Proyecto de la asignatura CORE

Desarrollo de un Blog

Tema 8: Adjuntar Imágenes.

CORE 2013-2014

ver: 2014-05-11

Índice

- ¿Qué es una imágen adjunta?
- ¿Cómo se sube un fichero?
- Usar Cloudinary para gestionar imágenes.
  - Crearse una cuenta en Cloudinary
  - ¿Como se sube, borra o se accede a un fichero?
- Definir el modelo de adjuntos y su relación con los posts (1-a-N).
- Definir las rutas (API REST) en routes/index.js.
- Crear el controlador controllers/attachment_controller.js.
- Crear las vistas views/attachment/*.ejs.
- Retocar el show de los posts para mostrar y gestionar los adjuntos de cada post.
- Desplegar en Heroku.
- Documentación y tutoriales de Cloudinary

  `http://cloudinary.com`

  `https://devcenter.heroku.com/articles/cloudinary`
- El ejemplo está disponible en la rama tema8.

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

¿Qué es una Imágen Adjunta?

- Queremos adjuntar imágenes (gif, jpeg, png, …) a los post publicados.
- Relación 1-a-N entre posts y adjuntos:
  - A cada post se le pueden adjuntar varias imágenes,
  - pero cada imagen solo es adjunta de un post.
- La creación de imágenes adjuntas y comentarios es muy parecida:
  - Repetiremos prácticamente el mismo desarrollo realizado para soportar comentarios.
- ¿Qué cambia?:
  - El campo de texto con el contenido del comentario se sustituye por un campo para la URL del fichero con la imagen a adjuntar.
    - También crearemos campos para el nombre del fichero, el mime type, etc.
  - El texto del comentario puede editarse, pero con los adjuntos no crearemos las rutas para editar el URL.
  - Los ficheros con las imágenes a adjuntar se subirán usando un formulario, se almacenarán en un servidor en la nube, y el URL donde se guarda cada fichero en la nube será el valor guardado en la base de datos.
  - Sólo puede adjuntar imágenes el autor del post.

Subir ficheros

Navegador: Formulario de Subida

- Para subir un fichero usando el navegador hay que usar un formulario.

```
<form method="post"
      action="/subir"
      enctype="multipart/form-data">

      Image::
      <input type="file" name="adjunto" accept="image/*"/>
      <input type="submit" value="Subir"/>
</form>
```

- Detalles:
  - El **método** HTTP es **POST**.
  - El valor del atributo **action** es la URL que maneja la subida de los ficheros.
  - El valor de **enctype** debe ser "**multipart/form-data**".
  - Para cada fichero a subir usar un campo input de tipo "**file**".
  - El atributo **name** identifica al fichero a subir.
  - Uso **accept** para restringir los tipos de ficheros que se pueden subir.

Servidor: Atender la Petición

- El servidor debe atender la petición HTTP para obtener el contenido del fichero que se está subiendo.
- El contenido es multipart/form-data:
  - Llegarán varias partes con los campos y fichero que se suben desde el formulario
  - Se usan lineas para separar cada parte.
  - Cada parte tiene unas cabeceras que identifican su contenido.
- Recomendable usar algún módulo ya existente para procesar las peticiones con un body multiparte.
  - **formidable**, **connect-multiparty**, **multiparty**, **form-data**, **connect-busboy**, **busboy** ,…
    - *NOTA: **connect-multipart** ha sido deprecado.*

Notas

multiparty

Módulo que se encarga de manejar los cuerpos multiparte de las peticiones HTTP.

Puede usarse de varias maneras:

- Manera 1: Esperar a que termine la subida del fichero al servidor, el cual se guarda en algun directorio temporal del disco. Terminada la subida hago algo con el fichero creado.
  - Ver transparencia: Servidor: Ejemplo 1.
  - La subida de un fichero al servidor puede tardar bastante tiempo si el fichero es grande. Hay que esperar a que el fichero haya subido completamente antes de manejarlo.
- Manera 2: No esperar a que suba todo el fichero al servidor, sino capturar los eventos para obtener inmediatamente los datos que se están subiendo, y hacer algo con ellos. Así no se guarda el fichero en disco (*a menos que lo hagamos nosotros explícitamente*).
  - Cada parte subida es un readStream del que podemos leer los datos del fichero según van llegando.
    - Los datos que llegan pueden enviarse al destino final.
      - Por ejemplo, creando un writeStream conectado con el readStream de subida usando un pipe.
  - Ver transparencia: Servidor: Ejemplo 2.
  - Usaremos está opción con Cloudinary para que los datos que llegan al servidor se reenvíen directamente a Cloudinary.

Servidor: Ejemplo 1

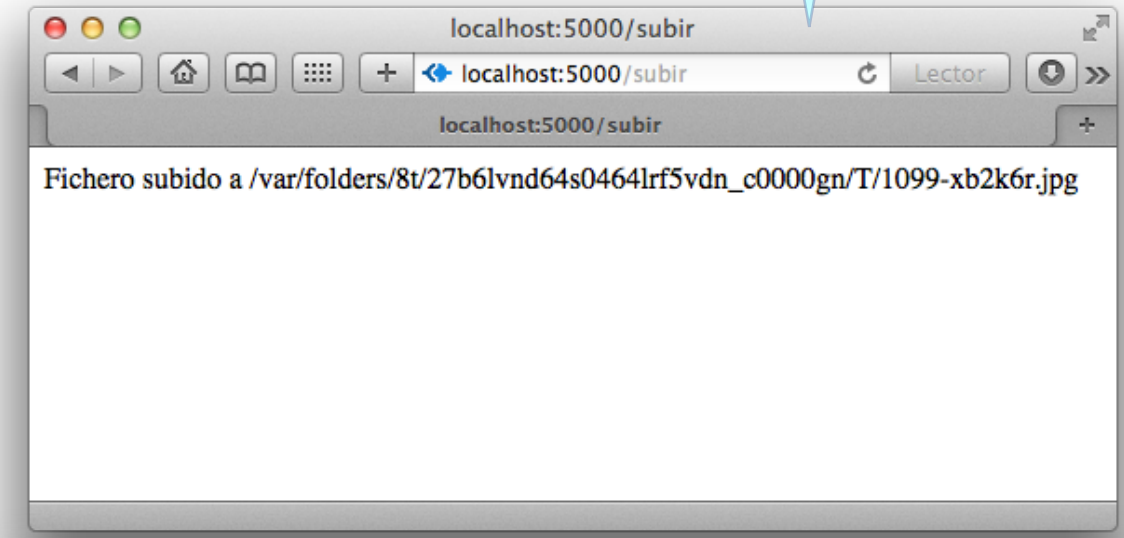
```
var multiparty = require('multiparty');

function subir(req, res, next) {

    var form = new multiparty.Form();

    form.parse(req, function(err, fields, files) {
      console.log(files);
      res.send('Fichero subido a ' + files.adjunto[0].path);
    });
};
```

Campos normales del formulario

Ficheros subidos por el formulario

Se sube el fichero a un directorio temporal.
Luego lo usaremos para hacer algo con él.

```
res.send('Fichero subido a ' + files.adjunto[0].path);
```




```
console.log(files);
```

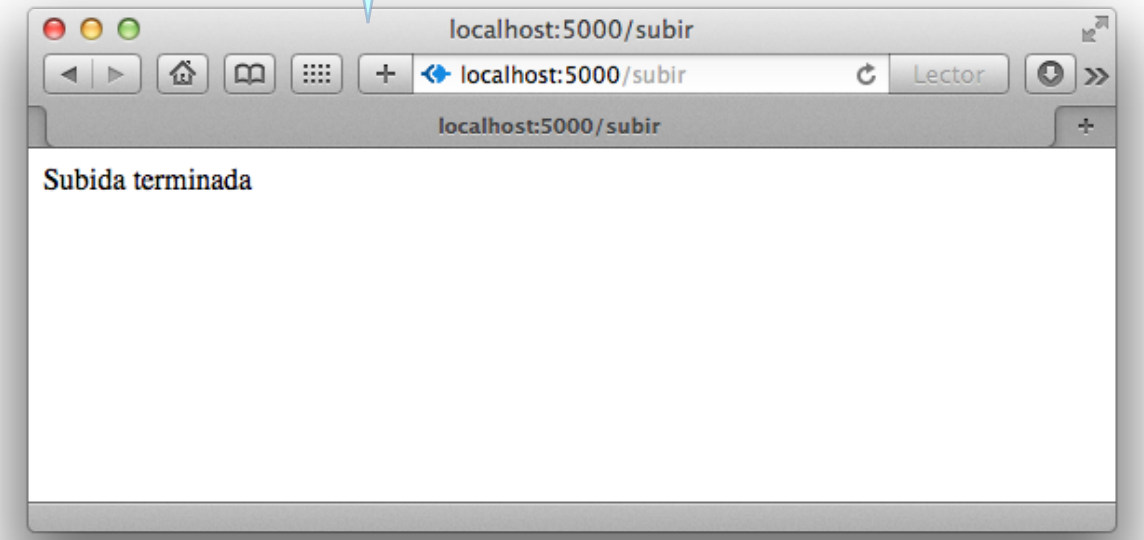
```
blog — ⌘1
santiago@alipori:~/Documents/teleco/docencia/core/2013-2014/Proyecto/blog_2014/v3/blog$ foreman start
16:09:28 web.1  | started with pid 1099
16:09:35 web.1  | POST /subirnm 404 21ms - 10.06kb
16:09:35 web.1  | GET /stylesheets/style.css 304 4ms
16:09:35 web.1  | GET /javascripts/misc.js 304 2ms
16:09:49 web.1  | GET /prueba.html 200 6ms - 788b
16:09:49 web.1  | GET /stylesheets/style.css 304 2ms
16:09:49 web.1  | GET /javascripts/misc.js 304 1ms
16:09:56 web.1  | { adjunto:
16:09:56 web.1  |    [ { fieldName: 'adjunto',
16:09:56 web.1  |        originalFilename: 'Flower.jpg',
16:09:56 web.1  |        path: '/var/folders/8t/27b6lvnd64s0464lrf5vdn_c0000gn/T/1099-xb2k6r.jpg',
16:09:56 web.1  |        headers: [Object],
16:09:56 web.1  |        ws: [Object],
16:09:56 web.1  |        size: 26364 } ] }
16:09:56 web.1  | POST /subir 200 11ms - 81b
```

Servidor: Ejemplo 2

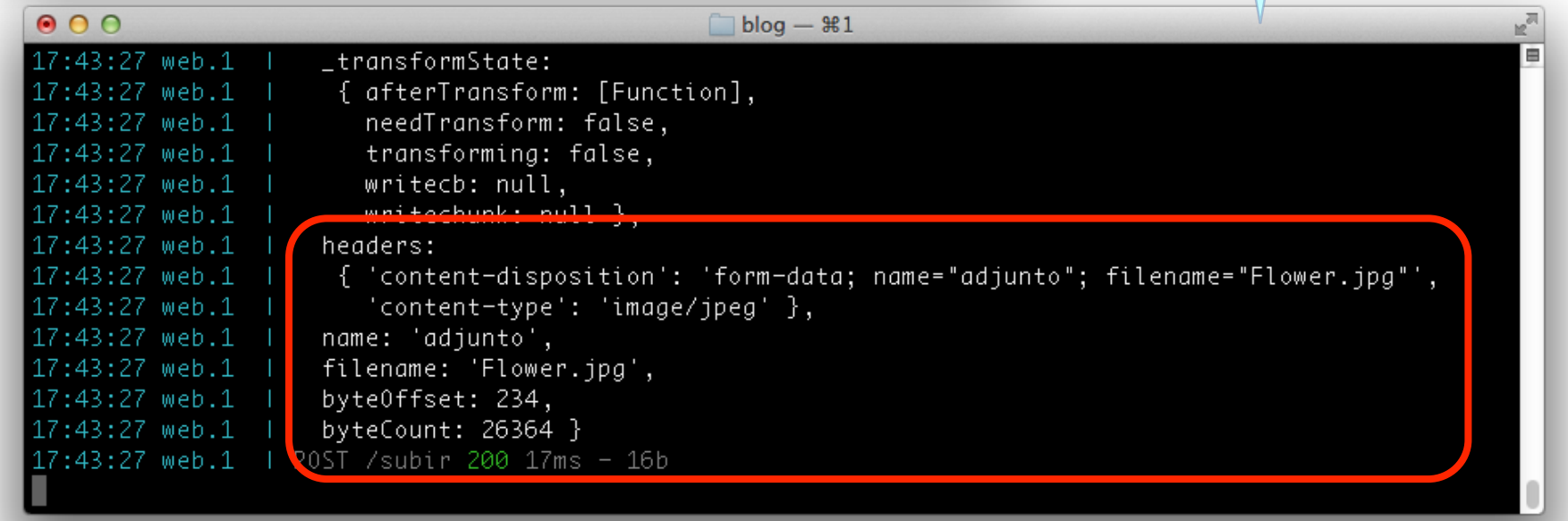
```
var multiparty = require('multiparty');
var fs = require('fs');

function subir(req, res, next) {

    var form = new multiparty.Form();

    form.on('error', function(error) {
      next(error);
    });

    form.on('part', function(part) {

        if (part.filename) {
            console.log(part);
            var out_stream = fs.createWriteStream('uploads/'+part.filename,
                                    {flags: 'w', encoding: 'binary', mode: 0644});
            part.pipe(out_stream);
        }
    });

    form.on('close', function() {
        res.send('Subida terminada');
    });

    form.parse(req);
};
```

No se guarda el fichero subido en el disco. Atiendo los eventos para recuperar los datos de cada parte. **part** es un Stream Reader que conecto con un Stream Writer que guarda los datos (fichero) en el directorio **uploads**.

Evento close -> res.send('Subida terminada');



Final de -> console.log(part);



Cloudinary

Cloudinary

http://cloudinary.com

Plan Gratuito

- Para usar Cloudinary de forma gratuita, hay que registrarse en su web.
  - El plan gratuito incluye **500MB** de almacenamiento, **50.000** ficheros y un ancho de banda **1GB/mes**.
  - **Heroku** dispone de un **addon** de Cloudinary, pero su uso requiere tener una cuenta de pago (no usaremos el addon) .

- Usar el siguiente enlace para registrarse: 

https://cloudinary.com/users/register/free

Instalación y Configuración

- Tras registrarnos en Cloudinary, nos asignan las claves para usar el servicio.
  - Lo más importante es que nos dan el valor que tenemos que asignar a la variable de entorno **CLOUDINARY_URL** para usar el servicio.
- Crear la variable de entorno **CLOUDINARY_URL**
  - En nuestra máquina de Heroku, ejecutar:

    ```
    $ heroku config:add CLOUDINARY_URL=???
    ```
  - En la máquina local de desarrollo, añadir al fichero **.env**:

    ```
    CLOUDINARY_URL=???
    ```
- Instalar el paquete Cloudinary:

  ```
  $ npm install --save cloudinary
  ```

Ej: Guardar Fichero Subido en Cloudinary

```
var cloudinary = require('cloudinary');
var multiparty = require('multiparty');

function subir(req, res, next) {
    var form = new multiparty.Form();

    form.on('error', function(error) { next(error); });

    form.on('part', function(part) {
        if (part.filename) {
           var out_stream = cloudinary.uploader.upload_stream(function(result) {
               if (! result.error) {
                  console.log("Cloudinary - OK");
                } else {
                  console.log("Cloudinary - Error: " + result.error.message);
                }
            });
            part.on('data', function(data) {out_stream.write(data);})
                .on('end',  function() {out_stream.end();})
                .on('error',function(error) {out_stream.end();})
        }
    });

    form.on('close', function() { res.send('Subida terminada.'); });

    form.parse(req);
};
```

- El API de Cloudinary proporciona varias formas de subir un fichero a sus servidores de almacenamiento.
  - Consultar la documentación de Cloudinary para más detalles.
- En el proyecto CoreBlog para subir ficheros a Cloudinary usaremos la función:
  - **cloudinary.uploader.upload_stream**.
- **cloudinary.uploader.upload_stream** proporciona un **writeStream** que usaremos para reenviar a Cloudinary el contenido del fichero que nos están subiendo según va llegando, sin escribir previamente el fichero en disco.
  - Nota: La función **upload_stream** no devuelve un objeto **writeStream** real, sino algo parecido que no soporta todos los eventos y métodos de los writeStreams. Internamente si crea un **writeStream**.
    - Esto provoca que no podamos conectar con un **readStream** usando **pipe**.
  - Lo que hacemos es atender los eventos de llegada de **datos**, **finalización** y **error** de **part** que es el **readStream** conectado al fichero que se está subiendo.

- Cuando termina la subida de datos a Cloudinary se invoca el callback pasado a **upload_stream**.
  - En caso de éxito, a este callback le pasa como parámetro un objeto con información sobre el fichero guardado:

```
{ public_id: 'vevipqr1ay3gopnzecbg',
  version: 1399535106,
  signature: 'e70fe8e126bb1',
  width: 820,
  height: 982,
  format: 'jpg',
  resource_type: 'image',
  created_at: '2014-05-08T07:45:06Z',
  bytes: 347967,
  type: 'upload',
  etag: 'e2b7f25905c3c72fa4be3008253f8d5b',
  url: 'http://res.cloudinary.com/core-upm/...../xx.jpg',
  secure_url: 'https://res.cloudinary.com/...../xx.jpg' }
```

    - **public_id** identifica al fichero.
    - **resource_type** indica si es una imagen.
    - **url** y **secure_url** son las URLS de acceso al fichero.

  - En caso de error, se pasa un mensaje explicativo:

```
{ error: { message: 'Invalid image file', http_code: 400 } }
```

- Cloudinary distingue entre ficheros con imágenes y otro tipo de ficheros (raw).
  - El tipo de fichero lo especificamos usando en los comandos del API la opción **resource_type**, pasando el valor "**image**" o "**raw**".
    - Podemos usar también el valor "**auto**" para que se detecte automáticamente el tipo.
    - El valor por defecto es "**image**".
  - En el proyecto CoreBlog solo adjuntamos imágenes, por lo que no usaremos la opción.

Obtener el URL de un Fichero

- Para obtener los datos de un fichero guardado en Cloudinary dado su **public_id**:

```
cloudinary.api.resource("xcbevjk3fkfoqu8qhvh0",
                        function(result) {
                           console.log(result);
                           if (! result.error) {
                              console.log('URL ='+result.url);
                           } else {
                              console.log(result.error.message);
                           }
                        });
```

- Para obtener un listado de todos los ficheros guardados en Cloudinary:

```
cloudinary.api.resources(function(result) {
                           console.log(result);
                           for (var i in result.resources) {
                              console.log(result.resources[i].url);
                           }
                        });
```

  - No se devuelven todos los recursos existentes. Se realiza paginación. Reinvocar nuevamente el mismo método pero pasando la opción **next_cursor** devuelta en la petición anterior.

Borrar Ficheros

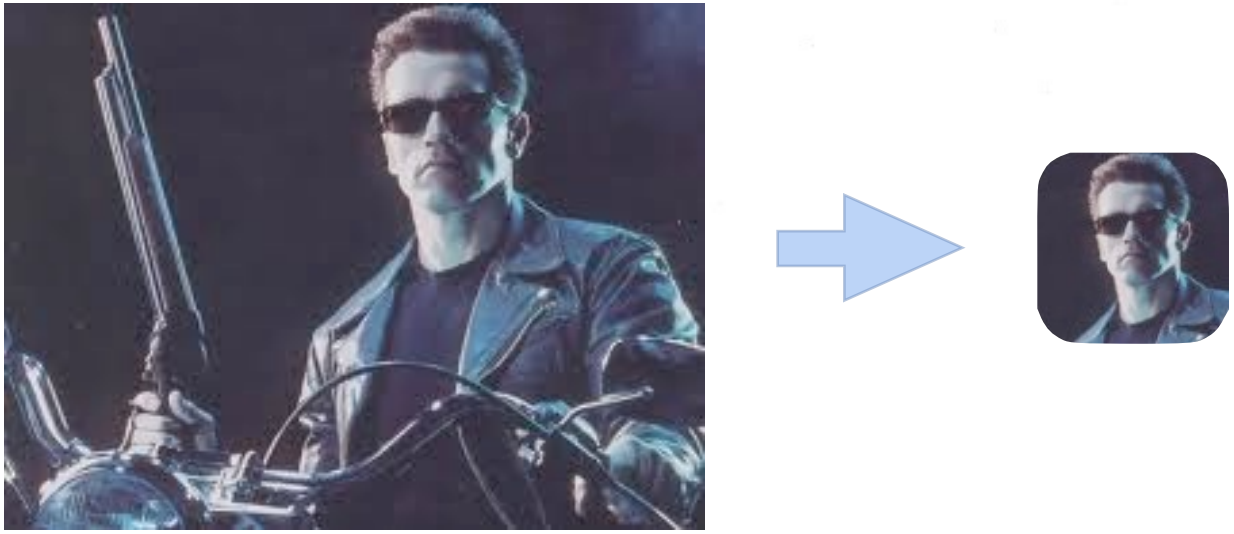
- Para borrar uno o más ficheros identificados por su **public_id**.

```
cloudinary.api.delete_resources(
        ["xcbevjk3fkfoqu8qhvh0",
         "ckjwpxrgk5to6ubxjqch"],
        function(result) {
           console.log(result);
        });
```

  - El primer parámetro puede ser el valor de un **public_id**, o un array de **public_id's**.

Transformaciones

- Cloudinary permite procesar los ficheros de imágenes: filtros de color, redimensionar, recortar, búsqueda de caras, marcas de agua, ...
  - Supongamos que el **public_id** de una imagen subida a Cloudinary es "terminator".



  - Podemos obtener un sello de tamaño 100x100 con los bordes redondeados y la cara de la persona de la imagen accediendo al URL:

```
http://res.cloudinary.com/core-upm/image/upload/
   c_thumb,g_face,w_100,h_100,r_20/terminator.jpg
```

CoreBlog:
Modelo, Rutas, Controlador, Vistas

Instalación de Paquetes

- Instalar **cloudinary**

```
$ npm install --save cloudinary
```

- Instalar **multiparty**

```
$ npm install --save multiparty
```

Definir Modelo **Attachment**

- En **models/attachment.js** añadimos la definición del modelo **Attachment.**
- En **models/index.js**:
  - Importamos la definición del modelo **Attachment**.
  - Declaramos la relación **1-a-N** con los **Posts**.

```
// Definicion del modelo Attachment:

module.exports = function(sequelize, DataTypes) {
  return sequelize.define('Attachment',
    { public_id: { type: DataTypes.STRING,
                   validate: {
                     notEmpty: { msg: "El campo public_id no puede estar vacío" }
                   }
      },
      url: { type: DataTypes.STRING,
             validate: {
                notEmpty: { msg: "El campo url no puede estar vacío" }
             }
      },
      filename: { type: DataTypes.STRING,
                  validate: {
                    notEmpty: { msg: "El campo filename no puede estar vacío" }
                  }
      },
      mime: { type: DataTypes.STRING,
              validate: {
                notEmpty: { msg: "El campo mime no puede estar vacío" }
              }
            }
    });
}
```

models/attachment.js

```
. . .
// Importar la definicion de las clases.
var Post = sequelize.import(path.join(__dirname,'post'));
var User = sequelize.import(path.join(__dirname,'user'));
var Comment = sequelize.import(path.join(__dirname,'comment'));
var Attachment = sequelize.import(path.join(__dirname,'attachment'));

// Relaciones
User.hasMany(Post, {foreignKey: 'AuthorId'});
User.hasMany(Comment, {foreignKey: 'AuthorId'});
Post.hasMany(Comment);
Post.hasMany(Attachment);
Post.belongsTo(User, {as: 'Author', foreignKey: 'AuthorId'});
Comment.belongsTo(User, {as: 'Author', foreignKey: 'AuthorId'});
Comment.belongsTo(Post);
Attachment.belongsTo(Post);

// Exportar los modelos:
exports.Post = Post;
exports.User = User;
exports.Comment = Comment;
exports.Attachment = Attachment;

// Crear las tablas en la base de datos que no se hayan creado aun.
sequelize.sync();
```

models/index.js

Definir las Rutas

- Cambios en **routes/index.js**:
  - Se carga el controlador de las imágenes adjuntas.
  - Se definen las rutas de las imágenes adjuntas:
    - Se ha optado por definirlas anidadas a los posts.
    - No se crearán rutas para editar las imágenes adjuntas.
    - Usar el middleware **sessionController.loginRequired** para forzar a que sólo puedan añadir imágenes adjuntas los usuarios logeados.
    - Usar el middleware **postController.loggedUserIsAuthor** para forzar a que sólo el autor del post pueda adjuntar imágenes al post.
  - Se configura el auto-load de los adjuntos.

```
. . .
var attachmentController = require('../controllers/attachment_controller');
. . .

/* Autoloading */
. . .
router.param('attachmentid', attachmentController.load);

/* Rutas de las imagenes adjuntas */

router.get('/posts/:postid([0-9]+)/attachments',
  attachmentController.index);

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

router.post('/posts/:postid([0-9]+)/attachments',
   sessionController.loginRequired,
   postController.loggedUserIsAuthor,
   attachmentController.create);

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

routes/index.js

Controlador de Adjuntos

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

// Tamaño maximo del fichero a subir.
const MAX_UPFILE_SIZE_KB = 500;

/*
*  Autoload: attachmentid.
*/
exports.load = function(req, res, next, id) {

   models.Attachment
        .find(id)
        .success(function(attachment) {
            if (attachment) {
                req.attachment = attachment;
                next();
            } else {
                req.flash('error', 'No existe ninguna imagen adjunta con id='+id+'.');
                next(new Error('No existe ninguna imagen adjunta con id='+id+'.'));
            }
        })
        .error(function(error) {
            next(error);
        });
};
```

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

    models.Attachment
        .findAll({where: {PostId: req.post.id},
                  order: [['updatedAt','DESC']]})
        .success(function(attachments) {
              res.render('attachments/index', {
                  attachments: attachments,
                  post: req.post
              });
        })
        .error(function(error) {
            next(error);
        });
};
```

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

    res.render('attachments/new', {post: req.post});
};
```

```
// POST /posts/33/attachments
exports.create = function(req, res, next) {

    var valid_adjunto = false; // true si he subido una imagen aceptable.

    var form = new multiparty.Form();

    form.on('error', function(error) {
      next(error);
    });

    form.on('close', function() {
        if ( ! valid_adjunto) { // False: contesto yo al navegador.
            req.flash('error','No se ha aceptado la imagen adjuntada.');
            res.redirect('/posts/' + req.post.id );
        }
    });
```

Continua el método ...

```
    form.on('part', function(part) {
        if (part.filename) { // esta parte sube un fichero
            if (part.byteCount > MAX_UPFILE_SIZE_KB*1024) { // Controlar tamaño maximo:
                req.flash('error', 'Tamaño máximo permitido es '+MAX_UPFILE_SIZE_KB+'KB.');
                part.resume(); // Emitir data y descartar contenido.
                return;
            }
           valid_adjunto = true; // El callback de Cloudinary envia respuesta al navegador.
            var out_stream = cloudinary.uploader.upload_stream(function(result) {
                console.log(result);
                if (! result.error) {
                    var attachment = models.Attachment.build({
                          public_id: result.public_id,
                          url: result.url,
                          filename: part.filename,
                          mime: part.headers["content-type"],
                          PostId: req.post.id});
                   attachment.save()
                       .success(function() {
                           req.flash('success', 'Adjunto subido con éxito.');
                           res.redirect('/posts/' + req.post.id );
                       })
                       .error(function(error) {next(error);});
                } else {
                  req.flash('error', result.error.message);
                  res.redirect('/posts/' + req.post.id );
                }
            });
            part.on('data', function(data) {out_stream.write(data);})
                .on('end',  function() {out_stream.end();})
                .on('error',function(error) {out_stream.end();})
        }
    });
    form.parse(req);
};
```

```
// DELETE /posts/33/attachments/66
exports.destroy = function(req, res, next) {

    // Borrar el fichero en Cloudinary.
    cloudinary.api.delete_resources(req.attachment.public_id,
                                    function(result) {});

    // Borrar entrada en la base de datos.
    req.attachment.destroy()
        .success(function() {
            req.flash('success', 'Adjunto eliminado con éxito.');
            res.redirect('/posts/' + req.post.id );
        })
        .error(function(error) {
            next(error);
        });
};
```

Vistas de Adjuntos

```
<h2>Nuevo Adjunto</h2>

<form method='post' action='/posts/<%= post.id %>/attachments'
      enctype='multipart/form-data' accept="image/*">

  <div class='field'>
    <label for="attachment">Seleccione imagen a adjuntar
                           (máximo 500KB):</label><br />
    <input type="file" id="attachment" name="adjunto" />
  </div>

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


</form>

<a href="/posts/<%= post.id %>"> Cancelar </a>
```

views/attachments/new.ejs

```
<h2> Adjuntos: <%= attachments.length %> </h2>

<ul>
   <% for (var i in attachments) { %>
      <li>
          <a href='<%= attachments[i].url %>' type='<%= attachments[i].mime %>'
             target='_blank'> <%= attachments[i].filename %> </a>
          (<em> <%= attachments[i].updatedAt.toLocaleDateString() %> </em>)

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

            <% var formname = 'fai' + i; %>
            <form style='display:inline;'
                  method='post'
                  action='/posts/<%= post.id %>/attachments/<%= attachments[i].id %>'
                  id='<%= formname %>'>
              <input type='hidden' name='_method' value='delete'>
              <a href=""
                 onclick="confirmarSubmit('¿Seguro que desea borrar la imagen?',
                                          '<%= formname %>'); return false"> Borrar </a>
            </form>
          <% } %>

          <br />
          <img src='<%= attachments[i].url%>' style='max-width:50%;' />
      </li>
   <% }; %>
</ul>

<% if (session.user && session.user.id == post.AuthorId) {%>
   <a href="/posts/<%= post.id %>/attachments/new"> Crear nuevo Adjunto </a>
<% } %>
```

views/attachments/index.ejs

Mostrar Adjuntos en **Post@show**

- Un buen sitio para mostrar los adjuntos de un **Post** es en la vista **show** de los posts.
  - Para ello, retocamos el método **show** de **postController** y la vista **views/posts/show.ejs**.

controllers/post_controller.js

```
// GET /posts/33
exports.show = function(req, res, next) {
    // Buscar el autor
    models.User
        .find(req.post.AuthorId)
        .success(function(user) {
            // Si encuentro al autor lo añado como el atributo author, si no {}.
            req.post.author = user || {};

            // Buscar imagenes adjuntas
            req.post.getAttachments({order: [['updatedAt','DESC']]})
               .success(function(attachments) {
                    // Buscar comentarios del post
                    models.Comment
                        .findAll({where: {PostId: req.post.id},
                                  order: [['updatedAt','DESC']],
                                  include: [{ model: models.User, as: 'Author' }]
                        })
                        .success(function(comments) {
                           var new_comment = models.Comment.build({
                                       body: 'Introduzca el texto del comentario'});
                           res.render('posts/show', {
                               post: req.post,
                               comments: comments,
                               comment: new_comment,
                               attachments: attachments,
                               validate_errors: {}
                           });
                        }).error(function(error) {next(error);});
                }).error(function(error) {next(error);});
        }).error(function(error) { next(error); });
};
```

```
<h2>Post</h2>

<article>
    <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"> Editar </a>
    <% } %>
</article>

<hr />
<%- include ../attachments/index.ejs %>
<hr />
<%- include ../comments/index.ejs %>
<hr />

<% if (session.user) { %>
   <blockquote>
      <%- include ../comments/new.ejs %>
   </blockquote>
   <hr />
<% } %>

<a href="/posts"> Volver al índice de Posts</a>
```

views / posts / show.ejs

Borrar Adjuntos al Borrar un Post

post_controller.js

```
exports.destroy = function(req, res, next) {
    var Sequelize = require('sequelize');
    var chainer = new Sequelize.Utils.QueryChainer
    var cloudinary = require('cloudinary');

    req.post.getComments() // Obtener los comentarios
       .success(function(comments) {
           for (var i in comments) {
               chainer.add(comments[i].destroy()); // Eliminar un comentario
           }
           req.post.getAttachments() // Obtener los adjuntos
              .success(function(attachments) {
                  for (var i in attachments) {
                      chainer.add(attachments[i].destroy()); // Eliminar un adjunto

                      // Borrar el fichero en Cloudinary.
                      cloudinary.api.delete_resources(attachments[i].public_id,
                                    function(result) {});
                  }
                  chainer.add(req.post.destroy()); // Eliminar el post
                   chainer.run()   // Ejecutar el chainer
                     .success(function(){
                         req.flash('success', 'Post eliminado con éxito.');
                         res.redirect('/posts');
                     })
                      .error(function(errors){next(errors[0]);})})
              .error(function(error) {next(error);});})
       .error(function(error) {next(error);});};
```

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 tema8:master
(local)$ git push heroku master
```

Copiar en la rama **master** de **Heroku**. El primer comando copia en contenido **local** de la rama **tema8** 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

Pregunta

- Añadir al recurso User la foto del usuario.
  - Guardándola en Cloudinary.