Proyecto de la asignatura CORE

Desarrollo de un Blog

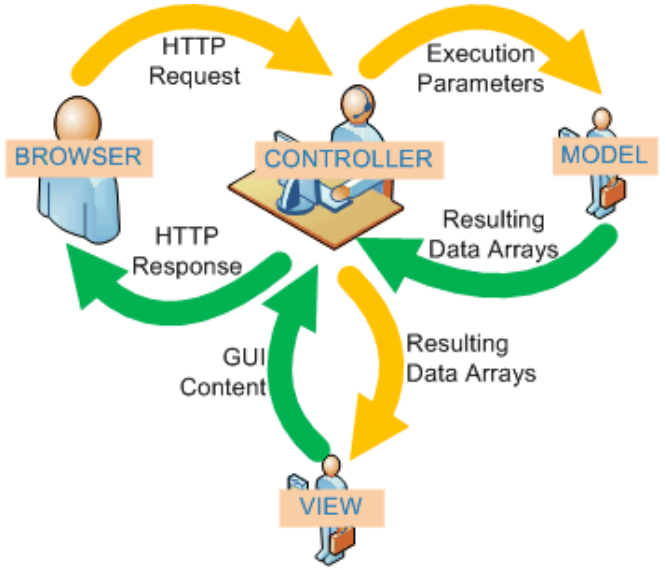
Tema 2: Crear los Posts.

CORE 2013-2014

ver: 2014-04-28

Índice

- Documentación.
- Crear el recurso REST posts: los artículos publicados.
  - Diseñar API REST.
  - Patrón MVC:
    - Crear el módulo controlador
      - con los middlewares de las rutas.
    - Crear el modelo.
      - Con Sequelize y SQLite.
    - Crear las vistas.
- Desplegar en Heroku.



Documentación

- La documentación de los temas anteriores.
  - nodejs, express
- sequelize
  - `http://www.sequelizejs.com`
- El ejemplo está disponible en la rama **tema2**.

  `http://github.com/CORE-UPM/blog_2014`

API REST

Diseñar Rutas REST para Posts

- Las rutas que implementaremos seguirán el mismo estilo que usan los frameworks Sinatra, Rails

Método HTTP	URL	Acción
`GET`	`/posts`	`postController.index`
`GET`	`/posts/new`	`postController.new`
`GET`	`/posts/:postid`	`postController.show`
`POST`	`/posts`	`postController.create`
`GET`	`/posts/:postid/edit`	`postController.edit`
`PUT`	`/posts/:postid`	`postController.update`
`DELETE`	`/posts/:postid`	`postController.destroy`

Notas

Crearemos el recurso **REST** de los posts al estilo **rails**.

Hay que crear las rutas. Una ruta declara que método hay que ejecutar cuando llega una determinada petición HTTP.
Crearemos las siguientes rutas:

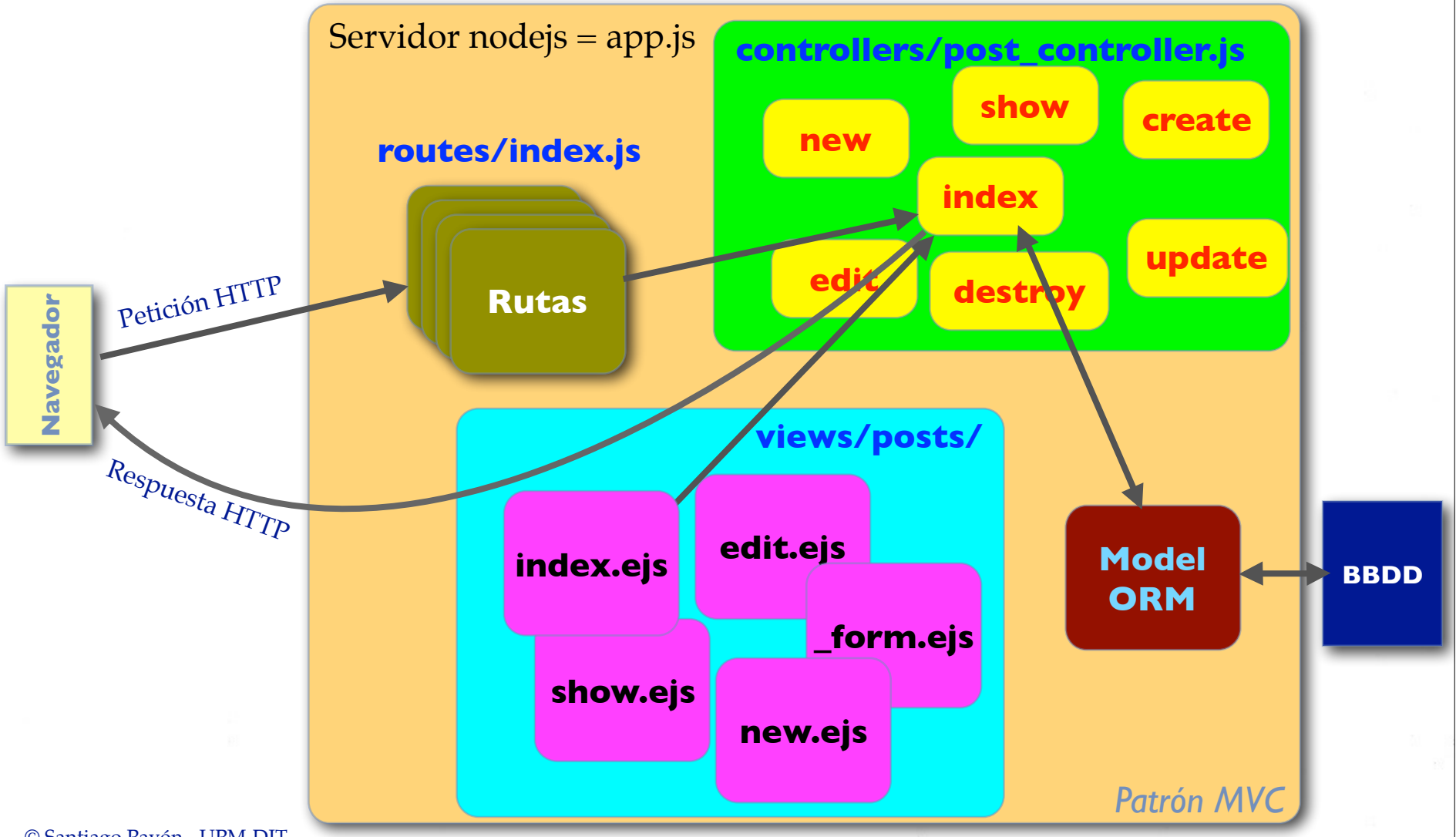
- **`GET /posts`** -> ejecuta la función **index** que devuelve un listado de todos los posts existentes.
- **`GET /posts/new`** -> ejecuta la función **new** que devuelve un formulario para crear un nuevo post.
- **`GET /posts/33`** -> ejecuta la función **show** que muestra el post con **id** 33.
- **`POST /posts`** -> ejecuta la función **create** que crea un nuevo objeto post.
- **`GET /posts/33/edit`** -> ejecuta la función **edit** que devuelve un formulario para editar el post con **id** 33.
- **`PUT /posts/33`** -> ejecuta la función **update** que actualiza los campos del post con **id** 33.
- **`DELETE /posts/33`** -> ejecuta la función **destroy** que elimina el post con **id** 33.

Estas rutas las definiremos en el módulo **`routes/index.js`**.
Todas las funciones anteriores se agruparán en un módulo node.js, que las exportará. Así al requerir el módulo, obtendremos un objeto cuyos atributos (llamados **index**, **new**, **show**, **create**, **edit**, **update** y **destroy**) apuntarán a las funciones antes descritas. Este módulo es lo que llamamos el controlador de los Posts. Lo crearemos en el directorio **`controllers/post_controller.js`**.

Las operaciones que se hacen en estas funciones son obtener los parámetros de las peticiones HTTP (**req.query**, **req.param**, **req.body**, ...), buscar objetos en la base de datos (**sequelize find**, **sequelize findAll**), comprobar si los valores son correctos (**sequelize validate**), salvar objetos en la base de datos (**sequelize save**), borrar objetos de la base de datos (**sequelize destroy**), etc.

Estas funciones también crearán los datos que hay que mostrar en las páginas HTML que se devuelven al navegador. Para presentar (**res.render**) estos datos se usarán plantillas EJS parametrizadas con variables. Estas variables se sustituyen por los datos concretos a mostrar.

Esto es lo que hay que hacer:



Notas

Las plantillas EJS las crearemos en el directorio **views/posts** y son las siguientes:

- **new.ejs** para mostrar el formulario de creación de un post.
  - Parámetro: un objeto **Post** llamado **post**.
- **edit.ejs** para mostrar el formulario de edición de un post existente.
  - Parámetro: un objeto **Post** llamado **post**.
- **index.ejs** para mostrar el listado de todos los post existentes.
  - Parámetro: un **array** de objetos **Post** llamado **posts**.
- **show.ejs** para mostrar un único post.
  - Parámetro: un objeto **Post** llamado **post**.
- **_form.ejs** Dado que los formularios creados por **new.ejs** y **edit.ejs** tienen el mismo contenido, crearemos **_form.ejs** donde se ha extraído la parte común de ambos formularios. Este fichero se incluye en **new.ejs** y en **edit.ejs** usando un **include**.
  - Parámetro: un objeto **Post** llamado **post**.

Las peticiones que hacen los navegadores a los servidores web sólo usan los métodos **GET** y **POST**. Para soportar otros métodos HTTP, se crea un parámetro, llamado **_method**, cuyo valor es el método HTTP que se desea usar.

Controlador

- El módulo controlador de los posts se implementará en el fichero:

  `controllers/post_controller.js`

- Exportará los middlewares usados en la creación de las rutas.

controllers/post_controller.js

```
// Autoload :postid
exports.load = function(req, res, next, id) {
   req.post = {id: id, title:"hola mundo"};
   next(); // Ir al siguiente middleware
};

// GET /posts
exports.index = function(req, res, next) {
   res.end("Listado de todos los Posts.");
};

// GET /posts/33
exports.show = function(req, res, next) {
   res.end("Ver el post " + req.params['postid'] + ".");
};

// GET /posts/new
exports.new = function(req, res, next) {
   res.end("Obtener formulario para crear un nuevo posts.");
};

// POST /posts
exports.create = function(req, res, next) {
   res.end("Crear un nuevo post.");
};

// GET /posts/33/edit
exports.edit = function(req, res, next) {
   res.end("Obtener formulario para editar el post " + req.params['postid'] + ".");
};

// PUT /posts/33
exports.update = function(req, res, next) {
   res.end("Actualizar el post " + req.params['postid'] + ".");
};

// DELETE /posts/33
exports.destroy = function(req, res, next) {
   res.end("Borrar el post " + req.params['postid'] + ".");
};
```

De momento estas funciones sólo devuelven un string en la respuesta HTTP.

Crear las Rutas

- Editamos el fichero **routes/index.js**.
  - Requerimos el módulo controlador de los Posts que contiene los middlewares.
    - Añadimos:

```
var postController = require('../controllers/post_controller');
```

  - Y definimos las rutas añadiendo:

```
router.param('postid',postController.load);  // autoload :postid

router.get('/posts', postController.index);
router.get('/posts/new', postController.new);
router.get('/posts/:postid([0-9]+)', postController.show);
router.post('/posts', postController.create);
router.get('/posts/:postid([0-9]+)/edit', postController.edit);
router.put('/posts/:postid([0-9]+)', postController.update);
router.delete('/posts/:postid([0-9]+)', postController.destroy);
```

Instalar **connect.methodOverride**

- Para soportar los métodos **PUT** y **DELETE** del protocolo HTTP hay que instalar el middleware **methodOverride** proporcionado por el módulo **connect**.
- Instalamos el módulo ejecutamos:

```
$ npm install --save connect
```

- Y actualizamos **apps.js** para cargar el middleware:

```
. . .
var connect = require('connect');
. . .
app.use(connect.methodOverride());
. . .
```

```
var express = require('express');
var path = require('path');
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var partials = require('express-partials');
var connect = require('connect');
var routes = require('./routes/index');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(connect.methodOverride());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use(partials());

app.use('/', routes);

. . .
```

app.js

Probar el Controlador

- Para probar las rutas que hemos definido ejecutamos en un terminal los siguientes comandos:

```
$ curl localhost:3000/posts
$ curl localhost:3000/posts/new
$ curl localhost:3000/posts/66
$ curl -d _method=post localhost:3000/posts
$ curl localhost:3000/posts/66/edit
$ curl -d _method=put localhost:3000/posts/66
$ curl -d _method=delete localhost:3000/posts/66
```

  - Nota: el middleware **methodOverride** se encarga de modificar el método de la petición HTTP por el valor especificado en la variable **_method**.

El Modelo

SQLite y Sequelize

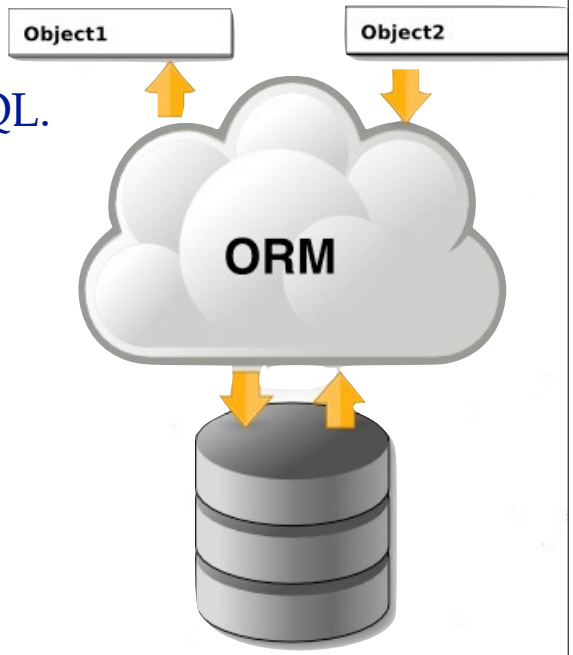
El Modelo

- Necesitamos dos cosas:

1. Una base de datos para almacenar los datos.
   - En local usaremos **SQLite**.
   - En Heroku usaremos **Postgres**.
2. Un **ORM** (Object-Relational Mapping) para acceder a la base de datos.
   - Usaremos **sequelize**.
     - Oculta los detalles de uso de la base de datos.
     - Sólo manejaremos objetos javascript.

Sequelize

- Documentación:
  - `http://www.sequelizejs.com`
- Es un módulo que proporciona ORM (Object-Relational Mapping)
  - Se crea una correspondencia entre los datos y las tablas de la base de datos, y objetos y clases en javascript.
    - Usamos la base de datos a través de estos objetos y clases.
- Características:
  - Soporta varias bases de datos: MySQL, SQLite, PostgreSQL.
  - Definición de esquemas.
  - Sincronización con la base de datos,
  - Validaciones
  - Métodos CRUD
  - Relaciones 1-a-1, 1-a-N, N-a-N.
  - Migraciones
  - . . .



Notas

Usaremos un paquete llamado **sequelize** que se encargará de conectar los objetos manejados por la aplicación con los registros almacenados en la base de datos. Esto es un **ORM**. Significa mapeo entre objetos en la aplicación y registros en la base de datos relacional.

Para que la aplicación funcione es necesario que la base de datos y sus tablas estén ya creadas. Hay varias formas de preparar la base de datos.

1. Podemos conectarnos con un cliente al servidor de base de datos, y crear las tablas, índices, o lo que necesitemos invocando sentencias SQL.

2. También podemos dejar que la aplicación cree automáticamente las tablas que necesite cuando definamos los modelos de objetos. Por ejemplo, invocando a **sequelize.sync()**.
   - En los primeros temas de este proyecto seguiremos esta opción.

3. También podemos aplicar **migraciones**. Son programas creados especialmente para hacer evolucionar la base de datos, creando nuevas tablas, índices, campos, etc. Adicionalmente, las migraciones permiten dar marcha atrás en las evoluciones realizadas.
   - Esta opción es la mejor. Se explicará en temas posteriores.

Notas

Sequelize lo usamos para dos cosas:

1. Para usarlo en la aplicación para manejar la base de datos (ORM).
2. Usarlo como una aplicación independiente para gestionar las tablas de la base de datos (migraciones).

Uso 1º- Es un paquete que nos proporciona el ORM a la aplicación. En la aplicación usaremos clases y objetos javascript, y sequelize se encargará del acceso a la base de datos.

Para usar Sequelize hay que crear un objeto Sequelize describiendo como se accede a la base de datos.

Una vez creado el objeto Sequelize, definiremos los modelos de datos, es decir, como son los objetos javascripts que tienen su almacenamiento en la base de datos. Crearemos así el modelo Post, y en temas posteriores los modelos User, Comment, Attachment, etc. La definición de cada modelo de datos la haremos en un fichero independiente que cargaremos con la función sequelize.import.

Una vez definido un modelo, ya podemos crear objetos nuevos (create, build), buscar objetos en la base de datos (find, findAll), modificar los objetos, salvarlos (save) en la base de datos, borrar (destroy) los objetos, etc.

En el fichero models/models.js es donde crearé el objeto Sequelize, definiré todos los modelos usados en la aplicación, y en un futuro, declararé las asociaciones entre los modelos.

Notas

Uso 2º- El paquete Sequelize proporciona un comando, **sequelize**, con el que se pueden gestionar migraciones.

Cada migración es un fichero javascript donde se programan dos cosas:

- Que cambios hay que hacer en la base de datos para que evolucione y soporte una versión nueva de la aplicación. Normalmente, las evoluciones consisten en crear nuevas tablas, o campos en las tablas ya existentes.
- Que cambios hay que hacer para deshacer los cambios realizados en el punto anterior, y volver así al estado anterior.

Estas tareas son las que realizan las funciones **up** y **down** de los ficheros de migración.

Para poder usar este programa, es necesario crear un directorio para guardar los ficheros de migración, y crear un fichero de configuración con los parámetros de acceso a la base de datos.

Después de realizar esta inicialización, ya podemos crear nuevas migraciones, aplicarlas y deshacerlas.

El uso de migraciones se explicará en el Tema 99.

Instalar Sequelize y SQLite

- Instalar el módulo **Sequelize**:

```
$ npm install --save sequelize@1.7.x
```

Versión estable

- Para usar **Sequelize** con **SQLite,** hay que instalar el módulo **sqlite3:**

```
$ npm install --save sqlite3
```

  - La opción **--save** actualiza automáticamente **package.json** con las dependencias de **sequelize** y **sqlite3**.
    - Añade a la sección de dependencias las líneas:

```
"sequelize": "~1.7.3"
"sqlite3": "~2.2.3"
```

Esquema de la Tabla Posts

- La tabla **Posts** de la base de datos almacenará los posts publicados por los usuarios:
- Tendrá las siguientes columnas:
  - **`id`**
    - Clave primaria.
    - En un entero que se autoincrementa automáticamente.
  - **`AuthorId`**
    - id del autor del post.
    - Clave externa a la futura tabla de usuarios.
  - **`title`**
    - String con el título del post.
  - **`body`**
    - Text con el texto del post.
  - **`createdAt`**
    - Fecha de creación del post.
  - **`updatedAt`**
    - Fecha de actualización del post.

Definir el Modelo **Post**

- Editaremos el fichero **models/index.js** y definiremos como es el modelo (o la clase) **Post** respaldado por la tabla **Posts** la base de datos.
  - Especificaremos el nombre de la tabla, los campos o atributos, sus tipos, las validaciones, etc.
- El fichero **models/index.js** es un módulo de nodejs que exporta la clase **Post** creada.
- La definición de la clase **Post** la haremos en un fichero independiente, **models/post.js**, que se importará desde **models/index.js**.
  - Al definir la clase **Post** no hay que declarar los atributos **id**, **createdAt** y **updatedAt**. Se crean siempre automáticamente.
  - Tampoco hay que crear el atributo **UserId**. Se añadirá automáticamente cuando creemos el recurso **User** y definamos la asociación **1aN** con **Post**.

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

var Sequelize = require('sequelize');

// Configurar Sequelize para usar SQLite.
// No hay Bases de datos, ni usuarios, ni passwords.
// El fichero con la BBDD es blog.sqlite.

var sequelize = new Sequelize(null,null,null,
                              {dialect:"sqlite",
                               storage:"blog.sqlite",
                              });


// Importar la definicion de las clases.
// La clase Post se importa desde el fichero post.js.
var Post = sequelize.import(path.join(__dirname,'post'));

// Exportar los modelos:
exports.Post = Post;


// Crear las tablas en la base de datos que no se hayan creado aun.
// En un futuro lo haremos con migraciones.
sequelize.sync();
```

Añadir **blog.sqlite** a **.gitignore**

models/index.js

```
// Definicion del modelo Post:

module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Post',
      { title: {
            type: DataTypes.STRING,
            validate: {
               notEmpty: {msg: "El campo del título no puede estar vacío"}
            }
        },
        body: {
            type: DataTypes.TEXT,
            validate: {
               notEmpty: {msg: "El cuerpo del post no puede estar vacío"}
            }
        }
      });
}
```

models/post.js

El Controlador

Notas

Sobre como se usa auto-load:

En las rutas de **show**, **edit**, **update** y **destroy** se usa el parámetro **:postid** para indicar a que post nos referimos. Estos cuatro métodos deben acceder a la base de datos para buscar el post con el id especificado en la URL. Hay que repetir el mismo código cuatro veces.

Para no repetir el mismo código cuatro veces se usa **app.param**.

app.param precarga objetos antes de que se ejecuten los middlewares de las rutas.

app.param toma como argumentos el nombre de una variable usada en la definición de rutas, y un middleware para cargar el objeto relacionado con el parámetro.

En nuestro caso:

- la variable es **:postid**,
- el middleware busca en la base de datos el post cuyo id es el valor de **:postid** en la ruta. El objeto **Post** encontrado lo añade al objeto **Request** como un atributo más: **req**.**post**. Si no se encuentra el post, invoca **next** con un error explicativo.
  - Los middlewares de autoload tienen un cuarto parámetro donde reciben el **id** del objeto a cargar: `function(req, res, next, id)`

De esta forma, cuando se ejecuten estas acciones **show**, **edit**, **update** o **destroy**, en **req**.**post** estará ya disponible el objeto Post con el que tienen que trabajar.

Tareas el Controlador Post

- En general, dependiendo de la petición HTTP recibida, el controlador debe:
  - Crear, Leer, Modificar o Borrar en la tabla Posts de la base de datos lo que sea necesario.
    - Estas operaciones las realiza usando el modelo Post.
  - Y generar (renderizar) la vista adecuada para mostrar en el navegador del usuario.
    - Las vistas se generan a partir de ficheros que toman como parámetro (en variables locales) datos calculados por el controlador (sacados de la BBDD, de la sesión ,…).

post_controller.js

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

exports.load = function(req, res, next, id) {

  models.Post
        .find({where: {id: Number(id)}})
        .success(function(post) {
           if (post) {
             req.post = post;
             next();
           } else {
             next(new Error('No existe el post con id='+id+'.'));
           }
        })
        .error(function(error) {
           next(error);
        });
};
```

```
// GET /posts
exports.index = function(req, res, next) {

    models.Post
        .findAll({order: [['updatedAt','DESC']]})
        .success(function(posts) {
            res.render('posts/index', {
                posts: posts
            });

        })
        .error(function(error) {
            next(error);
        });
};
```

```
// GET /posts/33
exports.show = function(req, res, next) {

   res.render('posts/show', {post: req.post});
};
```

```
// GET /posts/new
exports.new = function(req, res, next) {

    var post = models.Post.build(
        { title: 'Introduzca el título',
          body:  'Introduzca el texto del artículo'
        });

    res.render('posts/new', {post: post});

};
```

```
// POST /posts
exports.create = function(req, res, next) {

    var post = models.Post.build(
        { title: req.body.post.title,
          body:  req.body.post.body
        });

    var validate_errors = post.validate();
    if (validate_errors) {
        console.log("Errores de validación:", validate_errors);
        res.render('posts/new', {post: post});
        return;
    }

    post.save()
        .success(function() {
            res.redirect('/posts');
        })
        .error(function(error) {
            next(error);
        });
};
```

```
// GET /posts/33/edit
exports.edit = function(req, res, next) {

    res.render('posts/edit', {post: req.post});
};
```

```
// PUT /posts/33
exports.update = function(req, res, next) {

    req.post.title = req.body.post.title;
    req.post.body = req.body.post.body;

    var validate_errors = req.post.validate();
    if (validate_errors) {
        console.log("Errores de validación:", validate_errors);
        res.render('posts/edit', {post: req.post});
        return;
    }
    req.post.save(['title', 'body'])
        .success(function() {
            res.redirect('/posts');
        })
        .error(function(error) {
            next(error);
        });
};
```

```
// DELETE /posts/33
exports.destroy = function(req, res, next) {


    req.post.destroy()
        .success(function() {
            res.redirect('/posts');
        })
        .error(function(error) {
            next(error);
        });
};
```

Las Vistas

Las vistas

- Los ficheros **EJS** con las vistas los crearé en el directorio **views/posts**.
- Se crearán los siguientes ficheros de vistas:
  - **`index.ejs`**
    - Muestra un resumen de todos los posts que el controlador le pasa en el array **posts**.
  - **`new.ejs`**
    - Muestra un formulario para crear un nuevo post. Se pasa un objeto **post** nuevo.
  - **`edit.ejs`**
    - Muestra un formulario para editar el post que el controlador le pasa en la variable **post**.
  - **`_form.ejs`**
    - Vista parcial usada por **new.ejs** y **edit.ejs** con los campos del formulario. Hereda el objeto **post** que le han pasado como parámetro a **new.ejs** y a **edit.ejs**.
  - **`show.ejs`**
    - Muestra todo el contenido del post que el controlador le pasa en la variable **post**.

Las Vistas de los Posts

posts: Variable donde me pasan los posts a mostrar. Es un array.

```
<header>
  <h2> Posts </h2>
</header>

<% for (var i in posts) { %>
  <article>
     <header>
        <h3> <a href='/posts/<%= posts[i].id %>'><%= posts[i].title %> </a> </h3>
     </header>

     <p> <%= posts[i].body.slice(0,300) %> ... </p>

     <footer>
        <% var formname = 'fpi' + i; %>
        <form method='post' action='/posts/<%= posts[i].id %>' id='<%= formname %>'>
          <input type='hidden' name='_method' value='delete'>
          <a href="/posts/<%= posts[i].id %>/edit"> Editar </a>
          <a href=""
             onclick="confirmarSubmit('¿Seguro que desea borrar el post?',
                                      '<%= formname %>'); return false"> Borrar </a>
        </form>
     </footer>
  </article>
<%  }; %>
<footer>
  <nav> <a href="/posts/new"> Crear nuevo Post </a> </nav>
</footer>
```

Identificador único para cada formulario creado.

views/posts/index.ejs

```
<h2>Nuevo Post</h2>

<form method='post' action='/posts'>

  <% include _form.ejs %>

</form>


<a href="/posts"> Volver  </a>
```

Cuerpo del formulario

views/posts/new.ejs

```
<h2>Editar Post</h2>

<form method='post' action='/posts/<%= post.id %>'>

  <input type='hidden' name='_method' value='put'>

  <% include _form.ejs %>

</form>


<a href="/posts"> Volver </a>
```

Variable donde me pasan el post a editar.

Cambiar el método HTTP

Cuerpo del formulario

views/posts/edit.ejs

```
<div class="field">
  <label for="post_title">Título</label><br />
  <input type="text" id="post_title" name="post[title]"
         size='80' value='<%= post.title%>' />
</div>

<div class="field">
  <label for="post_body">Contenido</label><br />
  <textarea id="post_body" name="post[body]" rows="20"
            cols="80"><%= post.body %></textarea>
</div>

<div class="actions">
  <input name="commit" type="submit" value="Salvar" />
</div>
```

Para construir la query

post: Variable donde me pasan el objeto post a editar o crear.

Para construir la query

views/posts/_form.ejs

```
<h2>Post</h2>

<p>
  <b><%= post.title %></b>
  <br />
  by
  <em>Anónimo</em>
</p>

<p>
  <%= post.updatedAt.toLocaleDateString() %>
</p>

<p><%- escapeText(post.body) %></p>


<a href="/posts/<%= post.id %>/edit"> Edit </a>


<a href="/posts"> Back  </a>
```

post: Variable donde me pasan el objeto post a mostrar.

escapeText es un helper que me he creado en **app.js** para evitar injección de código y sustituir los retornos de carro por etiquetas
.

views/posts/show.ejs

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
   <link rel="stylesheet" type="text/css" href="/stylesheets/style.css">
   <title>CORE</title>
   <script src="/javascripts/misc.js"></script>
</head>

<body>
   <header>
     <h1> COmputación en REd - CORE</h1>
     <nav>
         <a href="/">Home</a>
         <a href="/posts">Posts</a>
         <a href="/creditos.html">Créditos</a>
     </nav>
   </header>

   <section>
     <%- body %>
   </section>

   <footer>
      CORE 2013-2014 - UPM
   </footer>
</body>
</html>
```

ATENCION:
Cargo el fichero Javascript que tiene la funcion confirmarSubmit usada en index.ejs

ATENCION:
He metido un enlace para acceder a los Posts

views/layout.ejs

```
var confirmarSubmit = function(msg,formname) {

    if (confirm(msg)) {
       document.getElementById(formname).submit();
    }
}
```

public/javascripts/misc.js

Helper escapeText

- Queremos que el cuerpo completo de un post (**post.body**) se muestre desde **show.ejs**, cumpliendo estas dos condiciones:
  - Los caracteres problemáticos deben escaparse para evitar inyección de código.
  - Los retornos de carro deben sustituirse por etiquetas **
** para conservar los párrafos introducidos por el usuario.
    - Nota: podríamos usar la etiqueta **<pre>** para conservar los párrafos, pero queda muy feo si las líneas de texto son muy cortas o muy largas. Queda mejor usar **<p>** y retocar el cuerpo del post con la función **escapeText** que me he inventado.
- Ninguna de las directivas **<%- %>** y **<%= %>** nos permiten cumplir las dos condiciones
  - Si se usa **<%- %>** puede haber inyección de código.
  - Si se usa **<%= %>** perdemos los retornos de carro usados por el usuario para crear párrafos.
- Por estas razones he creado el helper **escapeText** que realiza las dos tareas.

Creo una variable global accesible por las vistas

```
app.locals.escapeText =  function(text) {
   return String(text)
          .replace(/&(?!\w+;)/g, '&amp;')
          .replace(/</g, '&lt;')
          .replace(/>/g, '&gt;')
          .replace(/"/g, '&quot;')
          .replace(/\n/g, '<br>');
};
```

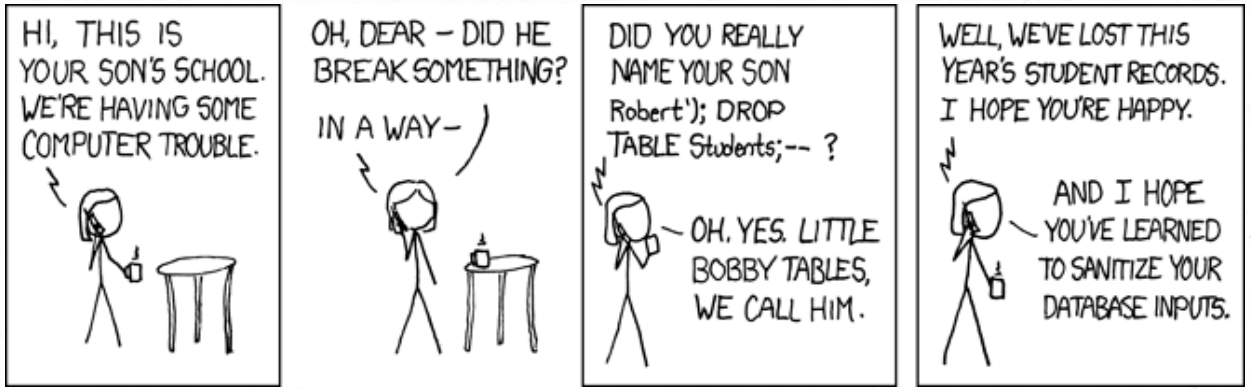
app.js

Otros Cambios Realizados

- Cambiar el estilos de los posts:
  - Para destacar los posts publicados, modificar **public/stylesheets/style.css** añadiendo:

```
article {
  border: 2px solid #ffffff;
  padding: 7px;
  margin: 20px;
  background-color: #e5f5ff;
}
```

Inyección de Código



¿Qué es la Inyección de Código?

- Es un problema de seguridad que aparece cuando generamos sentencias (para ejecutar) usando datos proporcionados por el usuario.
  - Los datos del usuario pueden contener fragmentos de sentencias SQL para borrar una base de datos, strings que al usarse en condiciones booleanas provocan una evaluación siempre verdadera, código javascript para inundar la pantalla de ventanas emergentes, etc.
- Hay que desconfiar siempre de los datos proporcionados por el usuario.
  - Estos datos pueden llegar al programa al rellenar los campos de un formulario, al recibir una petición HTTP, en una cookie, etc.
- Hay que validar siempre los datos recibidos antes de usarlos.

Inyección SQL

- Supongamos que ejecutamos el siguiente código para buscar en la base de datos un usuario con el **login** y el **password** introducidos en un formulario:

```
var login  = req.query.login;
var passwd = req.query.password;
var sql = "SELECT * FROM Users "+
                  "WHERE login='"+login+"' "+
                   " AND password='"+passwd+"';"
execute(sql, una_callback_cualquiera);
```

- Este programa busca el usuario y seguramente llamará a la callback especificada con el registro encontrado, o con un null.

- Si en el formulario se introdujo **pepe** y **1234**, la sentencia SQL ejecutada es:

```
SELECT * FROM Users WHERE login='pepe'
                      AND password='1234';
```

- Si existe el usuario **pepe**, y su password es **1234**, entonces se llama a la callback con el registro encontrado.

- **Ataque 1**: Vamos a introducir en el formulario los siguientes valores:

  login = x

  password = x' OR '1=1

  - En este caso la sentencia SQL ejecutada será:

```
SELECT * FROM Users WHERE login='x'
       AND password='x' OR '1=1';
```

  - Dado que 1=1 es siempre verdadero, entonces siempre se seleccionan todos los usuarios existentes,
    - y se llama a la callback dada.

- **Ataque 2**: ¿Piense qué pasaría si como valor del password se introduce el siguiente valor:

```
x'; DROP TABLE Users; --
```

Inyección Javascript

- Ejecute la aplicación Blog que estamos desarrollando, y cree un post con el siguiente título:

```
El ataque de la mujer de Thor
<script src="http://code.jquery.com/jquery.min.js"></script>
<script>
   $("body").css("background-image",
   "URL(http://img.poprosa.com/original/11445.jpg)").css(
   "background-repeat", "repeat");
</script>
```

  - Visualice en la aplicación (acción show) el post creado.
  - Modifique ahora el fichero **views/posts/show.ejs**, sustituyendo la línea:

```
<%= post.title %>
```

    - por esta otra:

```
<%- post.title %>
```

  - Visualice otra vez el post creado antes.

- **¿Nota alguna diferencia? Explique qué pasa.**

Despliegue en Heroku

No Instalar SQLite en Heroku

- En Heroku no se puede usar SQLite.
  - El sistema de fichero no guarda permanentemente los ficheros que creamos nosotros. Desaparecen al cabo de un rato.
    - **blog.sqlite** se borrará perdiendo los datos
- Para que el paquete **`sqlite`** no se instale en Heroku, editamos **`package.json`** y movemos su dependencia a una sección nueva

```
"devDependencies": {
  "sqlite3": "~2.2.3"
},
```

Dependencias necesarias sólo para desarrollar

- Para que en Heroku no se instalen las dependencias de desarrollo (devDependencies) debemos configurar que estamos en modo producción.
- Crear la variable de entorno **`NODE_ENV=production`**.

```
$ heroku config:set NODE_ENV=production
```

Usar Postgres en Heroku

- Heroku nos permite usar gratuitamente distintas bases de datos.
- Vamos a usar **Heroku Postgres**.
  - Instalar el módulo **pg** y guardar la dependencia en **package.json**

```
$ npm install --save pg
```

  - Instalar el addon de postgres con el plan de desarrollo gratuito:

```
$ heroku addons:add heroku-postgresql:dev
```

    - La documentación de heroku postgres puede verse ejecutando:

```
$ heroku addons:docs heroku-postgresql
```

- Los datos de acceso al servidor postgres se guardan en una variable de entorno:

```
$ heroku config
HEROKU_POSTGRESQL_GOLD_URL:⤤
        postgres://user:passwd@host:port/database
```

- También puede obtenerse esta información desde el sitio heroku postgres

```
https://postgres.heroku.com
```

- Ahora hay que retocar **models/index.js** para usar los valores adecuados de acceso a la base de datos, para los casos de ejecutar en local y en Heroku.

  - **PELIGRO**: Dado que mi fichero **models/index.js** está en github, no debo escribir los valores de acceso en él. Los vería cualquiera y podría atacar mi base de datos.
    - Usaré una variable de entorno para almacenar los datos de acceso a la BBDD, y modificaré **models/index.js** para que lea los valores de esta variable de entorno.
      - Nota: nadie puede ver las variables de entorno que he creado en Heroku.

- En Heroku la variable de entorno con los datos de acceso a la base de datos se llamaba **HEROKU_POSTGRESQL_YELLOW_URL** (o algo parecido).
  - Copiaré el valor de esta variable en la nueva variable llamada **DATABASE_URL**.
    - Esto puede hacerse manualmente o ejecutando el comando:

```
$ heroku pg:promote HEROKU_POSTGRESQL_YELLOW_URL
```

- Para la ejecución en local crearé la misma variable de entorno (**DATABASE_URL**) en un fichero llamado **.env** en la raíz del proyecto, y usaré el comando **foreman** para lanzar el servidor localmente.
  - El comando **foreman** lanza localmente el servidor cargando antes el entorno de **.env**.

- Con estas adaptaciones, la configuración de **Sequelize** en el fichero **models/index.js**, y **.env** queda así:

models/index.js

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

var Sequelize = require('sequelize');

// Configurar Sequelize para usar SQLite. Uso una expresion regular para extraer
// los valores de acceso a la base de datos
var vals = process.env.DATABASE_URL.match(/(.*)\:\/\/(.*?)\:(.*)@(.*)\:(.*)\/(.*)/);

var DATABASE_PROTOCOL = vals[1];
var DATABASE_DIALECT = vals[1];
var DATABASE_USER = vals[2];
var DATABASE_PASSWORD = vals[3];
var DATABASE_HOST = vals[4];
var DATABASE_PORT = vals[5];
var DATABASE_NAME = vals[6];

var sequelize = new Sequelize(DATABASE_NAME, DATABASE_USER, DATABASE_PASSWORD,
            { dialect:  DATABASE_DIALECT,
              protocol: DATABASE_PROTOCOL,
              port:     DATABASE_PORT,
              host:     DATABASE_HOST,
              storage:  process.env.DATABASE_STORAGE, // solo local en .env
              omitNull: true                          // para postgres
            });

// Importar la definicion de las clases.
// La clase Post se importa desde el fichero post.js.
var Post = sequelize.import(path.join(__dirname,'post'));

// Exportar los modelos:
exports.Post = Post;

// Crear las tablas en la base de datos que no se hayan creado aun.
// En un futuro lo haremos con migraciones.
sequelize.sync();
```

.env

```
DATABASE_URL=sqlite://:@:/
DATABASE_STORAGE=blog.sqlite
```

Ejecución Local con **foreman**

- Para arrancar localmente los mismos procesos definidos en el fichero **`Procfile`**, ejecutaremos:

```
$ foreman start
```

  - Útil para probar la aplicación localmente usando los mismos comandos que se ejecutarán al desplegar..

- **`foreman`** usa las variables de entorno definidas en el fichero **`.env`** situado en la raíz del proyecto.

  - Crear un fichero **`.env`** con las mismas variables configuradas con "**`heroku config`**", pero con los valores adecuados para el entorno local de ejecución.

  - Dado que cada desarrollador puede que use una base de datos distinta, no meteré el fichero **`.env`** en **git**. Por tanto hay que añadir **`.env`** al fichero **`.gitignore`**.

Despliegue en Heroku

- Congelar cambios en git.
  - Ejecutar comandos **`git add`**, **`git commit`**, etc.
- Entrar en modo mantenimiento:

```
(local)$ heroku maintenance:on
```

- Actualizar versión en Heroku ejecutando **sólo uno** de estos comandos:

```
(local)$ git push -f heroku tema2:master
(local)$ git push heroku master
```

Copiar en la rama **master** de **Heroku**. El primer comando copia en contenido **local** de la rama **tema2** en la rama **master** de **Heroku**. El segundo comando copia el contenido **local** de la rama **master** en la rama **master** de **Heroku**. La opción **-f** (forzar) puede usarse para forzar la operación en caso de problemas.

- Salir del modo mantenimiento:

```
(local)$ heroku maintenance:off
```

Examen

Preguntas

- Búsquedas:
  - Añadir un botón para buscar en el título y en el cuerpo de los posts un determinado texto.