# BAB 3
# PELAKSANAAN KERJA MAGANG

## 3.1 Kedudukan dan Koordinasi

Selama periode magang, setiap tugas dan proyek dilaksanakan dengan berfokus pada pemahaman sistem yang digunakan, pengembangan fitur, serta pelaksanaan pengujian dan *debugging* untuk memastikan kualitas hasil kerja. Koordinasi dilakukan secara rutin bersama Bpk. Darul Mukminin selaku *Java team lead* dan pembimbing teknis yang memberikan arahan serta *feedback* untuk menjaga kesesuaian antara hasil pekerjaan dan standar yang ditetapkan.

Seluruh kegiatan terdokumentasi secara sistematis sebagai bagian dari proses evaluasi berkala. Dalam evaluasi tersebut, *team lead* memberikan penilaian, saran, dan rekomendasi untuk meningkatkan kemampuan teknis serta mutu hasil kerja.

## 3.2 Tugas yang Dilakukan

Salah satu tugas yang dilakukan adalah merancang dan mengimplementasikan sistem *logging* pada aplikasi E-Signing. Sistem ini berfungsi untuk merekam seluruh aktivitas pengguna selama menggunakan aplikasi, seperti proses autentikasi, pengunggahan dokumen, hingga penandatanganan digital. Dalam implementasinya, sistem *logging* dirancang agar mampu menyimpan data secara terstruktur, sehingga setiap aktivitas dapat ditelusuri dengan mudah apabila terjadi kesalahan atau indikasi penyalahgunaan. Selain itu, penerapan sistem ini juga mendukung prinsip transparansi dan akuntabilitas dalam pengelolaan dokumen digital, karena memungkinkan pengembang maupun pihak berwenang untuk melakukan audit terhadap riwayat tindakan yang tercatat.

## 3.3 Uraian Pelaksanaan Magang

Pelaksanaan kerja magang diuraikan seperti pada Tabel 3.1.

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

| Minggu Ke - | Pekerjaan yang dilakukan |
|---|---|
| 1 | Memulai pembuatan *logging system* untuk aplikasi E-Signing: menyiapkan `Request.java`, `RequestParser.Java`, `Response.java` |
| 2 | Membuat *logging system* untuk aplikasi E-Signing: dimulai dari `LogService.java` |
| 3 | Implementasi alur dan logika dari *logging system*: struktur penyimpanan, format penamaan, mekanisme *backup* |
| 5 | Testing dan debugging *logging system* |
| 6 | Revisi alur dan logika dari *logging system*: struktur folder penyimpanan, penamaan *log file* |
| 7 | Testing dan debugging *logging system* |
| 8 | Implementasi alur dan logika dari *logging system*: struktur penyimpanan, format penamaan, mekanisme *backup* |
| 9 | Revisi logika dan metode deteksi pergantian hari untuk *backup log file*: tidak menggunakan *logical threads* dan penyederhanaan *trigger condition* |
| 10 | Penyempurnaan fitur *log rotation* berbasis ukuran: implementasi `rotateActiveLog()`, penanganan *rotation marker*, dan pemindahan file ke direktori `backuplogs` dengan format nama yang ditentukan |
| 11 | Pengembangan sistem penamaan alfabet berurutan untuk file cadangan: implementasi `incrementAlphabet()` dan `compareSuffixes()` yang mendukung pola A–Z, AA–ZZ, dst. |
| 12 | Perbaikan dan optimasi proses pemulihan log yang tertinggal: implementasi `backupLeftoverActiveLogs()` untuk memindahkan *active logs* yang tidak sempat dirotasi saat *shutdown*; penambahan penanda sesi baru pada *file* yang dipindahkan |

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

| Minggu Ke - | Pekerjaan yang dilakukan |
|---|---|
| 13 | Implementasi mekanisme pembersihan otomatis file .lck pada direktori `activelogs` dan `backuplogs`. Implementasi `cleanupLockFiles()` dan `deleteLockFiles()` untuk mencegah konflik *file handler* |
| 14 | Integrasi kontrol sinkronisasi `(synchronized(lock))` pada seluruh proses penulisan log untuk memastikan *thread safety* dan mencegah *race condition* pada rotasi harian serta rotasi ukuran |
| 15 | Pengujian terhadap rotasi harian dan rotasi ukuran: simulasi pergantian hari, *stress-test* penulisan *log*, dan validasi keakuratan penamaan *file* cadangan dan konsistensi *timestamp* |
| 16 | Merapihkan *codebase*: penyederhanaan kode, penyesuaian *formatter log*, peningkatan stabilitas saat inisialisasi ulang `FileHandler`, implementasi `shutdown()` untuk memastikan penutupan *handler* yang aman |

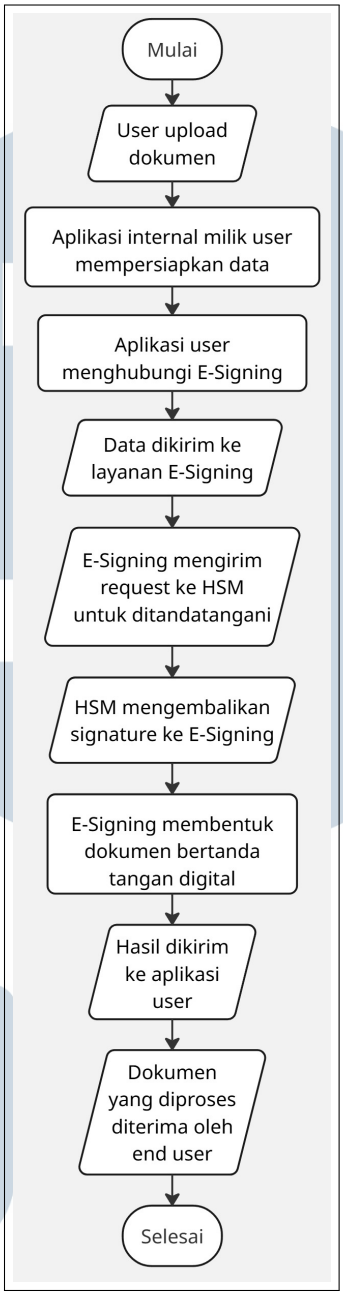
### 3.3.1 E-Signing

E-Signing merupakan sistem penandatanganan dokumen secara *digital* yang berfungsi untuk memverifikasi identitas, mengautentikasi dokumen, dan memfasilitasi transaksi tanpa perlu pertemuan fisik. Pada subbab ini, uraian sistem disajikan sebagai gambaran konseptual tingkat tinggi (*high-level overview*) yang bertujuan untuk memberikan pemahaman terhadap alur kerja secara umum, dan bukan sebagai dokumentasi teknis resmi maupun spesifikasi implementasi sistem. Ilustrasi yang disajikan disusun berdasarkan observasi proses kerja serta penjelasan lisan dari pihak terkait selama pelaksanaan kegiatan magang.

Secara umum, sistem ini dirancang untuk meningkatkan efisiensi dan menjaga keamanan melalui pemanfaatan *Hardware Security Module (HSM)* sebagai pengelola kunci kriptografi, serta mekanisme *logging* untuk mencatat aktivitas pengguna guna mendukung transparansi, akuntabilitas, dan kebutuhan audit.

Secara konseptual, alur proses aplikasi *E-Signing* sebagaimana diilustrasikan pada Gambar 3.1 dapat dijelaskan sebagai berikut:

1. Proses diawali ketika *end user* mengunggah dokumen melalui aplikasi internal yang digunakan dalam lingkungan kerja.
2. Dokumen yang diunggah diproses oleh aplikasi internal sebagai tahap persiapan sebelum dilakukan proses penandatanganan secara *digital*.
3. Sistem AS400, sebagai sistem internal penghubung, membangun koneksi dengan layanan *E-Signing* untuk memungkinkan pertukaran data secara andal.
4. Setelah koneksi terbentuk, sisi AS400 mengirimkan *request* atau *payload* yang memuat data dokumen ke aplikasi *E-Signing*.
5. Aplikasi *E-Signing*, sebagai layanan penandatanganan, meneruskan permintaan tersebut ke *Hardware Security Module (HSM)* melalui antarmuka layanan untuk melakukan proses kriptografi yang diperlukan.
6. HSM melakukan operasi kriptografi menggunakan kunci yang tersimpan secara aman dan mengembalikan hasil tanda tangan digital ke aplikasi *E-Signing*.
7. Aplikasi *E-Signing* menyisipkan nilai tanda tangan digital ke dalam struktur tanda tangan elektronik yang digunakan (misalnya *XML Digital Signature*) sebagai hasil pemrosesan.
8. Struktur tanda tangan yang telah terbentuk kemudian dikirimkan kembali ke sistem AS400 sebagai keluaran proses penandatanganan.
9. Dokumen yang telah ditandatangani secara *digital* selanjutnya diterima oleh *end user* sebagai hasil akhir dari proses *E-Signing*.

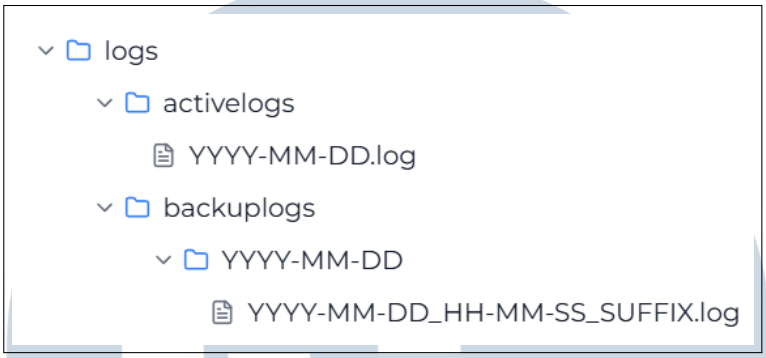



Gambar 3.1. Ilustrasi/contoh konseptual alur aplikasi E-Signing

### 3.3.2 Logging System

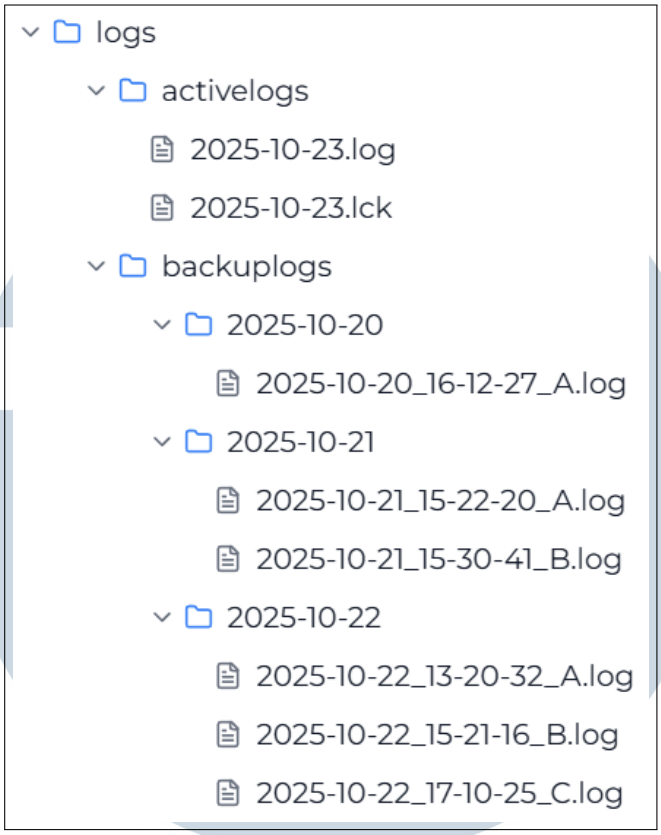
Sistem *logging* pada aplikasi E-Signing berfungsi untuk mencatat seluruh aktivitas yang terjadi selama penggunaan aplikasi, mulai dari koneksi *client* sampai proses verifikasi dan penandatanganan dokumen. Pencatatan ini diimplementasikan melalui `LogService`, yang secara otomatis membuat dan memperbarui catatan aktivitas setiap kali terjadi interaksi pada sistem. Setiap *log file* merupakan *text*

*file* yang disimpan di *local storage* dengan struktur penyimpanan yang terorganisir. Ilustrasi dari struktur penyimpanan ditunjukkan pada Gambar 3.2.




Gambar 3.2. Struktur penyimpanan

Direktori utama dibagi menjadi dua bagian, yaitu *active log* dan *backup log*. Pada masing-masing direktori tersebut, *log file* disimpan dalam folder berdasarkan tanggal pembuatan (per hari) untuk memudahkan proses penelusuran. *File* pada *backup log* menggunakan format penamaan khusus berupa `YYYY-MM-DD_HH-MM-SS_ALPHABET` untuk memastikan kejelasan versi dan mencegah duplikasi. Berikut adalah contoh penyimpanan *log file* pada Gambar 3.3.




Gambar 3.3. Contoh penyimpanan log file

*Logging system* memiliki mekanisme *log rotation* atau *backup* yang dijalankan berdasarkan tiga kondisi utama, yaitu:

- Pertama, apabila ukuran berkas *log* telah mencapai batas maksimum yaitu 10 MB, sistem akan secara otomatis melakukan proses *backup* dan membuat *folder* dan *file* baru.
- Kedua, rotasi dilakukan setiap pergantian hari. Contohnya ketika perubahan dari jam 23.59 ke 00.00, maka *log file* akan di-*backup* walaupun ukuran *file* lebih kecil dari 10 MB.
- Ketiga, pada *edge case*, jika *service*/aplikasi dimatikan, kemudian dijalankan kembali, sistem akan memeriksa keberadaan *log* aktif yang belum di-*backup*, dan memindahkannya ke direktori *backup* sebelum membuat *folder* dan *file* yang baru.

### 3.3.3 Proses dan Alur Logging System

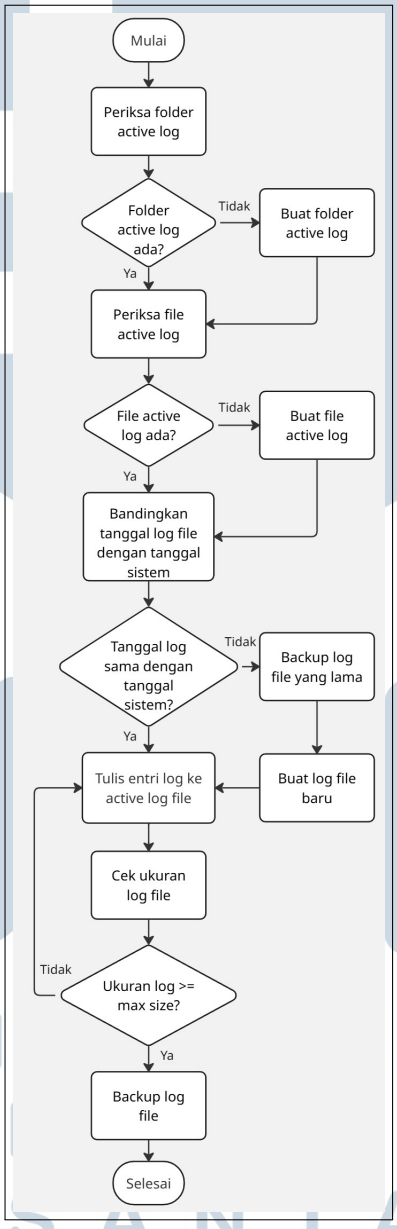
Uraian pada bagian ini bersifat konseptual dan disusun untuk memberikan gambaran umum mengenai alur kerja *logging system*. Penyajian dalam bentuk deskripsi dan *flowchart* tidak merepresentasikan spesifikasi teknis, rancangan arsitektur, maupun dokumentasi resmi sistem. Diagram yang ditampilkan merupakan abstraksi logika proses yang disusun berdasarkan perilaku sistem pada tahap implementasi serta arahan yang diperoleh selama pelaksanaan kerja praktik, tanpa mengubah atau menetapkan desain awal sistem.

Secara umum, proses *logging system* berjalan melalui rangkaian langkah yang bertujuan menjaga konsistensi, ketertiban, dan kesinambungan data jejak sistem. Mekanisme ini memastikan setiap aktivitas dicatat dalam berkas yang sesuai, dipindahkan ketika terjadi pergantian waktu, serta dibackup apabila ukuran berkas melampaui batas yang ditentukan. Dengan pendekatan tersebut, sistem tidak hanya mempertahankan integritas *log*, tetapi juga mencegah penumpukan data yang berpotensi memengaruhi kinerja.

Alur proses *logging system* dijelaskan secara konseptual sebagai berikut. Ilustrasi *flowchart* pada Gambar 3.4 digunakan untuk menggambarkan alur kerja sistem secara umum berdasarkan implementasi yang ada, dan tidak dimaksudkan sebagai spesifikasi teknis maupun rancangan arsitektur sistem.

1. Proses *logging* diawali dengan pemeriksaan keberadaan folder *active log*. Apabila *folder* tersebut belum tersedia, sistem akan membuatnya terlebih dahulu.
2. Setelah itu, sistem memeriksa apakah di dalam folder *active log* telah terdapat *log file* yang aktif.
3. Jika *log file* aktif belum tersedia, sistem membuat berkas *log* baru untuk digunakan sebagai *active log file*.
4. Apabila *log file* aktif sudah tersedia, sistem membandingkan tanggal pada berkas *log* dengan tanggal sistem saat ini.
5. Jika tanggal pada *log file* tidak sama dengan tanggal sistem, maka *log file* lama dibackup dan sistem membuat *log file* baru untuk tanggal berjalan.
6. Setelah *active log file* yang sesuai siap digunakan, sistem menuliskan entri *log* ke dalam berkas tersebut.

7. Selanjutnya, sistem melakukan pemeriksaan terhadap ukuran *log file*.
8. Apabila ukuran *log file* telah mencapai atau melebihi batas maksimum yang ditentukan, sistem melakukan proses *backup* terhadap berkas *log*. Jika ukuran belum mencapai batas, proses *logging* dilanjutkan pada *active log file* yang sama.




Gambar 3.4. Ilustrasi konseptual alur proses *logging system*

Pada saat terjadi pergantian hari, sistem tidak hanya melanjutkan proses pencatatan, tetapi juga melakukan pengelolaan *log file* agar data dari hari yang

berbeda tidak tercampur. Penjelasan berikut menguraikan tahapan proses *logging* yang berlangsung ketika terjadi transisi tanggal dalam sistem.

1. Pada kondisi operasi normal (*running system*), aplikasi secara berkelanjutan melakukan pencatatan data (*ongoing logging*) ke dalam satu *active log file*. Selama tanggal sistem belum berubah, seluruh entri log akan terus ditulis ke berkas yang sama.

2. Skenario pergantian hari ditandai ketika entri log terakhir tercatat pada pukul 23.59.59, yang merepresentasikan batas akhir hari berjalan. Pada tahap ini, sistem masih menyimpan data pada *active log file* yang sedang digunakan.

3. Ketika terdapat entri log baru yang masuk setelah pergantian hari, yaitu pada pukul 00.00.00, sistem melakukan pemeriksaan terhadap tanggal saat ini. Perbedaan tanggal antara entri sebelumnya dan entri yang baru diinterpretasikan sebagai indikator terjadinya pergantian hari.

4. Berdasarkan kondisi tersebut, sistem mengeksekusi proses *backup* terhadap *current log file*. Proses ini bertujuan memisahkan data log hari sebelumnya agar tidak tercampur dengan data log hari yang baru, serta menjaga integritas dan keterlacakan informasi.

5. Setelah proses *backup* selesai, sistem membuat *log file* baru yang dikhususkan untuk tanggal yang baru. Seluruh entri log berikutnya, termasuk entri pada pukul 00.00.00 dan setelahnya, ditulis ke dalam berkas log yang baru.

6. Sistem kemudian kembali ke kondisi *ongoing logging* normal dengan menggunakan *active log file* yang sesuai dengan tanggal berjalan.

Pada kondisi tertentu, pergantian hari dapat terjadi ketika aplikasi atau server tidak sedang berjalan. Dalam skenario ini, sistem tetap harus memastikan bahwa berkas *log* dari hari sebelumnya tidak digunakan kembali pada hari yang baru. Oleh karena itu, pada saat aplikasi dijalankan ulang, sistem melakukan pemeriksaan terhadap *log file* yang tersisa dari sesi sebelumnya sebelum melanjutkan proses pencatatan. Alur proses *logging* pada kondisi *restart* di hari yang berbeda dijelaskan sebagai berikut:

1. Ketika aplikasi dijalankan kembali, sistem terlebih dahulu memeriksa keberadaan *active log file* dari sesi sebelumnya di dalam direktori *logging*.

2. Jika *active log file* ditemukan, sistem melakukan pembacaan metadata atau informasi tanggal yang terkait dengan berkas tersebut.

3. Sistem kemudian membandingkan tanggal *log file* terakhir dengan tanggal sistem saat aplikasi dijalankan.

4. Apabila tanggal pada *log file* berbeda dengan tanggal saat ini, kondisi tersebut diinterpretasikan sebagai pergantian hari yang terjadi selama aplikasi tidak aktif.

5. Berdasarkan hasil pemeriksaan tersebut, sistem mengeksekusi proses *backup* terhadap *active log file* dari hari sebelumnya, sehingga data log lama dipisahkan dan diamankan.

6. Setelah proses *backup* selesai, sistem menghasilkan *log file* baru yang disesuaikan dengan tanggal saat ini.

7. Seluruh proses pencatatan berikutnya kemudian dilanjutkan ke dalam *log file* yang baru, dan sistem kembali beroperasi pada kondisi *ongoing logging* normal.

### 3.3.4 Request Class

Kelas `Request` pada Kode 3.1 digunakan sebagai representasi struktur data permintaan yang dipertukarkan dalam sistem. Struktur dasar kelas ini telah tersedia pada implementasi awal dan mencakup metode utilitas untuk konversi data *digest*. Oleh karena itu, pengembangan pada bagian ini tidak mencakup perancangan arsitektur kelas secara keseluruhan, melainkan difokuskan pada penyempurnaan struktur data dan optimasi pemrosesan.

Kontribusi yang dilakukan pada kelas ini meliputi penataan atribut utama pesan serta penambahan mekanisme penyimpanan sementara (*caching*) untuk hasil konversi *digest*. Secara spesifik, atribut `header`, `trxCode`, `reference`, `digestHex`, `signatureBase64`, dan `flag` digunakan sebagai kontainer langsung bagi elemen pesan yang diterima sistem. Selain itu, variabel `cachedDigestBytes` dan `cachedDigestBase64` ditambahkan untuk menyimpan hasil konversi agar tidak dilakukan perhitungan ulang pada pemanggilan berikutnya.

Metode `getDigestBytes` diimplementasikan sebagai bagian dari kontribusi untuk mengonversi nilai `digestHex` dari representasi heksadesimal ke dalam

bentuk *byte array*. Metode ini menerapkan mekanisme *lazy initialization*, yaitu proses konversi hanya dilakukan ketika nilai pertama kali dibutuhkan dan hasilnya disimpan di dalam cache. Pendekatan ini bertujuan meningkatkan efisiensi pemrosesan tanpa mengubah alur logika utama sistem.

Bagian lain dari kelas, termasuk metode `getDigestBase64` dan metode utilitas `hexStringToByteArray`, merupakan bagian dari implementasi awal sistem yang sudah ada. Kode tersebut ditampilkan untuk memberikan konteks integrasi antara komponen yang dikembangkan dengan struktur kelas yang telah ada.

```
1  public class Request
2  {
3    public String header, trxCode, reference, digestHex,
       signatureBase64, flag;
4
5    private byte[] cachedDigestBytes;
6    private String cachedDigestBase64;
7
8    public byte[] getDigestBytes() {
9      if (cachedDigestBytes == null) {
10       cachedDigestBytes = hexStringToByteArray(digestHex);
11     }
12     return cachedDigestBytes;
13   }
14
15   public String getDigestBase64()
16   {
17     if (cachedDigestBase64 == null) {
18       cachedDigestBase64 = Base64.getEncoder().encodeToString(
       getDigestBytes());
19     }
20     return cachedDigestBase64;
21   }
22
23   private static byte[] hexStringToByteArray(String hex)
24   {
25     int len = hex.length();
26     byte[] data = new byte[len / 2];
27     for (int i = 0; i < len; i += 2)
28   {
29       data[i / 2] = (byte)((Character.digit(hex.charAt(i), 16) <<
       4) + Character.digit(hex.charAt(i + 1), 16));
30     }
```

```
31      return data;
32    }
```

Kode 3.1: Request.java

### 3.3.5 RequestParser Class

Kelas `RequestParser` pada Kode 3.2 berfungsi untuk memecah pesan mentah (*raw message*) menjadi objek `Request` berdasarkan format protokol yang telah ditentukan. Struktur dasar kelas ini—termasuk pemetaan segmen pesan menggunakan indeks karakter tetap serta pembentukan objek `Request`—merupakan bagian dari implementasi awal sistem yang kemudian disempurnakan pada tahap pengembangan lanjutan.

Pengembangan pada kelas ini diawali dengan penyusunan rancangan awal, yang kemudian direfaktor dan distandarisasi oleh pengembang senior sebagai pembimbing teknis pada versi akhir sistem. Kode yang ditampilkan pada laporan merepresentasikan implementasi final yang digunakan dalam sistem, sedangkan kontribusi pada tahap awal difokuskan pada perumusan struktur pemrosesan dan penambahan mekanisme validasi format data sebelum pesan diteruskan ke tahap berikutnya.

Validasi dilakukan terhadap dua komponen utama, yaitu `digestHex` dan `signature`. Metode `isValidHexString` memastikan bahwa nilai `digestHex` memiliki panjang 64 karakter dan hanya terdiri dari karakter heksadesimal yang sah. Sementara itu, metode `isValidBase64String` memverifikasi bahwa nilai tanda tangan digital dapat didekode menggunakan skema *Base64*, sehingga mencegah pemrosesan data yang tidak valid atau rusak.

```
1  public class RequestParser
2  {
3    public static Request parse(String message)
4    {
5      String header = message.substring(0, 4);
6      String trxCode = message.substring(4, 8);
7      String reference = message.substring(8, 43);
8      String digestHex = message.substring(43, 107);
9      String signature = message.substring(107, 619).replaceAll("\\s
       +$", "");
10     String flag = message.substring(619, 620);
11
```

```
12      return new Request(header, trxCode, reference, digestHex,
        signature, flag);
13    }
14
15    private static boolean isValidHexString(String hex)
16    {
17      if (hex == null || hex.length() != 64) {
18        return false;
19      }
20      return hex.matches("[0-9A-Fa-f]+");
21    }
22
23    private static boolean isValidBase64String(String base64)
24    {
25      if (base64 == null || base64.isEmpty()) {
26        return false;
27      }
28      try
29      {
30        Base64.getDecoder().decode(base64);
31        return true;
32      }
33      catch (IllegalArgumentException e) {
34        return false;
35      }
36    }
```

Kode 3.2: RequestParser.java

### 3.3.6 Response Class

Kelas `Response` pada Kode 3.3 digunakan untuk membentuk pesan balasan dalam format *fixed-length* sesuai dengan protokol komunikasi sistem. Struktur kelas ini, termasuk penentuan urutan field dan spesifikasi panjang setiap komponen pesan, telah ditetapkan dalam implementasi awal sistem dan pada tahap akhir direfaktor serta distandarisasi oleh pengembang senior.

Kontribusi pengembangan pada kelas ini dilakukan pada tahap awal, khususnya dalam perapihan struktur atribut dan penyesuaian mekanisme pemformatan agar setiap elemen pesan berada pada posisi (*offset*) yang sesuai dengan spesifikasi protokol. Setiap atribut—`header`, `trxCode`, `reference`, `digestBase64`, `signatureBase64`, dan `flag`—mewakili komponen pesan dengan

panjang tetap. Untuk menjaga kesesuaian format tersebut, digunakan fungsi `padRightWithSpaces()`, yang memastikan nilai *string* tidak melebihi batas panjang dan akan dipenuhi dengan karakter spasi apabila lebih pendek. Pendekatan ini mencegah pergeseran struktur data yang berpotensi menimbulkan kesalahan pada proses pembacaan di sisi penerima.

Metode `serialize()` menyusun seluruh atribut ke dalam satu rangkaian teks sesuai urutan protokol, dengan setiap *field* diproses menggunakan `padRightWithSpaces()` agar panjangnya konsisten dengan spesifikasi. Kode yang ditampilkan pada laporan merepresentasikan implementasi final yang digunakan dalam sistem. Sementara itu, kontribusi pada tahap awal difokuskan pada penyusunan struktur data dan penyesuaian mekanisme pemformatan, yang selanjutnya disempurnakan dan diintegrasikan pada versi akhir oleh pengembang senior.

```
1  public class Response
2  {
3    public String header, trxCode, reference, digestBase64,
       signatureBase64, flag;
4
5    private static String padRightWithSpaces(String input, int
       totalLength)
6    {
7      if (input == null) input = "";
8      if (input.length() >= totalLength) {
9        return input.substring(0, totalLength);
10     }
11     return String.format("%-" + totalLength + "s", input);
12   }
13
14   public String serialize()
15   {
16     StringBuilder response = new StringBuilder();
17     response.append(header);
18     response.append(padRight(trxCode, 4));
19     response.append(padRight(reference, 35));
20     response.append(padRight(digestBase64, 64));
21     response.append(padRight(signatureBase64, 512));
22     response.append(padRight(flag, 1));
23     return response.toString();
24   }
```

Kode 3.3: Response.java

### 3.3.7 LogService Class

Bagian awal kelas `LogService` berisi deklarasi variabel inti yang mengatur seluruh proses pencatatan dan rotasi *log*. Variabel seperti `LOG_SIZE`, `activeLogFile`, dan `fileHandler` digunakan untuk mengatur batas ukuran *log*, lokasi penyimpanan, dan mekanisme penulisan. Objek `lock` memastikan operasi pencatatan berjalan aman dalam konteks multithreading.

Blok *static* menjalankan proses inisialisasi ketika kelas pertama kali dimuat. Prosedurnya mencakup pembuatan direktori `logs` jika belum tersedia, pembersihan berkas kunci lama melalui `cleanupLockFiles()`, serta penentuan tanggal dan akhiran alfabet yang berlaku untuk rotasi. Setelah itu, metode `backupLeftoverActiveLogs()` memindahkan *log* aktif dari hari sebelumnya, dan `initializeActiveLog()` menyiapkan berkas *log* baru untuk digunakan. Dengan langkah-langkah ini, sistem memastikan lingkungan pencatatan berada dalam kondisi stabil sebelum proses *logging* berjalan.

```
1  public class LogService
2  {
3    private static final Logger logger = Logger.getLogger(LogService
       .class.getName());
4    private static final int LOG_SIZE = 2 * 1024;
5    private static File activeLogFile;
6    private static FileHandler fileHandler;
7    private static String currentDate;
8    private static String currentAlphabet;
9
10   private static LocalDateTime lastLogTimestamp;
11   private static final Object lock = new Object();
12
13   static
14   {
15     File logsDir = new File("logs");
16     if (!logsDir.exists()) {
17       logsDir.mkdirs();
18     }
19     cleanupLockFiles();
20     currentDate = LocalDate.now().format(DATE_FORMATTER);
21     currentAlphabet = getNextAvailableSuffix(currentDate);
22     backupLeftoverActiveLogs();
23     initializeActiveLog();
24   }
```

```
25    ...
26 }
```

Kode 3.4: LogService.java

### 3.3.8 Metode dalam LogService Class

Dalam kelas `LogService` terdapat berbagai macam metode yang dirancang untuk memastikan fungsionalitas dari *logging system* berjalan dengan seharusnya dan menghasilkan *log file* yang konsisten. Berikut adalah beberapa dari metode-metode yang ada di dalam kelas `LogService`.

#### A Metode deleteLockFiles()

Potongan kode `deleteLockFiles()` dirancang untuk menghapus *lock files* jika masih ada yang tersisa di direktori `activelogs`. *lock file* tersebut dapat tertinggal jika misalnya, aplikasi/servis *crash* atau server dimatikan. Java `FileHandler` secara otomatis membuat berkas *lock* dengan ekstensi `.lck` pada direktori tempat `activelogs`. *Lock file* berfungsi sebagai pengunci agar tidak terjadi *write* bersamaan jika ada lebih dari satu *instance* `FileHandler`.

Fungsi `deleteLockFiles(directory)` melakukan pembersihan secara rekursif pada dua direktori utama: `activelogs` dan `backuplogs`. Metode ini membaca seluruh isi direktori melalui `listFiles()`, kemudian menelusuri setiap entri. Jika entri adalah subdirektori, fungsi dipanggil kembali untuk melakukan pemeriksaan mendalam. Jika entri adalah berkas dengan akhiran `.lck`, maka berkas tersebut dihapus karena dianggap sebagai residu yang tidak lagi relevan. Dengan pendekatan ini, seluruh *lock file* yang tersisa dari eksekusi sebelumnya dapat dibersihkan secara menyeluruh sebelum `FileHandler` dipasang ulang, sehingga sistem dapat menjalankan proses pencatatan tanpa hambatan dan tanpa risiko gagal membuka berkas *log*.

```
1 private static void deleteLockFiles(File directory)
2 {
3   for (int i = 0; i < files.length; i++) {
4     File file = files[i];
5     if (file.isDirectory()) {
6       deleteLockFiles(file);
7     }
8     else if (file.getName().endsWith(".lck")) {
```

```
        file.delete();
      }
    }
}
```

Kode 3.5: Metode cleanupLockFiles() dan deleteLockFiles()

## B Metode backupLeftoverActiveLogs

Metode `backupLeftoverActiveLogs()` memindai seluruh *file* di direktori `logs/activelogs` dan menangani log yang tidak lagi berasal dari tanggal berjalan. Setiap *file* diproses satu per satu. Jika pembacaan waktu modifikasi gagal, *file* dilewati. Timestamp terakhir *file* diambil menggunakan `Files.getLastModifiedTime()`, kemudian dikonversi ke `LocalDateTime`. Bila tanggalnya sama dengan `currentDate`, *file* dianggap masih aktif dan tidak disentuh.

Untuk *file* yang tanggalnya berbeda, fungsi menyiapkan direktori cadangan `logs/backuplogs/<tanggal>` dan membuatnya bila belum ada. Sebelum dipindahkan, fungsi menambahkan penanda sesi baru dengan `StandardOpenOption.APPEND` untuk memberikan batas yang jelas antara sesi pencatatan. Nama *file* cadangan dibuat berdasarkan timestamp asli *file* ditambah suffix unik yang diperoleh melalui `getNextAvailableSuffix(fileDate)`.

Terakhir, *file* dipindahkan menggunakan `Files.move()` dengan opsi penggantian bila *file* tujuan sudah ada. Setiap kegagalan—baik saat membaca metadata *file* maupun saat pemindahan—dicatat melalui `logger`. Dengan alur ini, sistem memastikan log lama tersimpan rapi dalam struktur harian tanpa mengganggu log aktif hari berjalan.

```
private static void backupLeftoverActiveLogs()
{
  for (int i = 0; i < logFiles.length; i++)
  {
    File logFile = logFiles[i];

    LocalDateTime fileTime;

    fileTime = LocalDateTime.ofInstant(Files.getLastModifiedTime(
    logFile.toPath()).toInstant(), java.time.ZoneId.systemDefault()
    );

```

```
11      String fileDate = fileTime.toLocalDate().format(DATE_FORMATTER
        );
12      if (fileDate.equals(currentDate)) {
13        continue;
14      }
15
16      File backupDir = new File("logs/backuplogs", fileDate);
17      if (!backupDir.exists()) {
18        backupDir.mkdirs();
19      }
20
21      File backupFile = new File(backupDir, backupName);
22      Files.move(logFile.toPath(), backupFile.toPath(),
        StandardCopyOption.REPLACE_EXISTING);
23    }
24  }
```

Kode 3.6: backupLeftoverActiveLogs()

### C Metode performDayRotation

Metode `performDayRotation()` pada Kode 3.7 berfungsi menjalankan proses rotasi harian pada sistem pencatatan log. Operasi ini dibungkus dalam blok `synchronized` untuk memastikan bahwa rotasi tidak mengalami kondisi balapan ketika diakses secara bersamaan oleh beberapa *thread*. Langkah pertama yang dilakukan adalah memastikan direktori cadangan sesuai tanggal aktif tersedia; jika belum ada, direktori tersebut dibuat. Bila berkas log aktif masih ada dan memiliki isi, sistem menuliskan penanda rotasi harian, menentukan waktu rotasi berdasarkan `lastLogTimestamp` atau *timestamp* saat ini, lalu membangun nama berkas cadangan menggunakan `String.format()` dengan pola waktu dan akhiran alfabet yang sedang digunakan. Setelah penulis log ditutup, berkas log aktif dipindahkan ke direktori cadangan menggunakan `Files.move()` agar tidak hilang atau tertimpa.

Setelah proses pemindahan selesai atau jika tidak ada berkas log aktif yang perlu dipindahkan, sistem memperbarui tanggal operasi ke `newDate` dan menghitung akhiran alfabet baru melalui `getNextAvailableSuffix()`. Sebuah berkas log baru kemudian disiapkan pada direktori `logs/activelogs` dengan nama sesuai tanggal yang diperbarui. Jika berkas tersebut belum ada, maka sistem membuatnya dan menginisialisasi kembali penulis log. Tahap akhir adalah

pencatatan penanda sesi log baru sebagai indikasi bahwa sistem telah memulai siklus pencatatan harian yang baru.

```
public static void performDayRotation(String newDate)
{
  synchronized(lock)
  {
    try
    {
      File backupDir = new File("logs/backuplogs", currentDate);

      if (activeLogFile != null && activeLogFile.exists() &&
    activeLogFile.length() > 0)
      {
        LocalDateTime rotationTime;
        if (lastLogTimestamp != null) {
          rotationTime = lastLogTimestamp;
        }
        else {
          rotationTime = LocalDateTime.now();
        }
        String backupName = String.format("%s_%s.log",
    rotationTime.format(DATETIME_FORMATTER), currentAlphabet);

        closeLogWriter();
        File backupLogFile = new File(backupDir, backupName);
        Files.move(activeLogFile.toPath(), backupLogFile.toPath(),
     StandardCopyOption.REPLACE_EXISTING);
      }
      else {
        closeLogWriter();
      }
    }
    catch (IOException e) {
      logger.log(Level.SEVERE, "Day rotation failed", e);
    }
  }
}
```

Kode 3.7: performDayRotation()

### D Metode initializeActiveLog

Metode `initializeActiveLog()` pada Kode 3.8 bertugas untuk memastikan bahwa berkas log harian aktif siap digunakan sebelum proses pencatatan dimulai. Sistem pertama-tama memverifikasi keberadaan direktori `logs/activelogs` dan membuatnya apabila belum tersedia. Setelah itu, metode menentukan berkas log aktif berdasarkan `currentDate`. Jika berkas sudah ada dan ukurannya mencapai atau melampaui batas yang ditentukan oleh `LOG_SIZE`, sistem segera memicu *rotation* melalui pemanggilan `rotateActiveLog()` untuk mencegah penumpukan data yang berlebihan dalam satu berkas.

Apabila berkas belum ada atau ukurannya masih dalam batas yang wajar, sistem membuka penulis log melalui `initializeLogWriter()`. Jika berkas tersebut merupakan berkas baru, metode turut menuliskan penanda awal sesi menggunakan `writeRotationMarker("NEW LOG SESSION STARTED")`. Dengan demikian, prosedur ini memastikan bahwa setiap sesi pencatatan dimulai dengan struktur log yang bersih dan sesuai standar manajemen log yang telah ditetapkan.

```
1  private static void initializeActiveLog()
2  {
3    File activeLogsDir = new File("logs/activelogs");
4
5    activeLogFile = new File(activeLogsDir, currentDate + ".log");
6    try
7    {
8      boolean newFile = !activeLogFile.exists();
9      if (!newFile && activeLogFile.length() >= LOG_SIZE) {
10       rotateActiveLog();
11     }
12     else {
13       initializeLogWriter();
14     }
15   }
16   catch (IOException e) {
17     throw new RuntimeException("Failed to init active log", e);
18   }
19 }
```

Kode 3.8: initializeActiveLog()

## E Metode rotateActiveLog

Fungsi `rotateActiveLog()` pada Kode 3.9 melakukan rotasi terhadap *log file* aktif ketika ukuran atau kondisi tertentu mengharuskan pembuatan sesi log baru. Seluruh proses dijalankan dalam blok `synchronized` untuk memastikan rotasi tidak dilakukan bersamaan oleh *thread* lain.

Pertama, fungsi memeriksa apakah *log file* aktif valid. Jika tidak ada atau tidak lagi tersedia, proses dihentikan. Bila valid, sistem menuliskan penanda rotasi menggunakan `writeRotationMarker()`. Waktu rotasi ditentukan dari `lastLogTimestamp` jika tersedia, atau dari `LocalDateTime.now()`. Nama *log file* cadangan disusun menggunakan *timestamp* tersebut dan *suffix* alfabet saat ini.

Direktori `logs/backuplogs/<tanggal>` kemudian dipastikan ada; jika belum dibuat, direktori akan diinisialisasi. Setelah *writer* ditutup, *log file* aktif dipindahkan ke direktori *backup* menggunakan `Files.move()` dengan opsi penggantian bila sudah ada *file* lama.

Setelah pemindahan berhasil, *suffix* alfabet diperbarui melalui `incrementAlphabet()`. Sistem kemudian membuat *log file* baru untuk melanjutkan pencatatan log, membuka *writer* baru, dan menambahkan penanda bahwa sesi log baru telah dimulai. Jika terjadi kesalahan I/O di tengah proses, fungsi melempar `RuntimeException` untuk memastikan eror tidak terabaikan.

```
1  private static void rotateActiveLog()
2  {
3    synchronized(lock)
4    {
5      try
6      {
7        LocalDateTime rotationTime;
8        if (lastLogTimestamp != null) {
9          rotationTime = lastLogTimestamp;
10       }
11       else {
12         rotationTime = LocalDateTime.now();
13       }
14       closeLogWriter();
15       Files.move(activeLogFile.toPath(), backupLogFile.toPath(),
     StandardCopyOption.REPLACE_EXISTING);
16       activeLogFile.createNewFile();
17       initializeLogWriter();
18     }
```

```
    catch (IOException e) {
      throw new RuntimeException("Rotation failed", e);
    }
  }
}
```

Kode 3.9: rotateActiveLog()

## F Metode writeToActiveLog

Metode `writeToActiveLog()` pada Kode 3.10 bertanggung jawab menuliskan entri log baru sekaligus memastikan bahwa siklus rotasi harian dan rotasi ukuran berkas berjalan dengan benar. Seluruh proses ditempatkan di dalam blok `synchronized` untuk mencegah akses bersamaan yang dapat menyebabkan inkonsistensi data. Sistem terlebih dahulu memperoleh waktu saat ini dan mem-formatnya menggunakan `DATE_FORMATTER`. Jika tanggal baru berbeda dari `currentDate`, metode memicu rotasi harian melalui `performDayRotation()` agar pencatatan dipisahkan berdasarkan hari. Setelah itu, nilai `lastLogTimestamp` diperbarui untuk menjaga *history* waktu log.

Setelah penulisan dilakukan, sistem memeriksa ukuran berkas log aktif. Jika ukuran telah mencapai batas `LOG_SIZE`, maka rotasi ukuran dieksekusi melalui `rotateActiveLog()` agar berkas tidak melampaui kapasitas yang ditetapkan. Seluruh prosedur dibungkus dalam blok *try-catch* untuk memastikan jika ada kegagalan penulisan, maka akan tercatat melalui `logger.log()`.

```
private static void writeToActiveLog(String level, String message)
{
  synchronized(lock)
  {
    try
    {
      LocalDateTime now = LocalDateTime.now();
      String newDate = now.format(DATE_FORMATTER);
      if (!newDate.equals(currentDate)) {
        performDayRotation(newDate);
      }
      lastLogTimestamp = now;
      if (activeLogFile != null && activeLogFile.length() >=
    LOG_SIZE) {
        rotateActiveLog();
      }
```

```
16      }
17      catch (Exception e) {
18        logger.log(Level.SEVERE, "Write failed", e);
19      }
20    }
21  }
```

Kode 3.10: writeToActiveLog()

### 3.3.9 Hasil

Berikut adalah contoh-contoh untuk isi dari *log file* saat pencatatan *event* atau *transaction*. Contoh ini disajikan untuk memberikan gambaran menyeluruh mengenai bagaimana sistem mendokumentasikan setiap aktivitas operasional, mulai dari proses inisialisasi layanan hingga interaksi yang terjadi antara *client* dan *service*. Melalui keluaran *log* tersebut, pola pencatatan, struktur informasi, serta parameter-parameter penting yang dicatat—seperti identifikasi peristiwa, waktu kejadian, dan detail koneksi—dapat diamati secara jelas.

#### A Inisialisasi Log File

*Logging system* mencatat rangkaian peristiwa sejak layanan E-Signing pertama kali diaktifkan. Saat aplikasi dijalankan, sistem menghasilkan entri awal yang menandai dimulainya sesi pencatatan *log* baru, diikuti dengan informasi bahwa layanan telah siap menerima koneksi. Ketika *client* kemudian terhubung, *log* merekam *source IP address* koneksi dan setiap pencatatan ditetapkan sebuah *event ID* yang unik. Setelah koneksi diterima, sistem menuliskan detail awal proses permintaan (*request*) yang menunjukkan bahwa komunikasi antara *client* dan *service* telah resmi dimulai. Hasil dari *log file* terdapat pada Gambar 3.5.

```
1  [2025-10-26 15:05:28.515] [INFO] ###### NEW LOG SESSION STARTED ######
2  [2025-10-26 15:05:28.517] [INFO] Service is Listening on Port 9201...
3  [2025-10-26 15:05:45.001] [INFO] Client connected: /127.0.0.1:59077 [event=f5ac0771]
4  [2025-10-26 15:05:45.006] [INFO] [f5ac0771] [CONNECTION]
5      from=/127.0.0.1:59077
6      -> Request started
```

Gambar 3.5. Inisialisasi log file

## B Logging untuk Request dan Response

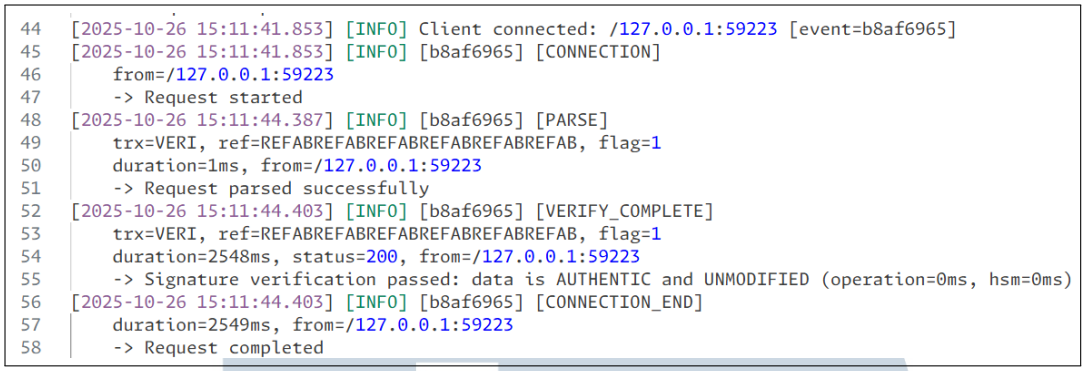
Setelah koneksi berhasil dibuka, sistem mencatat tahap pemrosesan permintaan secara terstruktur. *Log* pertama menunjukkan bahwa aplikasi berhasil melakukan *parsing* terhadap *request* yang diterima, termasuk informasi transaksi `SIGN`, *reference*, serta *flag* yang diberikan. Tahap berikutnya, proses penandatanganan (*signing*) diselesaikan dan direkam sebagai `SIGN_COMPLETE`. *Log* ini memuat durasi total pemrosesan, status keberhasilan (200), serta rincian waktu operasional internal, seperti eksekusi operasi dan interaksi dengan HSM. Setelah seluruh proses selesai, sistem menutup sesi dengan mencatat `CONNECTION_END`, yang menandai bahwa permintaan dari *client* telah dipenuhi sepenuhnya dan koneksi dapat diakhiri. Pada Gambar 3.6 adalah hasil dari *log file* saat *client* mengirim *request* untuk tanda tangan (`SIGN`).

```
[2025-10-26 15:06:39.133] [INFO] [f5ac0771] [PARSE]
    trx=SIGN, ref=REFABREFABREFABREFABREFABREFAB, flag=1
    duration=2ms, from=/127.0.0.1:59077
    -> Request parsed successfully
[2025-10-26 15:06:39.169] [INFO] [f5ac0771] [SIGN_COMPLETE]
    trx=SIGN, ref=REFABREFABREFABREFABREFABREFAB, flag=1
    duration=54151ms, status=200, from=/127.0.0.1:59077
    -> Sign completed (operation=22ms, hsm=19ms)
[2025-10-26 15:06:39.295] [INFO] [f5ac0771] [CONNECTION_END]
    duration=54295ms, from=/127.0.0.1:59077
    -> Request completed
```

Gambar 3.6. Hasil log file untuk request dan response

## C Logging untuk Request Verifikasi

Selain `SIGN`, *client* juga dapat memverifikasi *signature* dengan cara mengirim permintaan *verification*, sistem mencatat tahap awal `PARSE` yang menunjukkan bahwa permintaan dengan transaksi `VERI` dan referensi `REFABREFABREFABREFABREFABREFAB`. Proses kemudian berlanjut ke tahap `VERIFY_COMPLETE`, di mana *service* menyelesaikan pemeriksaan tanda tangan digital dan memastikan bahwa data yang diterima bersifat *authentic* dan *unmodified*. Log ini juga memuat status keberhasilan `200`. Setelah verifikasi selesai tanpa kendala, sistem menutup sesi dengan mencatat `CONNECTION_END`, menandakan bahwa seluruh proses permintaan telah selesai ditangani dan koneksi dengan *client* dapat diakhiri. Hasil dari *log file* saat proses verifikasi ada pada Gambar 3.7.




Gambar 3.7. Hasil log file saat verifikasi

## D Struktur Request dan Response

Pada Gambar 3.8 adalah contoh saat *client* mengirim *request* untuk `SIGN`. Struktur *request* yang dikirim oleh *client* memiliki format dan spesifikasi yang sudah ditetapkan. Sistem ini menggunakan format *fixed-width* dengan panjang tetap 620 karakter untuk setiap *request* dan *response*. Struktur *request* terdiri dari beberapa bagian:

1. *Header*: Berisi kode `026C` yang merepresentasikan panjang pesan dalam format *big-endian*.
2. *Transaction Code*: Menentukan jenis operasi yang diminta. Terdapat dua jenis operasi:
   - `SIGN`: Untuk melakukan penandatanganan digital terhadap *digest* dokumen.
   - `VERI`: Untuk melakukan verifikasi terhadap tanda tangan digital yang sudah ada.
3. *Reference*: Nomor referensi transaksi yang dapat digunakan untuk pelacakan dan *audit trail*. Contoh: `REFABREFABREFABREFABREFABREFAB`
4. *Digest*: *Hash* kriptografis dari dokumen dalam format heksadesimal. Contoh: `AAAA0000BBBB1111CCCC2222DDDD3333EEEE4444FFFF5555666777788889999`.
5. *Signature*: Tanda tangan digital dalam format *Base64*. Untuk operasi `SIGN`, bagian ini diisi dengan spasi kosong karena tanda tangan akan dihasilkan oleh *Hardware Security Module* (HSM). Untuk operasi `VERI`, bagian ini harus berisi tanda tangan yang akan diverifikasi.

6. *Flag*: Indikator status, 1 jika proses berhasil dan 0 jika proses mengalami *error*.

Dalam contoh ini, bagian *signature* (512 karakter) diisi dengan spasi kosong karena tanda tangan belum ada dan akan dihasilkan oleh HSM. Lalu *response* mengembalikan tampilan protokol/format dengan *digest* dokumen yang sudah diubah menjadi *Base64* dan dilengkapi dengan *signature* yang dihasilkan oleh *HSM*. *Flag* mengembalikan nilai 1 yang menandakan proses berhasil.

```
C:\Users\dylan>nc localhost 9201
026CSIGNREFABREFABREFABREFABREFABREFAB     AAAA0000BBBB1111CCCC2222DDDD3333EEEE4444FFFF555566667777888899999
                                                                                                        1
026CSIGNREFABREFABREFABREFABREFABREFAB     qqoAALu7ERHMzCIi3dOzM+7uRET//1VVZmZ3d4iImZk=
TTsBJwGc9XiyAIuriC/41Rhyy+LRyzjaO9O8fABfceW9kAVaVJ1xtGA9esXLXyGOxsuQT/VZ1LozimGA+T7kJzmxvj9y/K9+VOvj/IVR1LB
9AACkwxqdUIPXmV1yVjYrVQA2llzqLckGI9LOJOEICh7TrOCfAaZpEa7Fg7STJtBN+mvulbXSclprl1C37LLYCZYdzW9MpidZ8Mi1jV7n8V
ZIfITtP5VhrITtv/7gGj+Smp84ZPkrtczskEeq+ZUuQ2Vv8dnkFJL2o21SW9szjUGFkNiuoj+LpuqxFyc82S3vf3sy6tYcguIRdqM7m1Tip
q+g7N7wSaH8VD7pAuOJMg==
                                                                                                        1
```

Gambar 3.8. Struktur request dan response

## E Struktur Request Verifikasi

Untuk contoh verifikasi (`VERI`) pada Gambar 3.9, *digest* dapat dikirim dalam format *Base64* (hasil konversi dari HSM) atau tetap dalam format heksadesimal. *Parser* akan secara otomatis mendeteksi format yang digunakan. Proses verifikasi tanda tangan digital melibatkan beberapa tahap yang memastikan integritas data dan keaslian penandatangan:

1. Pengiriman *Request* Verifikasi: *Client* mengirim *request* `VERI` yang berisi *digest* dokumen dan tanda tangan digital yang akan diverifikasi. *Request* ini menggunakan format yang sama dengan *request* penandatanganan, yaitu 620 karakter dengan struktur *fixed-width*.
2. Konversi *Digest* ke *Base64*: Setelah *digest* tersimpan dalam format heksadesimal secara internal, *server* mengkonversi *digest* tersebut ke format *base64* melalui metode `getDigestBase64()`. Proses ini melibatkan:
   - Konversi *string* heksadesimal menjadi *array of bytes*.
   - Encoding *array of bytes* tersebut menggunakan *Base64 encoder*.

3. Pembentukan Dokumen XMLDSig: *Digest* dalam format *base64* dimasukkan ke dalam *template* XML *Digital Signature* (XMLDSig) yang sudah disiapkan.

4. Verifikasi oleh *HSM*: *HSM* (*Hardware Security Module*) melakukan operasi verifikasi kriptografis dengan langkah-langkah berikut:

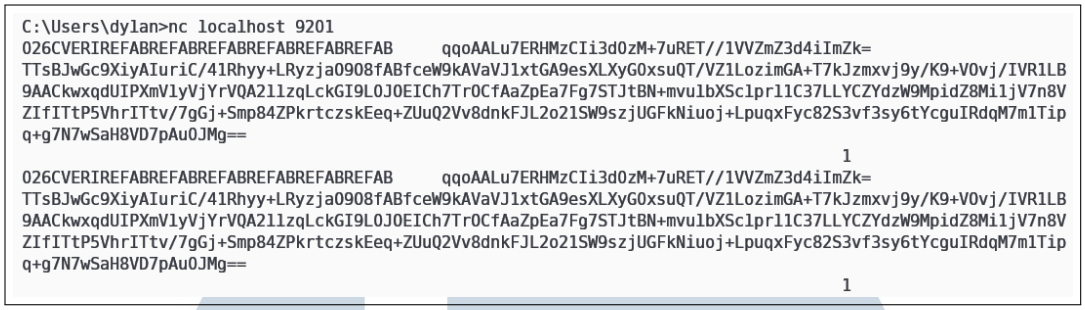
    - Menerima dokumen XML yang berisi *digest* dan tanda tangan digital yang akan diverifikasi
    - Menggunakan *public key* yang berkorespondensi dengan *private key* yang digunakan saat penandatanganan
    - Melakukan operasi dekripsi asimetris terhadap tanda tangan untuk mendapatkan *hash* asli
    - Menghitung ulang *hash* dari dokumen XML yang diterima
    - Membandingkan *hash* hasil dekripsi dengan *hash* yang dihitung ulang

5. Penentuan Hasil Verifikasi: Berdasarkan hasil operasi HSM, *server* menentukan status verifikasi:

    - Jika kedua *hash* identik: verifikasi berhasil (`flag=1`), yang berarti tanda tangan valid dan data tidak mengalami modifikasi sejak ditandatangani
    - Jika *hash* berbeda: verifikasi gagal (`flag=0`), yang mengindikasikan bahwa data telah dimodifikasi atau tanda tangan tidak sesuai

6. Pengiriman *Response*: *Server* membentuk *response* dengan struktur 620 karakter yang berisi:

    - *Digest* dalam format *Base64* (sama dengan yang diverifikasi)
    - Tanda tangan digital yang sama dengan yang dikirim dalam *request*
    - *Flag* yang menunjukkan hasil verifikasi (`1` untuk berhasil, `0` untuk gagal)




Gambar 3.9. Struktur request dan response saat verify

## F Contoh Log File Saat Error Handling

Sebagai contoh untuk *error handling*, pada Gambar 3.10 ditampilkan ketika tahap `PARSE` gagal karena nilai *transaction code* yang diterima tidak sesuai dengan spesifikasi. *Log* menampilkan pesan `ERROR: IllegalArgumentException: Invalid transaction code: IGas`. Hal ini menunjukkan bahwa *client* mengirimkan kode transaksi yang tidak dikenali oleh sistem. Sesuai desain, hanya dua nilai yang valid, yaitu `SIGN` dan `VERI`. Setiap nilai di luar itu dianggap tidak sah dan mengakibatkan respons `status=400` dengan jenis kesalahan `PARSE_ERROR`. Sistem kemudian menutup koneksi setelah mencatat bahwa permintaan tidak dapat diproses.

```
[2025-10-28 14:06:28.489] [INFO] Client connected: /127.0.0.1:38775 [event=aad4f507]
[2025-10-28 14:06:28.489] [INFO] [aad4f507] [CONNECTION]
    from=/127.0.0.1:38775
    -> Request started
[2025-10-28 14:06:29.327] [SEVERE] [aad4f507] [PARSE]
    status=400, error=PARSE_ERROR, from=/127.0.0.1:38775
    -> Failed to parse or process request
    ERROR: IllegalArgumentException: Invalid transaction code: IGas
[2025-10-28 14:06:29.327] [INFO] [aad4f507] [CONNECTION_END]
    duration=832ms, from=/127.0.0.1:38775
    -> Request completed
```

Gambar 3.10. Error transaction code

Contoh kedua pada Gambar 3.11 adalah saat terjadi kegagalan pada tahap `VALIDATION`. Sistem mendeteksi bahwa pesan yang diterima tidak memenuhi panjang yang ditetapkan, yaitu 620 karakter. *Log* mencatat `Incomplete message: expected 620 chars, received 615`. Error ini menunjukkan bahwa *client* mengirim pesan yang lebih pendek dari format *fixed-width* yang diwajibkan. Karena setiap field harus menempati posisi dan panjang tertentu, kekurangan satu karakter saja langsung membuat pesan tidak dapat

diparsing dengan benar. Sistem memberikan respons `status=400` dengan error `MSG_LENGTH_MISMATCH`. Koneksi ditutup setelah proses dianggap gagal.

```
[2025-10-26 15:08:45.802] [INFO] Client connected: /127.0.0.1:59194 [event=2555766d]
[2025-10-26 15:08:45.805] [INFO] [2555766d] [CONNECTION]
    from=/127.0.0.1:59194
    -> Request started
[2025-10-26 15:08:49.354] [SEVERE] [2555766d] [VALIDATION]
    status=400, error=MSG_LENGTH_MISMATCH, from=/127.0.0.1:59194
    -> Incomplete message: expected 620 chars, received 615
    ERROR: IllegalArgumentException: Message length mismatch
[2025-10-26 15:08:49.355] [INFO] [2555766d] [CONNECTION_END]
    duration=3550ms, from=/127.0.0.1:59194
    -> Request completed
```

Gambar 3.11. Error message length

Contoh berikutnya adalah pada Gambar 3.12 dimana ada indikasi bahwa *signature* yang dikirim *client* tidak cocok dengan *digest* dan *public key* yang terkait. Proses tercatat dengan hasil `VERIFY_COMPLETE` yang menandakan bahwa proses verifikasi dan pengecekan sudah selesai, namun hasil akhirnya tetap tertulis `DATA MAY BE TAMPERED OR SIGNATURE MISMATCH`. Kegagalan ini dapat disebabkan oleh beberapa kemungkinan, contohnya seperti:

- Tanda tangan digital telah berubah atau rusak.
- *Digest* tidak sesuai dengan dokumen asli.
- *Client* mengirimkan *signature* yang tidak dihasilkan oleh *HSM* yang sama.

```
[2025-10-26 15:12:30.854] [INFO] Client connected: /127.0.0.1:59234 [event=82e240b2]
[2025-10-26 15:12:30.854] [INFO] [82e240b2] [CONNECTION]
    from=/127.0.0.1:59234
    -> Request started
[2025-10-26 15:12:38.033] [INFO] [82e240b2] [PARSE]
    trx=VERI, ref=REFABREFABREFABREFABREFABREFAB, flag=1
    duration=1ms, from=/127.0.0.1:59234
    -> Request parsed successfully
[2025-10-26 15:12:38.040] [INFO] [82e240b2] [VERIFY_COMPLETE]
    trx=VERI, ref=REFABREFABREFABREFABREFABREFAB, flag=0
    duration=7182ms, status=200, from=/127.0.0.1:59234
    -> Signature verification failed: DATA MAY BE TAMPERED OR SIGNATURE MISMATCH (operation=0ms, hsm=0ms)
[2025-10-26 15:12:38.042] [INFO] ###### LOG ROTATION ######
```

Gambar 3.12. Error saat verifikasi

## 3.4 Kendala dan Solusi yang Ditemukan

Berbagai kendala ditemukan selama proses perancangan, implementasi, dan pengujian sistem, yang berpotensi memengaruhi keandalan serta konsistensi

layanan. Kendala tersebut mencakup aspek teknis pada pengelolaan *logging*, keandalan sistem dalam menghadapi kondisi tidak normal, serta konsistensi data pendukung seperti *timestamp* dan mekanisme *backup*. Setiap permasalahan dianalisis berdasarkan dampaknya terhadap stabilitas sistem, kemudian diikuti dengan solusi yang diterapkan tanpa menambah kompleksitas arsitektur. Pendekatan ini bertujuan memastikan bahwa sistem tetap berjalan secara konsisten, aman, dan mudah dipelihara dalam lingkungan operasional yang bersifat *multithreaded*.

### 3.4.1 Thread Safety dan Race Condition

Salah satu kendala yang menjadi perhatian dalam perancangan sistem ini adalah potensi terjadinya *race condition* serta isu *thread safety* pada pengelolaan berkas *log*.

- Kendala: Kondisi ini dapat muncul ketika beberapa *thread* berjalan secara paralel dan secara bersamaan mencoba menulis atau melakukan rotasi terhadap berkas *log*. Pada beberapa *build* awal, sistem masih menggunakan `FileWriter` untuk penulisan *log* serta *thread* Java terpisah untuk mendeteksi pergantian hari, misalnya melalui mekanisme `new Thread(() -> { while (running) { ... } }).` Pendekatan ini meningkatkan risiko akses bersamaan terhadap berkas *log*. Tanpa pengendalian akses yang memadai, kondisi tersebut berpotensi menyebabkan inkonsistensi data atau kegagalan proses *backup*.
- Solusi: Sistem pencatatan dimodifikasi dengan menggunakan `FileHandler` dari *Java Logging API* sebagai pengganti `FileWriter`. `FileHandler` menyediakan mekanisme penguncian berkas (*file locking*) dan pengelolaan penulisan yang lebih terkontrol, sehingga akses simultan dari beberapa *thread* dapat ditangani secara aman. Selain itu, deteksi pergantian hari diubah untuk tidak menggunakan*thread* Java ataupun kode lain yang memicu pembuatan proses baru.

### 3.4.2 Konsistensi Timestamp

- Kendala: Perbedaan waktu pencatatan antar entri di dalam *log* dan di nama *log file* berpotensi menimbulkan ambiguitas dalam proses analisis dan *audit*, terutama ketika sistem menangani banyak permintaan secara paralel.

- Solusi: *Timestamp* pada penamaan berkas *log* ditentukan berdasarkan waktu (tanggal dan jam) dari entri *log* terakhir yang tercatat, sehingga nama berkas merepresentasikan periode aktual aktivitas pencatatan dan memudahkan proses pelacakan serta *audit*.

### 3.4.3 Log File Backup Handling

- Kendala: Permasalahan utama pada *backup* berkas *log* terletak pada penentuan waktu pemicu proses. Rancangan awal melakukan *backup* otomatis saat pergantian hari (23.59–00.00) dengan pemantauan waktu aktif, yang berpotensi menimbulkan isu *thread safety* dan konflik dengan proses penulisan *log*. Selain itu, belum tersedia mekanisme *backup* ketika aplikasi atau server dimatikan, sehingga berkas *log* berisiko tidak tersimpan dengan baik.

- Solusi: Kebijakan *backup* diubah menjadi berbasis kondisi deterministik tanpa pemantauan waktu aktif. Pergantian hari dipicu oleh adanya entri *log* baru. Jika tanggal entri berbeda dari *log file* aktif, berkas lama dibackup dan *log file* baru dibuat. Mekanisme ini juga dijalankan saat aplikasi dinyalakan kembali, sehingga *log* dari hari sebelumnya dibackup sebelum sesi pencatatan baru dimulai.