Linguagens e Ambientes de Programação (2018/2019)

Teórica 21 (27/mai/2019)

XML e seu processamento em JavaScript.
JSON e MongoDB.
Scripting do lado do servidor - Node.js.
Toolkits e frameworks para a Web.
Exemplo dum framework para a Web - Express.js.

XML

XML (eXtensible Markup Language) é uma linguagem que permite representar informação de forma estruturada. Nos sistemas de informação atuais, o XML é imensamente usado, tanto para representar informação armazenada nos servidores, como para representar informação transferidas entre servidores.

Olhando para um documento XML, o aspeto geral é semelhante ao duma página Web (documento HTML). No entanto há importantes diferenças entre os dois formatos:

- O HTML serve para representar o aspeto gráfico de páginas Web usando um conjunto predefinido de **marcas**, e.g. <p>, <h1>, <table>. Permite-se que um documento HTML tenha algumas pequenas omissões, como por exemplo a falta de fechamento de algumas marcas.
- O XML serve para representar informação em geral (independentemente de considerações gráficas) e permite aos utilizadores inventarem as suas próprias marcas. As marcas são escolhidas de forma a exprimir de forma intuitiva o significado atribuído aos dados. Um documento XML tem de cumprir rigidamente a sintaxe prevista.

Em todo o caso, o formato HTML é um caso particular do formato XML, se esquecermos o detalhe do formato HTML sem um pouco menos rígido sintaticamente.

Para exemplificar, eis um pequeno documento XML com alguma informação sobre uma senhora chamada Alice e os seus filhos:

```
<person gender='F'>
    Alice
    <children>
        <person gender='M'> Andre <children></children> 5 years old </person>
        <person gender='F'> Maria <children></children> 4 years old </person>
    </children>
    35 years old
</person>
```

Neste exemplo são usadas apenas duas marcas: <person> e <children>. Quando as marcas são bem escolhidas, o documento XML torna-se autoexplicativo. Você deverá conseguir responder a estas perguntas: Qual a idade de Alice? Quantos filhos tem? Qual o nome e idade dos filhos? Ela tem algum neto?

De forma geral, um documento XML tem a estrutura duma árvore n-ária. A parte do texto correspondente à árvore completa ou qualquer subárvore chama-se um **elemento XML**.

Sintaticamente, cada **elemento XML** começa com a abertura duma **marca**, por exemplo, `<person>`, e termina com o fecho desta mesma marca, `</person>`. A abertura duma marca pode, opcionalmente, incluir alguns **atributos** com os valores correspondentes, como neste exemplo `<person gender='M'>`. Finalmente, entre a abertura e o fecho duma marca, encontra-se a **conteúdo** do elemento, que consiste em texto simples com subelementos XML intercalados. A parte de texto simples é tradicionalmente chamada de **PCDATA**. No exemplo da Alice, o conteúdo do elemento principal consiste no seguinte: o segmento de PCDATA "Alice"; em seguida, um subelemento a descrever os filhos de Alice; finalmente, o segmento de PCDATA "35 anos de idade".

Um elemento sem conteúdo, como <children></children> pode ser abreviado `<children/>`.

Observemos agora um documento XML grande, contendo uma famosa comédia escrita por William Shakespeare: A Midsummer Night's Dream.

Tecnologias associadas ao XML

Há um grande número de tecnologias associadas ao XML. Eis algumas:

- **XML schema**: Servem para descrever um **tipo de documento XML**, com um conjunto específico de marcas e restrições sintáticas no conteúdo. Por exemplo, todas as peças de teatro de Shakespeare podem ser representadas usando o mesmo conjunto de marcas e a mesma estrutura geral - neste sentido pode falar-se no tipo das peças de teatro de Shakespeare.
- **XML Namespaces**: Permitem evitar conflitos de nomes, constituindo a única forma de reutilizar definições. Assim não há problema em tomar um documento existente e estendê-lo com nova informação e novas marcas.

  Style Sheets: São mecanismos de especificação da forma como documentos XML devem ser visualizados. Estão disponíveis dois desses mecanismos: XSL - XML Style Sheets e CSS - Cascading Style Sheet. O primeiro mecanismo é o mais poderoso, e permite especificar transformações de XML em XML ou HTML (a sintaxe da linguagem de transformação é, ela própria, baseada em XML e usa o namespace predefinido "xsl"; os nós da árvore a serem tratados especificam-se usando uma linguagem de expressões chamada XPath). O segundo mecanismo permite associar a cada tipo de elemento informação de formatação, e.g. fonte, cor, margem, etc.
- **Protocolos de comunicação**: **SOAP** (Simple Object Access Protocol) consiste num protocolo de transferência de dados entre aplicações; as mensagem têm uma estrutura convencional, sendo codificadas usando o formato XML e recorrendo-se às marcas do namespace "soap". **WSDL** (Web Services Description Language) é protocolo de mais alto nível que estabelecer serviços a que os clientes acedem usando SOAP.

- **APIs para aceder a XML**: Facilitam a escrita de programas que manipulam XML. Estão disponíveis duas APIs padronizadas: **DOM** (Document Object Model) e **SAX** (Simple API for XML). A primeira é usada para construir explicitamente em memória a árvore que corresponde a um documento XML, fornecendo também primitivas para navegar e alterar essa árvore. A segunda permite processar um documento XML sem o ter de carregar em memória, funcionando com base em eventos que são ativados sempre que o parser reconhece algo: o início ou final dum elemento XML, o conteúdo de elemento XML, um erro de parsing, etc. As API DOM e SAX estão implementadas em diversas linguagens, incluindo Java (javax.xml.transform.dom, javax.xml.transform.sax).

- **Estabelecimento de relações**: **XLink** e **XPointer** são dois mecanismos para a criação de links entre documentos XML.

Processamento de XML em JavaScript

Quando, a resposta a um pedido AJAX vem em XML temos de saber processar o documento recebido. A forma de processar XML é idêntica à do processamento de HTML. Está aqui o essencial do que é preciso saber: The XML DOM.

Eis um pequeno exemplo que ilustra tudo o que é preciso saber se quisermos processar um documento XML sem o alterar:

Click Me

Código do exemplo:

```
<SCRIPT TYPE="text/javascript">
function showXML(x) {
    var str = ""
    str += "node tag = " +  x.nodeName + "\n";
    str += "parent name = " +  x.parentNode.nodeName + "\n";
    str += "number of children = " +  x.childNodes.length + "\n";
    str += "tags of the children:" + "\n";
    for( var i = 0; i < x.childNodes.length ; i++ ) {
        if( x.childNodes[i].nodeName == "#text" )
            str += "    " +  x.childNodes[i].nodeName + " (" + x.childNodes[i].textContent + ")\n";
        else
            str += "    " +  x.childNodes[i].nodeName + "\n";
    }
    str += "number of attributes = " +  x.attributes.length + "\n";
    str += "names and values of the attributes:" + "\n";
    for( var i = 0; i < x.attributes.length ; i++ ) {
        str += "    " + x.attributes[i].nodeName + " = " + x.attributes[i].textContent + "\n";
    }
    alert(str);
}

function people(x) {
    var nodes = x.getElementsByTagName("person");
    for( var i = 0 ; i < nodes.length ; i++ )
        showXML(nodes[i]);
}

function xml() {
    var parser = new DOMParser();
    var serializer = new XMLSerializer();
    var text =
        "<person gender='F'>" +
        "Alice" +
        "<children>" +
            "<person gender='M'> Andre <children></children> 5 years old </person>" +
            "<person gender='F'> Maria <children></children> 4 years old </person>" +
        "</children>" +
        "35 years old" +
        "<person>";
    var xmlDoc = parser.parseFromString(text,"text/xml");
    people(xmlDoc);
}
</SCRIPT>
<INPUT TYPE="button" VALUE="Click Me" ONCLICK="xml()">
```

JSON e MongoDB

JSON (JavaScript Object Notation) é outro formato que permite representar informação de forma estruturada. Tem o mesmo poder expressivo do XML, mas é mais simples.

O formato JSON é independente de qualquer linguagem de programação, mas usa as mesmas convenções sintáticas dos objetos literais do JavaScript. Cada objeto JSON é constituido por uma coleção de pares etiqueta/valor e as sequências são representadas por arrays.

Eis um exemplo em XML

```
<person gender='F'><name>Alice</name><age>35</age><children><person gender='M'></person><person gender='F'></person></children></person>
```

e a respestiva tradução em JSON, feita neste site:

```
{
  "person": {
    "-gender": "F",
    "name": "Alice",
    "age": "35",
    "children": {
      "person": [
        { "-gender": "M" },
        { "-gender": "F" }
      ]
```

```
        }
      }
    }
```

O sistema de base de dados MongoDB guarda documentos em formato JSON, permitindo indexar por quaisquer campos, mesmo que estejam muito aninhados no interior de subobjetos.

A naturalidade da modelação dos dados e o seu poder de indexação têm tornado o MongoDB bastante popular (apesar de tudo está em 4º lugar, atrás de concorrentes bem mais poderosos).

O MongoDB é um "document-oriented database system" e a facilidade e informalidade da sua utilização é bem-vinda em muitas aplicações Web. Contudo, se a base de dados tiver uma grande quantidade de relações e de normalização, então deverá ser preferivel usar uma base de dados relacional clássica.

Scripting do lado do servidor - Node.js

Como se disse na aula anterior, em 1993 foram introduzidos novos protocolos que passaram a permitir maior dinamismo do lado do servidor, em particular, a possibilidade de gerar dinamicamente páginas HTML. Nessa altura a programação do lado do servidor começou a ser feita essencialmente em Perl e C, mas rapidamente começaram a aparecer e a ser usadas novas linguagens. Exemplos de linguagens usadas atualmente do lado do servidor: PHP, Java, ASP, Python, Ruby, Scala, TCL, JavaScript.

Um desenvolvimento relativamente recente, foi o aparecimento do Node.js. Trata-se dum interpretador de JavaScript que vem com uma biblioteca que o torna apto a ser usado do lado do servidor. Foi criado em 2009 por Ryan Dahl. Tem tido uma adoção rápida pela comunidade da programação para a Web.

Baseia-se na ideia de usar o modelo de eventos, já conhecido do lado do cliente, também do lado do servidor. Cada pedido que chega é considerado um evento e é tratado assincronamente. Tal como no caso do cliente, existe uma **thread dos eventos** que executa continuamente um ciclo que retira e processa os eventos guardados numa fila de espera. Os eventos são tratados sequencialmente e enquanto não terminar o tratamento dum evento não se inicia o tratamento do seguinte.

Apetece criticar o uso do modelo de eventos do lado do servidor argumentando que o tratamento dum pedido pode ser demorado - realmente pode envolver a consulta duma base de dados ou o carregamento dum ficheiro grande, por exemplo. Mas isso não é problema no caso do Node.js porque ele tem o cuidado de lançar em threads auxiliares as operações de biblioteca mais demoradas, gastando pouco tempo na thread dos eventos. Por outras palavras, muitas das operações da biblioteca do Node.js funcionam de forma assíncrona. As threads auxiliares, ao terminarem, geram elas mesmo eventos de finalização que permitem ao programador aceder ao resultado da operação e prosseguir com mais ações. No exemplo abaixo, o método de biblioteca **readFile** faz o carregamento integral do ficheiro cujo nome é passado no primeiro argumento. O últmo argumento de **readFile** é a função que fará o tratamento do reaultado da leitura quando o evento de finalização ocorrer.

Como já se disse na aula anterior, o uso de lógica assíncrona tem consequências dramáticas no estilo de programação! Nos programas clássicos, o programa consegue controlar todo o fluxo de execução de forma estrita. Na programação assíncrona, escreve-se código para reagir a eventos e não há possibilidade de controlar o fluxo de execução geral. Vamos ver um exemplo.

Exemplo - Servidor de ficheiros escrito em Node.js

Primeiro mostramos o código HTML do lado do browser. Vamos testar o servidor na mesma máquina do browser e por isso usamos o domínio "localhost". Escolhemos a porta 8080 por uma questão de tradição. Note que o formulário imediatamente abaixo só funcionará com o servidor estiver ativo, algo que só acontecerá durante a própria aula.

Get file: aaa Submit

Código:

```
<FORM NAME="form1" ACTION="http://localhost:8080">
    Get file: <INPUT NAME="q" VALUE="aaa"> <INPUT TYPE="submit">
</FORM>
```

Agora o script do lado do servidor. O código é suficientemente autoexplicativo, mas pode ser necessário consultar a documentação de alguns objetos de biblioteca (http, url, path, fs). Para correr o script, basta dar o comando

```
nodejs fileserver.js
```

do lado so servidor e usar o URL `http://localhost:8080/file?q=myfile.html` do lado do browser.

Agora o código do servidor. Note que o servidor HTTP é criado no último bloco de código e ativado usando o método `listen`. Repare também que há duas funções que funcionam assincronamente: `fs.exists` e `fs.readFile`.

```
// fileserver.js

var http = require("http");
var path = require("path");
var url = require("url");
var fs = require("fs");

function handleError(error, response) {
    console.log("ERROR");
    response.writeHeader(500, {"Content-Type": "text/plain"});
    response.write(error + "\n");
    response.end();
}

function handleNotFound(response) {
    console.log("FILE NOT FOUND");
    response.writeHeader(404, {"Content-Type": "text/plain"});
    response.write("404 Not Found\n");
    response.end();
}
```

```
function handleIgnore() {
    console.log("IGNORE");
}

function handleReply(file, response) {
    console.log("OK");
    response.writeHeader(200);
    response.write(file, "binary");
    response.end();
}

var server = http.createServer(function(request, response) {
    console.log("Request " + request.url);
    var urlParsed = url.parse(request.url, true);
    if(urlParsed.query.q == undefined) {
        handleIgnore();
        return;
    }
    var localPath = urlParsed.query.q;
    var fullPath = path.join(process.cwd(),localPath);
    console.log("FullPath " + fullPath);
    fs.exists(fullPath, function(exists) {
        if(!exists)
            handleNotFound(response);
        else
            fs.readFile(fullPath, "binary",
                        function(error, file) {
                          if(error) handleError(error, response);
                          else handleReply(file, response);
                        });
    });
});
server.listen(8080);
console.log("Fileserver is running on port 8080");
```

Toolkits e framework para a Web

No mundo da programação Web, a nomenclatura usada para descrever coleções de código relacionado e reutilizável é muito fluida, mas vamos tentar introduzir aqui alguns termos, mais ou menos consensuais quando usados em determinados contextos:

- **API** - Genericamente descreve a interface externa dum conjunto de código, mas por vezes usa-se este termo para designar um módulo duma biblioteca, por exemplo quando se fala da API XMLHttpRequest.
- **Biblioteca** - Coleção de código que está disponível para ser chamado noutros programas. As operações disponíveis funcionam aproximadamente no mesmo nível de abstração. As operações podem ser muito diversas e muitas vezes não têm um objetivo específico comum, como por exemplo no caso da biblioteca do JavaScript. Mas também pode haver um objetivo bem definido, como no caso da biblioteca jQuery.
- **Toolkit** - Trata-se duma biblioteca especialmente rica, focada num domínio específico, e onde as operações trabalham a um nível de abstração elevado. Para uma biblioteca ser considerada um toolkit, é necessário que ofereça uma solução quase completa para o tipo de aplicação em causa. Exemplos: Bootstrap, Google Web Toolkit, etc.
- **Framework** - Trata-se dum sistema de programação focado num domínio específico e completamente genérico, cujas funcionalidades são adaptadas às necessidades usando código escrito pelo programador. O sistema de base já executa e prevê a maioria das necessidades do domínio específico (por vezes incluindo questões complicadas como segurança, escalabilidade e tempo de resposta), mas tem um comportamento por omissão que, muitas vezes, consiste em não fazer quase nada. Usando herança redefinem-se as partes desejadas, podendo ser chamadas funcionalidades do framework que estão disponíveis mas ainda não eram invocadas no código genérico. Por outras palavras, trata-se duma **biblioteca invertida** no sentido em que o fluxo de execução do programa é controlado pelo framework, sendo ele que chama o código do programador (exatamente o inverso do que acontece numa biblioteca normal). Exemplos: CakePHP, JavaServer Faces, Spring, Lift, Ruby on Rails, Express.js, etc.

De todas estas variantes, os frameworks possuem a curva de aprendizagem mais longa. O investimento de tempo tem de ser razoavelmente grande. Não admira: é preciso conhecer todas as abstrações usadas pelo criador do framework e o nosso código tem de se submeter a essas abstrações, implementando especializações delas. Contudo, depois de aprendido o framework, há vantagens no nível de produtividade e na qualidade das aplicações escritas.

Exemplo dum frameworks para a Web - Express.js

Introdução

Para dar uma ideia de como o uso dum framework pode tornar mais expedita a escrita dum site Web, vamos usar o framework Express.js (de momento, talvez o mais popular para Node.js) e este excelente tutorial desenvolvido por Krasimir Tsonev em 2013. Para consulta, está também aqui a documentação da API do Express.js.

Para começar, uma descrição rápida do site pretendido. O site deve permitir ao utilizador aceder a diversos tipos de informação, nas categorias "Blog", "Services", "Carrers", "Contacts". Deve também haver um painel de controlo protegido por password, para o administrador poder introduzir nova informação. É tudo.

Note que a forma essencial de usar diretamente o Node.js já foi apresentada atrás. No entanto, dentro dum framework o uso do Node.js é indireto: em muitas situações, trabalha-se a um nível de abstração mais elevado e parte da tarefa de programação consistem em configurar coisas, por exemplo especificando qual o código que deve correr em resposta a um pedido envolvendo um URL particular.

Instalação

Você é aconselhado a fazer uma instalação efetiva para ganhar um pouco de experiência e poder testar a aplicação.

Assumimos que estamos a trabalhar num sistema Linux com suporte para o sistema de instalação de pacotes **apt**, por exemplo Ubuntu ou Debian.

Passo 1 - Garantir que se tem a última versão do Node.js instalada e atualizar o gestor de pacotes **npm** do JavaScript.

```
# curl -sL https://deb.nodesource.com/setup_0.12 | sudo -E bash -
...
# sudo apt-get install -y nodejs
...
# node -v
v0.12.14
# sudo npm install npm -g
...
# npm -v
3.9.0
```

Passo 2 - Instalar o sistema de comunicações seguras Kerberos, o sistema de base de dados MongoDB, o framework Express.js e o o sistema de testes unitários jasmine-node. Note que o MongoDB é grande, ficando a ocupar cerca de 1GB no disco.

```
# sudo apt-get install libkrb5-dev mongodb
...
# sudo npm install express-generator -g
...
# sudo npm install jasmine-node -g
...
```

Passo 3 - Descarregar e instalar a aplicação já escrita:

```
# wget https://github.com/tutsplus/build-complete-website-expressjs/archive/master.zip
...
# unzip master.zip
...
# cd build-complete-website-expressjs-master/app
...
# npm install           # descarrega e instala todas as bibliotecas de que a aplicação depende
...
# jasmine-node ./tests   # corre todos os testes unitários para validar instalação.
...
```

Passo 4 - Lançar a aplicação. Numa consola fazer:

```
# cd build-complete-website-expressjs-master/app
...
# node app.js
Successfully connected to mongodb://127.0.0.1:27017
Express server listening on port 3000
GET / 200 1244ms - 2.54kb
GET /images/logo.png 304 158ms
GET /stylesheets/style.css 200 851ms - 3.25kb
GET /images/nav-bg.jpg 304 1ms
GET /images/home-title.jpg 304 1ms
GET /blog 200 29ms - 1.67kb
GET /images/logo.png 304 1ms
GET /stylesheets/style.css 304 3ms
GET /images/nav-bg.jpg 304 1ms
GET /services 200 7ms - 1.3kb
GET /images/logo.png 304 1ms
GET /stylesheets/style.css 304 4ms
GET /careers 200 5ms - 1.3kb
GET /stylesheets/style.css 304 1ms
GET /images/logo.png 304 1ms
GET /contacts 200 3ms - 1.3kb
GET /images/logo.png 304 0ms
GET /stylesheets/style.css 304 2ms
...
```

Passo 5 - Para usar a aplicação de forma normal:

```
# google-chrome http://localhost:3000
...
```

Para usar a aplicação em modeo de administrador:

```
# google-chrome http://localhost:3000/admin
...
```

Para aceder ao painel de diagnósticos do MongoDB:

```
# google-chrome http://localhost:28017
...
```

Padrão arquitectural MVC

Como é tipico de muitas aplicações Web, esta aplicação também usa o padrão arquitectural Modelo-Visualizador-Controlado (Model-View-Controller).

O padrão arquitectural MVC decompõe uma aplicação em três partes: (1) gestão dos dados, (2) lógica e (3) interface com o utilizador. Esta decomposição ajuda a tornar o programa mais compreensível e mais fácil de manter a longo prazo.

Eis alguma informação sobre os Modelos, Controladores e Visualizadores da nossa aplicação:

- **Modelos** - Cada modelo representa uma colecção de dados. Encarrega-se da gestão desses dados e da sua validação. Cada modelo está normalmente associado a uma tabela da base de dados, mas também pode estar associado a um ficheiro por exemplo.

Na nossa aplicação, para simplificar, todos os dados são do mesmo tipo e são guardados na mesma coleção. Cada registo tem os seguintes campos: title, text, picture, type. O campo type determina que é o controlador responsável pelo registo. Portanto temos apenas um modelo, a que foi dado o nome de `ContentModel`.

- **Controladores** - Cada controlador recebe os pedidos do exterior e determina como se deve responder a esses pedidos. A lógica da aplicação reside nos controladores. Costuma definir-se um controlador para cada modelo, mas por vezes há exceções, como no nosso caso em que definimos apenas um modelo.

  Na nossa aplicação há vários controladores: `Admin`, `Blog`, `Home`, `Page`. São os controladores que decidem como é que a aplicação responde a pedidos HTTP e para isso usam-se as funcionalidades de routing do Express.js.

- **Visualizadores** - Os visualizadores definem a forma de apresentar os dados ao utilizadores, mas por vezes também a forma de formatar os dados que devem ser enviados para outros serviços da Web. Um visualizador é sempre comandado por um controlador, recebe do controlador os dados, formata esses dados e envia-os para o browser.

  Na nossa aplicação só está definido um visualizador chamado `Base`. A parte de visualização ainda está pouco desenvolvida e será possível, no futuro, criar subclasses deste visualizador. O nosso visualizador produz HTML usando a primitiva de render do Express.js.

Geração automática

Como é habitual nestes fremeworks, o Express.js permite gerar o "esqueleto" duma aplicação, já com diversas coisas programadas e configuradas. No caso da nossa aplicação, o comando que Krasimir usou para criar o esqueleto da aplicação foi este:

```
# express --sessions --css less --hogan app
   create : app
   create : app/package.json
   create : app/app.js
   create : app/public
   create : app/public/images
   create : app/public/stylesheets
   create : app/public/stylesheets/style.less
   create : app/routes
   create : app/routes/index.js
   create : app/routes/users.js
   create : app/views
   create : app/views/index.hjs
   create : app/views/error.hjs
   create : app/bin
   create : app/bin/www

   install dependencies:
     $ cd app && npm install

   run the app:
     $ DEBUG=app:* npm start

   create : app/public/javascripts
```

Depois Krasimir instalou as biblitecas de que a aplicação depende usando o comando:

```
# npm install
app@0.0.0 /media/big/amd2/xtra-ubuntu-14.04/home/amd/aa/app/app
+-- body-parser@1.13.3
| +-- bytes@2.1.0
| +-- content-type@1.0.2
| +-- depd@1.0.1
| +-- http-errors@1.3.1
| | +-- inherits@2.0.1
| | `-- statuses@1.3.0
| +-- iconv-lite@0.4.11
| +-- on-finished@2.3.0
| | `-- ee-first@1.1.1
| +-- qs@4.0.0
| +-- raw-body@2.1.6
| | +-- bytes@2.3.0
| | +-- iconv-lite@0.4.13
| | `-- unpipe@1.0.0
| `-- type-is@1.6.13
|   +-- media-typer@0.3.0
|   `-- mime-types@2.1.11
|     `-- mime-db@1.23.0
+-- cookie-parser@1.3.5
| +-- cookie@0.1.3
| `-- cookie-signature@1.0.6
+-- debug@2.2.0
| `-- ms@0.7.1
+-- express@4.13.4
| +-- accepts@1.2.13
| | `-- negotiator@0.5.3
| +-- array-flatten@1.1.1
| +-- content-disposition@0.5.1
| +-- cookie@0.1.5
| +-- depd@1.1.0
| +-- escape-html@1.0.3
| +-- etag@1.7.0
| +-- finalhandler@0.4.1
| +-- fresh@0.3.0
| +-- merge-descriptors@1.0.1
| +-- methods@1.1.2
| +-- parseurl@1.3.1
| +-- path-to-regexp@0.1.7
| +-- proxy-addr@1.0.10
| | +-- forwarded@0.1.0
| | `-- ipaddr.js@1.0.5
```

```
| +-- range-parser@1.0.3
| +-- send@0.13.1
| | +-- depd@1.1.0
| | +-- destroy@1.0.4
| | +-- mime@1.3.4
| | `-- statuses@1.2.1
| +-- serve-static@1.10.2
| +-- utils-merge@1.0.0
| `-- vary@1.0.1
+-- hjs@0.0.6
| `-- hogan.js@3.0.2
|   +-- mkdirp@0.3.0
|   `-- nopt@1.0.10
|     `-- abbrev@1.0.7
+-- less-middleware@1.0.4
| +-- less@1.7.5
| | +-- clean-css@2.2.23
| | | `-- commander@2.2.0
| | +-- graceful-fs@3.0.8
| | +-- mime@1.2.11
| | +-- mkdirp@0.5.1
| | | `-- minimist@0.0.8
| | +-- request@2.40.0
| | | +-- aws-sign2@0.5.0
| | | +-- forever-agent@0.5.2
| | | +-- form-data@0.1.4
| | | | +-- async@0.9.2
| | | | +-- combined-stream@0.0.7
| | | | | `-- delayed-stream@0.0.5
| | | | `-- mime@1.2.11
| | | +-- hawk@1.1.1
| | | | +-- boom@0.4.2
| | | | +-- cryptiles@0.2.2
| | | | +-- hoek@0.9.1
| | | | `-- sntp@0.2.4
| | | +-- http-signature@0.10.1
| | | | +-- asn1@0.1.11
| | | | +-- assert-plus@0.1.5
| | | | `-- ctype@0.5.3
| | | +-- json-stringify-safe@5.0.1
| | | +-- mime-types@1.0.2
| | | +-- node-uuid@1.4.7
| | | +-- oauth-sign@0.3.0
| | | +-- qs@1.0.2
| | | +-- stringstream@0.0.5
| | | +-- tough-cookie@2.2.2
| | | `-- tunnel-agent@0.4.3
| | `-- source-map@0.1.43
| |   `-- amdefine@1.0.0
| +-- mkdirp@0.3.5
| `-- node.extend@1.0.10
|   `-- is@0.3.0
+-- morgan@1.6.1
| +-- basic-auth@1.0.4
| `-- on-headers@1.0.1
`-- serve-favicon@2.3.0
```

O conteúdo do ficheiro `/package.json` ficou o seguinte:

```
{
  "name": "app",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "hjs": "~0.0.6",
    "less-middleware": "1.0.x",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

Para correr a aplicação vazia, fazer:

```
# npm start
```

e o seguinte link - http://localhost:3000 - permite ver o que a aplicação vazia faz.

Para aprender mais sobre este exemplo, agora só mesmo estudando o tutorial, consultando ocasionalmente a documentação da API do Express.js.

#80