4 - Express Tutorial Part 2: Creating a skeleton website

Este segundo artículo de nuestro Tutorial Express muestra cómo puede crear un "esqueleto" para un proyecto de un sitio web que luego puede completar con rutas, plantillas/vistas, y llamadas a base de datos específicas del sitio.

Visión General

Este artículo muestra cómo puede crear un sitio web "esqueleto" usando la herramienta Generador de Aplicaciones Express, que luego puede completar con rutas, vistas/plantillas, y llamadas a base de datos específicas del sitio. En este caso usaremos la herramienta para crear el framework para nuestro website Local Library, al que luego agregaremos todo el código que el sitio necesite. El proceso es extremadamente simple, requiriendo sólo que se invoque el generador en la línea de comandos con un nombre para el nuevo proyecto, opcionalmente especificando también el motor de plantillas y el generador de CSS a utilizar.

Las siguientes secciones muestran cómo puede llamar al generador de aplicaciones, y proporcionan una pequeña explicación sobre las diferentes opciones para vistas y CSS. También explicaremos como está estructurado el esqueleto del sitio web. Al final, mostraremos como puede ejecutar el sitio web para verificar que funciona.

Nota: El *Generador de Aplicaciones Express* no es el único generador para aplicaciones Express, y el proyecto generado no es la única forma viable para estructurar sus archivos y directorios. El sitio generado, sin embargo, tiene una estructura modular que es fácil de extender y comprender. Para información sobre una *mínima* aplicación Express, vea el Ejemplo Hello world (Express docs).

Usando el generador de aplicaciones

Ya debe haber instalado el generador como parte de Configurar un entorno de desarrollo de Node. Como un rápido recordatorio, la herramienta generadora se instala para todos los sitios usando el manejador de paquetes NPM, como se muestra:

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

El generador tiene un número de opciones, las cuales puede observar en la línea de comandos usando el comando `--help` (o bien `-h`):

```
> express --help


  Usage: express [options] [dir]


  Options:


    -h, --help          output usage information
        --version       output the version number
    -e, --ejs           add ejs engine support
        --pug           add pug engine support
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -v, --view <engine>  add view <engine> support
(ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
    -c, --css <engine>   add stylesheet <engine> support
(less|stylus|compass|sass) (defaults to plain css)
        --git           add .gitignore
    -f, --force         force on non-empty directory
```

Simplemente puede especificar `express` para crear un proyecto dentro del directorio actual usando el motor de plantillas *Jade* y CSS plano (si especifica un nombre de directorio entonces el proyecto será creado en un subdirectorio con ese nombre).

```
express
```

También puede seleccionar el motor de plantillas para las vistas usando –`view` y/o un motor generador de CSS usando `--css`.

Nota: Las otras opciones para elegir motores de plantillas (e.g. `--hogan`, `--ejs`, `--hbs` etc.) están descontinuadas. Use `--view` (o bien `-v`)!

¿Cuál motor de vistas debo usar?

El *Generador de Aplicaciones Express* le permite configurar un número de populares motores de plantillas, incluyendo EJS, Hbs, Pug (Jade), Twig, y Vash, aunque si no se especifica una opcion de vista, selecciona Jade por defecto. Express puede soportar un gran número de motores de plantillas aquí una lista.

Nota: Si quiere usar un motor de plantillas que no es soportado por el generador entonces vea el artículo Usando motores de plantillas con Express (Express docs) y la documentación de su motor de plantillas.

Generalmente hablando debe seleccionar un motor de plantillas que le brinde toda la funcionalidad que necesite y le permita ser productivo rápidamente — o en otras palabras, en la misma forma en que selecciona cualquier otro componente. Alguna de las cosas a considerar cuando se comparan motores de plantillas:

- Tiempo de productividad — Si su equipo ya tiene experiencia con un lenguaje de plantillas entonces es probable que sean más productivos usando ese lenguaje. Si no, debería considerar la curva de aprendizaje relativa del motor de plantillas candidato.
- Popularidad y actividad — Revise la popularidad del motor y si tiene una comunidad activa. Es importante obtener soporte para el motor cuando tenga problemas durante la vida útil del sitio web.
- Estilo — Algunos motores de plantillas usan marcas específicas para indicar inserción de contenido dentro del HTML "ordinario", mientras que otros construyen el HTML usando una sintaxis diferente (por ejemplo, usando indentación (sangría) y nombres de bloque).
- Tiempo Renderizado/desempeño.
- Características — debe considerar si los motores que elija poseen las siguientes características disponibles:
    - Herencia del diseño: Le permite definir una plantilla base y luego "heredar" sólo las partes que desea que sean diferentes para una página particular. Típicamente esto es un mejor enfoque que construir plantillas incluyendo un número de componentes requeridos, construyéndolas desde cero cada vez.

- Soporte para incluir: Le permite construir plantillas incluyendo otras plantillas.
- Control conciso de la sintaxis de variables y ciclos.
- Habilidad para filtrar valores de variables a nivel de las plantillas (ej. convertir variables en mayúsculas, o darle formato a una fecha).
- Habilidad para generar formatos de salida distintos al HTML (ej. JSON o XML).
- Soporte para operaciones asíncronas y de transmisión.
- Pueden ser usadas tanto en el cliente como en el servidor. Si un motor de plantillas puede ser usado del lado del cliente esto da la posibilidad de servir datos y tener todo o la mayoría del renderizado del lado del cliente.

Tip: En Internet hay muchos recursos que le ayudarán a comparar diferentes opciones.

Para este proyecto usaremos el motor de plantillas Pug (este es el recientemente renombrado motor Jade), ya que es de los más populares lenguajes de plantillas Express/JavaScript y es soportado por el generador por defecto.

¿Cuál motor de hojas de estilo CSS debería usar?

El *Generador de Aplicaciones Express* le permite crear un proyecto que puede usar los más comunes motores de hojas de estilos CSS: LESS, SASS, Compass, Stylus.

Nota: CSS tiene algunas limitaciones que dificultan ciertas tareas. Los motores de hojas de estilos CSS le permiten usar una sintaxis más poderosa para definir su CSS, y luego compilar la definición en texto plano para su uso en los navegadores .

Como los motores de plantillas, debería usar el motor CSS que le permita a su equipo ser más productivo. Para este proyecto usaremos CSS ordinario (opción por defecto) ya que nuestros requerimientos no son lo suficientemente complicados para justificar el uso de un motor CSS.

¿Cuál base de datos debería usar?

El código generado no usa o incluye ninguna base de datos. Las aplicaciones *Express* pueden usar cualquier mecanismo de bases de datos soportado por

Node (*Express* por sí mismo no define ningún comportamiento o requerimiento para el manejo de bases de datos).

Discutiremos la integración con una base de datos en un posterior artículo.

Creando el proyecto

Para el ejemplo que vamos a crear la app *Local Library*, crearemos un proyecto llamado *express-locallibrary-tutorial usando la librería de plantillas Pug* y ningún motor CSS.

Primero navegue a donde quiera crear el proyecto y luego ejecute el *Generador de Aplicaciones Express en la línea de comandos como se muestra*:

```
express express-locallibrary-tutorial --view=pug
```

El generador creará (y listará) los archivos del proyecto.

```
   create : express-locallibrary-tutorial

   create : express-locallibrary-tutorial/package.json

   create : express-locallibrary-tutorial/app.js

   create : express-locallibrary-tutorial/public/images

   create : express-locallibrary-tutorial/public

   create : express-locallibrary-tutorial/public/stylesheets

   create : express-locallibrary-tutorial/public/stylesheets/style.css

   create : express-locallibrary-tutorial/public/javascripts

   create : express-locallibrary-tutorial/routes

   create : express-locallibrary-tutorial/routes/index.js

   create : express-locallibrary-tutorial/routes/users.js

   create : express-locallibrary-tutorial/views

   create : express-locallibrary-tutorial/views/index.pug
```

```
   create : express-locallibrary-tutorial/views/layout.pug

   create : express-locallibrary-tutorial/views/error.pug

   create : express-locallibrary-tutorial/bin

   create : express-locallibrary-tutorial/bin/www



   install dependencies:

     > cd express-locallibrary-tutorial && npm install



   run the app:

     > SET DEBUG=express-locallibrary-tutorial:* & npm start
```

Al final de la lista el generador mostrará instrucciones sobre como instalar las dependencias necesarias (mostradas en el archivo **package.json**) y luego como ejecutar la aplicación (las instrucciones anteriores son para Windows; en Linux/macOS serán ligeramente diferentes).

Ejecutando el esqueleto del sitio web

En este punto tenemos un esqueleto completo de nuestro proyecto. El sitio web no hace mucho actualmente, pero es bueno ejecutarlo para ver cómo funciona.

1. Primero instale las dependencias (el comando `npm nstall` recuperará todas las dependencias listadas en el archivo **package.json** del proyecto).

```
cd express-locallibrary-tutorial

npm install
```

2. Luego ejecute la aplicación.
    - En Windows, use este comando (se pueden ejecutar por separado si el & os da algún problema.):

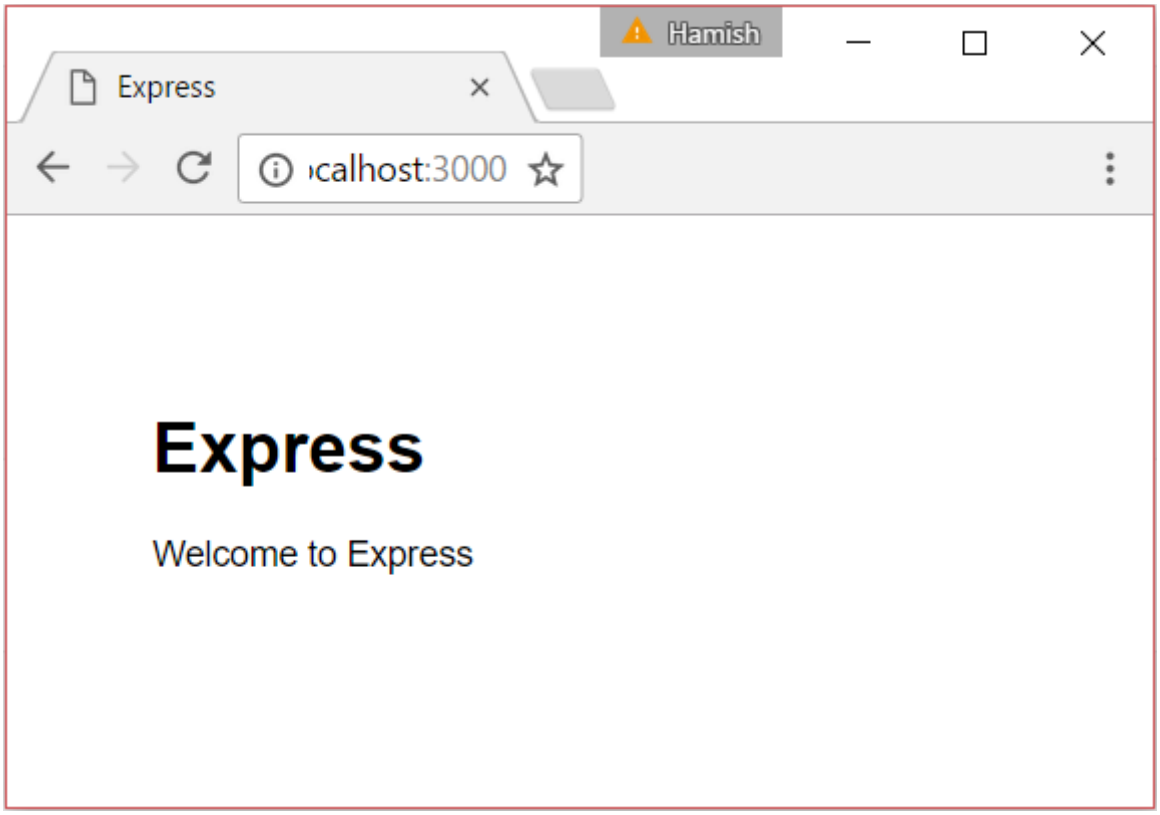
```
SET DEBUG=express-locallibrary-tutorial:* & npm start
```

- En macOS o Linux, use este comando:

```
DEBUG=express-locallibrary-tutorial:* npm start
```

3. Luego cargue en su navegador http://localhost:3000/ para acceder a la aplicación.

Debería ver una página parecida a esta:



Tiene una aplicación Express funcional, ejecutándose en *localhost:3000*.

Nota: También podría ejecutar la app usando el comando `npm start`. Especificado la variable DEBUG como se muestra habilita el logging/debugging por consola. Por ejemplo, cuando visite la página mostrada arriba verá la información de depuración como esta:

```
>SET DEBUG=express-locallibrary-tutorial:* & npm start



> express-locallibrary-tutorial@0.0.0 start D:\express-locallibrary-
tutorial
```

```
> node ./bin/www

  express-locallibrary-tutorial:server Listening on port 3000 +0ms

GET / 200 288.474 ms - 170

GET /stylesheets/style.css 200 5.799 ms - 111

GET /favicon.ico 404 34.134 ms - 1335
```

Habilite el reinicio del servidor cuando los archivos sean modificados

Cualquier cambio que le haga a su sitio web Express no será visible hasta que reinicie el servidor. rápidamente, tener que detener y reiniciar el servidor cada vez que hacemos un cambio, se vuelve irritante, así que es beneficioso tomarse un tiempo y automatizar el reinicio del servidor cuando sea necesario.

Una de las herramientas más sencillas para este propósito es nodemon. Éste usualmente se instala globalmente (ya que es una "herramienta"), pero aquí lo instalaremos y usaremos localmente como una dependencia de desarrollo, así cualquier desarrollador que esté trabajando con el proyecto lo obtendrá automáticamente cuando instale la aplicación. Use el siguiente comando en el directorio raíz del esqueleto del proyecto:

```
npm install --save-dev nodemon
```

Si abre el archivo **package.json** de su proyecto verá una nueva sección con esta dependencia:

```
  "devDependencies": {

    "nodemon": "^1.14.11"

  }
```

Debido a que la herramienta no fue instalada globalmente no podemos ejecutarla desde la línea de comandos (a menos que la agreguemos a la ruta) pero podemos llamarla desde un script NPM porque NPM sabe todo sobre los paquetes instalados. Busque la sección `scripts` de su package.json. Inicialmente contendrá una línea, la cual comienza con `"start"`. Actualícela colocando una coma al final de la línea, y agregue la línea `"devstart"` mostrada abajo:

```
  "scripts": {

    "start": "node ./bin/www",

    "devstart": "nodemon ./bin/www"

  },
```

Ahora podemos iniciar el servidor casi exactamente como antes, pero especificando el comando devstart:

- En Windows, use este comando:

```
SET DEBUG=express-locallibrary-tutorial:* & npm run devstart
```

- En macOS or Linux, use este comando:

```
DEBUG=express-locallibrary-tutorial:* npm run devstart
```

Nota: Ahora si modifica cualquier archivo del proyecto el servidor se reiniciará (o lo puede reiniciar `rs` en la consola de comandos en cualquier momento). Aún necesitará recargar el navegador para refrescar la página.

Ahora tendremos que llamar "`npm run <nombre del script>`" en vez de `npm start`, porque "start" es actualmente un comando NPM que es mapeado al nombre del script. Podríamos haber reemplazado el comando en el script *start* pero sólo queremos usar *nodemon* durante el desarrollo, así que tiene sentido crear un nuevo script para este comando.

El proyecto generado

Observemos el proyecto que hemos creado.

Estructura del directorio

El proyecto generado, ahora que ha instalado las dependencias, tiene la siguiente estructura de archivos (los archivos son los elementos que **no** están precedidos con "/"). El archivo **package.json** define las dependencias de la aplicación y otra información. También define un script de inicio que es el punto de entrada de la aplicación, el archivo JavaScript **/bin/www**. Éste establece algunos de los manejadores de error de la aplicación y luego carga el

archivo **app.js** para que haga el resto del trabajo. Las rutas se almacenan en módulos separados en el directorio **/routes**. las plantillas se almacenan en el directorio /**views**.

```
/express-locallibrary-tutorial
    app.js
    /bin
        www
    package.json
    /node_modules
        [about 4,500 subdirectories and files]
    /public
        /images
        /javascripts
        /stylesheets
            style.css
    /routes
        index.js
        users.js
    /views
        error.pug
        index.pug
        layout.pug
```

Las siguientes secciones describen los archivos con más detalle.

package.json

El archivo **package.json** define las dependencias de la aplicación y otra información:

```
{

  "name": "express-locallibrary-tutorial",

  "version": "0.0.0",

  "private": true,

  "scripts": {

    "start": "node ./bin/www",

    "devstart": "nodemon ./bin/www"

  },

  "dependencies": {

    "body-parser": "~1.18.2",

    "cookie-parser": "~1.4.3",

    "debug": "~2.6.9",

    "express": "~4.16.2",

    "morgan": "~1.9.0",

    "pug": "~2.0.0-rc.4",

    "serve-favicon": "~2.4.5"

  },

  "devDependencies": {

    "nodemon": "^1.14.11"

  }
```

```
}
```

Las dependencias incluyen el paquete *express* y los paquetes para el motor de plantillas elegido (*pug*). Adicionalmente, tenemos los siguientes paquetes que son útiles en muchas aplicaciones web:

- body-parser: Esto analiza la parte del cuerpo de una solicitud HTTP entrante y facilita la extracción de diferentes partes de la información contenida. Por ejemplo, puede usar esto para leer los parámetros POST.
- cookie-parser: Se utiliza para analizar el encabezado de la cookie y rellenar req.cookies (esencialmente proporciona un método conveniente para acceder a la información de la cookie).
- debug: Una pequeña utilidad de depuración de node modelada a partir de la técnica de depuración del núcleo de node.
- morgan: Un middleware registrador de solicitudes HTTP para node.
- serve-favicon: Middleware de node para servir un favicon (este es el icono utilizado para representar el sitio dentro de la pestaña del navegador, marcadores, etc.).

La sección de scripts define un script de "*start*", que es lo que invocamos cuando llamamos a npm start para iniciar el servidor. Desde la definición del script, puede ver que esto realmente inicia el archivo JavaScript **./bin/www** con *node*. También define un script "*devstart*", que invocamos cuando llamamos a npm run devstart en su lugar. Esto inicia el mismo archivo **./bin/www**, pero con *nodemon* en lugar de node.

```
  "scripts": {

    "start": "node ./bin/www",

    "devstart": "nodemon ./bin/www"

  },
```

www file

El archivo **/bin/www** es el punto de entrada de la aplicación. Lo primero que hace es require () el punto de entrada de la aplicación "real" (**app.js**, en la raíz del proyecto) que configura y devuelve el objeto de la aplicación express ().

```
#!/usr/bin/env node
```

```
/**

 * Module dependencies.

 */



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

Note: `require()` es una función de node global que se usa para importar módulos en el archivo actual. Aquí especificamos el módulo app.js utilizando una ruta relativa y omitiendo la extensión de archivo opcional (.js).

El resto del código en este archivo configura un servidor HTTP de node con la aplicación configurada en un puerto específico (definido en una variable de entorno o 3000 si la variable no está definida), y comienza a escuchar e informar errores y conexiones del servidor. Por ahora no necesita saber nada más sobre el código (todo en este archivo es "repetitivo"), pero siéntase libre de revisarlo si está interesado.

app.js

Este archivo crea un objeto de aplicación rápida (aplicación denominada, por convención), configura la aplicación con varias configuraciones y middleware, y luego exporta la aplicación desde el módulo. El siguiente código muestra solo las partes del archivo que crean y exportan el objeto de la aplicación:

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

var app = express();

...

module.exports = app;
```

De vuelta en el archivo de punto de entrada **www** anterior, es este objeto module.exports que se proporciona al llamante cuando se importa este archivo.

Permite trabajar a través del archivo **app.js** en detalle. Primero importamos algunas bibliotecas de node útiles en el archivo usando require (), incluyendo *express*, *serve-favicon*, *morgan*, *cookie-parser* y *body-parser* que previamente

descargamos para nuestra aplicación usando NPM; y *path*, que es una biblioteca central de nodos para analizar rutas de archivos y directorios.

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

var path = require('path');

var favicon = require('serve-favicon');

var logger = require('morgan');

var cookieParser = require('cookie-parser');

var bodyParser = require('body-parser');
```

Luego require () modules de nuestro directorio de rutas. Estos modules/files contienen código para manejar conjuntos particulares de "routes" relacionadas (rutas URL). Cuando extendemos la aplicación esqueleto, por ejemplo, para enumerar todos los libros de la biblioteca, agregaremos un nuevo archivo para tratar las rutas relacionadas con los libros.

```
var index = require('./routes/index');

var users = require('./routes/users');
```

Note: En este punto, acabamos de importar el módulo; aún no hemos utilizado sus rutas (esto sucede un poco más abajo en el archivo).

Luego creamos el objeto `app` usando nuestro módulo *express* importado, y luego lo usamos para configurar el motor de vista (plantilla). Hay dos partes para configurar el motor. Primero establecemos el valor `views` ' ' para especificar la carpeta donde se almacenarán las plantillas (en este caso, la subcarpeta ***/ vistas***). Luego establecemos el valor `view engine` ' ' para especificar la biblioteca de plantillas (en este caso, "pug").

```
var app = express();



// view engine setup

app.set('views', path.join(__dirname, 'views'));

app.set('view engine', 'pug');
```

El siguiente conjunto de funciones llama `app.use()`para agregar las bibliotecas de *middleware* a la cadena de manejo de solicitudes. Además de las bibliotecas de terceros que importamos anteriormente, usamos el middleware `express.static` para que *Express* sirva todos los archivos estáticos en el directorio **/ public** en la raíz del proyecto.

```
// uncomment after placing your favicon in /public

//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: false }));

app.use(cookieParser());

app.use(express.static(path.join(__dirname, 'public')));
```

Ahora que todo el resto de middleware está configurado, agregamos nuestro código de manejo de rutas (previamente importado) a la cadena de manejo de solicitudes. El código importado definirá rutas particulares para las diferentes *partes* del sitio:

```
app.use('/', index);

app.use('/users', users);
```

Nota: Las rutas especificadas anteriormente ('/' y ' `/users'`) se tratan como un prefijo a las rutas definidas en los archivos importados. Entonces, por ejemplo, si el módulo de **usuarios** importados define una ruta para `/profile`, accedería a esa ruta en `/users/profile`. Hablaremos más sobre rutas en un artículo posterior.

El último middleware del archivo agrega métodos de manejo para errores y respuestas HTTP 404.

```
// catch 404 and forward to error handler

app.use(function(req, res, next) {
```

```
  var err = new Error('Not Found');

  err.status = 404;

  next(err);

});



// error handler

app.use(function(err, req, res, next) {

  // set locals, only providing error in development

  res.locals.message = err.message;

  res.locals.error = req.app.get('env') === 'development' ? err : {};



  // render the error page

  res.status(err.status || 500);

  res.render('error');

});
```

El objeto de la aplicación Express (aplicación) ahora está completamente configurado. El último paso es agregarlo a las exportaciones del módulo (esto es lo que permite que sea importado por **/ bin / www**).

```
module.exports = app;
```

Rutas

El archivo de ruta **/routes/users.js** se muestra a continuación (los archivos de ruta comparten una estructura similar, por lo que no necesitamos mostrar también **index.js**). Primero carga el módulo *express* y lo usa para obtener un objeto `express.Router`. Luego especifica una ruta en ese objeto y, por último, exporta el enrutador desde el módulo (esto es lo que permite que el archivo se importe a **app.js**).

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

var router = express.Router();



/* GET users listing. */

router.get('/', function(req, res, next) {

  res.send('respond with a resource');

});



module.exports = router;
```

La ruta define una devolución de llamada que se invocará siempre que se detecte una solicitud HTTP `GET` con el patrón correcto. El patrón coincidente es la ruta especificada cuando se importa el módulo (' `/users`') más lo que esté definido en este archivo (' `/`'). En otras palabras, esta ruta se utilizará cuando se reciba una URL de `/users/`.

Consejo: Pruebe esto ejecutando el servidor con nodo y visitando la URL en su navegador:http: // localhost: 3000 / users /. Debería ver un mensaje: 'responder con un recurso'.

Una cosa de interés anterior es que la función de devolución de llamada tiene el tercer argumento ' `next`' y, por lo tanto, es una función de middleware en lugar de una simple devolución de llamada de ruta. Si bien el código no usa actualmente el argumento `next`, puede ser útil en el futuro si desea agregar varios controladores de ruta a la ruta `'/'`.

Vistas (plantillas)

Las vistas (plantillas) se almacenan en el directorio **/ views** (como se especifica en **app.js**) y se les da la extensión de archivo **.pug** .

El método `Response.render()` se utiliza para representar una plantilla específica junto con los valores de las variables nombradas que se pasan en un objeto y luego enviar el resultado como respuesta. En el siguiente código de **/routes/index.js** puede ver cómo esa ruta muestra una respuesta usando la plantilla "index" pasando la variable de plantilla "title".

```
/* GET home page. */

router.get('/', function(req, res) {

  res.render('index', { title: 'Express' });

});
```

La plantilla correspondiente para la ruta anterior se proporciona a continuación (**index.pug**). Hablaremos más sobre la sintaxis más adelante. Todo lo que necesita saber por ahora es que la variable `title` (con valor `'Express'`) se inserta donde se especifica en la plantilla.

```
extends layout



block content

  h1= title

  p Welcome to #{title}
```

Resumen

Ahora ha creado un proyecto de sitio web esqueleto para la biblioteca local y ha verificado que se ejecuta utilizando *node* . Lo más importante es que también comprende cómo está estructurado el proyecto, por lo que tiene una buena idea de dónde debemos realizar cambios para agregar rutas y vistas para nuestra biblioteca local.

A continuación, comenzaremos a modificar el esqueleto para que funcione como un sitio web de biblioteca.