**NODE JS BEST PRACTICES: https://github.com/goldbergyoni/nodebestpractices**

**5. Fundamentals.**

Axios simple example:

const axios = require('axios')

axios.get('https://www.google.com').then((res) => {

    console.log(res);

}).catch((err) => {

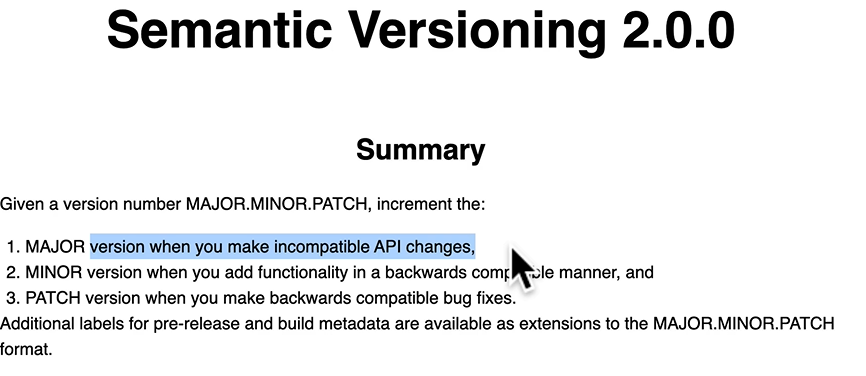
    console.log(err);

}).then(() => {

    console.log('All done');

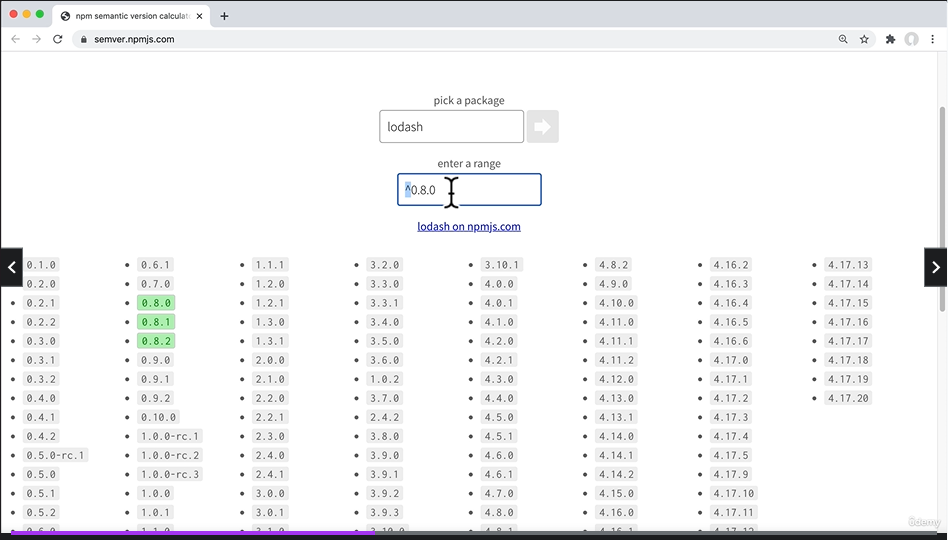
})

**Версионирование** делается по следующему принципу:



*Версия указывается в файле* ***package.json, но только примерная/совместимая. Точная версия указывается в package-lock.json, если есть версия, указанная в этом файле, то ставится именно она. Если только в package.json, то ставится совместимая.***

**Проверить совместимые версии можно на сайте:**



**54. Проблемы секьюрности пакетов NPM**

Обычно при выявлении проблем с секьюрностью, кидает ошибки в терминал, но можно запустить команду **npm audit**, для автоматического фикса: **npm audit fix (меняет версии пакетов!!!)**

**PLANETS PROJECT**

.CSV – Comma separated values

Установим парсер файлов csv: **csv-parse**

**Streaming large database files**

Когда мы хотим прочитать большой файл, загрузить его сразу может быть тяжелой или неподъемной задачей для сервера, поэтому надо использовать СТРИМЫ.

В модуле fs есть метод createReadStream. Он в свою очередь генерирует события: data, end, error

Как соединить парсинг из либы и чтение файла? Через метод **pipe()**, т.к. парсер был разработан с учетом работы в стримах.

const { parse } = require('csv-parse');

const fs = require('fs')

const habitablePlanets = []

function isHabitablePlanets(planet) {

    return planet['koi\_disposition'] === 'CONFIRMED'

        && planet['koi\_insol'] > 0.36 && planet['koi\_insol'] < 1.11

        && planet['koi\_prad'] < 1.6

}

fs.createReadStream('kepler\_data.csv')

    .pipe(parse({  // подключение парсера к стриму

        comment: '#',

        columns: true,

    }))

    .on('data', (planet) => { // обработчик события 'data'

        if (isHabitablePlanets(planet)) { habitablePlanets.push(planet) }

    })

    .on('error', (error) => {

        console.log(error); // обработчик события 'error'

    })

    .on('end', () => { // обработчик события 'end'

        console.log(habitablePlanets.map((planet)=>{

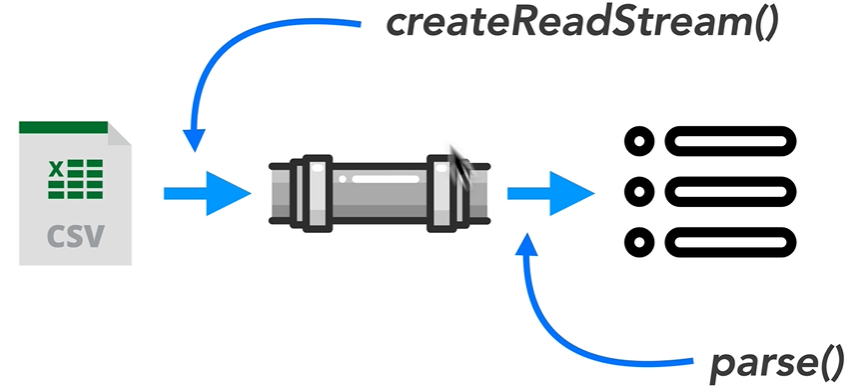
            return planet['kepler\_name']

        }));

        console.log(`=${habitablePlanets.length} habitable planets found!`);

        console.log('done');

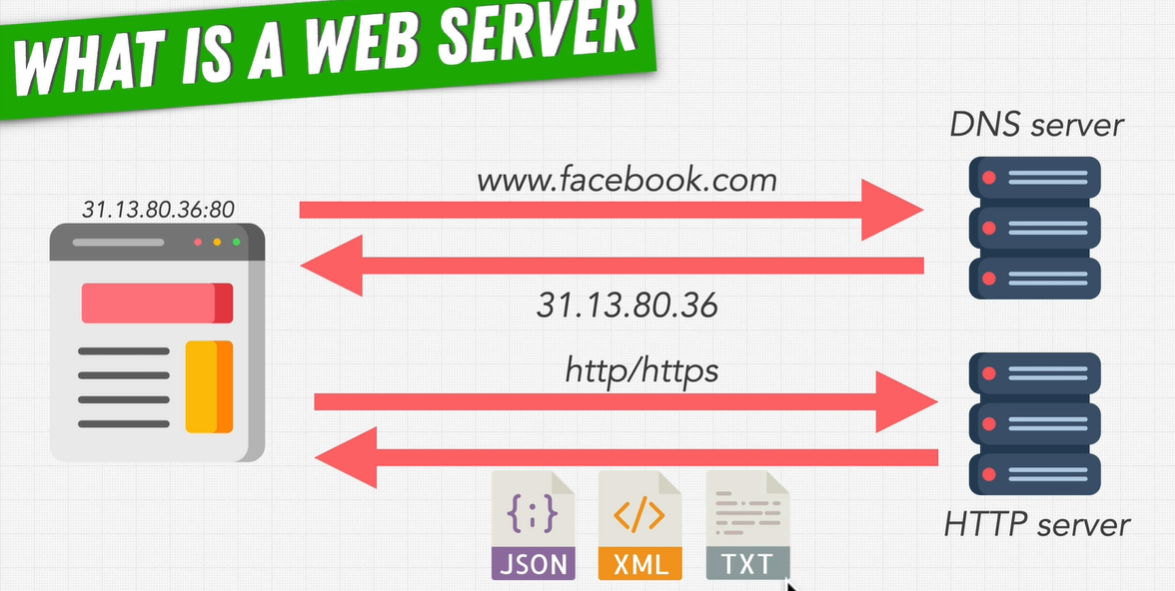
    })



WEB SERVERS

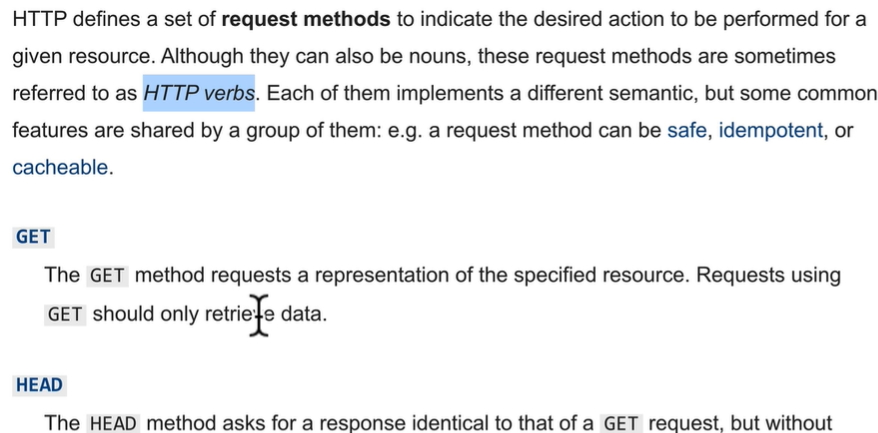
**68. Поток данных**

Схема работы сети:

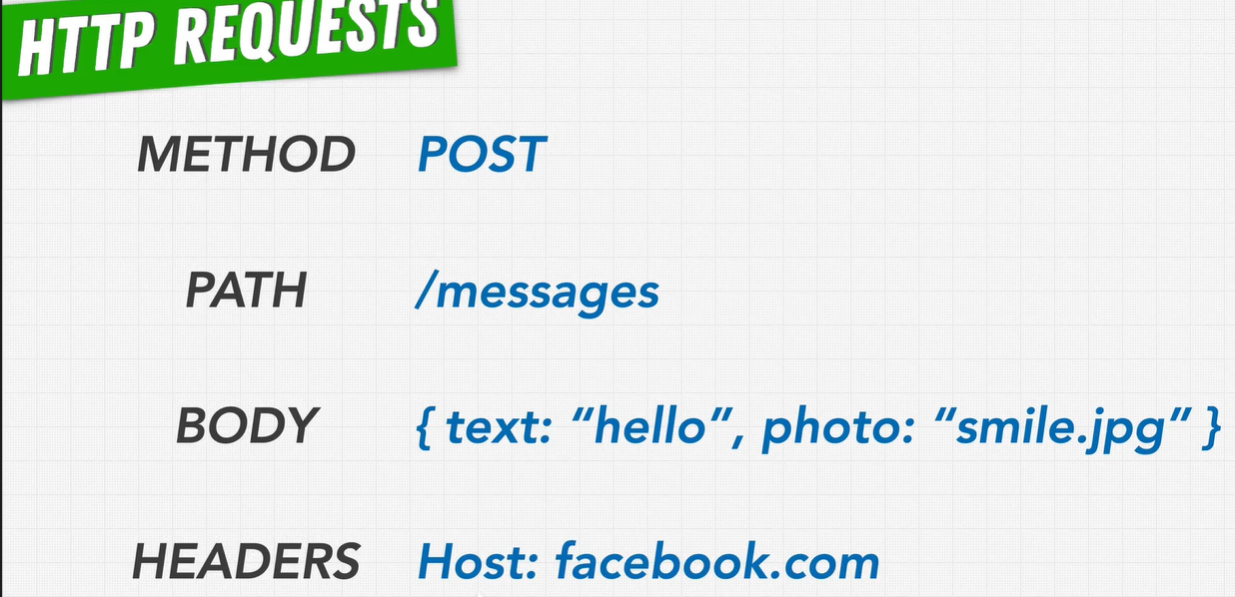


**69. Запросы и ответы**

HTTP сервер может принимать и отвечать только HTTP язык, но бэк может быть написан на любом языке!



**Запрос на сервер состоит из:**



Host – обязательно

**Ответ с сервера состоит из:**



**72. Простой Веб-сервер**

* **Передавать данные можно только в виде строки JSON.stringify()**
* **Тип данных определяется в Content-Type:**

res.writeHead(200, {

        'Content-Type': 'application/json'

    })

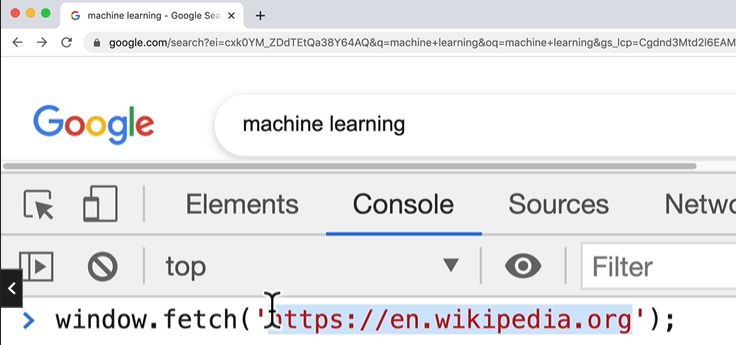
**74. SAME origin policy**

**Origin** – это комбинация трех элементов в URL:

1. Protocol: http/http
2. Host: [www.google.com](http://www.google.com)
3. Port (он есть в запросе):

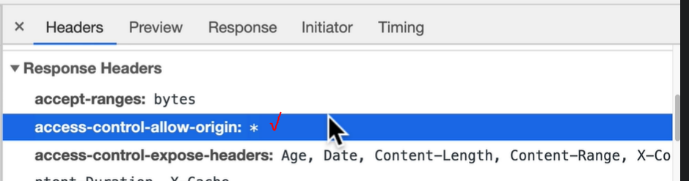
**Одинаковыми можно считать только те два ориджина, у которых одинаковая комбинация этих трех элементов (SAME ORIGIN POLICY).** Т.е. если разный протокол – разные ориджины.

**SOP** – это секьюрная фича браузера, которая ограничивает браузер в том, что он может загрузить, а что нет.

*Т.е. пока я нахожусь на сайте google браузер не может отправлять запросы на facebook, кроме POST запроса (т.к. предотвращение запросов GET позволяет не допустить утечку данных).*

Как пример: можно отправить фетчем запрос на Википедию, находясь на странице гугла, сделать это не даст! **Здесь мы создаем Cross Origin Request, он нарушает SOP**

**77. CROSS Origin policy**

Если нужно сделать возможным загрузку данных с другого origin, надо их отключать в хедерах:

Также можно указать, с какого именно ориджина можно загружать данные:



EXPRESS

**84. Basics**

Пример простого конфига Express. Content-type указывать не надо, Express может сам его определить (кроме тех случаев, когда надо отправить JSON)

const express = require('express')

const app = express()

const PORT = 3000

app.get('/', (req, res) =>{

    res.send('Hello')

})

app.post('/messages', (req, res) =>{

    res.send('Updating message')

})

app.listen(PORT, () => {

    console.log(`Listening on ${PORT}...`);

})

**85. Query Params (Parametrized Routes)**

Прописываются в роуте и достаются из request

app.get('/friends/:friendId', (req, res) => {

    const friendId = +req.params.friendId

    const friend = friends.find(friend => friend.id === friendId)

    if (friend) res.json(friend)

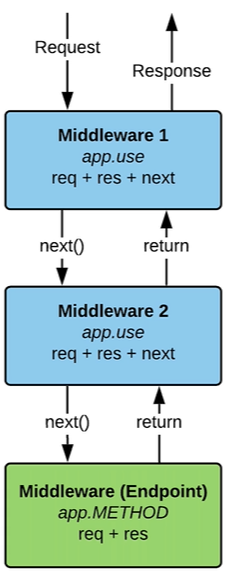
    else res.status(404).json({

        error: 'Friend not found'

    })

})

**85. MIDDLEWARES**

**Middleware** – это функция, которая работает между получением реквеста и отправкой респонза. Запускаются по порядку, определенному в коде!

Мидлвары вставляются в поток с помощью метода app.**use()**

Стандартный мидлвар имеет следующий вид (**next** – это следующий мидлвар):

app.use((req, res, next) =>{

})

Движение реквеста в эндпоинт (вниз по схеме) называется **Downstream middleware,** вверх по схеме **Upstream middleware**

Пример простого мидлвара, который логает реквесты:

app.use((req, res, next) => {

    console.log(`${req.method} ${req.url}`);

    next() // вызов следующего мидлвара

})

Перехват РЕКВЕСТА на пути к серверу идет до next(), перехват респонза к клиенту идет после next(). В мидлваре ниже мы видим сколько времени обрабатывается запрос. Postman покажет сколько времени прошло с момента отправки запроса на сервер до получения ответа.

app.use((req, res, next) => {

    const start = Date.now()

    console.log(`${req.method} ${req.url}`);

    next() // до этой строки это Downstream, после - Upstream, здесь можем перехватить респонз на пути к клиенту

    const delta = Date.now() - start

    console.log(delta);

})

**85. POST запросы в Express**

В Express есть встроенный метод, который парсит json.

Добавим этот мидлвар вторым по счету:

app.use(express.json())

После каждой записи в респонз, надо закрывать его! Если один раз засетал данные, второй раз пересетить нельзя. После сета пишем **RETURN**!

app.post('/friends', (req, res) => {

    if (!req.body.name) {

        return res.status(400).json({

            error: 'Bad request'

        })

    }

    const newFriend = {

        id: friends.length,

        name: req.body.name,

    }

    friends.push(newFriend)

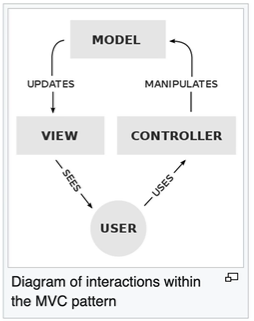
    res.status(200).json(newFriend)

})

**86. MODEL-VIEW-CONTROLLER**

**Это дизайн паттерн**

**USER** Отправляет запрос в **CONTROLLER**, который потом манипулирует **MODEL**(БД), после этого **VIEW** реагирует на изменения в **MODEL** и показывает их **USER,** который отправляет запрос в **CONTROLLER,** который вызывает изменения в **MODEL** и так по кругу.



В Node **CONTROLLER** – это функции, которые принимают запрос, обрабатывают его и отправляют респонз.

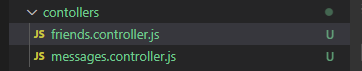
В **MODEL (БД)** происходят изменения в базе данных, в ней также функции, с помощью которых мы взаимодействуем с БД

Представление данных в **MODEL** и у **USER** могут и чаще всего отличаются!!! Поэтому типы данных в MODEL и в USER надо четко разграничивать!

**Согласно этому паттерну, мы будем делить приложение на эти слои**

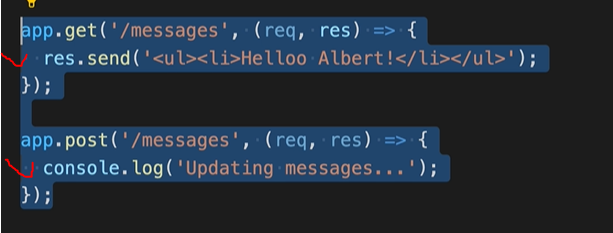
**92. MODEL-VIEW-CONTROLLER in EXPRESS**

**Создадим Контроллер. Это будет слой, в котором реализована ФУНКЦИОНАЛЬНОСТЬ для конкретной модели.**



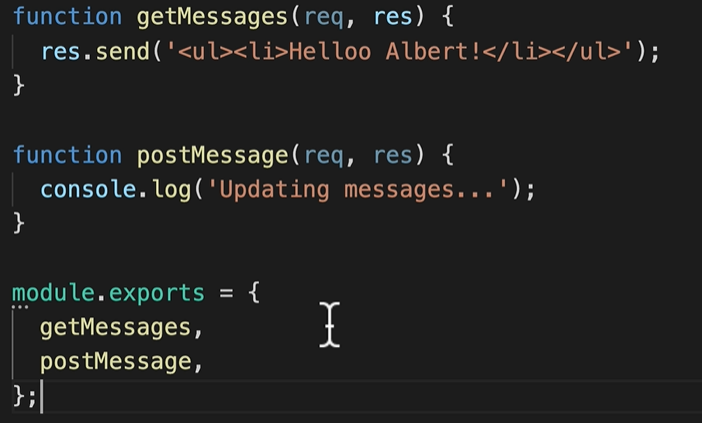
Теперь перенесем логику, реализованную в колбэках для роутов в контроллер, каждой присвоим имя:

server.js



=>

messages.controller.js:



То же самое сделаем для запросов Friends. При этом сам массив перенесем в отдельную папку **MODEL**

**Friends.model.js**

const friends = [

    { id: 1, name: "Name1" },

    { id: 2, name: "Name2" },

    { id: 3, name: "Name3" },

];

module.exports = friends

При импорте в friends.controller.js назовем масиив friends как **model**

const model =  require('../models/friends.model')

function postFriend(req, res) {

    if (!req.body.name) {

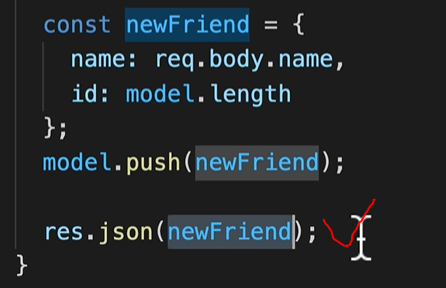
        return res.status(400).json({

            error: "Bad request",

        });

…

**Итого, на текущий момент мы реализовали CONTROLLER, а также MODEL. Что же в данном случае VIEW? В нашем случае VIEW это JSON, который мы отправляем в респонзе на фронт!**



**92. EXPRESS ROUTER**

В EXPRESS есть свой роутер, чтобы систематизировать роуты и сделать приложение более модульным.

const friendsRouter = express.Router()

Далее имплементим его как **middleware**.

Каждый роут прописываем по первой директории? Остальные роуты прописываем относительным путем. Таким образом получается изолированный роутер для friends.

Когда мы выносим урл, определяющий роутер, это будет **BASE URL**

const messagesRouter = express.Router()

friendsRouter.post("/", friendsController.postFriend);

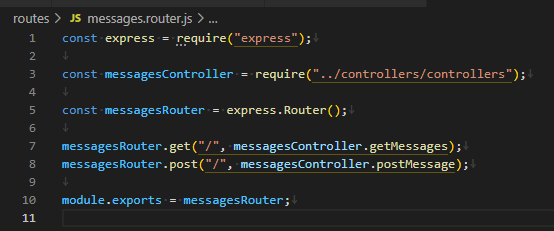
friendsRouter.get("/", friendsController.getFriends);

friendsRouter.get("/:friendId", friendsController.getFriend);

**/friends – это BASE URL**

app.use('/friends', friendsRouter) // mounting router an the app project

Перенесем все роуты в отдельную папку **routes:**



Далее создадим **custom middleware и добим его в роутер. Так можно реализовать любой функционал, который нужен для обработки запросов/ответов.**

friendsRouter.use((req, res, next) => {

    console.log('ip address: ', req.ip);

    next()

})

**94. Проектирование REST API**

**Коллекция либо список чего-либо**: сущ. во множественном числе

**Получить/добавить/удалить элемент из списка или коллекции:** сущ. во множественном числе + uri params(:id)

**REST** – **Re**presentational **S**tate **T**ransfer

**API является RESTful:**

- Использует существующие стандарты (HTTP(протокол, стандарт), JSON(данные), URL(путь к серверу, ресурсу))

- Эндпоинты – это коллекции данных

- Использование методов GET, POST, PUT, DELETE

- Наличие Клиента и Сервера

- Запросы всегда STATELESS (без состояния, т.е. он никак не законнекчен к состоянию на клиенте. Мы не храним и не используем историю и содержимое запросов) и CACHEABLE (мы можем сохранять результаты для будущего использования – например, если коллекция не менялась, не обязательно проверять ее на изменения перед следующим запросом)

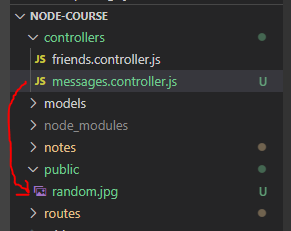
**96. Отправка Файла**

Для отправки файлов можно воспользоваться методом **res.sendFile()** у респонза. Но в нем нужно указать путь к файлу. Для этого надо пользоваться модулем **‘path’.**

В разных операционных системах пусти к файлам могут отличаться. Например, в Линуксе, папки разделяются /, а в Винде \. Модуль path помогает избежать проблем с этим.

Итак, укажем путь к файлу:

path.join(\_\_dirname, '..', 'public', 'random.jpg')



**\_\_dirName** – это глобальная переменная в Node, которая содержит в себе путь к файлу, в которым прописывается эта переменная (т.е. указывает на текущую папку).

‘**..**’ – один уровень вверх

‘**public**’ – вложенная папка

‘**random.jpg**’ – нужный для отправки файл.

Итого:

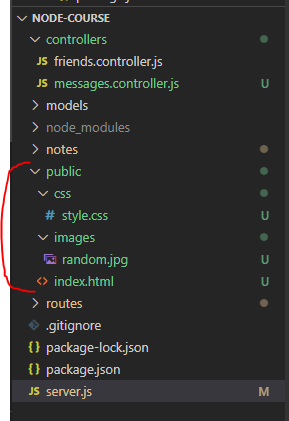
res.sendFile(path.join(\_\_dirname, '..', 'public', 'random.jpg'))

**97. Отправка нескольких Файлов**

Любой веб сервер (Веб, реакт, ангуляр и пр.) отправляет клиенту множество файлов для отображения контента. Для отправки множества файлов надо использовать **Express Static File Middleware**

В файле server.js добавим **мидлвар с указанием папки со статическими файлами.** В качестве урла будем использовать ‘/site’. Здесь будем использовать path.join + dirname, так node сможет построить путь к папке public из любой папки приложения (по факту строится путь из корня)

app.use('/site', express.static(path.join(\_\_dirname, 'public')))

в папке public создадим простой набор файлов для отображения html страницы – html, картинка, стили.

Если у приложения большое количество пользователей есть смысл хранить все static файлы приложения в **ОБЛАКЕ** (AWS как пример)

**97. Шаблоны (Template Engines)**

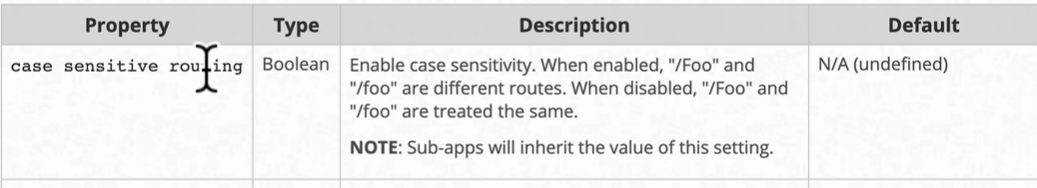
В Node js их много, будем использовать **handlebars (hbs). С помощью Template Engines в шаблоны html можно вставлять переменные в {{}} скобки для динамически изменяемого контента ДО отправки на клиента.**

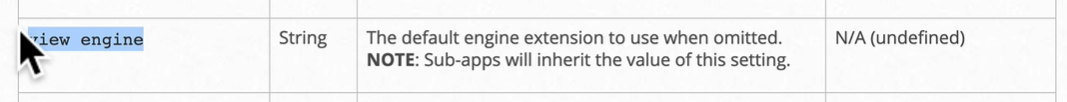
Установим пакет **hbs:**

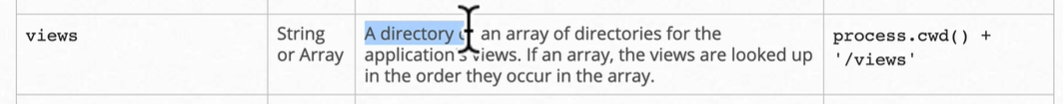
*npm install hbs*

Далее идем в настройки Ноды и задаем следующие настройки.









app.set('view engine', 'hbs')

app.set('views', path.join(\_\_dirname, 'views'))

Далее создадим template:

**Index.hbs**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="stylesheet" href="css/styles.css" />

    <title>{{title}}</title>

</head>

<body>

    <h1>{{caption}}</h1>

    <img src="images/random.jpg" alt="ski">

</body>

</html>

И допишем в server.js:

app.get("/", (req, res) => {

    res.render('index.hbs', {

        title: 'MyFriends are clever',

        caption: 'Let\' go skiing'

    })

})

NASA PROJECT

**101. Architecture**

Создадим в проекте две папки: **Server и Client**

В каждой из них должен быть свой файл package.json, плюс один в корневой папке проекта! **Итого 3 фала package.json!** Для этого в каждой папке запускаем npm init

НАСТРОЙКА СЕРВЕРА:

Будем запускать сервер на порте 8000 в режиме разработки, но в режиме деплоя пусть это будет ПЕРЕННАЯ КРУЖЕНИЯ:

const PORT = process.env.PORT || 8000

Как это проверить? Мы можем написать порт в package.json и при запуске переменная будет меняться на порт 5000:

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "watch": "nodemon src/server.js",

    "start": "PORT=5000 node src/server.js"

  },

**Создадим сервер с помощью http модуля и передадим в него приложение express. Теперь все, что мы передаем в express будет на сервере. Также теперь мы можем вынести приложение в отдельный стартовый файл app.js**

const http = require('http')

const express = require('express')

const app = express()

const PORT = process.env.PORT || 8000

const server = http.createServer(app)

server.listen(PORT, () => {

    console.log(`Listening on ${PORT}...`);

})

**Вынесем создание приложения в отдельный файл и импортнем его server.js**

const http = require('http')

const app = require('./app')

const PORT = process.env.PORT || 8000

const server = http.createServer(app)

server.listen(PORT, () => {

    console.log(`Listening on ${PORT}...`);

})

**Таким образом можно сказать, что Express – это просто middleware в родном http сервере ноды**

Создадим **РОУТЕР**, который по сути **тоже middleware**

const express = require("express");

const planetsController = require("../planets.controller.js")

const planetsRouter = express.Router();

planetsRouter.get('/planets', planetsController.getAllPlanets)

module.exports = planetsRouter

Для фронта напишем первый запрос:

const API\_URL = 'http://localhost:8000'

async function httpGetPlanets() {

  // TODO: Once API is ready.

  // Load planets and return as JSON

  const response = await fetch(`${API\_URL}/planets`)

  return await response.json()

}

НО запрос сейчас не пройдет из-за ошибки CORS. На фронте и на бэке разные ориджины (порты 5000 и 3000), поэтому будем использовать **CORS MIDDLEWARE**

**101. CORS middleware**

В express есть middleware, в котором можно отключить корсы, либо прописать whitelist. Установим:

*npm install cors*

Добавим как middleware, пропишем в whitelist ориджин ФРОНТА (полную конфигурацию см. в доке):

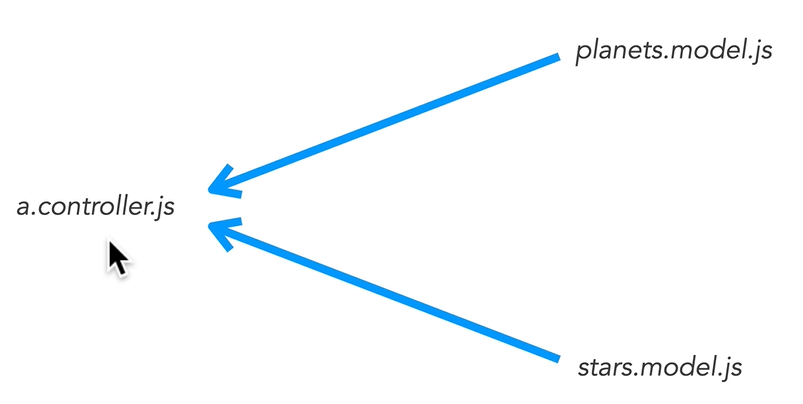
app.use(cors({

    origin: 'http://localhost:3000'

}))

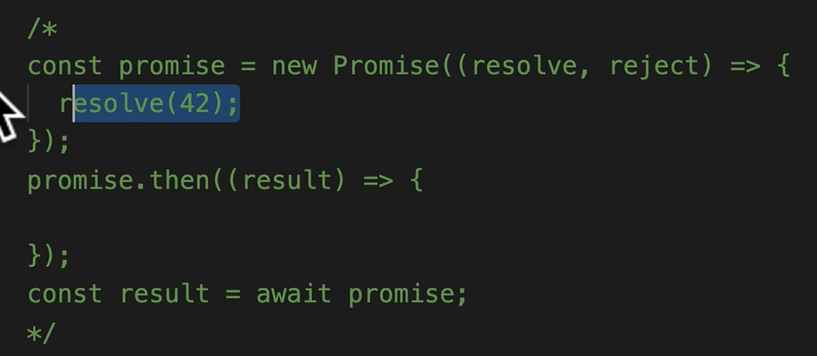
**110. Models/Controllers/Routers**

В контроллере могут быть несколько моделей, одна модель может участвовать в нескольких контроллерах:



**110. Load(Parse) data on SERVER startup!!!**

Если во время запуска сервера надо что-то спарсить/подгрузить, это надо делать асинхронно. Для этого воспользуемся промисами. Для получения резолва от промиса можно использовать метод .then, можно await:



Завернем функцию, которая парсит данные из файла в промис:

async function loadPlanetsData() {

    return new Promise((resolve, reject) => {

        fs.createReadStream(path.join(\_\_dirname, '..', '..', 'data', 'kepler\_data.csv'))

            .pipe(parse({

                comment: '#',

                columns: true,

            }))

            .on('data', (data) => {

                if (isHabitablePlanet(data)) {

                    habitablePlanets.push(data);

                }

            })

            .on('error', (err) => {

                console.log(err);

                reject(err)

            })

            .on('end', () => {

                console.log(`${habitablePlanets.length} habitable planets found!`);

                resolve()

            });

    })

}

module.exports = {

    loadPlanetsData,

    planets: habitablePlanets

}

Далее в файле sever.js будем «ждать» пока промис зарезолвится:

const http = require('http')

const app = require('./app')

const { loadPlanetsData } = require('./models/planets.model')

const PORT = process.env.PORT || 8000

const server = http.createServer(app)

async function startServer() {

    await loadPlanetsData() // ждем, пока зарезолвится

    // сервер, затем начинаем слушать порт

    server.listen(PORT, () => {

        console.log(`Listening on ${PORT}...`);

    })

}

startServer()

АЛЬТЕРНАТИВА – начиная с ноды версии 15 есть модуль require(‘stream/promises’), который можно использовать вместо промисов.

**114. Автоматизация Fullstack приложения с помощью NPM**

Сейчас приходится запускать каждое приложение отдельно – фронт и бэк. Можно автоматизировать запуск.

Проинициализируем package.json в корневой папке проекта. Для одновременного запуска клиента и сервера будем использовать либу **concurrently** (есть и другие способы).

Пропишем в корневом package.json следующие скрипты и запускаем приложение **npm run start**:

  "scripts": {

    "server" : "cd server && npm run watch", //cd в папку server и запуск скрипта

    "client" : "cd client && npm run start",//cd в папку client и запуск скрипта

    "start": "concurrently \"npm run server\" \"npm run client\"", // объединение

  },

&& - выполнение команд последовательно, запуск второй только после окончания первой

& - запуск параллельно, не ждет, пока первая закончится

**115. SERVING ReactJS Front-end into production**

Для этого надо:

Файлы из папки билда фронта копировать в папку сервера:

Скрипт на Клиенте:

  "scripts": {

    "start": "react-scripts start",

    "build": "set BUILD\_PATH=../server/public && react-scripts build",

    "test": "react-scripts test",

    "eject": "react-scripts eject"

  },

**Теперь код Клиента, оптимизированный под прод, находится рядом с кодом Сервера**

**Для работы всего приложения надо внести изменения в app.js – добавить middleware, который будет считывать статические данные(static):**

**Теперь можно запускать только сервер и будет работать фронт+бэк (ЭТО ИСПОЛЬЗУЕТСЯ ДЛЯ ПРОДА)!**

Пропишем скрипт для деплоя, который будет билдить фронт и запускать сервер (переход по папкам можно сделать с помощью --prefix):

"deploy": "cd client && npm run build && cd .. && cd server && npm run watch",

**117. Логирование запросов**

**С помощью MORGAN**

Логи запросов обычно записываются в отдельный файл, этот файл может достигать огромных размеров.

Установим *npm install morgan* в папке сервера.

Т.к. это мидлвар, добавим его в цепочку мидлваров в файле приложения app.js:

const path = require('path')

const express = require('express')

const cors = require('cors')

const morgan = require('morgan')

const planetsRouter = require("./routes/planets/planets.router")

const app = express() // поток данных идет последовательно сверху вниз

app.use(cors({

    origin: 'http://localhost:3000'

}))

app.use(morgan('combined')) // логирует запросы

app.use(express.json()) //парсит входящий json

app.use(express.static(path.join(\_\_dirname, '..', 'public')))

app.use(planetsRouter) // если путь совпадает, выполняет функцию

app.get('/', (req, res) => {

    res.sendFile(path.join(\_\_dirname, '..', 'public', 'index.html'))

})

module.exports = app

В мидлваре опишем формат логов – ‘combined’, который мы хотим получить.

**120. ПРОБЛЕМА: одинаковые роуты на клиенте и на сервере**

**Если в браузере в адресной строке ввести путь, который мы не прописали на бэке, Express кинет ошибку,** потому что не находит роут для get запроса. Чтобы избавиться от ошибки, надо просто добавить \* в запрос, который выдает Index.html файл. Таким образом, Express будет проходить по роутерам и, если не найдет соответствующий роут на бэке, он не будет кидать ошибку, а будет выдавать index.html в деплое:

app.get('/\*', (req, res) => {

    res.sendFile(path.join(\_\_dirname, '..', 'public', 'index.html'))

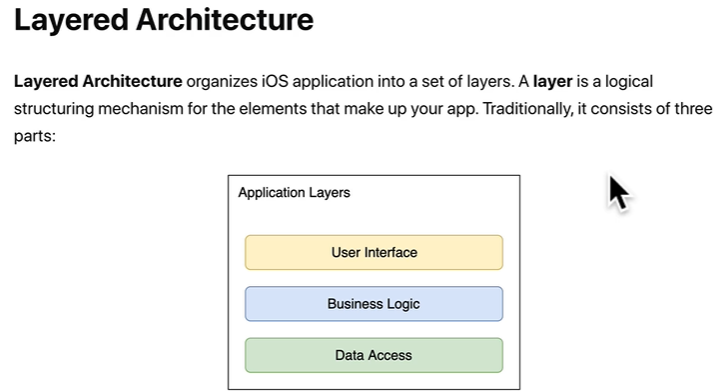
})

**При этом React Router считает роут и покажет нужную страницу!**

**121. Data Models: Building Data Access Layer (DAL)**

**LAYERED ARCHITECTURE – реализация принципа SEPARATION of CONCERNS**

**Как четко разграничить функциональность, которая должна быть в model, а какая в controller?**

В идеальном случае, контроллер не должно волновать, как хранится и обрабатываются данные в модели (будь это база данных, облако и пр.). **Контроллер** должен работать только с req и res и **НЕ ПРОИЗВОДИТЬ НИКАКИХ ДЕЙСТВИЙ С MODEL**, а model должен предоставлять только методы для чтения/записи данных. **НАПРЯМУЮ ИМПОТИРОВАТЬ МОДЕЛЬ В КОТРОЛЛЕР НЕЛЬЗЯ. НУЖНО ЛИБО СОЗДАТЬ ФУНКЦИЮ, КОТОРАЯ БУДЕТ ВОЗВРАЩАТЬ МОДЕЛЬ, ЛИБО КЛАСС.**

Пример из BP:

export class FeedbackController {

  constructor(model = defaultModel, adminPermission = USER\_PERMISSION.ADMIN) {

    this.model = model;

    this.adminPermission = adminPermission;

    this.skipGlobalClear = adminPermission === USER\_PERMISSION.ADMIN

      ? false

      : null;

  }

  listFeedbackQuestions = async (context) => {

    const results = await defaultModel.listFeedbackQuestions(context);

    return { results };

  };

  updateQuestion = async (context, { question }) => defaultModel

    .updateQuestion(context, { question });

. . .

. . .

. . .

export default new FeedbackController();

**122. PUT (POST) planet**

**- Начнем разработку с MODEL**

**124. Валидация POST запроса**

**Все входящие запросы надо валидировать. Валидировать будем в конроллере. В BP валидация в Хелпере?**

**Валидация даты/времени делается через isNaN():**

**BP:**

 if (Number.isNaN(date.getTime())) {

        throw new Error(`Property "${property.label}" must be a date`);

      }

**Курс:**

if (isNaN(launch.launchDate)) {

        return res.status(400).json(

            { error: 'Invalid launch date' }

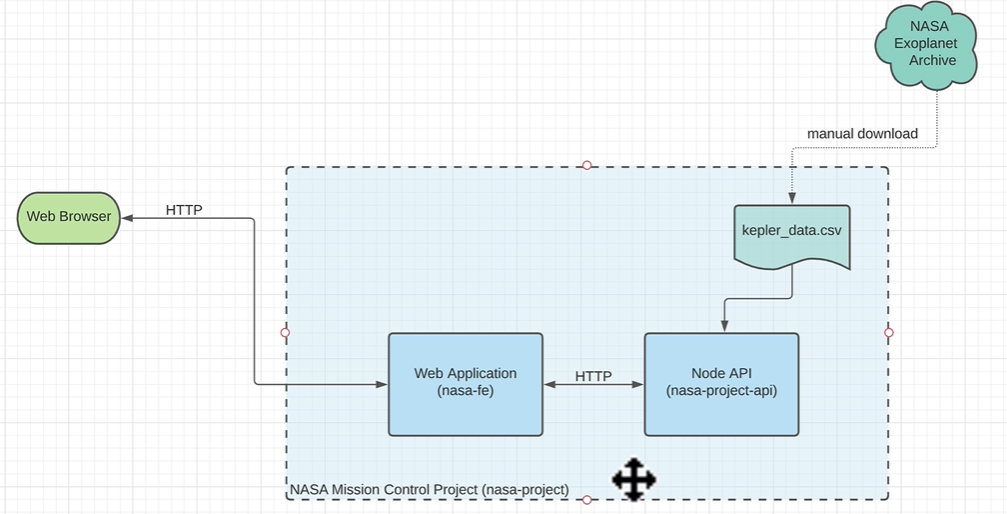
        );

    }

**122. Удаление (запрос с query param)**

**В контроллере проверяем, существует ли такой launch (НЕ В МОДЕЛИ). Но в МОДЕЛИ создадим функцию, которая будет возвращать элемент, если он найден.**

**Общая схема приложения (такие можно рисовать в Lucid):**



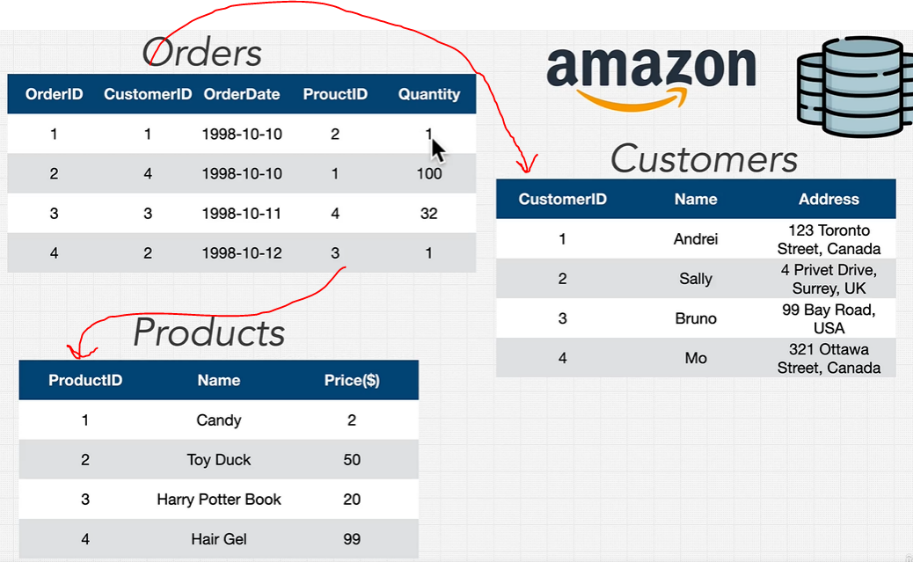
10. TESTING APIs (не прошел)

11. IMPROVING NODE PERFORMANCE (не прошел)

12. DATABASES

**Database** – collection of data. База данных обособлена от сервера и не является его частью. Сервет отправляет запросы на получение, добавление, апдейт данных в БД. В ответ на запрос БД отвечает серверу.

**Основные типы БД (SQL и NoSQL)**

**SQL**  Structured Query Language **(данные хранятся в формате таблиц) – декларативный язык**

- У каждой строки (записи есть id)

- Таблицы взаимосвязаны между собой различными id

**5 типов ДБ:**

1. **Relational Model** (MySQL, Postgres)
2. **Document Model** (MongoDB, FireBase)
3. **KeyValue** (Redis)
4. **Graph Model**
5. **Wide Columnar Model**

SQL Playgrounds:

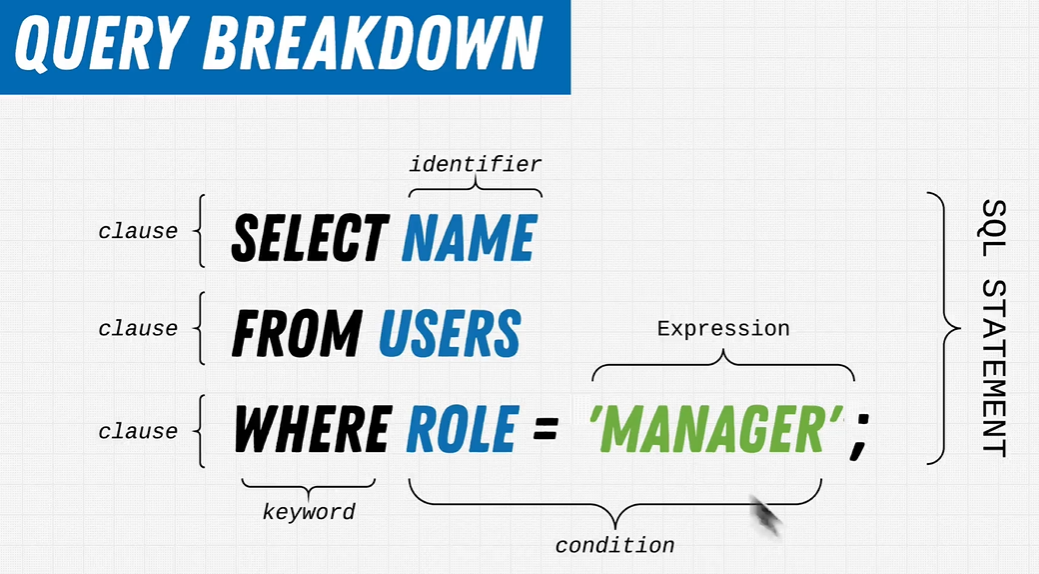
<https://www.w3schools.com/sql/trysql.asp?filename=trysql_asc>

<https://www.db-fiddle.com/>

**Queries SQL**

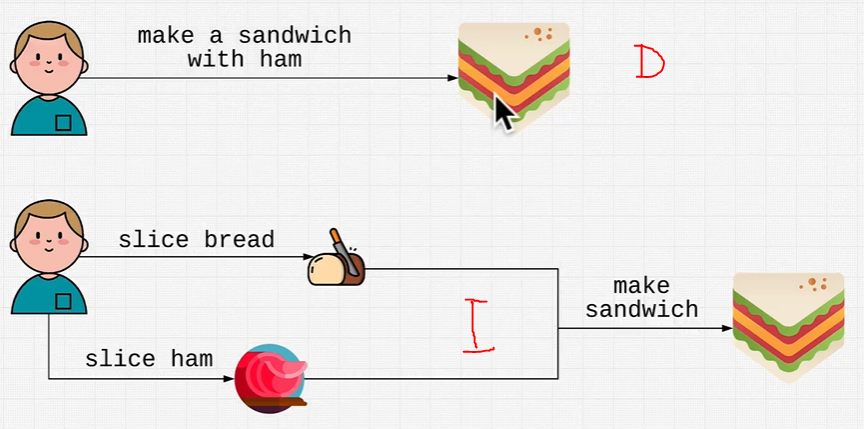
SELECT \* FROM USERS ( ‘select all data from users table’)

Запрос SQL может состоять из трех частей:



**Декларативный язык(sql)** – мы говорим, **ЧТО** должно произойти

**Императивный язык(java)** – мы говорим, **КАК** должно произойти



**Язык SQL стандартизирован**, но многие компании, для того, чтобы кастомизировать БД под себя, делают поверх свою стандартизацию.

**Таблицы**

У каждой таблицы должно быть уникальное имя.

Таблица состоит из колонок и рядов.



**Колонки**

Каждая колонка должна иметь имя, в колонке хранится только один тип данных.

Ряд колонок называется **DEGREE (of the relation)**



Тип данных, который может храниться в БД это **DOMAIN** (дата, имя, фамилия, число).

Все значения в колонках – это **ATTRIBUTES**

**Ряды (tuples)**

**Tuple** – одна запись в таблице

Все tuples в таблице называются **cardinality**

**Primary and Foreign keys**

С помощью **ключей** строятся **зависимости** между таблицами

**PRIMARY KEY** – уникальный ключ строки в таблице

**FOREIGN KEY** – ключ, который дает ссылку на другую таблицу с **primaryKey === foreignKey**



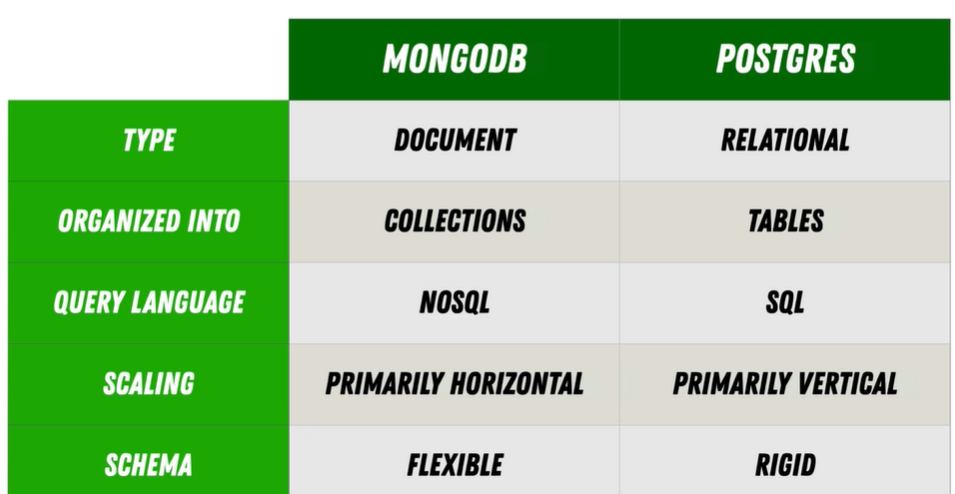
**Relational DB(MongoDB) vs NoSQL(PostgreSQL)**

**NoSQL** – Document Model DB (запись – это объект в формате JSON). В поле можно добавить массив элементов, что в SQL сделать нельзя, там надо создавать отдельную таблицу. В NoSQL данные могут дублироваться, и обновлять их надо везде. Тогда как в SQL данные содержатся только в одном месте таблице)

Примеры: соцсеть – лучше выбрать noSQL, т.к. там будет один юзер и все остальные данные обрастают вокруг него.

Блог – много разных сущностей с одинаковым весом, лучше выбрать SQL

**Миграция** – переход с одной схемы на другую





**Масштабируемость**

**Вертикальное масштабирование** – добавление памяти, больше мощности БД(серверу). Все про улучшение ОДНОЙ физической машины.

**Горизонтальное масштабирование** – увеличение количества физических машин (легко NoSQL, сложно SQL).

**Sharding (фрагментация)**

**Sharding -** разделение большой базы на мелкие. Например, разбиение юзеров по алфавиту на несколько серверов.

Выбираем БД для проекта

**Добавление БД в проект(создание кабинета и настройка кабинета)**

**API Должен быть stateless, т.е. нода должна работать только с данными, которые приходят в реквесте и данными, которые есть в БД. Никакого внутреннего состояния быть не должно!!!**

Для работы с SQL в объектно-ориентированных языках могут возникнуть сложности, называемые *object-relational impedance mismatch*. Она заключается в том, что в ооп должна быть описана схема relations такая же, как в БД. Типы должны соответствовать. С NoSQL это проще, т.к. по сути коллекция — это объект json(bson), который намного проще описать, чем взаимосвязи таблиц:

<https://en.wikipedia.org/wiki/Object%E2%80%93relational_impedance_mismatch>

Установим БД к себе. Будем использовать бесплатный Community Server (облако)

* Далее создаем кластер
* Далее создаем user:

**User**: nasa-api

**Pass**: gU1LARKvIQ3uxP2W

* Далее добавим ip, которые будут иметь доступ к БД. В продакшене мы будем указывать конкретный ip сервера. Пока что наш сервер – это наша локальная машина, так что укажем ip нашего компа. Но в целях обучения дадим доступ с любого api, поэтому укажем 0.0.0.0/0
* Дале выбираем Connect и выбираем Connect your Application, т.к. работаем с Нодой.
* Выбираем версию ноды
* Копируем строку доступа

**Добавление БД в Ноду (установка Mongoose)**

Для работы с MongoDB будем использовать пакет **Mongoose**.

- Добавим константу MONGOOSE\_URL как connection string:

const http = require('http')

const app = require('./app')

const { loadPlanetsData } = require('./models/planets.model')

const PORT = process.env.PORT || 8000

const MONGO\_URL = `mongodb+srv://nasa-api:gU1LARKvIQ3uxP2W@nasacluster.y5uozou.mongodb.net/?retryWrites=true&w=majority`

const server = http.createServer(app)

* Установим пакет в server:

*npm install mongoose*

* Добавим подключение БД при старте сервера:

const http = require('http')

const mongoose = require('mongoose')

const app = require('./app')

const { loadPlanetsData } = require('./models/planets.model')

const PORT = process.env.PORT || 8000

const MONGO\_URL = `mongodb+srv://nasa-api:gU1LARKvIQ3uxP2W@nasacluster.y5uozou.mongodb.net/nasa?retryWrites=true&w=majority`

const server = http.createServer(app)

mongoose.connection.once('open', () => {

    console.log('MongoDB connection ready...');

})

mongoose.connection.on('error', (err) => {

    console.error(err)

})

async function startServer() {

    await mongoose.connect(MONGO\_URL)

    await loadPlanetsData()

    server.listen(PORT, () => {

        console.log(`Listening on ${PORT}...`);

    })

}

startServer()

**OBJECT MODELING Mongoose**

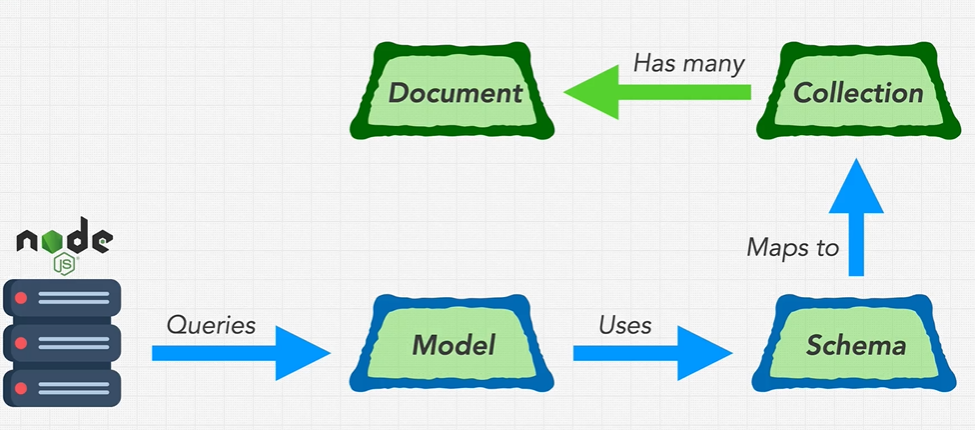
В MongoDB данные хранятся в **Collections** (коллекциях).

Коллекции состоят из нескольких или 0 (пустая коллекция) **Documents** (документов). Документ — это “объект”, в котором хранятся данные в виде BSON (JSON like data).

**Schema objects** (схема) - описывают структуру (типизирует) Коллекции. Это нужно в том числе для **валидации** данных (живет в mongoose).

**Models (JS object, с которым мы будем работать в Ноде)** – имплементация схем, модель использует схемы.

Из НОДЫ мы делаем **запрос (crud) на модель**



**Создание схемы Schema**

* Создадим файл схемы рядом с моделью
* Опишем схему launches в этом файле (на сайте mongoDB есть все типы данных SchemaTypes для схемы):

const launchesSchema = mongoose.Schema(

    {

        flightNumber: {

            type: Number,

            required: true,

        },

        mission: {

            type: String,

            required: true,

        },

        rocket: {

            type: String,

            required: true,

        },

        launchDate: {

            type: Date,

            required: true,

        },

        target: {

            type: String,

            required: true,

        },

        customers: [String],

        upcoming: {

            type: Boolean,

            required: true,

        },

        success: {

            type: Boolean,

            required: true,

            default: true

        },

    }

)

**Создание Model из схемы Schema**

Чтобы создать схему воспользуемся методом model:

const mongoose = require('mongoose')

const planetsSchema = mongoose.Schema({

    keplerName: {

        type: String,

        required: true,

    }

})

module.exports = mongoose.model('Planet', planetsSchema)

**Launch** – **имя Документа (?)** (**ОДНОЙ** сущности, всегда будет в ед. числе. Поэтому имя не Launches). Mongoose переформатирует потом это имя в множественное число малыми буквами – “**launches”. Т.е. будет коллекция launches.**

mongoose.model('Launch', launchesSchema) – **возвращает объект**, с которым мы можем работать. Импортируем его в файл model:

**Mongoose models VS MVC**

Mongoose Model – это не совсем то же, что и модель в MVC.

Schema и Model – это объекты/классы, которые предоставляет нам MongoDB, чтобы обращаться к коллекциям в БД.

Model MVC – это более общее понятие, касающееся всех типов БД.

**Finding, Creating and Inserting Documents в Коллекцию**

Для манипуляции с документами в MongoDB будем использовать встроенные методы model.

Эта функция вызывается каждый раз, когда сервер стартует, это не правильно. MongoDB Предоставляет функцию insert + update = **upsert**

**INSERT (upsert)** срабатывает только тогда, когда документа такого же нет. Это помоет избежать дублирования при перезапуске сервера.

Импортируем planets model в файл model, добавим документ:

const planets = require('./planets.mongo')

…

            .on('data', async (data) => {

                if (isHabitablePlanet(data)) {

                    // habitablePlanets.push(data); // БЫЛО

                    await planets.create(data);// создаем документ

                }

            })

**FIND** – см. полную доку на сайте Mongoose <https://mongoosejs.com/docs/api/model.html>

async function getAllPlanets() {

    return await planets.find({}) // {} - вернет все записи в БД

}

**Все операции Mongoose асинхронные!!! Поэтому надо не забыть сделать функции async/await, которые импортируют операцию Mongoose. В данном случае она импортируется в контроллер**

async function httpGetAllPlanets(req, res) {

    return res.status(200).json(await getAllPlanets())

}

**UPSERT** – если документа нет, добавляет его (insert), если он есть – обновляет (update)

            .on('data', async (data) => {

                if (isHabitablePlanet(data)) {

                    await planets.updateOne(

                        {

                            keplerName: data.kepler\_name // фильтр

                        },

                        {

                            keplerName: data.kepler\_name // это добавим/обновим

                        },

                        {

                            upsert: true // режим Upsert

                        });

                }

            })

И вынесем в отдельную функцию + добавим обработку ошибок:

async function savePlanet(planet) {

    try {

        await planets.updateOne(

            { keplerName: planet.kepler\_name },

            { keplerName: planet.kepler\_name },

            { upsert: true }

        );

    } catch (error) {

        console.error(`Could not save planet ${err}`)

    }

}

По окончании запроса будем резолвить по длине массива найденных планет:

 .on('end', async () => {

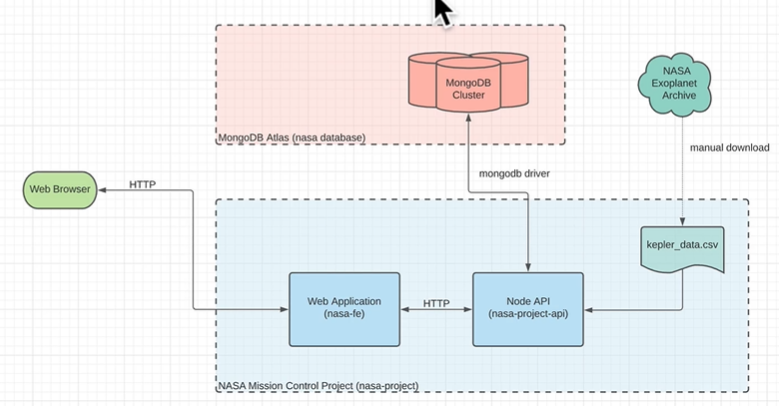
                const countPlanetsFound = (await getAllPlanets()).length

                console.log(`${countPlanetsFound} habitable planets found!`);

                resolve()

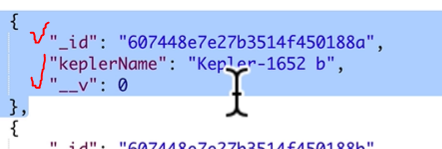
            });

**ОБНОВИМ СХЕМУ АРХИТЕКТУРЫ ПРИЛОЖЕНИЯ**



**Что такое ObjectID ???**

Есть поля, которые MongoDB генерирует автоматически:



**\_id – это и есть ObjectID. Значение этого поля всегда будет уникальным в этом во всех инстансах кластера.** Он содержит в себе **timestamp. Т.е. его можно конвертировать в дату/время и узнать, когда он был создан.**

**\_\_v – это version key, версия схемы, есл поменяется схема, поменяется и значение этого поля. Его автоматически добавляет mongoose.**

**Исключение полей из респонза**

**Исключим поля \_id и \_\_v из респонза. Для этого в метод find БД добавим projection:**

async function getAllPlanets() {

    return await planets.find({}, {

        '\_\_v': 0,

        '\_id': 0

    })

}

**Сохранение Launch в БД**

Для этого опять будем использовать **updateOne** с опцией **upsert: true**

**Referential integrity (Ссы́лочная це́лостность)**

Primary и Foreign ключи должны быть синхронными. Если вносятся изменения, связанные ключи должны быть везде одинаковые – БД их сама обновляет, либо запрещает такое изменение.

Следование этому правилу – сильная сторона реляционных БД, в нашем случае есть несколько подходов, которые мы должны реализовать самостоятельно.

**Например, мы хотим добавить launch, в поле которого будет имя планеты. Но мы не можем добавить новый launch с планетой, которая не существует в БД. Это суть проблемы.**

1. **Можно добавить валидацию перед сохранением** – проверить, есть ли такая планета в БД, перед тем как сохранить/апдейтнуть launch.

async function saveLaunch(launch) {

    const planet = await planets.findOne({

        keplerName: launch.target

    })

    if (!planet) {

        throw new Error('No matching planets found')

    }

    await launchesDatabase.updateOne(

        { flightNumber: launch.flightNumber },

        launch,

        { upsert: true },

    )

}

1. **….**

**Auto increment in MongoDB**

Когда мы сохраняем новый launch, нам надо увеличивать latestFlightNumber на 1 каждый раз.

В MySQL есть встроенная фича AUTO\_INCREMENT, в PostgreSQL есть такая же SERIAL

Несколько БД объединены в **Кластер** – В БД есть коллекция документов.

Один из подходов к решению данной проблемы это реализация сохранения порядкового номера **отдельным процессом** (разбиение более сложной задачи на простые).

Для этого создадим отдельную функцию:

async function getLatestFlightNumber() {

    const latestLaunch = await launchesDatabase

        .findOne()

        .sort('-flightNumber').flightNumber // встроенный метод MongoDB

    if (!latestLaunch) {

        return DEFAULT\_FLIGHT\_NUMBER // если не найдено, начинаем с 100 создавать

    }

    return latestLaunch.flightNumber

}

“-“ – сортировка в обратном порядке.

.findOne() – возвращает первый элемент в коллекции.

**Методы в Mongoose работают «вместе». Т.е. сортировка работает совместно с findOne, а не последовательно. Т.е. если есть сортировка в методах, поиск будет выполнен в отсортированных документах.**

Далее определим функцию (отдельную от сохранения), которая будет инкрементить/сетать latestFlighNumber:

async function scheduleNewLaunch(launch) {

    const newFlightNumber = await getLatestFlightNumber() + 1

    const newLaunch = Object.assign(launch, {

        success: true,

        upcoming: true,

        customers: ['Zero to Mastery', 'NASA'],

        flightNumber: newFlightNumber,

    })

    saveLaunch(newLaunch)

}

И приведем в соответствие функцию в контроллере:

async function httpAddNewLaunch(req, res) {

    const launch = req.body;

    if (

        !launch.mission ||

        !launch.rocket ||

        !launch.launchDate ||

        !launch.target

    ) {

        return res.status(400).json({ error: "Missing required launch property" });

    }

    launch.launchDate = new Date(launch.launchDate);

    if (isNaN(launch.launchDate)) {

        return res.status(400).json({ error: "Invalid launch date" });

    }

    await scheduleNewLaunch(launch);

    return res.status(201).json(launch);

}

**Апдейт записей в MondoDB**

В запросы мы получаем flightNumber, по нему мы можем проверить, есть ли такой документ в коллекции:

Model:

async function existsLaunchWithId(launchId) {

    return await launchesDatabase.findOne({

        flightNumber: launchId

    })

}

Там же у удаляемой записи будем апдейтить поля:

async function abortLaunchById(launchId) {

    const aborted = await launchesDatabase.updateOne({

        flightNumber: launchId

    }, {

        upcoming: false,

        success: false

    })

    return aborted.modifiedCount === 1 // modifiedCount приходит по дефолту от mongoDB

}

Далее в контроллере при успешном аборте будет отправлять в качестве респонза ok: true

async function httpAbortLaunch(req, res) {

    const launchId = Number(req.params.id);

    // проверяем, есть ли такой лаунч

    const existsLaunch = await existsLaunchWithId(launchId)

    if (!existsLaunch) {

        return res.status(404).json({ error: "Launch not found" });

    }

    // пробуем отменить лаунч (апдейт, не удаление)

    const aborted = await abortLaunchById(launchId);

    if (!aborted) {

        return res.status(400).json({ error: "Launch not aborted" });

    }

    // если все ок - отправляем ok на фронт

    return res.status(201).json({

        ok: true

    });

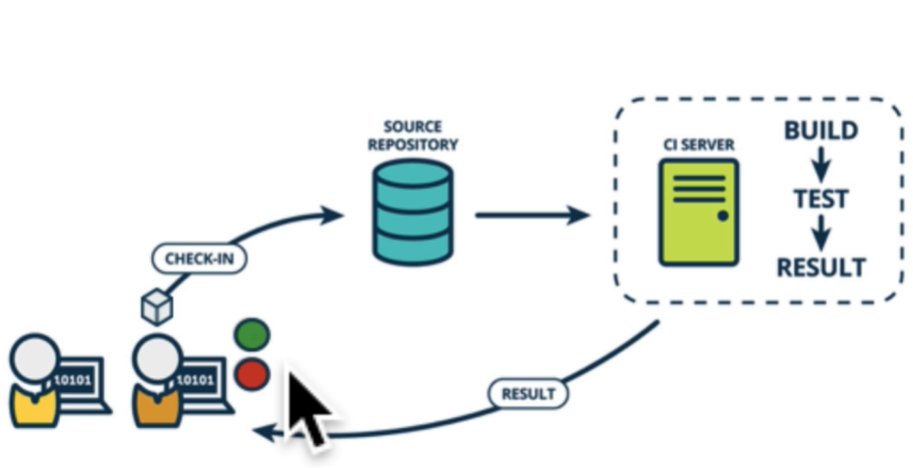
}

15. CI / CD

**Введение**

CI – это один из best practice, когда новые изменения в коде непрерывно интегрируются в общий проект с проверкой того, что все работает и тесты проходят. Вместо того, чтобы тестить на ошибки и проверять стабильно прямо перед релизом, все проверки качества должны быть постоянными и происходить в процессе разработки. Также код проверяется на то, чтобы он был работоспособным на разных окружениях и ОС.

Для этого создается CI сервер, которых делает необходимы проверки: прогоняет тесты, собирает билд, линтер, проверка секьюрности и пр. И по результатам процесса, дает ответ, прошел CI или упал



15. GraphQL

**Введение**

GarphQL – язык запросов.

У GraphQL своя система типов. Для работы с GraphQL нужен только один POST end-point.