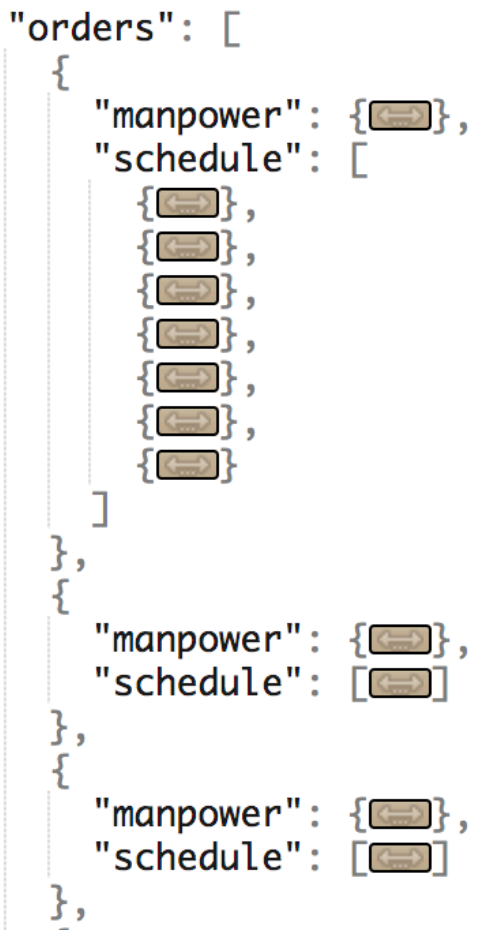
# BAB IV
# IMPLEMENTASI DAN PENGUJIAN

## 4.1 Alur Proses

Pembuatan aplikasi penjadwalan bermula dari membuat flow atau alur dimana data dalam database dapat diolah sehingga menampilkan data yang sesuai. Data ini diambil dari database server lewat sambungan dari *service*. Membuat koneksi adalah hal pertama yang dilakukan setelah mengetahui letak database dan posisinya terhadap *service*. Kemudian dari data yang telah tersedia dilakukan sortir berdasar tanggal trip akan dilaksanakan. Proses ini dilakukan secara manual dengan cara *coding* dan dibantu oleh NPM, yang memiliki library Moment.js. Moment.js bekerja dengan cara mengekstraksi waktu dengan format yang spesifik. Output yang dihasilkan memiliki banyak ragam. Pada *code* yang dibuat, akan dipilih salah satu jenis output yang paling sesuai sebelum memprosesnya lebih jauh.

Setelah data selesai diseleksi berdasarkan waktu, maka proses selanjutnya yakni mengambil hasil data dari *database*. Proses ini dibantu oleh *library mongoose*. Dengan menggunakan *mongoose*, mengambil data dengan ketentuan tertentu jadi lebih mudah. Data yang berhasil diambil dapat segera diproses lebih lanjut.

Format data yang diinginkan merupakan jadwal *driver* per minggu. Yang artinya, dalam kurun waktu satu minggu, jumlah data yang ditampilkan akan menyesuaikan berapa banyak *driver* yang tersedia. Data yang ada pada proses ini adalah data order dari tiap-tiap *driver* dalam kurun waktu satu minggu. Maka dari itu, enkapsulasi data yang sesuai akan seperti gambar dibawah :



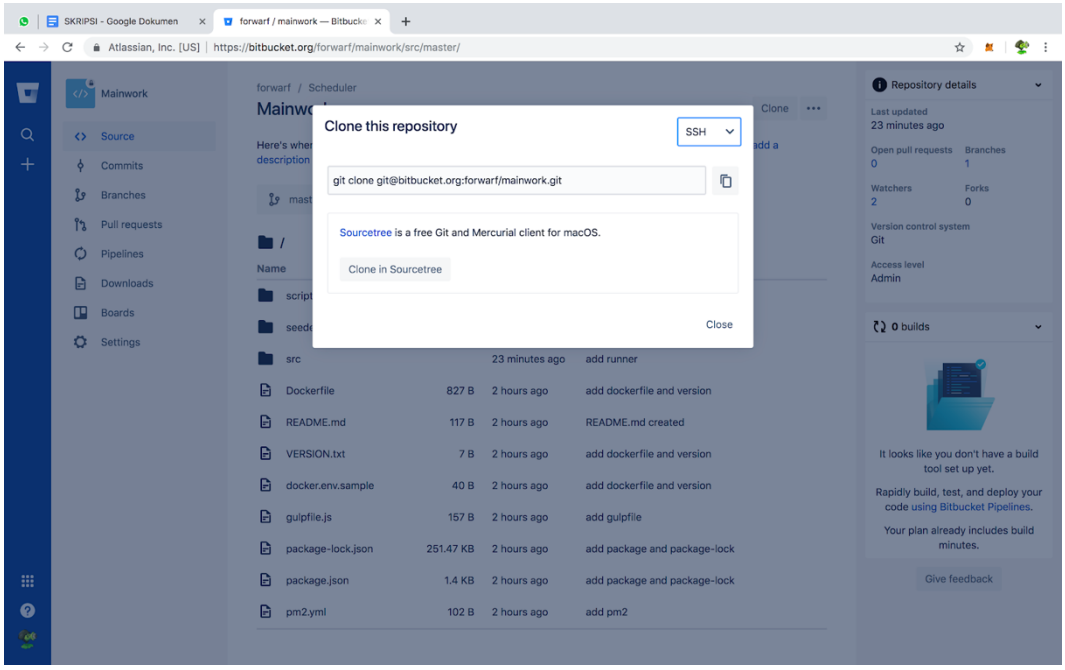

*Gambar 4. 1. Struktur Enkapsulasi Data yang Diinginkan*

Data seperti gambar diatas akan terus berulang sebanyak jumlah *driver*. Setiap *driver* akan memiliki *schedule* dengan isi yang berbeda satu sama lain. Jumlah order dan tipe order akan berbeda tergantung order yang telah dipesan *client*. Setelah semua data ditata sedemikian rupa, program akan mengirim data tersebut ke *frontend* dalam bentuk JSON.

## 4.2 Inisiasi Service

Tahap pertama dalam serangkaian proses pembuatan aplikasi penjadwalan adalah inisiasi *service*. Ketika *service* dibangun dan program mulai dirakit, proses tersebut terjadi di sebuah mesin seperti komputer atau laptop. Di dalam laptop, perlu disediakan alokasi memori agar pengembangan *service* tersimpan. Demi

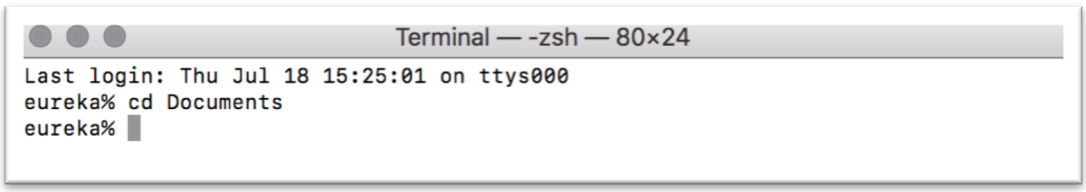
memastikan ini, dibuatlah sebuah proyek menggunakan *text editor*. Proses ini diawali dengan melakukan clone proyek utama.




*Gambar 4. 2. Clone Proyek dari Bitbucket*

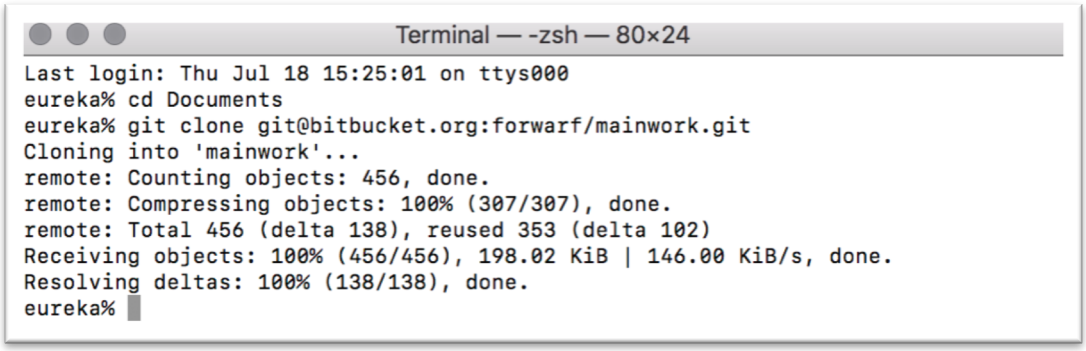
Akses terhadap proyek utama hanya dapat diperoleh dari izin pemilik proyek. Ketika berhasil didapat, yang perlu dilakukan cukup pergi menuju repositori dalam *cloud* yang disebut dan melakukan log in untuk mendapat link clone. Dalam kasus ini, repositori yang digunakan adalah Bitbucket.

Clone dilakukan di dalam terminal, karena itu yang pertama harus dilakukan adalah memastikan direktori terminal kita berada di alokasi memori yang telah ditentukan sebelumnya. Perintah yang dilakukan dalam terminal seperti berikut :



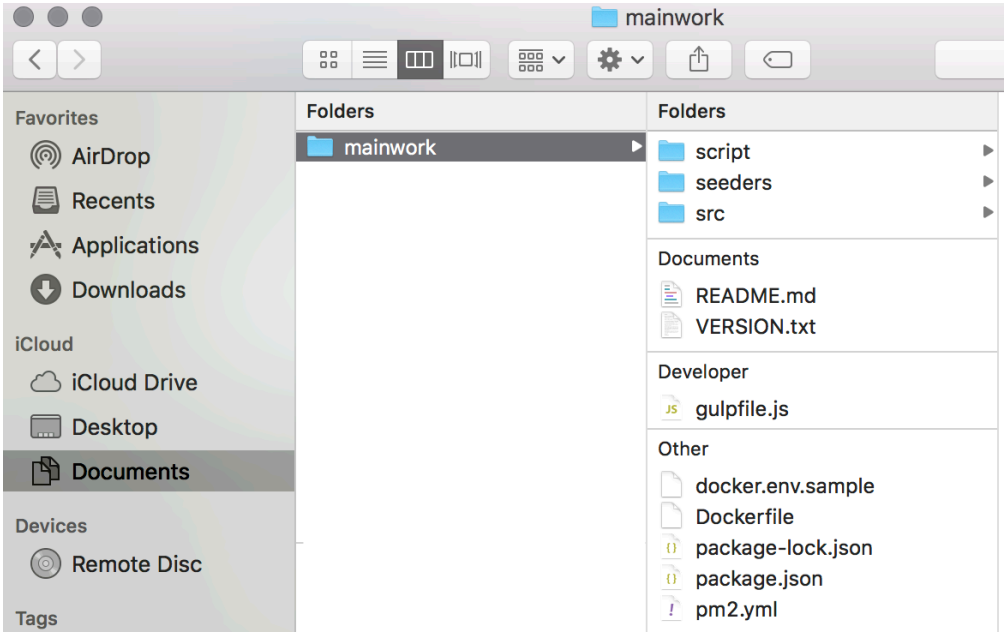

*Gambar 4. 3. Merubah Direktori Terminal Men Document*

Pada gambar diatas, direktori terminal berganti dari ~ menjadi ~/Documents, yakni tempat proyek akan disimpan. Selanjutnya cukup *copy url* yang didapat dari Bitbucket kemudian dieksekusi ke dalam terminal. Lama proses ini berbeda-beda tergantung dari besar proyek dan cepat atau tidak koneksi internet.




*Gambar 4. 4. Cloning Proyek dari Terminal*

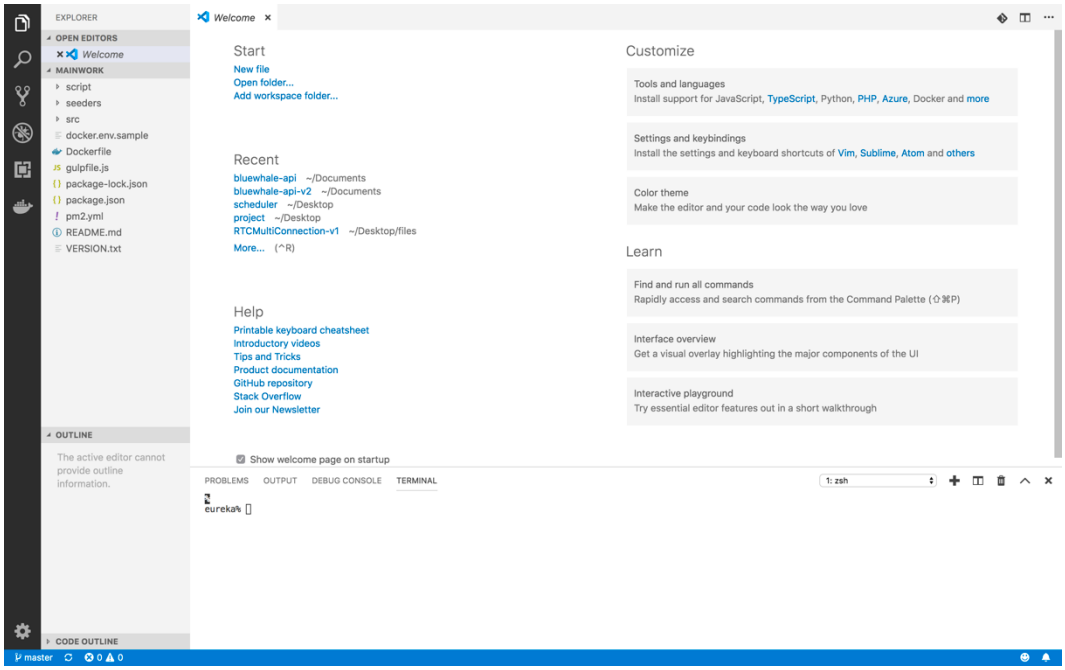
Ketika proses clone selesai, lebih baik dilakukan cek ulang menuju direktori proyek untuk memastikan proyek telah tersimpan dengan benar di dalam mesin yang digunakan. Proses pengecekan ini bertujuan mengantisipasi kerusakan file yang dapat mengakibatkan kerja developer tidak tersimpan atau rusak.




*Gambar 4. 5. Proyek Tersimpan di Direktori Dokumen*

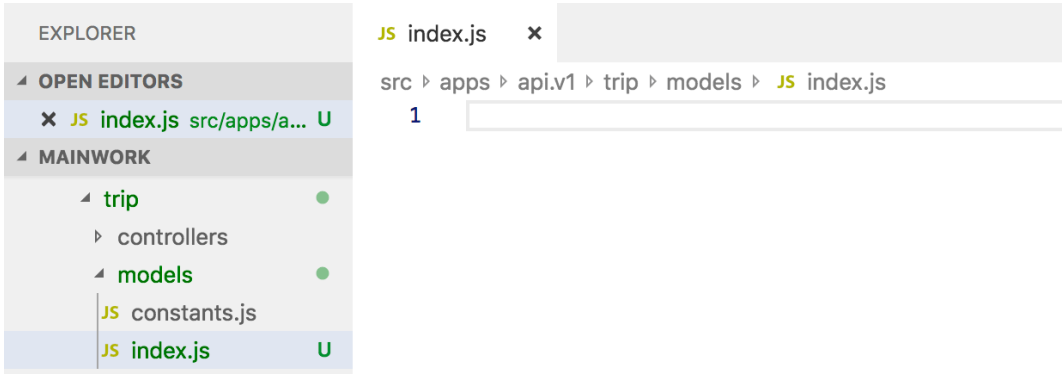
## 4.3 Inisiasi Program

Ketika Service tersimpan ke dalam laptop, proses pembuatan program baru dapat mulai dikerjakan. Pengerjaan dilakukan dengan membuka proyek menggunakan Visual Studio Code.



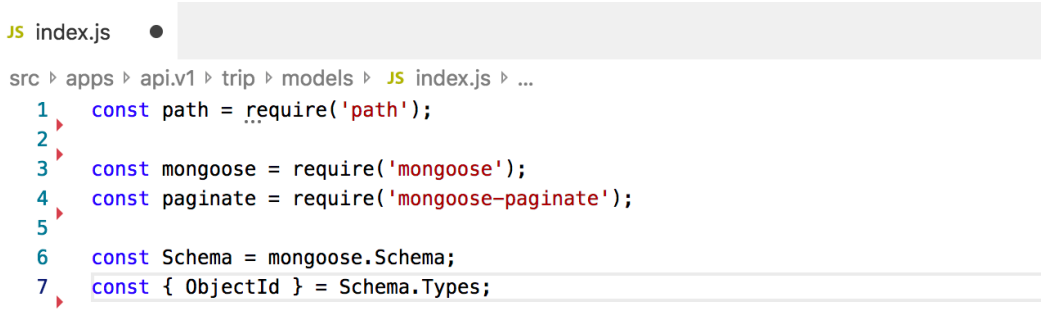

*Gambar 4. 6. Tampilan Proyek Dibuka dari Visual Studio Code*

Setelah terbuka, buat file berisi skema mongo yang akan digunakan. File ini disimpan di dalam *folder* src/apps/app.v1/trip/models dan diberi nama index.js.




*Gambar 4. 7. Membuat file index.js*

Kemudian di dalam file dilakukan deklarasi variabel agar terhubung dengan database yang diakses oleh service. Hal ini dibantu oleh package path dan mongoose, jadi deklarasi yang dibutuhkan seperti gambar dibawah :




*Gambar 4. 8. Membuat Koneksi Menuju Database*

Sesudah koneksi terbuka, dilakukan deklarasi skema berdasarkan *collection* dan *field*. Nama dan tipe data yang digunakan dalam skema tidak akan berubah untuk setiap *document* yang dimuat di dalam database. Berikut rincian code yang dibuat :

```
const TripSchema = new mongoose.Schema({
    _type: { type: String, default: 'Trip' },
    order: {
        type: ObjectId,
        ref: 'Order', // Data diambil dari Order
        index: true,
    },
  alarms: { // Set alarm untuk pengingat driver
    safe: { type: Number, default: 120 },
    alert: { type: Number, default: 60 },
  },
    vehicleOptions: {
        vehicleModel: {
```

```
                type: ObjectId,
                ref: 'VehicleModel', // Data diambil dari
VehicleModel
                index: true,
            },
            seat: String,
            luggage: String,
        },
        manPowerOptions: {
            language: { // Opsi bahasa yang dikuasai driver
                type: String,
                enum: [
                    LANGUAGE.ENGLISH,
                    LANGUAGE.MANDARIN,
                    LANGUAGE.BAHASA,
                    LANGUAGE.TAMIL,
                    LANGUAGE.OTHERS,
                ],
            },
            race: { // Ras driver
                type: String,
                enum: [
                    RACE.CHINESE,
                    RACE.INDIAN,
                    RACE.MALAY,
                    RACE.CAUCASIAN,
                    RACE.OTHERS,
```

```
            ],
        },
    },
    remarks: String, // Didapat dari description
    deploymentStatus: { // Status trip sekarang
        type: String,
        enum: [
            DEPLOYMENT_STATUS.UNACK,
            DEPLOYMENT_STATUS.ACK,
            DEPLOYMENT_STATUS.DELIVERED,
            DEPLOYMENT_STATUS.ONGOING,
            DEPLOYMENT_STATUS.EN_ROUTE,
            DEPLOYMENT_STATUS.ARRIVED,
        ],
    },
alarmStatus: { // Apabila driver melewatkan alarm
  type: String,
  enum: [
    ALARM_STATUS.MISSED_1,
    ALARM_STATUS.MISSED_2,
    ALARM_STATUS.RESPONDED,
    ALARM_STATUS.READY,
            ALARM_STATUS.NONE,
  ],
  default: ALARM_STATUS.NONE,
},
```

```
        tripType: { // Tipe trip yang disediakan
            type: String,
            enum: [
                TRIP_TYPE.DISPOSAL,
                TRIP_TYPE.NORMAL_TRANSFER,
                TRIP_TYPE.ARRIVAL_TRANSFER,
                TRIP_TYPE.DEPARTURE_TRANSFER,
            ],
        },
       guest: {
         type: [], // Berisi alamat email
       },
           numberOfGuest: {
               type: Number,
               default: 0,
           },


     // Untuk trip bertipe transfer dan disposal
     pickupDateTime: Date, // Merupakan data kapan transfer
dimulai
           pickupAddress: String,
           pickupLocation: [Number],


           destinationAddress: String,
           destinationLocation: [Number],
```

```
        arriveDateTime:  Date, //  Dari  bandara  menuju
manapun
        departureDateTime:  Date, //  Dari  manapun  menuju
bandara

        endDestinationAddress: String,
        endDestinationLocation: [Number],

        dropDateTime: Date, // Untuk trip disposal

    // Pilihan waktu tambahan
    // Digunakan ketika driver selesai mengerjakan trip
    // Kemudian client ingin memberikan bonus kepada driver
    additionalDisposalTime: {
       type: Number,
       default: 0,
     },

        issueType: {
            type: String,
            enum: [
                ISSUE_TYPE.CANT_FIND,
            ],
        },
        issueMessage: String,
     issueStatus: {
       type: String,
```

```
        enum: [
          ISSUE_STATUS.READ,
          ISSUE_STATUS.DONE,
        ],
      },

       orderStatus: {
           type: String,
           enum: [
               ORDER_STATUS.CONFIRMED,
               ORDER_STATUS.UNCONFIRMED,
               ORDER_STATUS.REJECT,
           ],
           default: ORDER_STATUS.UNCONFIRMED,
       },
     rejectReason: String, // Reject dari operator

       isRequestCancel: { type: Boolean, default: false }, // Cancel dari client
       manPowerVehicle: {
           manPower: {
               type: ObjectId,
               ref: 'ManPower',
               index: true,
           },
           vehicleManagement: {
               type: ObjectId,
```

```
            ref: 'VehicleManagement',
            index: true,
        },
    },

// Paging gambar
pagingFlightNumber: String, // Nomer penerbangan
 pagingImage: String, // Gambar atau link gambar
 pagingCustomerName: String,

 signatureImageUrl: String, // Tanda tangan client

 isCompleteOrder: { type: Boolean, default: false },

    lock: { // Status terkunci dari operator
        by: {
            type: ObjectId,
            ref: 'User',
            index: true,
        },
        status: { type: Boolean, default: false },
    },

 feedback: {
   toClient: { // Feedback dari driver untuk client
     message: String,
```

```
      rating: Number,
    },
    toDriver: { // Feedback dari driver untuk client
      message: String,
      rating: Number,
    },
  },
  reports: [String],
  rating: { type: Number, default: 0 },
    createdAt: { type: Date, default: Date.now },
    updatedAt: { type: Date, default: Date.now },
}, {
    collection: 'trips',
});
```

Ketika program pertama kali dijalankan dan koneksi ke database dibuat, MongoDB akan membaca skema lalu menerjemahkan menjadi collection dan fields. *Code* diatas dibuat berdasarkan kebutuhan data trip yang ditentukan oleh pemilik proyek. Namun demikian, data ini tidak akan bisa dipakai diluar file sebelum didaftarkan ke dalam *life-cycle* Express.js. Setelah melakukan input code, fungsi yang dapat diakses dideklarasi ke dalam module.exports.

```
module.exports = mongoose.model('Trip', TripSchema);
```

*Gambar 4. 9. Deklarasi Skema Agar Bisa Dipakai di Luar File*

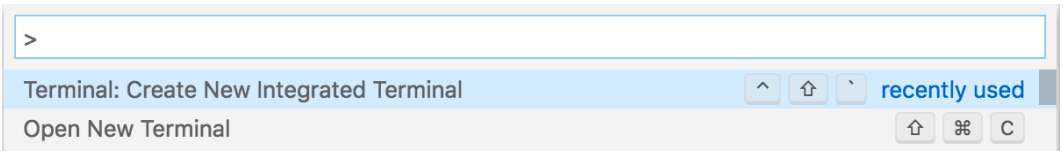
Pada file index.js, hanya terdapat satu fungsi yang terdaftar. Dengan *code* di atas, file lain dalam proyek bisa mengakses fungsi dengan memanggil Trip dari TripSchema.

## 4.4 Instalasi NPM dan Package yang Dibutuhkan

Dalam membuat sebuah *service*, banyak hal yang harus diperhitungkan sebelumnya. Bagaimana *service* dapat menopang jalan aplikasi, serta bagaimana *service* menopang program-program yang berjalan di dalamnya. Untuk mendapat hasil yang diinginkan, merupakan sebuah kewajaran jika *service* berdiri ditopang oleh *framework*. Node.js memiliki *framework* populer yakni Express.js atau biasa disebut *express*. Untuk dapat mengaplikasikan *express*, dibutuhkan *package manager* yakni NPM.

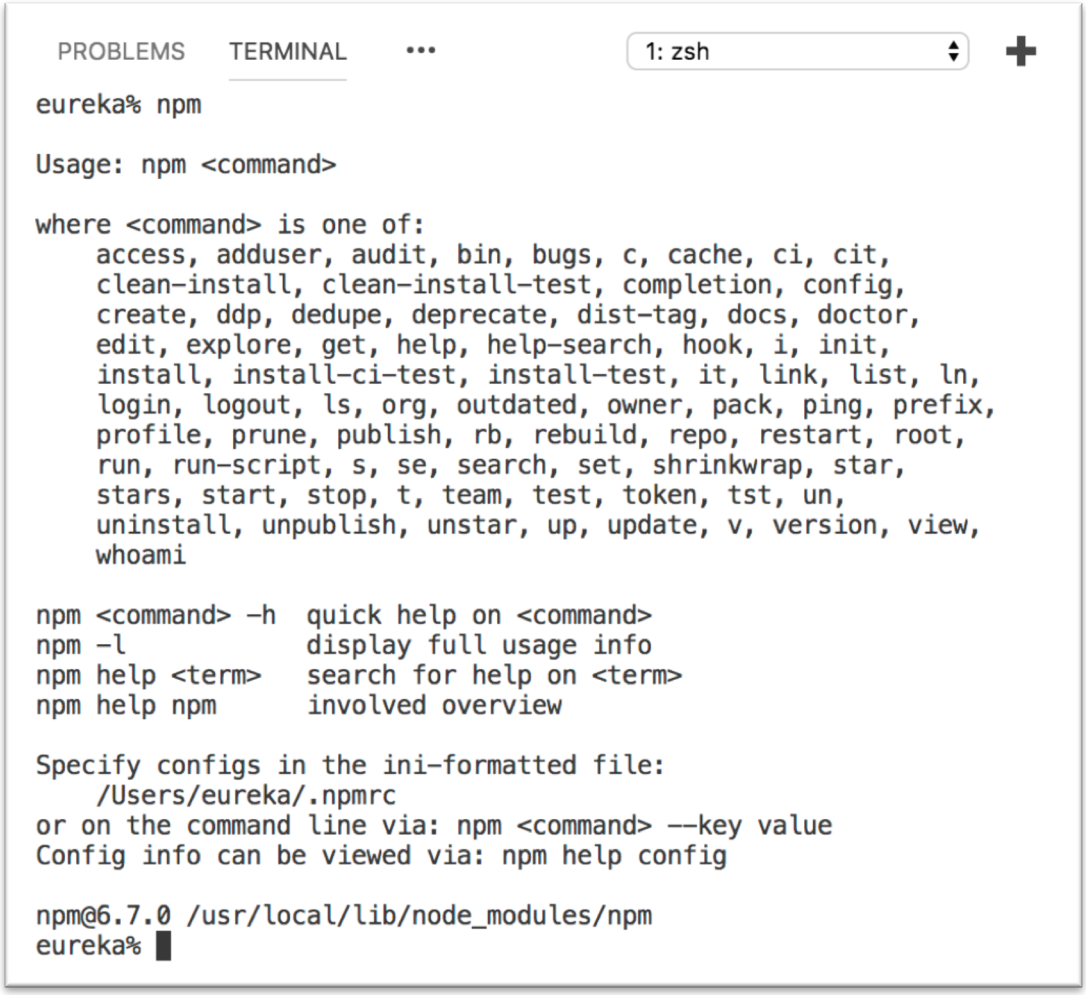
NPM didistribusikan bersamaan dengan Node. Ketika sebuah perangkat komputer mengunduh Node, secara otomatis NPM akan terinstall secara bersamaan ( NPM, 2018 ). Node dan NPM dapat di download melalui website resmi *https://nodejs.org/en/download* yang merupakan kolaborasi antara Linux dan Node.js Foundation.

Untuk menggunakan NPM dibutuhkan program berbasis *console* seperti terminal, cmd atau *iterm*. Visual Studio Code telah mengantisipasi kolaborasi berbasis console dan menyiapkan *integrated terminal* di dalam aplikasinya. Cukup mengetik terminal pada *searchbar* dan akan muncul pilihan terminal.




*Gambar 4. 10. Terminal Integrasi Visual Studio Code*

Terminal yang muncul terintegrasi dengan terminal atau *cmd* yang ada pada mesin lokal. Maka dari itu *integrated terminal* dapat mengakses semua direktori yang ada dalam mesin dengan melakukan integrasi perintah pada terminal atau *cmd* lokal. Untuk melakukan perintah menggunakan NPM, cukup ketik npm pada terminal.



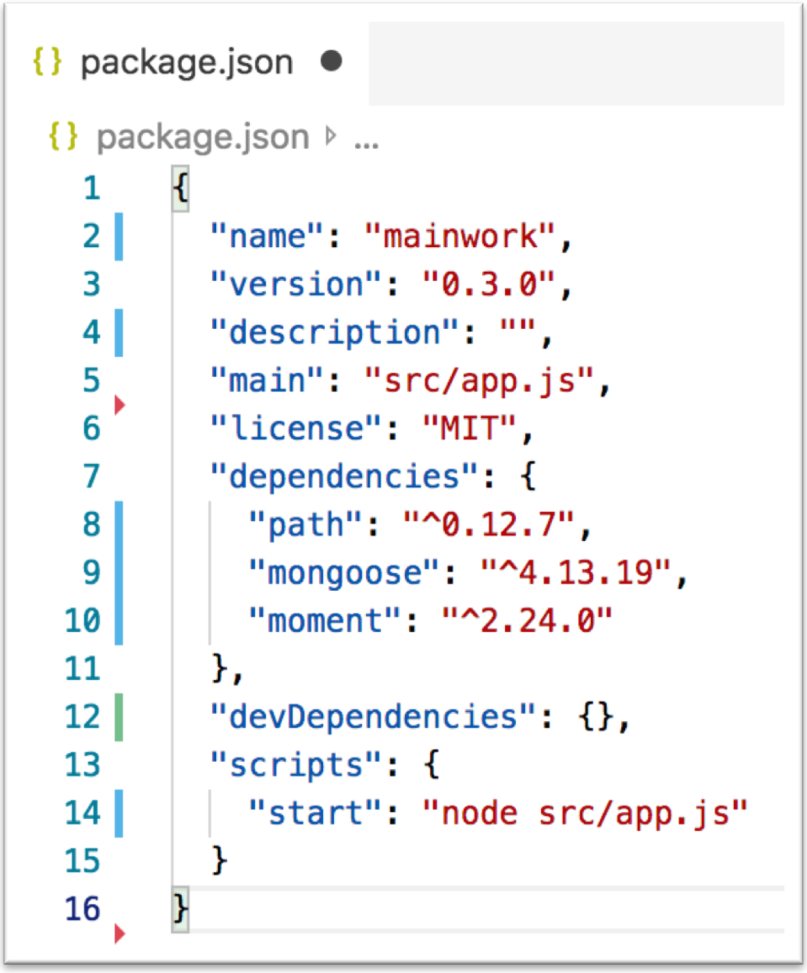

*Gambar 4. 11. Perintah NPM Dalam Terminal*

NPM tidak akan melakukan apapun tanpa perintah yang spesifik di belakang nama. Karena input yang ditulis tidak lengkap, NPM akan mengeluarkan saran dan contoh perintah yang dapat digunakan seperti pada *gambar 4.11*. Dalam pembuatan aplikasi ini, dibutuhkan package seperti Moment, Mongoose, dan Path. Ketiga package tersebut akan diinstal menggunakan perintah *npm install*.

```
eureka% npm install path

> bcrypt@3.0.4 install /Users/eureka/Documents/mainwork/node_modules/bcrypt
> node-pre-gyp install --fallback-to-build

node-pre-gyp WARN Using request for node-pre-gyp https download
[bcrypt] Success: "/Users/eureka/Documents/mainwork/node_modules/bcrypt/lib/binding/l
crypt_lib.node" is installed via remote
npm WARN bluewhale-api@0.3.0 No repository field.

+ path@0.12.7
added 936 packages from 646 contributors and audited 2890 packages in 29.599s
found 75 vulnerabilities (2 low, 11 moderate, 62 high)
  run `npm audit fix` to fix them, or `npm audit` for details
eureka% npm install mongoose --save
npm WARN bluewhale-api@0.3.0 No repository field.

+ mongoose@4.13.19
added 1 package from 1 contributor, removed 1 package, updated 2 packages and auditec
 2890 packages in 10.42s
found 74 vulnerabilities (2 low, 10 moderate, 62 high)
  run `npm audit fix` to fix them, or `npm audit` for details
eureka% npm install moment
npm WARN bluewhale-api@0.3.0 No repository field.

+ moment@2.24.0
updated 1 package and audited 2890 packages in 6.344s
found 74 vulnerabilities (2 low, 10 moderate, 62 high)
  run `npm audit fix` to fix them, or `npm audit` for details
eureka%
```

*Gambar 4. 12. Instalasi Package Menggunakan NPM*

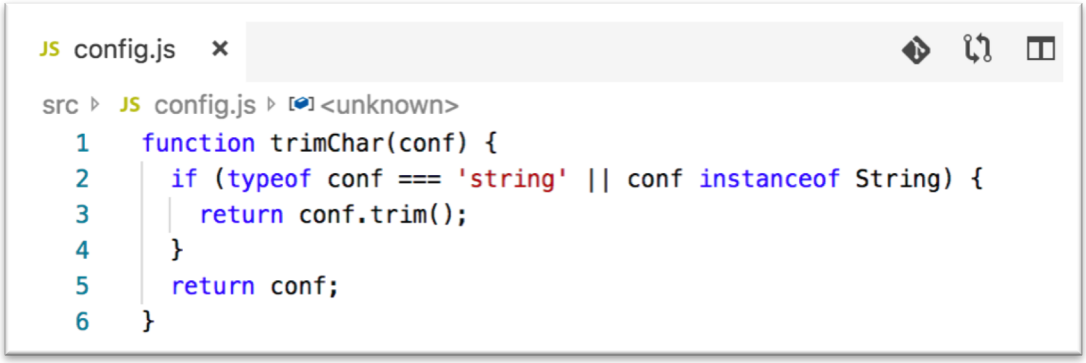
Setelah instalasi berhasil, *package-package* tersebut akan terdaftar ke dalam package.json. Di dalamnya dapat dilihat nama package dan versi yang digunakan di dalam proyek yang tengah dikerjakan.




*Gambar 4. 13. Isi package.json*

## 4.5 Koneksi Service ke Database

Untuk menyambungkan program dengan server database, pertama kita harus mendefinisikan alamat database dan mendaftarkannya ke dalam *configuration file*. Pada kasus ini, *configuration file* pada program terdapat di file dengan nama *config.js*.




*Gambar 4. 14. Fungsi trimChar*

*Function trimChar* berfungsi untuk menghapus semua spasi yang mungkin tidak sengaja ditambahkan pada URL ( *Uniform Resource Locator* ) atau alamat yang disambungkan kepada program. URL bersifat spesifik, kesalahan penulisan akan membuat koneksi tidak terjalin. Karena itu fungsi ini dibuat demi mengantisipasi kesalahan penulisan berupa spasi dalam URL.

```
module.exports = {
  app: {
    port: process.env.APP_PORT || 3000,
    ip: trimChar(process.env.APP_IP) || '0.0.0.0',
  },
  mongo: {
    uri: trimChar(process.env.MONGO_URL) || 'mongodb://localhost:27018/cprac_dev',
  }
};
```

*Gambar 4. 15. Fungsi Module Exports pada config.js*

*Module exports* merupakan salah satu bagian dari life-cycle express.js. Di dalamnya terdapat *code* yang mendefinisikan dimana program *service* akan berjalan. Karena semua proses ini merupakan proses dibalik layar, artinya seluruh *code* yang ada akan disimpan didalam *cloud server* di suatu tempat. Karena itu sangat penting untuk berhati-hati dalam penulisan alamat URL.

*app* mendefinisikan alamat *service* dapat diakses. Ketika program dijalankan, *service* akan terbuka dan dapat menerima *request*. Ketika aplikasi web atau aplikasi yang berinteraksi dengan *user* lain seperti aplikasi *mobile* ingin

mendapat akses ke *service*, mereka harus mengarahkan request mereka ke alamat *app*.

Selain alamat *app*, *module exports* juga berisi alamat yang bisa diakses *service*. Pada segmen ini, alamat *server database* mongoDB dituliskan. Sesuai *code* diatas, jika ingin mendapatkan akses pada *database*, program hanya perlu memanggil *path* menuju *mongo*.

## 4.6 Koneksi Program ke Service

Program yang bertugas menelusuri data sehingga menghasilkan jadwal *driver* pada dasarnya tidak akan bisa mengerjakan tugasnya jika tidak memiliki koneksi ke *database*. *Service* yang merupakan roda kerja aplikasi juga tidak akan mampu menunjang kebutuhan fungsi apabila tidak terkoneksi pada *database*. Karena itu, sebuah *service* pasti akan menyediakan koneksi ke *database*. Tujuan *service* menyediakan ini adalah agar semua program di dalam *service* yang memerlukan koneksi ke database dapat dengan mudah mendapatkan aksesnya.

Karena akses ke *database* telah disediakan oleh *service*, program biasanya akan mendeklarasikan akses ini di awal file. Berikut deklarasi yang digunakan oleh program :

```
const path = require('path');

const moment = require('moment');
```

*Gambar 4. 16. Koneksi Program untuk Mengakses Package*

## 4.7 Deklarasi Model Database ke Dalam Database

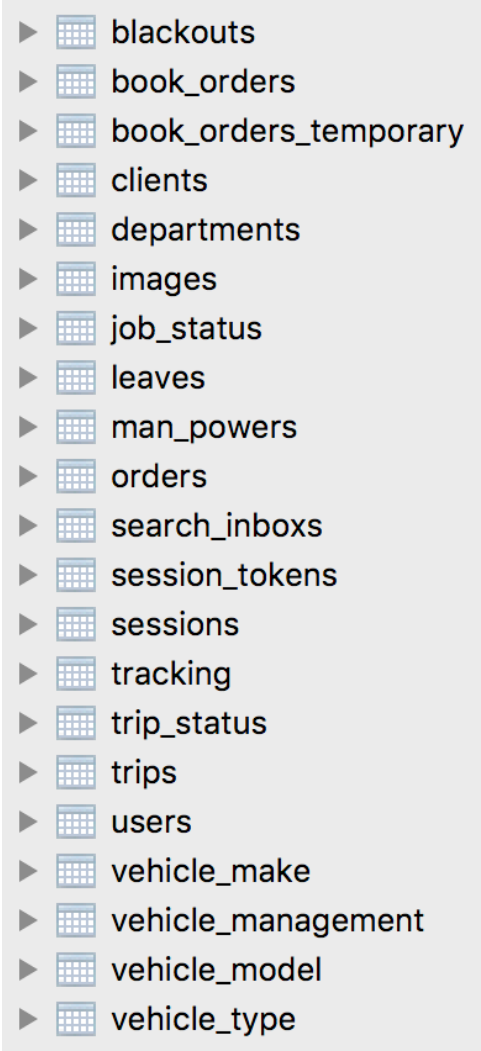
Selanjutnya dibutuhkan koneksi ke skema database agar *collection* dan *document* yang dituju tepat sasaran. Skema modul trips telah terlebih dahulu dibuat dan untuk menggunakan service tersebut, cukup daftarkan koneksi file program ke file skema.

```
const pathToTripModel = path.resolve('src', 'apps/api.v1/trip/models');
const TripModel = require(pathToTripModel);
```

*Gambar 4. 17. Deklarasi path Menuju File Skema*

## 4.8 Model Data

Tujuan utama dari aplikasi penjadwalan ini adalah untuk menampilkan data trip yang diinginkan dari trip-trip yang ada. Dengan banyaknya data di dalam database aplikasi, hal pertama yang perlu dilakukan adalah mensortir data apa saja yang perlu ditampilkan. Database yang digunakan untuk menjalankan aplikasi online transportation dengan metode book-first ini terdiri dari banyak tabel yang dibedakan berdasarkan kategori data, yang meliputi :




*Gambar 4. 18. Collections dalam Database*

Dari list tabel tersebut, yang perlu ditampilkan pada aplikasi terjadwal adalah data di dalam tabel *trips*. Isi dari tabel *trips* tersebut adalah dokumen yang mengandung data-data spesifik seperti *id trip, pickupAddress* dan *pickupDateTime*. Data *pickupDateTime* akan diambil untuk menentukan apakah trip tersebut akan ditampilkan pada aplikasi penjadwalan minggu ini. Sementara itu dalam suatu document, akan terdapat banyak data lain yang mungkin ditampilkan dalam rincian jadwal. Data ini memiliki kerangka yang sama persis dengan skema *trips*.

```
{
    "_id" : ObjectId("59dda19ba60cdd001c1bf32f"),
```

```
    "tripType" : "Normal Transfer",

    "pickupAddress" : "Airport Blvd, Singapore",

    "destinationAddress" : "80 Mandai Lake Rd, Singapore
729826",

    "pickupDateTime"         :         ISODate("2017-10-
18T04:00:00.000Z"),

    "pagingFlightNumber" : "",

    "remarks" : "",

    "order" : ObjectId("59dda19ba60cdd001c1bf32e"),

    "updatedAt" : ISODate("2017-10-11T04:44:11.141Z"),

    "createdAt" : ISODate("2017-10-11T04:44:11.141Z"),

    "rating" : 0,

    "reports" : [],

    "lock" : {

        "status" : false

    },

    "isCompleteOrder" : false,

    "isRequestCancel" : false,

    "orderStatus" : "Confirmed",

    "additionalDisposalTime" : 0,

    "endDestinationLocation" : [],

    "destinationLocation" : [
```

```
            103.793023,
            1.4043485
        ],
        "pickupLocation" : [
            103.9915308,
            1.3644202
        ],
        "numberOfGuest" : 3,
        "guest" : [
            "af@gmail.com"
        ],
        "alarmStatus" : "Ready",
        "manPowerOptions" : {
            "language" : "Mandarin",
            "race" : "Chinese"
        },
        "vehicleOptions" : {
            "vehicleModel"                                    :
ObjectId("59db3d15ae73bc001cf63338"),
            "seat" : "5",
            "luggage" : "2"
```

```
    },
    "alarms" : {
        "alert" : 15,
        "safe" : 30
    },
    "_type" : "Trip",
    "__v" : 0,
    "deploymentStatus" : "Acknowledged",
    "manPowerVehicle" : {
        "vehicleManagement"                                    :
ObjectId("59dee8ada60cdd001c1bf47d"),
        "manPower"                                             :
ObjectId("59e5a5f59f0a35001c33ef49")
    }
}
```

## 4.9 Pengambilan dan Seleksi Data

Pembuatan *code* yang bertugas menampilkan data jadwal dimulai disini. Pertama, buat konstruktor ( const ) dengan nama listOrdersByWeekWeb dengan parameter *date* dan *arrow function* berisi *object*.

```
const listOrdersByWeekWeb = ({ date }) => {

}
```

*Gambar 4. 19. Pembuatan konstruktor*

Di dalam object tersebut dilakukan pengecekan hari dalam satu minggu dimulai dari hari senin dan diakhiri dengan minggu. Berikut adalah *code* dan hasil proses yang terjadi :

```
const dateObject = moment.utc(date, 'YYYY-MM-DD');

const today = dateObject.startOf('week').toDate();
const tomorrow = dateObject.endOf('week').toDate();

const getDays = (startDate, endDate) => {
    const dates = [];

    const currDate = moment.utc(startDate).startOf('day');
    const lastDate = moment.utc(endDate).endOf('day');

    while (currDate.diff(lastDate) <= 0) {
        dates.push(currDate.clone().toDate());
        currDate.add('1', 'days');
    }

    return dates;
};
```

*Gambar 4. 20. Code Untuk Mengecek Hari Dalam Satu Minggu*

```
eureka% node src/getdays.js
[ 2019-07-07T00:00:00.000Z,
  2019-07-08T00:00:00.000Z,
  2019-07-09T00:00:00.000Z,
  2019-07-10T00:00:00.000Z,
  2019-07-11T00:00:00.000Z,
  2019-07-12T00:00:00.000Z,
  2019-07-13T00:00:00.000Z ]
eureka%
```

*Gambar 4. 21. Hasil Code Mendapatkan Hari Dalam Satu Minggu*

Ketika data telah selesai melewati sortir waktu, proses selanjutnya melibatkan rekonstruksi data dengan tujuan akhir menghasilkan data yang terstruktur. Data-data ini kemudian diproses melalui *query database*. *Query* pertama bertujuan untuk mengambil semua data trip yang memiliki *pickupDateTime* pada hari di minggu tersebut.

```
pickupDateTime: {
    $gte: today,
    $lte: tomorrow,
},
```

*Gambar 4. 22. Queri Pertama*

*Query* kedua bertujuan untuk memastikan bahwa *driver* yang bertugas melaksanakan trip tersebut benar ada dan tidak sedang dalam kondisi yang dibatasi.

```
'manPowerVehicle.manPower': {
    $exists: true,
},
```

*Gambar 4. 23. Query Kedua*

*Query* ketiga bertujuan untuk memastikan bahwa order dengan *trip id* tersebut benar terdaftar dalam *database*. Proses ini bertujuan mengatasi kemungkinan trip yang sudah dibatalkan untuk masuk ke jadwal trip mingguan.

```
'manPowerVehicle.manPower': {
    $exists: true,
},
```

*Gambar 4. 24. Query Ketiga*

*Query* keempat adalah *query* yang berfungsi memastikan bahwa trip yang akan dimasukkan kedalam jadwal trip mingguan statusnya telah terkonfirmasi. Yang artinya, trip tersebut telah memiliki *driver* yang akan melaksanakan eksekusinya.

```
order: {
    $exists: true,
},
```

*Gambar 4. 25. Query Keempat*

Setelah melakukan restriksi akan data yang diinginkan, data yang tersisa pasti merupakan data yang benar. Akan tetapi jika tidak dipanggil, data tersebut hanya akan tersimpan di database tanpa bisa diolah lebih lanjut. Maka dari itu perlu dilakukan *query* untuk memanggil data yang telah benar.

```
const trips = TripModel.find(query)
    .populate([
        { path: 'order',
        populate: [{ path: 'client', select: 'fullName' }],
        },
        { path: 'manPowerVehicle.manPower', select: 'fullName' },
    ])
    .exec();
```

*Gambar 4. 26. Query Untuk Memanggil Data Dari Collections Lain*

Selain data trip yang benar, rincian seperti nama *client* dan *manpower* yang menangani trip ini harus ditampilkan. Karena kedua data tersebut tidak berada pada tabel yang sama, diperlukan sebuah fungsi yang mampu mengambil data dari tabel yang berbeda. Demi memudahkan prosesnya, digunakan *mongoose* lebih tepatnya *mongoose populate*. Dengan fungsi ini *mongoose* dapat keluar dari *schema* tabel program dan menuju tabel yang diinginkan untuk selebihnya mengambil data.

*Path* dalam *populate* mendefinisikan tabel tujuan sementara *select* menunjuk pada kategori data yang ingin diambil. *Path: 'client', select: 'fullName'* berarti pergi ke tabel *client* dimana data yang akan diambil adalah value kategori *fullName*. Ketika proses selesai dijabarkan cukup butuh satu kata perintah agar mongoose mulai mengerjakan tugasnya, *execute* ( .exec() ).

## 4.10 Enkapsulasi Data Menjadi JSON

Pada tahap ini telah didapat semua data *driver* dalam kurun waktu satu minggu. Namun demikian, bentuk data yang didapat masih acak dan perlu dilakukan manipulasi. Karena jadwal yang diminta adalah jadwal per *driver*, enkapsulasi pertama yang dilakukan adalah menata data berdasarkan *driver* atau juga disebut groupByDriver.

```
const groupByDriver = (results) => {
    const tripsByDrivers = [];
    return tripsByDrivers;
};
```

*Gambar 4. 27. Membuat Fungsi Group By Driver*

Konstruktor groupByDriver memiliki parameter *result* dan menghasilkan *output* bertipe *array* dengan nama tripsByDriver.

```
results.forEach((trip) => {
    const tripDriver = trip.manPowerVehicle.manPower;
    const tripDriverId = tripDriver.id.toString();

    let driverSchedule = tripsByDrivers.find((tripByDriver) => {
        if (tripByDriver &&
                tripByDriver.driverId &&
                tripDriverId === tripByDriver.driverId) {
                    return true;
        }

        return false;
    });
```

*Gambar 4. 28. Looping Untuk Mendapatkan List Driver*

Pada gambar 4.2 dilakukan *looping* data dengan parameter trip berisi *id driver* yang ditugaskan menjalankan trip. Tujuan *looping* ini agar tiap data trip terintegrasi dengan data *driver*.

```
if (!driverSchedule) {
    driverSchedule = {
        driverId: tripDriverId,
        driver: tripDriver,
        trips: [],
    };
    driverSchedule.trips.push(trip);

    tripsByDrivers.push(driverSchedule);
} else {
    driverSchedule.trips.push(trip);
}
```

*Gambar 4. 29. Mengambil Data Driver yang Sesuai Trip*

Ketika *id driver* cocok dengan *id driver* yang melaksanakan trip, rincian data trips berupa id pelaksana akan dilempar ke dalam variable bernama driverId. Kemudian data trip akan dilempar ke dalam variabel *driver*. Baik driverId maupun *driver* memiliki jumlah yang banyak dan semuanya harus ditampilkan. Maka dari itu data yang diperoleh dilempar ke variabel baru untuk selanjutnya diproses lebih lanjut. Apabila *driver* tidak memiliki jadwal trip, data kosong akan dilempar ke dalam variabel trip yang dimiliki olehnya. Semua data yang diperoleh pada proses ini disimpan ke dalam *array* tripsByDriver untuk digunakan lebih lanjut.

Semua data yang telah diambil adalah data semua *driver* selama satu minggu. Namun demikian, data-data tersebut belum terisi oleh data trip yang akan dikerjakan. Karena itu dibutuhkan sebuah *code* dengan fungsi *looping* trip.

```
const mappingDriverSchedule = drivers =>
    drivers.map((driver) => {…
    });
```

*Gambar 4. 30. Membuat Fungsi Mapping Driver Schedule*

*Code* di atas memiliki maksud parameter mappingDriverSchedule berisi map atau peta dari banyak *driver* yang memiliki trip. Looping dilakukan dengan cara mengecek trip per hari serta menampilkan berapa banyak trip yang harus dilakukan. Trip yang harus dikerjakan dibagi menjadi tiga jenis, yakni transfer normal, transfer disposal, dan transfer kedatangan. Ketiga jenis tersebut sebelumnya telah dideklarasi di dalam skema trips.

```
drivers.map((driver) => {
    const driverScheduleInAweek = [];

    daysInAWeek.forEach((currentDate) => {
        const startDay = moment.utc(currentDate).startOf('day');
        const endDay = moment.utc(currentDate).endOf('day');
        const schedule = {
            trf: 0,
            dis: 0,
            mo: 0,
            triptrf: [],
            tripdis: [],
            tripmo: [],
        };
```

*Gambar 4. 31. Membuat Fungsi Penambahan Total Trip Per Hari*

Dalam penjadwalan, data pertama yang ditampilkan adalah jumlah total trip yang dilakukan. Maka karena itu dilakukan loop setiap hari untuk mengecek tipe trip dan melakukan penambahan pada setiap trip yang ditemukan. Berikut *code* yang dibuat untuk melakukan penjumlahan trip dalam sehari berdasar jenis trip.

```
driver.trips.forEach((trip) => {
    const pickupDateTime = moment.utc(trip.pickupDateTime);

    if (pickupDateTime.isBetween(startDay, endDay, null, '[]')) {
        switch (trip.order.orderType) {
            case ORDER_TYPE.TRF: {
                schedule.trf += 1;
                schedule.triptrf.push(trip);
                break;
            }
            case ORDER_TYPE.DIS: {
                schedule.dis += 1;
                schedule.tripdis.push(trip);
                break;
            }
            case ORDER_TYPE.MO: {
                schedule.mo += 1;
                schedule.tripmo.push(trip);
                break;
            }
            default:
                break;
        }
    }
});
```

*Gambar 4. 32. Fungsi Penambahan Trip Berdasarkan Tipe Trip*

Setelah data per hari dan total trip telah didapat, saatnya melempar data hasil proses ke dalam array driverScheduleInAweek. Dengan *code* di bawah, semua urutan data yang sudah diproses disimpan.

```
driverScheduleInAweek.push({
    date: currentDate,
    schedule,
});
```

*Gambar 4. 33. Menyimpan Value Hasil Jadi*

Ketika semua data telah didapat, looping proses belum akan berhenti. Untuk mencegah terjadinya data ganda, dibuat sebuah perintah singkat untuk menghapus data trip yang sudah dipanggil dan diproses.

```
delete driver.trips;
```

*Gambar 4. 34. Menghapus Data yang Sudah Sekali Diproses*

Ketika proses mapping ini selesai, data yang dikeluarkan berisi *driver* dan jadwal *driver* dalam satu minggu. Code dibawah dibuat dengan tujuan tersebut :

```
return {
    driver,
    schedule: driverScheduleInAweek,
};
```

*Gambar 4. 35. Hasil Jadi Dalam Variabel driverScheduleInAweek*

Sampai disini telah didapat 2 array berisikan list *driver* yang ada dan rincian trip. Baik itu total trip yang dilakukan setiap hari yang tersimpan di dalam variabel groupByDriver dan rincian trip tiap-tiap *driver* yang disimpan dalam variabel berisi mapping bernama mappingDriverSchedule.

Step terakhir yang harus dilakukan adalah memastikan fungsi listOrdersByWeekWeb menghasilkan hanya satu output. Hal ini dilakukan karena tiap *request* membutuhkan *response*. Dan *response* yang baik adalah *response* yang berisi satu variabel. Hal ini juga membantu JSON dikirim hanya sekali dengan data yang sudah mencukupi. Berikut fungsi yang dibuat untuk menyatukan kedua jadwal:

```
const format = trips
    .then(groupByDriver)
    .then(mappingDriverSchedule);

return format;
```

*Gambar 4. 36. Menggabungkan Dua Format Data Menjadi Satu*

Semua proses yang dimulai dari mendapatkan tanggal dalam satu minggu hingga format jadwal terakhir tidak menjamin bahwa code akan selalu berjalan dengan baik. Karena itu perlu dibuat antisipasi jika sesuatu terjadi secara salah. Antisipasi yang dibuat untuk kasus ini adalah dengan memasukkan semua *code* ke dalam *try-catch loop* lalu menambahkan deskripsi untuk situasi code gagal. Berikut garis besar *code* :

```
const listOrdersByWeekWeb = ({ date }) => {
    try {
        const dateObject = moment.utc(date, 'YYYY-MM-DD');

        const today = dateObject.startOf('week').toDate();
        const tomorrow = dateObject.endOf('week').toDate();

        const getDays = (startDate, endDate) => {…
        };

        const daysInAWeek = getDays(today, tomorrow);

        const query = {…
        };

        const trips = TripModel.find(query)
            .populate([…
            .exec();

        const groupByDriver = (results) => {
            const tripsByDrivers = [];
            results.forEach((trip) => {…
            });
            return tripsByDrivers;
        };
        const mappingDriverSchedule = drivers =>
            drivers.map((driver) => {…
            });

        const format = trips
            .then(groupByDriver)
            .then(mappingDriverSchedule);

        return format;
    } catch (e) {
        throw e;
    }
};
```

*Gambar 4. 37. Seluruh Code Yang Digunakan Untuk Memproses Data*

Semua code di atas baru akan bisa dipakai setelah fungsi didaftarkan ke dalam life-cycle express.

```
module.exports = {
    listOrdersByWeekWeb,
};
```

*Gambar 4. 38. Memasukkan Hasil Proses Ke Dalam Express*

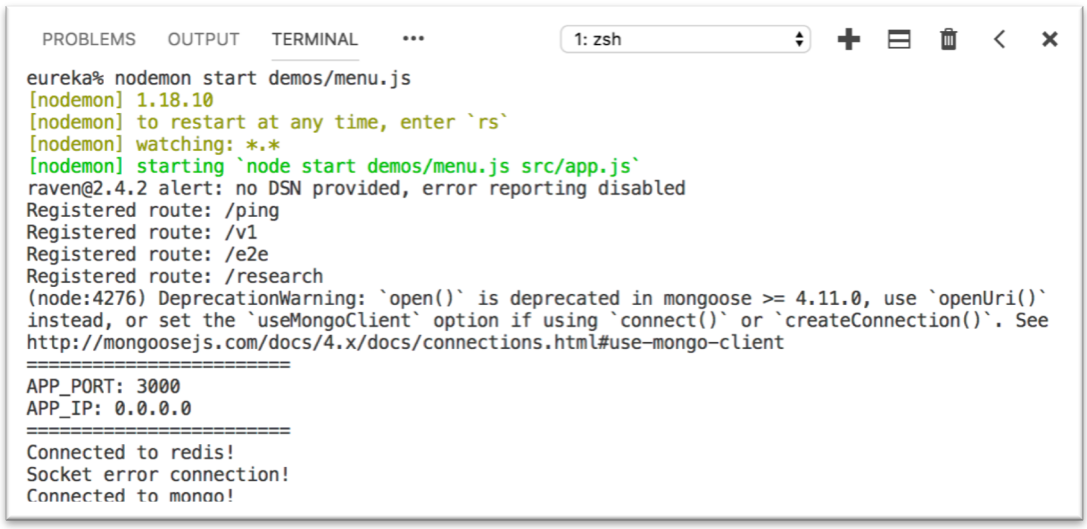
Selain agar dapat diakses oleh file lain, program penjadwalan harus memiliki *endpoint* yang terdaftar di dalam Service agar selanjutnya dapat diakses oleh *client*. Pembuatan *endpoint* ini dilakukan di index.js dalam *folder* utama *trip*.

```
apps ▸ api.v1 ▸ trip ▸ JS index.js ▸ ...
const weekScheduleHandler = require('./controllers/list-by-week');

app.get('/list-by-week', secure.auth, weekScheduleHandler);

module.exports = app;
```

*Gambar 4. 39. Pembuatan Endpoint Agar Bisa Diakses Client*

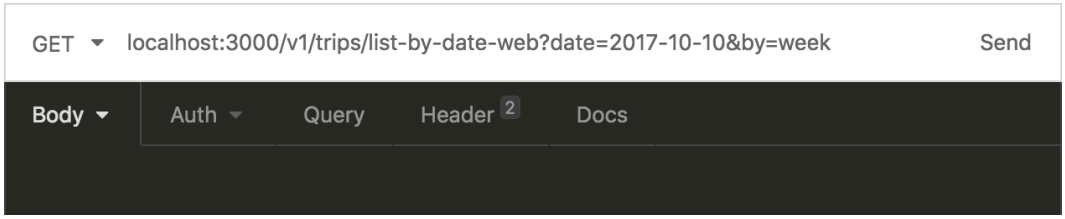
## 4.11 Pengujian

Pengujian dilakukan dengan cara menyalakan service dan menggunakan fungsi penjadwalan yang dibuat. Pastikan service terkoneksi dengan database.



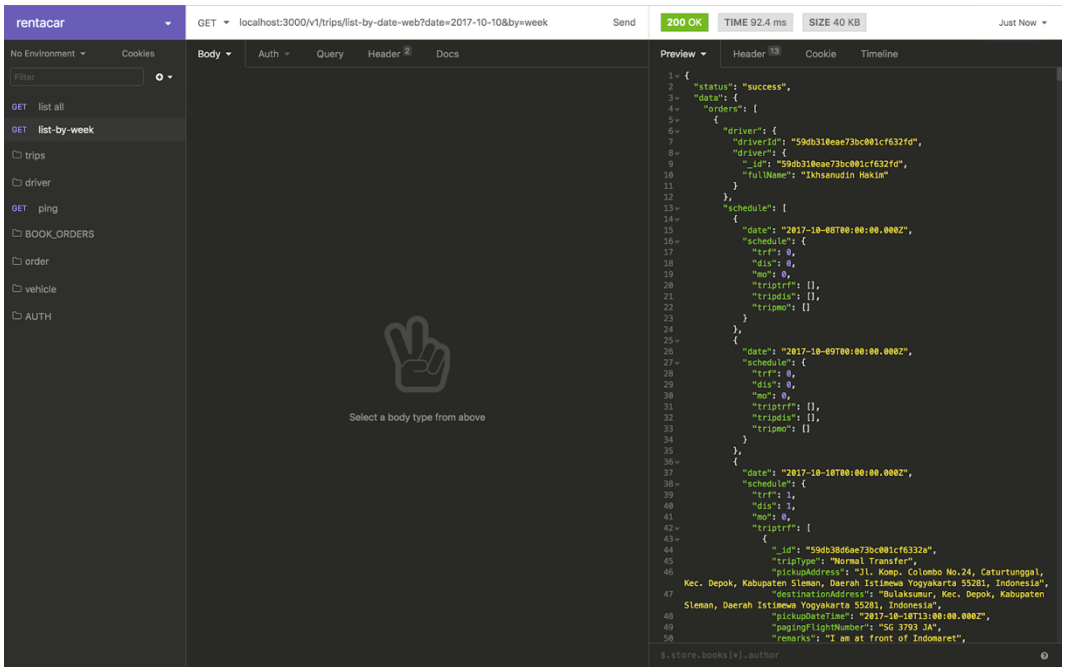

*Gambar 4. 40. Perintah Memulai Service*

Dalam proses ini dibutuhkan aplikasi *REST-API client* yakni insomnia. Buka aplikasi dan tulis *endpoint* aplikasi penjadwalan.




*Gambar 4. 41. Memasukkan Tujuan Endpoint Service Penjadwalan*

Setelah semua proses dilakukan dengan benar, insomnia akan mengakses service dan memberikan hasil akhir berupa *gambar 4.41* sementara hasil akhir yang lebih lengkap terlampir di *lampiran 1*.




*Gambar 4. 42. Hasil Data JSON yang Dikirim Service Sebagai Respon*