



# Разработка веб-приложений. Node.js

Практические работы



## **Упражнение 1. Установка Node.js и среды разработки**

---

### **О чем это упражнение:**

В этой лабораторной работе Вы установите Node.js и среду разработки Visual Studio Code.

### **Что Вы должны будете сделать:**

1. Установить Node.js
2. Установить Visual Studio Code

## **Раздел 1. Установка Node.js**

Подробные инструкции по установке Node.js могут отличаться в зависимости от используемой операционной системы. На сайте <https://nodejs.org/en/download/> можно скачать установочные файлы или архивы, а также найти инструкции по установке.

В данных лабораторных работах будет использована операционная система CentOS, а Node.js будет установлен из официального репозитория <https://rpm.nodesource.com/>

1. Реквизиты для доступа к виртуальной машине:  
Учетная запись: **nodejs**  
Пароль: **nodejs**
2. Запустите терминал. Это можно сделать, кликнув правой кнопкой мыши по рабочему столу и выбрав пункт **Open Terminal**.
3. Подключите официальный репозиторий Node.js:

```
# sudo yum install -y gcc-c++ make
```

```
# curl -sL https://rpm.nodesource.com/setup_10.x | sudo bash -E -
```

4. Установите Node.js

а. Данная команда установит не только Node.js, но и NPM:

```
# sudo yum install nodejs
```

5. Проверьте, что Node.js и NPM корректно установлены и определите их версию:

```
# node -v
```

```
# npm -v
```

## Раздел 2. REPL

Интерфейс REPL (Read-Eval-Print Loop) – встроенный командный интерфейс Node.js, позволяющий тестировать или разрабатывать код.

1. Запустите Node.js командой `node`
2. Создайте переменные `foo` и `bar` присвойте ей значения 5 и 3.

Выведите на консоль:

- a. Сумму
- b. Разницу
- c. Произведение
- d. Частное и остаток от деления

```
node
> foo = 5
5
> bar = 3
3
> foo + bar
8
> foo - bar
2
> foo * bar
15
> foo / bar
1.6666666666666667
> foo % bar
2
```

3. Создайте массив `fruits` и перечислите три любых фрукта

- a. Добавьте в начало массива строку `'lime'`
- b. Добавьте в конец массива строку `'kiwi'`
- c. Выведите нулевой элемент массива
- d. Для каждого элемента массива

```
> fruits = ['apple', 'orange', 'plum']
[ 'apple', 'orange', 'plum' ]
> fruits.unshift('lime')
4
> fruits
[ 'lime', 'apple', 'orange', 'plum' ]
> fruits.push('kiwi')
5
> fruits
[ 'lime', 'apple', 'orange', 'plum', 'kiwi' ]
> fruits.length
```

## 4. Создайте конструктор объекта `person`

- a. Объект должен иметь свойства `name`, `lastName`, `age`
- b. Создайте два объекта `John Doe, 57` и `Jane Doe, 49`

```
> function Person (name, lastname, age) {  
... this.name = name;  
... this.lastName = lastname;  
... this.age = age;  
... return this;  
... }  
undefined  
> john = new Person ('John', 'Doe', 57)  
Person { name: 'John', lastName: 'Doe', age: 57 }  
> jane = new Person ('Jane', 'Doe', 49)  
> jane = new Person ('Jane', 'Doe', 49)  
Person { name: 'Jane', lastName: 'Doe', age: 49 }
```

- c. Выведите на экран значение существующего свойства (например, `age`)
- d. Выведите на экран значение несуществующего свойства (например, `birthDay`)

```
> john.age  
57  
> john.birthDay  
Undefined
```

e. Создайте объект `john_clone = john`

- f. Поменяйте возраст объекта `john_clone`. Проверьте возраст для обоих объектов. Изменился ли он и почему?

```
> john_clone = john  
Person { name: 'John', lastName: 'Doe', age: 57 }  
> john_clone.age = 99  
99  
> john.age  
99  
>
```

## 5. Опционально. Создайте функцию-счетчик

- a. Функция должна при первом вызове выдавать 0, а при последующих 1-2-3 и т.д.
- b. Функция не должна использовать глобальных переменных

```
> function counter_generator () {  
... c=0;
```

```
... return function () {return c++};  
... }  
undefined  
> counter = counter_generator();  
[Function]  
> counter ()  
0  
> counter ()  
1  
> counter()  
2
```

### 6. Сохраните код из текущей сессии в файл `test.js`

```
> .save test.js  
Session saved to: test.js
```

### 7. Выйдите из интерфейса REPL

```
> .exit
```

### 8. Запустите файл `test.js`

```
node test.js
```

### **Раздел 3. Установка среды разработки**

Бесплатная среда разработки Microsoft Visual Studio доступна для скачивания на сайте <https://code.visualstudio.com/> и в официальных репозиториях Microsoft. Подробные инструкции об установке на CentOS можно найти в <https://code.visualstudio.com/docs/setup/linux>

#### **1. Подключите репозиторий Microsoft Visual Studio Code.**

##### **а. Импортируйте ключи репозитория Microsoft:**

```
# sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

##### **б. Создайте репо файл для создания Visual Studio Code репозитория:**

```
# sudo nano /etc/yum.repos.d/vscode.repo
```

##### **с. Вставьте следующее содержимое в файл:**

```
[code]
name=Visual Studio Code
baseurl=https://packages.microsoft.com/yumrepos/vscode
enabled=1
gpgcheck=1
gpgkey=https://packages.microsoft.com/keys/microsoft.asc
```

##### **д. Сохраните файл сочетанием клавиш **Ctrl + X**, выберите сохранение изменений клавишей **Y** и нажмите **Enter**, чтобы подтвердить имя файла.**

##### **е. Проверьте, что текст сохранился в файл следующей командой:**

```
# cat /etc/yum.repos.d/vscode.repo
```

##### **ф. Также можно скопировать файл репозитория из каталога `/nodejs-labfiles/install`**

```
# sudo cp /nodejs-labfiles/install/vscode.repo /etc/yum.repos.d/
```

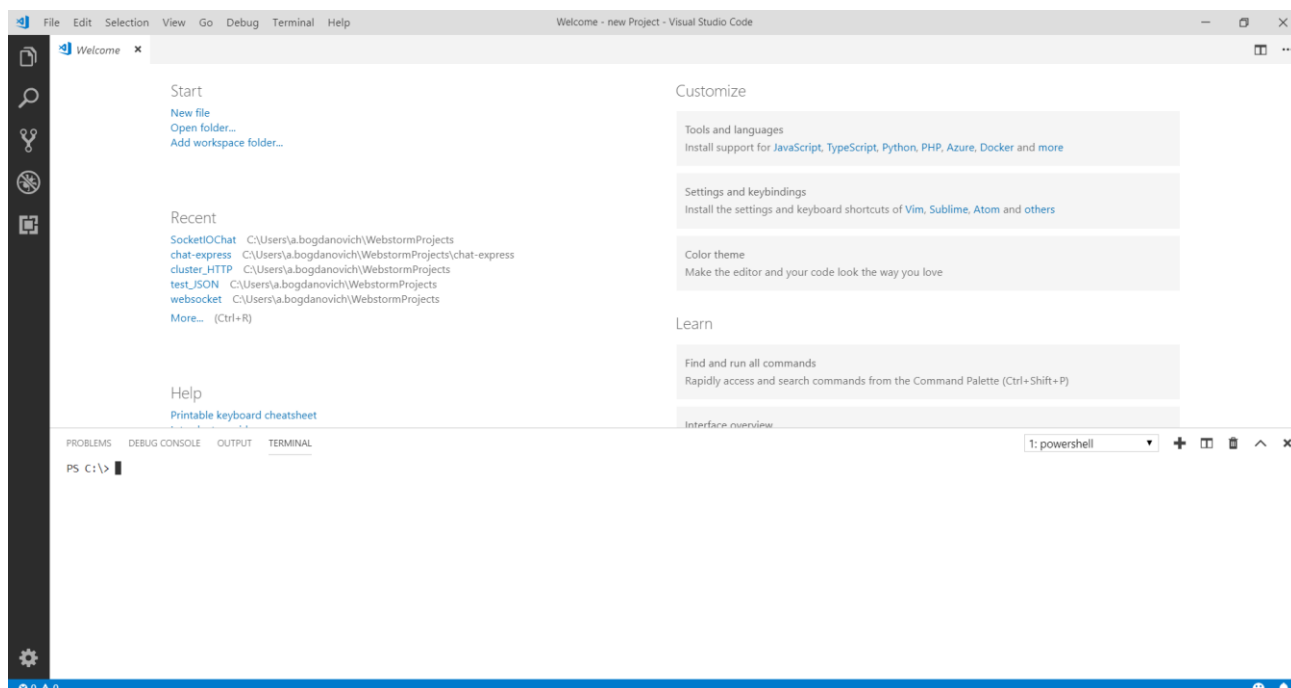
#### **2. Установите Visual Studio Code**

##### **а. В процессе установки система запросит подтверждение: ответьте **y**.**

```
# sudo yum install code
```



3. Установка завершена. Запустите Visual Studio Code командой `code`, чтобы проверить, что все установилось корректно. Должно открыться приветственное окно:





## Упражнение 2. Создание и подключение пользовательских модулей

---

### О чем это упражнение:

В этой лабораторной работе Вы изучите и разработаете проект на Node.js и познакомитесь с концепцией модульной архитектуры

### Что Вы должны будете сделать:

1. Создать проект user Project, выдающий приветствие пользователям на различных языках
2. На примере проекта будут рассмотрены задачи создания и подключения модуля, подключения группы модулей, а также продемонстрирован эффект кеширования модулей и принципы поиска модулей при подключении

## Раздел 1. Создание собственного модуля User

1. Создайте новый каталог для проекта `userProject`
2. В среде разработки VisualStudioCode выберите **File-Open Folder** и перейдите в каталог проекта
3. Создайте файл `server.js` (**File-New File**)
4. Добавьте конструктор объекта `User` со свойством `name` и методом `welcome()`

```
function User (name) {  
  this.name = name;  
  this.welcome = function () { return "hello " + this.name }  
}
```

5. Создайте два объекта для пользователей с использованием этого конструктора

```
var john = new User ("John");  
var jane = new User ("Jane");  
  
console.log (john.welcome());
```

6. Проверьте работу проекта
  - a. Откройте терминал **Terminal – New Terminal**
  - b. Запустите файл `server.js`

```
node server.js
```

7. Предположим, что со временем проект будет развиваться, и код конструктора для пользователя будет расширяться. Node.js предполагает организацию кода в виде модулей, поэтому вынесем код конструктора `User` в отдельный модуль
  - a. Создайте новый файл `user.js` и перенесите в него код

```
//user.js  
function User (name) {  
  this.name = name;  
  this.welcome = function () { return "hello " + this.name }  
}
```

- b. Подключите модуль `user` в основной файл

```
// server.js  
require ('./user');  
var john = new User ("John");  
var jane = new User ("Jane");
```

### с. Попробуйте запустить проект

```
node server.js
...\userProject\server.js:5
var john = new User ("John");
              ^
ReferenceError: User is not defined
    at Object.<anonymous>
```

- d. В отличие от JavaScript, где глобально определенная функция в модуле становилась глобальной во всем проекте, в NodeJS функции и переменные внутри модуля – локальные. Для передачи их в родительский модуль используется объект `module.exports`

```
//user.js
function User (name) {
    this.name = name;
    this.welcome = function () { return "hello " + this.name}
}
module.exports = User;

// server.js
var User = require ('./user');
```

### е. Проверьте, что проект работает

## Раздел 2. Подключение группы модулей. Поддержка языков

Добавьте в проект возможность вывода приветственных фраз на различных языках.

1. Создайте модуль `ru.json`

```
{  
  "Hello": "Привет"  
}
```

2. Подключите его в модуль `user.js`

```
//user.js  
var phrases = require('./ru.json')  
function User (name) {  
  this.name = name;  
  this.welcome = function () { return phrases.Hello + this.name }  
}  
module.exports = User;
```

3. Проверьте работу проекта
4. Поскольку проект может поддерживать большое количество языков, которые могут добавляться со временем, было бы неудобно подключать модуль для каждого языка отдельно
  - a. Создайте каталог `phrases`
  - b. Перенесите в него модуль `ru.json` и добавьте модуль `eng.json`
  - c. Создайте файл `index.js` для единой точки подключения

```
// phrases/index.js
```

```
module.exports.ru = require ('./ru.json');  
module.exports.eng = require ('./eng.json');
```

- d. Измените подключение модуля `ru.json` в файле `user.js` на подключение каталога `phrases`. Обратите внимание, что нет принципиальной разницы между подключением модуля и каталога, если файл с таким именем не найден, будет подключен файл `index.js` из каталога с соответствующим именем

```
var phrases = require ('./phrases');
```

е. Измените конструктор создания пользователя для учета языковых пожеланий

```
//user.js
var phrases = require ('./phrases');

function User (name, lang) {
    this.name = name;
    this.welcome = function () { return phrases[lang].Hello + this.name}
}
module.exports = User;

// server.js
var User = require ('./user');

var john = new User ("John", 'eng');
var vasya = new User ("Вася", 'ru' );

console.log (john.welcome());
console.log (vasya.welcome());
```

ф. Проверьте работу проекта

### **Раздел 3. Кэширование модуля (опционально)**

При большом количестве языков загрузка модуля `phrases` может занимать долгое время, т.к. подключение файлов блокирует выполнение JavaScript до окончания загрузки. Однако при загрузке модуль кэшируется и повторного чтения файлов и вычисления не происходит.

1. Подключите модуль `phrases` в основной файл `server.js`
  - a. Добавьте замер времени загрузки модуля. Для этого можно использовать функции `console.time('label')` и `console.timeEnd('label')`
  - b. Добавьте аналогичный код в `user.js`

```
// server.js
console.time('first load');
var phrases = require ('phrases');
console.timeEnd('first load');
var User = require ('./user');

// user.js
console.time('second load');
var phrases = require ('phrases');
console.timeEnd('second load');
```

#### 2. Запустите проект

```
userProject> node server.js

first load: 3.645ms
second load: 0.565ms
```



## Раздел 4. Поиск модуля (опционально)

Предположим, что проект развивается и что возникла необходимость перенести `user.js` в отдельный каталог.

1. Подключение каталога вместо подключения файла более гибко и позволяет в дальнейшем проще развивать проект. Измените подключение модуля `user.js` на подключение каталога `user`

- a. Создайте каталог `user`
- b. Переместите файл `user.js` в каталог и переименуйте его в `index.js`
- c. Подключение файла `user.js` не требуется изменять

```
// server.js
var phrases = require ('./phrases');
var User = require ('./user');
```

- d. Однако файл для подключения модуля `phrases` потребуется изменить путь

```
// user/index.js
var phrases = require ('../phrases');
```

При дальнейших изменениях файловой структуры необходимо обновлять пути для подключения модулей. Среда разработки частично помогает решить эту задачу с помощью автоматического обновления путей при перемещении файлов. Однако для часто подключаемых модулей лучше выбрать иной метод подключения, например, переместить его в каталог `node_modules`

2. Создайте каталог `node_modules` и переместите в него каталог `phrases`

- a. Измените способ подключения модуля

```
// user/index.js
var phrases = require ('phrases');
```

```
// server.js
var phrases = require ('phrases');
```

3. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB2-user`



## Упражнение 3. Использование модулей Node.js

---

### О чем это упражнение:

В этой лабораторной работе Вы познакомитесь с основными модулями Node.js, которые используются при работе с файлами.

### Что Вы должны будете сделать:

1. Ознакомиться с официальной документацией модулей Node.js
2. Создать модуль, читающий файл
3. На примере чтения файла будут рассмотрены особенности работы синхронных и асинхронных функций
4. Создать модуль, читающий файл с использованием потоков

## **Раздел 1 Работа с документацией**

Node.js и встроенные модули снабжены подробной документацией. Рекомендуется изучить ее перед началом работы

1. Перейдите на сайт <https://nodejs.org/> в раздел **DOCs** и откройте документацию, соответствующую установленной версии Node.js
  - a. Проверить версию Node.js можно командой `node -v`
2. Изучите список встроенных модулей (боковая панель слева)
3. Откройте документацию по модулю `console`  
<https://nodejs.org/dist/latest-v10.x/docs/api/console.html>
  - a. Изучите доступные методы, например
    - i. `console.log()` и `console.error()`
    - ii. `console.time()` и `console.timeEnd()`
4. Для каждого модуля определен параметр стабильность, показывающий допустимо ли использовать этот модуль в продуктивных решениях
  - a. Какова стабильность у модуля `console`?
  - b. Какова стабильность у модуля `async_hooks`?
  - c. Какова стабильность у модуля `domain`?

## Раздел 2. Модули работы с файловой системой

За работу с файловой системой отвечает модуль `fs`. Полная документация всех методов <https://nodejs.org/api/fs.html>

За формирование путей файловой системы отвечает модуль `path`. Полная документация всех методов <https://nodejs.org/api/path.html>

1. Создайте новый модуль в рамках текущего проекта
2. Прочитайте текущий файл и выведите содержимое в консоль
  - a. Синхронный вариант

```
var fs = require ('fs');
var data = readFileSync (__filename);
console.log (data);
```

- b. Асинхронный вариант

```
fs.readFile (__filename, function (err, data) {
  if (err) throw err;
  console.log (data);
})
```

3. Обратите внимание, что результат работы функций не строка, а объект буфер – массив бинарных данных

```
node fs.js
<Buffer 76 61 72 20 66 73 20 3d 20 72 65 71 75 69 72 65 20 28 27 66 73 27 29
3b 0d 0a 74 72 79 20 7b 0d 0a 20 20 20 20 76 61 72 20 64 61 74 61 20 3d 20 66
73 ... >
```

- a. Для перевода информации в строку можно использовать метод `Buffer.toString()`  
`console.log (data.toString());`

- b. Или указание кодировки файла при чтении

```
var data = fs.readFileSync (__filename, 'utf-8');
fs.readFile (__filename, 'utf-8', function (err,data){...})
```

4. При чтении произвольного файла необходимо передавать в функцию чтения путь к нему. Предположим, нам необходимо прочитать файлы из каталога `public`
  - a. Для формирования путей используйте модуль `path`

```
var fs = require ('fs');
var path = require('path');

var filename = 'good.file';

fs.readFile(path.join (__dirname, 'public', filename), function (err, data) {
  if (err) throw err;
  console.log (data.toString());
})
```

### 5. Проверьте работу проекта.

- a. Создайте файл `'good.file'` в текущем каталоге и запишите в него текст «I'm a good file»

```
nano good.file
    I'm a good file
```

- b. Или скопируйте файл из каталога `/nodejs-labfiles/LAB3-files`

```
cp -a /nodejs=labfiles/LAB3-files/good.file public/good.file
```

### 6. Обработка ошибок – важная задача разработчика сервера, т.к. ошибка может привести к остановке сервера или неверной логике работы

- a. В примерах выше при ошибке чтения файла мы выдавали исключение и сервер завершался. Это позволяет не пропустить ошибку в тестовой среде. Однако в продуктивной среде необходимо обрабатывать основные проблемы
- b. Подумайте, какие ошибки могут возникнуть при чтении файла? Проверьте следующие проблемы:

- i. Чтение несуществующего файла `'nonexistent.file'`

- c. Отсутствие доступа к файлу `'bad.file'`. Создайте файл и отключите доступ к нему командой `chmod` или скопируйте файл из каталога `/nodejs-labfiles/LAB3-files`

```
nano bad.file
    I'm a bad file
```

```
chmod 000 bad.file
```

или

```
cp -a /nodejs=labfiles/LAB3-files/bad.file public/bad.file
```

- d. Проверить доступ к файлу можно с помощью функций `fs.access()`, `fs.stat()` и других. Однако рекомендованный вариант – обработать ошибку при ее получении
- i. Полный перечень ошибок можно найти в документации по библиотеке LibUV  
<http://docs.libuv.org/en/v1.x/errors.html>

```
var fs = require ('fs');
var path = require('path');

var filename = 'bad.file';
var path_to_file = path.join (__dirname, 'public', filename);

fs.readFile(path_to_file, function (err, data) {
  if (err) {
    switch (err.code) {
      case 'ENOENT': {
        console.log ('No such file'); return}
      case 'EPERM': {
        console.log ('File is not readable'); return}
      default: console.log (err);
    }
  }
  else console.log (data.toString());
})
```

## Раздел 3. Синхронные и асинхронные функции (опционально)

Асинхронные функции позволяют более эффективно использовать ресурсы сервера и обрабатывать запросы иных клиентов во время ожидания результата длительной операции, например чтения файла.

1. Реализуйте синхронный и асинхронный вариант функции и замерьте следующие времена для чтения большого файла:
  - a. От начала чтения файла до завершения чтения файла
  - b. От начала чтения файла до выполнения следующей строки кода

```
console.time('sync'); console.time('sync next line');
fs.readFileSync ('public/big.file');
console.timeEnd('sync'); console.timeEnd('sync next line');

console.time('async'); console.time('async next line');
fs.readFile('public/big.file', function (err, data) {
  if (err) throw err;
  console.timeEnd('async');
})
console.timeEnd('async next line');
```

- c. Создайте большой файл, записав в него 100M нулей или скопируйте файл из каталога /nodejs-labfiles/LAB3-files

```
dd if=/dev/zero of=public/bigfile bs=1M count=100
или
cp -a /nodejs=labfiles/LAB3-files/bad.file public/bad.file
```

- d. Запустите проект и замерьте результат

```
node fs.js
sync: 32.684ms
sync next line: 36.622ms
async next line: 0.593ms
async: 14.525ms
```



## Раздел 4. Поток

Один из способов работы с файлами – использование потоков.

1. Прочитайте файл `good.file` с помощью потоков и выведите содержимое на экран

```
var fs = require ('fs');
var path = require('path');

var filename = 'bad.file';
var path_to_file = path.join (__dirname, 'public', filename);

var fileStream = fs.createReadStream(path_to_file);
fileStream.on('readable', function (){
    var buf = fileStream.read();
    console.log (buf);
} )
```

2. Запустите скрипт и проверьте работу кода
  - a. Обратите внимание, что последний считанный блок данных – всегда `null`
  - b. Иногда требуется устанавливать дополнительную обработку на то, чтобы не обрабатывать этот блок

```
node .\fs.js
<Buffer 49 27 6d 20 67 6f 6f 64>
Null
```

3. Проверьте работу потоков при чтении для большого файла `big.file`

- a. Для наглядности выводите не данные, а их размер

```
var fileStream = fs.createReadStream(path_to_file);
fileStream.on('readable', function (){
    var buf = fileStream.read();
    if (buf) console.log (buf.length);
} )
```

- b. Как можно видеть, одновременно в памяти сервера находится только 64Кб, а не весь файл как при обычном чтении файла

```
node .\fs.js
65536
65536
...
65536
15504
file is closed
```

4. У файловых потоков есть собственные события `open` и `close`, которые так же можно обрабатывать

- а. Установите обработчики на эти события

```
fileStream.on('open', function () {
  console.log ('file is opened');
})
fileStream.on('close', function () {
  console.log ('file is closed');
})
```

5. Так как потоки являются наследниками класса `EventEmitter`, у них есть событие `error`

- а. Если событие `error` не обработано явным образом, то при наступлении этого события поток выдает ошибку  
б. Проверьте работу кода для файла `bad.file`

```
node.\fs.js
events.js:174
    throw er; // Unhandled 'error' event
    ^
Error: EPERM: operation not permitted, open '\public\bad.file'
Emitted 'error' event at:
    at lazyFs.open (internal/fs/streams.js:115:12)
    at FSReqWrap.oncomplete (fs.js:141:20)
```

6. Установите обработку для файла события `error`

```
fileStream.on('error', function (err){
  console.log (err);
})

node .\fs.js
{ [Error: EPERM: operation not permitted, open ' \public\bad.file']
  errno: -4048,
  code: 'EPERM',
  syscall: 'open',
  path: '\public\bad.file' }
```

7. С готовыми решениями можно ознакомиться в `/nodejs-solutions/LAB3-fs`

## Упражнение 4. Пакетный менеджер NPM

---

### О чем это упражнение:

В этой лабораторной работе Вы познакомитесь с пакетным менеджером NPM

### Что Вы должны будете сделать:

1. Инициализировать проект `userProject`
2. Ознакомиться с функционалом NPM
3. Расширить функционал проекта `userProject` с помощью подключаемого модуля `moment`
4. Создать модуль журналирования с возможностью вывода отформатированных сообщений, окрашенных в разные цвета

## Раздел 1. Инициализация текущего проекта

1. Инициализируйте текущий проект с помощью команды `npm init`

```
npm init
```

```
Press ^C at any time to quit.  
package name: (userproject)  
version: (1.0.0)  
description: my test project  
entry point: (fs.js) server.js  
test command:  
git repository:  
keywords:  
author:  
license: (ISC)
```

2. Изучите созданный файл `package.json`

```
{  
  "name": "userproject",  
  "version": "1.0.0",  
  "description": "my test project",  
  "main": "server.js",  
  "dependencies": {},  
  "devDependencies": {},  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "start": "node server.js"  
  },  
  "author": "",  
  "license": "ISC"  
}
```

3. Запустите проект командой `npm start`

```
npm start
```

## Раздел 2. Установка, обновление и удаление пакетов NPM

### 1. Установите модуль `underscore`

```
npm install underscore
```

### 2. Проверьте изменения в проекте

#### а. Проверьте, как изменился файл `package.json`

```
{
  "name": "userproject",
  "version": "1.0.0",
  "description": "my test project",
  "main": "server.js",
  "dependencies": {
    "underscore": "^1.9.1"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

#### б. В каталоге `node_modules` появился каталог `underscore`.

Изучите файлы в нем

#### с. Посмотрите подробное описание пакета

```
npm view underscore
```

```
underscore@1.9.1 | MIT | deps: none | versions: 35
JavaScript's functional programming helper library.
http://underscorejs.org
```

```
keywords: util, functional, server, client, browser
```

```
dist
```

```
.tarball: https://registry.npmjs.org/underscore/-/underscore-1.9.1.tgz
```

```
.shasum: 06dce34a0e68a7bab29b365b8e74b8925203961
```

```
.integrity: sha512-
```

```
5/4etnCkd9c8gwgowi5/om/mY05ajCa0gdzj/ow+0eQV9WxKBDZw5+ycmKmeaTXjInS/W0B
zpGLo2xR2aBwZdg==
```

```
.unpackedSize: 111.0 kB
```

```
maintainers:
```

```
- jashkenas <jashkenas@gmail.com>
```

```
- jridgewell <justin+npm@ridgewell.name>
```

```
dist-tags:
  latest: 1.9.1  stable: 1.9.0
```

```
published 10 months ago by jashkenas jashkenas@gmail.com
```

### 3. Подключите underscore в проект и проверьте его работу

- а. Например, выведите максимальный элемент массива  
`[1, 2, 3]`

```
// server.js
_ = require ('underscore');
console.log (_.max([1,2,3]));
```

- б. Подробную информацию о возможностях пакета можно  
получить на сайте <https://underscorejs.org/>

### 4. Удалите пакет underscore

```
npm uninstall underscore
```

### 5. Найдите все пакеты по ключевому слову, например, date

```
npm search date
```

### 6. Установите пакет moment, но укажите предыдущую версию, например, 2.20.0

```
npm install moment@2.20.0
```

### 7. Выведите все пакеты, версия которых устарела

```
npm outdated
Package Current Wanted Latest Location
moment 2.20.0 2.24.0 2.24.0 userproject
```

### 8. Обновите все устаревшие пакеты

```
npm update
```

### 9. Выведите список подключенных модулей текущего проекта

```
npm ls
```

## Раздел 3. Работа с датами. Модуль *moment*

Пусть при создании пользователя передается информация о дате его рождения. Добавьте функцию, высчитывающую его возраст.

1. Убедитесь, что модуль `moment` установлен, и подключите его в модуль `user.js`
  - a. Добавьте в конструктор объекта свойство `birthDate`
  - b. Проверьте, что переданная пользователем дата существует

```
var moment = require ('moment'); // npm install moment
function User (name, date, lang) {
  this.name = name;
  this.birthDate = new moment(date, 'DD-MM-YYYY');
  if (!this.birthDate.isValid()) {
    console.log('Date is invalid for user ' + this.name);
  }

  this.welcome = function () { return phrases[lang].Hello + ", " + this.name }
}
module.exports = User;
```

2. Добавьте функцию, высчитывающую возраст пользователя

```
function User (name, date, lang) {
  this.name = name;
  if (!(lang in phrases)) {
    this.birthDate = new moment(date, 'DD-MM-YYYY');
    if (!this.birthDate.isValid()) {
      console.log('Date is invalid for user ', this.name);
    }
  }
  this.welcome = function () { return phrases[lang].Hello + ", " + this.name }
  this.age = new moment().diff(this.birthDate, 'years')
}
```

3. Измените в модуле `server.js` вызов конструктора пользователей и проверьте работу проекта

```
var john = new User ("John", '01-22-1991', 'eng');
var vasya = new User ("Вася", '09-09-1954', 'ru');

console.log (john.birthDate.format('DD.MM'));
console.log (vasya.age);
```

## Раздел 4. Журналирование. Модули *util* и *colors*

До сих пор для вывода сообщений был использован метод `console.log()`. Это один из самых простых, но функциональных методов. В качестве недостатков `console.log()` можно назвать:

- Отсутствие указания на то, из какого модуля получено сообщение
- Отсутствие временной отметки
- Отсутствие возможности классифицировать сообщения по уровням info-warning-error
- Нет цветовой подсветки наиболее важных сообщений

Разработайте собственный модуль `logger`, добавляющий этот функционал.

### 1. Создайте модуль `logger.js`

- Пусть модуль журналирования при инициализации принимает имя модуля-источника сообщений
- Пусть модуль журналирования имеет три метода
  - `info (message)` – выдача сообщения зеленым цветом
  - `warn (message)` – выдача сообщения желтым цветом
  - `err (message)` – выдача сообщения красным цветом
- Для формирования текста используйте метод `util.log()`
- Для указания цвета сообщения используйте подключаемый модуль `colors`
  - Необходимо предварительно его установить

```
var colors = require('colors');    // npm install colors
var util = require('util');

module.exports = logger;
function logger (source) {
  this.info = function (message) {
    util.log ("%s: ", source, message.green)
  }
  this.warn = function (message) {
    util.log ("%s: ", source, message.yellow)
  }
  this.err = function (message){
    util.log ("%s: ",source, message.red)
  }
  return this;
}
```



2. Подключите модуль `logger.js` в модули `server.js` и `user.js`
  - а. Добавьте вывод отладочных сообщений

```
// server.js
var Logger = require('./logger');
var logger = new Logger ('server.js');
...
logger.info (vasya.welcome());
// user/index.js
var Logger =require('./logger');
var logger = new Logger ('user.js');

function User (name, date, lang) {
    ...
    logger.info ('user ' + this.name + ' is created' );
}
module.exports = User;
```

3. Опционально. Добавьте в модуле `user.js` проверку на поддержку запрошенного пользователем языка. Если такого языка нет, выдайте предупреждение и переключитесь на английский

```
// user/index.js
function User (name, date, lang) {
    this.name = name;
    if (!(lang in phrases)) {
        logger.warn ('No support for ' + lang + '. Switching to eng');
        lang = 'eng';
    }
    this.welcome = function () {
        return phrases[lang].Hello + ", " + this.name
    }
    logger.info ('user ' + this.name + ' is created' );
}
module.exports = User;

// server.js
var john = new User ("John" , 'eng');
var vasya = new User ("Вася", 'ru' );
var roberto = new User ("Roberto", 'es');
```

4. Проверьте работу кода
5. Опционально. Дополнительные возможности модуля `colors` можно посмотреть в примерах `node_modules/colors/examples`

`node .\node_modules\colors\examples\normal-usage.js`

4. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB4-user`

## Упражнение 5. Статический веб-сервер

---

### О чем это упражнение:

В этой лабораторной работе Вы создадите статический веб-сервер на основе протоколов HTTP и HTTPS.

### Что Вы должны будете сделать:

1. Создать проект `httpServer`, предоставляющий клиенту HTML-страницу с поддержкой запроса дополнительных ресурсов (изображений, скриптов и пр.)
2. Создать ключи шифрования и мигрировать проект на протокол HTTPS

## Раздел 1. Веб-сервер HTTP

### 5. Создайте новый проект `httpServer`

- a. Инициализируйте его командой `npm init`
- b. Создайте файл `server.js`

### 6. Установите и подключите в проект модуль `http`

```
var http = require('http');
```

### 7. Запустите веб-сервер на порте 3000

```
var server = http.createServer(function (req, res) {  
    console.log('new request');  
}).listen(3000);  
console.log('server running on port 3000');
```

### 8. Запустите проект и попытайтесь подключиться через браузер по адресу `http://127.0.0.1:3000`

- c. Также тестирование различных параметров подключения возможно с помощью утилиты `curl`. Документация доступна по адресу <https://curl.haxx.se/docs/httpscripting.html>

```
curl http://127.0.0.1:3000
```

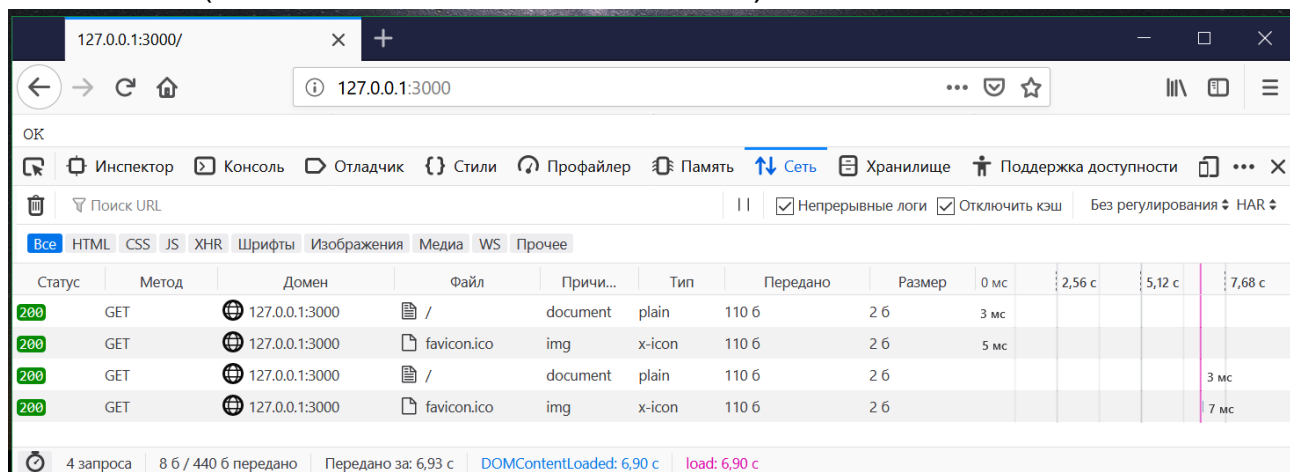
### 9. При каждом подключении

- d. Выводите на консоль заголовки запроса, метод HTTP и URL запрошенного ресурса,
- e. Возвращайте клиенту код успешного обращения 200 и сообщение OK

```
var server = http.createServer(function (req, res) {  
    console.log('new request');  
    console.log (req.headers);  
    console.log(req.method);  
    console.log (req.url);  
  
    res.writeHead(200);  
    res.end ('OK');  
  
}).listen(3000);  
console.log('server running on port 3000');
```

- f. Остановите и перезапустите сервер, чтобы был исполнен измененный код

- g. Проверьте работу клиента в браузере. Для более подробной информации воспользуйтесь инструментами разработчика (CTRL+SHIFT+I в Mozilla Firefox)



- h. Обратите внимание, что браузер инициирует два запроса при каждом обращении – GET / и GET /favicon.ico
10. Создайте примитивный статический веб-сервер.
- i. Если пользователь обращается к главной странице (GET /) – выдать страницу public/index.html
- j. Если пользователь обращается к иному адресу или использует другой метод – выдать код 404

```
var http = require('http');
var fs = require('fs');

function send404(response) {
  response.writeHead(404, { 'Content-Type': 'text/plain' });
  response.write('Error 404: Resource not found.');
```

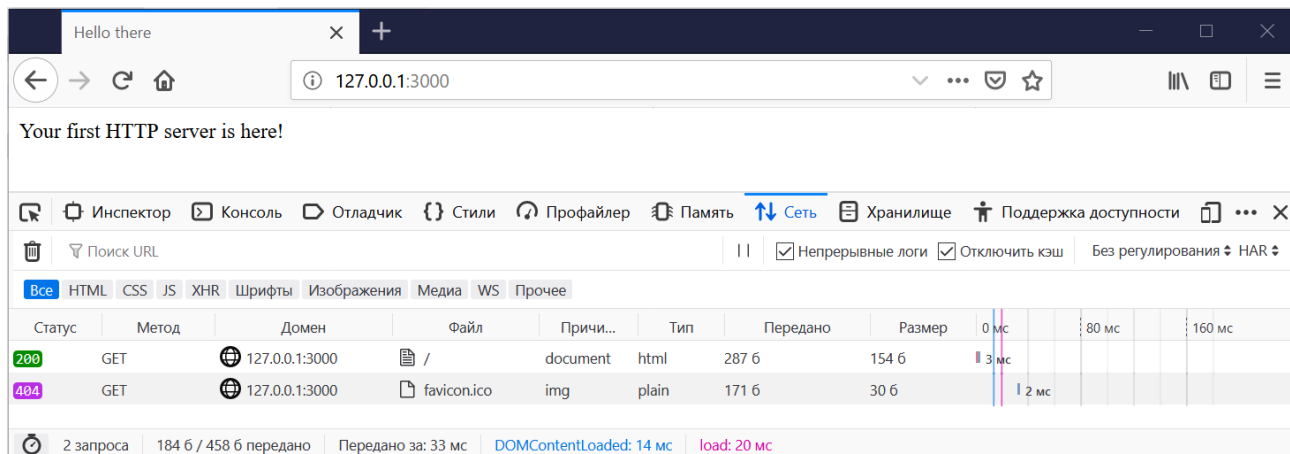
```
  response.end();
}

var server = http.createServer(function (req, res) {
  if (req.method == 'GET' ) {
    if (req.url=='/') {
      fs.readFile ('./public/index.html', function (err, data){
        if (err) throw err;
        res.writeHead(200, {'content-type': 'text/html'});
        res.end (data);
      })
    } else send404(res);
  } else send404(res);
}).listen(3000);
console.log('server running on port 3000');
```

- k. Пример файла public/index.html

```
<!--index.html-->
<html>
  <head>
    <title>Hello there</title>
  </head>
  <body>
    Your first HTTP server is here!
  </body>
</html>
```

## I. Проверьте работу кода



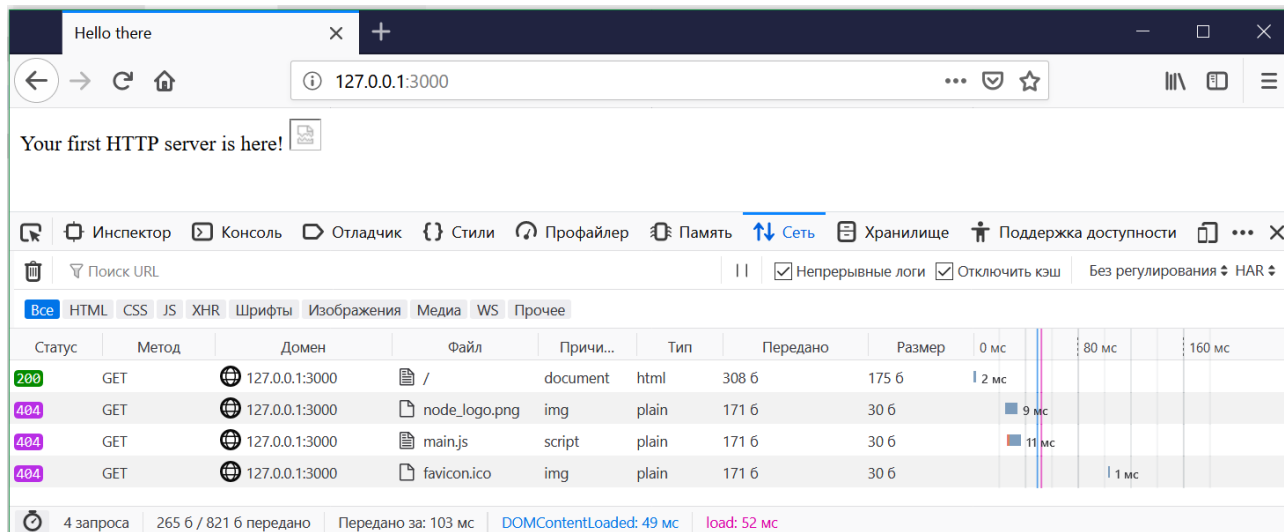
11. Приведенный код содержит многочисленные недостатки, которые проявятся при развитии проекта. Например, модифицируйте `index.html` таким образом, чтобы он подключал внешний JavaScript и изображение

```
<!--index.html-->
<html>
  <head>
    <title>Hello there</title>
  </head>
  <body>
    Your first HTTP server is here!
    <img src='./node_logo.png'>
    <script src='./main.js'></script>
  </body>
</html>
```

m. Изображение и скрипт можно скопировать из `/nodejs-files/http-project`

```
cp /nodejs-labfiles/http-project/* public
```

- п. Проверьте работу кода. Что происходит при загрузке изображения или внешнего скрипта?



- о. Измените код сервера так, чтобы загрузка элементов страницы работала корректно
  - i. Для автоматического определения типа передаваемой информации используйте модуль `mime-types`.

```
npm install mime-types
```

- ii. Перед выдачей файла убедитесь, что файл существует – иначе выдайте ошибку 404
- iii. При прочих ошибках – выдайте код 500 и текст ошибки клиенту

```
var http = require('http');
var fs = require('fs');
var mime = require('mime-types');
var path = require('path');

function send500(response) {
  response.writeHead(500, { 'Content-Type': 'text/plain' });
  response.write('Error 500: Internal server error');
  response.end();
}

var server = http.createServer(function (req, res) {
  console.log (req.method, req.url);
  if (req.method == 'GET' ) {
    if (req.url == '/') {
      filepath = './public/index.html';
```

```

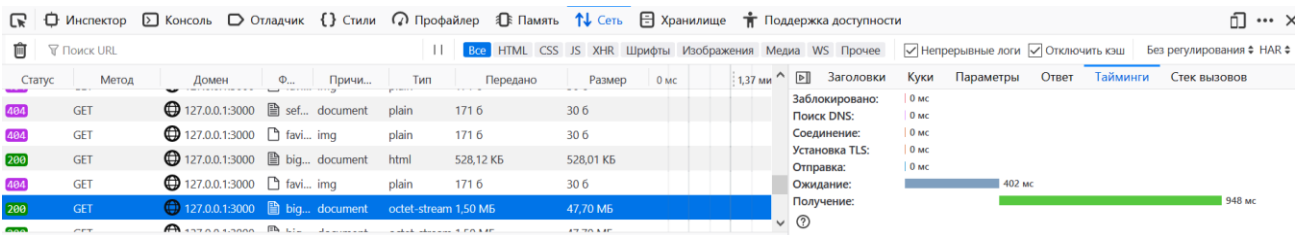
    } else filepath = path.resolve('./public/' + req.url);

    fs.readFile (filepath, function (err, data){
      if (err) {
        if (err.code == 'ENOENT') send404(res)
        else send500 (res)
      } else {
        res.writeHead (200, mime.lookup(filepath));
        res.end (data);
      }
    })
  } else send404(res);
}).listen(3000);

```

12. Еще одна проблема разработанного сервера – низкая производительность при загрузке больших файлов. В текущей реализации сервер сначала считывает данные, и только после окончания загрузки передает их клиенту.

- р. На время считывания данные хранятся в памяти, расходуя ресурсы сервера.
- q. Протестируйте работу сервера при обращении к файлу `big.file` (файл можно скопировать из `/bodejs-labfiles/project`)



- г. Для решения этой проблемы модули `fs` и `http` поддерживают потоки. Замените функцию `fs.readFile ()` на работу с `fs.ReadStream()`.
  - і. Не забудьте поставить обработку на событие `error` у потока

```

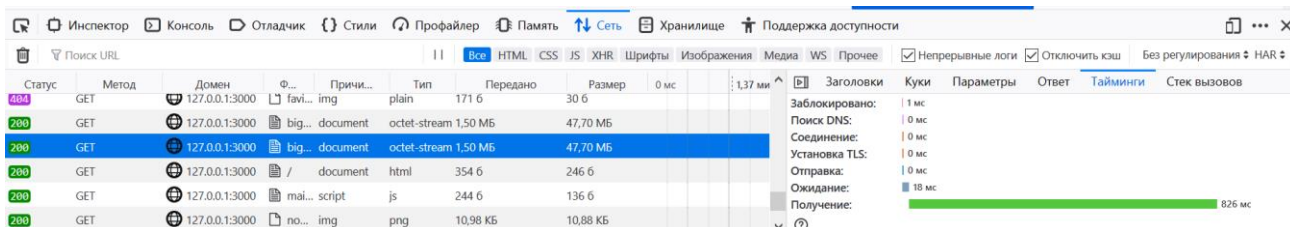
var server = http.createServer(function (req, res) {
  if (req.method == 'GET' ) {
    if (req.url=='/') {
      filepath = './public/index.html';
    } else filepath = path.resolve('./public/' + req.url);
    var fileStream = fs.ReadStream(filepath);
    fileStream.on ('error', function (err) {
      if (err.code == 'ENOENT') send404(res)
      else send500 (res);
    })
  }
});

```



```
res.writeHead(200, mime.lookup(filepath));
    stream.pipe(res);
  } else send404(res);
}).listen(3000);
```

### s. Проверьте время загрузки файла



13. Помимо рассмотренных проблем необходимо помнить об обработке различных нестандартных ситуаций, например:
  - t. Кириллические символы в названии файла
    - i. Создайте в каталоге `public` файл с именем `мой.файл` и попробуйте загрузить его из браузера
    - ii. Кириллические символы в URL кодируются перед отправкой в строку  
`%D0%BC%D0%BE%D0%B9.%D1%84%D0%B0%D0%B9%D0%BB`
    - iii. Для декодирования имени файла на сервере можно использовать функцию `decodeURI(filepath)`
  - u. Проверка на то, что предоставляемый файл находится в каталоге `/public`
  - v. И многое другое
14. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB5-http`

## Раздел 2. Веб-сервер HTTPS

Передача персональной информации пользователей через незашифрованное соединение недопустима. Для обслуживания шифрованного соединения используется протокол HTTPS, который использует протокол SSL для шифрования данных.

1. Сгенерируйте ключ SSL

```
openssl genrsa 1024 > key.pem
```

2. Сгенерируйте сертификат

а. Ответьте на вопросы

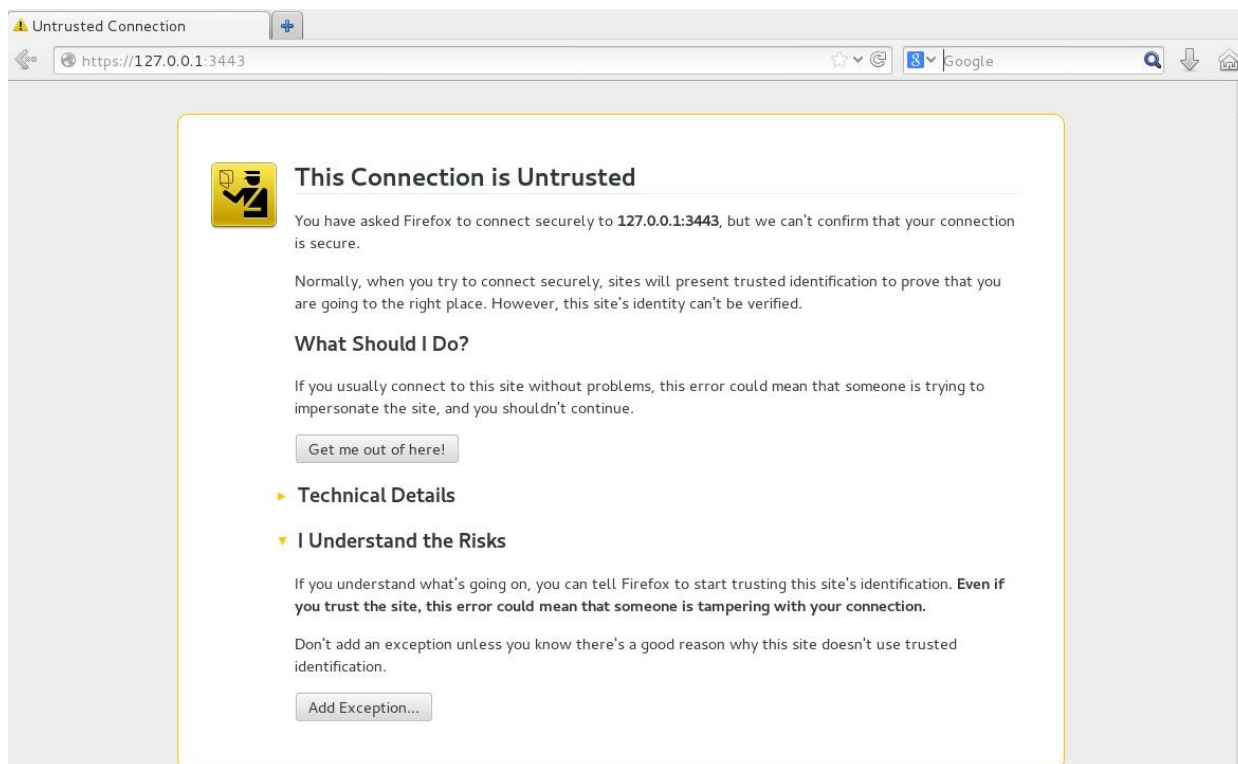
```
openssl req -x509 -new -key key.pem > cert.pem
```

3. Подключите в проект модуль HTTPS и замените функцию `http.createServer()` на `https.createServer()`. Остальной код можно оставить без изменений

```
var https = require('https');  
var server = https.createServer(options, function (req, res) {  
  ...  
}).listen(3443);
```

4. Проверьте работу проекта

- а. Поскольку ключ и сертификат выданы владельцем сервера, они не являются доверенными и при подключении пользователи получат предупреждение.
- б. Выберите „I Understand the risk” и добавьте сертификат в список исключений



5. Традиционный порт, используемый HTTPS – 443, HTTP – 80. Для запуска веб-сервера на этих портах требуются административные права
  - a. Измените код так, чтобы сервер работал на порте 443
  - b. Запустите сервер командой  
`sudo npm start`
  - c. Подключитесь к серверу по адресу <https://127.0.0.1> (порт указывать не требуется)
6. Предположим, необходимо обеспечить перенос сайта с HTTP на HTTPS так, чтобы это было прозрачно для пользователей
  - a. Добавьте в проект HTTP сервер, который будет перенаправлять любой запрос на HTTPS
  - b. Традиционный код ответа о перенаправлении запроса – 301

```
http.createServer(function (req, res) {  
  res.writeHead(301, { "Location": "https://" + req.headers['host'] + req.url });  
  res.end();  
}).listen(80);
```

7. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB5-https`



## Упражнение 6. Методы взаимодействия с клиентом

---

### О чем это упражнение:

В этой лабораторной работе Вы изучите различные технологии взаимодействия сервера с клиентом и реализуете макет чата и сервера мониторинга.

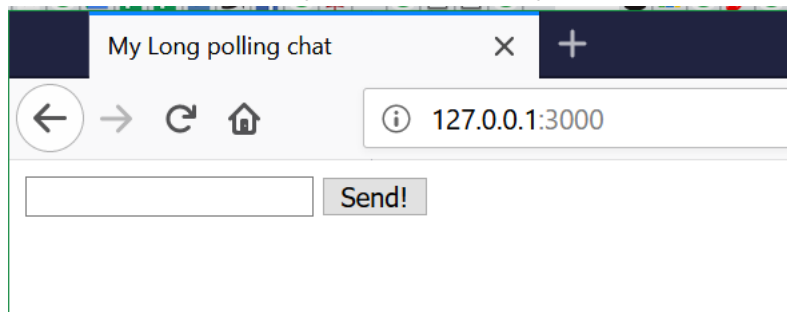
### Что Вы должны будете сделать:

1. Создать чат на основе длинных запросов
2. Создать сервер мониторинга на основе `WebSocket`
3. Создать чат с использованием библиотеки `Socket.IO`

## Раздел 1. Чат на основе длинных запросов

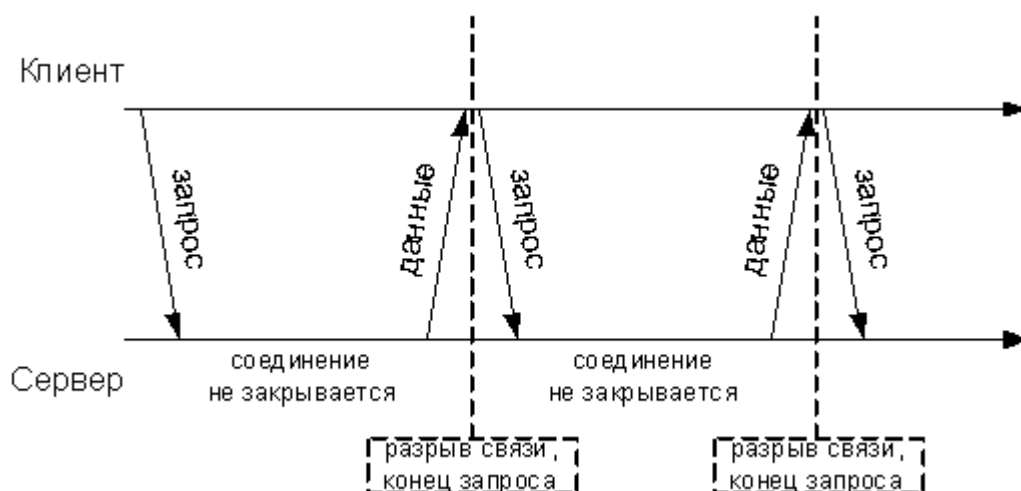
Создайте примитивный чат, работающий на основе длинных запросов.

- При подключении по адресу <http://127.0.0.1:3000/> сервер должен выдавать клиенту html-страницу



- После загрузки страницы клиент обращается к серверу и «подписывается» на получение сообщений
- При нажатии на кнопку "Send!" клиент передает серверу сообщение, а сервер передает это сообщение всем клиентам
- Клиент, получив сообщение от сервера, публикует его и открывает запрос к серверу на очередные обновления.

Общая схема длинных запросов приведена на рисунке ниже



1. Создайте новый проект `Chat_LongPolling`
2. Скопируйте заготовку кода сервера и клиента из каталога `/nodejs-labfiles/LAB6-chat`
3. Изучите HTML-код клиента, указанный в заготовке `index.html`.  
Описаны два объекта HTML: форма `publish` с текстовым полем `message` и кнопкой отправки и список `messages`, в котором будут публиковаться сообщения, полученные от сервера

```
<form id="publish">
  <input type="text" name="message">
  <input type="submit" value="Send!" class="btn">
</form>
<ul id="messages"> </ul>
```

4. Изучите JavaScript-код из `index.html`. Этот код типичен для любых клиентов, работающих по схеме длинных запросов.

а. При нажатии на кнопку «Send!» происходит отправка сообщения серверу

```
publish.onSubmit = function () {
  var xhr = new XMLHttpRequest ();
  xhr.open("POST", "/publish", true);
  xhr.send (JSON.stringify({message:this.elements.message.value}));
  this.elements.message.value="";
  return false;
};
```

б. При загрузке страницы запускается запрос на обновления от сервера.

с. Как только сообщение получено, оно публикуется в списке. После чего запускается новый запрос

д. При ошибке запрос на подписку повторяется через 500 мс

```
subscribe();
```

```
function subscribe (){
  var xhr = new XMLHttpRequest ();
  xhr.open("GET", "/subscribe", true);
  xhr.onload = function (){
    var li=document.createElement ('li');
    li.textContent = this.responseText;
    messages.appendChild (li);
    subscribe();
  };
  xhr.onerror = xhr.onabort = function () {
    setTimeout (subscribe, 500);
  };

  xhr.send('');
};
```

5. Изучите заготовку кода сервера `server.js`

а. При обращении к главной странице – передать клиенту `index.html`

- b. При обращении к <http://127.0.0.1:3000/subscribe> – зарегистрировать клиента
- c. При обращении к <http://127.0.0.1:3000/publish> – получить сообщение и передать его зарегистрированным клиентам
- d. При неверном подключении – выдать ошибку 404

```
var http = require ('http');
var fs = require ('fs');

http.createServer (function(req,res) {
  switch (req.url){
    case '/':
      //
      break;

    case '/subscribe':
      //
      break;

    case '/publish':
      //
      break;

    default:
      res.statusCode = 404;
      res.end ("Not found");
  }
}).listen (3000);
```

### 6. Разработайте код для передачи главной страницы

```
case '/':
  sendFile ("index.html", res);
  break;
...

function sendFile (fileName, res) {
  var fileStream = fs.createReadStream(fileName);
  fileStream.on ('error', function () {
    res.statusCode = 500;
    res.end("Server error");
  })
  fileStream.pipe (res);
}
```

### 7. Разработайте логику подписки и публикации



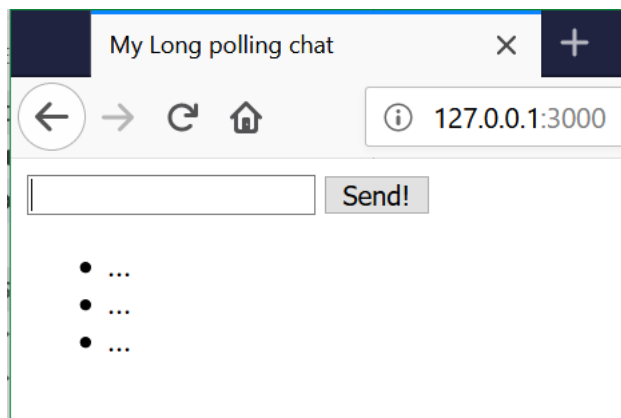
- a. Создайте отдельный модуль `chat.js`
- b. Функция подписки `subscribe` должна сохранять ссылку на поток ответа клиента
- c. Функция публикации `publish` должна передавать во все сохраненные потоки ответа

```
var clients = [];  
  
exports.subscribe = function (req,res) {  
  console.log ("new client");  
  clients.push (res);  
}  
  
exports.publish =function (message) {  
  console.log ("publish %s", message, clients.length);  
  clients.forEach (function (res){  
    res.end (message);  
  })  
  clients = [];  
}
```

### 8. Подключите модуль `chat` в основной модуль сервера

```
var chat = require ('./chat');  
...  
  
case '/subscribe':  
  chat.subscribe (req,res);  
  break;  
  
case '/publish':  
  chat.publish ("...");  
  res.end ("ok");  
  break;
```

9. Проверьте работу веб-сервера. В текущей версии сервер вместо сообщения передает заглушку " . . . "



10. Реализуйте логику получения сообщения из сообщения `publish`.

- a. `Request` как поток с возможностью чтения реализует состояния `'readable'` (в потоке есть данные, которые можно прочитать через `request.read()`) и `'end'` (данных в потоке больше нет)
- b. Не забудьте удалить `null` в конце считанного потока данных

```
case '/publish':
  var data = '';
  var body = '';
  req.on ('readable', function () {
    data = req.read ();
    if (data != null ) body += data;
  }).on ('end', function () {
    body = JSON.parse (body);
    chat.publish (body.message);
    res.end ("ok");
  });
  break;
```

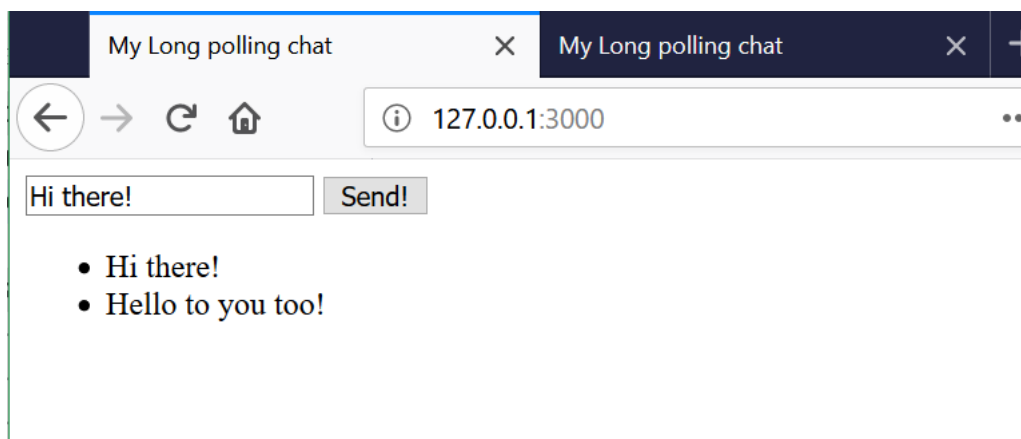
11. Код, приведенный выше, имеет как минимум две проблемы

- a. При получении некорректного JSON, функция `JSON.parse` выдаст исключение, которое приведет к отказу сервера
- b. Функция получения данных не проверяет их размер, поэтому возможно переполнение памяти сервера
- c. Добавьте проверку для этих ситуаций

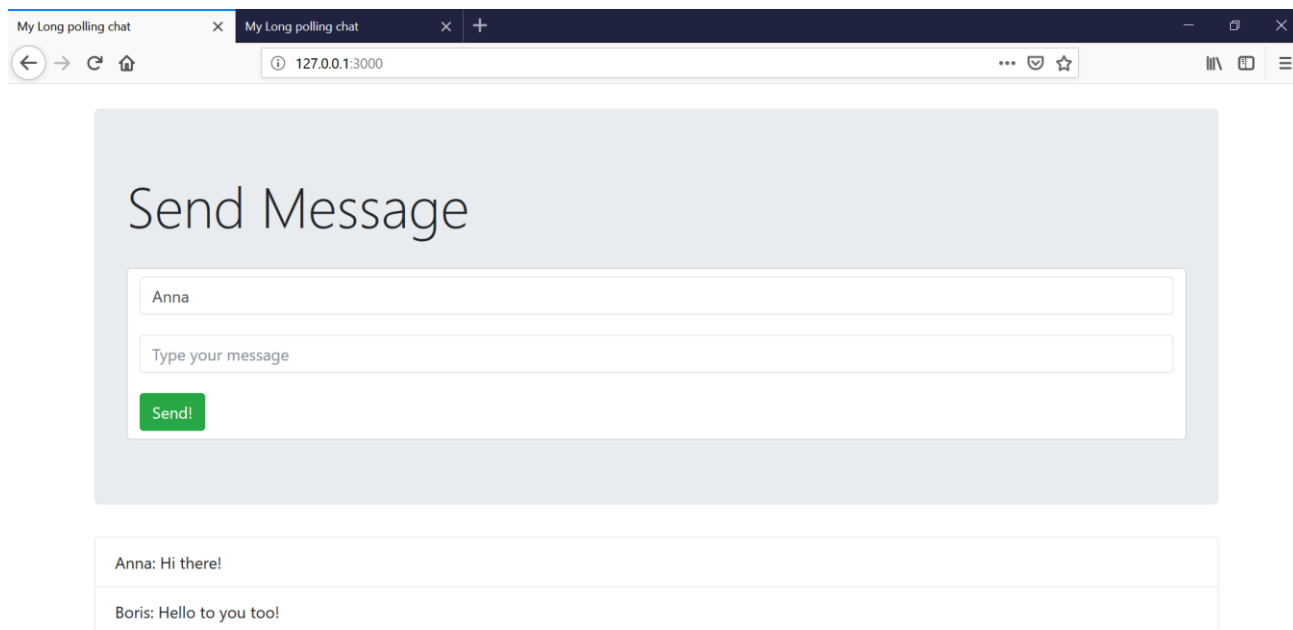
```
case '/publish':
  var data = '';
  var body = '';
  req.on ('readable', function () {
```

```
data = req.read ();
if (body.length > 100) {
  res.statusCode = 413;
  res.end ("Your message is too big!");
  return;
}
if (data != null ) body += data;
}).on ('end', function () {
  try {
    body = JSON.parse (body);
  } catch (e) {
    res.statusCode = 400;
    res.end ("Bad Request");
    return;
  }
  chat.publish (body.message);
  res.end ("ok");
});
break;
```

### 12. Проверьте работу чата



13. Опционально. Сделайте чат более удобным в использовании
- Добавьте поле с именем клиента. Модифицируйте функции отправки, приема и публикации сообщения для отображения автора сообщения
  - Добавьте стили CSS, например, используя библиотеку Bootstrap 4 и классы `form-control`, `list-group` и пр.



14. С готовым решением можно ознакомиться в [/nodejs-solutions/LAB6-chat](#)

## Раздел 2. Сервер мониторинга на основе WebSocket

На основе проекта ранее разработанного кода реализуйте веб-сервер, который раз в 100ms передает клиенту информацию об использовании памяти.

### 1. Создайте новый проект WebSocketMonitor

- Создайте модуль `server.js`, скопируйте код из проекта HTTPServer
- В конце модуля `server.js` экспортируйте объект `server`

```
// server.js
var server = http.createServer(function (req, res) {
  ...
});
server.listen(3000);

module.exports = server;
```

### 2. Создайте новый модуль ws-server.js

- Подключите модуль `server.js` в модуль `ws-server.js`

```
// ws-server.js
var httpserver = require('./server');
```

### 3. Установите и подключите библиотеку ws, реализующую WebSocket

```
// ws-server.js

var ws = require('ws');    // npm install ws
var websocketServer = new ws.Server ({server:httpserver});
```

### 4. Добавьте обработку нового подключения

- Добавьте проверку на закрытие соединения

```
websocketServer.on('connection', function (ws) {
  console.log ('client connected');

  ws.on('close', function () {
    console.log ('client disconnected');
  })
});
```

### 5. Добавьте периодическую статистику использования памяти клиенту

- Текущее использование памяти `process.memoryUsage()`

- b. С помощью `WebSocket` передавать можно только строки или бинарные данные, поэтому перед передачей преобразуйте объект в строку

```
websocketServer.on('connection', function (ws) {  
    var timer = setInterval (function () {  
        ws.send(JSON.stringify(process.memoryUsage()))  
    }, 100);  
  
    console.log ('client connected');  
  
    ws.on('close', function () {  
        console.log ('client disconnected');  
        clearInterval (timer);  
    })  
});
```

- 6. Со стороны клиента необходимо открыть соединение по протоколу `WebSocket` на порт соответствующий порт сервера

```
// client.js  
// создать подключение  
var socket = new WebSocket("ws://localhost:3000");
```

- 7. Также со стороны клиента необходимо обрабатывать

- a. События открытия и закрытия соединения
- b. События ошибок

```
socket.onopen = function() {  
    console.log("Connection opened");  
}  
socket.onclose = function (event) {  
    console.log ('Connection was closed', event.code, event.message)  
}  
socket.onerror = function (err) {  
    console.error (error.message);  
}
```

- c. Получение нового сообщения

```
socket.onmessage = function(event) {  
    showData(JSON.parse(event.data))  
};  
function showData(data) {  
    document.getElementById('rss').innerHTML = "RSS size is " + data.rss;  
    document.getElementById('heapTotal').innerHTML = "Heap total size is " +  
        data.heapTotal;  
    document.getElementById('heapUsed').innerHTML = "Heap used size is " +
```

```
        data.heapTotal;  
    }  
}
```

8. Добавьте в `index.html` поля для отображения информации

```
<h4 id='rss'></h4>  
<h4 id='heapTotal'></h4>  
<h4 id='heapUsed'></h4>
```

9. Проверьте работу проекта

### Data from server

RSS size is 26320896

Heap total size is 12828672

Heap used size is 12828672

15. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB6-monitor`

## Раздел 3. Чат на основе Socket.IO

На основе проекта ранее разработанного кода реализуйте простой чат с использованием Socket.IO.

Наглядную демонстрацию работы более сложного чата и исходный код можно найти <https://socket.io/demos/chat/>

8. Создайте новый проект SocketIOChat

a. Создайте модуль `server.js`, скопируйте код из проекта HTTPServer

9. Создайте новый модуль `socket-server.js`

b. Подключите модуль `server.js` в `socket-server.js`

c. Установите и подключите в проект библиотеку Socket.IO

```
// socketio-server.js
var server = require('./server');

const io = require('socket.io')(server);
```

10. Добавьте обработку нового соединения

```
io.on('connection', function (socket) {
  console.log ('new connection');
});
```

11. Внутри обработки соединения укажите, что при получении сообщения под грифом 'new message' необходимо отправить это сообщение всем текущим клиентам

```
io.on('connection', function (socket) {
  console.log ('new connection');

  socket.on('new message', function (data) {
    socket.broadcast.emit('new message', data);
  });
});
```

12. В качестве шаблона для клиентской страницы чата предлагается использовать ту же страницу, что и в разделе 1.

d. AJAX и WebSocket – встроенные объекты браузера и не требуют отдельного подключения, но Socket.IO – внешняя библиотека

e. Добавьте в `public/index.html` загрузку библиотеки

```
<script src="/socket.io/socket.io.js"></script>
```



13. Измените код в файле `public/client.js` на код с использованием `Socket.IO`
- f. При запуске страницы клиент должен подписаться на новые сообщения
  - g. При получении нового сообщения – выдать его на экран
  - h. При нажатии на кнопку – отправить сообщение сервера

```
14. var socket = io();
var socket = io();

publish.onsubmit = function () {
    socket.emit('new message', this.elements.message.value);

    this.elements.message.value="";
    return false;
};

socket.on('new message', function (newMessage) {
    var li=document.createElement ('li');
    li.textContent = newMessage;
    li.classList = "list-group-item";
    messages.appendChild (li);
});
```

15. Проверьте работу проекта
- i. Обратите внимание, что метод `io.broadcast` рассылает сообщение всем, КРОМЕ клиента, сгенерировавшего сообщение
16. `Socket.IO` предоставляет также функционал по подтверждению доставки сообщения это реализуется за счет дополнительного параметра-функции у `socket.emit('new message', data, function (data))` и у `socket.on ('new message', function (data))`.
- j. Пусть сервер при получении от клиента передаст его обратно клиенту (с помощью функции обратного вызова)
  - k. Клиент при отправке сообщения должен дожидаться подтверждения и выдать сообщение из подтверждения на экран

```
// client.js
var socket = io();

publish.onsubmit = function () {
    socket.emit('new message', this.elements.message.value,
```

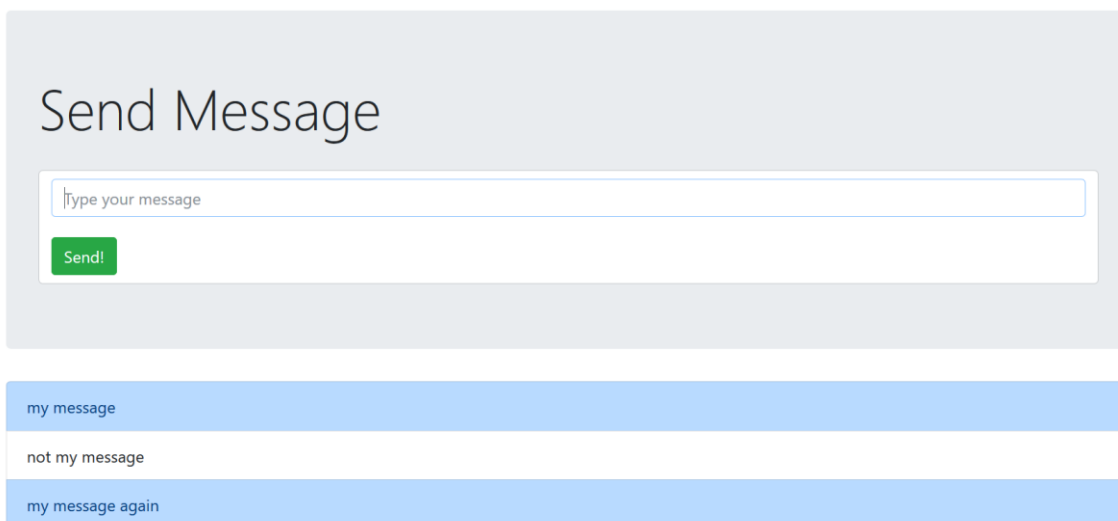
```
        function (newMessage){
            var li=document.createElement ('li');
            li.textContent = newMessage;
            li.classList = "list-group-item list-group-item-primary";
            messages.appendChild (li);
        });

        this.elements.message.value="";
        return false;
    };

// socketio-server.js
io.on('connection', function (socket) {
    console.log ('new connection');

    socket.on('new message', function (data, callback) {
        socket.broadcast.emit('new message', data);
        callback(data);
    });
});
```

### 17. Проверьте работу проекта



Send Message

Type your message

Send!

my message

not my message

my message again

18. Опционально. Добавьте дополнительные события
- l. Присоединение пользователя к чату и отключение
  - m. Пользователь печатает
  - n. И пр.
19. С готовым решением можно ознакомиться в [/nodejs-solutions/LAB6-socketio-chat](#)

## Упражнение 7. Фреймворк Express

---

### О чем это упражнение:

В этой лабораторной работе Вы продолжите развивать проект чата на основе длинных запросов, добавив журналирование, обработку ошибок и пр. с помощью функций промежуточной обработки.

### Что Вы должны будете сделать:

1. Модифицировать код проекта, добавив использование функций промежуточной обработки
2. Добавить собственную функцию обработки ошибок
3. Добавить авторизацию для администратора с помощью протокола авторизации HTTP

## **Раздел 1. Подключение функций промежуточной обработки для решения стандартных задач**

Модифицируйте веб-сервер, добавив ряд полезных и часто используемых функций:

- Журналирование
- Предоставление `favicon.ico`
- Предоставление клиенту не единого файла `html`, а группы файлов (изображения, скрипты, стили и пр.)

### 1. Установите и подключите в проект модуль `express`

```
npm install express
```

### 2. Создайте веб-сервер с помощью `express`

- а. Закомментируйте ранее написанный код для `http` сервера и проверьте работу кода ниже.
- б. Поведение веб-сервера по умолчанию – выдавать `404 Not Found` для любых запросов

```
var express = require ('express');  
var app = express ().listen (3000);
```

### 3. Добавьте предоставление страницы пользователю с помощью встроенной функции `express.static()`

- а. Создайте каталог `public` и переместите в него `index.html`
- б. Опционально: переформатируйте `index.html` таким образом, чтобы код JavaScript находился в отдельном файле

```
var app = express ()  
    .use (express.static('public'))  
    .listen (3000);
```

### 4. Предоставление иконки `favicon.ico` пользователю осуществляет модуль `serve-favicon`

- а. Установите и подключите в проект модуль `serve-favicon`

```
var app = express ()  
    .use (serve_favicon('favicon.ico'))  
    .use (express.static('public'))  
    .listen (3000);
```

5. Журналирование подключений к серверу осуществляет модуль `morgan` (предыдущее название `logger`)
- а. Обязательный параметр указывает подробность журналирования

```
var app = express ()
  .use (logger_morgan('tiny'))
  .use (serve_favicon('favicon.ico'))
  .use (express.static('public'))
  .listen (3000);
```

6. Функции промежуточной обработки могут быть подключены к определенному пути
- а. Подключите функции для `subscribe` и `publish`. Код функций аналогичен коду из соответствующих секций имеющегося `http`-сервера

```
var app = express ()
  .use (logger_morgan('tiny'))
  .use (serve_favicon('favicon.ico'))
  .use ('/subscribe', function (req,res) {
    chat.subscribe(req,res); })
  .use ('/publish', function (req,res) {
    // Код из секции publish
  })
  .use (express.static('public'))
  .listen (3000);
```

## Раздел 2. Обработка ошибок

Как Вы могли убедиться в предыдущих лабораторных работах при разработке веб-сервера без использования Express, любая необработанная ошибка приводит к завершению сервера.

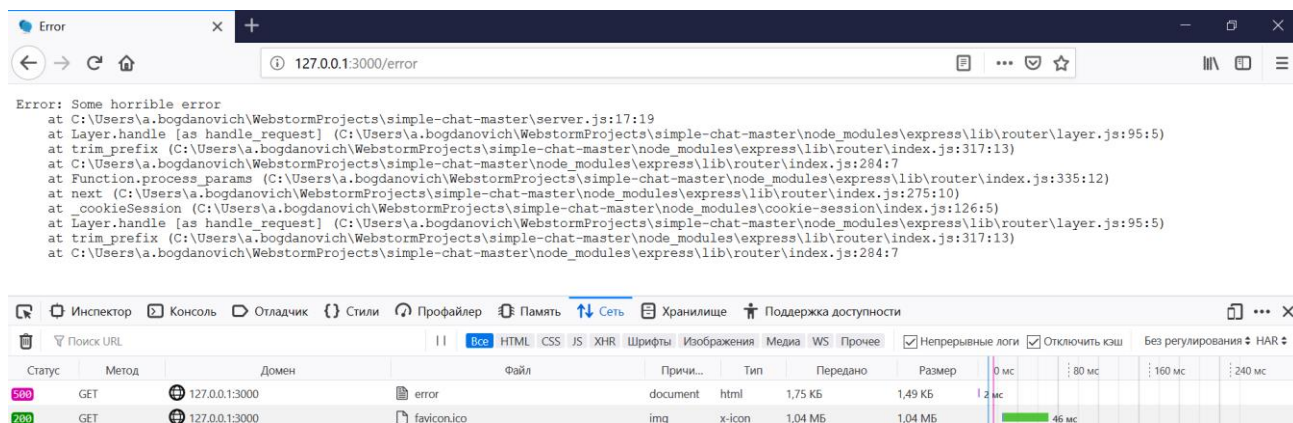
1. Проверьте стандартное поведение веб-сервера, разработанного с помощью Express, при возникновении ошибки

- а. Добавьте путь `/error` при обращении к которому происходит какая-либо ошибка (обращение к несуществующему файлу, обращение к несуществующему полю объекта и пр.)

```
app.use ('/error', function (req,res){
  throw new Error ('Some horrible error');
  res.end('error');
})
```

- а. Запустите сервер и перейдите на путь `/error`
- б. По умолчанию пользователь получит код ответа 500

Internal Error и текст ошибки, дальнейшая обработка этого запроса прекратится. Сервер при этом продолжит работать и обрабатывать иные запросы



2. Express предлагает возможность собственной обработки ошибок

- а. Добавьте в конец цепочки обработчиков обработку ошибки

```
app.use (function (err,req,res,next){
  res.write('<h1>Internal Server Error</h1>');
  res.write('<h2>Come back later</h2>');
  res.end(err.message);
})
```

### **Раздел 3. Разработка собственных функций промежуточной обработки. Авторизация HTTP (опционально)**

Модифицируйте веб-сервер, добавив авторизацию администратора.

Для этого в данном решении предлагается использовать базовую авторизацию с помощью браузера и кода HTTP 401 Unauthorized

- При подключении клиента проверьте заголовки HTTP Authorization, если они пустые – верните пользователю код 401
- Если заголовки непустые, то они должны содержать строку «Basic user:password», где строка user:password закодирована по алгоритму base64 (пример «Basic Qm9yaXM6MTIz»)
- Проверьте пароль администратора (в данном примере для простоты предлагается фиксированный пароль admin /admin ). Если пароль неверный – верните код 401, если все в порядке – перейдите к следующей функции

1. Разработайте функцию промежуточной обработки для авторизации
  - a. Помните, что после авторизации функция должна передавать управление следующему модулю по цепочке
  - b. Вынесите функцию в отдельный модуль

```
// auth.js
module.exports = function (req, res, next) {
  if (!req.headers.authorization) {
    res.set('WWW-Authenticate', 'Basic').sendStatus(401).end();
    return;
  }
  var auth = new Buffer.from(req.headers.authorization.split(' ')[1],
    'base64').toString().split(':');
  var user = auth[0];
  var pass = auth[1];

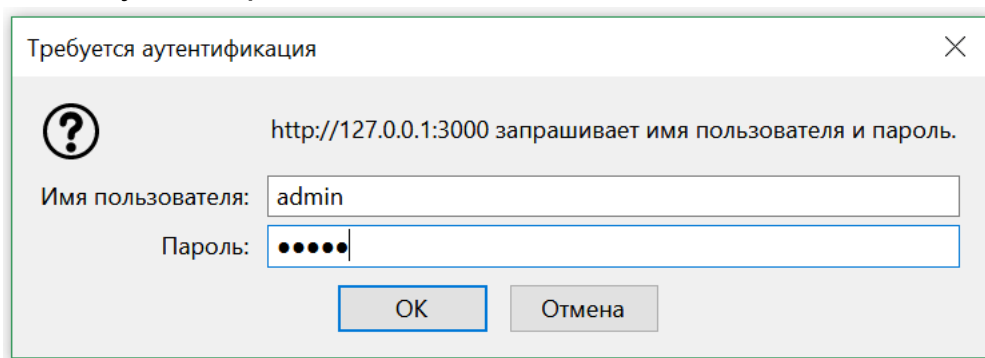
  if (user == 'admin' && pass == 'admin') {
    next(); // all good
  }
  else res.set('WWW-Authenticate', 'Basic').sendStatus(401).end();
}
```

2. Подключите эту функцию к общему приложению

```
var auth = require ('./auth');

var app = express ()
  .use (logger_morgan('tiny'))
  .use (serve_favicon('favicon.ico'))
  .use ('/admin', auth)
  .use ('/admin', function (req, res) {
    res.end('Hello admin');
  })
  ...
```

3. При подключении к чату теперь сервер запрашивает аутентификацию
  - а. Проверьте поведение сервера при успешной и неуспешной аутентификации



20. С готовым решением можно ознакомиться в /nodejs-solutions/LAB7-chat



## Упражнение 8. Генератор сайта

---

### О чем это упражнение:

В этой лабораторной работе Вы создадите веб-сайт с нуля, используя генератор Express, и на основе предыдущих разработок сделаете чат с авторизацией и отслеживанием состояния пользователей.

### Что Вы должны будете сделать:

1. Сгенерировать сайт и изучить содержимое файлов
2. Перенести ранее разработанный код в новый проект
3. Создать страницу регистрации и входа пользователей
4. Настроить пользовательские сессии

## Раздел 1. Генерация каркаса сайта

### 1. Сгенерируйте сайт с помощью express-generator

- a. Установите express-generator как глобальный модуль (утилиту)

```
sudo npm install express-generator -g
```

- b. Создайте «каркас» сайта

```
express express-chat
```

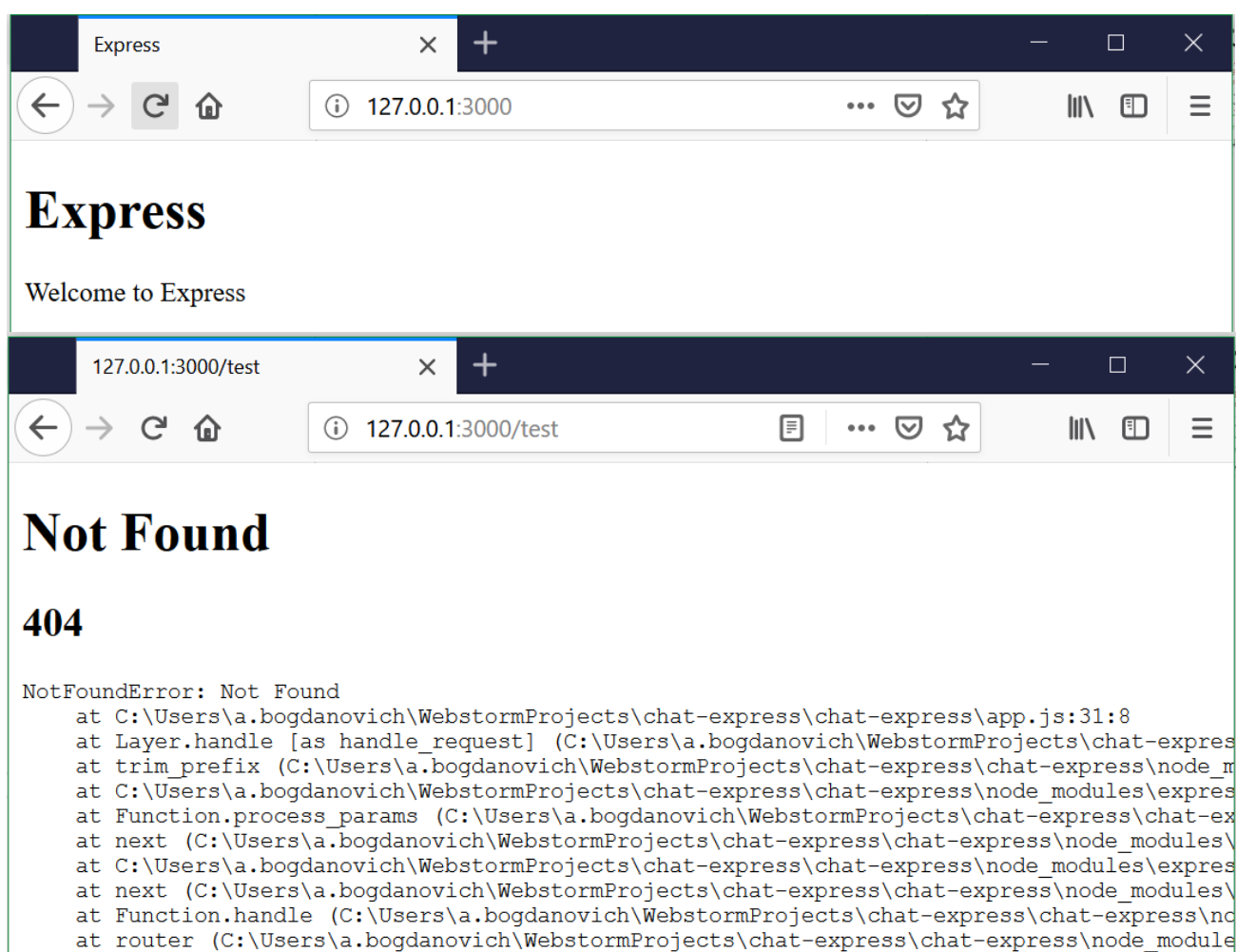
- c. Установите зависимости

```
cd express-chat; npm install
```

- d. Запустите сайт

```
npm start
```

- e. По умолчанию на сайте созданы две страницы (главная и /user), а на все остальные запросы сайт выдает 404



### 2. Изучите созданный каркас сайта

- Какие модули заранее подключены в проект?
- Какие страницы заранее созданы для сайта?
- С помощью каких модулей осуществляется обработка перехода на эти страницы?

Например, рассмотрим формирование главной страницы

- `app.js` подключает модуль `routes/index.js` и вызывает его при обращении к главной странице

```
var indexRouter = require('./routes/index');
app.use('/', indexRouter);
```

- `routes/index.js` вызывает генерацию HTML-страницы по шаблону `view/index.jade`, подставляя значение переменной `title`

```
var router = express.Router();
/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Express' });
});
module.exports = router;
```

- Файл шаблона `view/index.jade` определяет элементы страницы и подключает общий формат страниц `layout.jade`. Обратите внимание на два способа подстановки значения переменной `title`

```
extends layout

block content
h1= title
p Welcome to #{title}
```

- Файл `layout.jade` описывает общий формат страниц, например стиль

### 3. Вместо статических HTML-страниц в современной веб-разработке часто используют шаблоны веб-страниц, которые позволяют легко менять представление страницы для конкретной ситуации. По

умолчанию `express-generator` использует шаблоны Jade (новое название Pug)

- a. Измените код проекта таким образом, чтобы на главной странице было написано «Welcome to my chat!»

```
// routes/index.js
var router = express.Router();
/* GET home page. */
router.get('/', function(req, res) {
  res.render('index', { title: 'Welcome to my chat!' });
});
module.exports = router;
```

- b. Подключите свою библиотеку стилей в `layout` (например, bootstrap)

```
//- layout.jade
doctype html
html
  head
    title= title
    link(rel='stylesheet',
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/css/bootstrap.min.css")
  body
    block content
```

#### 4. Перенесите код чата в сгенерированный каркас.

- a. При обращении к главной странице, выдайте чат пользователю.
  - i. Шаблон страницы для страницы чата можно найти в `/nodejs-labfiles/LAB8-chat/chat.jade`, скопируйте его в каталог `views`

```
// routes/index.js
var express = require('express');
var router = express.Router();

router.get('/', function(req, res) {
  res.render('chat', {title: 'Chat'});
});
```

- a. Добавьте в файл `index.js` логику работы чата (используйте код, разработанный в предыдущих лабораторных).

- b. Обратите внимание на изменение пути для подключения модуля `chat.js`, т.к. подключение происходит из модуля `routes/index.js`

```
var chat = require ('../chat');

router.get ('/subscribe', function (req,res,next) {
  chat.subscribe(req,res);
});
router.post ('/publish', function (req,res,next) {
  ...
}
```

- с. Проверьте, что модуль `router` экспортирован из модуля `routes/index.js` и корректно подключен в `app.js`

```
// routes/index.js
module.exports = router;

// app.js
var indexRouter = require('./routes/index');
app.use('/', indexRouter);
```

## Раздел 2. Страница регистрации пользователей

Сгенерированный сайт легко расширять. Добавьте страницу для регистрации и аутентификации пользователей.

1. Добавьте в проект страницу для входа и регистрации пользователя в системе

- d. Создайте отдельный модуль маршрутизации для регистрации (как пример можно использовать `route/index.js`). Пока единственный функционал – выдавать пользователю страницу для входа в систему

- i. Шаблон страницы для страницы чата можно найти в `/nodejs-labfiles/LAB8-chat/login.jade`, скопируйте его в каталог `views`

```
// routes/login.js
var express = require('express');
var router = express.Router();

/* GET login page. */
router.get('/', function(req, res, next) {
  res.render('login', { title: 'Login' });
});
module.exports = router;
```

- e. Подключите модуль в проект

```
var loginRouter = require ('./routes/login');
app.use ('/login', loginRouter);
```

2. Шаблон страницы для регистрации содержит описание двух форм: для входа существующего пользователя и для регистрации нового
  - a. При попытке входа существующего пользователя форма отправляет запрос POST на `/login/existing`
  - b. При попытке регистрации нового пользователя форма отправляет запрос POST на `/login/new`
  - c. Добавьте обработку этих действий в модуль `routes/login.js`
  - d. В текущей реализации пока нет возможности сохранять зарегистрированных пользователей и их пароли в базе данных, поэтому проверку существования пользователя и

верности пароля пока пропустим. Код ниже проверяет только то, что пароль не пустой.

```
router.post ('/new', function (req,res,next){
    // Проверить, что нет пользователя с таким именем
    // Проверить введенные значения
    if (req.body.password == req.body.confirmPassword)
    {
        // Зарегистрировать пользователя
        // Перенаправить пользователя на страницу чата
        res.redirect ('/');

    } else res.render ('login', {title: 'Login', new_message:'Password and
        confirmation are not the same'})
});
router.post ('/existing', function (req,res,next){
    // Проверить существование пользователя
    // Проверить, что пароль верный
    if (req.body.password)
    {
        // Пометить пользователя, как авторизованного
        // Перенаправить пользователя на страницу чата
        res.redirect ('/');

    } else res.render ('login', {title: 'Login', exist_message:'Wrong password'})
});
```

### 3. Проверьте работу проекта.

- a. При обращении к главной странице <http://127.0.0.0:3000/> сайт должен перенаправлять пользователя на страницу регистрации
- b. При регистрации или попытке войти (любое имя пользователя и не пустой пароль) сайт должен перенаправлять пользователя в чат.

The screenshot shows a web browser window with two tabs. The active tab is titled 'Login' and shows a web page with two forms side-by-side. The browser's address bar displays '127.0.0.1:3000/login'.

**Sign Up Form:**

- Title: Sign Up
- Label: Name:
- Input: Enter your name
- Label: Password:
- Input: (empty)
- Button: Login

**Register Form:**

- Title: Register
- Label: Name:
- Input: Enter your name
- Label: Email:
- Input: address@email.com
- Label: Password:
- Input: (empty)
- Label: Confirm Password:
- Input: (empty)
- Button: Register



### Раздел 3. Куки и сессии пользователей

Текущая реализация предполагает, что пользователь каждый раз перенаправляется на страницу авторизации, а затем уже в чат.

Добавьте:

- Проверку на то, проходил ли уже этот пользователь авторизацию
- Отображение имени пользователя на странице чата и в сообщениях

Для хранения информации о состоянии пользователя удобнее всего использовать пользовательские сессии с хранением данных в куки клиента (`cookie-session`) памяти сервера или базе данных (`express-session`).

- Синтаксис и возможности модулей аналогичны
- Воспользуемся модулем `express-session` для более простого переноса информации в постоянное хранилище в следующей лабораторной

1. Установите и подключите модуль `express-session` как функцию промежуточной обработки
  - a. Подключить модуль `express-session` нужно ДО маршрутизаторов, чтобы иметь возможность использовать пользовательские сессии в модулях `index.js` и `login.js`
  - b. При подключении можно указать опции
    - i. `Secret` – ключ, который будет использоваться для подписи и шифрования
    - ii. `Cookie.httpOnly` – по умолчанию `true`, запрет на доступ к куки из браузера

```
// app.js
var expressSession = require ('express-session'); // npm install

var app = express();
...
app.use (expressSession ({ secret: 'work hard'}));
...
app.use('/', indexRouter);
```

2. В модуле `login.js` перед перенаправлением пользователя в чат добавьте сохранение состояния пользователя в объект `req.session`

### а. Для `/new` и `/existing`

```
router.post ('/existing', function (req,res,next){
  // Проверить существование пользователя
  // Проверить, что пароль верный
  if (req.body.password)
  {
    // Пометить пользователя, как авторизованного
    req.session.loggedIn = true;
    // Перенаправить пользователя на страницу чата
    res.redirect ('/');
  }
  else res.render ('login', {title: 'Login', exist_message:'Wrong password'})
});
```

### 3. В модуле `index.js` перед перенаправлением пользователя на страницу авторизации добавьте проверку состояния пользователя

```
// routes/index.js

/* GET chat page. */
router.get('/', function(req, res) {
  if (!req.session.loggedIn) res.redirect ('/login')
  else res.render ('chat', {title:'Chat'});
});
```

### 4. Проверьте работу проекта

- При первом обращении сайт должен перенаправить пользователя на страницу авторизации
- Если после успешно пройденной авторизации открыть еще одну вкладку браузера, пользователь сразу должен попасть в чат
- Закройте браузер и откройте его еще раз. Сохраняется ли состояние пользователя? Почему?

### 5. Сервер сохраняет идентификатор сессии, полученный от `express-session` при первом подключении, в куки.

- По умолчанию браузер при закрытии очищает сессионные куки.
- Если Вы хотите сохранять состояние пользователя в течение определенного времени вне зависимости от состояния браузера, добавьте параметр `maxAge` при подключении `express-session`

```
// app.js
```

```
app.use (expressSession ({
  secret: 'work hard',
  cookie: {maxAge:9000000}
}));
```

6. Объект `req.session` может хранить любую информацию, связанную с пользователем. Добавьте сохранение имени пользователя при успешной авторизации

```
router.post ('/existing', function (req,res,next){
  // Проверить существование пользователя
  // Проверить, что пароль верный
  if (req.body.password)
  {
    // Пометить пользователя, как авторизованного
    req.session.loggedIn = true;
    req.session.user_name = req.body.name;
    // Перенаправить пользователя на страницу чата
    res.redirect ('/');
  }
```

7. Добавьте отображение имени на странице чата

- Страница чата формируется из шаблона `chat.jade`
- Рендеринг шаблона вызывается из файла `routes/index.js`, при рендеринге передается переменная `title`
- Добавьте передачу переменной `user` и отображение этой переменной в заголовке. Обращение к переменной внутри шаблона - `#{user}`

```
// routes/index.js

/* GET chat page. */
router.get('/', function(req, res) {
  console.log (req.session.loggedIn);
  if (!req.session.loggedIn) res.redirect ('/login')
  else res.render ('chat', {title:'Chat', user:req.session.user_name});
});

//- chat.jade
extends layout

block content
  .container
    div.jumbotron
      h1.display-4 Welcome, #{user}
      br
      form#publish
```

```
input#name.form-control(type='text', placeholder='Type your message'
name='message')
br
button.btn.btn-primary(type='submit') Send
ul#messages.list-group

script(src="../../public/javascripts/client.js")
```

8. Для того чтобы сделать чат еще более удобным, добавьте имя пользователя при публикации сообщений

a. Измените функцию `chat.publish` так чтобы она принимала имя пользователя

```
// chat.js module
exports.publish =function (name, message) {
  clients.forEach (function (res){
    console.log (res);
    res.end (name + ": " + message);
  })
  clients = [];
}
```

b. Измените вызов функции `chat.publish`

```
// routes/index.js
router.post ('/publish', function (req,res,next) {
  ...
  chat.publish (req.session.user_name, body.message);
  ...
})
```

21. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB8-chat`

## Упражнение 9. Хранение данных

---

### О чем это упражнение:

В этой лабораторной работе Вы добавите в проект чата возможность хранить информацию о пользователях и пользовательских сессиях в базе данных MongoDB

### Что Вы должны будете сделать:

1. Установить базу данных MongoDB
2. Подключить базу данных к проекту Node.js
3. Научиться создавать, изменять, искать и удалять документы из базы данных
4. Настроить хранение пользователей чата в базе данных. Сделать полноценную регистрацию и аутентификацию пользователей
5. Настроить хранение пользовательских сессий в базе данных

## Раздел 1. Установка базы данных

Документацию по работе с базой данных MongoDB можно найти на сайте <https://docs.mongodb.com>

### 1. Создайте файл с описанием репозитория для установки MongoDB

```
Sudo nano /etc/yum.repos.d/mongodb-org-4.0.repo

[mongodb-org-4.0]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-
org/4.0/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-4.0.asc
```

- a. Также можно скопировать файл репозитория из каталога  
/nodejs-labfiles/install

```
# sudo cp /nodejs-labfiles/install/ mongodb-org-4.0.repo /etc/yum.repos.d/
```

### 2. Установите базу данных

```
sudo yum install -y mongodb-org
```

### 3. Запустите службу MongoDB

```
sudo service mongod start
```

### 4. Проверьте, что служба базы данных успешно запущена

- a. Откройте файл /var/log/mongodb/mongod.log
- b. Найдите в нем строку [initandlisten] waiting for  
connections on port <port>

```
tail /var/log/mongodb/mongod.log
```

### 5. Запустите командный интерфейс Mongo

- a. Выведите список доступных команд

```
Help
```

- b. Выведите список созданных баз данных

```
show dbs
```

с. Выведите список коллекций в базе данных `local`

```
use local  
show collections
```

д. Выведите содержимое документов из выбранной коллекции

```
db.local.find().pretty()
```

## Раздел 2. Хранение документов в базе данных

1. Для использования MongoDB из проекта Node.js необходимо установить драйвер работы с базой данных

```
$ npm install mongodb
```

2. Создайте отдельный модуль `createDB.js`
  - a. Подключите базу данных `User`
  - b. Не забывайте обрабатывать ошибки в асинхронных вызовах
  - c. Не забудьте закрыть базу данных по окончании работы

```
var MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://127.0.0.1:27017', function(err, database) {
  if (err) throw err;

  var db = database.db("User");
  database.close();
})
```

3. В коллекцию `users` добавьте документ, соответствующий пользователю `testUser`

```
MongoClient.connect('mongodb://127.0.0.1:27017', function(err, database) {
  if (err) throw err;

  var db = database.db("User");
  var collection = db.collection('test_insert');

  var testUser = {name: "John", lastName: "Doe", age: 53};
  collection.insertOne (testUser, function (err, data){
    if (err) throw err;
    console.log (data.ops);
    database.close();
  })
})
```

4. Запустите проект несколько раз.
5. Найдите в базе данных все документы пользователей с именем John

```
MongoClient.connect('mongodb://127.0.0.1:27017', function(err, database) {
  if (err) throw err;

  var db = database.db("User");
  var collection = db.collection('test_insert');
```



```
collection.find ({name:"John"}).toArray(function (err,data){
    if (err) throw err;
    console.log (data);
    database.close();
})
})
```

6. Найдите и удалите в базе банных все документы пользователей с именем John.

а. Выведите на экран число и содержание удаленных документов

```
MongoClient.connect('mongodb://127.0.0.1:27017', function(err,database) {
    if (err) throw err;

    var db = database.db("User");
    var collection = db.collection('test_insert');

    collection.find ({name:"John"}).toArray(function (err,data){
        if (err) throw err;
        console.log (data);
        collection.deleteMany ({name:"John"}, function (err,data){
            if (err) throw err;
            console.log (data.deletedCount);
            database.close();
        })
    })
})
```

## Раздел 3. Создание модели Mongoose для пользователей

В данном разделе Вы опишете модель пользователя базы данных. Предлагается следующая схема пользователя:

- Имя (уникальное)
- Почта
- Пароль
- Дата регистрации пользователя

Обратите внимание, что хранение паролей в открытом виде недопустимо, поэтому предлагается хранить в базе данных хеш HMAC от пароля с добавлением «соли» (случайной строки для затруднения перебора паролей)

### 1. Установите и подключите mongoose

#### а. Создайте файл `models/user.js`

```
// models/user.js
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/chat');
```

### 2. Опишите схему для пользователя

```
var schema = mongoose.Schema ({
  username: {
    type:String,
    unique:true,
    required: true
  },
  email: String,
  hashedPassword: {
    type: String,
    required: true
  },
  salt: {
    type: String,
    required:true
  },
  created: {
    type: Date,
    default: Date.now
  }
});
```

### 3. Пароль пользователя не должен храниться в базе данных, поэтому опишите пароль, как виртуальное свойство

- a. Метод `set` должен описывать установку пароля (вычисление хеш-суммы). Для вычисления хеш-суммы предлагается использовать модуль `crypto`
- b. Метод `get` должен описывать чтение пароля

```
schema.methods.hashPassword = function (password) {  
  return crypto.createHmac ('sha1', this.salt)  
    .update (password).digest ('hex');  
}  
schema.virtual('password')  
  .set (function (password) {  
    this._plainPassword = password;  
    this.salt = Math.random()+'';  
    this.hashPassword = crypto.createHmac ('sha1', this.salt)  
      .update (password).digest ('hex');  
  })  
  .get (function () {  
    return this._plainPassword  
  });
```

4. Опишите метод проверки пароля (при проверке сравнивается хеш предложенного пароля с тем, что сохранен в базе данных)

```
schema.methods.checkPassword = function (password) {  
  return this.hashPassword(password) == this.hashPassword;  
};
```

5. На основе описанной схемы создайте модель и экспортируйте ее

```
exports.User = mongoose.model ('User', schema);
```

## Раздел 4. Использование модели Mongoose для сохранения пользователей в базе данных

1. В файле routes/login.js подключите описанную модель

```
var User = require('../models/user').User;
```

2. Реализуйте логику входа пользователя на сайт.

Необходимо убедиться, что:

- а. Пользователь с таким именем существует
- б. Пароль указан верный

```
router.post ('/existing', function (req,res,next){

    // Пользователь с таким именем существует?
    User.findOne ({username: req.body.name}, function (err,user){
        if (err) throw err;

        if (user) {
            // Пароль верный?
            if (user.checkPassword (req.body.password)) {

                // Переправить пользователя в чат
                res.redirect ('/');

            } else {
                res.render ('login', {title: 'Login',
                    exist_message:'Wrong password'});
            }
        } else {
            res.render ('login', {title: 'Login',
                exist_message:'User does not exist'});
        }
    })
});
```

3. Реализуйте логику регистрации нового пользователя.

Необходимо проверить:

- а. Пользователя с таким именем не существует
- б. Введенные пароли совпадают

```
router.post ('/new', function (req,res,next){
    // Пользователь с таким именем уже существует?
    User.findOne ({username:req.body.name}, function (err,user){
        if (err) throw err;

        if (user) {
            res.render ('login', {title: 'Login',
```

```
        new_message: 'User with this name already exists'}));  
    } else {  
        // Введенные данные корректны?  
        if (req.body.password == req.body.confirmPassword)  
        {  
            // Зарегистрировать пользователя  
            var newuser = new User ({  
                username: req.body.name,  
                password: req.body.password,  
                email: req.body.email  
            });  
  
            newuser.save (function (err){  
                if (err) return next(err);  
  
                // Перенаправить в чат  
                res.redirect ('/');  
            });  
        } else {  
            res.render ('login', {title: 'Login',  
                new_message: 'Password and confirmation are not the same'});  
        }  
    }  
})  
  
});
```

## **Раздел 5. Сохранение сессий пользователей в базе данных**

В текущей реализации сервер хранит данные пользовательских сессий в памяти с помощью модуля `express-session`. Данный вариант не является допустимым только в тестовых решениях, т.к. в продуктивной среде может привести к переполнению памяти сервера при большом количестве сессий.

1. Добавьте хранение сессий в базе данных MongoDB
  - a. Установите и подключите модуль `connect-mongo`
  - b. Настройте хранение пользовательских данных в базе данных `session`

```
// models/session-store.js
Var expressSession = require ('express-session');
var MongoStore = require('connect-mongo')(expressSession);

var sessionStore = new MongoStore({
  url: 'mongodb://127.0.0.1:27017/session'
});

module.exports = sessionStore;
```

2. Добавьте хранилище `sessionStore` в конфигурацию `express-session`

```
// app.js
var sessionStore = require('./models/session-store');

var app = express();

app.use (expressSession ({
  secret: 'work hard',
  cookie: {maxAge:9000000},
  store: sessionStore
}));
```

22. С готовым решением можно ознакомиться в `/nodejs-solutions/LAB9-chat`

## Упражнение 10. Асинхронная разработка и тестирование

---

### О чем это упражнение:

В этой лабораторной работе Вы изучите тестирование с помощью модуля `assert` и фреймворка `Mocha` на примере различных реализаций функции, которая преобразует содержимое файла в объект `JSON`.

### Что Вы должны будете сделать:

1. Разработать и протестировать решение на основе синхронных функций
2. Разработать и протестировать решение на основе функций обратного вызова
3. Разработать и протестировать решение на основе `Promise` (с использованием ключевых слов `async` / `await`)

## **Раздел 1. Тестирование синхронных функций с помощью фреймворка Mocha**

Разработайте функцию `parseJSON_sync`, которая принимает имя файла в качестве параметра и выдает:

- В случае успеха объект JSON
- В случае некорректного содержимого ошибку с текстом `'Bad JSON'`
- В случае проблем чтения файла – ошибку файловой системы

Для тестирования используйте файлы `good.json`, `bad.json` и `nonexist.json`. Файлы можно скопировать из каталога `/nodejs-labfiles/LAB10-jsons`

### 1. Создайте новый каталог и новый модуль `sync_JSON.js`

```
var fs = require ('fs');

function loadJSONsync(filename) {
  var file = fs.readFileSync (filename);
  return JSON.parse (file);
}

module.exports = loadJSONsync;
```

### 2. Создайте модуль тестирования `test_sync_JSON.js` и разработайте код тестирования для фреймворка Mocha

- а. Непосредственно логику тестов реализуйте с помощью модуля `assert`

```
var assert = require('assert');
var loadJSON = require('./sync_JSON');

describe ('JSON test', function() {
  it ('good JSON', function () {
    var data = loadJSON('good.json');
    assert.equal (data.name, "John");
  });
  it ('bad JSON', function () {
    assert.throws (function () {
      loadJSON('bad.json')
    }, function (err){
```



```
        return ((err instanceof Error) && (err.message == 'Bad JSON'))
    })
});
it ('nonexistent JSON', function (){
    assert.throws (function (){
        loadJSON('nonexist.json')
    }, function (err){
        return ((err instanceof Error) && (err.code == 'ENOENT'))
    })
});
});
```

### 3. Установите Mocha как глобальную утилиту

```
sudo npm install -g mocha
```

### 4. Протестируйте модуль `sync_JSON.js`. Все ли тесты дают ожидаемый результат?

```
mocha .\test_sync_JSON.js
```

```
JSON test
  ✓ good JSON
  1) bad JSON
  ✓ nonexistent JSON

2 passing (36ms)
1 failing

1) JSON test
   bad JSON:
  SyntaxError: Unexpected token ; in JSON at position 24
    at JSON.parse (<anonymous>)
    at loadJSONsync (sync_JSON.js:18:17)
    at \test_JSON\test_sync_JSON.js:11:13
    at getActual (assert.js:578:5)
    at Function.throws (assert.js:690:24)
    at Context.<anonymous> (test_sync_JSON.js:10:16)
```

### 5. Исправьте код модуля `sync_JSON.js`

```
function loadJSONsync(filename) {
    var file = fs.readFileSync (filename);
    try {
        return JSON.parse (file);
    } catch (err) {
        throw new Error ('Bad JSON');
    }
}
```

## **Раздел 2. Разработка и тестирование асинхронных функций**

Несмотря на то, что синхронный вариант работает хорошо, на продуктивном сервере необходимо использовать асинхронные функции, т.к. это значительно повышает производительность.

Разработайте модуль `parseJSON_callback` с аналогичным функционалом.

1. Создайте новый модуль `parseJSON_callback.js` и разработайте код
  - a. Помните, что асинхронные функции не должны выбрасывать исключения через `throw`. Вместо этого необходимо обработать ошибку или передать ее первым аргументом в функцию обратного вызова
  - b. Код должен быть разработан таким образом, чтобы ни при каких обстоятельствах функция обратного вызова не была запущена дважды

```
var fs = require ('fs');

function loadJSONcb(filename, cb) {
  fs.readFile(filename, function (err, data) {
    if (err) cb(err);
    else cb(null, JSON.parse(data));
  });
};

module.exports = loadJSONcb;
```

2. Разработайте тесты для модуля `parseJSON_callback.js`
  - a. Для асинхронных функций Mocha предоставляет специальный параметр `done()`, который позволяет указать, когда необходимо проводить результата работы асинхронной функции

```
var assert = require('assert');
var loadJSON = require('./cb_JSON');

describe ('JSON test', function() {
  it ('good JSON', function (done){
    loadJSON('good.json', function (err, data) {
      done (err);
      assert.equal (data.name, "John");
    });
  });
});
```

```
    });  
  });  
  
  it ('bad JSON', function (done){  
    loadJSON('bad.json', function (err, data) {  
      if ((err instanceof Error) && (err.message == 'Bad JSON')) done ();  
      else done (err);  
    });  
  });  
  
  it ('nonexistent JSON', function (done){  
    loadJSON('nonexist.json', function (err, data) {  
      if ((err instanceof Error) && (err.code == 'ENOENT')) done ();  
      else done (err);  
    });  
  });  
})
```

### 3. Проверьте результаты тестирования

```
mocha .\test_cb_JSON.js
```

```
JSON test  
  ✓ good JSON  
  1) bad JSON  
  ✓ nonexistent JSON  
  
2 passing (25ms)  
1 failing  
  
1) JSON test  
   bad JSON:  
     Uncaught SyntaxError: Unexpected token ; in JSON at position 24  
       at JSON.parse (<anonymous>)  
       at \test_JSON\cb_JSON.js:21:28  
       at FSReqWrap.readFileAfterClose [as oncomplete]  
       (internal/fs/read_file_context.js:53:3)
```

### 4. Добавьте проверку на неверный формат файла

```
function loadJSON(filename, cb) {  
  fs.readFile(filename, function (err, data) {  
    if (err) cb(err);  
    else try {  
      cb(null, JSON.parse(data));  
    }  
    catch {  
      cb(new Error ("Bad JSON"));  
    }  
  });  
};
```

### 5. Проверьте результат тестирования.

```
mocha .\test_cb_JSON.js
```

```
JSON test
  ✓ good JSON
  ✓ bad JSON
  ✓ nonexistent JSON
```

6. Несмотря на то, что все три теста выдают положительный результат, код содержит ошибку. В ходе тестирования не была учтена возможность ошибки в передаваемой функции обратного вызова.

а. Добавьте еще один тест для проверки этой ситуации

```
it ('good JSON, bad callback', function (done){
  loadJSON('good.json', function (err, data) {
    done (err);
    throw new Error ("Test callback error");
  });
});
```

б. Запустите новый набор тестов

```
mocha .\test_cb_JSON.js
```

```
JSON test
  ✓ good JSON
  ✓ good JSON, bad callback
  1) good JSON, bad callback
  ✓ bad JSON
  ✓ nonexistent JSON

4 passing (38ms)
1 failing

1) JSON test
   good JSON, bad callback:
     Error: Bad JSON (and Mocha's done() called multiple times)
       at \test_JSON\cb_JSON.js:25:16
       at FSReqWrap.readFileAfterClose [as oncomplete]
(internal/fs/read_file_context.js:53:3)
```

с. Исправьте код

```
function loadJSON(filename, cb) {
  fs.readFile(filename, function (err, data) {
    if (err) {
      cb(err);
    }
  });
}
```

```
        return;
    }
    else try {
        var parse_data = JSON.parse(data);
    }
    catch {
        cb(new Error ("Bad JSON"));
        return;
    }
    cb (null,parse_data);
});
```

- d. Убедитесь, что три стандартных теста работают, а последний выдает ошибку „Test callback error”

```
mocha .\test_cb_JSON.js
```

```
JSON test
  ✓ good JSON
  ✓ bad JSON
  ✓ nonexistent JSON
  ✓ good JSON, bad callback
  1) good JSON, bad callback

4 passing (25ms)
1 failing

1) JSON test
   good JSON, bad callback:
     Uncaught Error: Test callback error
       at \test_JSON\test_cb_JSON.js:28:19
       at \test_JSON\cb_JSON.js:31:9
       at FSReqWrap.readFileAfterClose [as oncomplete]
(internal/fs/read_file_context.js:53:3)
```

## Раздел 3. Разработка асинхронных функций с помощью *async / await*

Асинхронный код на основе функций обратного вызова может быть сложен для разработки и понимания, поэтому в последних стандартах JavaScript была добавлена поддержка объектов Promise и ключевых слов `async / await`.

Переработайте код с использованием новых технологий.

### 1. Создайте модуль `async_JSON.js`

```
var fs = require('fs');
var util = require('util');
var readAsyncFile = util.promisify (fs.readFile);

async function loadJSONasync (filename) {
  var data = await readAsyncFile (filename);
  try {
    return JSON.parse (data);
  } catch (err) {
    throw new Error ('Bad JSON');
  }
}

module.exports = loadJSONasync;
```

### 2. Разработайте тесты для нового модуля

```
var assert = require('assert');
var loadJSON = require ('./async_JSON');

describe ('JSON test', function() {
  it ('good JSON', function (done){
    loadJSON('good.json')
      .then(function (data) {
        assert.equal (data.name, "John");
        done();
      }).catch (done)
  });
  it ('bad JSON', function (done){
    loadJSON('bad.json')
      .catch (function (err) {
        if ((err instanceof Error) && (err.message == 'Bad JSON')) done
      });
    else done (err);
  });
});
```

```
    });  
  });  
  it ('nonexistent JSON', function (done){  
    loadJSON('nonexist.json')  
    .catch (function (err) {  
      if ((err instanceof Error) && (err.code == 'ENOENT')) done ();  
      else done (err);  
    });  
  });  
})
```

23. С готовыми решениями можно ознакомиться в `/nodejs-solutions/LAB10-test`