**5. Fundamentals.**

Axios simple example:

const axios = require('axios')

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

    console.log(res);

}).catch((err) => {

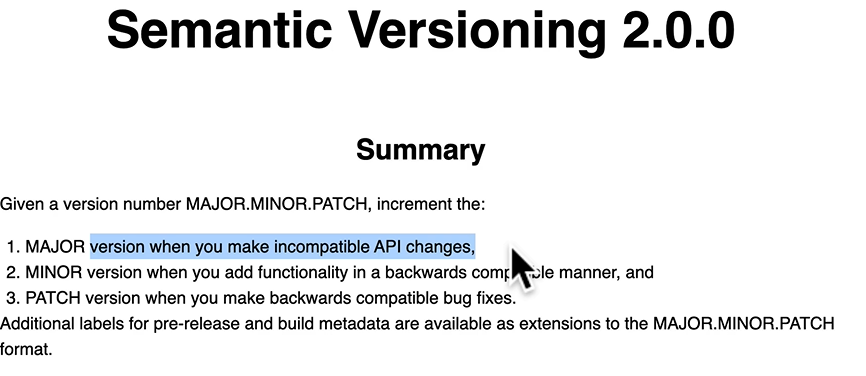
    console.log(err);

}).then(() => {

    console.log('All done');

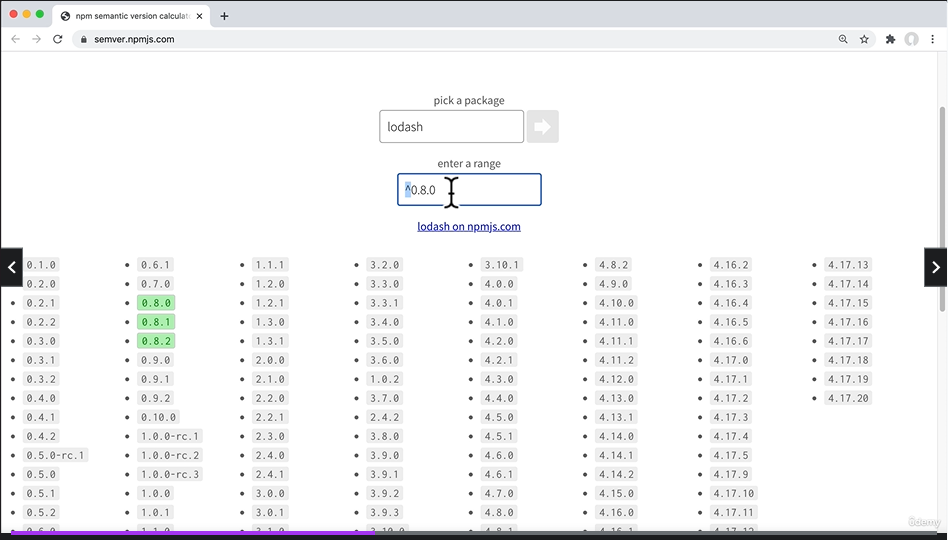
})

**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)=>{

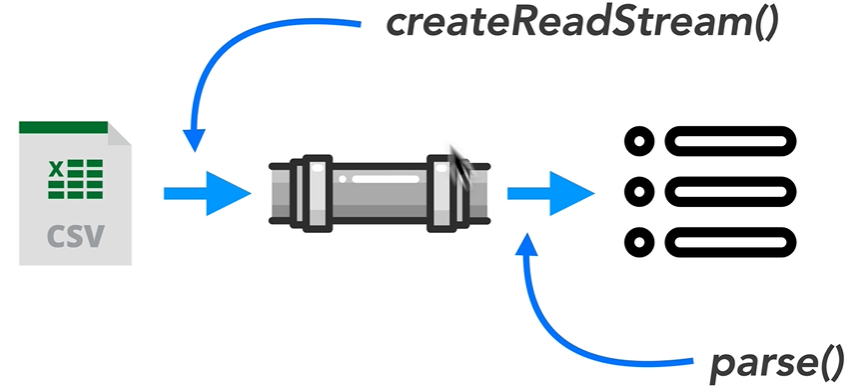
            return planet['kepler\_name']

        }));

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

        console.log('done');

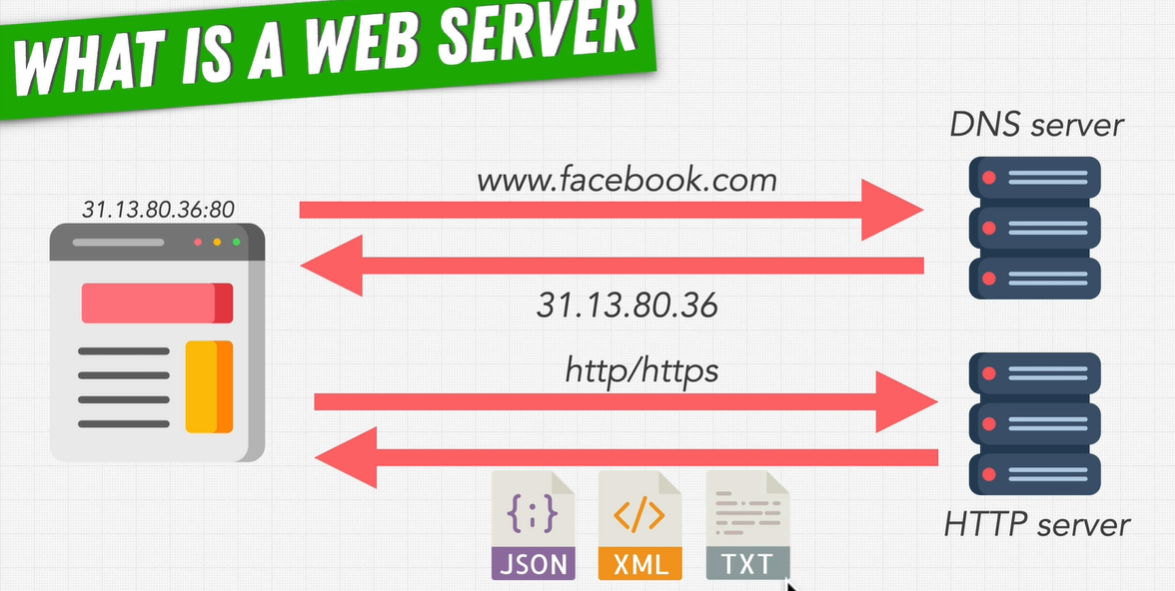
    })



WEB SERVERS

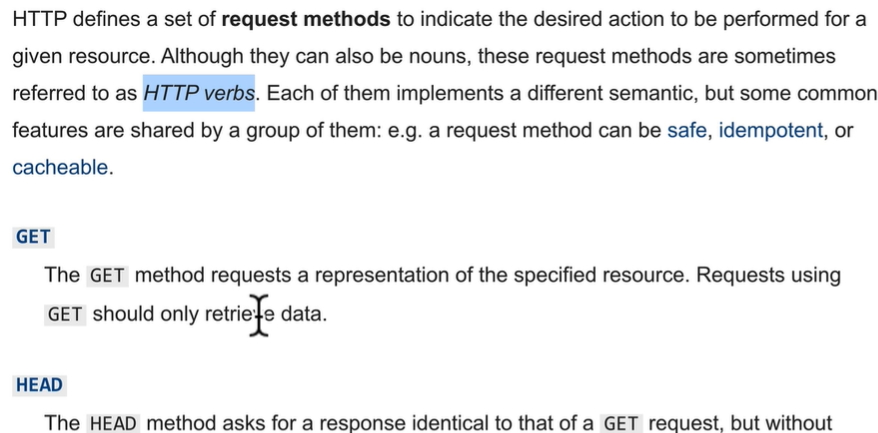
**68. Поток данных**

Схема работы сети:

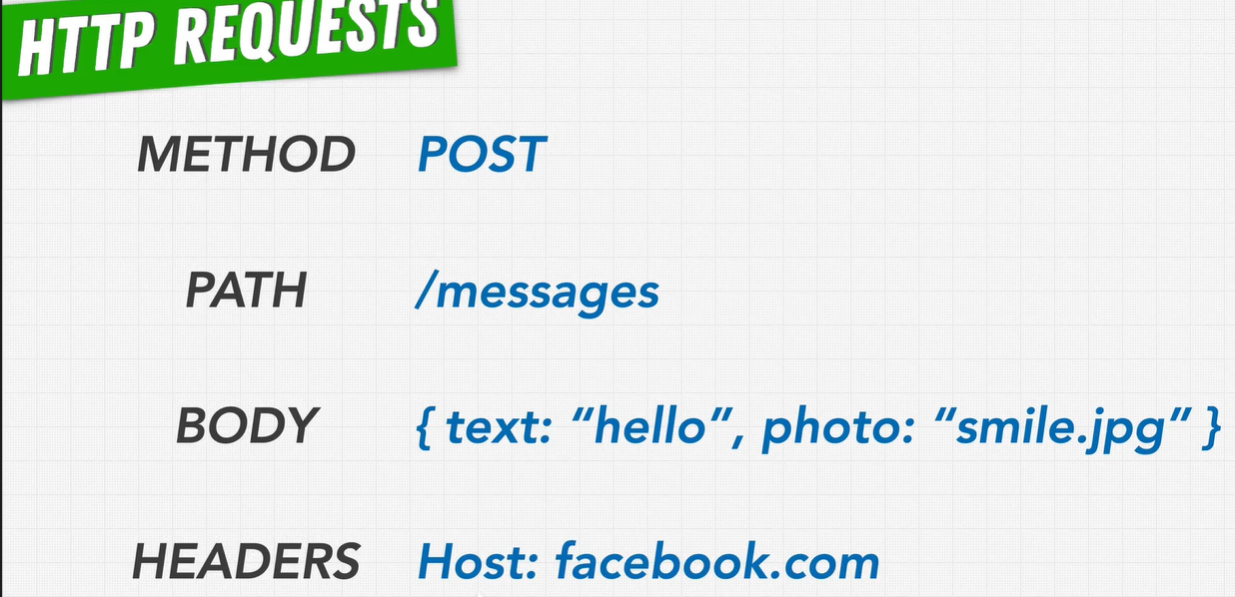


**69. Запросы и ответы**

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

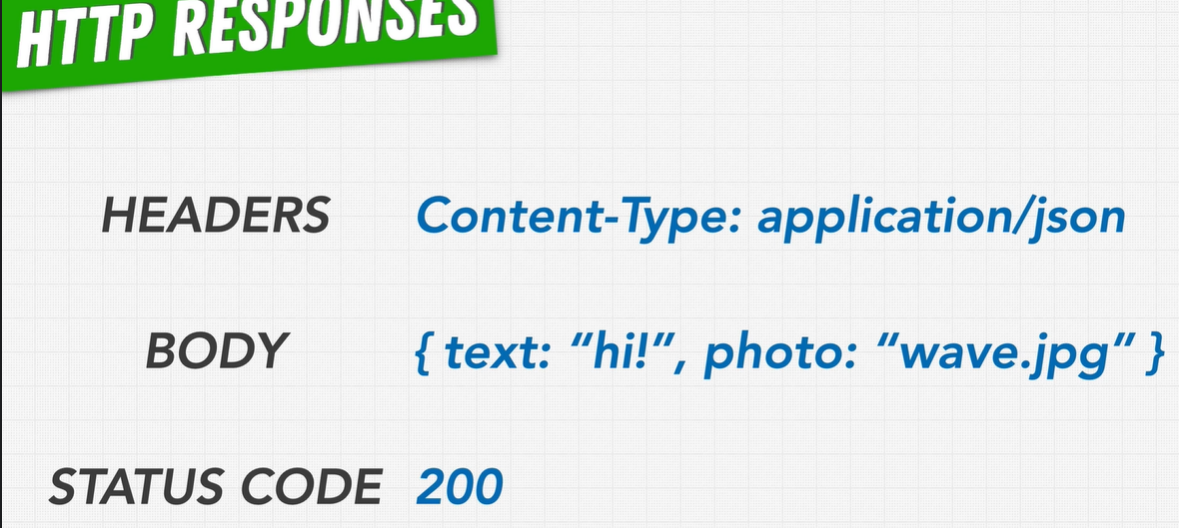


**Запрос на сервер состоит из:**



Host – обязательно

**Ответ с сервера состоит из:**



**72. Простой Веб-сервер**

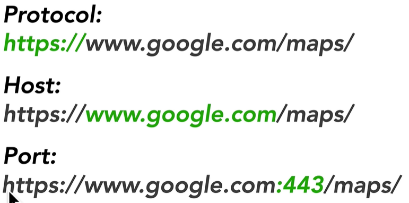
* **Передавать данные можно только в виде строки JSON.stringify()**
* **Тип данных определяется в Content-Type:**

res.writeHead(200, {

        'Content-Type': 'application/json'

    })

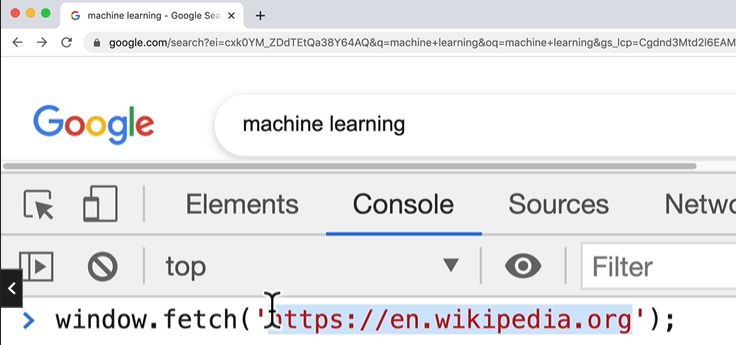
**74. SAME origin policy**

**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):

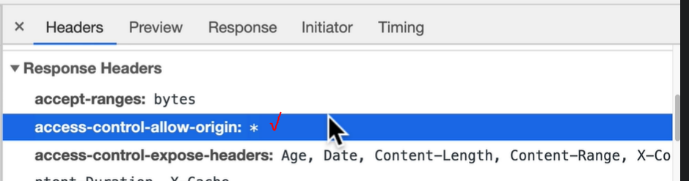
**Одинаковыми можно считать только те два ориджина, у которых одинаковая комбинация этих трех элементов (SAME ORIGIN POLICY).** Т.е. если разный протокол – разные ориджины.

**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)

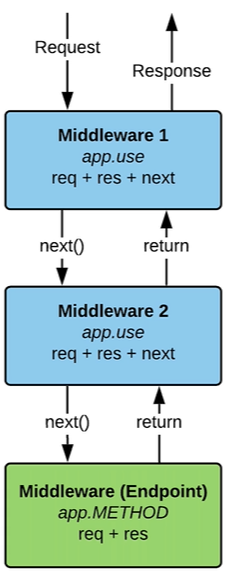
    else res.status(404).json({

        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)

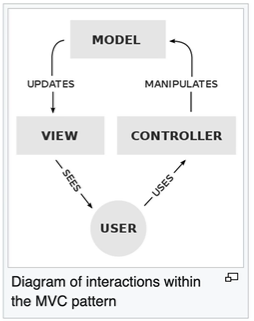
    res.status(200).json(newFriend)

})

**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.

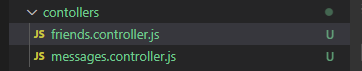
В **MODEL (БД)** происходят изменения в базе данных, в ней также функции, с помощью которых мы взаимодействуем с БД

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) {

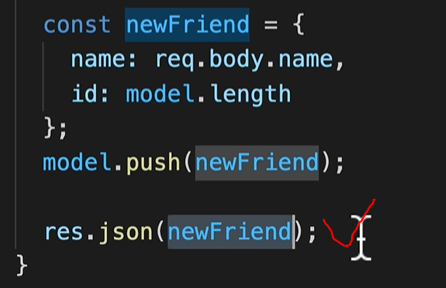
        return res.status(400).json({

            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);

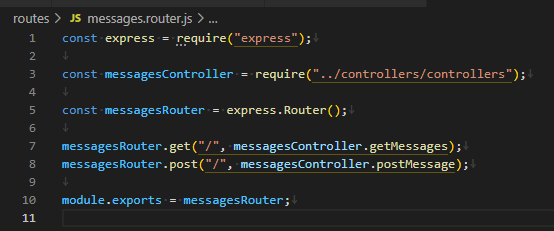
friendsRouter.get("/", friendsController.getFriends);

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)

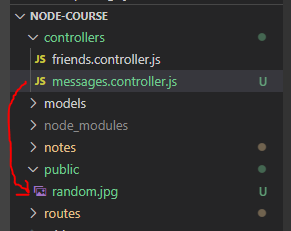
**96. Отправка Файла**

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:

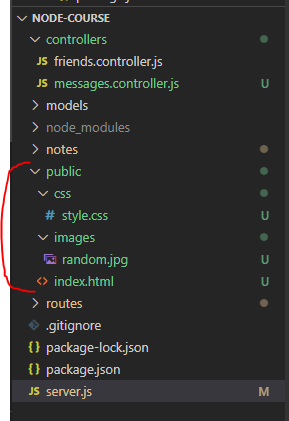
res.sendFile(path.join(\_\_dirname, '..', 'public', 'random.jpg'))

**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)**

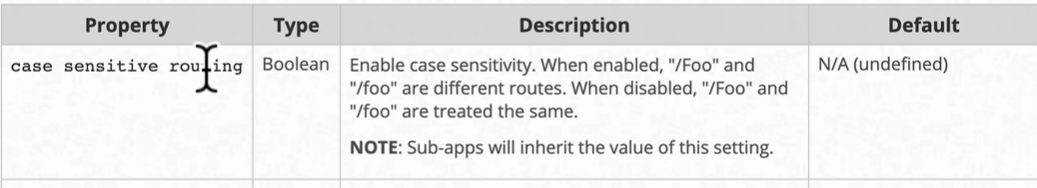
В Node js их много, будем использовать **handlebars (hbs). С помощью Template Engines в шаблоны html можно вставлять переменные в {{}} скобки для динамически изменяемого контента ДО отправки на клиента.**

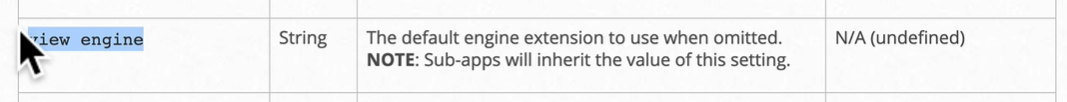
Установим пакет **hbs:**

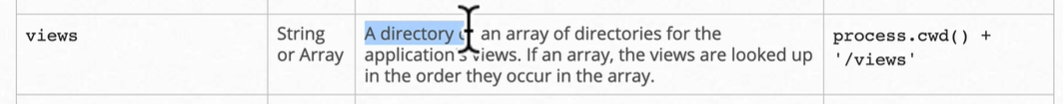
*npm install hbs*

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'

    })

})