

Implementasi NestJS dan Prisma pada pengembangan Backend Monolitik pada Aplikasi Web Antria

Tugas Akhir

diajukan untuk memenuhi salah satu syarat

memperoleh gelar sarjana

dari Program Studi Rekayasa Perangkat Lunak

Fakultas Informatika

Universitas Telkom

1302204044

Muhammad Rovino Sanjaya



Program Studi Sarjana Rekayasa Perangkat Lunak

Fakultas Informatika

Universitas Telkom

Bandung

2024

LEMBAR PENGESAHAN

**Implementasi NestJS dan Prisma pada pengembangan Backend Monolitik pada Aplikasi
Web Antria**

*Implementation of Monolithic Backend Development with NestJS and Prisma for Antria's
Web Application*

NIM: 1302204044

Muhammad Rovino Sanjaya

Tugas akhir ini telah diterima dan disahkan untuk memenuhi sebagian syarat memperoleh
gelar pada Program Studi Sarjana Rekayasa Perangkat Lunak
Fakultas Informatika
Universitas Telkom

Bandung, 2024

Menyetujui

Pembimbing I

Pembimbing II

(Dr. Mira Kania Sabariah, S.T., M.T.)

NIP: 14770011

(Monterico Adrian, S.T., M.T.)

NIP: 20870024

Ketua Program Studi
Sarjana Rekayasa Perangkat Lunak,

Dr. Mira Kania Sabariah, S.T., M.T.

NIP: 14770011

LEMBAR PERNYATAAN

Dengan ini saya, Muhammad Rovino Sanjaya, menyatakan sesungguhnya bahwa Tugas Akhir saya dengan judul ”**Implementasi NestJS dan Prisma pada pengembangan Backend Monolitik pada Aplikasi Web Antria**” beserta dengan seluruh isinya adalah merupakan hasil karya sendiri, dan saya tidak melakukan penjiplakan yang tidak sesuai dengan etika keilmuan yang berlaku dalam masyarakat keilmuan. Saya siap menanggung resiko/sanksi yang diberikan jika dikemudian hari ditemukan pelanggaran terhadap etika keilmuan dalam buku TA atau jika ada klaim dari pihak lain terhadap keaslian karya.

Bandung, 2024

Yang Menyatakan,

Muhammad Rovino Sanjaya

1. Pendahuluan	1
2. Kajian Pustaka	3
2.1 NodeJs	3
2.2 NestJs	3
2.3 Object Relational Mapping	3
2.4 PrismaJs	3
2.5 Arsitektur Monolitik	3
2.6 JSON Web Token	3
2.7 Anti Pattern	3
2.8 RESTful API	3
3. Sistem yang Dibangun	4
3.1 Request Life Cycle	5
3.1.1 Middleware	5
3.1.2 Guard	5
3.1.3 Interceptor	6
3.1.4 Controller	6
3.1.5 Service	6
3.2 Persiapan	7
3.2.1 Instalasi NestJS dan Prisma	7
3.2.2 Konfigurasi Prisma	7
3.3 Implementasi	8
3.3.1 Guards	8
3.3.2 Interceptor	8
3.3.3 Controller	10
3.3.4 Service	10
3.3.5 API Endpoint	10
3.4 Alur Pengujian	13
3.4.1 Unit Testing	13
3.4.2 Maintainability Testing	14
3.4.3 Performance Testing & Stress Testing	14
3.5 Deployment	15

4. Evaluasi	16
4.1 Hasil Pengujian	16
4.1.1 Unit Testing	16
4.1.2 Maintainability Testing	17
4.1.3 Performance Testing	18
4.2 Analisis Hasil Pengujian	22
4.2.1 Unit Testing	22
4.2.2 Performance Testing	22
4.2.3 Maintainability Testing	22
5. Kesimpulan	22
5.1 Saran	23

Implementasi NestJS dan Prisma pada pengembangan Backend Monolitik pada Aplikasi Web Antria

Muhammad Rovino Sanjaya¹, Mira Kania Sabariah², Monterico Adrian³

^{1,2,3}Fakultas Informatika, Universitas Telkom, Bandung

⁴Divisi Digital Service PT Telekomunikasi Indonesia

¹rovino@students.telkomuniversity.ac.id, ²mirakarnia@telkomuniversity.ac.id,

³monterico@telkomuniversity.ac.id

Abstrak

Populasi penduduk yang tinggi di Indonesia mengakibatkan antrian panjang dalam berbagai pelayanan publik atau pelayanan konsumen. Pelanggan harus datang ke tempat untuk mengambil antrian dan menunggu gilirannya, semakin panjang antrian, semakin lama waktu tunggu yang dibutuhkan. Hal ini tidak efisien dan menyebabkan banyak waktu terbuang hanya untuk menunggu antrian. Jika nomor antrian bisa didapatkan secara online dan dapat dipantau secara online, maka dapat mengurangi waktu yang terbuang. Dengan aplikasi antrian virtual dapat memperpendek antrian secara fisik, dengan cara antri secara virtual, melihat antrian yang sedang berjalan, dan booking tempat. Pada Pengembangan aplikasi ini, framework yang digunakan adalah NestJS dan PrismaJS dengan menerapkan RESTful API, Object Relational Mapping, dan menghindari Anti-Pattern. Framework NestJS mendukung pembuatan aplikasi ber-arsitektur monolitik dan *microservice*. Setelah aplikasi dibangun di arsitektur monolitik, aplikasi dapat dengan mudah dimigrasikan ke *microservice* saat penggunaan aplikasi sudah hampir mendekati batas muat pengguna.

Kata kunci : NestJS, PrismaJS, Anti-Pattern, Antrian, Backend, REST

Abstract

The high population in Indonesia results in long queues in various public services or consumer services. Customers have to come to the place to take a queue number and wait for their turn; the longer the queue, the longer the waiting time required. This is inefficient and causes a lot of time to be wasted just waiting in line. If queue numbers can be obtained online and monitored online, it can reduce wasted time. With a virtual queue application, it is possible to shorten physical queues by queuing virtually, viewing ongoing queues, and booking places. In the development of this application, the framework used is NestJS and PrismaJS by implementing RESTful API, Object Relational Mapping, and avoiding Anti-Patterns. The NestJS framework supports the creation of monolithic and microservice-architecture applications. After the application is built in a monolithic architecture, it can be easily migrated to a microservice architecture when the application usage is almost reaching the user load limit.

Keywords: NestJS, PrismaJS, Anti-Pattern, Queue, Backend, REST

1. Pendahuluan

Latar Belakang

Meningkatnya populasi di Indonesia mengakibatkan banyak pelanggan yang mengantri untuk mendapatkan layanan di bank, restoran, rumah sakit, dan tempat penyedia jasa lainnya. Mengantri merupakan kegiatan yang membosankan dan menguras waktu. Panjangnya antrian juga mampu berdampak pada mutu pelayanan di suatu tempat. Pelanggan yang harus menunggu lama berpotensi beralih ke pesaing, atau jika ada urusan lain yang lebih penting, maka pelanggan akan keluar dari tempat antrian, meninggalkan antriannya [8][4][18]. Solusi yang ada pada bank, kantor pos, dan rumah sakit saat ini menggunakan *ticketting* nomor antrian secara manual, dimana antrian yang sedang dilayani ditampilkan di layar pada ruang tunggu. Hal ini kurang efektif karena pelanggan harus berada di ruang tunggu[4].

Perkembangan teknologi yang cepat mengakibatkan penggunaan perangkat pintar atau *smartphone* merupakan hal lumrah, banyak bermunculan aplikasi antrian virtual seperti Antrique, Qiwee, ExaQue dimana pengguna dapat mengantri dari jarak jauh melalui aplikasi maupun *website*. Para pengguna aplikasi tersebut dapat melakukan hal

lain saat mengantri sebelum gilirannya. Namun, aplikasi-aplikasi tersebut memiliki kelemahan seperti tidak ada estimasi waktu antrian, dan masih belum ada yang berfokus ke sektor *food and beverage*.

Oleh karena itu, perlunya dikembangkan sebuah aplikasi yang memiliki fitur yang sama atau lebih dengan menutup kekurangan pada aplikasi tersebut. Pengembangan aplikasi menggunakan arsitektur monolitik karena mudahnya untuk dibuat dan di-*deploy* secara cepat untuk di iterasikan. Namun, arsitektur monolitik memiliki kelemahan seperti sulitnya untuk di-*maintenance*, *scale*, dan reliabilitas nya. Oleh karena itu, perlu diperhatikan bagaimana cakupan aplikasi kedepannya dan perlunya migrasi ke arsitektur *microservice* [6] [7].

Dalam pengembangan aplikasi web, pemilihan bahasa pemrograman untuk digunakan di *backend* sangatlah penting karena dapat mempengaruhi performa aplikasi yang dibangun. Dalam pemilihan bahasa pemrograman *backend*, banyak pilihan yang tersedia seperti PHP, Python, Ruby, PERL, dan banyak lagi. NodeJs merupakan *tools* yang memungkinkan bahasa JavaScript dapat dijalankan pada sisi *backend*. Dalam sisi performa, NodeJs lebih unggul dibanding PHP dan Python dalam sisi kecepatan melayani *request* dari client [19] [12].

NestJs merupakan *framework backend* dari Nodejs yang menggunakan bahasa typescript, dan bisa digunakan untuk pengembangan arsitektur berbasis *microservice* dan monolitik, jadi jika aplikasi dikembangkan pada arsitektur monolitik dapat dengan mudah dimigrasikan ke *microservice*. NestJs juga bisa digunakan bersamaan dengan *framework* PrismaJs untuk mengelola database [10]. PrismaJs merupakan *framework* Object Relational Mapping (ORM) [14], digunakan untuk mempercepat, dan mempermudah pengembangan aplikasi yang database-nya memiliki relasi yang kompleks dan sulit di-*maintenance* jika menggunakan Structured Query Language (SQL) [20].

Implementasi Application Programming Interface (API) yang digunakan adalah Representational State Transfer (RESTful) API, RESTful API adalah arsitektur untuk mempermudah komunikasi client-server agar efektif untuk transaksi data. Namun, pada implementasi RESTful API, ada beberapa hal yang perlu diperhatikan seperti keamanan saat transaksi atau komunikasi [3]. Keamanan yang lemah dapat mengakibatkan hacker dapat dengan mudah melakukan *request tampering*, mengambil data pengguna, dan dapat membocorkan data keuangan mitra. *Design pattern* juga perlu diperhatikan dalam penggunaan bahasa untuk API *endpoint* nya agar tidak terjadi *anti pattern*. *Anti pattern* terjadi saat penamaan API tidak sesuai dengan fungsi, atau ada fungsi sejenis tapi penamaannya berbeda jauh. Dengan menghindari *anti pattern*, dapat berakibat ke aplikasi yang lebih mudah di-*sustain* dan di-*maintain* [1] [2].

Berdasarkan uraian di atas, penelitian ini akan membuat sebuah *backend* aplikasi antrian dengan menggunakan arsitektur monolitik dengan *framework* NestJs dan PrismaJs sebagai *framework* nya. Setelah fitur-fitur aplikasi dibuat, perlu dilakukan unit testing untuk memvalidasi kodings yang telah ditulis. Hal ini bertujuan untuk meminimalisir bug dan mencegah terjadinya regresi saat fitur baru ditambahkan [16].

Topik dan Batasannya

Aplikasi antrian memerlukan backend developer untuk mengimplementasi-kan fungsi fungsi API dan manajemen database nya. Maka dapat dirumuskan permasalahan sebagai berikut:

1. Bagaimana mengembangkan software dengan index *maintainability* tinggi.
2. Bagaimana merancang API yang bebas dari *anti pattern*.
3. Bagaimana merancang sistem keamanan pada API untuk melayani *request*.

dan batasan masalah sebagai berikut:

1. Hanya berfokus kepada implementasi database menggunakan Prisma ORM.
2. Berfokus ke bagaimana membuat *endpoint* API yang tidak menimbulkan *anti pattern*.
3. Implementasi keamanan pada saat penanganan *request* menggunakan JSON Web Token (JWT).

Tujuan

Tujuan dari pengerjaan Tugas Akhir ini yaitu:

1. Mengimplementasikan Prisma ORM untuk mencapai index *maintainability* tinggi pada proyek.
2. Membuat API yang dapat dengan mudah dimengerti dan di *maintain*.
3. Mengamankan data pengguna dengan memerlukan *Authorization* pada setiap header request.

2. Kajian Pustaka

2.1 NodeJs

NodeJs adalah *runtime* javascript yang basisnya dibangun dari V8 Java-Script Engine. NodeJs berjalan dalam bentuk *event-driven*, dan menggunakan model *non blocking I/O*. meskipun menggunakan *event-driven* untuk melayani *request*, NodeJs dapat melayani jutaan koneksi dalam waktu bersamaan secara *asynchronous* [17].

2.2 NestJs

NestJs merupakan *framework* untuk Nodejs yang dikembangkan oleh Kamil Myśliwiec yang bertujuan untuk membuat aplikasi NodeJs yang efektif dan *scalable*. NestJs mendukung penggunaan bahasa typescript dan javascript. NestJs juga menggabungkan komponen-komponen dari Functional Programming, Object Oriented Programming, dan Functional Reactive Programming [13] [10].

2.3 Object Relational Mapping

Object Relational Mapping (ORM) adalah sebuah teknologi yang memeta-kan tabel database ke dalam objek, biasanya dipakai dalam bahasa yang berbasis Object Oriented Programming. Dengan menggunakan ORM, developer dapat berfokus ke *business logic* tanpa mengkhawatirkan penggunaan akses database yang rumit [9].

2.4 PrismaJs

PrismaJs adalah ORM Open Source, biasanya digunakan sebagai alternatif dari menggunakan Structured Query Language (SQL) secara langsung. PrismaJs mendukung penggunaan database MySQL, PostgreSQL, SQLite, SQL Server, CockroachDB, dan MongoDB. PrismaJs digunakan untuk mempermudah pengembangan database yang memiliki relasi yang kompleks dan besar, dengan cara memberikan API yang *type-safe* untuk *query* database nya dan mengembalikan hasil *query* dalam bentuk JavaScript Object Notation (JSON) [14].

2.5 Arsitektur Monolitik

Arsitektur Monolitik adalah arsitektur sebuah *software* dimana beberapa fungsi komponen yang berbeda seperti fungsi otorisasi, *business logic*, notifikasi, dan pembayaran. Semua fungsi tersebut berada dalam satu program dan *platform* yang sama. Arsitektur monolitik mudah untuk dikembangkan dan *di-deploy*. Namun, sulit untuk *di-maintenance* dan *di-scale* [6].

2.6 JSON Web Token

JSON Web Token (JWT) adalah sebuah token berbentuk *string* json yang dapat digunakan untuk melakukan otorisasi. Ukuran JWT tergolong kecil jadi dapat dengan cepat ditransfer antar client dan server. JWT menggunakan algoritma HMAC atau RSA untuk mengenkripsi *digital signature* yang digunakan. JWT memiliki 3 bagian pada *string* nya yang dipisahkan menggunakan ".", bagian ini berupa *header*, *payload*, dan *signature* [15].

2.7 Anti Pattern

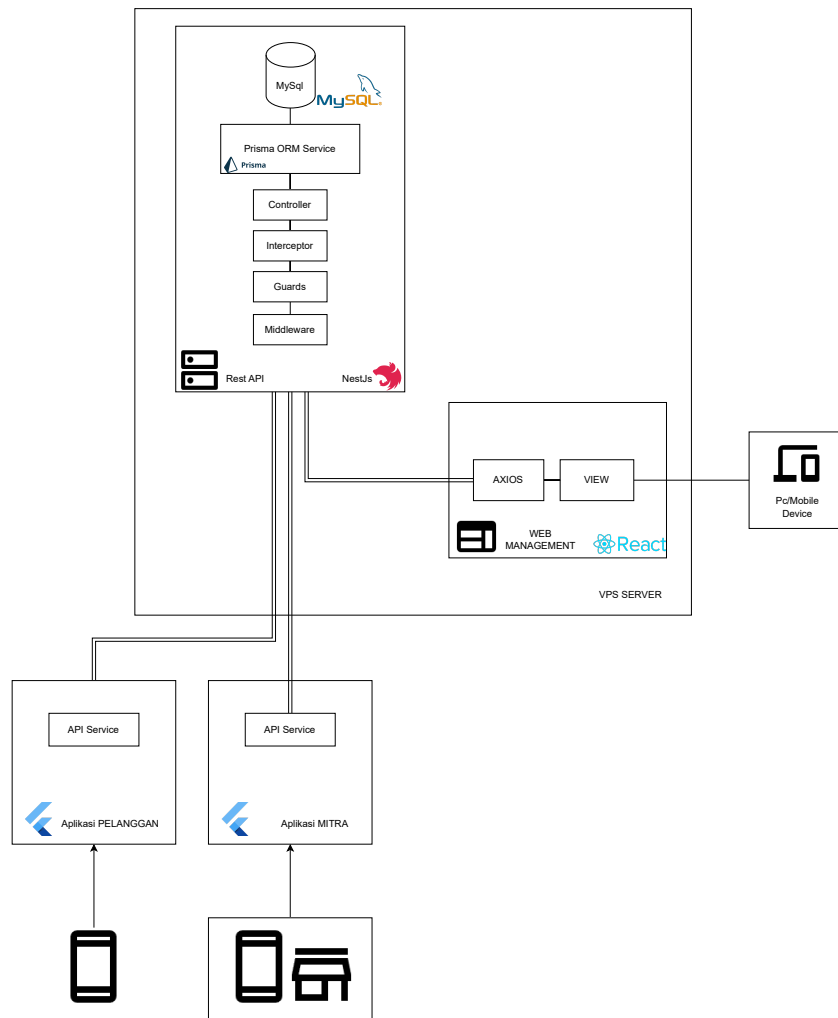
Anti Pattern terjadi jika pembuatan nama sebuah objek tidak konsisten dengan yang lain. Objek disini dapat berupa *endpoint* API, nama *variable*, nama fungsi, dan nama lain yang penggunaannya bersifat publik. Terjadinya *anti pattern* dapat mengakibatkan sulitnya untuk memahami suatu dokumentasi dan kodingan aplikasi [1] [2].

2.8 RESTful API

Representational State Transfer (RESTful) Application Programming Interface (API) adalah arsitektur untuk mempermudah komunikasi client-server agar efektif untuk transaksi data. Tipe data yang paling sering digunakan untuk transaksi client server adalah JSON. Karakteristik RESTful meliputi : Client-Server, Stateless, Layered Architecture, Caching, Code on Demand, dan Uniform Interface [5].

3. Sistem yang Dibangun

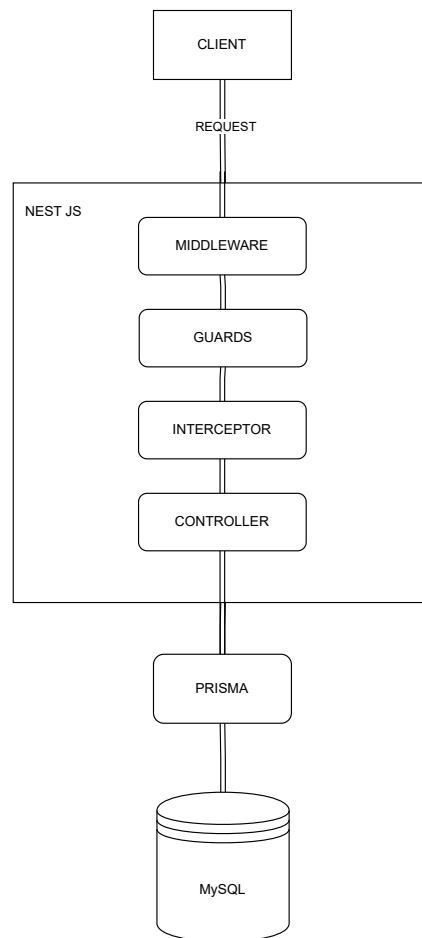
Implementasi dan perancangan RESTful API, Database, dan fungsi bisnis dikembangkan menggunakan *framework* NestJs. Pada NestJs terdapat beberapa komponen seperti: Middleware, Guards, Interceptor, Controller, dan Service. Service yang dipakai adalah PrismaJs untuk menghubungkan NestJs ke *Database System*. Desain arsitektur sistem secara keseluruhan pada gambar 1. Terdapat 3 Aplikasi frontend yang saling terhubung via backend, Aplikasi Pelanggan berupa flutter, Aplikasi Mitra berupa flutter, dan Website Dashboard Mitra berupa ReactJs.



Gambar 1. Arsitektur Desain Sistem

3.1 Request Life Cycle

Gambar 2 merupakan *Request Lifecycle* yang menjelaskan management API atau bagaimana alur *request* ditangani dari awal sampai akhir.



Gambar 2. Desain Sistem

3.1.1 Middleware

Pada Middleware, fungsi akan dipanggil sebelum masuk ke routing. fungsi middleware dapat mengakses data *request* dan *response*. beberapa fungsi Middleware seperti *logger*, dan cek notifikasi. Contoh penggunaan nya bisa dilihat pada listing 1, request dapat di hentikan maupun di alihkan ke middleware selanjutnya.

Listing 1: Middleware

```
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request...');
    next();
  }
}
```

3.1.2 Guard

Pada Guard, *request* akan dicek *authenticity*, untuk mengetahui validitas dari *request* tersebut. Tahap ini juga akan dicek keamanan session menggunakan JWT dan CSRF. Contoh penggunaan guard seperti pada listing 2, request yang menuju suatu controller akan di cek oleh guards, jika memiliki otorisasi maka akan dilanjutkan ke controller, jika tidak akan di reject dengan response forbidden access.

Listing 2: Penggunaan Guards pada Controller

```
@UseGuards(AuthGuard)
@Get('profile')
getProfile(@Request() req) {
  return req.user;
}
```

3.1.3 Interceptor

Setelah melalui Guard, Request akan masuk ke Interceptor. Dimana jika suatu *request* mempunyai suatu karakteristik yang ditentukan, maka akan menjalankan fungsi tambahan. Interceptor terjadi ketika *request* datang (*pre*), dan response (*post*). Contoh implementasi Interceptor terdapat pada listing 3.

Listing 3: Interceptor

```
@UseInterceptors(FileInterceptor('profile_picture',fileInterceptor()))
async update(@Param('id') id: string, @Body() data: Karyawan, @UploadedFile() file:
  Express.Multer.File): Promise<Karyawan> {}
```

3.1.4 Controller

Setelah melewati Interceptor, fungsi di Controller akan dijalankan. Jika pada Controller tersebut perlu data dari database maka akan turun ke service PrismaJs. Contoh karyawan controller pada listing 4 dimana request akan diteruskan ke service setelah password di hash.

Listing 4: Controller

```
@Controller('karyawan')
@Post()
async create(@Body() data: any): Promise<Karyawan> {
  const hashedPassword = await bcrypt.hash(data.password, 10);
  const karyawanData = { ...data, password: hashedPassword };
  return this.karyawanService.createKaryawan(karyawanData);
}
```

3.1.5 Service

Pada Service, fungsi akan melakukan database call menggunakan ORM ke database MySQL yang akan di kembalikan (return) ke Controller [10]. Contoh implementasi service pada listing 5 dimana berfungsi untuk mengambil banyak karyawan.

Listing 5: Service

```
async karyawans(params: {
  skip?: number;
  take?: number;
  cursor?: Prisma.KaryawanWhereUniqueInput;
  where?: Prisma.KaryawanWhereInput;
  orderBy?: Prisma.KaryawanOrderByWithRelationInput;
}): Promise<Karyawan[]> {
  const { skip, take, cursor, where, orderBy } = params;
  return this.prisma.karyawan.findMany({
    skip, take, cursor, where, orderBy,
  });
}
```

3.2 Persiapan

Untuk memulai proyek, ada beberapa tahap yang perlu dilakukan, pertama membuat proyek nestjs menggunakan node, lalu dilanjutkan menginstall dependensi yang diperlukan

3.2.1 Instalasi NestJS dan Prisma

Hal pertama yang dilakukan untuk menginstall NestJS adalah membuka terminal lalu menjalankan command npm untuk menginstall NestJS, dilanjut dengan menginstall prisma client.

Listing 6: terminal: npm

```
$ npm i -g @nestjs/cli
$ nest new antria
$ npm install prisma --save-dev
```

Command pada listing 6 untuk melakukan generasi folder proyek pada NestJS, dan menginstall prisma client sebagai dependensi. Prisma sendiri merupakan aplikasi CLI untuk membantu dalam manajemen database pada proyek NestJS menggunakan ORM.

3.2.2 Konfigurasi Prisma

Berdasarkan SRS, didapatkan beberapa entity pada database meliputi: Pelanggan, Mitra, Produk, OrderList, Pesanan, Antrian, Karyawan, Review, Chat, dan Analytic. Implementasi juga harus sesuai dengan ERD (Entity Relationship Diagram) yang telah dibuat pada gambar 3.

Listing 7: terminal: npx

```
$ npx prisma init
$ npx prisma migrate dev --name init
```

Command pada listing 7 untuk inisialisasi prisma pada proyek, berfungsi untuk generasi template config yang nanti harus diubah, seperti database connection dan database name nya. Pada tahap ini juga penulis perlu mendefinisikan model dan relasi pada database ke bentuk notasi modelling prisma.

Listing 8: scheme.prisma

```
model Pesanan {
  invoice      String      @id
  payment      Payment
  pemesanan    OrderType?  @default(ONLINE)
  takeaway     Boolean      @default(false)
  status       PaymentStatus @default(PENDING)
  oderlist     OrderList[]
  pelanggan    Pelanggan   @relation(fields: [pelangganId], references: [id])
  pelangganId  Int
  mitra        Mitra       @relation(fields: [mitraId], references: [id])
  mitraId      Int
  antrian      Antrian?
  antrianId    Int?
  created_at   DateTime     @default(now())
  updated_at   DateTime     @updatedAt
}

model OrderList {
  id          Int          @default(autoincrement()) @id
  quantity    Int          @default(1)
  note        String       @default("")
  pesanan     Pesanan      @relation(fields: [pesananId], references: [invoice])
  produk      Produk       @relation(fields: [produkId], references: [id])
  produkId    Int
  pesananId   String
}
```

Pada listing 8 penulis mendefinisikan skema model pesanan dan model orderlist beserta relasi nya pada prisma. syntax @relation berguna untuk mendefinisikan relasi nya pada model.

3.3 Implementasi

3.3.1 Guards

implementasi code guards dibuat 2 fungsi, guards untuk pengguna, dan guards untuk mitra. Contoh implementasi pada listing 9, disini penulis mendefinisikan guard untuk memeriksa JWT Token dari request yang diterima apakah valid.

Listing 9: Authentication Guards

```
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(
        token,
        {
          secret: jwtConstants.secret
        }
      );
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

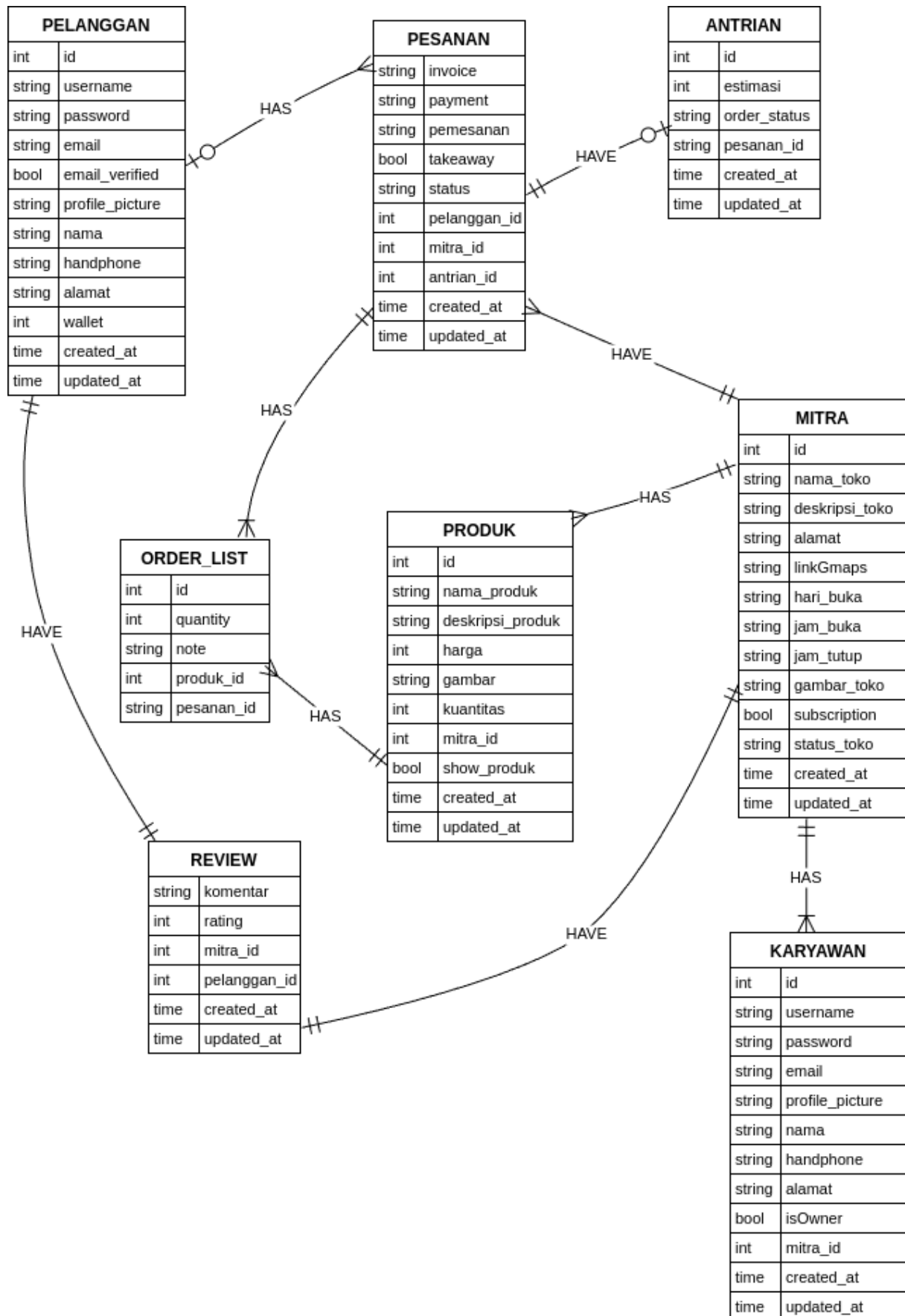
  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

3.3.2 Interceptor

Contoh implementasi terdapat pada listing 10, dimana penulis mengimplementasikan interceptor untuk mendeteksi dan mengambil file dari multipart request data untuk diolah dan disimpan pada server.

Listing 10: File Interceptor

```
@UseInterceptors(FileInterceptor('profile_picture',{
  storage: diskStorage({
    destination: './MediaUpload/',
    filename: (req, file, callback) => {
      const uniqueSuffix = uuidv4();
      const fileExtName = path.extname(file.originalname);
      const newFileName = `${uniqueSuffix}${fileExtName}`;
      callback(null, newFileName);
    }
  })
}))
```



Gambar 3. Entity Relationship Diagram

3.3.3 Controller

Controller minimal mengikuti banyak entity pada model. masing masing controller memiliki 4 fungsi yang merepresentasikan method http yaitu GET, POST, PUT, DELETE. Khusus untuk endpoint DELETE, pada beberapa controller data tidak secara langsung di delete, namun hanya diubah status nya dari enabled ke disabled.

1. Pelanggan controller
2. Mitra controller
3. Produk controller
4. OrderList controller
5. Pesanan controller
6. Antrian controller
7. Karyawan controller
8. Review controller
9. Analytic controller

3.3.4 Service

Sama seperti controller, banyak service minimal mengikuti banyak entity pada model. bedanya service berinteraksi langsung dengan ORM Prisma.

1. Pelanggan Service
2. Mitra Service
3. Produk Service
4. OrderList Service
5. Pesanan Service
6. Antrian Service
7. Karyawan Service
8. Review Service
9. Analytic Service

3.3.5 API Endpoint

Untuk menghindari Anti Pattern, API endpoint sesuai dengan nama entity pada model. setiap endpoint entity mendukung 4 method http yaitu POST, GET, PUT, dan DELETE.

1. Auth Endpoint, Tabel 1 memetakan controller auth pada endpoint REST API.

Tabel 1. Auth Endpoint Table

Path	Method	Description
/auth/login/pelanggan	POST	Endpoint untuk aplikasi mobile pelanggan, pengguna memasukkan credential login lalu akan mendapatkan JWT Token untuk mengakses endpoint API lain nya
/auth/login/mitra	POST	Endpoint untuk aplikasi mobile mitra, dan Web Dashboard. pengguna memasukkan credential login lalu akan mendapatkan JWT Token untuk mengakses endpoint API lain nya
Continued on next page		

Tabel 1 – continued from previous page

Path	Method	Description
/auth/profile	GET	Endpoint untuk decode JWT payload

2. Pelanggan Endpoint, Tabel 2 memetakan controller pelanggan pada endpoint REST API.

Tabel 2. Pelanggan Endpoint Table

Path	Method	Description
/pelanggan	GET	Endpoint untuk mengambil seluruh pelanggan terdaftar pada aplikasi
/pelanggan	POST	Endpoint untuk membuat akun pelanggan baru
/pelanggan/{id}	GET	Endpoint untuk mengambil akun pelanggan dengan id tertentu
/pelanggan/{id}	PUT	Endpoint untuk memperbarui akun pelanggan dengan id tertentu
/pelanggan/{id}	DELETE	Endpoint untuk menghapus (disable) akun pelanggan dengan id tertentu

3. Karyawan Endpoint, Tabel 3 memetakan controller karyawan pada endpoint REST API.

Tabel 3. Karyawan Endpoint Table

Path	Method	Description
/karyawan	GET	Endpoint untuk mengambil seluruh karyawan terdaftar pada aplikasi
/karyawan	POST	Endpoint untuk membuat akun karyawan baru
/karyawan/{id}	GET	Endpoint untuk mengambil akun karyawan dengan id tertentu
/karyawan/{id}	PUT	Endpoint untuk memperbarui akun karyawan dengan id tertentu
/karyawan/{id}	DELETE	Endpoint untuk menghapus (disable) akun karyawan dengan id tertentu
/karyawan/mitra/{mitraId}	GET	Endpoint untuk mengambil seluruh karyawan terdaftar pada aplikasi dan id mitra tertentu

4. Mitra Endpoint, Tabel 4 memetakan controller mitra pada endpoint REST API.

Tabel 4. Mitra Endpoint Table

Path	Method	Description
/mitra	GET	Endpoint untuk mengambil seluruh mitra terdaftar pada aplikasi
/mitra	POST	Endpoint untuk membuat akun mitra baru
/mitra/{id}	GET	Endpoint untuk mengambil mitra dengan id tertentu
/mitra/{id}	PUT	Endpoint untuk mengupdate mitra dengan id tertentu
/mitra/{id}	DELETE	Endpoint untuk menghapus (disable) mitra dengan id tertentu

5. Produk Endpoint, Tabel 5 memetakan controller produk pada endpoint REST API.

Tabel 5. Produk Endpoint Table

Path	Method	Description
/produk	GET	Endpoint untuk mengambil seluruh produk yang terdaftar pada aplikasi
/produk	POST	Endpoint untuk membuat produk baru pada mitra tertentu
/produk/{id}	GET	Endpoint untuk mengambil produk dengan id tertentu
Continued on next page		

Tabel 5 – continued from previous page

Path	Method	Description
/produk/{id}	PUT	Endpoint untuk memperbarui produk dengan id tertentu
/produk/{id}	DELETE	Endpoint untuk menghapus (disable) produk dengan id tertentu
/produk/mitra/{mitraId}	GET	Endpoint untuk mengambil produk dari id mitra tertentu

6. Pesanan Endpoint, Tabel 6 memetakan controller pesanan pada endpoint REST API.

Tabel 6. Pesanan Endpoint Table

Path	Method	Description
/pesanan	GET	Endpoint untuk mengambil seluruh pesanan
/pesanan	POST	Endpoint untuk membuat pesanan baru
/pesanan/{invoice}	GET	Endpoint untuk mengambil pesanan dengan invoice tertentu
/pesanan/{invoice}	PUT	Endpoint untuk memperbarui pesanan dengan invoice tertentu
/pesanan/{invoice}	DELETE	Endpoint untuk menghapus (disable) pesanan dengan invoice tertentu
/pesanan/mitra/{mitraId}	GET	Endpoint untuk mengambil seluruh pesanan dari id mitra tertentu

7. OrderList Endpoint, Tabel 7 memetakan controller orderlist pada endpoint REST API.

Tabel 7. OrderList Endpoint Table

Path	Method	Description
/orderlist	POST	Endpoint untuk membuat orderlist baru
/orderlist/{id}	GET	Endpoint untuk mengambil orderlist dengan id tertentu
/orderlist/{id}	PUT	Endpoint untuk memperbarui orderlist dengan id tertentu
/orderlist/{id}	DELETE	Endpoint untuk menghapus (disable) orderlist dengan id tertentu
/orderlist/invoice/{invoice}	GET	Endpoint untuk mengambil orderlist dengan invoice pesanan tertentu

8. Antrian Endpoint, Tabel 8 memetakan controller antrian pada endpoint REST API.

Tabel 8. Antrian Endpoint Table

Path	Method	Description
/antrian	POST	Endpoint untuk membuat antrian baru
/antrian/{id}	GET	Endpoint untuk mengambil antrian dengan id tertentu
/antrian/{id}	PUT	Endpoint untuk memperbarui antrian dengan id tertentu
/antrian/{id}	DELETE	Endpoint untuk menghapus (disable) antrian dengan id tertentu
/antrian/mitra/{mitraId}	GET	Endpoint untuk mengambil antrian pada mitra tertentu

9. Reviews Endpoint, Tabel 9 memetakan controller review pada endpoint REST API.

Tabel 9. Reviews Endpoint Table

Path	Method	Description
/reviews	GET	Endpoint untuk mengambil seluruh review
/reviews	POST	Endpoint untuk membuat review baru
/reviews/mitra/{mitraId}	GET	Endpoint untuk mengambil review pada mitra tertentu
/reviews/{mitraId}/{pelangganId}	GET	Endpoint untuk mengambil review pelanggan pada mitra tertentu
/reviews/{mitraId}/{pelangganId}	PUT	Endpoint untuk memperbarui review pelanggan pada mitra tertentu
/reviews/{mitraId}/{pelangganId}	DELETE	Endpoint untuk menghapus review pelanggan pada mitra tertentu

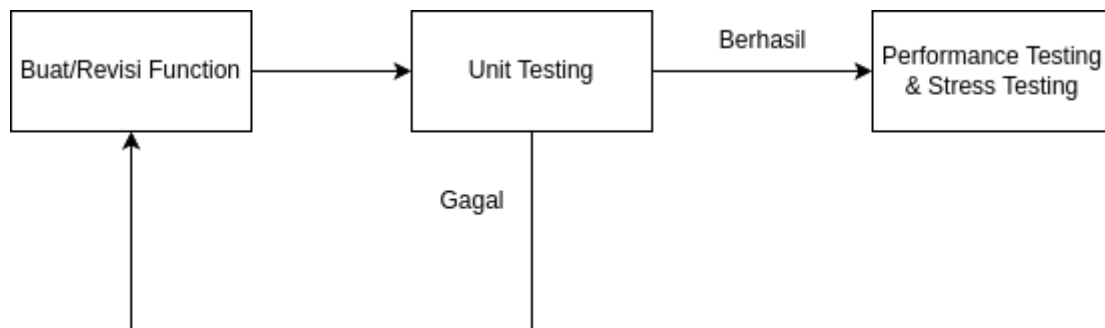
10. Image server endpoint, endpoint pada tabel 10 untuk menyajikan gambar ke client melalui REST API.

Tabel 10. Image server Endpoint Table

Path	Method	Description
/image/{fileName}	GET	Endpoint untuk menampilkan gambar dengan filename tertentu

3.4 Alur Pengujian

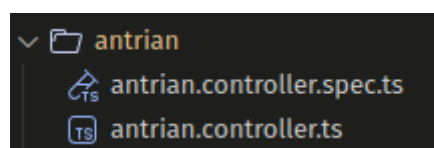
Pengujian menggunakan metode white box testing yaitu unit testing. Testing dilakukan pada setiap fungsi yang telah dibuat tanpa terhubung ke komponen lain. Setelah semua fungsi lulus unit testing, dilakukan performance dan load testing menggunakan library K6 pada node.js untuk mnegukur performansi aplikasi. Environment Testing menggunakan local build pada laptop penulis. Diagram alur pengujian dapat dilihat pada gambar 4.



Gambar 4. Alur Testing

3.4.1 Unit Testing

Pengujian dimulai dengan membuat file yang bernama component.spec.ts seperti terlihat pada gambar 5. Lalu pada file tersebut penulis mendeskripsikan test apa yang akan dibuat seperti contoh pada listing 11, pertama didefinisikan parameter yang diperlukan function lalu expect hasil keluaran fungsi sesuai dengan yang ditentukan.



Gambar 5. File spec.ts

Listing 11: Contoh Testing Menggunakan Jest

```
describe('canActivate', () => {
  it('should return true if user role is karyawan', async () => {
    const mockRequest = { user: { role: 'karyawan' } };
    const mockContext = { switchToHttp: () => ({ getRequest: () => mockRequest }) } as unknown
      as ExecutionContext;

    expect(await guard.canActivate(mockContext)).toBe(true);
  });

  it('should return false if user role is not karyawan', async () => {
    const mockRequest = { user: { role: 'admin' } };
    const mockContext = { switchToHttp: () => ({ getRequest: () => mockRequest }) } as unknown
      as ExecutionContext;

    expect(await guard.canActivate(mockContext)).toBe(false);
  });
});
```

3.4.2 Maintainability Testing

Pengujian terhadap aspek *maintainability* dari perangkat lunak dilakukan dengan menggunakan alat analisis statis bernama *SonarQube*. Alat ini memberikan penilaian berupa indeks rating yang dikategorikan dalam tingkatan A, B, C, D, dan E, yang masing-masing mencerminkan tingkat kualitas dari tiga aspek utama, yaitu *Security* (keamanan), *Reliability* (keandalan), dan *maintainability* (kemudahan pemeliharaan). Tingkatan A menunjukkan kualitas terbaik, sedangkan E menunjukkan kualitas terendah. Dengan menggunakan *SonarQube*, penulis dapat mengidentifikasi area dalam kode yang memerlukan perbaikan untuk meningkatkan kualitas perangkat lunak secara keseluruhan.

3.4.3 Performance Testing & Stress Testing

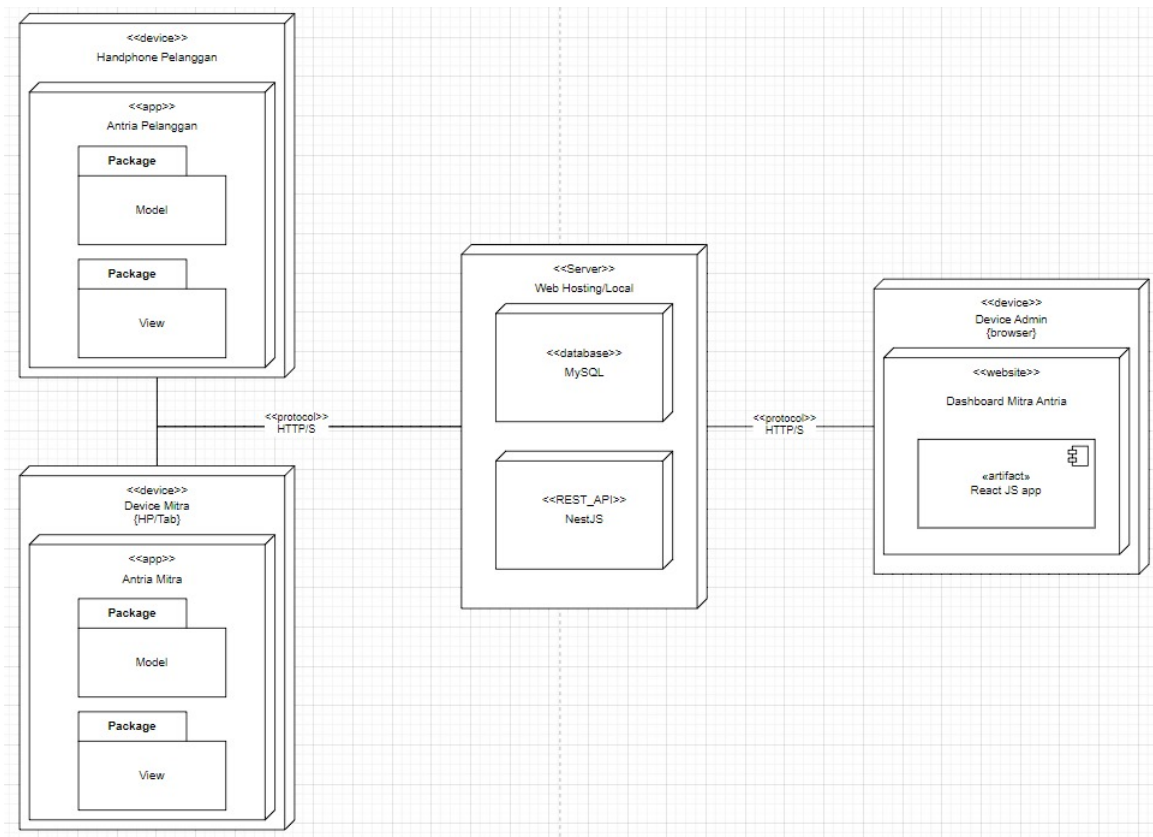
Pengujian performa dilakukan menggunakan K6 dengan cara melakukan request secara konkuren berdasarkan VU (Virtual User) yang di spesifikasikan. Pada listing 12, penulis mendefinisikan VU sebanyak 10, dengan stages pada detik 10 harus terdapat 100 VU yang melakukan, dan detik 20 terdapat 200 VU yang melakukan request, ini mengakibatkan fungsi akan menambah VU secara dinamis dalam 1 testing. Penulis juga mendefinisikan Request blocked tidak boleh lebih dari 30 milisecond, request connecting tidak boleh selama 100 milisecond untuk 90% request, durasi request tidak lebih dari 1 detik untuk 90% request, dan rate kegagalan request kurang dari 0,1% [11].

Listing 12: Konfigurasi dan threshold yang perlu dicapai pada K6

```
export const options = {
  vus: 10,
  thresholds: {
    http_req_blocked: ['p(100)<=30'],
    http_req_connecting: ['p(90)<100'],
    http_req_duration: ['p(90)<1000'],
    http_req_failed: ['rate<0.1'],
    http_req_receiving: ['p(90)<50'],
    http_req_sending: ['p(90)<10'],
  },
  tags: {
    environment: 'production',
  },
  stages: [
    { duration: '10s', target: 100 },
    { duration: '20s', target: 200 },
  ],
};
```

3.5 Deployment

Ketika backend selesai di kembangkan dan siap dipakai maka akan dilakukan deployment. Pada diagram 6 menjelaskan bagaimana server backend berkomunikasi dengan Aplikasi lain melalui protokol HTTPS dan memiliki database pusat yaitu di backend untuk sinkronisasi data dari satu aplikasi ke aplikasi lain.



Gambar 6. Deployment Diagram

4. Evaluasi

4.1 Hasil Pengujian

4.1.1 Unit Testing

Untuk unit testing, terdapat 2 Test suite pada setiap entity kecuali auth dimana auth memiliki 3 test suite yang ditotal sebanyak 21 test suite yang perlu dibuat dan dilakukan.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	96.69	100	96.55	96.32	
antrian	100	100	100	100	
antrian.controller.ts	100	100	100	100	
antrian.service.ts	100	100	100	100	
auth	100	100	100	100	
auth.controller.ts	100	100	100	100	
auth.guards.ts	100	100	100	100	
auth.service.ts	100	100	100	100	
constants.ts	100	100	100	100	
auth/dto	100	100	100	100	
loginMitra.dto.ts	100	100	100	100	
loginPelanggan.dto.ts	100	100	100	100	
image	100	100	100	100	
image.controller.ts	100	100	100	100	
image.service.ts	100	100	100	100	
karyawan	93.1	100	93.75	92.59	
karyawan.controller.ts	88.23	100	87.5	87.5	45-48
karyawan.service.ts	100	100	100	100	
mitra	94.59	100	93.33	94.02	
mitra.controller.ts	88.57	100	87.5	87.87	51-54
mitra.service.ts	100	100	100	100	
orderlist	100	100	100	100	
orderlist.controller.ts	100	100	100	100	
orderlist.service.ts	100	100	100	100	
pelanggan	92	100	92.3	91.3	
pelanggan.service.ts	100	100	100	100	
pelangganController.ts	87.87	100	83.33	87.09	47-50
pelanggan/dto	100	100	100	100	
createPelanggan.dto.ts	100	100	100	100	
pesanan	100	100	100	100	
pesanan.controller.ts	100	100	100	100	
pesanan.service.ts	100	100	100	100	
produk	88.23	100	87.5	86.66	
produk.controller.ts	78.94	100	77.77	77.77	37-40, 61-64
produk.service.ts	100	100	100	100	
review	100	100	100	100	
review.controller.ts	100	100	100	100	
review.service.ts	100	100	100	100	

Gambar 7. Hasil unit testing menggunakan jest

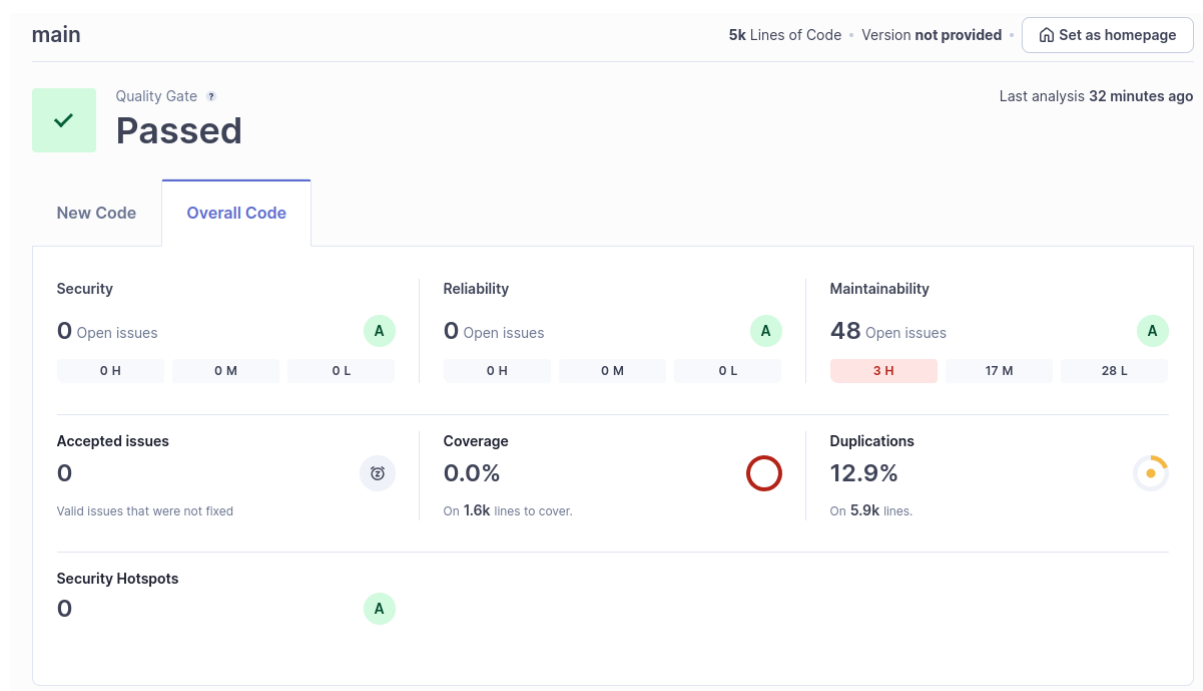
```
===== Coverage summary =====
Statements : 96.69% ( 585/605 )
Branches   : 100% ( 58/58 )
Functions  : 96.55% ( 140/145 )
Lines      : 96.32% ( 524/544 )

Test Suites: 21 passed, 21 total
Tests:       179 passed, 179 total
Snapshots:   0 total
Time:        39.923 s
```

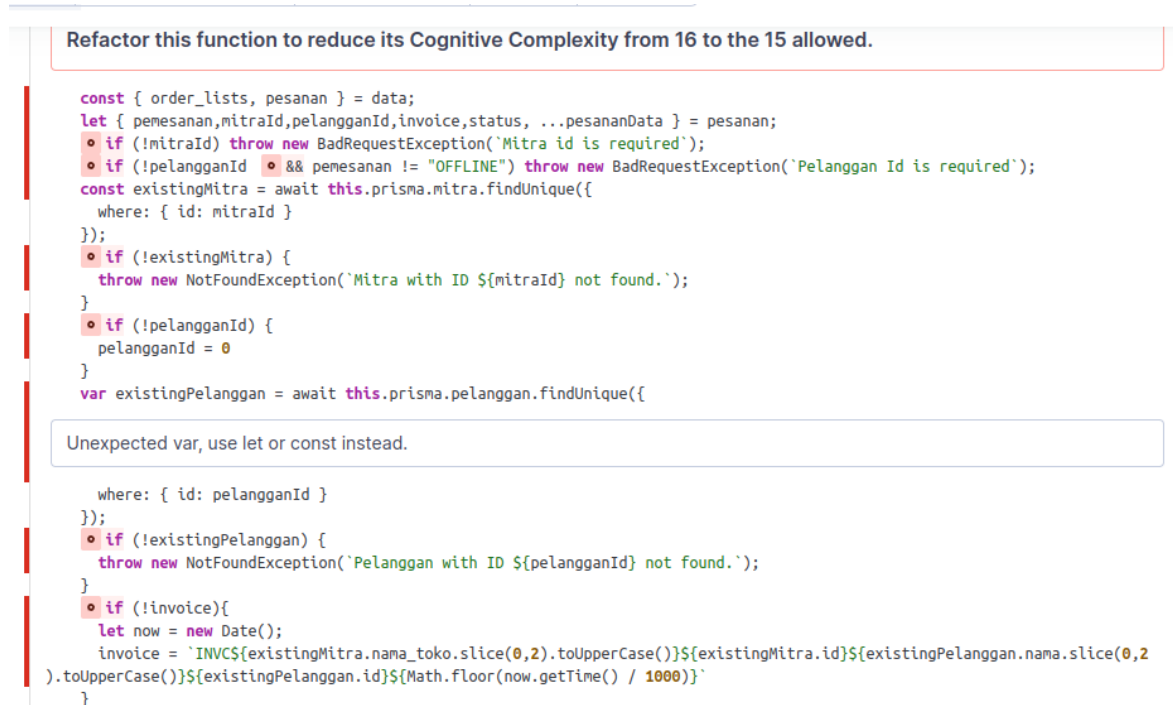
Gambar 8. Coverage Summary

4.1.2 Maintainability Testing

Pada bagian ini, penulis melakukan pengujian menggunakan *SonarQube* dan *Sonar Scanner* untuk menganalisis kode program yang telah dibuat. Proses analisis ini bertujuan untuk mengevaluasi kualitas kode dengan mengidentifikasi potensi masalah dalam aspek *Security*, *Reliability*, dan *maintainability*. Setelah analisis selesai, *SonarQube* memberikan skor indeks yang mencerminkan tingkat kualitas kode dalam kategori-kategori tersebut. Skor ini membantu penulis memahami area mana yang perlu ditingkatkan untuk memastikan kode yang lebih aman, andal, dan mudah dipelihara.



Gambar 9. Hasil Analysis SonarQube



Gambar 10. Cognitive Complexity Melebihi treshold untuk fungsi createPesanan

4.1.3 Performance Testing

Pada bagian performance testing, penulis menguji beberapa fitur yang mewakili method HTTP GET, POST, dan PUT. Untuk GET penulis menguji fungsi untuk mengambil seluruh mitra, untuk POST penulis menguji fungsi untuk Register user, untuk PUT penulis menguji fungsi update user.


```

      .io

execution: local
script: ./test/loadtest.spec.ts
output: -

scenarios: (100.00%) 1 scenario, 200 max VUs, 1m0s max duration (incl. graceful stop):
    * default: Up to 200 looping VUs for 30s over 2 stages (gracefulRampDown: 30s, gracefulStop: 30s)

✓ Status was 201:

checks.....: 100.00% ✓ 3605      ✗ 0
data_received.....: 41 MB   1.3 MB/s
data_sent.....: 1.5 MB   47 kB/s
✓ http_req_blocked.....: avg=26.36µs  min=3.23µs  med=5.93µs   max=8.33ms  p(90)=8.56µs  p(95)=266.48µs
✓ http_req_connecting.....: avg=14.23µs  min=0s     med=0s       max=8.17ms  p(90)=0s     p(95)=179.02µs
✓ http_req_duration.....: avg=9.73ms   min=5.06ms  med=6.69ms   max=198.54ms p(90)=11.91ms p(95)=17.86ms
    { expected_response:true }...: avg=9.73ms   min=5.06ms  med=6.69ms   max=198.54ms p(90)=11.91ms p(95)=17.86ms
✓ http_req_failed.....: 0.00% ✓ 0      ✗ 3605
✓ http_req_receiving.....: avg=108.39µs min=50.08µs med=103.34µs max=802.97µs p(90)=138.69µs p(95)=152.95µs
✓ http_req_sending.....: avg=34.47µs  min=14.65µs med=31.2µs   max=907.33µs p(90)=44.33µs p(95)=60.62µs
http_req_tls_handshaking.....: avg=0s       min=0s     med=0s       max=0s      p(90)=0s     p(95)=0s
http_req_waiting.....: avg=9.59ms   min=4.93ms  med=6.54ms   max=198.39ms p(90)=11.75ms p(95)=17.67ms
http_reqs.....: 3605 116.308379/s
iteration_duration.....: avg=1.01s    min=1s     med=1s       max=1.19s   p(90)=1.01s   p(95)=1.01s
iterations.....: 3605 116.308379/s
vus.....: 4 min=4 max=199
vus_max.....: 200 min=200 max=200

running (0m31.0s), 000/200 VUs, 3605 complete and 0 interrupted iterations
default ✓ [=====] 000/200 VUs 30s

```

Gambar 13. Hasil performance testing get semua mitra (GET)

[illegible]

```
running (0m31.0s), 000/200 VUs, 3610 complete and 0 interrupted iterations
default ✓ [=====] 000/200 VUs 30s
```

Gambar 14. Hasil performance testing get mitra berdasar id (GET)

```

      .io
execution: local
script: ./test/loadtest.spec.ts
output: -

scenarios: (100.00%) 1 scenario, 200 max VUs, 1m0s max duration (incl. graceful stop):
    * default: Up to 200 looping VUs for 30s over 2 stages (gracefulRampDown: 30s, gracefulStop: 30s)


✓ Status was 201:

checks.....: 100.00% ✓ 3521          X 0
data_received.....: 2.2 MB   71 kB/s
data_sent.....: 1.7 MB   55 kB/s
✓ http_req_blocked.....: avg=37.02µs min=2.63µs med=6.82µs max=21.28ms p(90)=11.16µs p(95)=291.84µs
✓ http_req_connecting....: avg=21.6µs min=0s med=0s max=20.95ms p(90)=0s p(95)=195.99µs
✓ http_req_duration.....: avg=29.97ms min=4.86ms med=18.44ms max=564.12ms p(90)=54.71ms p(95)=77.4ms
{ expected_response:true }...: avg=29.97ms min=4.86ms med=18.44ms max=564.12ms p(90)=54.71ms p(95)=77.4ms
✓ http_req_failed.....: 0.00% ✓ 0           X 3521
✓ http_req_receiving....: avg=119.29µs min=42.06µs med=105.64µs max=2.83ms p(90)=153.48µs p(95)=187.65µs
✓ http_req_sending.....: avg=52.07µs min=17.03µs med=43.78µs max=1.76ms p(90)=70.2µs p(95)=92.09µs
http_req_tls_handshaking....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
http_req_waiting.....: avg=29.8ms min=4.73ms med=18.27ms max=563.78ms p(90)=54.57ms p(95)=77.26ms
http_reqs.....: 3521     113.675404/s
iteration_duration.....: avg=1.03s min=1s med=1.01s max=1.56s p(90)=1.05s p(95)=1.07s
iterations.....: 3521     113.675404/s
vus.....: 2         min=2        max=199
vus_max.....: 200       min=200       max=200

running (0m31.0s), 000/200 VUs, 3521 complete and 0 interrupted iterations
default ✓ [=====] 000/200 VUs 30s
```

```
running (0m31.0s), 000/200 VUs, 3521 complete and 0 interrupted iterations
default ✓ [=====] 000/200 VUs 30s
```

Gambar 15. Hasil performance testing update user berdasar id (PUT)

4.2 Analisis Hasil Pengujian

4.2.1 Unit Testing

Semua test case pada unit testing lolos uji atau pass dengan code coverage 96,32% dengan LOC (Line Of Code) sekitar 524 dari 544 LOC. Jika dilihat pada gambar 7, LOC yang tidak teruji terdapat pada file karyawan, mitra, pelanggan, dan produk. Kode yang tidak teruji berupa interceptor untuk mendeteksi file gambar pada multipart request, fungsi tersebut tidak dapat diuji pada unit testing dikarenakan perlunya request autentik multipart dan tidak bisa di simulasikan secara efektif.

4.2.2 Performance Testing

Performance testing berfungsi untuk melihat seberapa kuat backend untuk melayani banyak request dalam secara bersamaan. Dilihat dari hasil testing ada beberapa hal yang ditemukan. Dari 5 Testing, 2 diantaranya belum memenuhi spesifikasi yang telah ditetapkan. Pada gambar 12 menampilkan hasil testing dari fungsi login user. Tertulis rata rata durasi sebesar 3 detik dengan delay terendah berada di 126 milisecond dan delay tertinggi berada di 6 detik, hal ini diakibatkan oleh fungsi login yang membanding satu string password dengan versi bcrypt, dengan cara string password di enkripsi terlebih dahulu lalu di bandingkan, hal ini memakan waktu yang mengakibatkan delay tinggi saat server over capacity. Lalu pada gambar 11 memiliki hasil yang mirip dengan fungsi login, hal ini dikarenakan fungsi register melakukan insert database dimana pada orm Prisma sedikit lebih lambat dibanding edit dan view data. Pada fungsi get all mitra pada gambar 13, rata-rata delay yang didapatkan tidak lambat yaitu sebesar 9,7 milisecond, sedangkan untuk get mitra by id pada gambar 14 sebesar 5 milisecond. Untuk fungsi terakhir yaitu update user by id pada gambar 15, rata-rata duration yang di dapat sebesar 30 milisecond.

4.2.3 Maintainability Testing

Pada gambar 9 menunjukkan hasil analisis dari SonarQube bahwa kualitas kode secara keseluruhan memenuhi standar yang ditetapkan. Namun, ada beberapa area yang perlu diperhatikan terkait *maintainability*. Persentase *Duplications* adalah 12.9%, yang menunjukkan adanya duplikasi kode dalam 5.9k baris kode. Meskipun ini tidak secara langsung mempengaruhi *maintainability* dalam skala besar, pengurangan duplikasi dapat meningkatkan kualitas kode secara keseluruhan. Pada aspek *maintainability*, SonarQube memberikan penilaian A, yang menunjukkan tingkat *maintainability* yang baik. Namun, pada gambar 10, ada satu isu khusus yang perlu diperbaiki: fungsi *createPesanan*. Fungsi ini memiliki *Cognitive Complexity* sebesar 16, yang melebihi batas yang diizinkan yaitu 15. Untuk meningkatkan *maintainability*, disarankan untuk merestrukturisasi fungsi ini agar *Cognitive Complexity* dapat dikurangi, sehingga mempermudah pemahaman dan pemeliharaan kode di masa depan.

5. Kesimpulan

Berdasarkan implementasi dan perancangan sistem menggunakan NestJs dan Prisma, dapat diambil beberapa kesimpulan terkait Anti Pattern, Keamanan, dan Maintainabilitas.

1. Anti Pattern

Penting untuk menghindari Anti Pattern dalam pengembangan aplikasi, yaitu praktik yang tidak dianjurkan atau kesalahan umum yang sering dilakukan. Dalam sistem yang dikembangkan, beberapa langkah yang telah diambil untuk menghindari Anti Pattern meliputi:

- **Penamaan Endpoint yang konsisten**
- **Separation of Concerns:** Memisahkan setiap komponen dalam aplikasi (Middleware, Guards, Interceptor, Controller, dan Service) sesuai dengan tanggung jawab masing-masing.
- **Menghindari Over-fetching dan Under-fetching**

2. keamanan

Keamanan adalah aspek penting dalam pengembangan aplikasi, beberapa langkah yang diambil untuk memastikan keamanan sistem meliputi:

- **JWT Authentication:** Penggunaan JWT untuk otentikasi dan otorisasi pengguna, memastikan hanya pengguna yang valid dapat mengakses endpoint tertentu.
- **Data Validation:** Validasi data pada setiap layer (Controller, Service) untuk memastikan integritas dan keamanan data.

- **Encryption:** Penggunaan bcrypt untuk meng-hash password pengguna sebelum disimpan di database.

3. Maintainabilitas

Maintainabilitas adalah kemampuan sistem untuk dapat dipelihara dan dikembangkan lebih lanjut dengan mudah dengan cara menggunakan Prisma sebagai ORM untuk manajemen database yang lebih mudah dan terstruktur.

5.1 Saran

Untuk meningkatkan performa dan skalabilitas sistem, terdapat beberapa hal yang perlu diperhatikan dan ditingkatkan, antara lain:

1. Performa

- **Tidak menggunakan Prisma ORM:** Penggunaan Prisma dapat meningkatkan maintainability dengan memangkas performa, dengan tidak menggunakan Prisma dan hanya menggunakan raw sql, dapat meningkatkan performa akses database.

2. Skalabilitas

- **Microservices Architecture:** Pertimbangkan untuk memecah aplikasi menjadi beberapa layanan mikro yang independen. Hal ini dapat meningkatkan skalabilitas dan memudahkan pengembangan serta pemeliharaan setiap komponen.
- **Kubernetes and Containerization:** Gunakan teknologi container seperti Docker dan orkestrator container seperti Kubernetes untuk memudahkan pengelolaan dan skala aplikasi. Teknologi ini memungkinkan aplikasi dijalankan di berbagai lingkungan dengan konsistensi yang tinggi.

Dengan meningkatkan aspek-aspek ini, sistem diharapkan dapat beroperasi dengan lebih efisien dan mampu menangani pertumbuhan pengguna dan data di masa mendatang.

Daftar Pustaka

- [1] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza. A large-scale empirical study on linguistic antipatterns affecting apis. In *Soft System Stakeholder Analysis Methodology*, pages 25–35. Institute of Electrical and Electronics Engineers Inc., 11 2018.
- [2] F. S. Alshraiedeh and N. Katuk. A uri parsing technique and algorithm for anti-pattern detection in restful web services. *International Journal of Web Information Systems*, 17:1–17, 1 2021.
- [3] M. I. Beer and M. F. Hassan. Adaptive security architecture for protecting restful web services in enterprise computing environment. *Service Oriented Computing and Applications*, 12:111–121, 6 2018.
- [4] M. Ghazal, R. Hamouda, and S. Ali. A smart mobile system for the real-time tracking and management of service queues. *International Journal of Computing and Digital Systems*, 5:305–313, 7 2016.
- [5] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck. Best practices for the design of restful web services. In *International Conferences of Software Advances (ICSEA)*, pages 392–397, 2015.
- [6] K. Gos and W. Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVI-th International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.
- [7] P. Jatkiewicz and S. Okrój. Differences in performance, scalability, and cost of using microservice and monolithic architecture. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1038–1041, 2023.
- [8] Y. L. Khong, B. C. Ooi, K. E. Tan, S. A. B. Ibrahim, and P. L. Tee. E-queue mobile application. In *SHS Web of Conferences*, volume 33, page 00033. EDP Sciences, 2017.
- [9] M. Lorenz, J.-P. Rudolph, G. Hesse, M. Uflacker, and H. Plattner. Object-relational mapping revisited-a quantitative study on the impact of database technology on o/r mapping strategies. *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.
- [10] K. Mysliwiec. Nestjs documentation.
- [11] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.
- [12] Q. Odeniran, H. Wimmer, and C. M. Rebman. Node.js or php? determining the better website server backend scripting language. *Issues in Information Systems*, 24:328–341, 2023.
- [13] A. D. Pham. Developing back-end of a web application with nestjs framework: Case: Integrify oy’s student management system. *Theseus*, 2020.
- [14] I. Prisma Data. Prismajs documentation.
- [15] A. Rahmatullo, A. P. Aldya, and M. N. Arifin. Stateless authentication with json web tokens using rsa-512 algorithm. *JURNAL INFOTEL*, 11(2):36–42, 2019.
- [16] P. Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.
- [17] H. Shah and T. R. Soomro. Node. js challenges in implementation. *Global Journal of Computer Science and Technology*, 17(2):73–83, 2017.
- [18] M. N. Uddin, M. Rashid, M. Mostafa, S. Salam, N. Nithe, and S. Z. Ahmed. Automated queue management system, 2016.
- [19] G. William, R. Anthony, and J. Purnama. Development of nodejs based backend system with multiple storefronts for batik online store. *ACM International Conference Proceeding Series*, 2020. Cited by: 0.
- [20] D. Zmaranda, L.-L. Pop-Fele, C. Győrödi, R. Győrödi, and G. Pecherle. Performance comparison of crud methods using net object relational mappers: A case study, 2020.

Lampiran

Lampiran dapat berupa detil data dan contoh lebih lengkapnya, data-data pendukung, detail hasil pengujian, analisis hasil pengujian, detail hasil survey, surat pernyataan dari tempat studi kasus, screenshot tampilan sistem, hasil kuesioner dan lain-lain.