# CONTROLADOR CARGA DE ARCHIVOS EN EL SERVIDOR

Aquí tienes cómo podrías hacerlo:

### Crear un UploadController

#### app/controllers/UploadController.php

php

Copiar código

namespace App\Controllers;

use Core\Controller;

class UploadController extends Controller {

/\*\*

\* ---------------------------------------------------------------------

\* @route POST upload/file

\* ---------------------------------------------------------------------

\* Maneja la subida de archivos al servidor.

\*

\* @return void

\*/

public function uploadFile() {

if ($\_SERVER['REQUEST\_METHOD'] === 'POST' && isset($\_FILES['file'])) {

$targetDir = "uploads/";

$targetFile = $targetDir . basename($\_FILES["file"]["name"]);

$uploadOk = 1;

$imageFileType = strtolower(pathinfo($targetFile, PATHINFO\_EXTENSION));

// Verificar si el archivo es una imagen real

$check = getimagesize($\_FILES["file"]["tmp\_name"]);

if ($check !== false) {

echo "File is an image - " . $check["mime"] . ".";

$uploadOk = 1;

} else {

echo "File is not an image.";

$uploadOk = 0;

}

// Verificar si el archivo ya existe

if (file\_exists($targetFile)) {

echo "Sorry, file already exists.";

$uploadOk = 0;

}

// Verificar el tamaño del archivo

if ($\_FILES["file"]["size"] > 500000) { // 500KB máximo

echo "Sorry, your file is too large.";

$uploadOk = 0;

}

// Permitir ciertos formatos de archivo

if ($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType != "jpeg"

&& $imageFileType != "gif") {

echo "Sorry, only JPG, JPEG, PNG & GIF files are allowed.";

$uploadOk = 0;

}

// Verificar si $uploadOk está configurado a 0 por un error

if ($uploadOk == 0) {

echo "Sorry, your file was not uploaded.";

// Si todo está bien, intenta subir el archivo

} else {

if (move\_uploaded\_file($\_FILES["file"]["tmp\_name"], $targetFile)) {

echo "The file ". basename($\_FILES["file"]["name"]). " has been uploaded.";

} else {

echo "Sorry, there was an error uploading your file.";

}

}

} else {

echo "No file uploaded.";

}

}

}

### Definir la Ruta en el Servidor

#### routes.php

php

Copiar código

return [

// Otras rutas...

'user/register' => ['handler' => 'UserController@register', 'method' => 'GET'],

'user/create' => ['handler' => 'UserController@create', 'method' => 'POST'],

// ...

'upload/file' => ['handler' => 'UploadController@uploadFile', 'method' => 'POST'],

];

### Resumen

* **UploadController**: Este nuevo controlador contiene el método uploadFile() que maneja la subida de archivos.
* **Rutas**: La ruta upload/file está asociada con el método uploadFile() del UploadController.

Con esta estructura, el UploadController se ocupa específicamente de la funcionalidad de subida de archivos, mientras que el UserController sigue enfocado en la lógica relacionada con los usuarios. Esto mejora la modularidad y la claridad del código.

# CARGA DE ARCHIVOS DESDE CLIENTE ANDROID

Para enviar una imagen capturada desde una aplicación Android a tu servidor mediante una solicitud POST, puedes seguir estos pasos. A continuación, te muestro un ejemplo paso a paso que incluye la captura de la imagen, su conversión a un archivo, y el envío al servidor utilizando Retrofit, una biblioteca de Android para hacer solicitudes HTTP.

**1. Configuración del Proyecto**

Primero, añade Retrofit y otras dependencias necesarias a tu archivo build.gradle:

groovy

Copiar código

dependencies {

implementation 'com.squareup.retrofit2:retrofit:2.9.0'

implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

implementation 'com.squareup.retrofit2:converter-scalars:2.9.0'

implementation 'com.squareup.okhttp3:logging-interceptor:4.9.0'

}

**2. Captura de la Imagen**

En tu actividad o fragmento, añade código para capturar una imagen usando Intent:

java

Copiar código

private static final int REQUEST\_IMAGE\_CAPTURE = 1;

private Uri photoUri;

private void dispatchTakePictureIntent() {

Intent takePictureIntent = new Intent(MediaStore.ACTION\_IMAGE\_CAPTURE);

if (takePictureIntent.resolveActivity(getPackageManager()) != null) {

// Create the File where the photo should go

File photoFile = null;

try {

photoFile = createImageFile();

} catch (IOException ex) {

// Handle error

}

// Continue only if the File was successfully created

if (photoFile != null) {

photoUri = FileProvider.getUriForFile(this, "com.example.android.fileprovider", photoFile);

takePictureIntent.putExtra(MediaStore.EXTRA\_OUTPUT, photoUri);

startActivityForResult(takePictureIntent, REQUEST\_IMAGE\_CAPTURE);

}

}

}

@Override

protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {

super.onActivityResult(requestCode, resultCode, data);

if (requestCode == REQUEST\_IMAGE\_CAPTURE && resultCode == RESULT\_OK) {

// Handle the image here

uploadImage(new File(photoUri.getPath()));

}

}

private File createImageFile() throws IOException {

// Create an image file name

String timeStamp = new SimpleDateFormat("yyyyMMdd\_HHmmss").format(new Date());

String imageFileName = "JPEG\_" + timeStamp + "\_";

File storageDir = getExternalFilesDir(Environment.DIRECTORY\_PICTURES);

File image = File.createTempFile(

imageFileName, /\* prefix \*/

".jpg", /\* suffix \*/

storageDir /\* directory \*/

);

return image;

}

**3. Configuración de Retrofit**

Configura Retrofit para manejar la solicitud POST:

java

Copiar código

public interface ApiService {

@Multipart

@POST("user/upload")

Call<ResponseBody> uploadImage(@Part MultipartBody.Part file);

}

public class ApiClient {

private static final String BASE\_URL = "http://your-server-url/mvc\_app/public/";

private static Retrofit retrofit = null;

public static Retrofit getClient() {

if (retrofit == null) {

OkHttpClient client = new OkHttpClient.Builder()

.addInterceptor(new HttpLoggingInterceptor().setLevel(HttpLoggingInterceptor.Level.BODY))

.build();

retrofit = new Retrofit.Builder()

.baseUrl(BASE\_URL)

.client(client)

.addConverterFactory(GsonConverterFactory.create())

.build();

}

return retrofit;

}

}

**4. Subida de la Imagen**

Sube la imagen al servidor utilizando Retrofit:

java

Copiar código

private void uploadImage(File file) {

ApiService apiService = ApiClient.getClient().create(ApiService.class);

// Create a request body with file and image media type

RequestBody requestFile = RequestBody.create(MediaType.parse("image/jpeg"), file);

// MultipartBody.Part is used to send also the actual file name

MultipartBody.Part body = MultipartBody.Part.createFormData("file", file.getName(), requestFile);

// Finally, execute the request

Call<ResponseBody> call = apiService.uploadImage(body);

call.enqueue(new Callback<ResponseBody>() {

@Override

public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {

if (response.isSuccessful()) {

// Handle success

Log.d("Upload", "Success");

} else {

// Handle failure

Log.e("Upload", "Failed: " + response.message());

}

}

@Override

public void onFailure(Call<ResponseBody> call, Throwable t) {

// Handle error

Log.e("Upload", "Error: " + t.getMessage());

}

});

}

**5. Permisos**

No olvides añadir los permisos necesarios en tu AndroidManifest.xml:

xml

Copiar código

<uses-permission android:name="android.permission.CAMERA"/>

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/>

<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"/>

<application

...

<provider

android:name="androidx.core.content.FileProvider"

android:authorities="${applicationId}.fileprovider"

android:exported="false"

android:grantUriPermissions="true">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/file\_paths"/>

</provider>

</application>

Crea el archivo file\_paths.xml en res/xml:

xml

Copiar código

<?xml version="1.0" encoding="utf-8"?>

<paths xmlns:android="http://schemas.android.com/apk/res/android">

<external-files-path

name="my\_images"

path="Pictures" />

</paths>

**Resumen**

Este flujo captura una imagen utilizando la cámara del dispositivo, guarda la imagen en almacenamiento externo, y luego sube la imagen al servidor mediante una solicitud POST usando Retrofit. Asegúrate de manejar adecuadamente los permisos y la seguridad tanto en la aplicación Android como en el servidor.

# CARGA DE ARCHIVOS DESDE EL CLIENTE WEB

Para permitir la carga de archivos desde el cliente al servidor de manera segura, eficiente y lógica, necesitas implementar varios aspectos clave, tanto del lado del cliente como del servidor. Aquí te presento una guía paso a paso:

### 1. Configuración del Formulario HTML para la Carga de Archivos

Primero, necesitas un formulario HTML que permita al usuario seleccionar y cargar un archivo:

html

Copiar código

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Upload File</title>

<link rel="stylesheet" href="<?php echo BASE\_URL; ?>css/styles.css">

</head>

<body>

<h1>Upload File</h1>

<form id="uploadForm" enctype="multipart/form-data">

<label for="file">Select file:</label>

<input type="file" id="file" name="file" required>

<button type="submit" id="btnUploadFile">Upload</button>

</form>

<div id="salida"></div>

<script type="module" src="<?php echo BASE\_URL; ?>js/upload.js"></script>

</body>

</html>

### 2. JavaScript para Manejar la Carga de Archivos

Utiliza JavaScript para capturar el evento de envío del formulario y enviar el archivo al servidor utilizando fetch:

javascript

Copiar código

// public/js/upload.js

document.getElementById("uploadForm").addEventListener("submit", uploadFile);

async function uploadFile(event) {

event.preventDefault(); // Evitar el envío del formulario tradicional

const formData = new FormData();

const fileInput = document.getElementById("file");

formData.append("file", fileInput.files[0]);

try {

const response = await fetch("<?php echo BASE\_URL; ?>user/upload", {

method: "POST",

body: formData

});

const result = await response.json();

if (response.ok) {

alert("File uploaded successfully");

document.getElementById("salida").innerHTML = "Result: " + JSON.stringify(result);

} else {

alert("Failed to upload file: " + result.message);

}

} catch (error) {

alert("Error: " + error.message);

}

}

### Ajustar el UploadController para Soportar Múltiples Archivos

#### app/controllers/UploadController.php

php

Copiar código

namespace App\Controllers;

use Core\Controller;

class UploadController extends Controller {

/\*\*

\* ---------------------------------------------------------------------

\* @route POST upload/files

\* ---------------------------------------------------------------------

\* Maneja la subida de múltiples archivos al servidor.

\*

\* @return void

\*/

public function uploadFiles() {

if ($\_SERVER['REQUEST\_METHOD'] === 'POST' && isset($\_FILES['files'])) {

$total = count($\_FILES['files']['name']);

$responses = [];

for ($i = 0; $i < $total; $i++) {

$targetDir = "uploads/";

$targetFile = $targetDir . basename($\_FILES["files"]["name"][$i]);

$uploadOk = 1;

$imageFileType = strtolower(pathinfo($targetFile, PATHINFO\_EXTENSION));

// Verificar si el archivo es una imagen real

$check = getimagesize($\_FILES["files"]["tmp\_name"][$i]);

if ($check !== false) {

$uploadOk = 1;

} else {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'File is not an image.'

];

$uploadOk = 0;

}

// Verificar si el archivo ya existe

if (file\_exists($targetFile)) {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'File already exists.'

];

$uploadOk = 0;

}

// Verificar el tamaño del archivo

if ($\_FILES["files"]["size"][$i] > 500000) { // 500KB máximo

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'File is too large.'

];

$uploadOk = 0;

}

// Permitir ciertos formatos de archivo

if ($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType != "jpeg"

&& $imageFileType != "gif") {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'Only JPG, JPEG, PNG & GIF files are allowed.'

];

$uploadOk = 0;

}

// Verificar si $uploadOk está configurado a 0 por un error

if ($uploadOk == 0) {

if (!isset($responses[$i])) {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'File was not uploaded.'

];

}

// Si todo está bien, intenta subir el archivo

} else {

if (move\_uploaded\_file($\_FILES["files"]["tmp\_name"][$i], $targetFile)) {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'success',

'message' => 'File has been uploaded.'

];

} else {

$responses[] = [

'file' => $\_FILES["files"]["name"][$i],

'status' => 'failed',

'message' => 'There was an error uploading your file.'

];

}

}

}

// Enviar respuesta JSON

header('Content-Type: application/json');

echo json\_encode($responses);

} else {

echo "No files uploaded.";

}

}

}

### Definir la Ruta en el Servidor

#### routes.php

php

Copiar código

return [

// Otras rutas...

'user/register' => ['handler' => 'UserController@register', 'method' => 'GET'],

'user/create' => ['handler' => 'UserController@create', 'method' => 'POST'],

// ...

'upload/files' => ['handler' => 'UploadController@uploadFiles', 'method' => 'POST'],

];

### Código del Cliente para Subir Múltiples Archivos

#### HTML

html

Copiar código

<!DOCTYPE html>

<html>

<head>

<title>Upload Files</title>

</head>

<body>

<h1>Upload Multiple Files</h1>

<form id="uploadFilesForm" enctype="multipart/form-data">

<input type="file" id="files" name="files[]" multiple required>

<br>

<button type="submit">Upload Files</button>

</form>

<div id="output"></div>

<script type="module" src="/mvc\_app/public/js/upload.js"></script>

</body>

</html>

#### JavaScript

javascript

Copiar código

// upload.js

document.getElementById('uploadFilesForm').addEventListener('submit', function(event) {

event.preventDefault();

const formData = new FormData();

const files = document.getElementById('files').files;

for (let i = 0; i < files.length; i++) {

formData.append('files[]', files[i]);

}

fetch('/mvc\_app/public/upload/files', {

method: 'POST',

body: formData,

})

.then(response => response.json())

.then(data => {

const output = document.getElementById('output');

output.innerHTML = '';

data.forEach(fileResponse => {

const message = `${fileResponse.file}: ${fileResponse.message}`;

const p = document.createElement('p');

p.textContent = message;

output.appendChild(p);

});

})

.catch(error => {

console.error('Error:', error);

});

});

### Consideraciones

1. **Subida de Archivos en el Servidor**: Manejar múltiples archivos simultáneamente y verificar que cumplan con los requisitos.
2. **Respuestas Informativas**: Proporcionar respuestas detalladas para cada archivo, indicando el éxito o el fallo y la razón de cualquier error.
3. **Cliente**: Uso de FormData en el cliente para manejar la subida de múltiples archivos.
4. **Escalabilidad y Seguridad**: Considerar el uso de almacenamiento en la nube para manejar un gran número de archivos y garantizar la seguridad y escalabilidad del sistema.

Esta estructura asegura que tu aplicación pueda manejar múltiples archivos eficientemente mientras mantiene la claridad y la separación de responsabilidades en el código.

**Implementación con Resumable Uploads (TUS Protocol)**

**Servidor (PHP con TUS):**

Para implementar el protocolo TUS en un servidor PHP, puedes usar una biblioteca como tus-php.

1. **Instalar tus-php**:

bash

Copiar código

composer require tus-php/tus-php

1. **Configurar el Servidor TUS**:

php

Copiar código

// server.php

require 'vendor/autoload.php';

use TusPhp\Tus\Server as TusServer;

$server = new TusServer('redis'); // or 'file' to use file-based storage

$server->setApiPath('/mvc\_app/public/upload/tus') // Tus server endpoint.

->setUploadDir('/path/to/uploads'); // Directory to store uploaded files.

$response = $server->serve();

$response->send();

1. **Configurar la Ruta en el Servidor**:

php

Copiar código

return [

// Otras rutas...

'upload/tus' => ['handler' => 'UploadController@uploadFiles', 'method' => 'POST'],

];

**Cliente (JavaScript con tus-js-client):**

1. **Instalar tus-js-client**:

bash

Copiar código

npm install tus-js-client

1. **Código del Cliente para Subir Archivos**:

javascript

Copiar código

import tus from 'tus-js-client';

document.getElementById('uploadFilesForm').addEventListener('submit', function(event) {

event.preventDefault();

const files = document.getElementById('files').files;

for (let i = 0; i < files.length; i++) {

const file = files[i];

const upload = new tus.Upload(file, {

endpoint: 'http://localhost/mvc\_app/public/upload/tus',

retryDelays: [0, 1000, 3000, 5000],

metadata: {

filename: file.name,

filetype: file.type

},

onError: function(error) {

console.log('Failed because: ' + error);

},

onProgress: function(bytesUploaded, bytesTotal) {

const percentage = (bytesUploaded / bytesTotal \* 100).toFixed(2);

console.log(bytesUploaded, bytesTotal, percentage + '%');

},

onSuccess: function() {

console.log('Download %s from %s', upload.file.name, upload.url);

}

});

upload.start();

}

});

**Resumen**

* **HTTP(S)** es generalmente adecuado y fácil de implementar para la mayoría de las aplicaciones.
* **WebSockets**, **gRPC**, **FTP/SFTP**, **Amazon S3 Presigned URLs**, y **TUS** son alternativas para casos específicos que requieren mayor eficiencia, manejo de interrupciones, o integración con servicios en la nube.
* Para subir múltiples archivos, **TUS** es una opción robusta que maneja cargas interrumpidas y reanudables eficientemente.

### Protocolo TUS

**TUS** (Resumable Upload Protocol) es un protocolo de red diseñado específicamente para facilitar la carga de archivos de manera confiable, especialmente en entornos donde las interrupciones son comunes. A continuación, se presenta una explicación breve:

#### Características Principales:

1. **Resumable Uploads**: Permite que las cargas de archivos se puedan pausar y reanudar, evitando que las interrupciones de la red obliguen a reiniciar la carga desde cero.
2. **Offset-Based**: Utiliza un enfoque basado en offset para manejar las cargas de archivos, lo que significa que los datos del archivo se envían en fragmentos y se puede continuar la carga desde el punto donde se interrumpió.
3. **HTTP Extension**: Extiende el protocolo HTTP estándar, lo que facilita la integración con infraestructuras web existentes.
4. **Client-Server Communication**: Los clientes envían solicitudes POST para iniciar la carga y PATCH para enviar fragmentos de archivos. Los servidores mantienen el estado de la carga y permiten la reanudación mediante la verificación de offsets.

#### Flujo Básico:

1. **Inicio de la Carga**: El cliente envía una solicitud POST al servidor para iniciar una nueva carga.
2. **Subida de Fragmentos**: El cliente envía fragmentos del archivo usando solicitudes PATCH. Cada fragmento incluye un header que indica el offset.
3. **Reanudación**: Si la carga se interrumpe, el cliente puede reanudarla enviando una solicitud HEAD al servidor para obtener el offset actual y continuar desde allí.

### Protocolo gRPC

**gRPC** (gRPC Remote Procedure Calls) es un marco de trabajo de comunicación de alto rendimiento desarrollado por Google, que permite la comunicación entre aplicaciones en diferentes lenguajes y plataformas. Aquí se explica brevemente:

#### Características Principales:

1. **Basado en HTTP/2**: Utiliza HTTP/2 como protocolo de transporte, lo que proporciona múltiples ventajas, como multiplexación de streams, compresión de headers y administración de conexiones más eficientes.
2. **Protocol Buffers**: Utiliza Protocol Buffers (protobuf) como mecanismo de serialización, lo que permite definir interfaces de servicio de manera eficiente y con soporte para múltiples lenguajes de programación.
3. **Comunicación Bidireccional**: Soporta comunicación bidireccional en tiempo real, permitiendo tanto solicitudes síncronas como asíncronas.
4. **Definición de Servicios**: Los servicios gRPC se definen en archivos .proto, donde se especifican los métodos de servicio y los tipos de mensajes.

#### Flujo Básico:

1. **Definición del Servicio**: Se define un servicio en un archivo .proto con los métodos RPC (Remote Procedure Call) y los tipos de mensajes.
2. **Generación de Código**: Se genera código cliente y servidor a partir del archivo .proto en los lenguajes de programación deseados.
3. **Implementación del Servidor**: Se implementa el servidor gRPC, que maneja las solicitudes RPC definidas en el archivo .proto.
4. **Comunicación del Cliente**: Los clientes se comunican con el servidor utilizando los métodos RPC generados, enviando y recibiendo mensajes protobuf.

### Resumen de Comparación:

* **TUS**:
  + Diseñado específicamente para cargas de archivos reanudables.
  + Maneja interrupciones en la red de manera eficiente.
  + Extiende el protocolo HTTP estándar.
* **gRPC**:
  + Orientado a la comunicación de alto rendimiento entre aplicaciones.
  + Basado en HTTP/2 y utiliza Protocol Buffers para la serialización de datos.
  + Soporta comunicación bidireccional en tiempo real.

Ambos protocolos tienen sus casos de uso específicos y pueden ser elegidos según los requisitos de la aplicación, ya sea la necesidad de cargas de archivos reanudables (TUS) o la comunicación eficiente entre servicios en diferentes plataformas y lenguajes (gRPC).