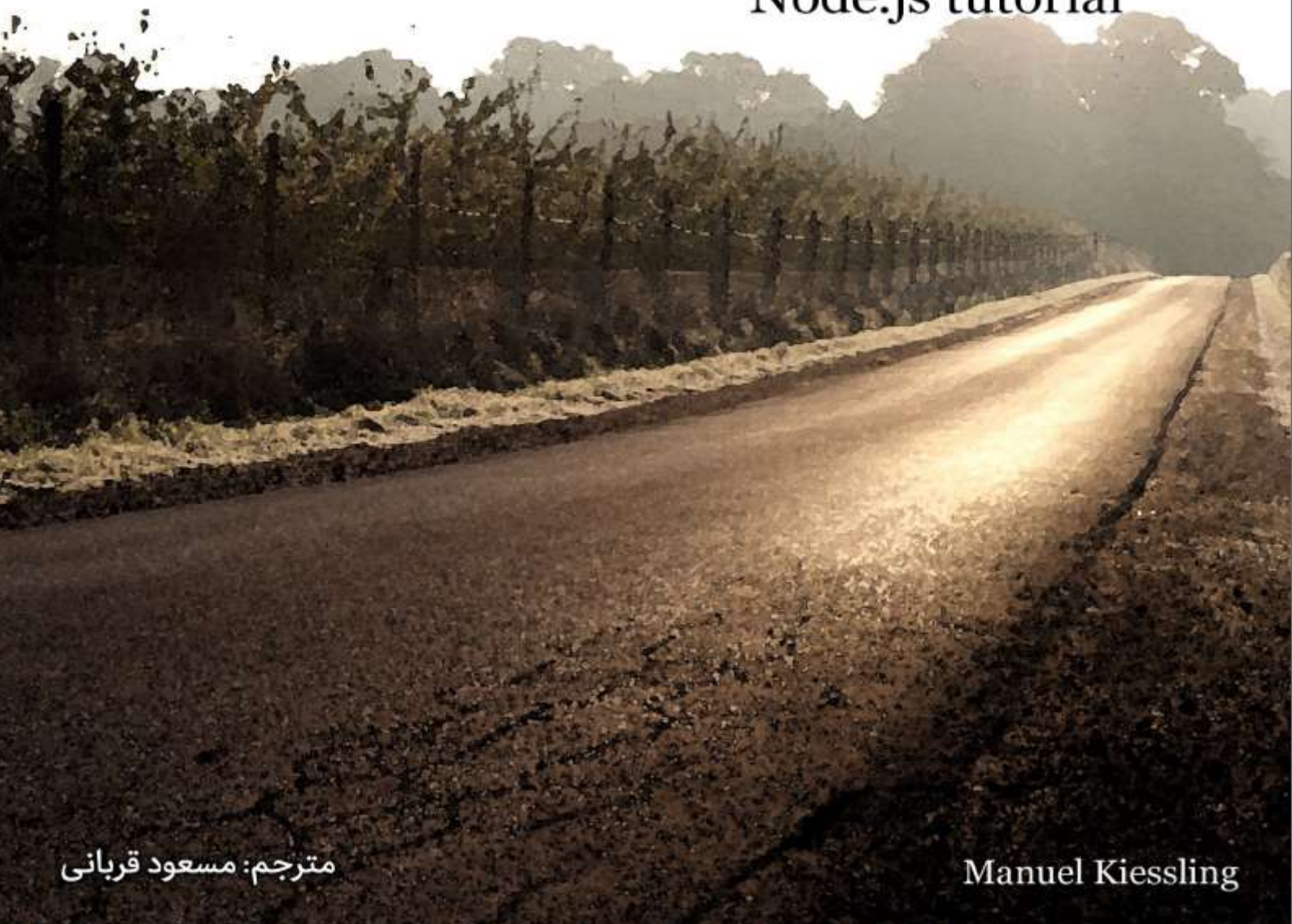
Leanpub

The Node Beginner Book

A comprehensive Node.js tutorial



مترجم: مسعود قربانی

Manuel Kiessling

Node برای مبتدی‌ها

آموزش جامع Node.js

هدف کلی این کتاب آشنا کردن شما با **توسعه نرم‌افزار به‌وسیله Node.js** و فراگیری مباحث مهم برنامه‌نویسی **پیشرفته جاوااسکریپت** در یک مسیر ۴۳ صفحه‌ای است.

دریافت سورس برنامه و نسخه pdf. از طریق:

- http://msud.ir/nbpersian

برای همکاری محتوای ترجمه، پیشنهادها و رفع اشکال‌ها می‌توانید در Github دنبال کنید:

- https://github.com/imasood/nbpersian

این کتاب را در توییتر به اشتراک بگذارید!

برای کمک به من و گسترش این کتاب می‌توانید آن را در توییتر یا شبکه‌های اجتماعی دیگر به اشتراک بگذارید.

هشتگ پیشنهادی برای این کتاب: #nodebeginner

با رجوع به لینک زیر می‌توانید اقدام به جستجو هشتگ فوق کنید و گفته‌های دیگران را پیدا کنید:

- https://twitter.com/hashtag/nodebeginner?src=hash

مجوز کتاب

ترجمه فارسی این کتاب تحت مجوز بین‌المللی کریتیو کامنز بوده(CC BY-NC-ND 4.0) و می‌توانید به رایگان انتشار دهید. به یاد داشته باشید نسخه انگلیسی و موجود در Leanpub تحت مجوز فوق نمی‌باشد.

فهرست مطالب

نکته ویژه

خواننده محترم، این کتاب به‌منظور یادگیری آغازین شما برای توسعه نرم‌افزار به‌وسیله Node.js است، نه کمتر و نه بیشتر.

بسیاری از خوانندگان می‌پرسند که پس از مطالعه این کتاب چه باید کنند که در پاسخ باید گفت به مباحثی همچون مدیریت پایگاه‌های داده، کار با فریم‌ورک‌ها، کار با واحدهای آزمایش(Unit Test) و مطالب بیشتر بپردازند.

به‌تازگی بر روی کتاب مشابه ای کار کرده‌ام با عنوان The Node Craftsman Book که هم‌اکنون بر روی وب‌سایت انتشارات Leanpub موجود است و به‌صورتبه‌صورت کلی موضوعات زیر را شامل می‌شود:

- اصول پایه‌ای Node.js
- کار با NPM و بسته‌ها(Packages)
- توسعه به‌وسیله Node.js بر پایه برنامه‌نویسی آزمون محور(Test-Driven)
- برنامه‌نویسی شیءگرا در جاوااسکریپت(OOP)
- توسعه به کمک AngularJs
- کار با پایگاه‌های داده

اگر مایل به دریافت کتاب The Node Craftsman Book هستید می‌توانید به لینک زیر رجوع کنید:

http://leanpub.com/nodecraftsman

پیشگفتار

هدف کلی این آموزش آشنا کردن شما با توسعه نرم‌افزار به‌وسیله Node.js است، در طول این آموزش، مباحث موردنیاز برای درک برنامه‌نویسی پیشرفته "جاوااسکریپت" را فرا خواهید گرفت و این آموزش فراتر از مثال "Hello World" خواهد بود.

در حال حاضر شما در حال مطالعه آخرین نسخه از این کتاب هستید و بروز رسانی‌ها شامل مواردی همچون رفع خطاها یا تغییرات جزئی خواهند بود که آخرین بروز رسانی به تاریخ ۱۰ اکتبر، ۲۰۱۵ باز می‌گردد.

نمونه کدهای موجود در این کتاب بر پایه نسخه 0.10.12 از Node.js نوشته و آزمایش شده است.

مخاطبان

این کتاب مسلماً مناسب خوانندگی خواهد بود که پس‌زمینه‌ای نسبت به موضوعات زیر داشته باشند: حداقل آشنایی با یکی از زبان‌های شیءگرا همچون Ruby، Python، PHP یا Java. آشنایی کوتاه با جاوااسکریپت و Node.js

کتاب، متمرکز بر توسعه‌دهندگانی است که با مباحث نوع داده(Data Type)، متغیرها، ساختارهای کنترلی و موارد دیگر آشنایی دارند که کتاب شامل این موضوعات نیست. در حال حاضر برای درک بهتر این کتاب به یادگیری موارد فوق نیاز دارید.

هرچند مباحثی همانند توابع و شیءها در جاوااسکریپت با دیگر زبان‌ها متفاوت است اما در این کتاب به توضیح بیشتر و عمیق‌تر آن‌ها خواهیم پرداخت.

ساختار کتاب

پس از مطالعه این کتاب، شما می‌بایست یک برنامه تحت وب ساخته باشید که به کاربران اجازه خواهد داد صفحات را مشاهده کنند و اقدام به آپلود فایل کنند.

البته باید در نظر داشته باشید که این مثال دنیا را تغییر نخواهد داد اما به ما کمک می‌کند که قدم بلندی برداریم و نمی‌خواهیم برنامه‌ای بسازیم که صرفاً در انتها بگوییم به‌اندازه کافی خوب است و به اهداف موردنظر رسیده‌ایم. ساخت این برنامه ساده اما در این حال کامل به ما جنبه‌های مختلف یک برنامه پیشرفته در Node.js را نشان خواهد داد.

ما به دنبال چگونگی توسعه جاوااسکریپت در Node.js هستیم و این فرق دارد با توسعه جاوااسکریپت در مرورگرها.

ابتدا با سُنَت قدیمی نوشتن برنامه "Hello World" شروع خواهیم کرد که یکی از ساده‌ترین برنامه‌های Node.js است.

سپس درباره ساخت یک برنامه واقعی صحبت خواهیم کرد که قصد ساخت آن را داریم. این موضوع به‌صورت قدم‌به‌قدم پیش خواهد رفت و این نیاز وجود دارد که قسمت‌های مختلف پیاده‌سازی برنامه را تشریح کرد.

همان‌طور که بیان شد، در مسیر این آموزش با بعضی از مباحث پیشرفته جاوااسکریپت آشنا خواهید شد که از آن‌ها استفاده خواهید کرد و این حس را خواهد داد که چرا باید بجای تکنیک‌هایی در دیگر زبان‌ها از این تکنیک‌ها استفاده کنیم.

سخن مترجم

به جرأت باید گفت یکی از انقلابی‌ترین گام‌های روبه‌جلو در حوزه وب ظهور **Node.js** بوده که روزبه‌روز بر محبوبیت این فناوری افزوده می‌شود. هدف از ترجمه این کتاب صرفاً تلاشی صادقانه و ناچیز برای پیشبرد دانش در حوزه وب فارسی بوده است، افزون بر این در ساختار ترجمه تمامی تلاش بر این بوده که ترجمه روان باشد و خواننده به‌راحتی بتواند مفاهیم را درک کند.

باید در نظر داشت که فقط این نسخه از کتاب(ترجمه فارسی) به‌صورت **رایگان** و با کسب اجازه از نویسنده کتاب انتشار یافته است.

جاوااسکریپت و Node.js

جاوااسکریپت و شما

قبل از پرداختن به مباحث فنی، باید به رابطه شما و جاوااسکریپت اشاره کرد. در این فصل به شما کمک خواهیم کرد که میزان رابطه خود با جاوااسکریپت را برآورد کنید.

اگر شما نیز همانند من دنیای وب را سال‌ها پیش با "HTML" شروع کرده‌اید مطمئناً با کلمه جذاب جاوااسکریپت آشنا هستید، اما احتمالاً جاوااسکریپت را صرفاً برای ایجاد تعامل بهتر کاربران با صفحات وب استفاده کرده‌اید.

باید پرسید خواسته حقیقی شما برای ساخت "یک برنامه واقعی" چیست، شما می‌خواهید بدانید که چگونه می‌توان یک برنامه پیچیده پیاده‌سازی کرد، شما یاد خواهید گرفت که از ابتدا همچون زبان‌های PHP، Ruby، Java در Node.js برنامه‌نویسی کنید.

با این اوصاف، همچنان نیاز دارید که یک چشمتان بر روی جاوااسکریپت باشد، مطمئناً با jQuery و مباحثی همچون Prototype آشنا هستید. باید دقت کرد که جاوااسکریپت واقعاً سرزمینی بزرگ و پیشرفته‌ای است و فقط به دستور *window.open()* ختم نمی‌شود.

هرچند تا به اینجا به قسمت Front-End پرداخته شد، اما درنهایت با jQuery هم قادر بودیم موارد زیبایی را پیاده‌سازی کنیم ولی باید گفت فقط به‌عنوان یک استفاده‌کننده جاوااسکریپت نه به‌عنوان توسعه‌دهنده جاوااسکریپت(JavaScript Developer). حال Node.js آمده است، می‌خواهیم از جاوااسکریپت در سمت سرور استفاده کنیم. چقدر می‌تواند عالی باشد؟

شما تصمیم گرفته‌اید جاوااسکریپت قدیمی را فراموش کنید(استفاده در مرورگر) و قسمت جدید آن را یاد بگیرید. اما صبر کنید، برنامه‌نویسی در Node.js فقط یک‌چیز است: چرا توسعه‌دهندگان Node.js این سبک را ابداع کرده‌اند و چرا ما نیاز داریم جاوااسکریپت را درک کنیم. واقعیت همین است.

نکته جالب: ازآنجاکه جاوااسکریپت دو بار زنده شده است احتمالاً این سومی است ☺ (کمک DHTML در اواسط دهه ۹۰، به وجود آمدن jQuery و حال Node در سمت سرور)، در مسیر نوشتن برنامه‌های Node.js حس نخواهید کرد که در حال استفاده از جاوااسکریپت هستید اما در حقیقت یاد خواهید گرفت که آن را توسعه دهید.

در حال حاضر شما یک توسعه‌دهنده باتجربه هستید و نمی‌خواهید تکنیک جدیدی را یاد بگیرید که در مسیر غلط(Hacking) از آن استفاده کنید، می‌خواهید به‌صورت کامل از آن در مسیر صحیح استفاده کنید.

مطمئناً مستندات عالی‌تری بیرون از این کتاب وجود دارد اما مستندات بعضی مواقع به‌تنهایی کافی نیست. هدف از تألیف این کتاب تنها فراهم شدن یک راهنمایی مناسب برای مبتدی‌ها است.

اخطار

واقعاً افرادی هستند که جاوااسکریپت را خیلی عالی می‌داند، که من یکی از آن‌ها نیستم.

من درنهایت به فردی خلاصه می‌شوم که فقط پاراگراف قبلی را توضیح داده‌ام اما مواردی را در مورد توسعه برنامه‌های تحت وب می‌دانم. لازم به ذکر است که در زمینه جاوااسکریپت و Node.js واقعاً یک تازه‌کار هستم و اخیراً جنبه‌های پیشرفته جاوااسکریپت را فراگرفته‌ام و یک فرد مجرب در این زمینه نیستم.

به همین دلیل است که هیچ کتابی با عنوان "از مبتدی به متخصص" وجود ندارد. این کتاب بیشتر به "از مبتدی به مبتدی پیشرفته" خواهد پرداخت.

اگر من با شکست مواجهه نشوم، این کتاب می‌تواند همان کتابی باشد که آرزو داشتم زمانی که Node.js را شروع کرده بودم آن را می‌داشتم.

جاوااسکریپت در سمت سرور

اولین برداشت شما از جاوااسکریپت استفاده آن در مرورگر است اما این فقط یک قرارداد ساده می‌باشد.

تعریف فوق اشاره می‌کند که شما چه کاری می‌توانید با زبان انجام دهید اما این تعریف درباره اینکه خود زبان چه کاری می‌تواند انجام دهد را مشخص نمی‌کند.

Node.js حقیقتاً یک موضوع دیگری است: به شما این اجازه را خواهد داد که جاوااسکریپت را در Backend اجرا کنید بدون نیاز به مرورگر.

بدین منظور باید در Backend تفسیر و اجرا صحیحی داشته باشیم. حال Node.js چگونه عمل می‌کند؟

Node.js به‌وسیله موتور جاوااسکریپت V8 که توسط شرکت Google توسعه‌یافته است می‌تواند یک محیط در زمان اجرا(Runtime) برای جاوااسکریپت را ایجاد کند همانند مرورگر Google Chrome که از آن استفاده می‌کنید.

به‌علاوه Node.js دارای ماژول‌های مفید زیادی است، بنابراین شما نیاز ندارید همه چیز را از اول بنویسید.

درنهایت باید گفت Node.js واقعاً دو چیز است: یک محیط در زمان اجرا و یک کتابخانه.

به‌منظور استفاده، نیاز به نصب و آماده‌سازی Node.js دارید. برای این منظور می‌توانید از مستندات رسمیNode.js استفاده کنید و درصورتی‌که به زبان انگلیسی تسلط ندارید می‌توانید به وبلاگ جامعه Node.js فارسی رجوع کنید که در زیر اشاره شده است:

- وب سایت رسمیNode.js
- Node فارسی
- نصب و آماده‌سازی محیط توسعه Node.js

برنامه "سلام دنیا"

خیلی خوب، بیاید اولین برنامه سنتی و قدیمی"سلام دنیا" را در Node.js بنویسیم.

ویرایشگر موردعلاقه خودتان را باز کنید و یک سند با عنوان *helloworld.js* ایجاد کنید. می‌خواهیم به‌وسیله این برنامه یک رشته با عنوان سلام دنیا را نمایش دهیم. برای این منظور دستور زیر را بنویسید:

```
1 | console.log("Hello World");
```

فایل را ذخیره کنید.

ابتدا خط فرمان(ترمینال) سیستم‌عامل خود را درجایی که فایل ذخیره شده است باز کنید و سپس با دستور زیر به‌وسیله Node.js برنامه را اجرا کنید:

```
1 | node helloworld.js
```

در خروجی خط فرمان باید Hello World نوشته شده باشد.

این مثال خیلی جالب نیست، درسته؟ در فصل بعد به تشریح یک برنامه پیچیده‌تر خواهیم پرداخت.

یک برنامه کامل تحت وب به کمک Node.js

اهداف برنامه

قصد ساخت یک برنامه ساده اما کامل با ویژگی‌هاویژگی‌های زیر را داریم:

- کاربر باید قادر باشد از برنامه ما داخل مرورگر استفاده کند.
- کاربر باید در آدرس http://domain/start یک صفحه خوش‌آمد گویی مشاهده کند که شامل یک فرم آپلود تصویر است.
- کاربر به‌وسیله فرم قادر به آپلود تصویر باشد و این تصویر با آدرس http://domain/upload آپلود خواهد شد و نمایش داده می‌شود.

ویژگی‌ها به‌اندازه کافی هستند، با جستجو کردن ممکن است در اینترنت به این هدف برسید اما آن چیزی نیست که ما می‌خواهیم اینجا انجام دهیم.

از این گذشته، نمی‌خواهیم با نوشتن چند خط کد ساده به هدف اصلی برسیم و هرچند ممکن است کدهایی که می‌نویسیم زیبا و درست باشند اما قصد داریم به برنامه لایه‌های بیشتری اضافه کنیم که منجر خواهد شد حس کنید در حال ساخت یک برنامه پیچیده Node.js هستید.

تشریح ساختار برنامه

بیاید ساختار برنامه را تشریح کنیم، بخش‌هایی که برای پیاده‌سازی برنامه نیاز داریم به تفکیک زیر شرح داده شده است:

- می‌خواهیم برنامه بر بستر وب و در مرورگر اجرا شود، بنابراین به یک HTTP سرور نیاز داریم.
- برنامه ما ممکن است به درخواست‌هایی مختلفی پاسخ دهد که بستگی به آدرس‌هایی دارد که تقاضا می‌شود، بدین ترتیب به یک مسیریاب نیاز خواهیم داشت که بتواند کنترلر یک درخواست را بیابد.
- درخواست‌هایی که به سرور وارد می‌شوند نیاز به مسیریابی دارند و درنهایت باید مدیریت شوند، بنابراین باید برای درخواست‌ها کنترلر تعریف کنیم. (Request Handlers)
- مسیریاب باید برای مدیریت درخواست‌های POST یک راه‌حل داشته باشد چراکهچراکه نیاز داریم اطلاعات را از آن بگیریم. (Request Data Handling)
- فقط نمی‌خواهیم درخواست‌ها را مدیریت کنیم، همچنین باید پاسخ‌هایی را برای آن‌ها در نظر بگیریم، به عبارت دیگر نیاز داریم یک بخش نمایش(View) داشته باشیم که محتوا را بر بستر مرورگر نمایش دهد.
- در آخر، کاربر باید قادر باشد فایل موردنظر خود را آپلود کند که نیاز داریم فایل های آپلودی را مدیریت کنیم. (Upload Handling)

چند لحظه فکر کنید که پیاده‌سازی این برنامه با **PHP** چگونه خواهد بود. مسلماً مثل رازی نیست که نیاز به تنظیم خاصی داشته باشد ممکن است فقط نیاز داشته باشیم که ماژول ***mod_php5*** از آپاچی سرور را نصب کنیم.

بسیار خوب، این برنامه با Node مقداری متفاوت‌تر خواهد بود چراکه فقط قصد پیاده‌سازی برنامه را نداریم در حقیقت می‌خواهیم یک **HTTP سرور** را به همراه برنامه پیاده‌سازی کنیم.

شاید فکر کنید کار سختی باشد، اما می‌بینید که در یک لحظه با Node آن را پیاده خواهیم کرد. قبول ندارید؟

در فصل بعد، از ابتدا به پیاده‌سازی یک HTTP سرور خواهیم پرداخت.

پیاده‌سازی برنامه

ایجاد یک HTTP سرور

در نقطه‌ای قرار داریم که قرار است اولین برنامه واقعی در محیط Node را آغاز کنیم، از این تعجب نکنیم که قرار است چگونه برنامه را پیاده‌سازی کنیم همچنین سازمان‌دهی کدها نیز مهم است.

باید همه‌ی کدها را داخل یک فایل بنویسیم؟ آموزش‌های موجود در سطح وب به شما آموزش می‌دهند که چطور یک HTTP سرور ساده در Node بنویسید که اکثر آن‌ها تمامی کدها را در یک فایل می‌نویسند. اگر بخواهیم برنامه‌ای خوانا، دسترس‌پذیر و زیباتر داشته باشیم چه باید کرد؟

مشخصاً این مسئله ساده است، تنها کافی است تفسیر درستی از برنامه داشته باشیم و کدها را سازمان‌دهی کنیم به‌گونه‌ای که قسمت‌های مختلف برنامه را در ماژول‌های مختلف قرار دهیم.

این سبک به ما کمک می‌کند که **فایل اصلی(Main File)** برنامه تمیزتر و خواناتر باشد نسبت به اینکه تمامی قطعات برنامه را داخل یک فایل قرار دهیم و به‌راحتی می‌توانیم ماژول‌های مختلف را در فایل اصلی فراخوانی کنیم.

بنابراین ابتدا باید فایل اصلی برای برنامه را ایجاد کنیم و ماژول HTTP سرور را در آن فراخوانی کنیم.

یکی از استانداردترین نام‌ها در دنیای برنامه‌نویسی عنوان ***index*** است که ما هم از این استاندارد برای فایل اصلی استفاده خواهیم کرد - *Index.js*

برای آنکه HTTP سرور را بسازیم، آن را در ماژولی به نام ***server.js*** پیاده‌سازی می‌کنیم. در ریشه پروژه فایلی به نام ***server.js*** ایجاد کنید و کد زیر را در آن قرار دهید:

```
1  var http = require("http");
2
3  http.createServer(function(request, response) {
4     response.writeHead(200, {"Content-Type": "text/plain"});
5     response.write("Hello World");
6     response.end();
7  }).listen(8888);
```

بسیار خوب، فقط کافی است برنامه را اجرا کنید. خط فرمان را در ریشه پروژه باز کنید و دستور زیر را وارد کنید:

```
1  node server.js
```

حال مرورگر را باز کنید و آدرس http://localhost:8888/ را وارد کنید. این آدرس باید یک صفحه وب را به شما نشان دهد که درون آن عبارت Hello World ذکر شده است.

دقیقاً در یک‌لحظه HTTP سرور ساخته شد و این بسیار جالب است، در قسمت بعد به تشریح قطعه کد بالا خواهیم پرداخت و کدهای برنامه را سازمان‌دهی می‌کنیم.

آنالیز HTTP سرور

بسیار خوب، در خط ابتدایی قطعه کد بالا ابتدا با *require* الزام کرده‌ایم که ماژول *http* فراخوانی شود و آن را در متغیر *http* قرار داده‌ایم که به ما کمک می‌کند به HTTP سرور دسترسی داشته باشیم.

سپس یکی از توابع ماژول *http* را صدا زدیم: *createServer* - این تابع یک شیء را به‌عنوان خروجی برمی‌گرداند و دارای متدی با عنوان *listen* است که یک مقداری عددی می‌پذیرد که معادل شماره **پورت** HTTP سرور است و برنامه تا زمانی که در حال اجرا است به این پورت گوش می‌دهد و در صورت وجود هرگونه درخواست به آن پاسخ خواهد داد.

حال کدهای داخل تابع *http.createServer* را نادیده بگیرید به شکل زیر:

```
1  var http = require("http");
2
3  var server = http.createServer();
4  server.listen(8888);
```

اگر قطعه کد بالا را همانند روش قبل اجرا کنید فقط یک HTTP سرور ایجاد کرده‌ایم که به پورت 8888 گوش می‌دهد و دیگر هیچ عملکردی نخواهد داشت.

نکته جالبی در قسمت پارامترهای تابع *createServer* وجود دارد، شاید انتظار داشتید که یک متغیر را به‌عنوان پارامتر عبور دهیم اما مشاهده می‌کنید که یک تابع را به‌عنوان پارامتر برای تابع *createServer* عبور داده‌ایم.

این نوع تعریف تابع همانند *createServer* تنها مختص جاوااسکریپت است. توابع به‌عنوان پارامتر می‌توانند در توابع دیگر مورد استفاده قرار گیرند.

آشنایی با توابع ناهم‌زمان

به عنوان مثال به کد زیر دقت کنید:

```
1  function say(word) {
2     console.log(word);
3  }
4
5  function execute(someFunction, value) {
6     someFunction(value);
7  }
8
9  execute(say, "Hello");
```

قطعه کد بالا را به‌دقت مطالعه کنید! در کد بالا در خط نهم تابع *say* را به‌عنوان اولین پارامتر از تابع *excute* عبور داده‌ایم که منجر به صدازدن تابع *say* در تابع *execute* خواهد شد.

بنابراین، تابع *say* جایگزین متغیر محلی *someFunction* در تابع *execute* خواهد شد و *execute* می‌تواند *someFunction* را صدا بزند.

همان‌طور که مشاهده کردید قادر هستیم یک تابع را با نام خودش به‌عنوان پارامتر از یک تابع دیگر عبور دهیم. حتی می‌توانیم یک تابع را به‌صورت درجا(in-place) استفاده کنیم. برای درک بهتر قطعه کد زیر را بررسی کنید:

```
function execute(someFunction, value) {
   someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

در این روش، حتی دیگر نیاز به گرفتن نام تابع نداریم و این نوع توابع را ناهم‌زمان می‌گوییم.

به‌طورکلی، این اولین قسمت است که مایلم آن را جاوااسکریپت پیشرفته نام‌گذاری کنم. اما قدم‌به‌قدم بیشتر آشنا خواهید شد و فعلاً می‌توانیم بگوییم که یک تابع را به‌عنوان پارامتر عبور داده‌ایم زمانی که تابع دیگری را صدا بزنیم.

چگونگی کار کردن HTTP سرور به کمک توابع ناهم‌زمان

با اطلاعاتی که کسب نموده‌ایم، بگذارید به HTTP سرور خود بازگردیم:

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

http.createServer(function(request, response) {
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("Hello World");
   response.end();
}).listen(8888);
```

در حال حاضر برای ما واضح است که کد بالا چه کاری انجام می‌دهد: استفاده از تابع createServer به‌عنوان یک نوع تابع ناهم‌زمان.

می‌توانیم شکل بهتری را در نظر بگیریم:

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

function onRequest(request, response) {
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("Hello World");
   response.end();
}

http.createServer(onRequest).listen(8888);
```

ممکن است بپرسید چرا این کار را انجام داده‌ایم.

برنامه‌نویسی رویداد محور ناهم‌زمان بر پایه Callbackها

برای درک اینکه چرا Node.js از این روش استفاده می‌کند بهتر است نحوه اجرا شدن کدها توسط Node.js را دریابیم. روش Node.js یک روش انحصاری نیست اما زیربنای مدل‌های آن با محیط‌های در زمان اجرا(Runtime Environments) مثل Python, Ruby, PHP یا Java تفاوت دارد.

به مثال زیر دقت کنید:

```
1  var result = database.query("SELECT * FROM hugetable");
2  console.log("Hello World");
```

لطفاً به مسئله اتصال به پایگاه داده فکر نکنید به فرض از قبل به پایگاه داده متصل بوده‌ایم. در خط اول یک کوئری(query) وجود دارد که سطرهای زیادی از جدول hugetable را فراخوانی می‌کند. در خط دوم عبارت Hello World را چاپ خواهد شد.

فرض کنید حجم کوئری بالا و سرعت بسیار پایین باشد، به‌قدری که به‌سختی می‌تواند چند سطر را بخواند.

در کدی که به روش فوق نوشته شده است ابتدا مفسر جاوااسکریپت نتیجه کوئری را کامل می‌خواند و سپس خط دوم می‌تواند اجرا شود.

اگر واقعاً کد فوق وجود داشته باشد، مطمئناً PHP قادر به اجرای آن است: اجرای کوئری و نمایش عبارت Hello World. اگر این کد به‌عنوان اسکریپت بخشی از یک صفحه وب باشد قاعدتاً کاربر چندین ثانیه منتظر خواهد ماند تا صفحه کامل بارگذاری شود.

هرچند این مدل اجرا در PHP یک مشکل عمومی حساب نمی‌شود و PHP می‌توانید این مثال را پیاده کند اما ممکن است تعدادی از درخواست‌ها ناکامل بمانند و بعضی از کاربران به نتیجه موردنظر نرسند.

مدل اجرا در Node.js متفاوت‌تر است – فقط یک فرآیند در Node.js وجود دارد. اگر یک کوئری سنگین در یک فرآیند وجود داشته باشد، تا زمانی که کوئری به پایان نرسد دیگر فرآیندها اجرا نخواهند شد و به حالت توقف در خواهند آمد.

برای حل این مشکل، راهکار جاوااسکریپت و سپس Node.js برنامه‌نویسی **رویداد محور(Event-Driven) به‌صورت ناهم‌زمان(Asynchronous)** است با استفاده از **حلقه رویداد(Event Loop)**.

برای درک این مفهوم به بررسی کد زیر خواهیم پرداخت:

```
1  database.query("SELECT * FROM hugetable", function(rows) {
2     var result = rows;
3  });
4  console.log("Hello World");
```

در اینجا، بجای آنکه در انتظار باشیم که database.query() نتیجه نهایی را به‌صورت مستقیم برای ما بازگرداند، ما آن را در قالب پارامتر دوم عبور می‌دهیم. حال یک تابع ناهم‌زمان داریم.

در حالت قبلی، کد نوشته شده به شکل هم‌زمان بود: ابتدا کوئری موردنظر اجرا می‌شد و فقط پس از اتمام کامل آن، قادر به نوشتن در خط فرمان بودیم.

حال با روشی که بیان شد Node.js قادر است که کوئری‌های دیتابیس را به‌صورت ناهم‌زمان مدیریت کند و database.query() را به شکل ناهم‌زمان ارائه کند که در این حالت Node.js ابتدا درخواست اجرا را ارسال می‌کند اما بجای آنکه منتظر باشد تا کوئری کامل به پایان برسد یادآوری می‌کند "در هر زمان که کوئری به پایان رسید نتیجه آن را ارسال کن و باید پس‌ازآن تابع ناهم‌زمان را اجرا کند که قبلاً database.query() را از آن عبور داده‌ایم." سپس، بلافاصله console.log() پس‌ازآن اجرا خواهد شد و وارد حلقه رویداد می‌شود. Node.js به‌طور مداوم این چرخه را دنبال خواهد کرد و درنهایت هیچ فرآیندی باقی نخواهند ماند.

موضوع فوق به ما توضیح خواهد داد که چرا HTTP سرور نیاز به یک تابع دارد برای پاسخ دادن به درخواست‌های ورودی – ممکن است Node.js شروع به کار کند و سپس متوقف شود، این حالت ادامه دارد تا زمانی که درخواست بعدی وارد شود که مسلماً این حالت کارآمد نیست. اگر یک کاربر دومی برای سرور، درخواست ارسال کند و در آن زمان سرور مشغول ارائه خدمت به درخواست اول باشد، درخواست دوم باید منتظر باشد تا درخواست اول به پایان برسد – به‌محض اینکه درخواست به تعداد انگشتان دست برسد HTTP سرور پاسخگوی نیاز شما در همه حالت نخواهد بود.

نکته بسیار مهمی وجود دارد که در مدل اجرایی ناهم‌زمانی، تک فرآیندی و رویداد محور یک عملکرد مقیاس‌پذیر خاص وجود ندارد. توانستیم یکی از مدل‌های اجرایی در Node.js را بررسی کنیم و باید گفت محدودیت‌های خاص خود را دارد که می‌توان به تک فرآیندی بودن آن اشاره کرد و این قابلیت فقط یک هسته CPU را در برمی‌گیرد. این مدل کاملاً نزدیک به برنامه است و باوجود شیوه بسیار ساده‌ای که در خود دارد به‌راحتی می‌تواند با شرایط هم‌زمانی مقابله کند.

ممکن است بخواهید تجربه بهتری نسبت به Node.js کسب کنید که می‌توانید مقاله بسیار عالی Felix Geisendoerfer's را در این رابطه مطالعه کنید – Understanding Node.js

اگر مایل باشید با مفهوم فوق بیشتر آشنا شویم. آیا می‌توانیم کد برنامه را طوری پیاده‌سازی کنیم که حتی بعد از ایجاد سرور به کار خود ادامه دهد حتی اگر هیچ درخواست HTTP وجود نداشته باشد و تابع Callback هم صدا نزده شده باشد؟ اجازه دهید امتحان کنیم:

```
1  var http = require("http");
2
3  function onRequest(request, response) {
4     console.log("Request received.");
5     response.writeHead(200, {"Content-Type": "text/plain"});
6     response.write("Hello World");
7     response.end();
8  }
9
10 http.createServer(onRequest).listen(8888);
11
12 console.log("Server has started.");
```

در کد فوق با استفاده از دستور console.log یک عبارت هرزمان که یک درخواست HTTP وجود داشته باشد چاپ می‌شود و درست بعد از شروع به کار HTTP سرور عبارت دیگری چاپ می‌شود.

زمانی که برنامه را اجرا کنیم(node server.js)، بلافاصله عبارت "Server has started." در خط فرمان چاپ خواهد شد علاوه بر این هرگاه درخواستی(به‌وسیله باز شدن آدرس http://localhost:8888/) به سمت سرور ارسال شود عبارت "Request received." در خط فرمان چاپ می‌شود.

تا به اینجا به‌صورت عملی جاوااسکریپتِ رویداد محور ناهم‌زمان به‌وسیله Callbackها را بررسی کردیم.

(نکته: احتمالاً هنگامی‌که درخواستی به سمت سرور ارسال شود در خط فرمان دو بار پیام "Request received." را مشاهده کنید به دلیل اینکه بیشتر مرورگرها سعی دارند که با ارسال هر درخواست اقدام به بارگذاری Favicon در آدرس http://localhost:8888/favicon.ico کنند.)

مسیریابی (۱): چگونگی کنترل درخواست‌ها در سرور

بسیار خوب، اجازه دهید به‌صورت سریع باقی کدهای سرور را آنالیز کنیم - تابع onRequest()

زمانی که تابع Callback یعنی onRequest صدا زده می‌شود، از خود دو پارامتر request و response را عبور می‌دهد.

request و response هردو از نوع شیء هستند و می‌توانیم از متدهای آن‌ها برای کنترل کردن درخواست‌های HTTP که رخ داده‌اند استفاده کنیم و همچنین به درخواست‌ها پاسخ دهیم.

کد ما نیز همانند عبارت فوق عمل می‌کند: هر زمان که یک درخواست دریافت شود به‌وسیله تابع response.writeHead() وضعیت 200 و content-type را برای هدر[۱] HTTP ارسال می‌کند و سپس تابع response.write() عبارت "Hello World" را برای بدنه[۲] HTTP ارسال می‌کند و بر روی صفحه‌نمایش داده می‌دهد. در انتها، تابع response.end() را فراخوانی می‌کنیم که پاسخ دادن به درخواست را پایان می‌دهد.

تا انتهای برنامه شیء request اهمیت خاصی برای برنامه ندارد و به جزئیات آن نمی‌پردازیم.

سازمان‌دهی کدها به کمک ماژول‌ها

در این مرحله به بررسی اینکه چگونه می‌توان برنامه را بهتر سازمان‌دهی کنیم خواهیم پرداخت. در داخل فایل server.js تکه کد بسیار ساده‌ای داریم که مرتبط به HTTP سرور است و در فصول قبل ذکر شد که برنامه دارای یک فایل اصلی است که با عنوان index.js از آن استفاده می‌کنیم که به‌عنوان نقطه شروع برنامه تلقی می‌شود و کمک می‌کند کدها را بهتر سازمان‌دهی کنیم.

اجازه دهید در مورد چگونگی استفاده از server.js به‌عنوان یک ماژول واقعی در Node.js صحبت کنیم که آیا واقعاً می‌توان از آن در فایل اصلی برنامه(index.js) استفاده کرد؟

همان‌طور که می‌دانید در کدها از یک ماژول به شکل زیر استفاده کردیم:

```
1  var http = require("http");
2
3  ...
4
5  http.createServer(...);
```

[۱] HTTP response header

[۲] HTTP response body

اگر برایتان سؤال شده است که ماژول http از کجا می‌آید؟ باید گفت که Node.js در هسته خود ماژول‌هایی را به‌صورت پیش‌فرض دارد و به برنامه‌نویس کمک می‌کند که چرخه برنامه‌نویسی را از ابتدا شروع نکند و با افزودن به کدها می‌توانیم از ماژول‌ها استفاده کنیم به شکلی که آن‌ها را در یک متغیر محلی فراخوانی کنیم.

این باعث می‌شود که متغیرهای محلی به شکل یک شیء استفاده شوند و به‌تمامی متدهای عمومیhttp دسترسی داشته باشند.

می‌توان برای سهولت برنامه‌نویسی از نام ماژول‌ها برای نام‌گذاری متغیرها استفاده کرد اما درهرحال در انتخاب نام متغیر آزاد هستیم، مانند قطعه کد زیر:

```
1  var foo = require("http");
2
3  ...
4
5  foo.createServer(...);
```

بسیار خوب، با نحوه استفاده از ماژول‌های داخلی Node.js آشنا شدید. چگونه می‌توانیم ماژول‌های خود را بسازیم و از آن‌ها استفاده کنیم؟

مشخصاً برنامه ما نباید زیاد تغییر داشته باشد، تبدیل کدهای برنامه به ماژول به این نیاز دارد که کد را به‌گونه‌ای بازنویسی کنیم که بتوانیم از قابلیت‌ها آن خروجی بگیریم و درنهایت ماژول را فراخوانی کنیم.

در حال حاضر، به شکل ساده‌ای نیاز داریم که از قابلیت‌های HTTP سرور استفاده کنیم: اسکریپت‌های ماژول برنامه را الزام کنند که سرور باید شروع به کار کند.

برای این ممکن، قطعه کد زیر را در تابعی با عنوان start قرار می‌دهیم و این تابع را به بیرون صادر می‌کنیم:

```
1  var http = require("http");
2
3  function start() {
4     function onRequest(request, response) {
5        console.log("Request received.");
6        response.writeHead(200, {"Content-Type": "text/plain"});
7        response.write("Hello World");
8        response.end();
9     }
10
11    http.createServer(onRequest).listen(8888);
12    console.log("Server has started.");
13 }
14
15 exports.start = start;
```

حال می‌توانیم فایل اصلی برنامه را با نام index.js ایجاد کنیم و در آن از HTTP سرور استفاده کنیم به‌وسیله کدهایی که هنوز در فایل server.js قرار دارد.

فایلی را با عنوان index.js در کنار فایل server.js ایجاد کنید و کد زیر را در آن قرار دهید:

```
1  var server = require("./server");
2
3  server.start(...);
```

همان‌طور که مشاهده می‌کنید، به‌وسیله فراخوانی فایل server.js و ارجاع دادن ماژول به یک متغیر با نام server از ماژول server همانند یک ماژول داخلی استفاده کرده‌ایم و درنهایت می‌توانیم از توابع آن استفاده کنیم.

همین بود، حال به کمک کد زیر می‌توانیم برنامه را با فایل اصلی آن شروع کنیم:

```
1 | node index.js
```

بسیار عالی، به کمک مفهوم فوق می‌توانیم برنامه را به ماژول‌های مختلف تقسیم‌بندی و استفاده کنیم.

تا به اینجا فقط اولین قسمت از برنامه را نوشته‌ایم که می‌تواند درخواست‌های HTTP را دریافت کند. بر روی درخواست‌ها باید کارهای دیگری نیز صورت گیرد و این بستگی به آدرس‌هایی دارد که از طرف مرورگر درخواست می‌شوند که باید واکنش‌های مختلفی را برای آن‌ها در نظر بگیریم.

برای همچنین برنامه بسیار ساده‌ای می‌توانیم نیازهایمان را با تابع کال‌بک onRequest مرتفع کنیم اما ازآنجایی‌که گفت شد، می‌خواهیم برنامه مقداری لایه‌های بیشتر در خود داشته باشد و این فرآیند جذاب‌تر شود.

برای آنکه برنامه بتواند به درخواست‌های متنوع HTTP پاسخ دهد باید یک **مسیریاب** در برنامه وجود داشته باشید، بنابراین باید یک ماژول با عنوان router ایجاد شود.

مسیریابی (۲): چرا باید درخواست‌ها را مسیریابی کنیم؟

برنامه باید قادر به تغذیه کردن درخواست‌ها باشد، ممکن است درخواست‌های POST و GET وارد مسیریاب شوند و در این مرحله مسیریاب باید تصمیم بگیرد کدام چه کدی را اجرا کند.(اجرا کردن کد، قسمت سوم برنامه را تشکیل می‌دهد: پس از دریافت درخواست، مسیریاب کد درست را برای آن برمی‌گرداند و اجرا می‌کند.)

بنابراین، نیاز داریم تا پارامترهای موجود در URL را که از طریق POST/GET ارسال می‌شوند را استخراج و استفاده کنیم. می‌توان در نظر گرفت که بخشی از روتر باشد یا سرور(ماژول جداگانه) اما در حال حاضر اجازه دهید که آن را بخشی از HTTP سرور استدلال کنیم.

تمامی اطلاعات موردنیاز را از طریق شیء request که به‌عنوان اولین پارامتر تابع کال‌بک onRequest عبور داده شده است بدست می‌آوریم. برای تجزیه و تفسیر بهتر request به تعدادی ماژول Node.js نیاز داریم که عبارت‌اند از: url و querystring

ماژول url این امکان را می‌دهد که قسمت‌های مختلف یک URL را استخراج کنیم و به‌وسیله ماژول querystring می‌توانیم از پارامترهای آدرس استفاده کنیم.

به مثال زیر توجه کنید:

```
                                 url.parse(string).query
                                                |
                            url.parse(string).pathname      |
                                          |                 |
                                          |                 |
                                        ----- -------------------
http://localhost:8888/start?foo=bar&hello=world
                                     ---       -----
                                      |          |
                                      |          |
                             querystring(string)["foo"]     |
                                                |
                                   querystringtring(string)["hello"]
```

البته می‌توانیم با استفاده از querystring بدنه یک درخواست POST را برای پارامترها تجزیه کنیم. در فصل‌های بعد با این موضوع بیشتر آشنا خواهید شد.

اجازه دهید با اضافه کردن کد زیر به تابع onRequest() متوجه شویم که کاربر چه آدرسی را در مرورگر وارد کرده است:

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

بسیار خوب. در این زمان برنامه می‌تواند درخواست‌های مبتنی بر URL را تشخیص دهید و این به ما اجازه می‌دهد که کنترلرها را براین اساس طراحی کنیم.

برنامه ما باید قادر باشد آدرس‌های start/ و upload/ را کنترل کند، در ادامه به این موضوع می‌پردازیم.

زمان واقعی برای پیاده‌سازی مسیریاب فرا رسیده است، یک فایل با عنوان router.js ایجاد کنید و سپس کد زیر را در آن قرار دهید:

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

البته کد فوق هیچ عملکرد خاصی ندارد اما در حال حاضر کافی است. ابتدا اجازه دهید قبل از اعمال کدهای اصلی بررسی کنیم که چگونه می‌توانیم مسیریاب را به سرور متصل کنیم.

HTTP سرور نیاز دارد بداند که چگونه می‌تواند از مسیریاب(Router) استفاده کند. به دلیل اینکه مسیرهای سختی را در زبان‌های دیگر آموخته و تجربه کرده‌ایم شاید حس کنید باید ارتباط تنگاتنگی مابین مسیریاب و سرور ایجاد شود، اما قصد داریم با تزریق این وابستگی ارتباط زوج سرور و مسیریاب را آزادتر کنیم.

ابتدا تابع ()start سرور را توسعه می‌دهیم که به ما این اجازه را می‌دهد روتر درخواست‌ها را مسیریابی کند:

```
1  var http = require("http");
2  var url = require("url");
3
4  function start(route) {
5     function onRequest(request, response) {
6        var pathname = url.parse(request.url).pathname;
7        console.log("Request for " + pathname + " received.");
8
9        route(pathname);
10
11       response.writeHead(200, {"Content-Type": "text/plain"});
12       response.write("Hello World");
13       response.end();
14    }
15
16    http.createServer(onRequest).listen(8888);
17    console.log("Server has started.");
18 }
19
20 exports.start = start;
```

و همچنین باید فایل index.js را توسعه داد که براین اساس تابع route را به سرور تزریق می‌کنیم:

```
1  var server = require("./server");
2  var router = require("./router");
3
4  server.start(router.route);
```

مجدداً، یک تابع را به‌عنوان پارامتر عبور داده‌ایم که در فصول پیش این مفهوم را بررسی کردیم.

اگر برنامه را اجرا کنیم(node index.js، برای همیشه) و درخواستی را برای آن ارسال کنیم، می‌توانیم بگوییم که برنامه از مسیریاب(Router) استفاده کرده و درخواست را از خود عبور داده است، خروجی خط فرمان شما باید مشابه زیر باشد:

```
1  $ node index.js
2  Request for /foo received.
3  About to route a request for /foo
```

(در اینجا درخواست favicon.ico/ از خروجی حذف شده است.)

عمل کنید بجای نام بردن

ممکن است دوباره بخواهیم در مورد برنامه‌نویسی تابعی صحبت کنیم.

عبور دادن توابع تنها یک مبحث فنی نیست، با توجه به مبحث طراحی نرم‌افزار ممکن است فلسفی باشد. چند لحظه فکر کنیم: درون فایل index ما توانستیم شیء router را از سرور عبور دهیم و همچنین سرور توانست تابع route را از این شیء فراخوانی کند.

ازاین‌رو توانسته‌ایم یک شیء را عبور دهیم و ممکن است سرور از این شیء استفاده کند و کاری بر روی آن انجام دهد. "سلام مسیریاب، ممکن است این آدرس رو برای من پیدا کنی؟"

متوجه هستیم که سرور نیاز به شیء ندارد بلکه فقط نیاز دارد چیزی را بگیرد و کاری بر روی آن انجام دهد، ما نیز به شیء نیاز نداریم بلکه نیاز به عمل داریم. نیاز به نام بردن نداریم، نیاز به فعل داریم.

درک این مسئله نیاز به تغییر ذهنیت دارد تا جایی که در هسته این ایده باید پرسیده شود چرا من باید برنامه‌نویسی تابعی را بفهمم.

و من این مسئله را زمانی متوجه شدم که مقاله Execution in the Kingdom of Nouns را مطالعه کردم. برای درک بیشتر حتماً مقاله فوق را مطالعه کنید و یکی از بهترین مقالات مرتبط با توسعه نرم‌افزار است که مطالعه آن همیشه لذت‌بخش است.

مسیریابی (۳): مسیریابی درخواست‌ها برای یافتن کنترلر حقیقی

برگردیم به برنامه، تا جایی که در نظر گرفتیم، HTTP سرور و مسیریاب بهترین دوست هستند و با هم مکالمه دارند.

البته این مفهوم "مسیریابی" کافی نیست، می‌خواهیم آدرس‌های مختلفی به شکل‌های مختلف را مدیریت کنیم. اینجا ممکن است بخواهیم برای درخواست‌های /start و /upload یک "منطق تجاری" را در نظر بگیریم.

در حال حاضر مسیریابی در روتر بی‌معنی است، درواقع روتر کار خاصی بر روی درخواست‌ها انجام نمی‌دهد چراکه برای یک برنامه پیچیده عملکرد کافی را ندارد.

می‌خواهیم هنگامی‌که یک درخواست به سمت کنترلر خود هدایت‌شده است، توابعی را صدا بزنیم.

برای کنترلر درخواست‌ها قصد داریم ماژول جدیدی را ایجاد کنیم. فایلی با عنوان requestHandlers.js ایجاد می‌کنیم و برای درخواست‌های start و upload توابع را تعریف می‌کنم و درنهایت به‌صورت زیر آن‌ها را export می‌کنیم:

```
1  function start() {
2        console.log("Request handler 'start' was called.");
3  }
4
5  function upload() {
6    console.log("Request handler 'upload' was called.");
7  }
8
```

```
9  exports.start = start;
10 exports.upload = upload;
```

کد فوق این امکان را به ما می‌دهد که واپایشگرها را به روتر متصل کنیم و بتوانیم درخواست‌ها را به سمت کنترلرها هدایت کنیم.

در این مرحله نیاز به تصمیم‌گیری داریم: آیا برای استفاده از کنترلرها در روتر کد مناسبی نوشته شده است یا می‌خواهیم مقدار بیشتری وابستگی تزریق کنیم؟ اگرچه تزریق وابستگی مشابه الگوهای دیگر است اما در این موضوع شرایط متفاوت است و با تزریق وابستگی‌ها ارتباط زوج روتر و کنترلرها آزادانه‌تر خواهد بود و می‌توان از روتر استفاده مجدد کرد.

مفهوم فوق اشاره می‌کند، کنترلرها را از سرور به مسیریاب عبور دهیم که بسیار غلط است به همین دلیل باید کنترلرها را از فایل اصلی(index.js) عبور دهیم و سپس از آن به روتر انتقال دهیم.

چطور آن‌ها را عبور دهیم؟ در حال حاضر در برنامه دو کنترلر وجود داد این مقدار ممکن است افزایش یا تغییر یابد و مطمئناً قصد نداریم کار بیهوده‌ای برای طراحی مسیریاب انجام دهیم یعنی هرگاه یک درخواست جدید داشته باشیم آن را به شکل if request==x مقایسه کنیم و سپس کنترلر را از روتر صدا بزنیم، این روش اصلاً درست و استاندارد نخواهد بود.

تعدادِ متغیری از درخواست‌ها(آدرس‌های URL) وجود دارد که به شکل رشته(string) هستند. بسیار خوب ظاهراً آرایه‌های انجمنی[1] برای این مقصود مفید باشند.

یافته‌ها در این زمینه ناامیدکننده هستند، آیا جاوااسکریپت می‌تواند آرایه‌های انجمنی را پیاده‌سازی کند یا نه؟ می‌دانیم که شیء ما واقعاً نیاز به استفاده از یک آرایه انجمنی دارد.

برای این موضوع می‌توانیم به نکته‌ای از مقاله سایت MSDN Microsoft بپردازیم:

> زمانی که در C++ و C# در خصوص شیءها صحبت می‌کنیم درواقع به پیاده‌سازی کلاس‌ها(Classes) و ساختمان‌های داده(Structs) اشاره می‌کنیم. شیءها تفاوت‌هایی در ویژگی‌ها و متدهایشان دارند که بستگی به قالبی دارد که به ارث برده‌اند. مسئله این است که مبحث شیءها در جاوااسکریپت مشابه C++ و C# نیست درواقع شیءها در جاوااسکریپت فقط مجموعه‌ای از نام/مقدار هستند که به یکدیگر جفت شده‌اند. شیءها را در جاوااسکریپت باید به شکل یک فرهنگ لغت نگاه کرد که مجموعه‌ای از کلید رشته‌ها در آن قرار دارد.

اگر جاوااسکریپت فقط مجموعه‌ای از نام/مقدارهای جفت شده باشند، چطور می‌تواند متدهایی هم داشته باشد؟ خوب، مقادیر می‌توانند رشته، عدد و ... باشند و حتی یک تابع(function).

در حال حاضر به نوشتن برنامه برمی‌گردیم. می‌خواهیم مجموعه‌ای از کنترلرها را به‌عنوان یک شیء عبور دهیم و بر این اساس به یک زوج آزاد برسیم. درواقع شیء را به داخل route() تزریق کنیم.

بگذارید شیءها را در کنار یکدیگر در فایل index.js را قرار دهیم:

```
1  var server = require("./server");
2  var router = require("./router");
3  var requestHandlers = require("./requestHandlers");
```

[1] آرایه انجمنی – ویکی پدیا

```
4
5  var handle = {}
6  handle["/"] = requestHandlers.start;
7  handle["/start"] = requestHandlers.start;
8  handle["/upload"] = requestHandlers.upload;
9
10 server.start(router.route, handle);
```

بااینکه کنترلر بیشتر از یک "چیز"(مجموعه از درخواست‌ها) است، پیشنهاد می‌کنم اسم آن را مثل یک فعل در نظر بگیریم به این دلیل که نتیجه این نام‌گذاری در روتر بیان روانی را به دنبال خواهد داشت که به‌زودی خواهیم دید.

تا اینجا که مشاهده کردید به‌راحتی می‌شود آدرس‌های مختلفی را به کنترلر موردنظر هدایت کرد: به‌وسیله اضافه کردن یک کلید/مقدار که با "/" به یکدیگر جفت شده‌اند، requestHandlers.start را می‌توانیم زیباتر و واضح‌تر بیان کنیم چراکه نه‌تنها درخواست‌های start/ بلکه درخواست‌های / را می‌توانند به کنترلر start هدایت شوند.

بعد از تعریف شیء آن را به‌عنوان پارامتر اضافی به داخل سرور عبور می‌دهیم. اجازه دهید تغییراتی در ماژول server.js ایجاد کنیم:

```
1  var http = require("http");
2  var url = require("url");
3
4  function start(route, handle) {
5     function onRequest(request, response) {
6        var pathname = url.parse(request.url).pathname;
7        console.log("Request for " + pathname + " received.");
8
9        route(handle, pathname);
10
11       response.writeHead(200, {"Content-Type": "text/plain"});
12       response.write("Hello World");
13       response.end();
14    }
15
16    http.createServer(onRequest).listen(8888);
17    console.log("Server has started.");
18 }
19
20 exports.start = start;
```

یک پارامتر با عنوان handle به تابع ()start اضافه کرده‌ایم و سپس شیء handle را به‌عنوان اولین پارامتر به داخل تابع کال‌بک ()route عبور داده‌ایم.

بر این اساس تابع ()route درون فایل router.js نیز باید تغییراتی داشته باشد:

```
1  function route(handle, pathname) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4        handle[pathname]();
5     } else {
6        console.log("No request handler found for " + pathname);
7     }
8  }
9
10 exports.route = route;
```

چه کاری اینجا انجام داده‌ایم؟ ما بررسی می‌کنیم که اگر یک کنترلر مطابق با درخواست کاربر وجود دارد و همچنین اگر آن کنترلر نیز وجود دارد، تابع مرتبط با آن را صدا می‌زنیم. بنابراین می‌توانیم به توابع کنترلر شیء(Object) دسترسی داشته باشیم درست مانند اینکه به یک المنت از یک آرایه انجمنی دسترسی داشته باشیم، حال برای handle[pathname](); تعریف روان‌تری داریم که در قبل به آن اشاره شد: " لطفاً، به این آدرس رسیدگی کن".

بسیار خوب، همه آن چیزی که نیاز داریم متصل کردن سرور، روتر و کنترلر به یکدیگر است! هنگامی‌که برنامه اجرا شود و یک درخواست http://localhost:8888/start از مرورگر ارسال کنیم درواقع می‌توانیم اطمینان حاصل کنیم که کنترلر موردنظر فراخوانی شده است:

```
1  Server has started.
2  Request for /start received.
3  About to route a request for /start
4  Request handler 'start' was called.
```

و اگر آدرس http://localhost:8888/ را در مرورگر باز کنیم به ما ثابت می‌کند که کنترلر این درخواست به‌درستی فراخوانی شده است:

```
1  Server has started.
2  Request for / received.
3  About to route a request for /
4  Request handler 'start' was called.
```

مسیریابی (۴): پاسخ دادن به درخواست‌ها

بسیار عالی، حال درواقع اگر تنها کنترلر بتواند چیزی به مرورگر برگرداند حتی بهتر خواهد شد.

به یاد داشته باشید، هنگامی‌که یک صفحه را درخواست می‌کنیم عبارت "Hello World" نمایش داده می‌شود که دلیل آن تابع onRequest در فایل server.js است.

"رسیدگی به درخواست‌ها" به معنی "پاسخ دادن به درخواست‌ها" است بدین منظور ما نیاز داریم که ایجاب کنیم کنترلر با مرورگر گفتگو کند درست مثل وظیفه‌ای که تابع onRequest انجام می‌دهد.

روش ساده‌ای که به‌عنوان توسعه‌دهنده با پیشینه PHP و Ruby ممکن است بخواهیم آن را دنبال کنیم: کنترلر محتوایی که کاربر می‌خواهد نمایش داده شود را به داخل تابع onRequest ارسال کند و به کاربر ارائه دهد.

ابتدا اجازه دهید این روش را پیاده‌سازی کنیم و بعد متوجه خواهید شد که این ایده مناسب نیست.

با قسمت کنترلر شروع می‌کنم و طوری کد را بازنویسی می‌کنم که عبارت موردنظر در مرورگر نمایش داده شود. نیاز داریم که در فایل requestHandlers.js تغییراتی ایجاد کنیم:

```
1  function start() {
2    console.log("Request handler 'start' was called.");
3    return "Hello Start";
4  }
5
6  function upload() {
7    console.log("Request handler 'upload' was called.");
```

```
8     return "Hello Upload";
9  }
10
11 exports.start = start;
12 exports.upload = upload;
```

به‌علاوه روتر باید آن چیزی که از کنترلر درخواست شده را به سرور بازگرداند، ازاین‌رو router.js را به شکل زیر ویرایش می‌کنیم:

```
1  function route(handle, pathname) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4        return handle[pathname]();
5     } else {
6        console.log("No request handler found for " + pathname);
7        return "404 Not found";
8     }
9  }
10
11 exports.route = route;
```

همان‌طور که می‌بینید، برای درخواست‌های که وجود ندارند و نمی‌توانند ره‌گیری شوند پیام ۴۰۴ در نظر گرفته شده است.

در آخر، سرور را بازنویسی خواهیم کرد تا بتواند با محتوایی که کنترلر به‌وسیله روتر برگشت داده به مرورگر پاسخ دهد. به شکل زیر:

```
1  var http = require("http");
2  var url = require("url");
3
4  function start(route, handle) {
5     function onRequest(request, response) {
6        var pathname = url.parse(request.url).pathname;
7        console.log("Request for " + pathname + " received.");
8
9        response.writeHead(200, {"Content-Type": "text/plain"});
10       var content = route(handle, pathname);
11       response.write(content);
12       response.end();
13    }
14
15    http.createServer(onRequest).listen(8888);
16    console.log("Server has started.");
17 }
18
19 exports.start = start;
```

اگر برنامه بازنویسی شده را اجرا کنیم، خواهید دید که همه چیز به‌خوبی کار می‌کند: در مرورگر درخواست http://localhost:8888/start نتیجه "Hello Start" را برمی‌گرداند، درخواست http://localhost:8888/upload عبارت "Hello Upload" را نمایش می‌دهد و اگر خارج از این دو باشد عبارت "404 Not Found" تولید می‌شود.

خوب پس چه مشکلی وجود دارد؟ پاسخ کوتاه: در زمان اجرا اگر یکی از کنترلرها نیاز به فرآیند non-blocking داشته باشد در آینده با مشکل مواجهه می‌شویم.

در ادامه به پاسخ بلند خواهیم پرداخت.

مسدودسازی و غیر مسدودسازی

همان‌طور که گفته شد، مشکلات زمانی ایجاد می‌شوند که یک کنترلر نیاز به فرآیند غیر مسدودسازی (non-blocking) داشته باشد اما اجازه دهید ابتدا در خصوص فرآیند مسدودسازی (blocking) صحبت کنیم و سپس فرآیند غیر مسدودسازی.

قبل از توضیح دادن مبحث فوق، بررسی خواهیم کرد اگر یک فرآیند مسدودساز به برنامه اضافه شود چه اتفاقی رخ خواهد داد.

برای این کار باید در کنترلر start قبل از بازگشت دادن رشته "Hello World" یک تأخیر ۱۰ ثانیه‌ای ایجاد شود. چون در جاوااسکریپت چیزی مانند ()sleep وجود ندارد، از یک ترفند هوشمندانه‌تری استفاده خواهیم کرد.

requestHandlers.js را به شکل زیر ویرایش کنید:

```
1  function start() {
2    console.log("Request handler 'start' was called.");
3
4    function sleep(milliSeconds) {
5      var startTime = new Date().getTime();
6      while (new Date().getTime() < startTime + milliSeconds);
7    }
8
9    sleep(10000);
3    return "Hello Start";
4  }
5
6  function upload() {
7    console.log("Request handler 'upload' was called.");
8    return "Hello Upload";
9  }
10
11 exports.start = start;
12 exports.upload = upload;
```

در کد فوق واضح است که اگر تابع ()start فراخوانی شود، Node.js ۱۰ ثانیه صبر می‌کند و سپس عبارت "Hello World" را برگشت خواهد داد و زمانی که ()upload فراخوانی شود بلافاصله محتوا را برگشت خواهد داد.

بعد از ایجاد تغییرات سرور را restart کنید. اجازه دهید بررسی کنیم که چه اتفاقی خواهد افتاد: ابتدا در مرورگر دو تب یا پنجره باز کنید سپس در آدرس بار پنجره اول آدرس http://localhost:8888/start را وارد کنید اما صفحه را **باز نکنید**.

در آدرس بار پنجره دوم آدرس http://localhost:8888/upload را وارد کنید و دوباره صفحه را **باز نکنید**.

حال، به این صورت عمل کنید: ابتدا در پنجره اول آدرس ("/start") را باز کنید سپس به‌سرعت به پنجره دوم بروید و آدرس ("/upload") را باز کنید.

چه چیز را متوجه شدید: همان‌طور که انتظار داریم آدرس /start مدت ۱۰ ثانیه زمان می‌برد تا بارگذاری شود و همچنین آدرس /upload مدت ۱۰ ثانیه زمان می‌برد تا بارگذاری شود درصورتی‌که هیچ‌گونه تابع ()sleep برای آن در نظر گرفته نشده است!

چرا؟ به دلیل اینکه کنترلر start() دارای یک فرآیند مسدودساز است. در حال حاضر در خصوص مدل اجرایی Node.js صحبت می‌کنیم که با فرآیندهای سنگینی مشکل ندارند اما باید اهمیت بدهیم که فرآیندهای دیگر Node.js با آن مسدود نشود و در عوض هرگاه فرآیندهای سنگینی اجرا شوند باید در پس‌زمینه برنامه قرار گیرند و رویدادهای آن باید به‌وسیله حلقه رویداد (Event Loop) به کار گرفته شود.

و متوجه شدیم که چرا روش فوق به ما اجازه استفاده از فرآیند غیر مسدودساز در برنامه را نمی‌دهد.

برای درک بهتر یک‌بار دیگر می‌خواهیم مستقیماً مشکل فوق را تجربه کنیم. برای این کار فایل requestHandlers.js را به شکل زیر ویرایش کنید:

```
var exec = require("child_process").exec;

function start() {
   console.log("Request handler 'start' was called.");
   var content = "empty";

   exec("ls -lah", function (error, stdout, stderr) {
      content = stdout;
   });

   return content;
}

function upload() {
   console.log("Request handler 'upload' was called.");
   return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

مشاهده می‌کنید در کد فوق یک ماژول جدید از Node.js به نام child_process به برنامه اضافه شده است و به ما این امکان را می‌دهد که به‌راحتی در کنترلر از فرآیند غیر مسدودساز استفاده کنیم: exec()

exec() چه کاری انجام می‌دهد؟ می‌تواند داخل Node.js یک دستور Shell اجرا کند. در مثال فوق با استفاده از دستور ("ls –lah") فهرست تمام فایل‌های داخل پوشه جاری را دریافت می‌کنیم و این اجازه را به ما می‌دهد که این لیست را در آدرس start/ که کاربر درخواست می‌کند نمایش دهیم.

برنامه را اجرا و آدرس http://localhost:8888/start را باز کنید.

صفحه به‌درستی بارگذاری شده است اما رشته "empty" نمایش داده می‌شود. چه مشکلی وجود دارد؟

خوب، ممکن است حدس زده باشید، exec() به‌صورت خیلی سحرآمیز و سریع دستور "ls –lah" را انجام می‌دهد و ازاین‌رو ماژول مفیدی است چراکه می‌توانیم به‌وسیله آن دستورات سنگینی (کپی کردن فایل‌های سنگین) را اجرا کنیم بدون آن‌که برنامه را به‌صورت اجباری به‌عنوان یک مسدودساز به حالت توقف کامل ببریم.

(اگر مایل باشید دستور "ls –lah" را با دستور سنگین‌تر "find /" جایگزین کنیم).

خوب، اجازه دهید کد را بازنویسی کنیم و بفهمیم که چرا ساختار فوق کار نمی‌کند.

مشکل درون exec() است، برای اینکه یک non-blocking داشته باشیم باید از تابع کال‌بک استفاده کنیم. در مثال یک تابع بی‌نام که به‌عنوان پارامتر دوم از تابع exec() عبور داده شده که ریشه مشکل ما در آن نهفته است:

```
function (error, stdout, stderr) {
   content = stdout;
}
```

برنامه به‌صورت هم‌زمان (synchronous) اجراشده‌اجراشده است به این معنی که بلافاصله پس از فراخوانی exec()، Node.js ادامه می‌دهد تا اطلاعات را بازگشت دهد. اینجاست که متغیر content هنوز رشته "empty" را در خود دارد و تابع کال‌بک عبور داده شده از exec() هنوز فراخوانی نشده است، با توجه به این واقعیت می‌توان گفت exec() نیز به‌صورت ناهم‌زمان اجراشده است.

دستور "ls -lah" بسیار سبک و سریع است به همین دلیل تابع کال‌بک نسبتاً سریع فراخوانی می‌شود اما بااین‌وجود حالت ناهم‌زمان رخ می‌دهد.

فکر کردن درباره یک دستور سنگین‌تر موضوع فوق را واضح‌تر می‌کند: برای این کار می‌توانیم از دستور "find /" استفاده کنیم که روی دستگاه من اجرای آن حدود ۱ دقیقه طول می‌کشد. درصورتی‌که در کنترلر "ls -lah" را با "find /" جایگزین کنم و زمانی که آدرس start/ را باز کنم بلافاصله یک پاسخ HTTP دریافت می‌شود که نشان می‌دهد exec() در پس‌زمینه در حال انجام کاری است، با این اوصاف Node.js برنامه را ادامه می‌دهد و تنها زمانی که دستور "find /" اجرایش به پایان رسیده باشد می‌توانیم فرض می‌کنیم که تابع کال‌بک عبور داده شده از exec() صدا زده خواهد شد.

اما چگونه می‌توان به این هدف رسید، مانند نمایش دادن فهرستی از فایل‌ها در پوشه جاری به کاربر؟

بعد از درک مباحث فوق می‌خواهیم در خصوص روش پاسخ دادن صحیح کنترلر به مرورگر صحبت کنیم.

پاسخ دادن ناهم‌زمان به درخواست‌ها

قصد دارم از عبارت " روش صحیح " استفاده کنم که خطرناک است چراکه اغلب تنها یک راه صحیح وجود ندارد؛ اما برای این مسئله باوجوداینکه در طول برنامه از توابع ناهم‌زمان استفاده می‌کنیم، فقط یک راه‌کار در Node.js ممکن است.

در حال حاضر برنامه قادر است اطلاعات را از کنترلر به HTTP سرور به‌وسیله لایه‌های برنامه (کنترلر ← روتر ← سرور) انتقال دهد.

روش جدیدی برای آن در نظر گرفته‌ایم: بجای هدایت اطلاعات به سرور، سرور را به سمت اطلاعات هدایت می‌کنیم، به‌صورت دقیق‌تر، می‌خواهیم شیء response را از طریق روتر به کنترلر تزریق کنیم، درنتیجه کنترلر قادر خواهد بود از توابع این شیء برای پاسخ دادن به درخواست‌ها استفاده کند.

توضیح دادن کافی است، در زیر به‌صورت گام‌به‌گام تغییرات برنامه را متوجه خواهید شد، server.js را به شکل زیر ویرایش کنید:

```
var http = require("http");
var url = require("url");

function start(route, handle) {
   function onRequest(request, response) {
      var pathname = url.parse(request.url).pathname;
```

```
7       console.log("Request for " + pathname + " received.");
8
9       route(handle, pathname, response);
10    }
11
12    http.createServer(onRequest).listen(8888);
13    console.log("Server has started.");
14  }
15
16  exports.start = start;
```

به‌جای انتظار بازگشت یک مقدار از تابع ()route، شیء response را به‌عنوان پارامتر سوم عبور داده‌ایم. علاوه بر این، متدهای response را از تابع onRequest حذف کرده‌ایم چراکه الآن انتظار داریم route از این روش استفاده کند:

```
1   function route(handle, pathname, response) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4       handle[pathname](response);
5     } else {
6       console.log("No request handler found for " + pathname);
7       response.writeHead(404, {"Content-Type": "text/plain"});
8       response.write("404 Not found");
9       response.end();
10    }
11  }
12
13  exports.start = start;
```

کد فوق همانند الگو مدنظر است: به‌جای انتظار بازگشت یک مقدار از کنترلر، شیء response را از آن عبور می‌دهیم.

اگر مسیریاب برای درخواست نتواند هیچ کنترلری را پیدا کند، سعی می‌کنیم پاسخ ۴۰۴ را برای مرورگر ارسال کنیم.

و در آخر باید فایل requestHandlers.js را به شکل زیر ویرایش کرد:

```
1   var exec = require("child_process").exec;
2
3   function start(response) {
4     console.log("Request handler 'start' was called.");
5
6     exec("ls -lah", function (error, stdout, stderr) {
7       response.writeHead(200, {"Content-Type": "text/plain"});
8       response.write(stdout);
9       response.end();
10    });
11  }
12
13  function upload(response) {
14    console.log("Request handler 'upload' was called.");
15    response.writeHead(200, {"Content-Type": "text/plain"});
16    response.write("Hello Upload");
17    response.end();
18  }
19
20  exports.start = start;
```

```
21 exports.upload = upload;
```

تابع کنترلر نیاز دارد که پارامتر response را بپذیرد و از آن استفاده کند تا بتواند مستقیماً به درخواست پاسخ دهد.

کنترلر start با استفاده از تابع کال‌بک درون exec() پاسخ را فراهم می‌کند و کنترلر upload هنوز رشته "Hello Upload" را نمایش می‌دهد اما این پاسخ را با استفاده از شیء response ارسال می‌کند

با اجرای مجدد انتظار می‌رود برنامه به‌درستی کار کند (node index.js).

اگر مایل باشید می‌توانید از یک دستور سنگین‌تری در start/ استفاده کنید که نشان می‌دهد درخواست upload/ مسدود نخواهد شد و بلافاصله نمایش داده می‌شود.

به شکل زیر در requestHandlers.js تغییرات زیر را اعمال کنید:

```
1  var exec = require("child_process").exec;
2
3  function start(response) {
4    console.log("Request handler 'start' was called.");
5
6    exec("find /",
7      { timeout: 10000, maxBuffer: 20000*1024 },
8      function (error, stdout, stderr) {
9        response.writeHead(200, {"Content-Type": "text/plain"});
10       response.write(stdout);
11       response.end();
12     });
13 }
14
15 function upload(response) {
16   console.log("Request handler 'upload' was called.");
17   response.writeHead(200, {"Content-Type": "text/plain"});
18   response.write("Hello Upload");
19   response.end();
20 }
21
22 exports.start = start;
23 exports.upload = upload;
```

در کد فوق درخواست HTTP برای http://localhost:8888/start ۱۰ ثانیه طول خواهد کشید تا بارگذاری شود اما درخواست http://localhost:8888/upload بلافاصله پاسخ داده می‌شود حتی اگر start/ هنوز در حال پردازش باشد.

یک گام به جلو

تابه‌حال، تمامی کارهایی که انجام شده است بسیار خوب و زیبا هستند؛ اما در حقیقت ارزشی برای کاربر قائل نبوده‌ایم.

سرور، روتر و کنترلرها درجای خود قرار دارند بنابراین الآن می‌توانیم قسمت محتوا (content) را برای سایت در نظر بگیریم و به کاربران این اجازه را بدهیم که با برنامه تعامل داشته باشند و بتوانند با انتخاب فایل موردنظر

اقدام به آپلود کنند و سپس فایل آپلود شده را در مرورگر خود مشاهده کنند. به‌منظور ساده‌تر کردن فرآیند آپلود فرض خواهیم کرد که کاربر فقط بتواند تصاویر را آپلود و مشاهده کند.

در قدم اول نگاه خواهیم کرد که چگونه درخواست‌های ورودی POST را کنترل کنیم و در قدم دوم با استفاده از یک ماژول خارجی Node.js فرآیند آپلود فایل را کنترل می‌کنیم. این روش را به دو دلیل انتخاب کرده‌ام.

- کنترل کردن درخواست‌های POST نسبتاً با Node.js ساده است، ولی شاید تا به اینجا آموختن کافی باشد اما ارزش تمرین بیشتر را دارد.
- کنترل کردن آپلود فایل (مانند درخواست‌های چند بخشه POST) در Node.js ساده نیست و بنابراین باید فراتر از محدوده این کتاب قدم برداریم و می‌توانیم برای این آموزش از یک ماژول خارجی استفاده کنیم.

مدیریت درخواست‌های POST

بیاید ساده به این موضوع نگاه کنیم: یک فیلد textarea را در نظر می‌گیریم که کاربر می‌تواند آن را پر کند و با یک درخواست POST آن را برای سرور ارسال کند. پس از دریافت و رسیدگی به این درخواست، برنامه موظف است اطلاعات درون textarea را نمایش دهد.

برای استفاده از این فرم نیاز داریم که در فایل requestHandlers.js بر روی کنترلر */start* تغییراتی را به شکل زیر اعمال کنیم:

```
function start(response) {
   console.log("Request handler 'start' was called.");

   var body = '<html>'+
      '<head>'+
      '<meta http-equiv="Content-Type" content="text/html; '+
      'charset=UTF-8" />'+
      '</head>'+
      '<body>'+
      '<form action="/upload" method="post">'+
      '<textarea name="text" rows="20" cols="60"></textarea>'+
      '<input type="submit" value="Submit text" />'+
      '</form>'+
      '</body>'+
      '</html>';

   response.writeHead(200, {"Content-Type": "text/html"});
   response.write(body);
   response.end();
}

function upload(response) {
   console.log("Request handler 'upload' was called.");
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("Hello Upload");
   response.end();
}

exports.start = start;
exports.upload = upload;
```

باید با باز کردن آدرس http://localhost:8888/start فرم را مشاهده کنید، در غیر این صورت اگر فرمی وجود نداشت برنامه را restart کنید.

اگر فکر می‌کنید که کدهای کنترلر زیبایی کافی را ندارند و زشت هستند باید گفت که در این آموزش تصمیم گرفته‌ام که سطح اضافی (جدا کردن view از controller) به برنامه اضافه نشود چراکه مبحث خاصی برای ما در زمینه آموزش جاوااسکریپت و Node.js نخواهد بود.

حال که به یک تازه‌کار پیشرفته تبدیل شده‌ایم نباید از این واقعیت تعجب کنیم که چگونه به‌وسیله توابع کال‌بک غیر مسدودساز، داده‌های (data) یک درخواست POST را به شکل ناهم‌زمان کنترل کنیم.

شاید حس کنید که درخواست‌های POST می‌توانند پتانسیل سنگین بودن را داشته باشند به این دلیل که هیچ‌چیز کاربر را از وارد کردن اطلاعات متوقف نمی‌کند و در این حال درخواست‌ها می‌توانند حجم‌های مگابایتی داشته باشند. رسیدگی به طیف مختلفی از داده‌ها در یک فرآیند ممکن است نتیجه آن فرآیند مسدودساز باشد.

برای اینکه فرآیند غیر مسدودساز داشته باشیم، Node.js این امکان را می‌دهد که داده‌های یک درخواست POST را به قطعات مختلفی تقسیم کنیم و توابع کال‌بک را به رویدادهای خاصی اختصاص دهیم و فراخوانی کنیم. این رویدادها شامل رویداد داده (یک قطعه جدید از POST می‌رسد) و پایان (تمامی قطعه‌ها دریافت شده‌اند) هستند.

نیاز داریم به Node.js بگوییم، زمانی که رویدادها رخ می‌دهند کدام توابع کال‌بک را صدا بزند. این کار در صورتی امکان‌پذیر است که به شیء *request*، شنود (listeners) اضافه کنیم و هر زمان که یک درخواست HTTP دریافت می‌شود تابع کال‌بک عبور داده شده از *onRequest* را صدا بزند.

که می‌تواند به شکل زیر باشید:

```
request.addListener("data", function(chunk) {
   // called when a new chunk of data was received
});

request.addListener("end", function() {
   // called when all chunks of data have been received
});
```

در پیاده‌سازی مفهوم فوق سؤال‌هایی مطرح هستند. در حال حاضر فقط در سرور می‌توانیم به شیء *request* دسترسی داشته باشیم و آن را از روتر و کنترلر عبور نمی‌دهیم، همانند کاری که با شیء *response* صورت گرفت.

به نظر من، کار یک HTTP سرور در برنامه گرفتن تمامی اطلاعات از درخواستی است که نیاز دارد بر روی آن کاری انجام شود؛ بنابراین پیشنهاد می‌کنم در سرور داده‌های POST پردازش شوند و بعد داده نهایی را به داخل روتر و کنترلر عبور دهیم که درنهایت می‌توان تصمیم گرفت چه کاری بر روی آن‌ها صورت گیرد.

بنابراین ایده این است که توابع کال‌بک *data* و *end* را در سرور قرار دهیم و تمامی قطعات داده POST را تا زمانی که داده‌ها به داخل روتر و سپس به کنترلر عبور داده شوند در تابع کال‌بک *data* جمع‌آوری کنیم و زمانی که داده‌ها دریافت شد تابع *route* را در *end* صدا بزنیم.

ابتدا باید فایل server.js را به شکل زیر بازنویسی کرد:

```
var http = require("http");
var url = require("url");
```

```
3
4  function start(route, handle) {
5     function onRequest(request, response) {
6        var postData = "";
7        var pathname = url.parse(request.url).pathname;
8        console.log("Request for " + pathname + " received.");
9
10       request.setEncoding("utf8");
11
12       request.addListener("data", function(postDataChunk) {
13          postData += postDataChunk;
14          console.log("Received POST data chunk '"+
15          postDataChunk + "'.");
16       });
17
18       request.addListener("end", function() {
19          route(handle, pathname, response, postData);
20       });
21
22    }
23
24    http.createServer(onRequest).listen(8888);
25    console.log("Server has started.");
26 }
27
28 exports.start = start;
```

به صورتی کلی در قطعه کد بالا سه کار صورت می‌گیرد که منجر می‌شود بتوانیم از داده‌ها در کنترلرها استفاده کنیم: ابتدا، تعریف می‌کنیم که انتظار برود داده‌هایی که دریافت می‌شوند ساختار UTF-8 داشته باشند همچنین یک شنونده رویداد به نام "data" اضافه کرده‌ایم که در آن، وقتی‌که یک قطعه جدید از داده‌های POST فرا می‌رسد مرحله‌به‌مرحله مقدار متغیر *postData* تازه و بروز می‌شود و زمانی که تمامی اطلاعات جمع‌آوری شدند با استفاده از تابع کال‌بک *end* اطلاعات را به داخل *router* انتقال می‌دهیم.

اضافه کردن ثبت گزارش در خط فرمان برای هر قطعه‌ای که دریافت می‌شود ممکن است ایده جالبی نباشد؛ اما در فصل بعدی به‌صورت جزئی این موضوع را بررسی خواهیم کرد.

اجازه دهید برنامه را جذاب‌تر کنیم، در صفحه */upload* می‌خواهیم محتوای دریافتی را نمایش دهیم. برای این کار نیاز داریم که متغیر *postData* را به کنترلر ببریم. باید فایل router.js را به شکل زیر بازنویسی کرد:

```
1  function route(handle, pathname, response, postData) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4        handle[pathname](response, postData);
5     } else {
6        console.log("No request handler found for " + pathname);
7        response.writeHead(404, {"Content-Type": "text/plain"});
8        response.write("404 Not found");
9        response.end();
10    }
11 }
12
13 exports.route = route;
```

و در requestHandlers.js باید ایجاب کنیم که کنترلر upload بتواند محتوا را نمایش دهد:

```
1  function start(response, postData) {
```

```
2     console.log("Request handler 'start' was called.");
3
4     var body = '<html>'+
5        '<head>'+
6        '<meta http-equiv="Content-Type" content="text/html; '+
7        'charset=UTF-8" />'+
8        '</head>'+
9        '<body>'+
10       '<form action="/upload" method="post">'+
11       '<textarea name="text" rows="20" cols="60"></textarea>'+
12       '<input type="submit" value="Submit text" />'+
13       '</form>'+
14       '</body>'+
15       '</html>';
16
17       response.writeHead(200, {"Content-Type": "text/html"});
18       response.write(body);
19       response.end();
20  }
21
22  function upload(response, postData) {
23     console.log("Request handler 'upload' was called.");
24     response.writeHead(200, {"Content-Type": "text/plain"});
25     response.write("You've sent: " + postData);
26     response.end();
27  }
28
29  exports.start = start;
30  exports.upload = upload;
```

همین بود، حال قادر هستیم داده‌های POST را دریافت و در کنترلرها استفاده کنیم.

نکته‌ای در این مبحث وجود دارد: آنچه از روتر و کنترلرها عبور داده شد، بدنه[1] کامل درخواست POST بود. احتمال دارد بخواهیم در اینجا از فیلدهای فرم که درخواست POST را کامل می‌کنند استفاده کنیم ازاین‌رو می‌توانیم به فیلد *text* اشاره کنیم که در قطعه کد زیر به کار گرفته شده است.

در فصول قبل در خصوص ماژول *querystring* صحبت کردیم که به شکل زیر به ما کمک می‌کند:

```
1   var querystring = require("querystring");
2
3   function start(response, postData) {
4      console.log("Request handler 'start' was called.");
5
6      var body = '<html>'+
7         '<head>'+
8         '<meta http-equiv="Content-Type" content="text/html; '+
9         'charset=UTF-8" />'+
10        '</head>'+
11        '<body>'+
12        '<form action="/upload" method="post">'+
13        '<textarea name="text" rows="20" cols="60"></textarea>'+
14        '<input type="submit" value="Submit text" />'+
15        '</form>'+
16        '</body>'+
17        '</html>';
18
19        response.writeHead(200, {"Content-Type": "text/html"});
```

[1] HTTP Body

```
20      response.write(body);
21      response.end();
22  }
23
24  function upload(response, postData) {
25     console.log("Request handler 'upload' was called.");
26     response.writeHead(200, {"Content-Type": "text/plain"});
27      response.write("You've sent the text: "+
28     querystring.parse(postData).text);
29     response.end();
30  }
31
32  exports.start = start;
33  exports.upload = upload;
```

همه چیز در خصوص بارگیری و تخلیه داده‌های POST توضیح داده شد و برای یک آموزش ابتدایی تا به اینجا بسیار عالی است.

مدیریت فایل‌ها

این فصل به ما کمک می‌کند که روش نصب کتابخانه‌های خارجی در Node.js و استفاده آن‌ها در داخل کدها را فرا بگیریم.

در اهداف برنامه، کاربران این اجازه را داشتند که اقدام به آپلود فایل تصویری و نمایش آن در مرورگر کنند. قصد داریم از یک ماژول خارجی (External) برای مدیریت فایل‌ها استفاده کنیم که *node-formidable* نام دارد و توسط Felix Geisendoerfer نوشته شده است که به‌خوبی تمامی اطلاعات فایل را تجزیه می‌کند. مدیریت فایل‌ها " تنها " اشاره به همان مدیریت داده‌های POST می‌کند که استفاده از راه‌حل آماده فوق حس خیلی خوبی ایجاد می‌کند.

برای استفاده از کدهای Felix باید اقدام به نصب ماژول کنیم، برای این منظور می‌توانیم از مدیر بسته‌های Node که به‌اختصار NPM نامیده می‌شود استفاده کنیم و ماژول خارجی را نصب کنیم.

برای این کار دستور زیر را در خط فرمان وارد کنید:

```
1  npm install formidable
```

اگر در خط فرمان نتیجه مشابه زیر بود:

```
1  formidable@*.*.* node_modules\formidable
```

ماژول به‌خوبی نصب شده است و می‌توانید از آن استفاده کنید.

حال کافی است به شکل زیر ماژول *formidable* به برنامه اضافه شود:

```
1  var formidable = require("formidable");
```

اصطلاح formidable برای فرمی به کار می‌رود که به‌وسیله HTTP POST ارسال شده است و در Node.js قابل استفاده است. تمام چیزی که نیاز داریم ساخت یک **فرم ورودی** است که درنهایت بتوانیم به‌وسیله شیء *request* در HTTP سرور به فیلدها و فایل‌های ارسال شده توسط این فرم دسترسی پیدا کنیم.

کد زیر به‌عنوان مثالی از یک پروژه node-formidable است که تعامل قسمت‌های مختلف را نمایش می‌دهد:

```
var formidable = require('formidable'),
   http = require('http'),
   sys = require('sys');

http.createServer(function(req, res) {
   if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
      // parse a file upload
      var form = new formidable.IncomingForm();
      form.parse(req, function(error, fields, files) {
         res.writeHead(200, {'content-type': 'text/plain'});
         res.write('received upload:\n\n');
         res.end(sys.inspect({fields: fields, files: files}));
      });
      return;
   }

   // show a file upload form
   res.writeHead(200, {'content-type': 'text/html'});
   res.end(
      '<form action="/upload" enctype="multipart/form-data" '+
      'method="post">'+
      '<input type="text" name="title"><br>'+
      '<input type="file" name="upload" multiple="multiple"><br>'+
      '<input type="submit" value="Upload">'+
      '</form>'
   );
}).listen(8888);
```

اگر کد فوق را در یک فایل قرار دهیم و سپس آن را اجرا کنیم، یک فرم را مشاهده خواهیم کرد که می‌توانیم در آن اقدام به آپلود فایل کنیم و اطلاعات فایل را در خط فرمان مشاهده کنیم که درنهایت خروجی زیر را تولید می‌کند:

```
received upload:

{ fields: { title: 'Hello World' },
   files:
      { upload:
         { size: 1558,
           path: '/tmp/1c747974a27a6292743669e91f29350b',
           name: 'us-flag.png',
           type: 'image/png',
           lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,
           _writeStream: [Object],
           length: [Getter],
           filename: [Getter],
           mime: [Getter] } } }
```

به‌منظور استفاده از این حالت آنچه نیاز داریم، اضافه کردن حالت منطقی formidable به ساختار برنامه است به‌علاوه باید متوجه باشیم که هنگام درخواست نمایش از طرف مرورگر چگونه از فایل‌های آپلودی استفاده کنیم.

ابتدا به مسئله دوم می‌پردازیم و سپس اولی: اگر یک فایل بر روی هارد دیسک محلی وجود داشته باشد، چگونه برای یک درخواست سمت مرورگر از آن استفاده کنیم؟

جای تعجب نیست، ماژول پیش‌فرضی برای این کار وجود دارد به نام *fs* که به‌وضوح این کمک را می‌کند که به‌وسیله سرور اقدام به خواندن محتوای فایل می‌کنیم.

اجازه دهید یک کنترلر برای آدرس */show* ایجاد کنیم که به ما کمک می‌کند فایل */tmp/test.png* را نمایش دهیم. البته در ابتدا شاید فکر کنید در حال ذخیره‌سازی یک فایل PNG واقعی هستیم.

(نکته: پوشه‌ای در کنار سایر فایل‌های برنامه با عنوان *tmp* ایجاد کنید سپس تصویری با عنوان *test.png* در پوشه قرار دهید.)

فایل requestHandlers.js را به شکل زیر ویرایش کنید:

```
var querystring = require("querystring"), fs = require("fs");

function start(response, postData) {
   console.log("Request handler 'start' was called.");

   var body = '<html>'+
      '<head>'+
      '<meta http-equiv="Content-Type" '+
      'content="text/html; charset=UTF-8" />'+
      '</head>'+
      '<body>'+
      '<form action="/upload" method="post">'+
      '<textarea name="text" rows="20" cols="60"></textarea>'+
      '<input type="submit" value="Submit text" />'+
      '</form>'+
      '</body>'+
      '</html>';

      response.writeHead(200, {"Content-Type": "text/html"});
      response.write(body);
      response.end();
}

function upload(response, postData) {
   console.log("Request handler 'upload' was called.");
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("You've sent the text: "+
   querystring.parse(postData).text);
   response.end();
}

function show(response) {
   console.log("Request handler 'show' was called.");
   response.writeHead(200, {"Content-Type": "image/png"});
   fs.createReadStream("/tmp/test.png").pipe(response);
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

همچنین نیاز داریم کنترلر جدید را به فایل index.js اضافه کنیم:

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;

server.start(router.route, handle);
```

با راه‌اندازی مجدد سرور (restart) و باز کردن آدرس http://localhost:8888/show در مرورگر، فایل ذخیره شده در */tmp/test.png* باید نمایش داده شده باشد.

بسیار خوب، همه مراحلی که تا اتمام برنامه باید انجام دهیم:

- ایجاد فرمی در کنترلر *start* برای آپلود کردن فایل
- افزودن node-formidable به کنترلر *upload* و ذخیره کردن فایل آپلودی در */tmp/test.png*
- قرار دادن تصویر آپلود شده در خروجی HTML به‌وسیله آدرس */upload*

مرحله اول ساده است. نیاز داریم که رمزگذاری *multipart/form-data* را به فرم HTML اضافه کنیم سپس فیلد textarea را حذف و یک فیلد آپلود فایل به فرم اضافه می‌کنیم و در آخر عنوان دکمه submit را به "Upload File" تغییر می‌دهیم، فایل requestHandlers.js را به شکل زیر ویرایش کنید:

```
var querystring = require("querystring"), fs = require("fs");

function start(response, postData) {
   console.log("Request handler 'start' was called.");

   var body = '<html>'+
      '<head>'+
      '<meta http-equiv="Content-Type" '+
      'content="text/html; charset=UTF-8" />'+
      '</head>'+
      '<body>'+
      '<form action="/upload" enctype="multipart/form-data" '+
      'method="post">'+
      '<input type="file" name="upload">'+
      '<input type="submit" value="Upload file" />'+
      '</form>'+
      '</body>'+
      '</html>';

      response.writeHead(200, {"Content-Type": "text/html"});
      response.write(body);
      response.end();
}

function upload(response, postData) {
   console.log("Request handler 'upload' was called.");
   response.writeHead(200, {"Content-Type": "text/plain"});
   response.write("You've sent the text: "+
   querystring.parse(postData).text);
   response.end();
}

```

```
33 function show(response) {
34    console.log("Request handler 'show' was called.");
35    response.writeHead(200, {"Content-Type": "image/png"});
36    fs.createReadStream("/tmp/test.png").pipe(response);
37 }
38
39 exports.start = start;
40 exports.upload = upload;
41 exports.show = show;
```

بسیار عالی، مرحله بعدی مقدار پیچیده‌تر است. مشکل اول: می‌خواهیم از فایل آپلود شده در کنترلر *upload* استفاده کنیم. نیاز داریم شیء *request* را از *form.parse* عبور دهیم.

اما در حال حاضر فقط شیء *response* و آرایه *postData* را داریم. به نظر می‌رسد باید شیء *request* را از مسیر سرور به روتر و سپس به کنترلر انتقال دهیم. ممکن است راه‌حل‌های بهتری وجود داشته باشد اما این روش در حال حاضر به‌خوبی به ما کمک می‌کند.

برای آپلود فایل نیازی به آرایه *postData* نداریم و می‌توانیم آن را از سرور و کنترلر حذف کنیم؛ اما هنوز مشکل بزرگ‌تری وجود دارد: ما در حال حاضر از رویدادهای *data* شیء *request* در سرور استفاده می‌کنیم به این معناست که *form.parse* هنوز نیاز به استفاده از آن رویدادها دارد و ممکن است اطلاعات زیادی از آن‌ها دریافت نکند (Node.js از میانگیر (*buffer*) برای هیچ داده‌ای استفاده نمی‌کند).

با فایل server.js شروع خواهیم کرد – ابتدا خط *request.setEncoding* و *postData* و را از آن حذف می‌کنیم و به‌جای آن شیء *request* را به داخل *router* عبور می‌دهیم:

```
1  var http = require("http");
2  var url = require("url");
3
4  function start(route, handle) {
5     function onRequest(request, response) {
6        var pathname = url.parse(request.url).pathname;
7        console.log("Request for " + pathname + " received.");
8        route(handle, pathname, response, request);
9     }
10
11    http.createServer(onRequest).listen(8888);
12    console.log("Server has started.");
13 }
14
15 exports.start = start;
```

ما به *postData* نیاز نداریم و به‌جای آن شیء *request* را عبور می‌دهیم و باید در router.js تغییراتی ایجاد کنیم:

```
1  function route(handle, pathname, response, request) {
2     console.log("About to route a request for " + pathname);
3     if (typeof handle[pathname] === 'function') {
4        handle[pathname](response, request);
5     } else {
6        console.log("No request handler found for " + pathname);
7        response.writeHead(404, {"Content-Type": "text/html"});
8        response.write("404 Not found");
9        response.end();
10    }
11 }
```

```
12
13 exports.route = route;
```

اکنون می‌توانیم از شیء *request* در کنترلر upload استفاده کنیم و ماژول formidable به اطلاعات فایلی که در حال ذخیره شدن در پوشه tmp/ است رسیدگی خواهد کرد، اما نیاز داریم که فایل را به test.png تغییر نام دهیم. بله می‌خواهیم همه چیز را ساده نگه‌داریم و فرض کنیم تصویر PNG آپلود شده است.

در منطق تغییر نام مقداری پیچیدگی اضافی وجود دارد: پیاده‌سازی Node در سیستم‌عامل ویندوز مشابه باقی سیستم‌عامل‌ها نیست، زمانی که سعی کنید نام یک فایل را تغییر دهید و یک فایل با نام مشابه فایل جدید وجود داشته باشد موجب رخ دادن خطا می‌شود به همین دلیل باید برای رفع این خطا، فایل را حذف کنیم.

```
1  var querystring = require("querystring"),
2      fs = require("fs"),
3      formidable = require("formidable");
4
5  function start(response) {
6    console.log("Request handler 'start' was called.");
7
8    var body = '<html>'+
9      '<head>'+
10     '<meta http-equiv="Content-Type" '+
11     'content="text/html; charset=UTF-8" />'+
12     '</head>'+
13     '<body>'+
14     '<form action="/upload" enctype="multipart/form-data" '+
15     'method="post">'+
16     '<input type="file" name="upload">'+
17     '<input type="submit" value="Upload file" />'+
18     '</form>'+
19     '</body>'+
20     '</html>';
21
22     response.writeHead(200, {"Content-Type": "text/html"});
23     response.write(body);
24     response.end();
25 }
26
27 function upload(response, request) {
28   console.log("Request handler 'upload' was called.");
29
30   var form = new formidable.IncomingForm();
31   console.log("about to parse");
32   form.parse(request, function(error, fields, files) {
33     console.log("parsing done");
34
35     /* Possible error on Windows systems:
36        tried to rename to an already existing file */
37     fs.rename(files.upload.path, "/tmp/test.png", function(err) {
38       if (err) {
39         fs.unlink("/tmp/test.png");
40         fs.rename(files.upload.path, "/tmp/test.png");
41       }
42     });
43     response.writeHead(200, {"Content-Type": "text/html"});
44     response.write("received image:<br/>");
45     response.write("<img src='/show' />");
46     response.end();
47   });
48 }
```

```
49
50 function show(response) {
51    console.log("Request handler 'show' was called.");
52    response.writeHead(200, {"Content-Type": "image/png"});
53    fs.createReadStream("/tmp/test.png").pipe(response);
54 }
55
56 exports.start = start;
57 exports.upload = upload;
58 exports.show = show;
```

همین بود، سرور را مجدداً راه‌اندازی (restart) کنید و یک فایل تصویری PNG از هارد دیسک محلی انتخاب کنید و سپس اقدام به آپلود کنید. درنهایت مشاهده می‌کنید که فایل در صفحه‌نمایش داده خواهد شد.

نتیجه‌گیری و چشم‌انداز

تبریک، مأموریت ما انجام شد! توانستیم به کمک Node.js یک برنامه تحت وب ساده اما کامل بنویسیم. در این کتاب در خصوص جاوااسکریپت در سمت سرور، برنامه‌نویسی تابعی، مسدودسازی و غیر مسدودسازی فرآیندها، توابع *Callback*، رویدادها، ماژول‌های داخلی و خارجی و خیلی بیشتر صحبت کردیم.

البته مباحثی دیگری هم وجود داشتند که صحبت نکردیم: درباره پایگاه‌های داده، نوشتن و کار با واحدهای آزمایش (Unit Test)، ساخت ماژول‌های خارجی و قابل نصب به‌وسیله NPM و حتی موارد ساده‌ای مثل کنترل درخواست‌های GET.

اما هدف این کتاب افراد مبتدی بوده و نمی‌توان درباره موضوعات یک‌به‌یک بحث کرد.