Proyecto de la asignatura CORE

Desarrollo de un Blog

Tema 6: Relación entre Usuarios y Posts.

CORE 2013-2014

ver: 2014-04-23

Índice

- Definir la relación 1-a-N entre User y Post.
- El usuario logeado es el autor de los nuevos posts.
- Mostrar quién es el autor de los posts.
- Sólo el autor puede editar o borrar sus posts.
- Los datos de un usuario sólo los puede editar el propio usuario.
- Desplegar en Heroku.
- El ejemplo está disponible en la rama **tema6**.

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

Relación entre Post y User

Notas

Entre los usuarios y los post hay una relación 1 a N. Un post pertenece a un único usuario (su autor), y un usuario puede tener varios posts (todos los que haya escrito).

Estas relaciones se reflejan en las tablas de la base de datos usando claves externas. Estas claves referencian a filas o registros de otras tablas.

Sequelize permite especificar las relaciones entre los modelos usando los métodos **belongsTo**, **hasOne** y **hasMany**. Al especificar estar relaciones se crean una serie de métodos que permiten añadir, borrar o consultar los datos de las clases conectadas (asociadas) con estas relaciones (lo veremos con más detalle en las siguientes transparencias).

Una característica muy cómoda soportada por Sequelize es la carga ansiosa de asociaciones. Consiste en recuperar datos de varias tablas con una única llamada a **find** o **findAll**. Se hace usando la opción **include**. Con esta opción se indica que se carguen también los objetos de las tablas relacionadas. Esta opción provoca que se realice una petición SQL de tipo JOIN.

Discusión: ¿Qué opción es mejor?

- Que el auto-load realice siempre llamadas a find con la opción include, aunque el valor precargado no se use luego en algunas vistas.
- No hacer llamadas a find usando include, y realizar varias llamadas a find.
  - ➡ Factores a tener en cuenta:
    - escribir menos código
    - eficiencia del código
    - minimizar accesos al proveedor de BBDD para disminuir costes.

Indicar Relación en el Modelo

- Entre usuarios y post hay una relación 1 a N.
  - un post es de un sólo usuario/autor.
  - un usuario puede haber publicado varios posts.
- Sequelize permite reflejar estas relaciones.
- Añadimos a **`models/models.js`** estas líneas:

```
User.hasMany(Post, {foreignKey: 'AuthorId'});
Post.belongsTo(User, {as: 'Author',
                      foreignKey: 'AuthorId'});
```

  - El módelo en el que se usa la función **belongsTo** es el modelo que posee la clave externa apuntando al modelo relacionado.
    - Es decir, la tabla **Posts** tiene la columna **AuthorId**.

Notas

Opciones usadas al especificar la asociación entre User y Post:

- **Sin opciones**:

```
User.hasMany(Post);
Post.belongsTo(User);
```

  - Con estas declaraciones, el campo de la tabla **Post** usado como clave externa que apunta a la tabla **User** debe llamarse **UserId**.
  - En este caso, al definir el modelo **Post** (con el método **sequelize.define**) no es necesario escribir explícitamente el campo **UserId**. (Nota: la **U** y la **I** son mayúsculas)
  - El campo usado como clave externa podría llamarse **user_id** si se hubiera usado la opción **underscored** al crear el objeto **sequelize**.
  - El valor del primer parámetro se usa para crear los nombres de los métodos de acceso de la asociación:
    - **addPost**, **removePost**, **hasPost** y **hasPosts**
    - **getUser** y **setUser**
  - Para cambiar el nombre de la asociación y de los métodos de acceso creados, se usa la opción **as**.

- **Opción foreignKey**:

```
User.hasMany(Post);
Post.belongsTo(User, {foreignKey: 'AuthorId'});
```

  - Si el campo de la tabla **Post** usado como clave externa que apunta a la tabla **User** no se llama **UserId**, entonces debe indicarse su nombre con la opción **foreignKey**.

- **Opción as**:

```
User.hasMany(Post);
Post.belongsTo(User, {as: 'Author', foreignKey: 'AuthorId'});
```

  - Con la opción **as** el nombre de la asociación es **Author**, y los métodos de acceso creados son **getAuthor**, **setAuthor**.

- Tenemos que añadir una columna **AuthorId** a la tabla **Posts**.
  - Recordad que la tabla **Posts** se creó en el tema 2 usando la sentencia **sequelize.sync()** y en ese momento no se creó esa columna.
  - Algunas opciones para añadir la columna **AuthorId**:
    1. Borrar el fichero de la base de datos **blog.sqlite**.
       - La próxima vez que ejecutemos el servidor, **sequelize.sync()** creará otra vez el fichero de la base de datos **blog.sqlite**, y una tabla **Posts** con la columna **AuthorId**.
       - Perderemos todos los datos.
    2. Añadir la sentencia `Post.sync({force: true});`
       - Destruye la tabla **Posts** y vuelve a crearla.
         - Una vez creada la nueva tabla, borrar esta sentencia.
       - Perderemos todos los datos de la tabla **Posts**.
    3. Usar una **migración** que cree esa columna.
       - Con esta opción no se pierden datos.
       - Las migraciones las estudiaremos más adelante.

models.js

```
. . .

var Post = sequelize.import(path.join(__dirname,'post'));
var User = sequelize.import(path.join(__dirname,'user'));

// La llamada User.hasMany(Post);
//  - crea un atributo llamado UserId en el modelo de Post
//  - y en el prototipo de User se crean los metodos getPosts, setPosts,
//    addPost, removePost, hasPost y hasPosts.
//
// Como el atributo del modelo Post que apunta a User se llama AuthorId
// en vez de UserId, he añadido la opcion foreignKey.
User.hasMany(Post, {foreignKey: 'AuthorId'});

// La llamada Post.belongsTo(User);
//  - crea en el modelo de Post un atributo llamado UserId,
//  - y en el prototipo de Post se crean los metodos getUser y setUser.
//
// Como el atributo del modelo Post que apunta a User se llama AuthorId
// en vez de UserId, he añadido la opcion foreignKey.
//
// Con el uso de la opcion "as" la relacion se llama Author, y los metodos
// de acceso creados son setAuthor y getAuthor.
Post.belongsTo(User, {as: 'Author', foreignKey: 'AuthorId'});

exports.Post = Post;
exports.User = User;

sequelize.sync();
```

Los métodos generados son Asíncronos

- Hay que recordar que node es asíncrono.
- Y los métodos de acceso que se generan al definir las relaciones también son asíncronos.
  - Es decir, **getPosts**, **addPost**, **removePost**, **hasPost**, **setAuthor**, etc. son asíncronos.
  - Hay que esperar a que se genere un evento **success** o **error**.
    - Al manejador de **success** se le pasa el callback que queremos ejecutar cuando se tengan los resultados pedidos.
      - El callback toma como argumento los objetos pasados en los métodos, o los resultados obtenidos.
    - Si se dispara el evento **error**, se ejecuta la callback que hayamos especificado pasándola como argumento el error producido.
  - Ejemplos:

```
user.addPost(post).success(function(post) {   });
user.getPost({where: ???}).success(function(post) {   });
```

Usuario Logeado = Autor

Usuario Logeado = Autor

- Queremos que el autor de un post nuevo sea el autor logeado.
  - Sólo es necesario cambiar en el controlador **post_controller.js** el método **create** para asignar el valor **req.session.user.id** al campo **AuthorId** del post creado.

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

    var post = model.Post.build(
        { title: req.body.post.title,
          body: req.body.post.body,
          AuthorId: req.session.user.id
        });

    . . .
```

Mostrar Autores de los Posts

Mostrar Autores de los Posts

- Quiero mostrar junto a cada post, el nombre del autor.
  - En **`views/posts/index.ejs`**, la vista que muestra el índice de posts añadiré el nombre del autor.
  - En **`views/posts/show.ejs`**, la vista que muestra un post añadiré el nombre del autor.
- Cambios a realizar:
  - En los métodos **index** y **show** de **post_controller.js** hay que añadir las sentencias necesarias para cargar también los objetos **User** de los autores de los posts.
  - En las vistas **index.ejs** y **show.ejs** de los posts se presentará el atributo **name** de los objetos **User** cargados.

post_controller.js - **index**

```
exports.index = function(req, res, next) {
    models.Post
        .findAll({order: [['updatedAt','DESC']],
                  include: [ { model: models.User,
                               as: 'Author' }
                             ]
                 })
        .success(function(posts) {
            res.render('posts/index', {
                posts: posts
            });

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

Para cada objeto **Post** se hace una precarga (carga ansiosa) de su objeto **User** asociado, indicando que en **models.js** lo hemos renombrado como **Author**.

post_controller.js - **show**

```
exports.show = function(req, res, next) {

    // Buscar el autor
    models.User
        .find({where: {id: req.post.AuthorId}})
        .success(function(user) {

            // Si encuentro al autor lo añado como el atributo author,
            // si no lo encuentro añado {}.
            req.post.author = user || {};

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

};
```

NO uso carga ansiosa aquí. Busco el autor con **find** y lo guardo en **req.post.author**, igual que lo hace la carga ansiosa.

Antiguo contenido de show.

Mostrar Autores en index.ejs

```
. . .

<% for (var i in posts) { %>
  <article>
     <header>
        <h3>
          <a href='/posts/<%= posts[i].id %>'> <%= posts[i].title %> </a>
        </h3>
        <em> <%= posts[i].author && posts[i].author.name || "Sin autor" %> </em>

     </header>

. . .
```

views/posts/index.ejs

Mostrar Autor en show.ejs

```
. . .
<p>
  <b><%= post.title %></b>
  <br />
  by
  <em><%= post.author && post.author.name || 'Sin autor' %></em>
</p>
. . .
```

views/posts/show.ejs

Sólo el Autor puede Editar o Borrar sus Posts

Sólo el Autor puede Editar o Borrar sus Posts

- Queremos que sólo el autor de un post pueda borrarlo o editar su contenido.
- Hay que hacer dos cosas.
  - Mostrar los enlaces de edición y borrado de cada post sólo a sus autores.
  - Evitar que usando directamente peticiones HTTP se pueda editar o borrar un post.
    - Por ejemplo, usando el comando **curl**.
    - Crearemos un middleware para proteger este tipo de acceso

Eliminar enlaces para no autores

- Los enlaces para editar o borrar un post se presentan en las vistas **index.ejs** y **show.ejs**.
- Añadiremos unas sentencias condicionales (**if**) en estos ficheros para que estos enlaces sólo estén disponibles para el autor del post.
- Por ejemplo, para que el enlace **edit** sólo le aparezca al autor del post se añade:

```
<% if (session.user && session.user.id == posts[i].AuthorId) {%>
   <a href="/posts/<%= posts[i].id %>/edit"> Edit </a>
<% } %>
```

```
<h2>Post</h2>

<p>
  <b><%= post.title %></b>
  <br />
  by
  <em><%= post.author && post.author.name || 'Sin autor' %></em>
</p>

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

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

<% if (session.user && session.user.id == post.AuthorId) {%>

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

<% } %>

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

La condición del if sólo la cumple el autor cuando está logeado.

views/posts/show.ejs

La condición del if sólo la cumple el autor cuando está logeado.

```
. . .

<footer>

<% if (session.user && session.user.id == posts[i].AuthorId) {%>

   <% 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>

. . .
```

views/posts/index.ejs

Middleware: Comprobar User logeado es el Autor

- A la aplicación web puede accederse directamente atacando al API REST.
  - desde una aplicación, usando el comando curl, etc.
- Hay que proteger las peticiones de borrado y edición de post para usuarios no logeados.
- Crearemos un middleware que compruebe si el autor es el usuario logeado.
  - Añadiremos este middleware a las rutas que queramos proteger.

Crear Middleware

- Creamos un middleware para comprobar que el usuario logeado es el autor.
- Lo añadimos al controlador de post (**post_controller.js**):

```
/*
* Comprueba que el usuario logeado es el author.
*/
exports.loggedUserIsAuthor = function(req, res, next) {

   if (req.session.user && req.session.user.id == req.post.AuthorId) {
      next();
   } else {
      console.log('Oper Prohibida: usuario logeado no es el autor.');
      res.send(403);
   }
};
```

Proteger Rutas con Middleware

```
router.get('/posts', postController.index);

router.get('/posts/new', sessionController.loginRequired,
                         postController.new);

router.get('/posts/:postid([0-9]+)', postController.show);

router.post('/posts', sessionController.loginRequired,
                      postController.create);

router.get('/posts/:postid([0-9]+)/edit', sessionController.loginRequired,
                                          postController.loggedUserIsAuthor,
                                          postController.edit);

router.put('/posts/:postid([0-9]+)', sessionController.loginRequired,
                                     postController.loggedUserIsAuthor,
                                     postController.update);

router.delete('/posts/:postid([0-9]+)', sessionController.loginRequired,
                                        postController.loggedUserIsAuthor,
                                        postController.destroy);
```

Sólo el propio usuario puede editarse

Sólo el propio usuario puede editarse

- Modificar la aplicación para que:
  - La edición de los datos de un usuario sólo lo pueda hacer el mismo usuario.
  - Nadie pueda borrar los usuarios.
- Hay que hacer estos cambios:
  - Quitar todos los enlaces de borrado de usuarios.
  - Los enlaces de edición de un usuario sólo se mostrarán si el usuario es el usuario logeado.
  - Crear un middleware que proteja las rutas de edición de usuarios.
  - Borrar las rutas de borrado de usuarios.

Enlaces de Edición y Borrado

- Los enlaces de edición y borrado de usuarios sólo están en las vistas **index.ejs** y **show.ejs** de los usuarios.
- Modificar estas vistas para:
  - Añadir un condicional en los enlaces de edición que compruebe que el usuario a editar es el usuario logeado.

```
<% if (session.user && session.user.id == users[i].id) {%>
   <a href="/users/<%= users[i].id %>/edit"> Editar </a>
<% } %>
```

    - Nota: recordad que **session** se pasa a las vistas usando un helper dinámico.
  - Eliminar los enlaces de borrado de usuarios.

Modificar Rutas

- Primer cambio:
  - Proteger las siguientes rutas:
    - ruta para obtener el formulario de edición de usuario.
    - ruta de actualización de los datos de un usuario.
  - Crearemos un middleware que compruebe si el autor al que afecta estas rutas es el propio usuario logeado.

- Segundo cambio:
  - Eliminar la ruta de borrado de usuarios.
    - En un futuro podríamos recuperar esta ruta para que esté disponible para un usuario administrador, o ...

Añadir a user_controller.js

```
/*
* Comprueba que el usuario logeado es el usuario al
* que se refiere esta ruta.
*/
exports.loggedUserIsUser = function(req, res, next) {

   if (req.session.user && req.session.user.id == req.user.id) {
      next();
   } else {
      console.log('Ruta prohibida: no soy el usuario logeado.');
      res.send(403);
   }
};
```

Proteger Rutas

```
router.get('/users', userController.index);

router.get('/users/new', userController.new);

router.get('/users/:userid([0-9]+)', userController.show);

router.post('/users', userController.create);

router.get('/users/:userid([0-9]+)/edit', sessionController.loginRequired,
                                          userController.loggedUserIsUser,
                                          userController.edit);

router.put('/users/:userid([0-9]+)', sessionController.loginRequired,
                                     userController.loggedUserIsUser,
                                     userController.update);

// router.delete('/users/:userid([0-9]+)', sessionController.loginRequired,
//                                         userController.destroy);
```

routes/index.js

Despliegue en Heroku

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 tema6:master
(local)$ git push heroku master
```

Copiar en la rama **master** de **Heroku**. El primer comando copia en contenido **local** de la rama **tema6** 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

- Añadir soporte para favoritos.
  - Un usuario puede indicar cuales son sus posts favoritos.
  - Y poder marcar alguno de ellos como muy importantes.