Разработка блога на Node.js, Express & Mongoose

Иван Андреевич Хромов
преподаватель

ГАПОУ СО
«Асбестовский политехникум»

Программирование в компьютерных системах, 4 курс
сделано в LaTeX

Аннотация

Node.js предлагает разработчику огромный выбор инструментов для работы с веб-приложениями. От самых примитивных (модуль `http` из стандартной библиотеки Node) до высокоуровневых фреймворков (Express.js, Hapi, Sails, Polka) с различным уровнем абстракции.

Абстрактивность веб-фреймворка можно поделить на полную (Hapi, Sails.js) и частичную (Express.js, Polka).

При полной абстракции, разработчик может опустить такие моменты как отправка HTTP-заголовков, кеширование и работу с сессиями. При частичной — данная функция ложится полностью или частично на плечи разработчика.

Как вы узнали из лекций, Node довольно молодая платформа. Но уже за 9 лет обрела огромную популярность.

Данная лабораторная работа посвящена разработке небольшой блоговой платформы без аутентификации пользователей. На выполнение работы выделяется 4 академических часа, после выполнения ее необходимо сдать преподавателю. Система оценивания следующая:

- Оценка «отлично» ставится в случае, если студент сдает работу без допущенных ошибок и отвечает на один вопрос преподавателя
- Оценка «хорошо» ставится в случае, если студент выполнил 100% работы и не ответил на поставленный вопрос
- Оценка «удовлетворительно» ставится в случае допущения грубых нарушений в работе
- Оценка «неудовлетворительно» ставится в случае, когда студент не сдал работу

Огромная просьба, выполнять работу лучше одному человеку, но в случае нехватки компьютеров разрешается объединиться в пары из двух человек.

Преподаватель вправе не поставить вам оценку, если во время работы были свободные ПК, но вы выполняли работу в паре.

В случае работы в паре, преподаватель задает разные вопросы каждому.

Подготовка рабочей среды

Для выполнения лабораторной работы вам необходимо иметь следующее ПО на вашем ПК:

- MongoDB (`https://mongodb.org`)
- Node.js (`https://nodejs.org`)
- Atom (`https://atom.io`) или
  Sublime Text (`https://sublimetext.com/3`) или
  Visual Studio Code (`https://code.visualstudio.com`)

При разработке вы можете использовать один из приведенных выше редакторов кода. Notepad++ использовать можно, но нормальной подсветки синтаксиса для нового диалекта JavaScript вы там не увидите. Используйте его на свой страх и риск.

Для работы создайте каталог `weblog` в любом месте. Убедитесь, что в пути к этому каталогу нет русских символов, это может привести к проблемам.

Условные обозначения

Условные обозначения, используемые в руководстве к выполнению лабораторной работой:

Исходный код:

```
const express = require('express')
const app = express()
```

Главным методом пояснения кода являются комментарии. Они не важны для исходного текста программы. Огромная просьба — **комментарии не списывать**!

Блоки «На заметку!»

Иногда на протяжении вашей работы вы увидите вот такие блоки, содержащие полезную информацию, а иногда сообщающие важные сведения, без которых понимание вами темы будет не полным. **Не игнорируйте их!**

Инициализация проекта

Для начала — создадим файл `package.json`. В этом файле будет храниться информация о проекте. В прошлый раз мы пропустили некоторые моменты. Например, не указывали лицензию проекта, а так же не забыли указать автора. Перейдите `cd C:\Projects\weblog` и выполните команду `npm init`.

На запрос `Author:` введите свое имя (лучше на английском), а затем при запросе `License:` укажите тип лицензии `MIT`.

Немного о лицензии MIT

Лицензия открытого программного обеспечения, разработанная Массачусетским технологическим институтом. Лицензия MIT является одной из самых ранних свободных лицензий, так как она относительно проста и иллюстрирует некоторые из основных принципов свободного лицензирования. Она является разрешительной лицензией, то есть позволяет программистам использовать лицензируемый код в закрытом программным обеспечением при условии, что текст лицензии предоставляется вместе с этим программным обеспечением. (Википедия)

Если вы выбрали в качестве редактора Sublime Text 3, Atom, или VS Code, в вашей системе можно открыть текущую директорию как корень проекта прямо из командной строки Windows:

```
# для Atom:
atom .
# Для Sublime Text 3:
subl .
# Для VS Code:
code .
```

Либо просто перетащите каталог `weblog` в окно редактора. Так работать с проектом станет намного удобнее!

Редактор — ваш верный боевой товарищ!

Неважно, какой редактор вы выбрали: будь то Atom, Sublime Text или VS Code — каждый редактор это просто редактор, пока вы не освоили его сочетания клавиш, поведение и не настроили его под свои нужды. На сайте каждого редактора есть справка по таким вопросам. Особенно прошу уделить внимание такой иногда необходимой вещи как множественный курсор.

В будущем нам потребуется подсветка синтаксиса для Handlebars. В Atom и Code уже есть поддержка данного языка шаблонов. Но в Sublime ее нет. Выхода у вас 2: поставить плагин Package Control и установить пакет Handlebars, либо при открытии файла `.hbs` переопределить режим подсветки на «HTML»

Ваш файл `package.json` теперь выглядит примерно так:

```
{
    "name": "weblog",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Ivan Khromov",
    "license": "MIT"
}
```

Настало время установить все необходимые пакеты. В командной строке выполните команду в той же директории:

```
npm i -S express morgan hbs
```

При установке могут возникнуть ошибки, когда у директории проекта стоит флаг «только для чтения». Думаю, вы справитесь с проверкой директории на наличие этого флага.

Возможно, вы ошиблись в написании имен пакетов. Проверьте внимательно. Как показывает практика, вы сможете перепутать имена пакетов `hbs` и `hds` — оба пакета есть в репозитории NPM, и они установятся без проблем.

Теперь создайте входную точку нашего приложения — файл **index.js**. В него мы поместим начальное содержимое. Это будет минимальное приложение на Express:

```
// подключаем Express
const express = require('express')
// Подключим Morgan
const morgan = require('morgan')
// Создаем экземпляр Express, с которым и будем работать
const app = express()
// зададим порт для нашего приложения
// будет доступно на 127.0.0.1:5757
let port = 5757

// укажем, что Express должен использовать Morgan,
// а также тип логирования обычный, нам подробный ни к чему =)
app.use(morgan('tiny'))

// зададим систему визуализации Handlebars
app.set('view engine', 'hbs')
// укажем, где Handlebars смотреть файлы представлений
app.set('views', __dirname + '/views')
// и опции. укажем, где у нас макет приложения
app.set('view options', { layout: 'layout/main' })

// зададим главный путь (route)
app.get('/', function(req, res) {
    res.render('homepage')
})

// и начинаем "слушать" входящие соединения и запросы:
app.listen(port, () => console.log(`~~ Запустились на порту ${port}`))
```

Теперь настала пора создать пару макетов. Для начала — главный макет приложения.

Создайте папку **views**, в ней папку **layout**. В нее поместите файл `main.hbs`:

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Weblog</title>
    </head>
    <body>
    {{{body}}}
    </body>
</html>
```

Теперь в папку **views** добавьте файл `mainpage.hbs`. Вот его *примерное* содержимое:

```html
<h1>Работает!</h1>
<a href="/blog">Посмотрите на мой блог &rsaquo;</a>
```

Теперь запустите наше приложение командой `node index.js`. Обратите внимание на вывод в вашей командной строке:

```
Запустились на порту 5757
GET / 200 227 - 149.402 ms
GET /favicon.ico 404 150 - 2.439 ms
GET /blog 404 143 - 1.709 ms
```

Это morgan. Он записывает в консоль результаты наших запросов. Это очень полезное middleware, так как помогает разработчику понять, какие запросы были выполнены.

Первое слово — это глагол запроса HTTP. В нашем случае — `GET`. Затем идет путь, по которому произошел запрос. Например - / (корень сайта, или главная страница). После этого идет длина контента в байтах, их у главной страницы 227. Ну а потом идет количество милисекунд, затраченных на генерацию и загрузку страницы

Почему `favicon.ico` — 404?

Дело в том, что браузер, когда вы загружаете страницу создает еще один независящий от вас запрос — он запрашивает фавикон сайта, который обычно находится по адресу `/favicon.ico`. Мы можем явно указать ему, где он находится при помощи meta-тега. Но если его нет, браузер все равно сделает этот запрос и, в нашем случае, получит от нашего веб-сервера ответный код «404 - Не найдено»

Nodemon

До этого момента мы нажимали сочетание клавиш Ctrl+C чтобы выйти из слушателя сервера и завершить работу программы. Конечно, многим это может надоесть. Особенно, когда перезагружать сервер придется очень часто. Данную проблему решает Nodemon — консольная утилита, которая, заметив изменения в вашем коде автоматически перезапустит его.

Установить ее очень просто. До этого мы устанавливали пакеты в папку `node_modules`, теперь установим ее глобально. Воспользуемся аргументов `-g`:

```
npm i -g nodemon
```

После этого проверьте, существует ли эта команда в системе:

```
nodemon --version
```

Вы должны получить ответ, например `1.14.11` — это версия Nodemon.

Давайте теперь сделаем изменения в файле `package.json`. После строки №6 (фигурная скобка, секция `scripts`) напишите:

```
"dev": "nodemon -e js,hbs,css index.js",
```

Хорошо. Теперь запуск нашего проекта будет осуществляться командой `npm run dev`. Согласитесь, это меньше, чем команда, описанная нами чуть выше. Обратите внимание, что теперь вывод нашего приложения будет немного другой. Сначала в свои руки инициативу возьмет NPM, а затем передаст эстафету Nodemon.

Nodemon в свою очередь, будет бесконечно наблюдать за изменениями в наших файлах. Только помните, он не так быстро работает, особенно на слабом железе. Ему нужно ~4 секунды на перезапуск программы.

Аргумент `-e` указывает через запятую список файлов, за изменениями в которых нужно следить. Мы укажем типичные для нашего проекта файлы:

- `js` — файлы исходного кода JavaScript
- `hbs` — файлы шаблонов Handlebars
- `css` — файлы каскадных таблиц стилей

Как правильно завершить Nodemon в консоли?

Все просто - нажмите Ctrl + C, а затем введите английскую «y» и Enter

Просто оставьте окно командной строки с запущенным Nodemon. Откройте новую командную строку в рабочей директории (директория проекта)

Подключение Mongoose

Прежде чем начать понимать, что такое Mongoose, вспомним немного про Microsoft® Entity Framework. Как вы помните, он являетя объектно-связующим, работающим по принципу объект-таблица. Называется эта методология и философия ORM — *object relation mapper*. Как понятно из аббревиатуры и из предыдущих лекций — он переносит сущности (*entity*) на языке C# в реальные объекты БД — таблицы и связи.

С MongoDB все иначе. Здесь не строки, а документы, не таблицы, а коллекции. Здесь нельзя применить стандартную ORM — она просто не нужна. А что если представить объект коллекции как сущность? Это абсолютно правильная позиция. Да, иногда некоторые элементы коллекции можно опустить, сделав их необязательными именно на уровне сущности. И тут нам на помощь приходит ODM.

Что такое ODM? Это объектно-документная связь (*object document mapper*). И нет, она как раз таки не работает со связью на уровне DBMS. Наоборот, она берет на себя проверку данных на уровне документа, чтобы он отвечал тем правилам, которые в него заложил разработчик. Тут на сцену и выходит Mongoose.js — библиотека для работы с MongoDB.

Чтобы добавить Mongoose в проект, обратимся к командной строке и введем следующую команду:

```
npm i -S mongoose
```

NPM экономит ваше время

Node Package Manager (NPM) не только умеет устанавливать пакеты, запускать скрипты и следить за чистотой кода. У него есть куча сокращений. Мы только что воспользовались одним из них — `npm i` это сокращение от `npm install`, а аргумент `-S` сокращение от `--save`

Теперь необходимо внести изменения в наш файл приложения. Сразу после строки импорта Express напишем:

```
// подключаем mongoose:
const mongoose = require('mongoose')
```

И после подключения morgan напишем код, который будет соединять нас с Mongo:

```
mongoose.connect("mongodb://localhost/weblog")
    .then(() => console.log('~~ Подключились к MongoDB!'))
    .catch((err) => console.log(err.message))
```

Здесь все довольно просто: вызываем функцию `connect`, а она возвращает объект Promise. Это довольно комплексная тема, поэтому мы, возможно, посвятим ей один урок в будущем. Пока что просто запомните — `then` выполняется при успешном подключении к MongoDB, а `catch` ловит ошибку и отображает ее в консоли.

На этом подключение Mongoose закончено. Давайте начнем ее использовать!

Модели данных

Для моделей данных выделим целую папку, и назовем ее `models`. Создайте её. Теперь нам нужно определиться с тем, как и что должны хранить наши модели.

Текущая задача требует от нас создать всего одну модель - `post.js`. В ней мы определим 3 поля - заголовок, содержимое и дата публикации.

Mongoose позволяет использовать нативные (встроенные) типы JavaScript как типы данных в MongoDB. На данный момент нам понадобится всего 2 типа: `String` и `Date`.

models/post.js

Определим модель для записи в блоге. Мы уже определились с данными, которые будем описывать.

В Mongoose создается сразу 2 сущности — схема и модель. Мы будем в дальнейшем работать с моделью, но схема описывает поля в нашей базе и непосредственно связана с моделью.

Создайте файл `post.js` в папке `models` и наполните его содержимым:

```
// подключим Mongoose
const mongoose = require('mongoose')
// и создадим экземпляр класса Schema
const Schema = mongoose.Schema

// определим схему данных
let postSchema = new Schema({
    title: String,
    content: String,
    published: Date
})
// и зададим модель
let Post = mongoose.model('Post', postSchema)
// и немного магии Node.js (объяснение ниже)
module.exports = Post
```

Сохраните этот файл и можете закрыть. Больше он нам не понадобится

`module.exports` — магия Node.js #1

Все, что мы написали можно назвать модулем. Именно благодаря им мы не засоряем код а можем разделить его на несколько папок и файлов. Мы как бы экспортируем какой-то объект, который можно потом получить при помощи `require('filename')`. Об этом мы поговорим далее.

`module.exports` здесь делает доступным подключение этого файла как модуля. Мы вскоре сделаем это в файле `index.js`

Создание и наполнение базы данных

Для начала, при помощи RoboMongo (Robo 3T) создайте базу данных **weblog**. В ней создайте коллекцию `posts` и добавьте туда несколько записей следующим образом:

```
db.posts.insert({
    title: "Запись в блоге №1",
    content: "<напишите тут что-нибудь>",
    published: ISODate("2018-01-30")
})
```

Добавьте так как минимум 10 записей, для наглядности.

Выборка записей. Путь /blog

Для начала после `let port = ...` в `index.js` добавьте:

```
const Post = require('./models/post')
```

Это подключит нашу модель. Мы как раз ее и экспортировали в предыдущих шагах, чтобы она была доступна через `require()`

Давайте создадим еще один путь (*route*) для нашего приложения. Мы уже создали главную страницу, давайте таким же образом создадим путь для блога:

```
// создаем путь
app.get('/blog', function(req, res) {
    // вызовем find - метод поиска по записям в БД
    // второй аргумент - переменная, куда будут записаны все записи
    Post.find({}, function(err, posts) {
        // если ошибка - останавливаемся и выводим ее в консоль
        if(err) throw err
        // если нет - визуализируем шаблон и передадим туда posts
        res.render('blog/index', { posts: posts})
    })
})
```

Создайте в папке `views` папку `blog`. Туда поместите файл `index.hbs`:

```
{{#each posts}}
<a href="/blog/post/{{id}}"><h3>{{title}}</h3></a>
{{/each}}
```

После этого в `index.js` добавим путь для просмотра записи:

```
// :id - шаблон для строки запроса. Мы сможем назвать его как угодно
// главное - он должен следовать синтаксису JS.
app.get('/blog/post/:id', function(req, res) {
    // для нахождения ТОЛЬКО ОДНОЙ записи, воспользуемся findOne:
    Post.findOne({ _id: req.params.id}, function(err, post) {
        if(err) throw err
        res.render('blog/post', { post: post })
    })
})
```

И создадим для него шаблон. Создайте файл `views/blog/post.hbs`:

```
<h1>{{post.title}}</h1>
<p>{{post.content}}</p>
<small>{{post.published}}</small>
```

Проверьте работоспособность приложения. Покажите результат преподавателю.

```
const express = require('express')
const mongoose = require('mongoose')
const morgan = require('morgan');
const app = express()
let port = 5757

const Post = require('./models/post')

app.use(morgan('tiny'))

mongoose.connect("mongodb://10.2.8.115/weblog")
    .then(() => console.log('~~ Подключились к MongoDB!'))
    .catch((err) => console.log(err.message))

app.set('view engine', 'hbs')
app.set('views', __dirname + '/views')
app.set('view options', { layout: 'layout/main' })

app.get('/', function(req, res) {
    res.render('homepage')
})

app.get('/blog', function(req, res) {
    Post.find({}, function(err, posts) {
        if(err) throw err
        res.render('blog/index', { posts: posts})
    })
})

app.get('/blog/post/:id', function(req, res) {
    Post.findOne({ _id: req.params.id}, function(err, post) {
        if(err) throw err
        res.render('blog/post', { post: post })
    })
})

app.listen(port, () => console.log(`~~ Запустились на порту ${port}`))
```