kendala yang dihadapi selama melaksanakan kegiatan magang di PT Bank Negara Indonesia Tbk, yaitu :

1. Mengunduh dan melakukan instalasi IDE alternatif, yaitu Eclipse sehingga jika terjadi masalah pada salah satu IDE dapat dicoba di IDE lain.

2. Banyak bertanya kepada pembimbing atau karyawan senior apabila mengalami kesulitan dalam menemukan alur suatu aplikasi.

3. Meminjam akun karyawan yang memiliki akses setiap kali akan melakukan *clone* atau mengakses *online repository*.

UMN
UNIVERSITAS MULTIMEDIA NUSANTARA