Carga de archivos usando Angular4 / Microservice

por Shamik Mitra MVB · 20 y 18 de marzo · Zona de microservicios · Tutorial

Aprenda a realizar un seguimiento de los microservicios implementados en infraestructura elástica, como contenedores o nubes, donde los nodos aumentan y disminuyen rápidamente.

Cargar un archivo es una característica regular de la programación web. Todas las empresas necesitan esta capacidad, y sabemos cómo cargar un archivo utilizando JSP / HTML como front-end y servlet / struts / Spring MVC como extremo del servidor. Pero, ¿cómo se puede lograr con una combinación de Angular 4 / microservicio?

En este tutorial, te mostraré paso a paso, pero antes de eso, déjame aclarar una cosa: supongo que tienes una comprensión básica de Angular 4 y microservicios.

Ahora saltemos directamente en la declaración del problema. Quiero crear una funcionalidad de carga que invoca un servicio de microservicio de FileUpload y almacena la imagen de perfil de un empleado.

Vamos a crear un proyecto de Angular 4 usando el complemento Angular.io en Eclipse. Después de crear la aplicación, modifique el archivo **app.component.ts** en el módulo de la aplicación.

```
importar { UploadFileService } de  './fileUploa
import { Component } de  '@ angular / core' ;
```

```
importar { HttpClient , HttpResponse , HttpEver

@Component ({
  selector : 'aplicación-raíz' ,
  templateUrl : './view.component.html' ,
  styleUrls : [ './app.component.css' ],
  proveedores : [ UploadFileService ]
})

 clase de  exportación AppComponent {
selectedFiles : FileList ;
   currentFileUpload : archivo ;
    constructor ( uploadService privado  : Uplc
  selectFile ( evento ) {
    esta . selectedFiles  =  evento . objetivo
  }
  upload () {

    esta . currentFileUpload  =  this . selecte
    esta . uploadService . pushFileToStorage (
     if ( instancia de evento de  HttpResponse
        consola . log ( '¡El archivo está compl
      }
    });

    esta . selectedFiles  =  undefined ;
  }

}
```

En el decorador @Component, cambié la URL de la plantilla a view.component.html, que en realidad contiene los componentes del formulario FileUpload. Después de eso, agrego un UploadService como proveedor, que realmente publica los archivos seleccionados en el microservicio.

Ahora, defino un método llamado selectFile, que captura un evento (evento OnChange en el campo de formulario fileUpload) y extrae el archivo de los campos de formulario de destino, en este caso, los

campos de formulario de archivo.

Luego agrego otro método llamado `upload` , que llama al servicio de carga de archivos y se suscribe a Observable <HttpResponse>.

Aquí está el archivo view.component.html:

```
< div  style = "text-align: center" >
        < etiqueta >
                < input  type = "file"  (change
                </ label >
                < button  [disabled] = "! selec
(click) = "cargar ()" > Cargar </ button >
        </ div >
```

Aquí, acabo de agregar un campo de carga de archivos, y cuando seleccionamos un archivo, se activa un evento onchange, que llama al método selectFile y le pasa ese evento.

Luego, llamo al método de carga.

Veamos el servicio de carga de archivos.

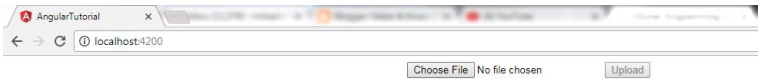
```
import { Injectable } de  '@ angular / core' ;
importar { HttpClient , HttpRequest , HttpEvent
importar { Observable } desde  'rxjs / Observab

@Injectable ()
 clase de  exportación UploadFileService {

  constructor ( http privado  : HttpClient ) {}

  pushFileToStorage ( file : File ): Observable
    const  formdatata : FormData  =  new  FormD
    formdata . append ( 'archivo' , archivo );

    const  req  =  new  HttpRequest ( 'POST' ,
      reportProgress : cierto ,
      responseType : 'texto'
```

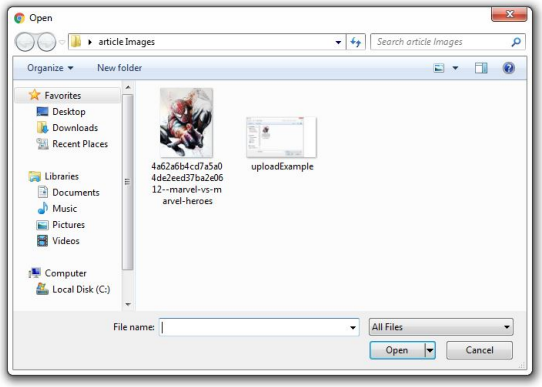
```
    }

    );

    devolver  este . http . solicitud ( req );
  }

}
```

Aquí creé un objeto Formdata y agregué el archivo cargado en él. Al utilizar Angular HTTP, publico los datos del formulario en un microservicio que se ejecuta en el puerto 8085 y publico un punto final REST llamado / profile / uploadpicture.

¡Hurra, escribimos con éxito la parte de la interfaz de usuario para subir archivos usando Angular4!

Si inicia Angular (ng serve), se verá así:





Construyamos la parte de microservicio. Cree un proyecto llamado servicio EmployeeFileUpload en STS o utilizando start.spring.io. Seleccione un módulo de cliente Eureka para registrar este microservicio con Eureka.

Después de eso, cambie el nombre de application.properties a la propiedad bootstrap. Agregue la siguiente entrada:

```
spring.unpplication.nunme
eureka.client.serviceUrl
server.port=8085
security.basic.enable:fa
```

```
4
management . security . enab
5
```

Mi servidor Eureka está ubicado en el puerto 9091. Doy un nombre lógico a este microservicio, llamándolo EmployeePhotoStorageService, que se ejecuta en el puerto 8085.

Ahora voy a crear un controlador REST, que acepta la solicitud de Angular y vincula el formulario multiparte.

Veamos los fragmentos de código de FileUploadController:

```
paquete  com . ejemplo . EmployeePhotoStorageSe
1
2
import  org . springframework . habas . fábrica
3
import  org . springframework . http . HttpStat
4
import  org . springframework . http . Response
5
import  org . springframework . estereotipo . C
6
import  org . springframework . web . se unen .
7
import  org . springframework . web . se unen .
8
import  org . springframework . web . se unen .
9
import  org . springframework . web . se unen .
10
import  org . springframework . web . se unen .
11
import  org . springframework . web . multipart
12
13
14
@RestController
15
16
 clase  pública FileController {
17
18
 @Autowired
19
 FileService  fileservice ;
20
21
 @CrossOrigin ( origins  =  "http: // localhost
22
 @PostMapping ( "/ profile / uploadpicture" )
```

```
23
    public ResponseEntity < String > handleFi
24
     Cadena de mensaje = "" ;
25
     prueba {
26
      fileservice . tienda ( archivo );
27
      message = "Ha cargado correctamente" + a
28
      return ResponseEntity . estado ( HttpStatus
29
     } catch ( Exception e ) {
30
      message = "Error al cargar la imagen de pe
31
      return ResponseEntity . estado ( HttpStatus
32
     }
33
    }
34
35
36
37  }
```

Algunas cosas para notar aquí: utilizo una anotación @CrossOrigin, y con esto, ordeno a Spring que permita que la solicitud provenga de localhost: 4200. En producción, el microservicio y la aplicación angular se alojan en diferentes dominios; para permitir la solicitud del otro dominio, debemos proporcionar la anotación de origen cruzado. Conecté automáticamente un servicio de FileUpload que realmente escribe el contenido del archivo en el disco.

Veamos el código FileService:

```
paquete com . ejemplo . EmployeePhotoStorageSe
1
2
importar java . nio . archivo . Archivos ;
3
importar java . nio . archivo . Camino ;
4
importar java . nio . archivo . Senderos ;
5
6
import org . springframework . estereotipo . S
7
import org . springframework . web . multipart
8
9
10
@Servicio
11
 clase pública FileService {
12
 private final Path rootLocation = Paths .
```

```
13
14
15
16
17
18
19
20
21
22
23
24
25
```

```
public  void  store ( archivo MultipartFile  )
 prueba {
  Sistema . a cabo . println ( archivo . getOr
  Sistema . a cabo . println ( rootLocation .
  Archivos . copie ( archivo . getInputStream
  } catch ( Exception  e ) {
  lanza una  nueva  RuntimeException ( "FAIL!"
  }
 }

}
```

Aquí, creo un directorio llamado ProfilePictureStore en el proyecto; es el mismo nivel que la carpeta src. Ahora copio el flujo de entrada del archivo a la ubicación usando el método estático Files.copy () de java.nio.

Ahora, para ejecutar este microservicio, tengo que escribir el archivo de inicio de la aplicación Spring. Veamos el código:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
paquete  com . ejemplo . EmployeePhotoStorageSe

import  org . springframework . arranque . Spri
import  org . springframework . arranque . autc
import  org . springframework . arranque . web
import  org . springframework . nube . cliente
import  org . springframework . nube . cliente
import  org . springframework . nube . netflix
import  org . springframework . contexto . anot
import  org . springframework . web . cliente .


@EnableDiscoveryClient
@SpringBootApplication
```

```
public  class  EmployeePhotoStorageService {

 public  static  void  main ( String [] args )

  SpringApplication . ejecutar ( EmployeePhotoS

 }


}
```

Ok, estamos todos listos. Solo falta la última pieza de este tutorial: el archivo pom.xml.

```
<? xml  version = "1.0"  encoding = "UTF-8" ?>

< proyecto
        xmlns = "http://maven.apache.org/POM/4.
        xmlns : xsi = "http://www.w3.org/2001/X
xsi : schemaLocation = "http://maven.apache.org
        < modelVersion > 4.0 . 0 </ modelVersio
        < groupId > com . ejemplo </ groupId >
        < artifactId > EmployeeFileUploadServic
        < versión > 0.0 . 1 - SNAPSHOT </ versi
        < packaging > jar </ packaging >
        < name > EmployeeDashBoardService </ na
        < description > Proyecto de demostració
        < padre >
                < groupId > org . springframewo
                < artifactId > spring - boot -
                < versión > 1.5 . 4. LANZAMIENT
                < relativePath />
                <! -  buscar  padre  del  repos
        </ parent >
        < propiedades >
                < proyecto . construir . source
                < proyecto . la presentación de
```

```
22                  < java . versión > 1.8 </ java
23                  < primavera - nube . versión >
24          </ propiedades >
25          < dependencias >
26                  < dependencia >
27                          < groupId > org . sprir
28                          < artifactId > spring -
29                  </ dependency >
30                  < dependencia >
31                          < groupId > org . sprir
32                          < artifactId > spring -
33                  </ dependency >
34                  < dependencia >
35                          < groupId > org . sprir
36                          < artifactId > spring -
37                  </ dependency >
38                  < dependencia >
39                          < groupId > org . sprir
40                          < artifactId > spring -
41                  </ dependency >
42                  < dependencia >
43                          < groupId > org . sprir
44                          < artifactId > spring -
45                  </ dependency >
46                  < dependencia >
47                          < groupId > org . sprir
48                          < artifactId > spring -
49                          < scope > prueba </ scc
50                  </ dependency >
51                  < dependencia >
52                          < groupId > org . sprir
53                          < artifactId > spring -
54                  </ dependency >
55
```

```
56                  < dependencia >
                            < groupId > org . sprir
57
                            < artifactId > spring -
58
                    </ dependency >
59
60                  < dependencia >
                            < groupId > org . sprir
61
                            < artifactId > spring -
62
                    </ dependency >
63
64          </ dependencias >
65          < dependencyManagement >
66                  < dependencias >
67                          < dependencia >
                                    < groupId > org
68
                                    < artifactId >
69
                                    < version > $ {
70
                                    < tipo > pom </
71
                                    < scope > impor
72
                            </ dependency >
73
                    </ dependencias >
74
75          </ dependencyManagement >
76          < construir >
77                  < plugins >
78                          < plugin >
                                    < groupId > org
79
                                    < artifactId >
80
                            </ plugin >
81
                    </ plugins >
82
83          </ build >
84  </ project >
```

Eso es. Si ejecutamos el microservicio y cargamos un archivo desde Angular, podemos ver que el archivo se almacena en la carpeta ProfilePictureStore. Muy fácil, ¿no?

Conclusión

Este es un ejemplo muy simple, o puedo decir, un prototipo de carga de archivos, sin validación ni transmisión de información desde la UI, aparte del archivo sin formato como comentarios, nombre de archivo, etiquetas, etc. Puede enriquecer este ejemplo básico utilizando el Objeto Formdata en Angular. Otra observación: llamo directamente a la instancia de microservicio de Angular. Ese no es el caso en producción, donde tiene que introducir un gateway Zuul API que acepte la solicitud de Angular, realice alguna comprobación de seguridad, luego se comunique con Eureka y enrute la solicitud al microservicio real para simplificar: me salté esa parte.

Temas: ANGULAR 4, ANGULAR, MICROSERVICIOS, TUTORIAL

Microservices Partner Resources

Pruebas funcionales de API: cómo hacerlo bien

por Stephen Feloney MVB · 22, 18 de marzo · Zona de rendimiento · revisión

API, o interfaz del programa de aplicación, es un sistema de métodos de comunicación que ofrece a desarrolladores y no desarrolladores acceso a programas, procedimientos, funciones y servicios. El protocolo más común utilizado en API es HTTP, junto con la arquitectura REST . Los desarrolladores que programan usando REST hacen que su código sea fácil de entender. Ellos y otros saben qué idioma usarán, cómo funcionan las funciones, qué parámetros se pueden usar, etc.

Los marcos populares para desarrollar API incluyen Swagger, WADL y RAML. Idealmente, al programar, los desarrolladores forman un "Contrato API", que describe cómo deben consumirse los servicios desarrollados en las API.

Prior to this standardization, programming was like the Wild West. Developers gave access to their code in the way they saw fit, and it was hard to develop public services and make them available because there were many ways to code. SOAP was the first attempt at standardization, but now REST is the dominant player.

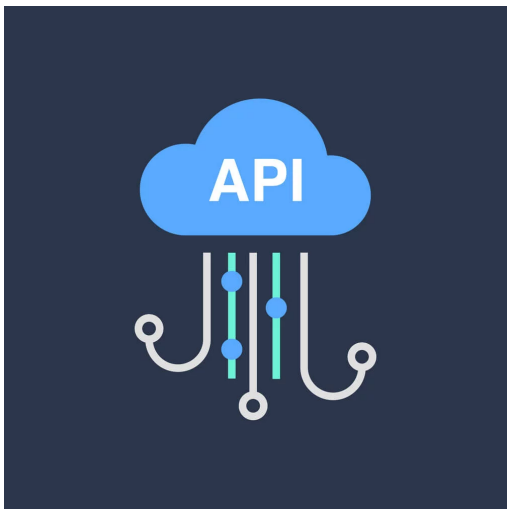
API testing creates a more reliable code. But historically, testing would take place at the GUI level. When a developer would finish their work, they would hand it off to the QA engineer. The engineers had limited time so they would test the code at the highest

level, the GUI. This would cover both the frontend and the backend development.

This worked for manual testing and for the beginning of automation testing, but isn't right for the age of agile and continuous testing. GUI testing is too brittle and GUI automated scripts break easily. In addition, teams can't wait for the entire system to be updated and the GUI to be ready before testing occurs.

In the age of agile, testing must take place at a lower level, i.e at the API level. Developers can even do it themselves. API tests, because of "API contracts", can even be created before development is complete. This means developers can validate their code based on pre-written tests (aka Test Driven Development).

But despite the known importance of API testing, it doesn't always get done. Agile Developers just don't have time. On average developers only code one day a week, the rest of their time is taken up with testing, documentation, validation, and meetings. So they try to have hardening sprints (that isn't really agile, is it?) where manual testing takes place, but it just takes too long. It's very difficult to be able to get functional API testing done in the two weeks you also need to develop, test, validate and get the documentation complete.



Automating API testing makes developing faster and clears up developers' time to do other things, like write code. Automating also enables covering the full scope of tests more easily: positive, negative, edge case, SQL injection, etc. This ensures nothing is left to chance and all parameters and permutations are tested. Agile development groups that attempt to test their APIs might test one or two positive test flows, or one positive and one negative,

and call that a success. But this is not thorough API testing and opens the door for unnecessary release risk because many variants are missed and full validation isn't achieved.

For example, let's say an API takes an author's name and a book release date. A positive flow will test the name and date and see if they work. Once the response is properly received the API works. Supposedly.

But what about the negative and edge cases? Fro example, inserting a proper date but no book, or changing the date format, or a correct date format for a year that doesn't exist, or a long name, or inserting a SQL code that grants data to the DB, etc. These are just a few examples out of many variants that need to be tested, even though they are not covered in the contract.

Developers and testers need an easy way to create tests that cover all of these aspects. We recommend you look for a solution that can take your Swagger or other framework files, test them comprehensively according to your API contract, and run them as part of your Continuous Integration process. This ensures you can focus on developing strong and durable code.

To start your API Functional tests you can request a BlazeMeter demo or simply put your URL in the box below to start testing.

Topics: CONTINUOUS INTEGRATION, DEVOPS, FUNCTIONAL API TESTING, API TESTING, SWAGGER, SOAP, DEVELOPERS, REST ARCHITECTURE, AGILITY, LOAD API TESTING

Published at DZone with permission of Stephen Feloney , DZone MVB. See the original article here.