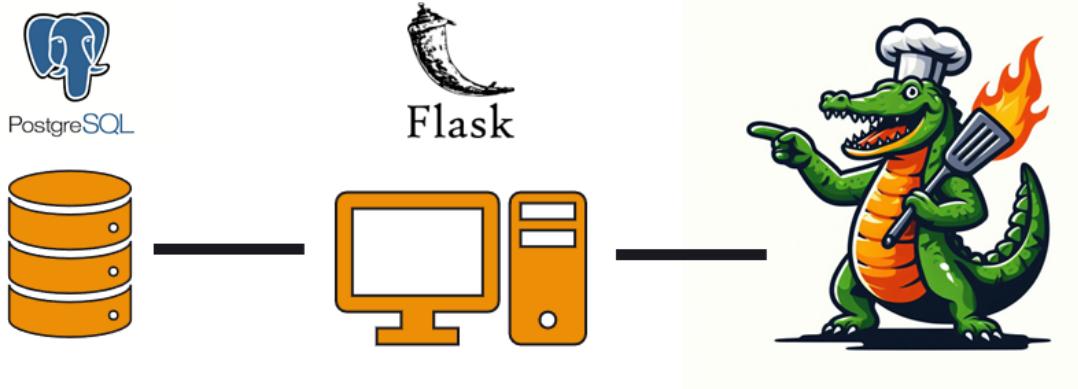


Tasty Bite - Manual de Usuario

TASTY BITE



Administración y Diseño de Base de Datos

Facundo José García Gallo

alu0101406329

Daniel Felipe Gómez Aristizabal

alu0101438139

Índice

1. Acciones sobre la base de datos

RECETA	POST	GET	GET	PUT	DELETE
USUARIO	POST	GET	GET	PUT	DELETE
INGREDIENTE	POST	GET	GET		
CATEGORÍA		GET	GET		
PATROCINADOR		GET	GET		
COMENTARIO	POST	GET		PUT	DELETE
PUNTUACIÓN	POST	GET		PUT	DELETE
RANKING			GET		
AMIGO	POST	GET			DELETE

2. Descripción de la base de datos

RECETA
USUARIO
INGREDIENTE
CATEGORÍA
PATROCINADOR
INTERACCIÓN
COMENTARIO
PUNTUACIÓN
RANKING
AMIGO
USUARIO_RECETA
RECETA_INGREDIENTE

3. APIs

4. Ejemplos de consultas

Introducción

El presente manual de usuario pretende ilustrar todas las posibles acciones que puede hacer un usuario sobre la API REST creada. Hay que aclarar que no todos los verbos HTTP están implementados para todas las entidades de la base de datos ya que se tomaron una serie de decisiones sobre la API, decisiones de diseño que se encuentran explicadas en el [apartado tres de este manual de usuario](#).

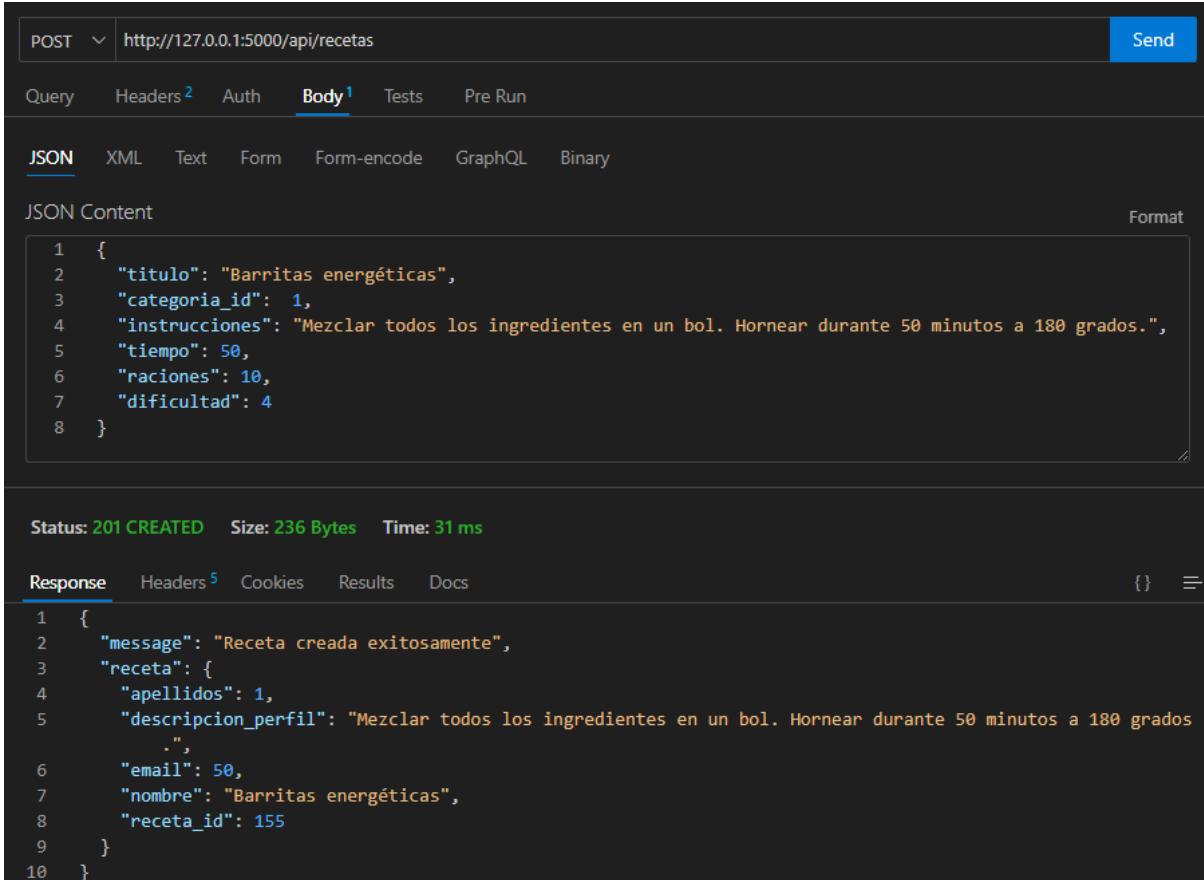
El manual de usuario se divide en varios apartados, concretamente en cuatro: el primero de ellos se encarga de aportar un ejemplo de utilización para cada acción que se puede realizar sobre la API, el segundo apartado describe las diferentes relaciones de la base de datos, el tercero describe qué parámetros se deben pasar para cada una de las acciones que mencionamos en el primer apartado y el último apartado da ejemplos sobre consultas que se pueden hacer sobre la base de datos.

Acciones sobre la base de datos

Receta

POST

URL : <http://127.0.0.1:5000/api/recetas>



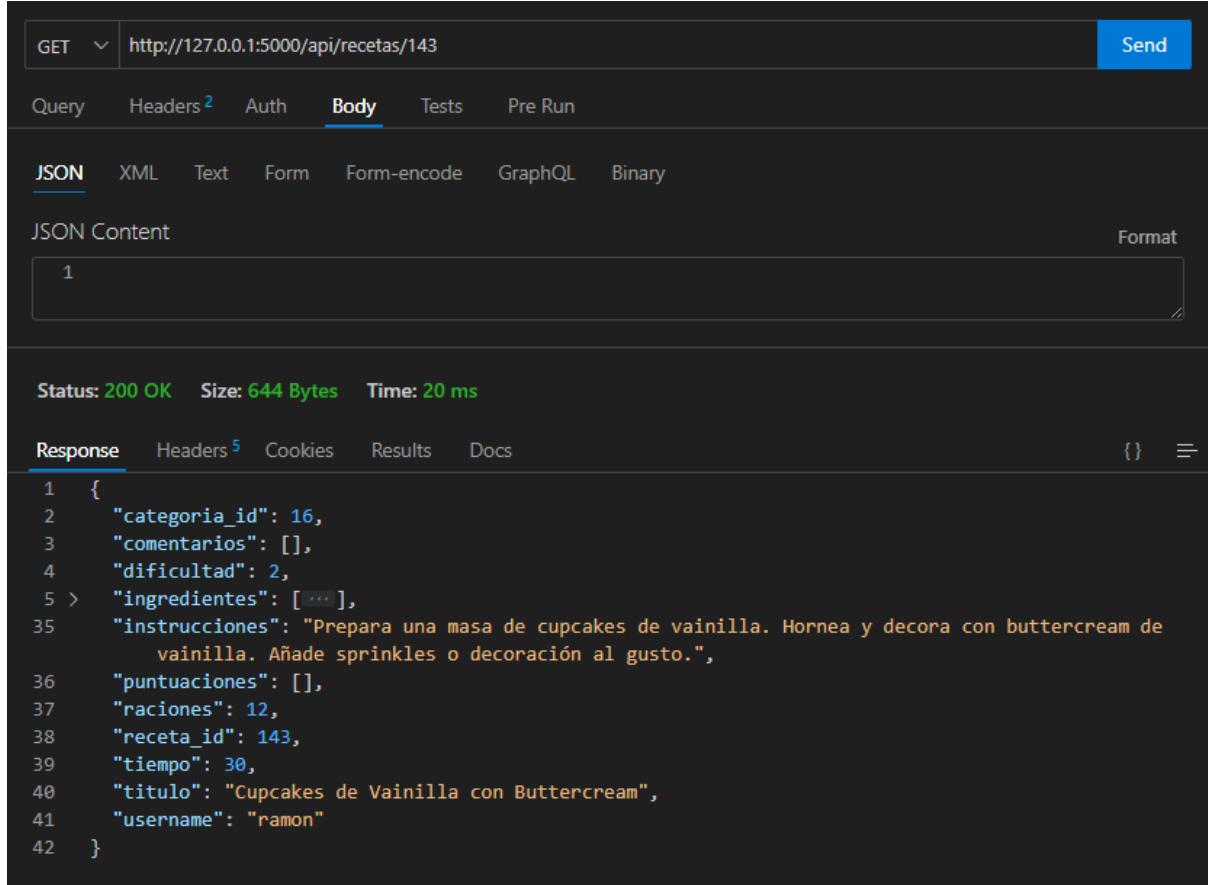
The screenshot shows a Postman request for creating a new recipe. The method is POST, the URL is <http://127.0.0.1:5000/api/recetas>, and the body is set to JSON. The JSON content is:

```
1 {  
2     "titulo": "Barritas energéticas",  
3     "categoria_id": 1,  
4     "instrucciones": "Mezclar todos los ingredientes en un bol. Hornear durante 50 minutos a 180 grados.",  
5     "tiempo": 50,  
6     "raciones": 10,  
7     "dificultad": 4  
8 }
```

The response status is 201 CREATED, size is 236 Bytes, and time is 31 ms. The response body is:

```
1 {  
2     "message": "Receta creada exitosamente",  
3     "receta": {  
4         "apellidos": 1,  
5         "descripcion_perfil": "Mezclar todos los ingredientes en un bol. Hornear durante 50 minutos a 180 grados  
.",  
6         "email": 50,  
7         "nombre": "Barritas energéticas",  
8         "receta_id": 155  
9     }  
10 }
```

GET

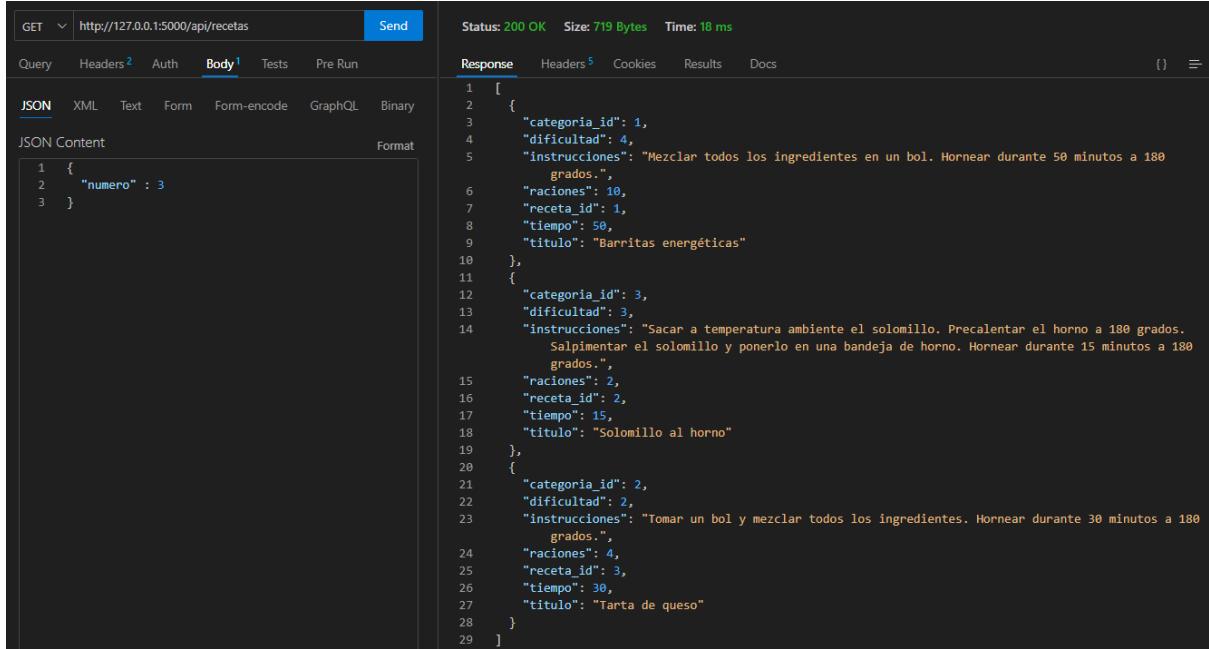
URL : <http://127.0.0.1:5000/api/recetas/143>

The screenshot shows a Postman request for a GET API endpoint. The URL is <http://127.0.0.1:5000/api/recetas/143>. The request body is empty. The response status is 200 OK, size is 644 Bytes, and time is 20 ms. The response JSON is:

```
1 {  
2     "categoria_id": 16,  
3     "comentarios": [],  
4     "dificultad": 2,  
5     "ingredientes": [...],  
35    "instrucciones": "Prepara una masa de cupcakes de vainilla. Hornea y decora con buttercream de vainilla. Añade sprinkles o decoración al gusto.",  
36    "puntuaciones": [],  
37    "raciones": 12,  
38    "receta_id": 143,  
39    "tiempo": 30,  
40    "titulo": "Cupcakes de Vainilla con Buttercream",  
41    "username": "ramon"  
42 }
```

GET

URL : <http://127.0.0.1:5000/api/recetas>

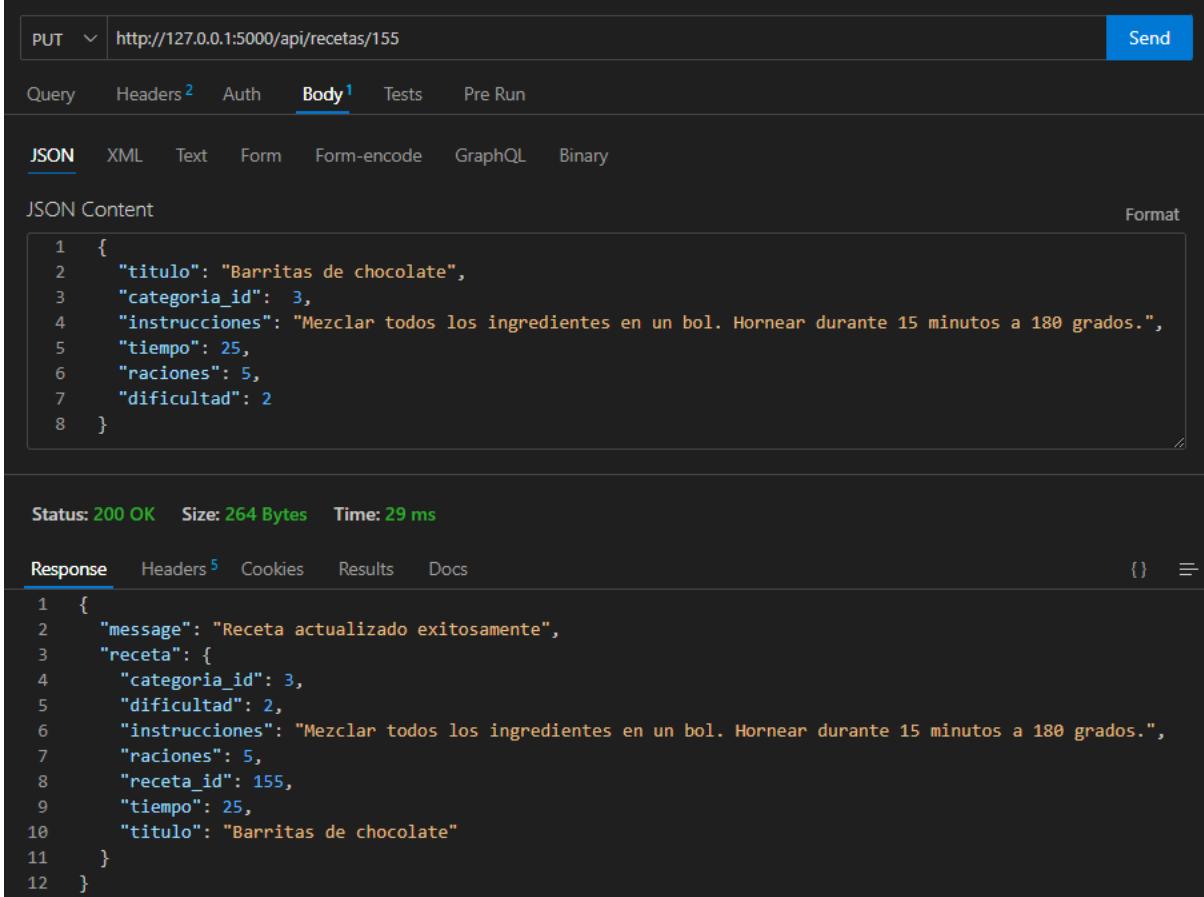


Status: 200 OK Size: 719 Bytes Time: 18 ms

Response	Headers	Cookies	Results	Docs
<pre> 1 [2 { 3 "categoria_id": 1, 4 "dificultad": 4, 5 "instrucciones": "Mezclar todos los ingredientes en un bol. Hornear durante 50 minutos a 180 6 grados.", 7 "raciones": 10, 8 "receta_id": 1, 9 "tiempo": 50, 10 "titulo": "Barritas energéticas" 11 }, 12 { 13 "categoria_id": 3, 14 "dificultad": 3, 15 "instrucciones": "Sacar a temperatura ambiente el solomillo. Precalentar el horno a 180 grados. 16 Salpicar el solomillo y ponerlo en una bandeja de horno. Hornear durante 15 minutos a 180 17 grados.", 18 "raciones": 2, 19 "receta_id": 2, 20 "tiempo": 15, 21 "titulo": "Solomillo al horno" 22 }, 23 { 24 "categoria_id": 2, 25 "dificultad": 2, 26 "instrucciones": "Tomar un bol y mezclar todos los ingredientes. Hornear durante 30 minutos a 180 27 grados.", 28 "raciones": 4, 29 "receta_id": 3, 29 "tiempo": 30, 29 "titulo": "Tarta de queso" 29 }] </pre>				

PUT

URL : http://127.0.0.1:5000/api/recetas/155



The screenshot shows a POSTMAN interface with a PUT request to `http://127.0.0.1:5000/api/recetas/155`. The request body is a JSON object:

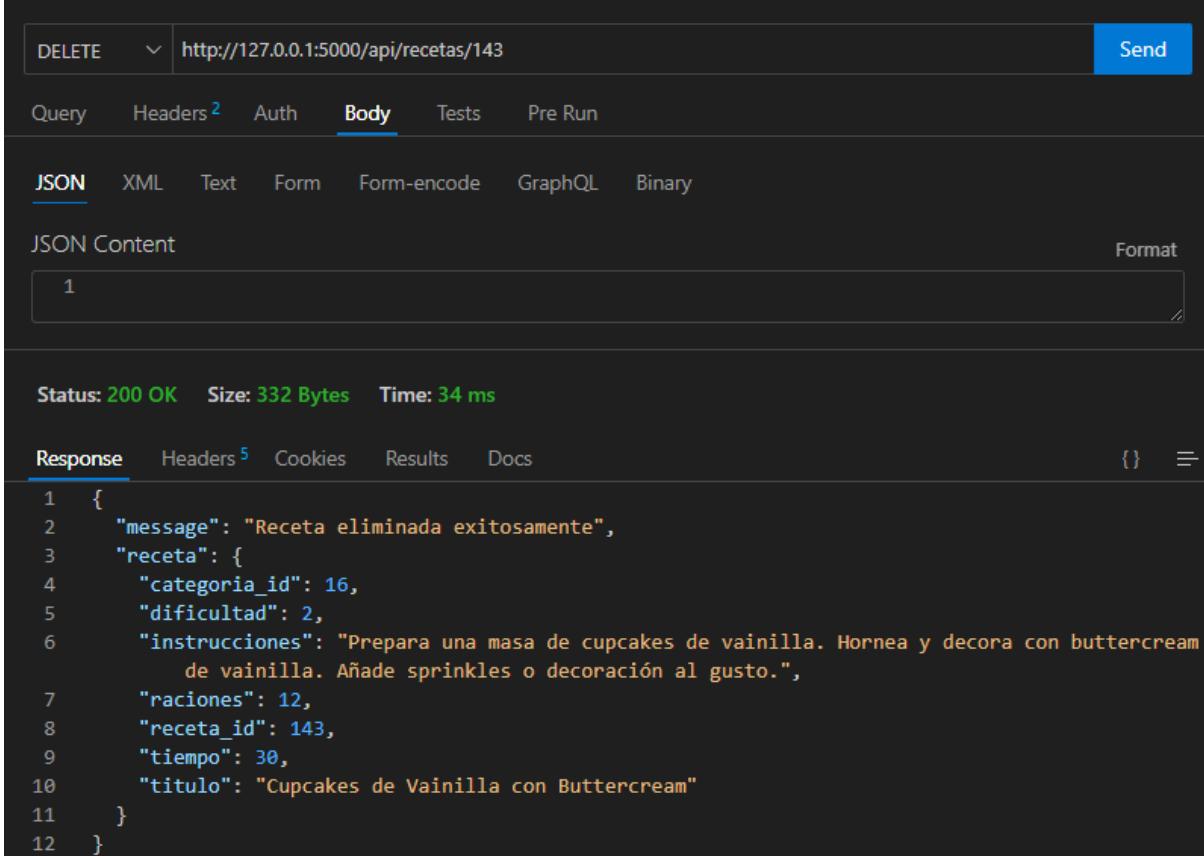
```
1 {  
2   "titulo": "Barritas de chocolate",  
3   "categoria_id": 3,  
4   "instrucciones": "Mezclar todos los ingredientes en un bol. Hornear durante 15 minutos a 180 grados.",  
5   "tiempo": 25,  
6   "raciones": 5,  
7   "dificultad": 2  
8 }
```

The response status is 200 OK, size is 264 Bytes, and time is 29 ms. The response body is:

```
1 {  
2   "message": "Receta actualizado exitosamente",  
3   "receta": {  
4     "categoria_id": 3,  
5     "dificultad": 2,  
6     "instrucciones": "Mezclar todos los ingredientes en un bol. Hornear durante 15 minutos a 180 grados.",  
7     "raciones": 5,  
8     "receta_id": 155,  
9     "tiempo": 25,  
10    "titulo": "Barritas de chocolate"  
11  }  
12 }
```

DELETE

URL : <http://127.0.0.1:5000/api/recetas/143>



DELETE <http://127.0.0.1:5000/api/recetas/143> Send

Query Headers ² Auth Body Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

1

Status: 200 OK Size: 332 Bytes Time: 34 ms

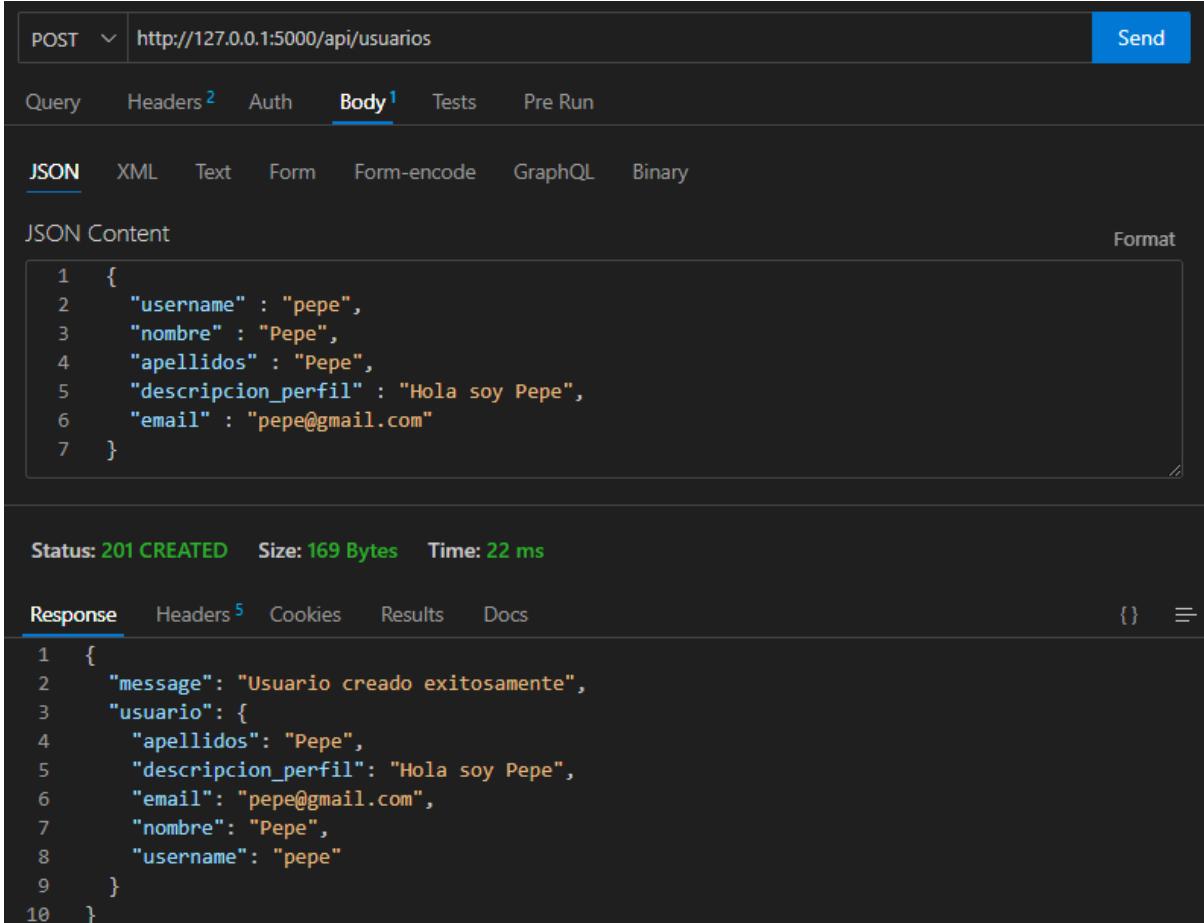
Response Headers ⁵ Cookies Results Docs { } ⚙

```
1 {
2     "message": "Receta eliminada exitosamente",
3     "receta": {
4         "categoria_id": 16,
5         "dificultad": 2,
6         "instrucciones": "Prepara una masa de cupcakes de vainilla. Hornea y decora con buttercream de vainilla. Añade sprinkles o decoración al gusto.",
7         "raciones": 12,
8         "receta_id": 143,
9         "tiempo": 30,
10        "titulo": "Cupcakes de Vainilla con Buttercream"
11    }
12 }
```

Usuario

POST

URL : <http://127.0.0.1:5000/api/usuarios>



The screenshot shows a Postman request and response interface. The request URL is <http://127.0.0.1:5000/api/usuarios>. The request method is POST. The body is in JSON format, containing the following data:

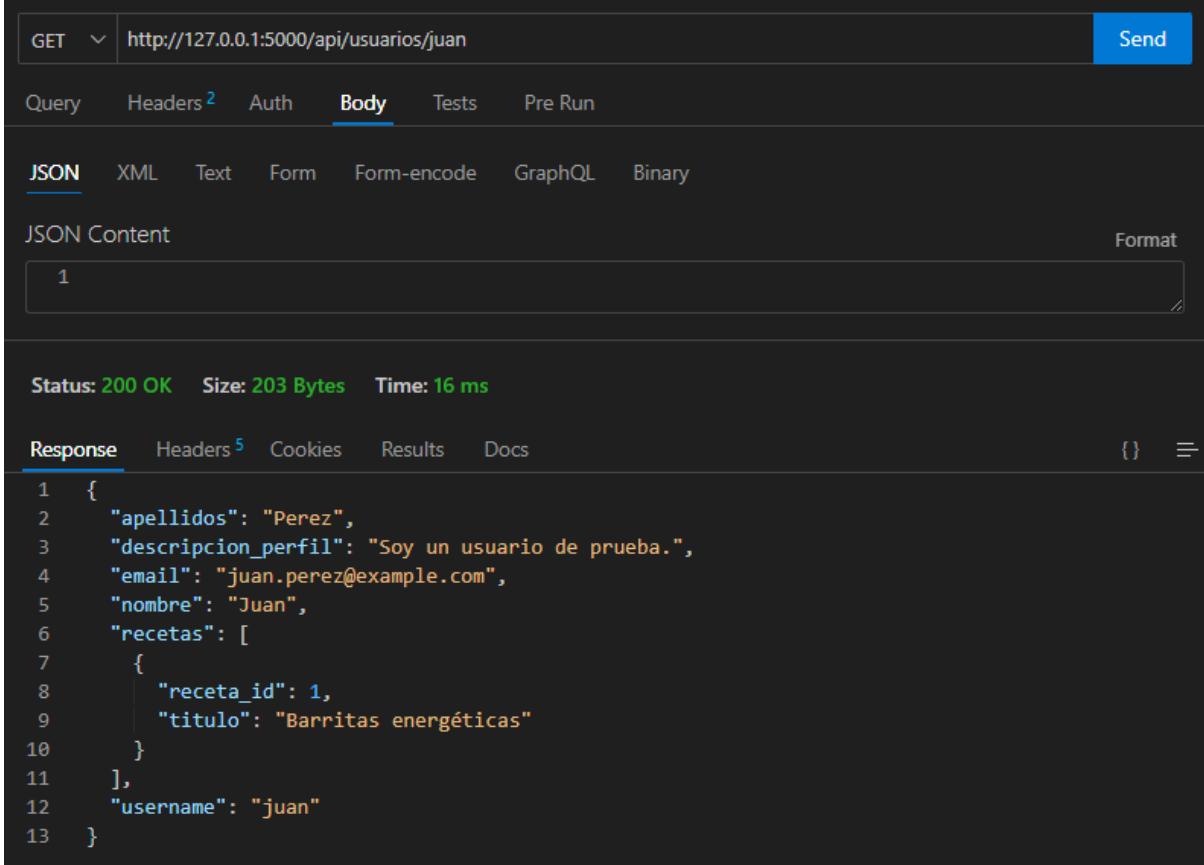
```
1  {
2    "username" : "pepe",
3    "nombre" : "Pepe",
4    "apellidos" : "Pepe",
5    "descripcion_perfil" : "Hola soy Pepe",
6    "email" : "pepe@gmail.com"
7 }
```

The response status is 201 CREATED, size is 169 Bytes, and time is 22 ms. The response body is:

```
1  {
2    "message": "Usuario creado exitosamente",
3    "usuario": {
4      "apellidos": "Pepe",
5      "descripcion_perfil": "Hola soy Pepe",
6      "email": "pepe@gmail.com",
7      "nombre": "Pepe",
8      "username": "pepe"
9    }
10 }
```

GET

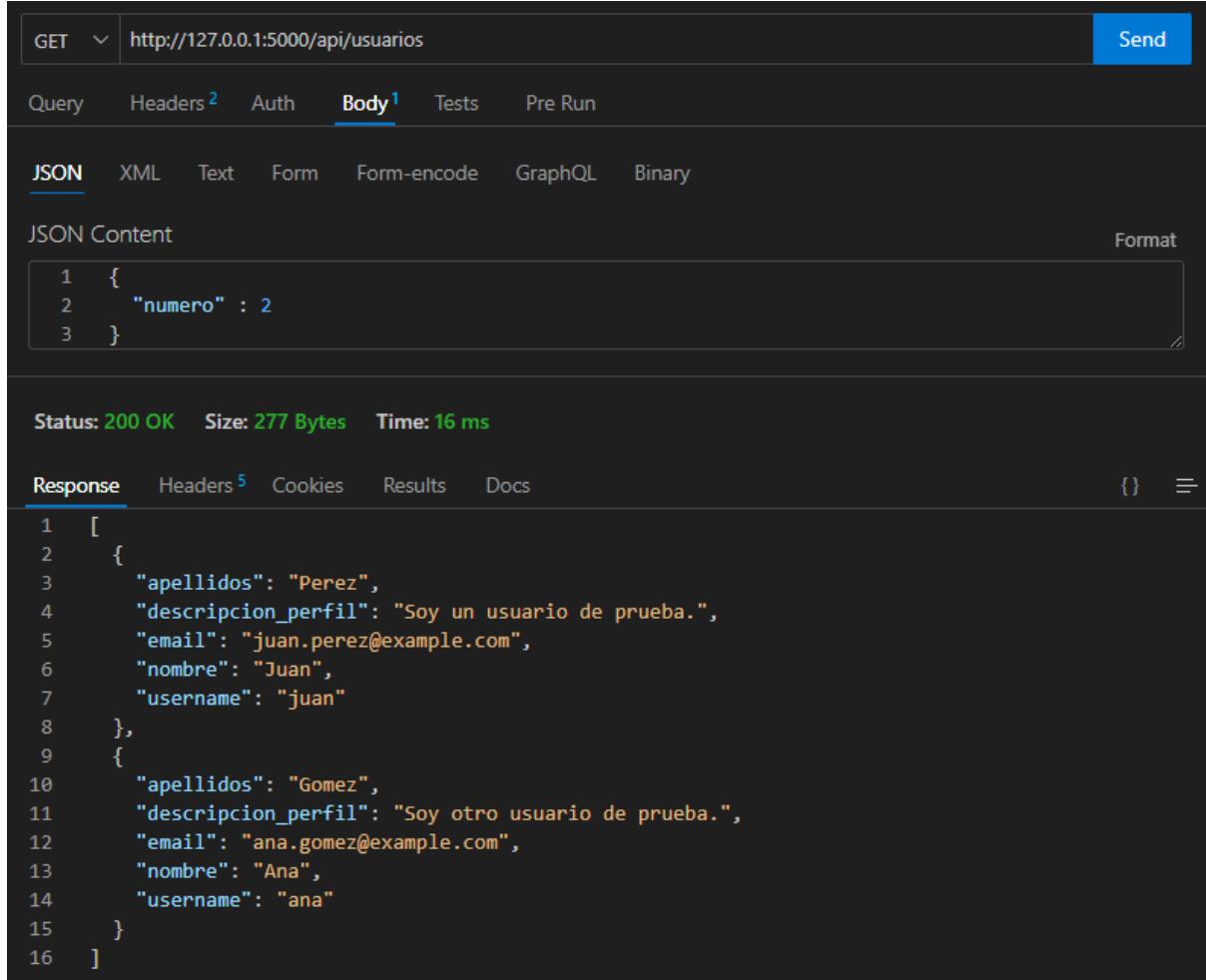
URL : http://127.0.0.1:5000/api/usuarios/juan



The screenshot shows a Postman request for a GET API endpoint. The URL is `http://127.0.0.1:5000/api/usuarios/juan`. The 'Body' tab is selected, showing JSON content with the number '1'. The response status is 200 OK, with 203 bytes and 16 ms time. The response body is a JSON object representing a user profile:

```
1 {  
2     "apellidos": "Perez",  
3     "descripcion_perfil": "Soy un usuario de prueba.",  
4     "email": "juan.perez@example.com",  
5     "nombre": "Juan",  
6     "recetas": [  
7         {  
8             "receta_id": 1,  
9             "titulo": "Barritas energéticas"  
10        }  
11    ],  
12    "username": "juan"  
13 }
```

GET

URL : <http://127.0.0.1:5000/api/usuarios>

GET <http://127.0.0.1:5000/api/usuarios> Send

Query Headers² Auth Body¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {  
2   "numero" : 2  
3 }
```

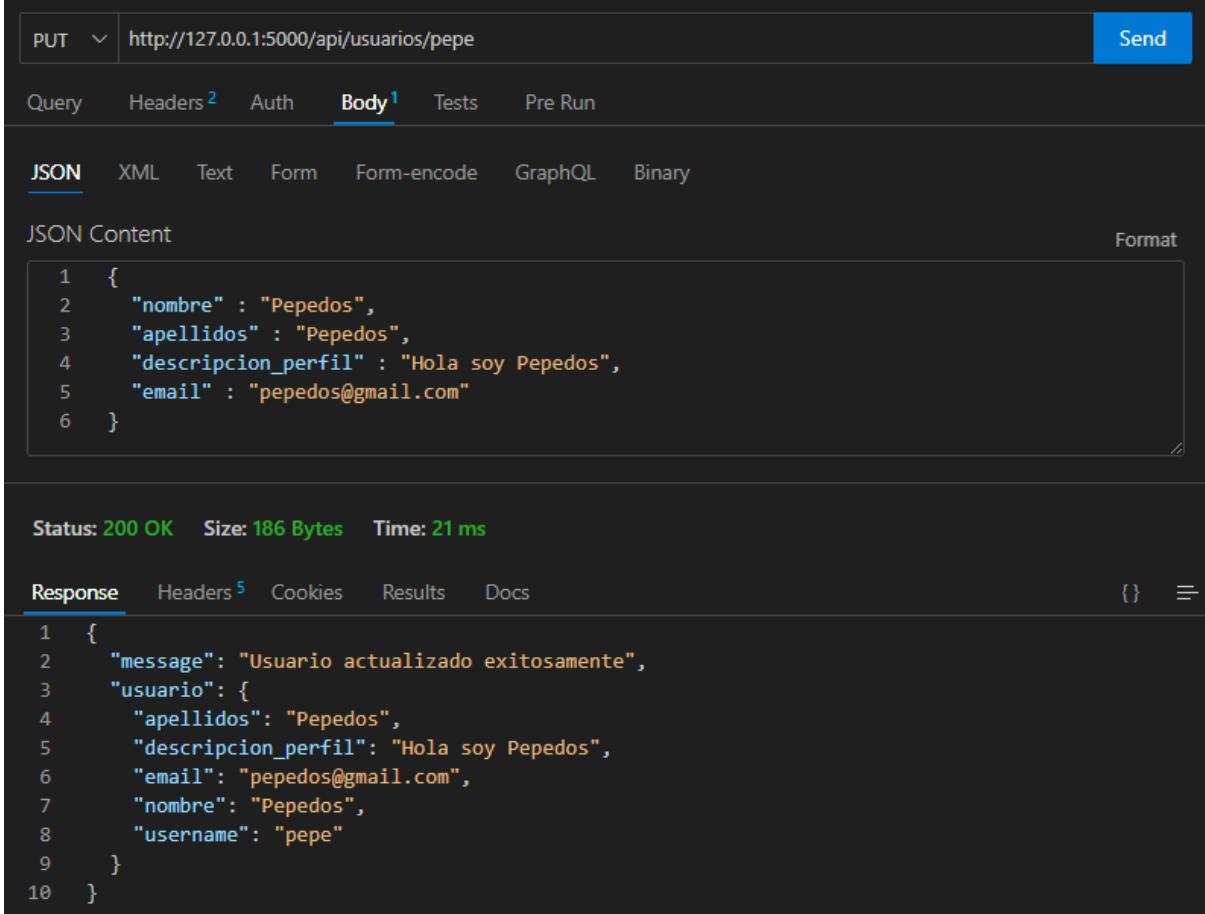
Status: 200 OK Size: 277 Bytes Time: 16 ms

Response Headers⁵ Cookies Results Docs {} ⚡

```
1 [  
2   {  
3     "apellidos": "Perez",  
4     "descripcion_perfil": "Soy un usuario de prueba.",  
5     "email": "juan.perez@example.com",  
6     "nombre": "Juan",  
7     "username": "juan"  
8   },  
9   {  
10    "apellidos": "Gomez",  
11    "descripcion_perfil": "Soy otro usuario de prueba.",  
12    "email": "ana.gomez@example.com",  
13    "nombre": "Ana",  
14    "username": "ana"  
15  }  
16 ]
```

PUT

URL : http://127.0.0.1:5000/api/usuarios/pepe



The screenshot shows the Postman application interface. At the top, there is a header bar with the method "PUT" and the URL "http://127.0.0.1:5000/api/usuarios/pepe". To the right of the URL is a blue "Send" button. Below the header, there are tabs for "Query", "Headers²", "Auth", "Body¹" (which is underlined in blue), "Tests", and "Pre Run". Under the "Body" tab, there are tabs for "JSON" (which is underlined in blue), "XML", "Text", "Form", "Form-encode", "GraphQL", and "Binary". The "JSON Content" section contains the following JSON code:

```
1 {
2   "nombre" : "Pepedos",
3   "apellidos" : "Pepedos",
4   "descripcion_perfil" : "Hola soy Pepedos",
5   "email" : "pepedos@gmail.com"
6 }
```

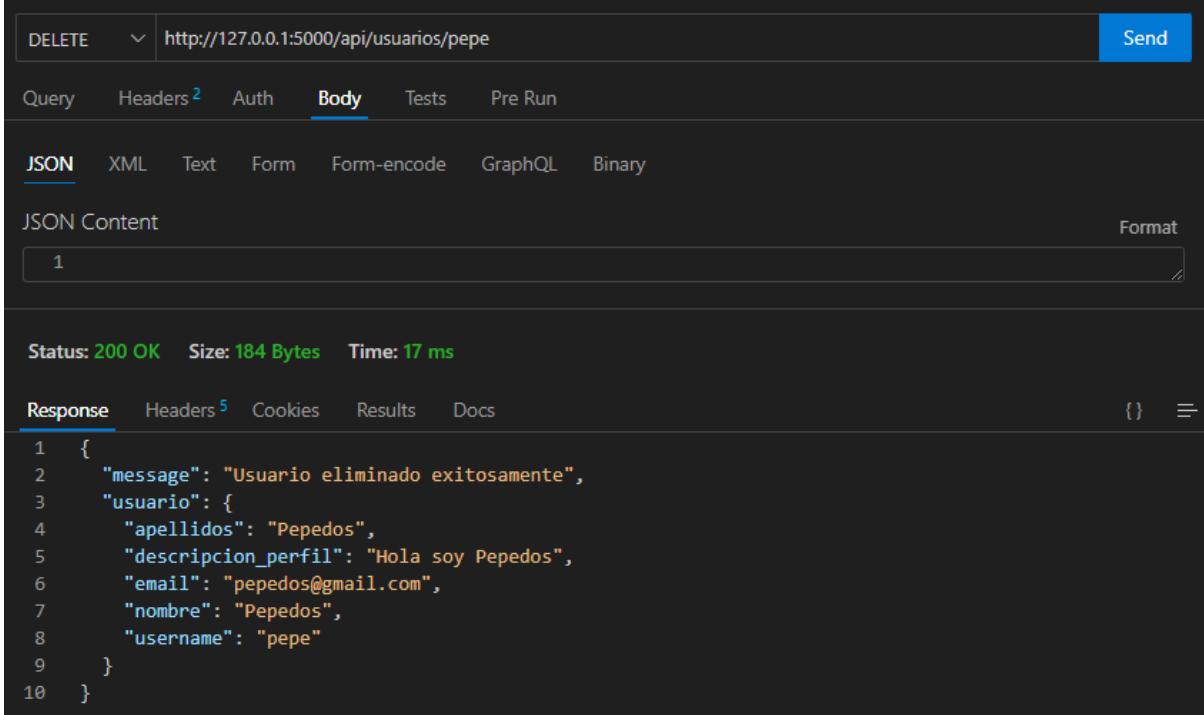
On the right side of the JSON content area, there is a "Format" button. Below the JSON content, the status bar shows "Status: 200 OK", "Size: 186 Bytes", and "Time: 21 ms".

At the bottom of the interface, there is a "Response" tab followed by "Headers⁵", "Cookies", "Results", and "Docs" tabs. The "Response" tab is currently active. The response body is displayed as:

```
1 {
2   "message": "Usuario actualizado exitosamente",
3   "usuario": {
4     "apellidos": "Pepedos",
5     "descripcion_perfil": "Hola soy Pepedos",
6     "email": "pepedos@gmail.com",
7     "nombre": "Pepedos",
8     "username": "pepe"
9   }
10 }
```

DELETE

URL : <http://127.0.0.1:5000/api/usuarios/juan>



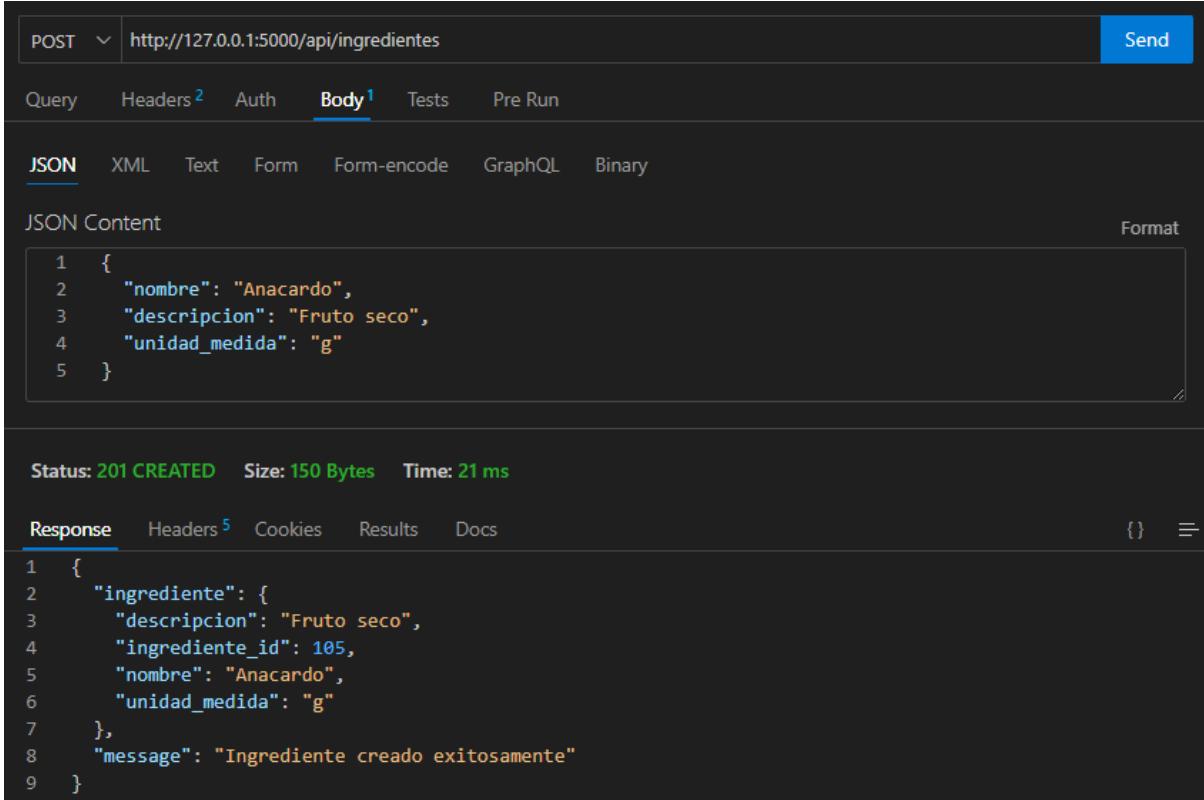
The screenshot shows the Postman application interface. At the top, there is a header bar with the method "DELETE" and the URL "http://127.0.0.1:5000/api/usuarios/pepe". To the right of the URL is a blue "Send" button. Below the header, there are tabs for "Query", "Headers 2", "Auth", "Body", "Tests", and "Pre Run". The "Body" tab is currently selected and has a sub-tab "JSON" which is also selected. In the "JSON Content" section, there is a single digit "1" in a text input field. To the right of this input field is a "Format" button. Below the body section, the status of the request is displayed as "Status: 200 OK", "Size: 184 Bytes", and "Time: 17 ms". The "Response" tab is selected, showing the JSON response from the server. The response is a multi-line JSON object:

```
1  {
2      "message": "Usuario eliminado exitosamente",
3      "usuario": {
4          "apellidos": "Pepedos",
5          "descripcion_perfil": "Hola soy Pepedos",
6          "email": "pepedos@gmail.com",
7          "nombre": "Pepedos",
8          "username": "pepe"
9      }
10 }
```

Ingrediente

POST

URL : <http://127.0.0.1:5000/api/ingredientes>



POST <http://127.0.0.1:5000/api/ingredientes> Send

Query Headers² Auth Body¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {  
2     "nombre": "Anacardo",  
3     "descripcion": "Fruto seco",  
4     "unidad_medida": "g"  
5 }
```

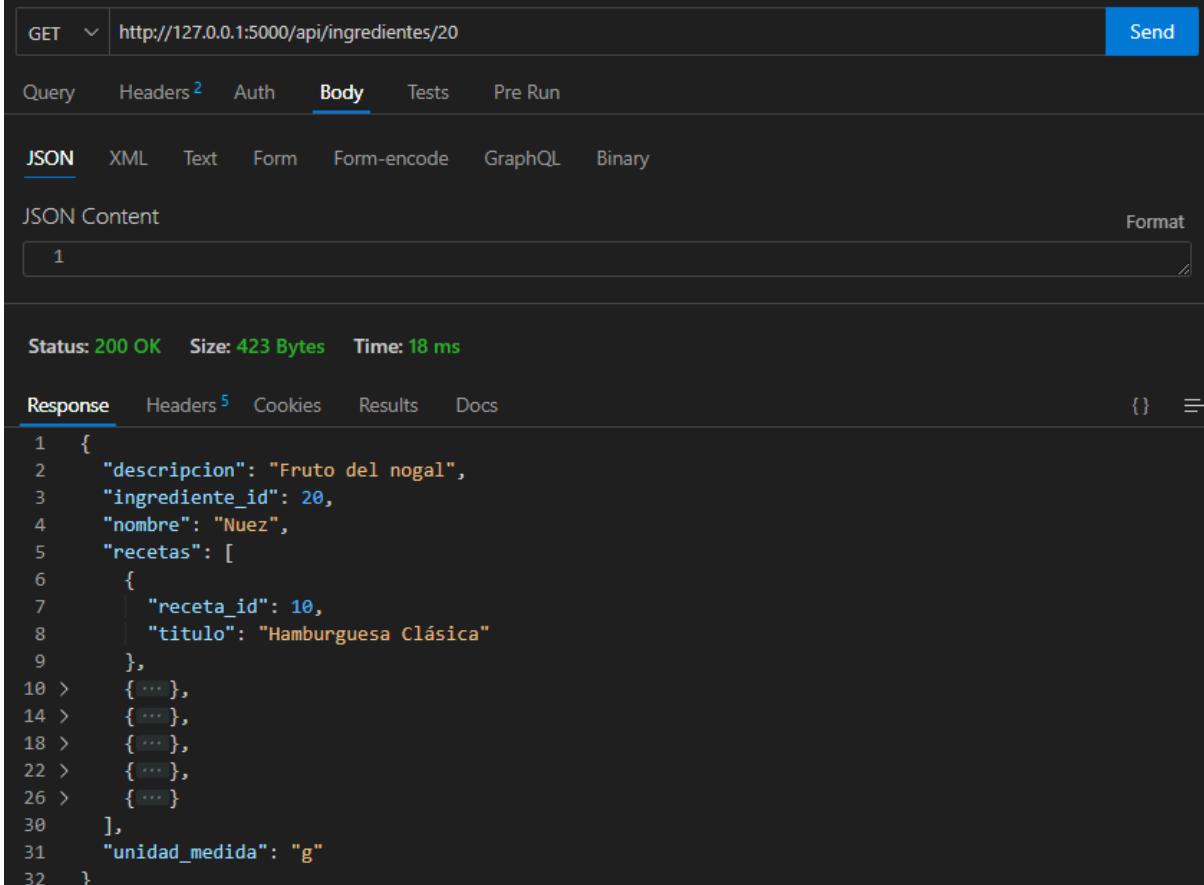
Status: 201 CREATED Size: 150 Bytes Time: 21 ms

Response Headers⁵ Cookies Results Docs {} ⚡

```
1 {  
2     "ingrediente": {  
3         "descripcion": "Fruto seco",  
4         "ingrediente_id": 105,  
5         "nombre": "Anacardo",  
6         "unidad_medida": "g"  
7     },  
8     "message": "Ingrediente creado exitosamente"  
9 }
```

GET

URL : http://127.0.0.1:5000/api/ingredientes/20

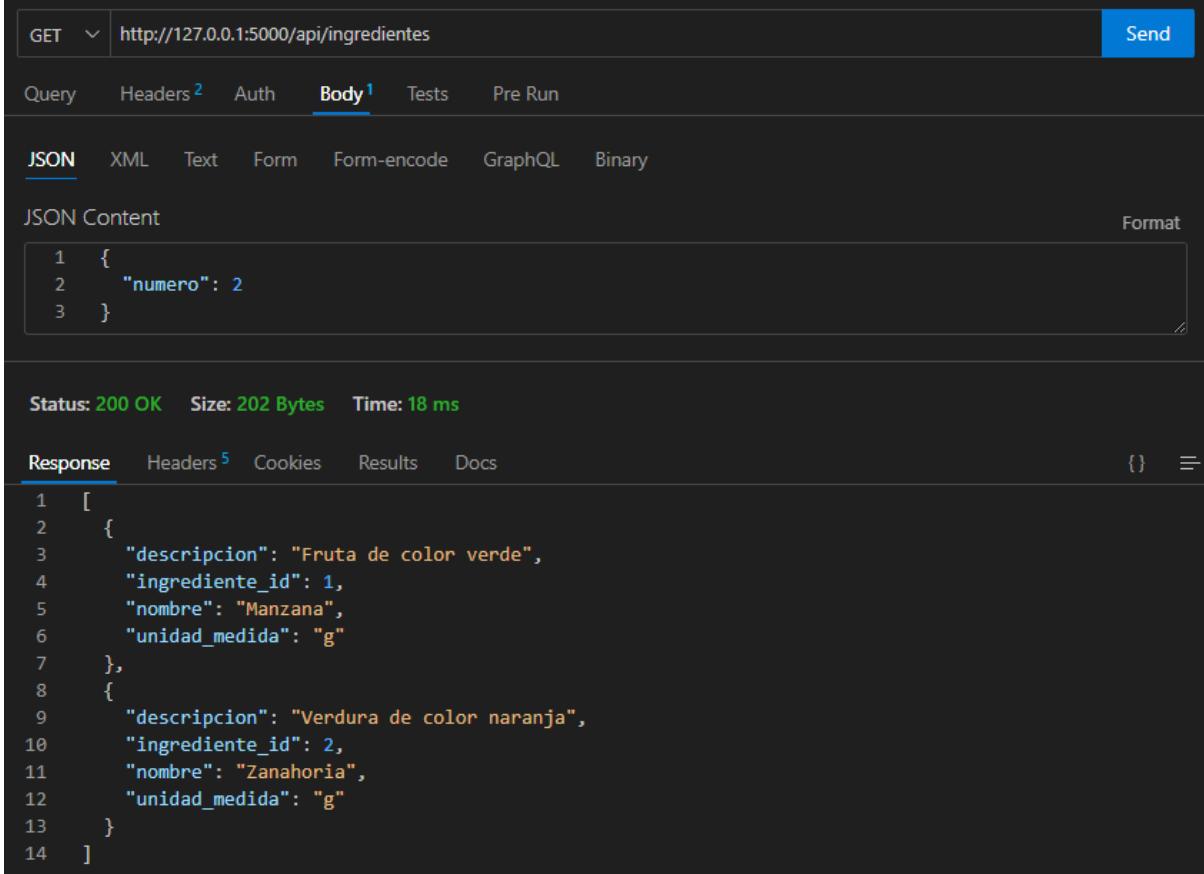


The screenshot shows a Postman request for the URL `http://127.0.0.1:5000/api/ingredientes/20`. The method is set to GET. The Body tab is selected, showing JSON content with the value `1`. The response status is 200 OK, size is 423 Bytes, and time is 18 ms. The response body is a JSON object:

```
1  {
2      "descripcion": "Fruto del nogal",
3      "ingrediente_id": 20,
4      "nombre": "Nuez",
5      "recetas": [
6          {
7              "receta_id": 10,
8              "titulo": "Hamburguesa Clásica"
9          },
10         { ... },
11         { ... },
12         { ... },
13         { ... }
14     ],
15     "unidad_medida": "g"
16 }
```

GET

URL : http://127.0.0.1:5000/api/ingredientes



The screenshot shows a Postman request and response interface. The request URL is `http://127.0.0.1:5000/api/ingredientes`. The Body tab is selected, showing a JSON payload:

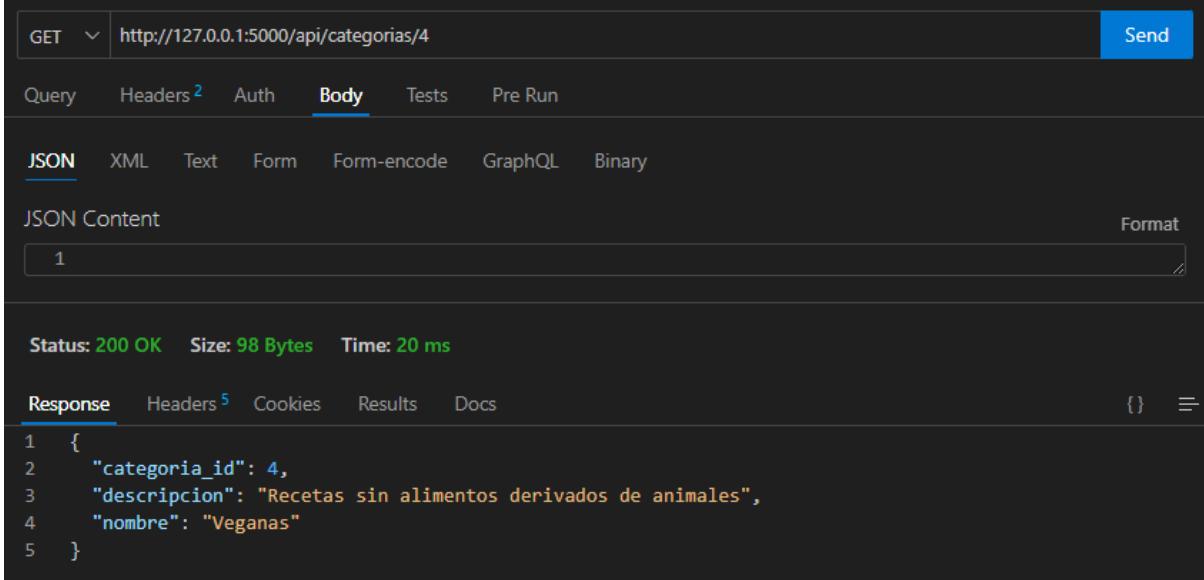
```
1  {
2      "numero": 2
3  }
```

The response status is `200 OK`, size is `202 Bytes`, and time is `18 ms`. The response body is a JSON array containing two objects:1 [
2 {
3 "descripcion": "Fruta de color verde",
4 "ingrediente_id": 1,
5 "nombre": "Manzana",
6 "unidad_medida": "g"
7 },
8 {
9 "descripcion": "Verdura de color naranja",
10 "ingrediente_id": 2,
11 "nombre": "Zanahoria",
12 "unidad_medida": "g"
13 }
14]

Categoría

GET

URL : <http://127.0.0.1:5000/api/categorias/4>



GET <http://127.0.0.1:5000/api/categorias/4> Send

Query Headers² Auth Body Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

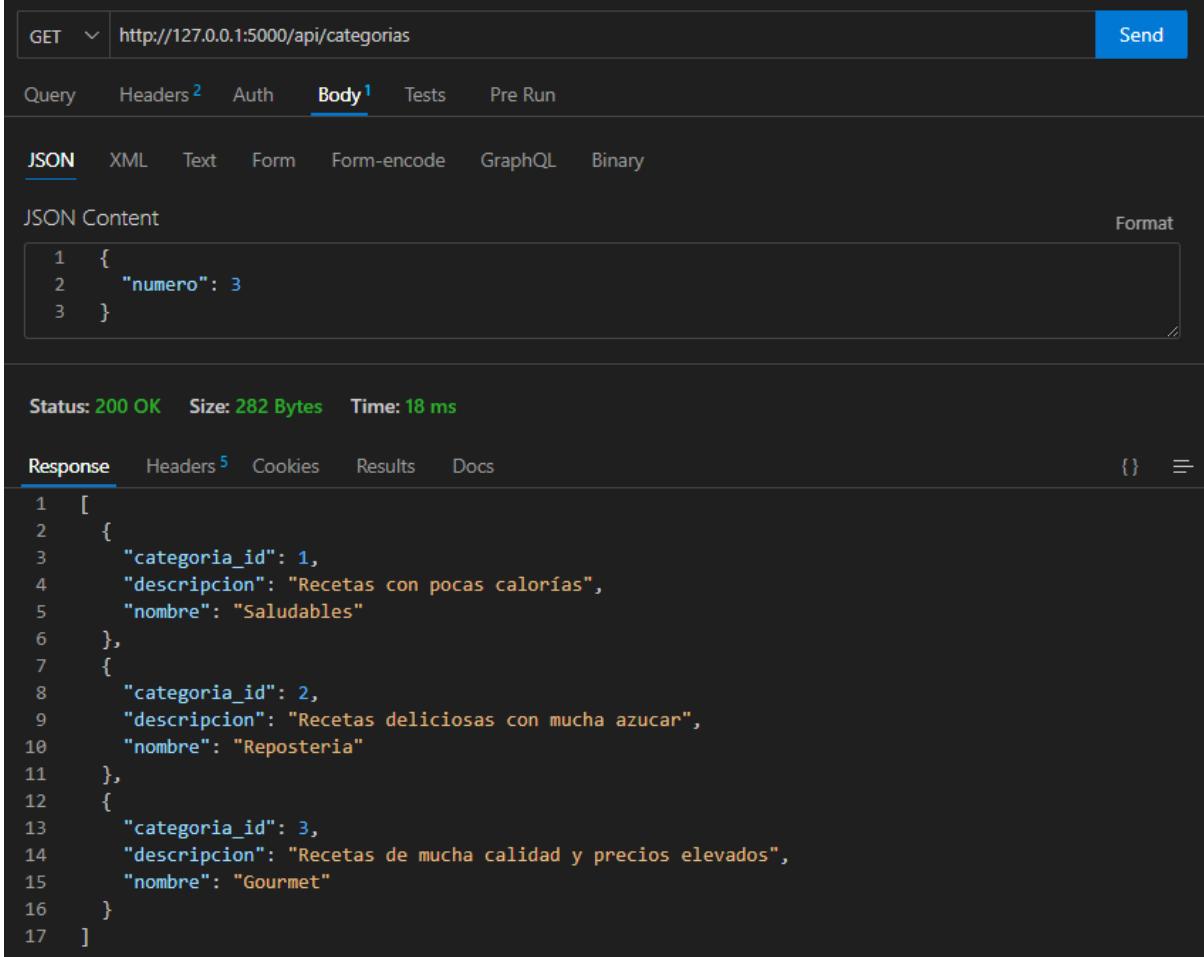
1

Status: 200 OK Size: 98 Bytes Time: 20 ms

Response Headers⁵ Cookies Results Docs {} ⚡

```
1 {  
2     "categoria_id": 4,  
3     "descripcion": "Recetas sin alimentos derivados de animales",  
4     "nombre": "Veganas"  
5 }
```

GET

URL : <http://127.0.0.1:5000/api/categorias>

GET <http://127.0.0.1:5000/api/categorias> Send

Query Headers ² Auth Body ¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1  {
2    "numero": 3
3 }
```

Status: 200 OK Size: 282 Bytes Time: 18 ms

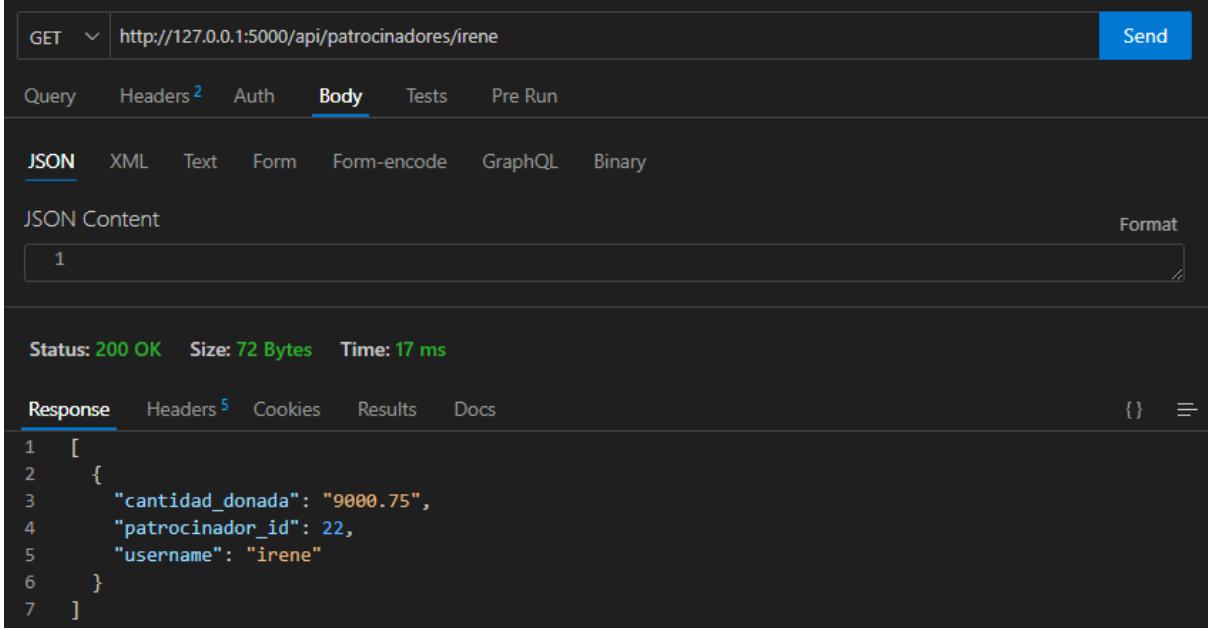
Response Headers ⁵ Cookies Results Docs {} ⚡

```
1 [
2   {
3     "categoria_id": 1,
4     "descripcion": "Recetas con pocas calorías",
5     "nombre": "Saludables"
6   },
7   {
8     "categoria_id": 2,
9     "descripcion": "Recetas deliciosas con mucha azúcar",
10    "nombre": "Repostería"
11  },
12  {
13    "categoria_id": 3,
14    "descripcion": "Recetas de mucha calidad y precios elevados",
15    "nombre": "Gourmet"
16  }
17 ]
```

Patrocinador

GET

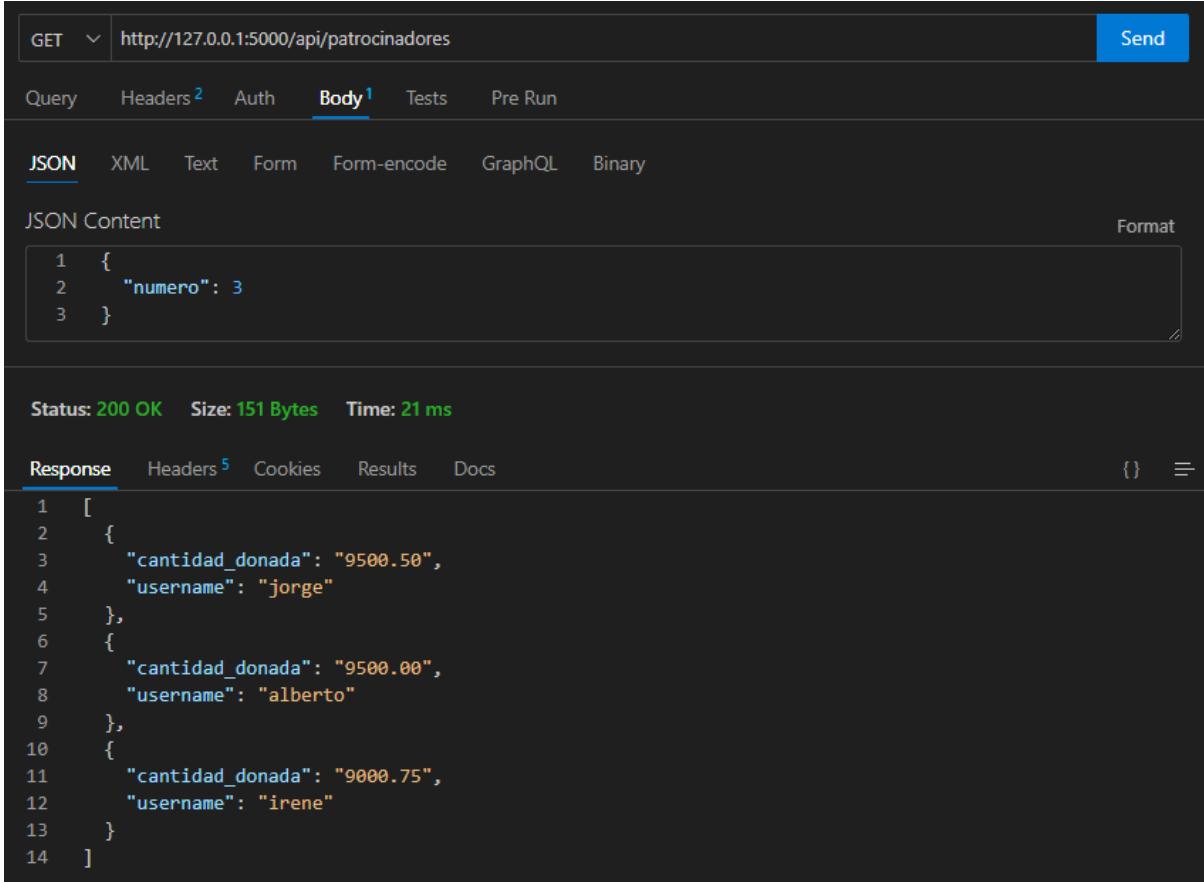
URL : <http://127.0.0.1:5000/api/patrocinadores/irene>



The screenshot shows a Postman request for the URL `http://127.0.0.1:5000/api/patrocinadores/irene`. The method is set to GET. The Body tab is selected, showing the JSON content: `1`. The response status is 200 OK, size is 72 Bytes, and time is 17 ms. The response body is a JSON array containing one object:

```
1 [  
2 {  
3     "cantidad_donada": "9000.75",  
4     "patrocinador_id": 22,  
5     "username": "irene"  
6 }  
7 ]
```

GET

URL : <http://127.0.0.1:5000/api/patrocinadores>

GET <http://127.0.0.1:5000/api/patrocinadores> Send

Query Headers² Auth Body¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {  
2   "numero": 3  
3 }
```

Status: 200 OK Size: 151 Bytes Time: 21 ms

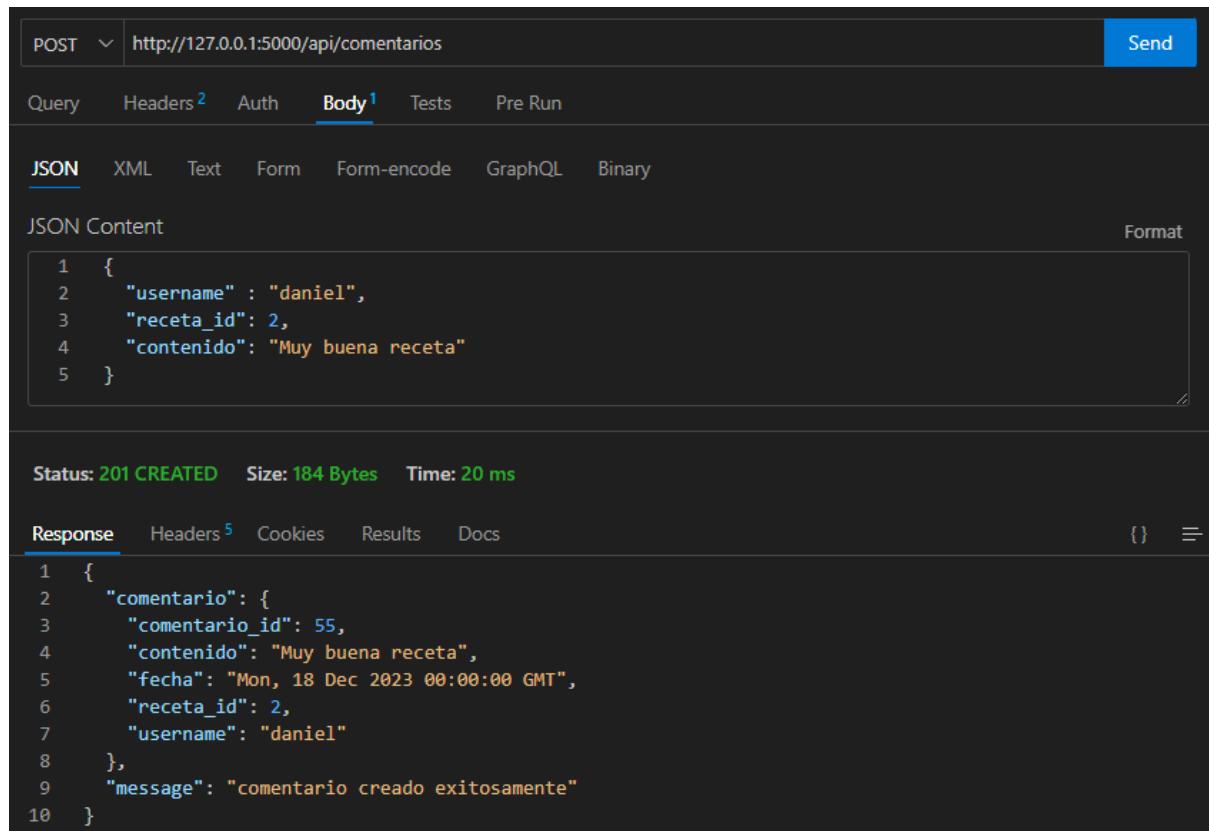
Response Headers⁵ Cookies Results Docs {} ⚡

```
1 [  
2   {  
3     "cantidad_donada": "9500.50",  
4     "username": "jorge"  
5   },  
6   {  
7     "cantidad_donada": "9500.00",  
8     "username": "alberto"  
9   },  
10  {  
11    "cantidad_donada": "9000.75",  
12    "username": "irene"  
13  }  
14 ]
```

Comentario

POST

URL : <http://127.0.0.1:5000/api/comentarios>



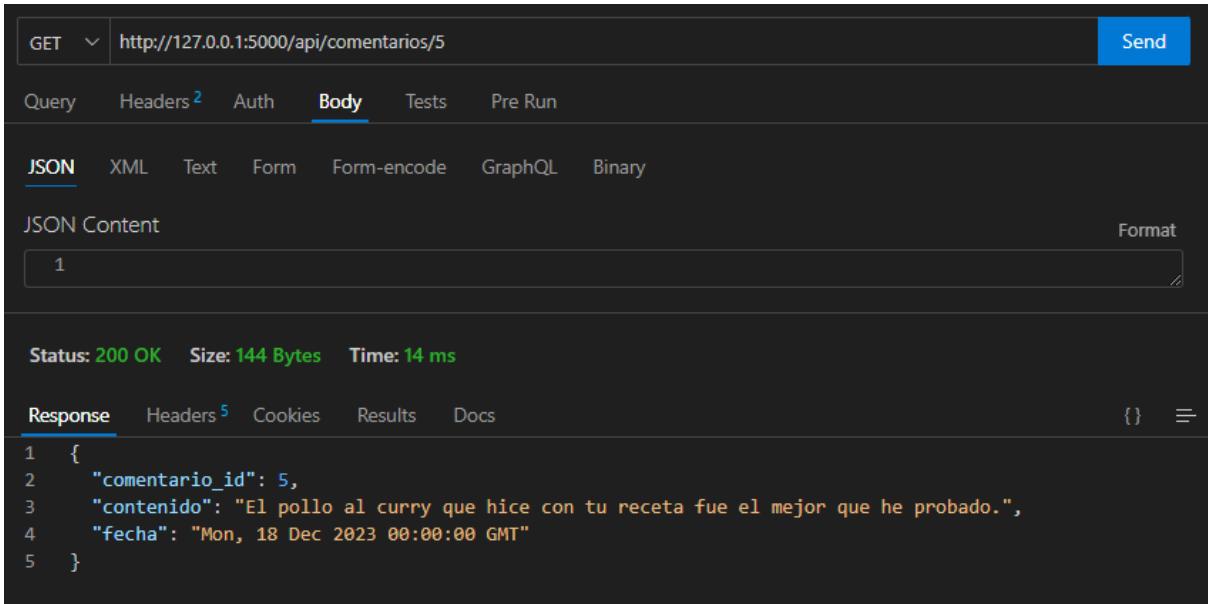
The screenshot shows a Postman request for a POST method at the URL <http://127.0.0.1:5000/api/comentarios>. The request body is in JSON format:

```
1 {  
2   "username" : "daniel",  
3   "receta_id": 2,  
4   "contenido": "Muy buena receta"  
5 }
```

The response status is 201 CREATED, size is 184 Bytes, and time is 20 ms. The response body is:

```
1 {  
2   "comentario": {  
3     "comentario_id": 55,  
4     "contenido": "Muy buena receta",  
5     "fecha": "Mon, 18 Dec 2023 00:00:00 GMT",  
6     "receta_id": 2,  
7     "username": "daniel"  
8   },  
9   "message": "comentario creado exitosamente"  
10 }
```

GET

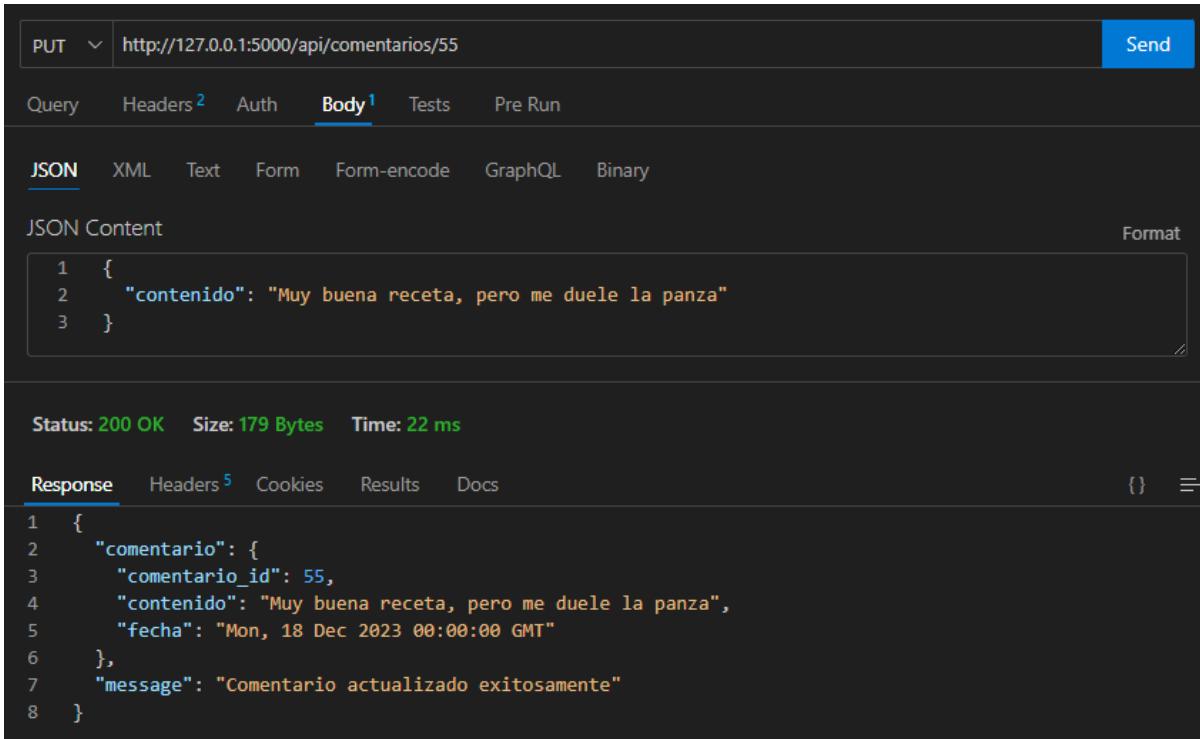
URL : <http://127.0.0.1:5000/api/comentarios/5>

The screenshot shows a Postman request for a GET operation on the URL `http://127.0.0.1:5000/api/comentarios/5`. The request body is set to JSON with the value `1`. The response status is **200 OK**, size is **144 Bytes**, and time is **14 ms**. The response body is a JSON object:

```
1 {  
2   "comentario_id": 5,  
3   "contenido": "El pollo al curry que hice con tu receta fue el mejor que he probado.",  
4   "fecha": "Mon, 18 Dec 2023 00:00:00 GMT"  
5 }
```

PUT

URL : http://127.0.0.1:5000/api/comentarios/55



The screenshot shows a POSTMAN interface with a PUT request to `http://127.0.0.1:5000/api/comentarios/55`. The Body tab is selected, showing a JSON content:

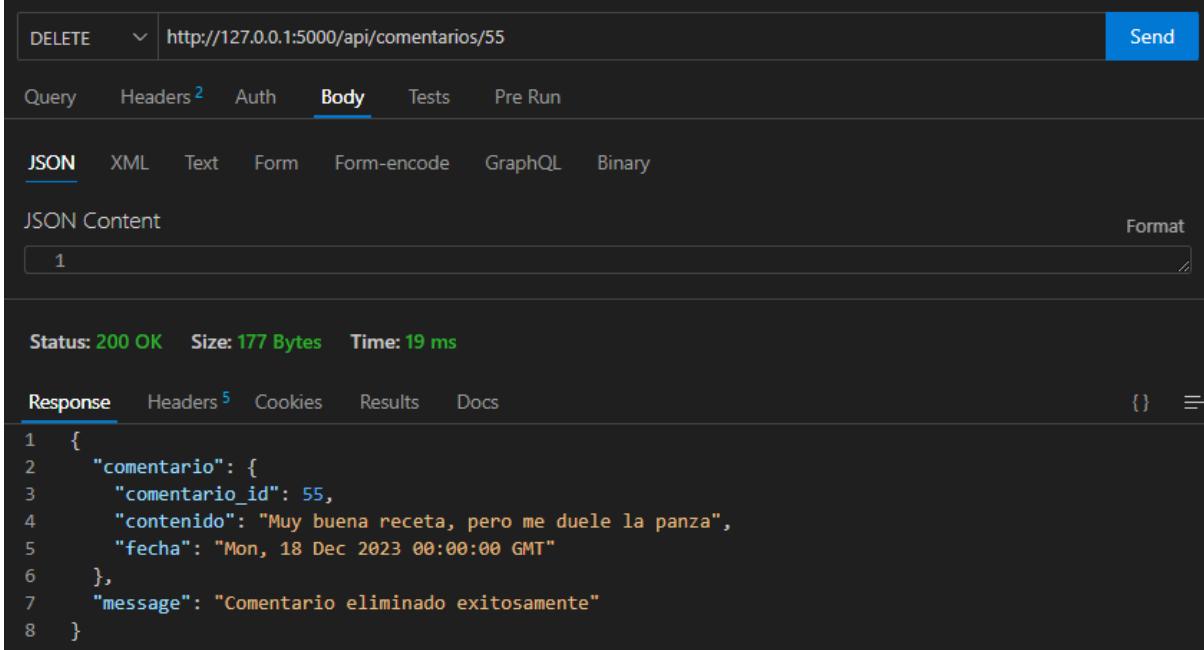
```
1 {  
2   "contenido": "Muy buena receta, pero me duele la panza"  
3 }
```

The Response tab shows the following JSON data:

```
1 {  
2   "comentario": {  
3     "comentario_id": 55,  
4     "contenido": "Muy buena receta, pero me duele la panza",  
5     "fecha": "Mon, 18 Dec 2023 00:00:00 GMT"  
6   },  
7   "message": "Comentario actualizado exitosamente"  
8 }
```

DELETE

URL : <http://127.0.0.1:5000/api/comentarios/55>



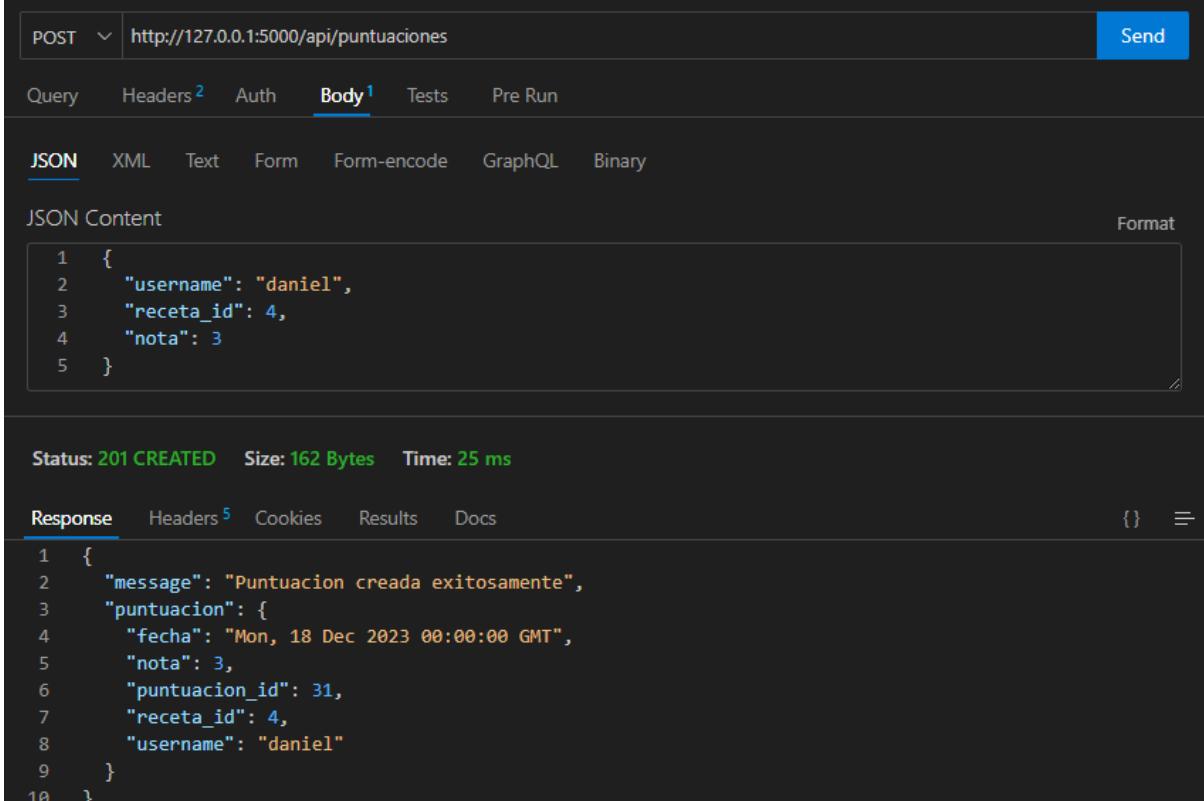
The screenshot shows a Postman request for a DELETE operation on the URL <http://127.0.0.1:5000/api/comentarios/55>. The request body is a JSON object with a single key 'message' containing the value 'Comentario eliminado exitosamente'. The response status is 200 OK, size is 177 Bytes, and time taken is 19 ms. The response body is a JSON object with a 'comentario' key, which contains an object with 'comentario_id': 55, 'contenido': 'Muy buena receta, pero me duele la panza', and 'fecha': 'Mon, 18 Dec 2023 00:00:00 GMT'. There is also a 'message' key with the value 'Comentario eliminado exitosamente'.

```
1 {  
2     "comentario": {  
3         "comentario_id": 55,  
4         "contenido": "Muy buena receta, pero me duele la panza",  
5         "fecha": "Mon, 18 Dec 2023 00:00:00 GMT"  
6     },  
7     "message": "Comentario eliminado exitosamente"  
8 }
```

Puntuación

POST

URL : <http://127.0.0.1:5000/api/puntuaciones>



POST <http://127.0.0.1:5000/api/puntuaciones> Send

Query Headers² Auth Body¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {  
2   "username": "daniel",  
3   "receta_id": 4,  
4   "nota": 3  
5 }
```

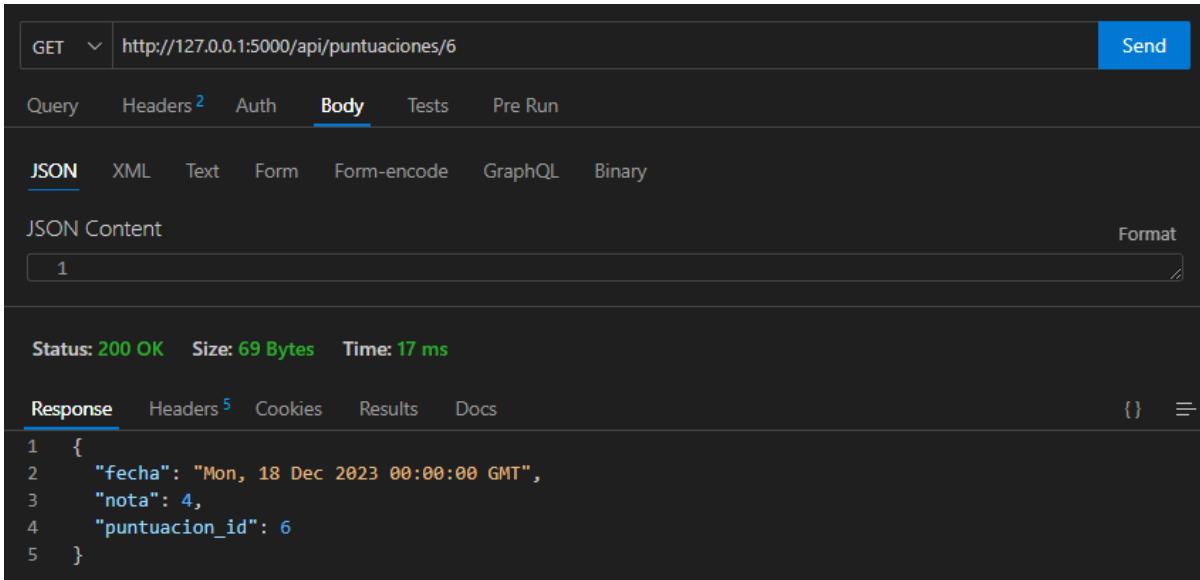
Status: 201 CREATED Size: 162 Bytes Time: 25 ms

Response Headers⁵ Cookies Results Docs {} ⚡

```
1 {  
2   "message": "Puntuacion creada exitosamente",  
3   "puntuacion": {  
4     "fecha": "Mon, 18 Dec 2023 00:00:00 GMT",  
5     "nota": 3,  
6     "puntuacion_id": 31,  
7     "receta_id": 4,  
8     "username": "daniel"  
9   }  
10 }
```

GET

URL : http://127.0.0.1:5000/api/puntuaciones/6

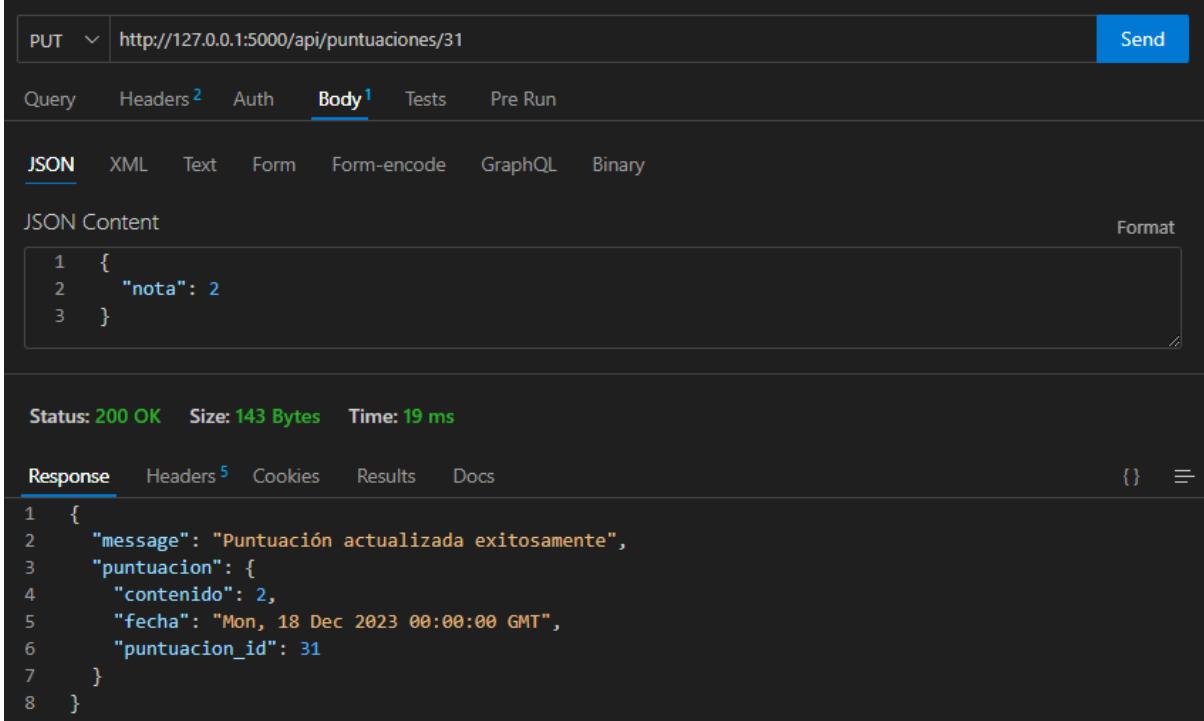


The screenshot shows a Postman request for the URL `http://127.0.0.1:5000/api/puntuaciones/6`. The request method is set to `GET`. The `Body` tab is selected, and the `JSON` tab is active. The JSON content is a single digit `1`. The response status is `200 OK`, the size is `69 Bytes`, and the time taken is `17 ms`. The response body is displayed as:

```
1 {  
2     "fecha": "Mon, 18 Dec 2023 00:00:00 GMT",  
3     "nota": 4,  
4     "puntuacion_id": 6  
5 }
```

PUT

URL : http://127.0.0.1:5000/api/puntuaciones/31



PUT http://127.0.0.1:5000/api/puntuaciones/31

Query Headers ² Auth Body ¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content

```
1  {
2    "nota": 2
3 }
```

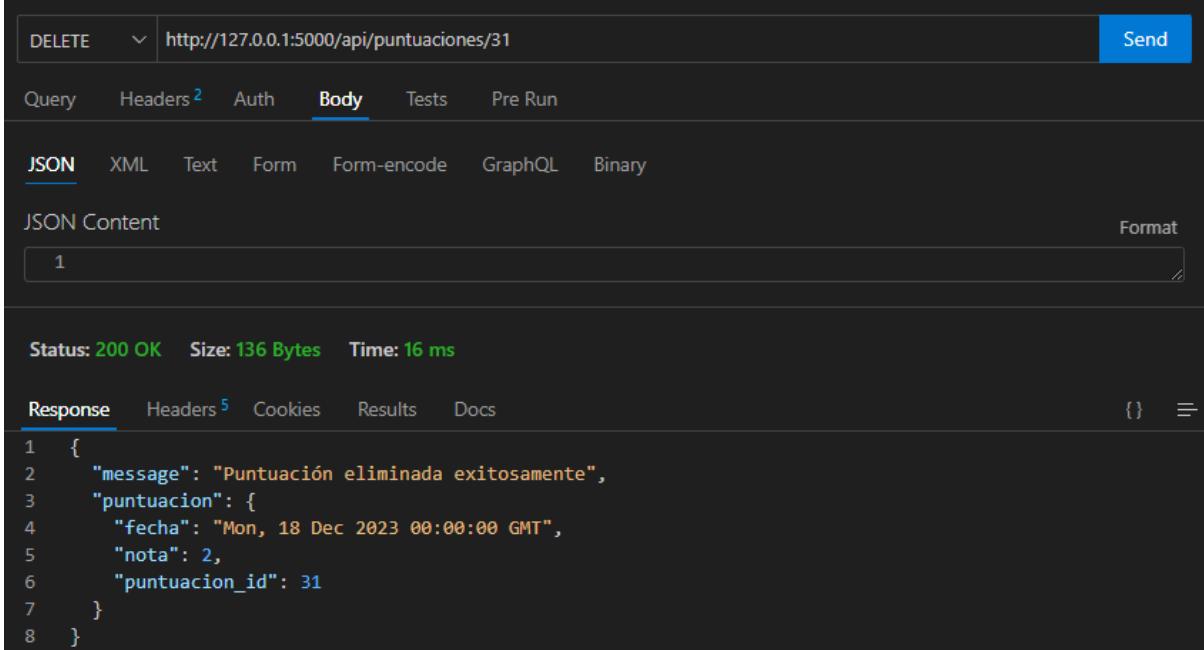
Status: 200 OK Size: 143 Bytes Time: 19 ms

Response Headers ⁵ Cookies Results Docs

```
1  {
2    "message": "Puntuación actualizada exitosamente",
3    "puntuacion": {
4      "contenido": 2,
5      "fecha": "Mon, 18 Dec 2023 00:00:00 GMT",
6      "puntuacion_id": 31
7    }
8 }
```

DELETE

URL : <http://127.0.0.1:5000/api/puntuaciones/31>



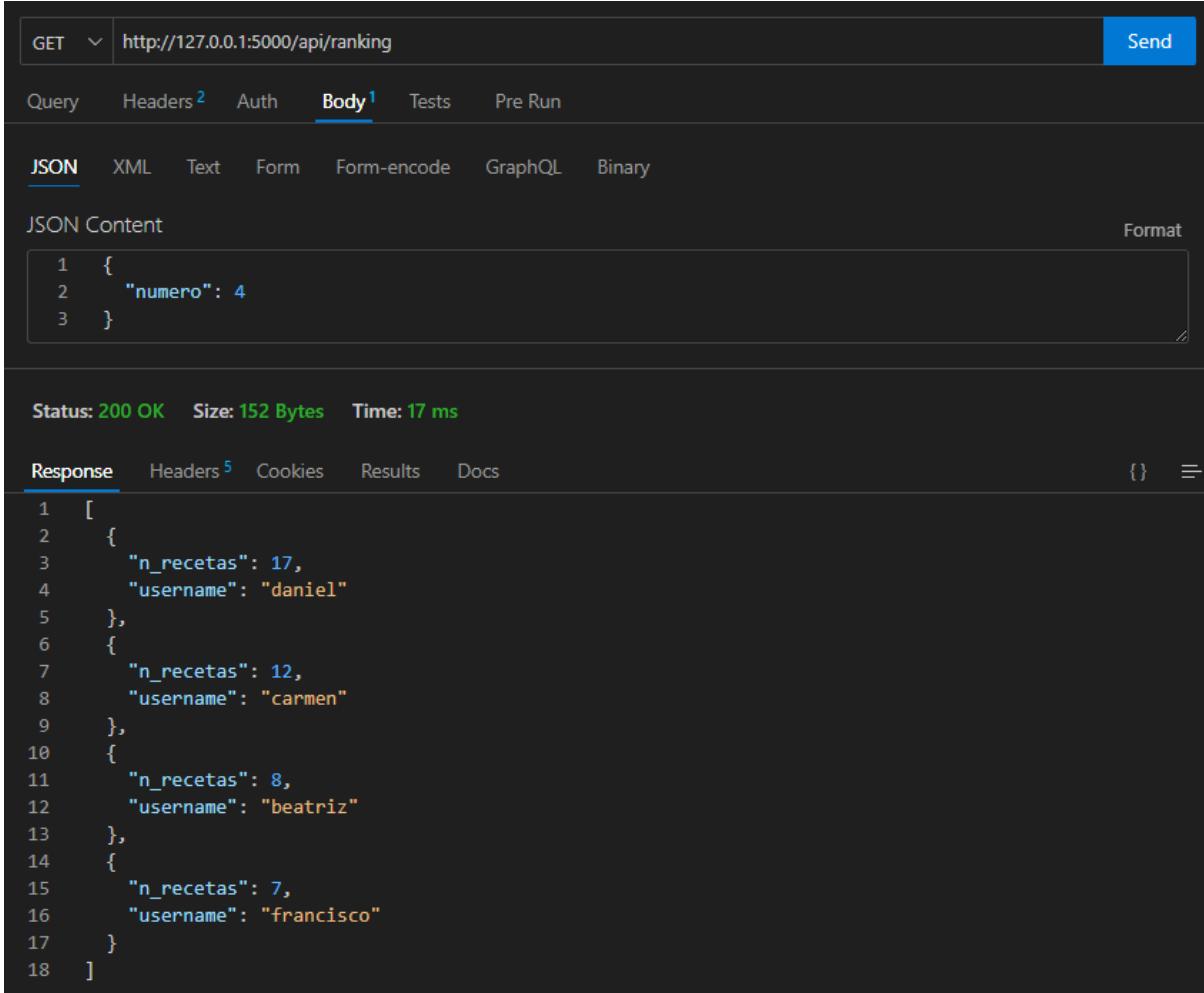
The screenshot shows a Postman request for a DELETE operation on the URL <http://127.0.0.1:5000/api/puntuaciones/31>. The request body contains the number 1. The response status is 200 OK, size is 136 Bytes, and time is 16 ms. The response JSON is:

```
1 {
2     "message": "Puntuación eliminada exitosamente",
3     "puntuacion": {
4         "fecha": "Mon, 18 Dec 2023 00:00:00 GMT",
5         "nota": 2,
6         "puntuacion_id": 31
7     }
8 }
```

Ranking

GET

URL : <http://127.0.0.1:5000/api/ranking>



GET <http://127.0.0.1:5000/api/ranking> Send

Query Headers ² Auth Body ¹ Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {  
2   "numero": 4  
3 }
```

Status: 200 OK Size: 152 Bytes Time: 17 ms

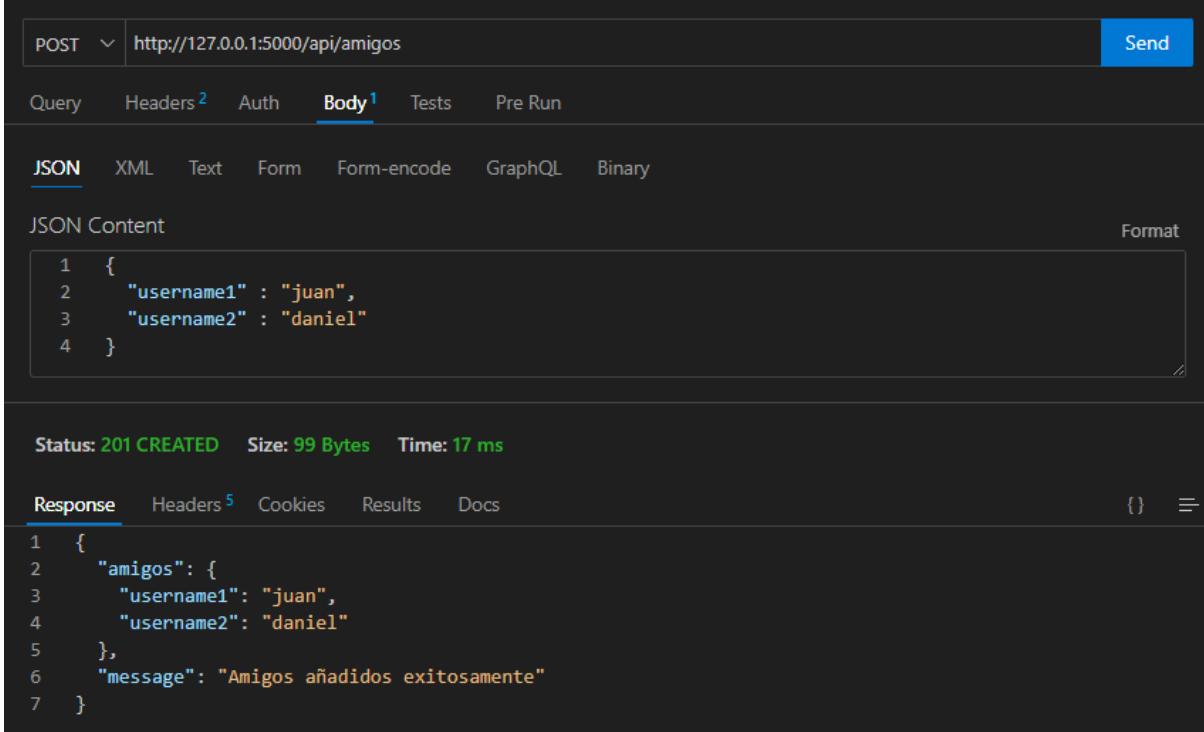
Response Headers ⁵ Cookies Results Docs {} ≡

```
1 [  
2   {  
3     "n_recetas": 17,  
4     "username": "daniel"  
5   },  
6   {  
7     "n_recetas": 12,  
8     "username": "carmen"  
9   },  
10  {  
11    "n_recetas": 8,  
12    "username": "beatriz"  
13  },  
14  {  
15    "n_recetas": 7,  
16    "username": "francisco"  
17  }  
18 ]
```

Amigo

POST

URL : <http://127.0.0.1:5000/api/amigos/juan>



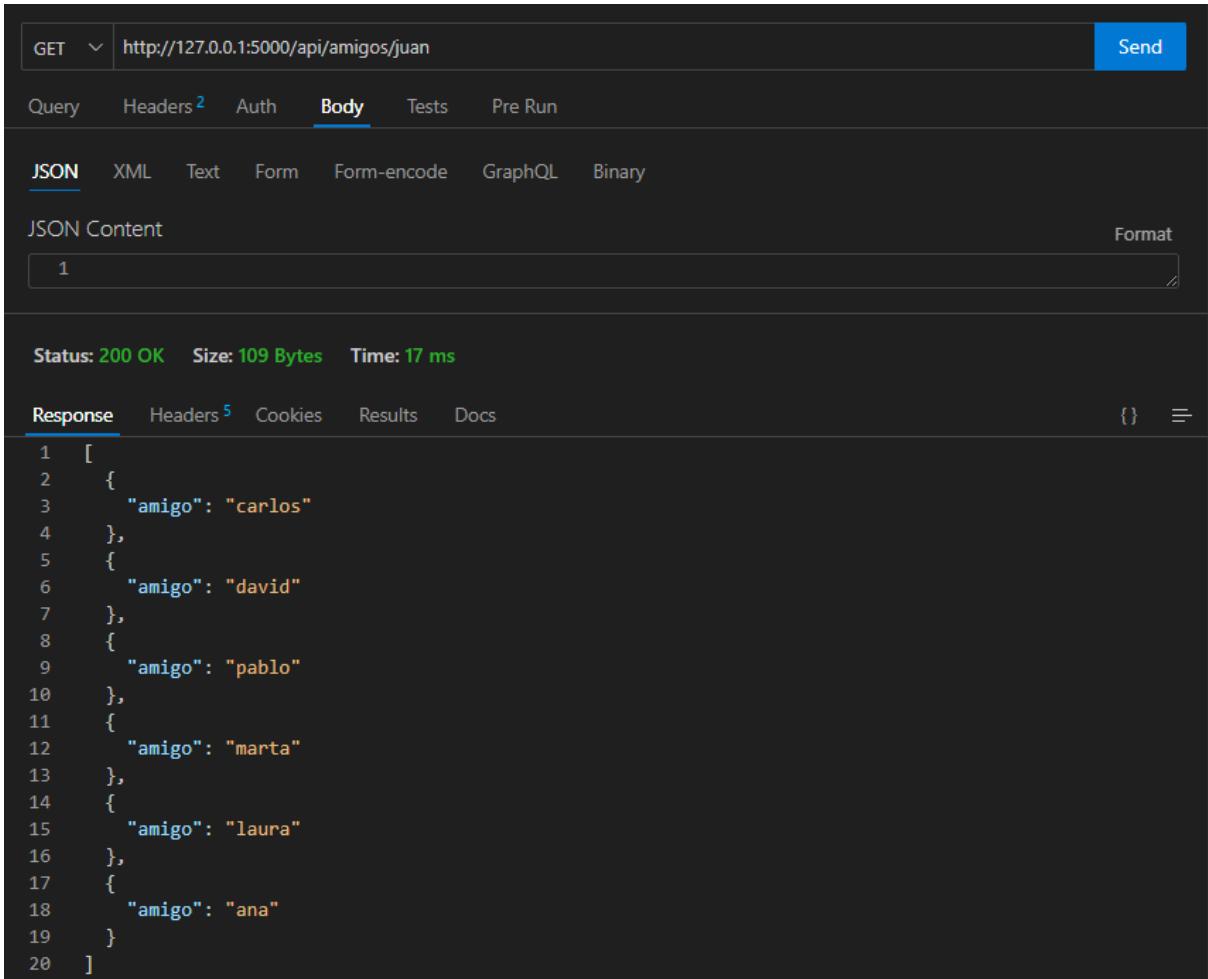
The screenshot shows a POST request in Postman. The URL is <http://127.0.0.1:5000/api/amigos>. The Body tab is selected, showing a JSON content with the following payload:

```
1 {
2   "username1" : "juan",
3   "username2" : "daniel"
4 }
```

The response status is 201 CREATED, size is 99 Bytes, and time is 17 ms. The Response tab shows the following JSON output:

```
1 {
2   "amigos": [
3     "username1": "juan",
4     "username2": "daniel"
5   ],
6   "message": "Amigos añadidos exitosamente"
7 }
```

GET

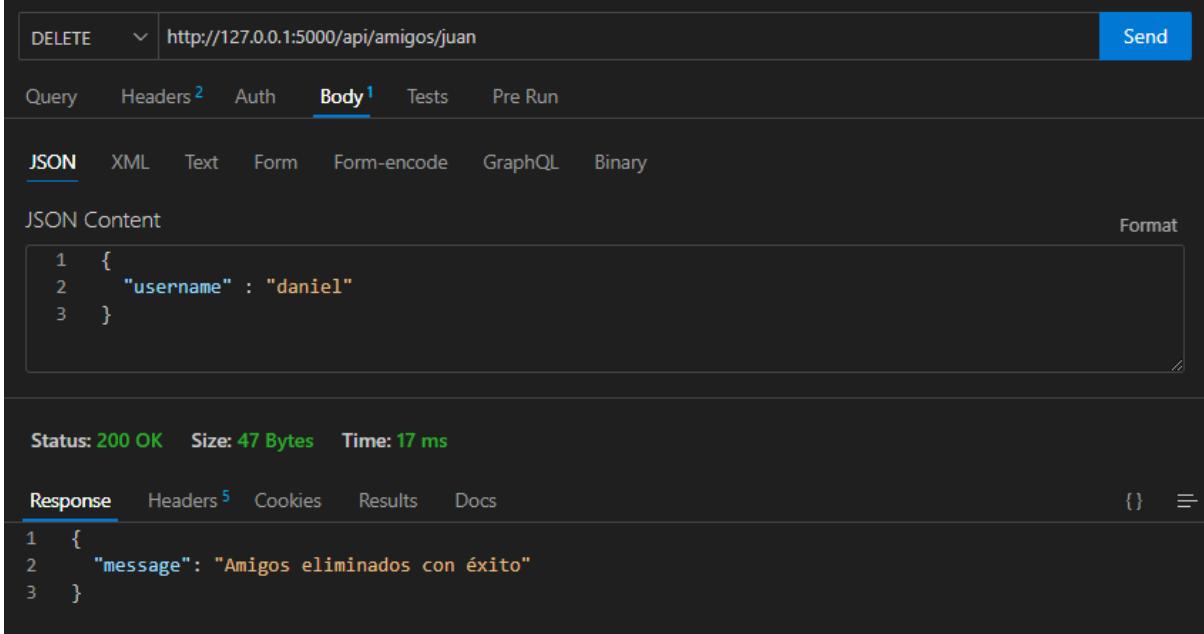
URL : <http://127.0.0.1:5000/api/amigos/juan>

The screenshot shows a Postman request configuration for a GET request to <http://127.0.0.1:5000/api/amigos/juan>. The 'Body' tab is selected, and the 'JSON' tab is active. The JSON content is a single digit '1'. The response status is 200 OK, size is 109 Bytes, and time is 17 ms. The response body is a JSON array containing five objects, each with a key 'amigo' and a value from 'carlos' to 'ana'.

```
[{"amigo": "carlos"}, {"amigo": "david"}, {"amigo": "pablo"}, {"amigo": "marta"}, {"amigo": "laura"}, {"amigo": "ana"}]
```

DELETE

URL : <http://127.0.0.1:5000/api/amigos/juan>



The screenshot shows a Postman API request interface. The method is set to **DELETE**, the URL is <http://127.0.0.1:5000/api/amigos/juan>, and the **Body** tab is selected with the **JSON** option chosen. The JSON content is:

```
1 {  
2   "username" : "daniel"  
3 }
```

The response section shows the status as **200 OK**, size as **47 Bytes**, and time as **17 ms**. The response body is:

```
1 {  
2   "message": "Amigos eliminados con éxito"  
3 }
```

Descripción de la base de datos

Tabla Receta

Una de las entidades más relevantes en nuestra base de datos sin duda alguna es *Receta*, para esta hemos definido como clave primaria un identificador que se genera a través de una serie que hemos creado. Además, hemos definido un campo *categoria_id*, el cual una clave foránea de la tabla *Categoría*, donde indicamos que si se eliminará dicha categoría, se pondrá NULL en los campos donde apareciera dicho identificador, esto se debe a que aunque la categoría se borre, no tiene sentido borrarle las recetas a nuestros usuarios.

Hemos definido que la receta tendrá un título, para evitar que sea demasiado grande, definimos una restricción que controla que no exceda los 100 caracteres. Algo similar ocurre con las instrucciones, en las cuales hemos indicado que no pueden contener más de 1000 caracteres.

En los campos numéricos controlamos que se acoten a ciertos números a través de restricciones, por ejemplo, el tiempo nos aseguramos que sea positivo, la idea es que represente los minutos. En las raciones, pretendemos que se indique el número de personas que podrían comer con las cantidades indicadas, por lo que se espera sea positivo. Por último, la dificultad pretendemos que se mueva en el rango de 1 a 5.

```

1  CREATE TABLE receta (
2    receta_id INTEGER NOT NULL,
3    titulo TEXT NOT NULL,
4    categoria_id INTEGER,
5    instrucciones TEXT NOT NULL,
6    tiempo INTEGER NOT NULL,
7    raciones INTEGER NOT NULL,
8    dificultad INTEGER NOT NULL,
9    CONSTRAINT tiempo_no_negativo CHECK (tiempo >= 0),
10   CONSTRAINT raciones_no_negativas CHECK (raciones >= 0),
11   CONSTRAINT rango_dificultad CHECK (dificultad >= 1 AND dificultad <= 5),
12   CONSTRAINT longitud_instrucciones CHECK (LENGTH(instrucciones) <= 1000),
13   CONSTRAINT longitud_titulo CHECK (LENGTH(titulo) <= 100 AND LENGTH(titulo) > 0)
14 );
15
16 ALTER TABLE public.receta OWNER TO postgres;
17
18 CREATE SEQUENCE public.receta_id_seq
19   AS integer
20   START WITH 1
21   INCREMENT BY 1
22   NO MINVALUE
23   NO MAXVALUE
24   CACHE 1;
25
26 ALTER TABLE public.receta_id_seq OWNER TO postgres;
27
28 ALTER SEQUENCE public.receta_id_seq OWNED BY public.receta.receta_id;
29
30 ALTER TABLE ONLY public.receta ALTER COLUMN receta_id SET DEFAULT nextval('public.receta_id_seq'::regclass);
31
32 ALTER TABLE ONLY public.receta
33   ADD CONSTRAINT receta_pk PRIMARY KEY (receta_id);
34
35 ALTER TABLE ONLY public.receta
36   ADD CONSTRAINT categoria_id_fkey FOREIGN KEY (categoria_id) REFERENCES public.categoria(categoria_id) ON UPDATE SET NULL ON DELETE SET NULL;
```

Tabla Usuario

Si antes mencionamos, como *Receta* era una de las entidades más relevantes de nuestra base de datos, sin duda alguna debemos decir lo mismo de *Usuario*, recordemos que serán los usuarios quienes crearan las recetas. Cada usuario tendrá un *username*, que servirá como clave primaria en la tabla, por tanto este debe ser único para cada usuario en la base de datos.

Los usuarios tendrán también su nombre y apellidos, los cuales no deben ser nulos y controlaremos que no sean excesivamente extensos. Con el fin de evitar entradas innecesariamente largas en la tabla. Veamos que incluimos restricciones para asegurarnos que los usuarios no introduzcan espacio al principio o final de sus nombre, con el fin de proporcionar mayor consistencia a los datos, aspecto que también controlamos en el *username*.

La descripción, al ser un campo de texto, también será necesario controlar su tamaño, para esto, nos aseguraremos que al menos tenga un carácter, pero no más de 250. Con el fin de evitar nuevamente entradas innecesarias o muy extensas. Por otro lado, hemos añadido una restricción que comprueba la validez del correo electrónico a través de una expresión regular.

```

● ● ●

1 CREATE TABLE usuario (
2   username TEXT,
3   nombre TEXT NOT NULL,
4   apellidos TEXT NOT NULL,
5   descripcion_perfil TEXT NOT NULL,
6   email TEXT NOT NULL,
7   CONSTRAINT no_espacios_username CHECK (username !~ '\s'),
8   CONSTRAINT no_espacios_nombre CHECK (nombre ~ '^[A-Za-z\s]*$' AND nombre !~ '^[\s|\s$]'),
9   CONSTRAINT longitud_username CHECK (LENGTH(username) <= 20 AND LENGTH(username) > 0),
10  CONSTRAINT longitud_nombre CHECK (LENGTH(nombre) <= 50),
11  CONSTRAINT no_espacios_apellidos CHECK (nombre ~ '^[A-Za-z\s]*$' AND nombre !~ '^[\s|\s$]'),
12  CONSTRAINT longitud_apellidos CHECK (LENGTH(apellidos) <= 50),
13  CONSTRAINT longitud_descripcion CHECK (LENGTH(descripcion_perfil) <= 250),
14  CONSTRAINT email_valido CHECK (email ~* '^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$')
15 );
16
17 ALTER TABLE public.usuario OWNER TO postgres;
18
19 ALTER TABLE ONLY public.usuario
20   ADD CONSTRAINT username_pk PRIMARY KEY (username);

```

Tabla Ingrediente

Los ingredientes contarán con una clave primaria que ha sido creada a partir de una serie que se ha definido. Un ingrediente contendrá un nombre y una descripción, debemos controlar que el contenido de estos campos sea distinto de cero y menos a determinado rango, con el fin de evitar entradas vacías o entradas demasiado largas, para conseguir lo anterior, hemos implementado dos restricciones que controlan que la entrada se mueva entre 1 y 50 caracteres para el nombre y 1 y 100 para la descripción del ingrediente.

Ya que deseamos que la unidad de medida tenga una forma determinada, con el fin de evitar que cada usuario final defina su propio forma y provoquen inconsistencias, hemos definido una restricción que comprueba que la unidad de medida que se introduzca coincida con alguna de las que se permiten, para definirla, hemos buscado las convenciones actuales y las más comunes.

Además, veamos que el nombre del ingrediente, cuenta con una restricción que impide que se añadan dos ingredientes con exactamente el mismo nombre, además de evitar espacio para evitar lo mismo. Esto evita entradas duplicadas de ingredientes en la tabla.

```

1 CREATE TABLE ingrediente (
2     ingrediente_id INTEGER NOT NULL,
3     nombre TEXT NOT NULL,
4     descripcion TEXT NOT NULL,
5     unidad_medida TEXT NOT NULL,
6     CONSTRAINT nombre_ingrediente UNIQUE (nombre),
7     CONSTRAINT no_espacios_nombre CHECK (nombre !~ '\s'),
8     CONSTRAINT longitud_nombre CHECK (char_length(nombre) > 0 AND char_length(nombre) <= 50),
9     CONSTRAINT longitud_descripcion CHECK (char_length(descripcion) > 0 AND char_length(descripcion) <= 100),
10    CONSTRAINT unidad_medida_valida CHECK (unidad_medida IN ('kg', 'g', 'l', 'ml', 'taza', 'cda', 'pizca'))
11 );
12
13 ALTER TABLE public.ingrediente OWNER TO postgres;
14
15 CREATE SEQUENCE public.ingrediente_id_seq
16     AS integer
17     START WITH 1
18     INCREMENT BY 1
19     NO MINVALUE
20     NO MAXVALUE
21     CACHE 1;
22
23 ALTER TABLE public.ingrediente_id_seq OWNER TO postgres;
24
25 ALTER SEQUENCE public.ingrediente_id_seq OWNED BY public.ingrediente.ingrediente_id;
26
27 ALTER TABLE ONLY public.ingrediente ALTER COLUMN ingrediente_id SET DEFAULT nextval('public.ingrediente_id_seq'::regclass);
28
29 ALTER TABLE ONLY public.ingrediente
30     ADD CONSTRAINT ingrediente_pk PRIMARY KEY (ingrediente_id);

```

Tabla Categoría

En nuestra tabla *Categoría* contaremos con una serie que será usada para definir los identificadores de manera correcta, dicho campo *categoria_id* será usado como clave primaria de la tabla.

Una categoría contará con su nombre y una breve descripción que la define, es importante controlar que no se creen categorías con nombre que contengan espacios al final al principio o el medio. Con esto buscamos evitar la creación de categorías repetidas pero con pequeñas diferencias. Hemos añadido una restricción que se encarga de controlar lo anterior a partir de una expresión regular.

En adición a lo anterior, creamos una restricción que evita crear dos categorías con exactamente el mismo nombre, esto lo hacemos indicando que el nombre, debe ser unique. Si nos fijamos, esta restricción sumada a las anteriores, impide que se agreguen categorías iguales, lo que dota de mayor coherencia a la tabla y modela mejor el funcionamiento de una categoría en la base de datos.

Además, con el fin de evitar las descripciones excesivamente largas, hemos añadido una restricción que verifica que la descripción tenga una longitud menor o igual a 250 caracteres.

```
 1 CREATE TABLE categoria (
 2   categoria_id INTEGER NOT NULL,
 3   nombre TEXT NOT NULL,
 4   descripcion TEXT NOT NULL,
 5   CONSTRAINT nombre_categoria UNIQUE (nombre),
 6   CONSTRAINT no_espacios_nombre CHECK (nombre ~ '^[A-Za-z\s]*$' AND nombre !~ '^\s|\s$'),
 7   CONSTRAINT longitud_descripcion CHECK (LENGTH(descripcion) <= 250)
 8 );
 9
10 ALTER TABLE public.categoria OWNER TO postgres;
11
12 CREATE SEQUENCE public.categoria_id_seq
13   AS integer
14   START WITH 1
15   INCREMENT BY 1
16   NO MINVALUE
17   NO MAXVALUE
18   CACHE 1;
19
20 ALTER TABLE public.categoria_id_seq OWNER TO postgres;
21
22 ALTER SEQUENCE public.categoria_id_seq OWNED BY public.categoria.categoria_id;
23
24 ALTER TABLE ONLY public.categoria ALTER COLUMN categoria_id SET DEFAULT nextval('public.categoria_id_seq'::regclass);
25
26 ALTER TABLE ONLY public.categoria
27   ADD CONSTRAINT categoria_pk PRIMARY KEY (categoria_id);
```

Tabla Patrocinador

La idea es que un usuario puede hacer un patrocinio o varios, por tanto, puede aparecer en repetidas ocasiones dentro de la tabla, es por esto que su username será una Foreign Key, pero no hará parte de la clave primaria. Respecto a lo anterior, se ha generado una serie, la cual será usada como clave primaria de la tabla.

Con el fin de evitar número excesivamente raros, hemos decidido que por entrada no se pueden superar los 6 dígitos en el campo cantidad donada, de dichos 6 dígitos, 2 serán la parte decimal. Además no se permitirán entradas negativas en la cantidad que se done, esto se consigue gracias a una restricción que verifica que sea mayor o igual a 0.

```
● ● ●
1 CREATE TABLE patrocinador(
2     patrocinador_id INTEGER NOT NULL,
3     username TEXT,
4     cantidad_donada DECIMAL(6,2),
5     CONSTRAINT cantidad_donada_no_negativa CHECK (cantidad_donada >= 0)
6 );
7
8 ALTER TABLE public.patrocinador OWNER TO postgres;
9
10 CREATE SEQUENCE public.patrocinador_id_seq
11     AS integer
12     START WITH 1
13     INCREMENT BY 1
14     NO MINVALUE
15     NO MAXVALUE
16     CACHE 1;
17
18 ALTER TABLE public.patrocinador_id_seq OWNER TO postgres;
19
20 ALTER SEQUENCE public.patrocinador_id_seq OWNED BY public.patrocinador.patrocinador_id;
21
22 ALTER TABLE ONLY public.patrocinador ALTER COLUMN patrocinador_id SET DEFAULT nextval('public.patrocinador_id_seq'::regclass);
23
24 ALTER TABLE ONLY public.patrocinador
25     ADD CONSTRAINT username_fkey FOREIGN KEY (username) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;
26
27 ALTER TABLE ONLY public.patrocinador
28     ADD CONSTRAINT patrocinador_pk PRIMARY KEY (patrocinador_id);
```

Tabla Interacción

Cada interacción se encuentra asociada a una receta y a un usuario, representando que dicha interacción ha sido realizada por el username sobre la receta_id. Por otra parte, la interacción cuenta con un identificador que es generado a través de una secuencia y que funcionará como clave primaria de la tabla.

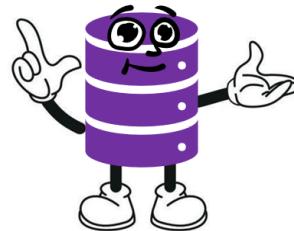
Veamos que la interacción puede representar a una puntuación o un comentario, es por esto que debemos controlar que las entradas tengan algunos de dichos valores a nulo y estrictamente el otro no. A través de una restricción comprobamos exactamente lo anterior, se verifica que si tenemos un comentario_id, no debemos tener un valor para la puntuacion_id y viceversa.

Por último, destacar que si se elimina un comentario o una puntuación, dicha entrada asociada en esta tabla también se borrará, ya que por cómo está implementado, no tiene sentido guardar una interacción que no existe además que no tendríamos ningún dato relevante del comentario o puntuación borrado.

Importante

Veamos que definir de esta forma la tabla, ofrece la capacidad de escalar fácilmente en funcionalidades, ya que si en un futuro se desea añadir un nuevo modo de interacción, como por ejemplo compartir. Solo debemos añadir una nueva columna que almacene a dicho atributo y crear la tabla respectiva, esto se consigue gracias a la abstracción de la implementación específica de lo que supone una puntuación o un comentario en tablas aparte.

Además, dicha abstracción permite que si hay que realizar cambios en la forma que se comenta o puntúa una receta, no se vean afectadas las demás entidades, al estar en tablas separadas.



```
● ● ●  
1 CREATE TABLE interaccion (  
2     interaccion_id INTEGER NOT NULL,  
3     comentario_id INTEGER DEFAULT NULL,  
4     puntuacion_id INTEGER DEFAULT NULL,  
5     username TEXT NOT NULL,  
6     receta_id INTEGER NOT NULL,  
7     CONSTRAINT check_interaccion CHECK (  
8         (comentario_id IS NOT NULL AND puntuacion_id IS NULL) OR  
9         (comentario_id IS NULL AND puntuacion_id IS NOT NULL))  
10    );  
11  
12 ALTER TABLE public.interaccion OWNER TO postgres;  
13  
14 CREATE SEQUENCE public.interaccion_id_seq  
15     AS integer  
16     START WITH 1  
17     INCREMENT BY 1  
18     NO MINVALUE  
19     NO MAXVALUE  
20     CACHE 1;  
21  
22 ALTER TABLE public.interaccion_id_seq OWNER TO postgres;  
23  
24 ALTER SEQUENCE public.interaccion_id_seq OWNED BY public.interaccion.interaccion_id;  
25  
26 ALTER TABLE ONLY public.interaccion ALTER COLUMN interaccion_id SET DEFAULT nextval('public.interaccion_id_seq'::regclass);  
27  
28 ALTER TABLE ONLY public.interaccion  
29     ADD CONSTRAINT interaccion_pk PRIMARY KEY (interaccion_id);  
30  
31 ALTER TABLE ONLY public.interaccion  
32     ADD CONSTRAINT username_fkey FOREIGN KEY (username) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;  
33  
34 ALTER TABLE ONLY public.interaccion  
35     ADD CONSTRAINT receta_id_fkey FOREIGN KEY (receta_id) REFERENCES public.receta(receta_id) ON UPDATE CASCADE ON DELETE CASCADE;  
36  
37 ALTER TABLE ONLY public.interaccion  
38     ADD CONSTRAINT comentario_id_fkey FOREIGN KEY (comentario_id) REFERENCES public.comentario(comentario_id) ON UPDATE CASCADE ON DELETE CASCADE;  
39  
40 ALTER TABLE ONLY public.interaccion  
41     ADD CONSTRAINT puntuacion_id_fkey FOREIGN KEY (puntuacion_id) REFERENCES public.puntuacion(puntuacion_id) ON UPDATE CASCADE ON DELETE CASCADE;  
42  
43 CREATE OR REPLACE FUNCTION eliminar_comentario() RETURNS TRIGGER AS $$  
44 BEGIN  
45     DELETE FROM comentario WHERE comentario_id = OLD.comentario_id;  
46     RETURN OLD;  
47 END;  
48 $$ LANGUAGE plpgsql;  
49  
50 CREATE TRIGGER eliminar_comentario_trigger  
51 AFTER DELETE ON interaccion  
52 FOR EACH ROW EXECUTE PROCEDURE eliminar_comentario();  
53  
54 CREATE OR REPLACE FUNCTION eliminar_puntuacion() RETURNS TRIGGER AS $$  
55 BEGIN  
56     DELETE FROM puntuacion WHERE puntuacion_id = OLD.puntuacion_id;  
57     RETURN OLD;  
58 END;  
59 $$ LANGUAGE plpgsql;  
60  
61 CREATE TRIGGER eliminar_puntuacion_trigger  
62 AFTER DELETE ON interaccion  
63 FOR EACH ROW EXECUTE PROCEDURE eliminar_puntuacion();
```

Tabla Comentario

Hemos creado un comentario_id el cual se llenará a partir de una serie que hemos creado, este campo funcionará como clave primaria de la tabla, permitiendo entonces, añadir múltiple comentario por ejemplo con el mismo contenido y fecha, hecho que es fácil presentarse, evidentemente estos deberán distinguirse para poder determinarlos como interacciones separadas en la tabla respectiva.

Los comentarios solo necesitan que el usuario introduzca el contenido, ya que los otros dos campos se llenan automáticamente. Decidimos que un comentario no debería superar los 500 caracteres, con el fin de evitar entradas excesivamente grandes en la tabla, para controlar esto hemos añadido una restricción que comprueba la longitud del contenido, además controla que el comentario tenga una entrada de al menos un carácter, para evitar entradas vacías a la tabla.

Para el campo fecha hemos creado una función que se lanzará antes de la inserción y actualización sobre la tabla comentario. Esta función asigna como valor al campo fecha, de la nueva entrada, la fecha actual.

```
 1 CREATE TABLE comentario (
 2   comentario_id INTEGER NOT NULL,
 3   fecha DATE NOT NULL,
 4   contenido TEXT NOT NULL,
 5   CONSTRAINT contenido_comentario CHECK (char_length(contenido) > 0 AND char_length(contenido) <= 500)
 6 );
 7
 8 ALTER TABLE public.comentario OWNER TO postgres;
 9
10 CREATE SEQUENCE public.comentario_id_seq
11   AS integer
12   START WITH 1
13   INCREMENT BY 1
14   NO MINVALUE
15   NO MAXVALUE
16   CACHE 1;
17
18 ALTER TABLE public.comentario_id_seq OWNER TO postgres;
19
20 ALTER SEQUENCE public.comentario_id_seq OWNED BY public.comentario.comentario_id;
21
22 ALTER TABLE ONLY public.comentario ALTER COLUMN comentario_id SET DEFAULT nextval('public.comentario_id_seq'::regclass);
23
24 ALTER TABLE ONLY public.comentario
25   ADD CONSTRAINT comentario_pk PRIMARY KEY (comentario_id);
26
27 CREATE OR REPLACE FUNCTION establecer_fecha() RETURNS TRIGGER AS $$%
28 BEGIN
29   NEW.fecha := CURRENT_DATE;
30   RETURN NEW;
31 END;
32 $$ LANGUAGE plpgsql;
33
34 CREATE TRIGGER establecer_fecha_comentario
35 BEFORE INSERT OR UPDATE ON comentario
36 FOR EACH ROW
37 EXECUTE PROCEDURE establecer_fecha();
```

Tabla Puntuación

En *Puntuación* el identificador lo generamos a partir de una serie que creamos y usamos dicho campo como la clave primaria de la tabla. Además, similar a como ocurre con *Comentario*, necesitaremos que se indique tan solo la nota, ya que los demás campos se rellenan automáticamente. Para controlar que la nota se mueva en el rango de 1 a 5, hemos incluido una restricción que lo comprueba. Finalmente, hemos incluido en trigger y la función encargada de añadir la fecha actual una vez se inserta una entrada.

```
● ● ●
1 CREATE TABLE puntuacion (
2     puntuacion_id INTEGER NOT NULL,
3     fecha DATE NOT NULL,
4     nota INTEGER NOT NULL,
5     CONSTRAINT nota_check CHECK (nota > 0 AND nota <= 5)
6 );
7
8 ALTER TABLE public.puntuacion OWNER TO postgres;
9
10 CREATE SEQUENCE public.puntuacion_id_seq
11     AS integer
12     START WITH 1
13     INCREMENT BY 1
14     NO MINVALUE
15     NO MAXVALUE
16     CACHE 1;
17
18 ALTER TABLE public.puntuacion_id_seq OWNER TO postgres;
19
20 ALTER SEQUENCE public.puntuacion_id_seq OWNED BY public.puntuacion.puntuacion_id;
21
22 ALTER TABLE ONLY public.puntuacion ALTER COLUMN puntuacion_id SET DEFAULT nextval('public.puntuacion_id_seq'::regclass);
23
24 ALTER TABLE ONLY public.puntuacion
25     ADD CONSTRAINT puntuacion_pk PRIMARY KEY (puntuacion_id);
26
27 CREATE OR REPLACE FUNCTION establecer_fecha() RETURNS TRIGGER AS $$%
28 BEGIN
29     NEW.fecha := CURRENT_DATE;
30     RETURN NEW;
31 END;
32 $$ LANGUAGE plpgsql;
33
34 CREATE TRIGGER establecer_fecha_puntuacion
35 BEFORE INSERT OR UPDATE ON puntuacion
36 FOR EACH ROW
37 EXECUTE PROCEDURE establecer_fecha();
```

Tabla Ranking

En cuanto a la tabla *Ranking*, vemos que es bastante sencilla, esta nos sirve para almacenar para cada usuario su número de recetas y de esta forma poder obtener un registro actualizado de la actividad de los usuarios en la parte creativa. En esta tabla, el usuario final no hace adiciones manuales, sino que se rellena automáticamente a partir de un trigger implementado en la tabla *usuario_receta* donde por cada inserción se actualiza la tabla *Ranking*.

Veamos que no controlamos la entrada ordenada de la tabla, esto se debe a una decisión de diseño ya que consideramos que se harán bastantes inserciones en la tabla *usuario_receta* y por tanto, saldría realmente costoso ir actualizando la tabla cada vez. En su lugar, si el usuario final desea obtener la tabla *Ranking* ordenada, podrá ordenarla en la propia consulta, ahorrando entonces recursos de cómputo.

```
● ● ●
1 CREATE TABLE ranking (
2   username TEXT,
3   n_recetas INTEGER
4 );
5
6 ALTER TABLE ONLY public.ranking
7   ADD CONSTRAINT username_fkey FOREIGN KEY (username) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;
```

Tabla Amigo

La tabla amigo modela la amistad entre dos usuarios y no tiene relevancia el orden en el que se presente la relación, ya que se dicha relación es bidireccional, es decir, tanto username1 es amigo de username2, como username2 es amigo de username1.

Respecto a lo anterior, debemos prevenir que se intente agregar dos entradas con los mismos username pero en orden diferente, ya que esto no introduce una entrada incorrecta en lo que supone la lógica de la tabla, para esto hemos desarrollado una función *comprobar_amistad()* que se encarga de verificar si existe una entrada que representa la amistad, pero en un orden distinto al de la entrada nueva, en caso de que se cumpla, se lanza una excepción. Esta función será lanzada por un trigger antes de cada inserción en la tabla amigo por cada fila.

Evidentemente debemos evitar que se intente agregar una amistad entre el usuario y sí mismo, para esto añadimos una constraint que verifica que los username sean diferentes.

```
1 CREATE TABLE amigo (
2   username1 TEXT,
3   username2 TEXT,
4   CONSTRAINT amigos_diferentes CHECK (username1 != username2)
5 );
6
7 ALTER TABLE public.amigo OWNER TO postgres;
8
9 ALTER TABLE ONLY public.amigo
10 ADD CONSTRAINT username1_fkey FOREIGN KEY (username1) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;
11
12 ALTER TABLE ONLY public.amigo
13 ADD CONSTRAINT username2_fkey FOREIGN KEY (username2) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;
14
15 ALTER TABLE ONLY public.amigo
16 ADD CONSTRAINT amigo_pk PRIMARY KEY (username1, username2);
17
18 CREATE OR REPLACE FUNCTION comprobar_amistad() RETURNS TRIGGER AS $$%
19 BEGIN
20   IF EXISTS (SELECT 1 FROM amigo WHERE (username1 = NEW.username2 AND username2 = NEW.username1)) THEN
21     RAISE EXCEPTION 'La amistad ya existe.';
22   END IF;
23   RETURN NEW;
24 END;
25 $$ LANGUAGE plpgsql;
26
27 CREATE TRIGGER comprobar_amistad_before_insert
28 BEFORE INSERT ON amigo
29 FOR EACH ROW EXECUTE PROCEDURE comprobar_amistad();
```

Tabla usuario_receta

Finalmente, recordemos que debemos modelar que una receta es creada por un único usuario, para esto hemos definido la tabla *usuario_receta*, donde el username será la clave foranea de la tabla *Usuario* y *receta_id* la clave foranea de *Receta*. Además, estos dos campos componen la clave primaria, con el fin de evitar introducir la misma receta y usuario más de una vez, pero además serán unique, con el fin de que ningún otro usuario pueda añadir una receta ya agregada.

Como mencionamos anteriormente, tenemos un trigger que se dispara despues de una inserción, actualización o eliminación de una entrada a la tabla, con el fin de mandar a actualizar la tabla *Ranking* la cual depende directamente del número de apariciones de un usuario en esta tabla *usuario_receta*.

Importante destacar que hemos definido la actualización y eliminación en cascada, por tanto, si un usuario o receta se borrara, se eliminaría en cascada todas las entradas presentes en la tabla de dicha clave foránea. Esto se ha decidido ya que si una receta o un usuario no existen, no tiene sentido definir su relación.

```

1 CREATE TABLE usuario_receta (
2   username TEXT,
3   receta_id INTEGER,
4   CONSTRAINT receta_id_unica UNIQUE (receta_id)
5 );
6
7 ALTER TABLE public.usuario_receta OWNER TO postgres;
8
9 ALTER TABLE ONLY public.usuario_receta
10 ADD CONSTRAINT username_fkey FOREIGN KEY (username) REFERENCES public.usuario(username) ON UPDATE CASCADE ON DELETE CASCADE;
11
12 ALTER TABLE ONLY public.usuario_receta
13 ADD CONSTRAINT receta_id_fkey FOREIGN KEY (receta_id) REFERENCES public.receta(receta_id) ON UPDATE CASCADE ON DELETE CASCADE;
14
15 ALTER TABLE ONLY public.usuario_receta
16 ADD CONSTRAINT usuario_receta_pk PRIMARY KEY (username, receta_id);
17
18 CREATE OR REPLACE FUNCTION actualizar_ranking() RETURNS TRIGGER AS $$%
19 BEGIN
20   -- Actualizar el ranking del usuario si ya existe
21   IF EXISTS (SELECT 1 FROM ranking WHERE username = NEW.username) THEN
22     UPDATE ranking
23     SET n_recetas = (SELECT COUNT(*) FROM usuario_receta WHERE username = NEW.username)
24     WHERE username = NEW.username;
25   -- Si no existe, insertar un nuevo registro en el ranking
26   ELSE
27     INSERT INTO ranking(username, n_recetas)
28     VALUES (NEW.username, (SELECT COUNT(*) FROM usuario_receta WHERE username = NEW.username));
29   END IF;
30   RETURN NEW;
31 END;
32 $$ LANGUAGE plpgsql;
33
34 CREATE TRIGGER actualizar_ranking_trigger
35 AFTER INSERT OR UPDATE OR DELETE ON usuario_receta
36 FOR EACH ROW EXECUTE PROCEDURE actualizar_ranking();
37
38 CREATE OR REPLACE FUNCTION eliminar_receta() RETURNS TRIGGER AS $$%
39 BEGIN
40   DELETE FROM receta WHERE receta_id = OLD.receta_id;
41   RETURN OLD;
42 END;
43 $$ LANGUAGE plpgsql;
44
45 CREATE TRIGGER eliminar_receta_trigger
46 AFTER DELETE ON usuario_receta
47 FOR EACH ROW EXECUTE PROCEDURE eliminar_receta();
```

Tabla receta_ingrediente

Toda receta tiene sus ingredientes, la idea en esta tabla es representar la relación entre un ingrediente y una receta, es decir, los ingredientes que se necesitan para realizar una determinada receta. Es por esto que *receta_id* e *ingrediente_id* serán claves foráneas de *Receta* e *Ingrediente* respectivamente. Además, ambos atributos componen la clave primaria, ya que deseamos que solo se pueda incluir una vez el mismo ingrediente para una determinada receta.

Destacar, que en caso de eliminarse un ingrediente o una receta, hemos definido que se eliminen las entradas presentes en nuestra tabla *receta_ingredient* que contengan dicha clave foránea, con el fin de evitar errores y ahorrar espacio, ya que no tiene sentido seguir guardando dicha información en la base de datos.

```
CREATE TABLE receta_ingredient (
    receta_id INTEGER,
    ingrediente_id INTEGER
);

ALTER TABLE public.receta_ingredient OWNER TO postgres;

ALTER TABLE ONLY public.receta_ingredient
    ADD CONSTRAINT receta_id_fkey FOREIGN KEY (receta_id) REFERENCES public.receta(receta_id) ON UPDATE CASCADE ON DELETE CASCADE;
ALTER TABLE ONLY public.receta_ingredient
    ADD CONSTRAINT ingrediente_id_fkey FOREIGN KEY (ingrediente_id) REFERENCES public.ingrediente(ingrediente_id) ON UPDATE CASCADE ON DELETE CASCADE;
ALTER TABLE ONLY public.receta_ingredient
    ADD CONSTRAINT receta_ingredient_pk PRIMARY KEY (receta_id, ingrediente_id);
```

API REST

A continuación, se presentan los distintos recursos que proporciona nuestra API, indicando cada uno de los métodos y URLs disponibles y las acciones asociadas a cada uno.

Receta APIs

Método	URL	Acción
POST	/api/recetas	Añadir una nueva receta especificando los valores en el body
GET	/api/recetas/:receta_id	Obtener una receta por :receta_id
GET	/api/recetas	Obtener todas las recetas o las primeras n recetas, especificando n en el body
PUT	/api/recetas/:receta_id	Modificar una receta por :receta_id especificando los valores en el body
DELETE	/api/recetas/:receta_id	Eliminar una receta por :receta_id

Parámetros	Definición	POST	GET	GET	PUT	DELETE
:receta_id	Número entero para identificar la receta	-	✓	-	✓	✓
Body	Definición	POST	GET	GET	PUT	DELETE
titulo	Cadena de texto [1 - 100] caracteres para describir brevemente la receta	✓	-	-	*	-
categoria_id	Número entero que identifica a la categoría que pertenece la receta	✓	-	-	*	-
instrucciones	Cadena de texto [1 - 1000] caracteres para apellidos del usuario	✓	-	-	*	-
tiempo	Número entero que representa los minutos aproximados que tarda en hacerse la receta	✓	-	-	*	-
raciones	Número entero que representa cuántas personas pueden comer de la receta.	✓	-	-	*	-

dificultad	Número entero entre 1 y 5 que representa la dificultad de la receta	✓	-	-	★	-
numero	Número de usuarios a devolver	-	-	★	-	-
✓ → Obligatorio ★ → Opcional - → No aplica						

Hemos considerado conveniente proporcionar todas las operaciones CRUD sobre una receta, ya que es el elemento fundamental en la aplicación. Al devolver una receta, hemos reunido todos los atributos que contiene de otras tablas, como los ingredientes, de esta forma cuando se recupera una receta se tienen todos los elementos necesarios para poder representarla.

Usuario APIs

Método	URL	Acción
POST	/api/usuarios	Añadir un nuevo usuario especificando los valores en el body
GET	/api/usuarios/:username	Obtener un usuario por :username
GET	/api/usuarios	Obtener los 10 primeros usuarios o los primeros n usuarios, especificando n en el body
PUT	/api/usuarios/:username	Modificar un usuario por :username especificando los valores en el body
DELETE	/api/usuarios/:username	Eliminar un usuario por :username

Parámetros	Definición	POST	GET	GET	PUT	DELETE
:username	Cadena de texto [1 - 20] caracteres para identificar al usuario	-	✓	-	✓	✓
Body	Definición	POST	GET	GET	PUT	DELETE
username	Cadena de texto [1 - 20] caracteres para identificar al usuario	✓	-	-	-	-

nombre	Cadena de texto [1 - 50] caracteres para nombre del usuario		-	-		-
apellidos	Cadena de texto [1 - 50] caracteres para apellidos del usuario		-	-		-
descripcion_perfil	Cadena de texto [1 - 250] caracteres para breve descripción del usuario		-	-		-
email	Correo electrónico del usuario		-	-		-
numero	Número de usuarios a devolver	-	-		-	-
→ Obligatorio → Opcional - → No aplica						

Recordemos que los recursos que proporcionamos no tienen porque coincidir exactamente con las tablas que tenemos en la base de datos, por ejemplo, no hemos incluido un recurso específico para *usuario_receta*, pero si recuperamos un usuario, obtendremos todas sus recetas.

Por otro lado, en el método que devuelve 10 o X usuarios, hemos decidido no devolver toda la información de un usuario, sino únicamente lo primordial para poder acceder a cualquier recurso referente al usuario. De esta forma hacemos un poco más ligera la respuesta y evitamos pasar elementos que a lo mejor no se necesiten. Si el usuario necesitará más información, siempre podrá usar el get individual con el identificador correspondiente.

decir que no pusimos una tabla *usuario_receta* porque ya mostramos las recetas de un usuario en la info del propio usuario → lo mismo pasa con los ingredientes y las recetas

no mostremos las recetas que hizo cada usuario cuando pedimos por ejemplo 20 usuarios porque lo vimos engorroso tenemos que explicar que hace cada consulta

Ingrediente APIs

Método	URL	Acción
POST	/api/ingredientes	Añadir un nuevo ingrediente especificando los valores en el body
GET	/api/ingredientes/:ingrediente_id	Obtener un ingrediente por :ingrediente_id
GET	/api/ingredientes	Obtener los 10 primeros ingredientes o los primeros n ingredientes, especificando n en el body

Parámetros	Definición	POST	GET	GET
:ingrediente_id	Número entero de identificación del ingrediente	-	✓	-
Body	Definición	POST	GET	GET
nombre	Cadena de texto [1 - 50] caracteres para nombre del ingrediente	✓	-	-
descripcion	Cadena de texto [1 - 100] caracteres para breve descripción del ingrediente	✓	-	-
unidad_medida	Cadena de texto que puede tomar los valores de ('kg', 'g', 'l', 'ml', 'taza', 'cda', 'pizca')	✓	-	-
numero	Número de ingredientes a devolver	-	-	★
✓ → Obligatorio ★ → Opcional - → No aplica				

En este caso, consideramos que no es correcto proporcionar las operaciones de modificación y eliminación de ingredientes, ya que puede llevar a modificaciones indebidas de estos y al ser elementos que se ven involucrados con múltiples recetas y por tanto múltiples usuarios, lo mejor es no proporcionar las funciones de modificación y borrado, para que solo se pueda hacer de forma controlada por el administrador de la base de datos.

Si que permitimos la creación de ingredientes, porque esta no supone un problema en múltiples usuarios o recetas y además tenemos controlada la creación de ingredientes repetidos, por lo tanto se espera que si se añade un ingrediente, sea realmente algo nuevo.

En la creación de un ingrediente se debe indicar de forma correcta la unidad de medida, para más información consultar el [manual de usuario](#).

Categoría APIs

Método	URL	Acción
GET	/api/categorias/:categoria_id	Obtener una categoría por :categoria_id
GET	/api/categorias	Obtener las 10 primeras categorías o las primeras <i>número</i> categorías, especificando <i>número</i> en el body

Parámetros	Definición	GET	GET
:categoria_id	Número entero del identificador de categoría	✓	-
Body	Definición	GET	GET
numero	Número de categorías a devolver	-	✳
✓ → Obligatorio ✳ → Opcional - → No aplica			

Hemos decidido que la creación de categorías no debe permitirse al usuario final de la API, sino que debe ser de forma controlada por parte del administrador de la base de datos. Esto se debe a que las categorías, son elementos que se ven relacionados con muchas recetas y por tanto también usuarios, debido a eso tampoco deben modificar o eliminar las categorías.

Patrocinador APIs

Método	URL	Acción
GET	/api/patrocinadores/:patrocinador_id	Obtener un patrocinador por :patrocinador_id
GET	/api/patrocinadores	Obtener los 10 primeros patrocinadores o los primeros n patrocinadores, especificando n en el body, ordenando los patrocinadores por cantidad total donada

Parámetros	Definición	GET	GET
:patrocinador_id	Número entero del identificador del patrocinio	✓	-
Body	Definición	GET	GET
numero	Número de categorías a devolver	-	★
✓ → Obligatorio ★ → Opcional - → No aplica			

Los patrocinadores son usuarios puntuales que han donado a la plataforma de alguna forma, si que permitiremos acceder a dicho valor, pero solo podrán ser modificados a través del administrador de la base de datos.

Comentario APIs

Método	URL	Acción
POST	/api/comentarios	Añadir un nuevo comentario especificando los valores en el body
GET	/api/comentarios/:comentario_id	Obtener un comentario por :comentario_id
PUT	/api/comentarios/:comentario_id	Modificar un comentario por :comentario_id especificando los valores en el body
DELETE	/api/comentarios/:comentario_id	Eliminar un comentario por :comentario_id

Parámetros	Definición	POST	GET	PUT	DELETE
:comentario_id	Número entero del identificador del comentario	-	✓	✓	✓
Body	Definición	POST	GET	PUT	DELETE
username	Cadena de texto [1 - 20] caracteres para identificar al usuario	✓	-	-	-
receta_id	Número entero que identifica a una receta	✓	-	-	-
contenido	Cadena de texto [1 - 500] caracteres con el contenido del comentario	✓	-	✓	-
numero	Número de comentarios a devolver	-	-	-	-
✓ → Obligatorio - → No aplica					

En el desarrollo de la API, decidimos que si queremos ver todos los comentarios de una receta, podemos verlos desde la consulta específica de una receta (revisar tabla [Receta APIs](#)), lo mismo ocurre cuando consultamos un usuario específico (revisar tabla [Usuario APIs](#)), donde podremos ver sus comentarios. En este caso los comentarios, si que es esencial proporcionar todas las operaciones CRUD.

Puntuación APIs

Método	URL	Acción
POST	/api/puntuaciones	Añadir una nueva puntuación especificando los valores en el body
GET	/api/puntuaciones/:puntuacion_id	Obtener una puntuación por :puntuacion_id
PUT	/api/puntuaciones/:puntuacion_id	Modificar una puntuación por :puntuacion_id especificando la puntuación en el body
DELETE	/api/puntuaciones/:puntuacion_id	Eliminar una puntuación por :puntuacion_id

Parámetros	Definición	POST	GET	PUT	DELETE
:puntuacion_id	Número entero del identificador de la puntuación	-	✓	✓	✓
Body	Definición	POST	GET	PUT	DELETE
username	Cadena de texto [1 - 20] caracteres para identificar al usuario	✓	-	-	-
receta_id	Número entero que identifica a una receta	✓	-	-	-
nota	Número entre 1 a 5 que representa la calificación a una receta	✓	-	✓	-
numero	Número de comentarios a devolver	-	-	-	-

✓ → Obligatorio
- → No aplica

Similar a los comentarios, hemos proporcionado todas las operaciones CRUD para las puntuaciones y podemos obtener todas las puntuaciones de una recetas, si consultamos una receta en específico (revisar tabla [Receta APIs](#)), o todas las puntuaciones que un usuario ha dado, si consultamos un usuario en específico (revisar tabla [Usuario APIs](#)).

Ranking APIs

Método	URL	Acción
GET	/api/ranking	Obtener el ranking de usuarios, especificando el número de usuarios que deben aparecer en el body, basándose en el número de recetas aportadas a la plataforma

Body	Definición	GET
numero	Número de usuarios más arriba del ranking a devolver	★
★ → Opcional		

En el caso del ranking, solamente se podrá acceder a ver los datos, por defecto aparecen los 10 primeros, ordenados por número de recetas subidas a la plataforma, es decir el usuario con más recetas es el primero en el ranking. Si se indica por el body el *numero* de usuarios, se devuelve esa cantidad en orden.

Amigo APIs

Método	URL	Acción
POST	/api/amigos	Añadir dos amigos, ambos username se deben pasar en el body
GET	/api/amigos/:username	Obtener todos los amigos de :username
DELETE	/api/amigos/:username	Eliminar un amigo para :username, se debe pasar el username del amigo a eliminar en el body

Parámetros	Definición	POST	GET	DELETE
:username	Cadena de texto [1 - 20] caracteres para identificar al usuario	-	✓	✓
Body	Definición	POST	GET	DELETE
username1	Cadena de texto [1 - 20] caracteres para identificar a uno de los amigos	✓	-	-
username2	Cadena de texto [1 - 20] caracteres para identificar al otro amigo	✓	-	-
username	Cadena de texto [1 - 20] caracteres que identifica al amigo que se desea eliminar	-	-	✓
✓ → Obligatorio - → No aplica				

Podemos crear un amigo, pasando por el body los usuarios que deseamos representar su amistad. Podemos además eliminar y ver las amistades de un usuario indicado. Sin embargo, no proporcionamos un método de modificación ya que no tiene sentido en la entidad.

Ejemplos de consultas

A continuación hemos realizado una serie de consultas para poder comprobar el funcionamiento de la base de datos. Las consultas van desde aspectos muy simples hasta consultas más complejas en las que se realizan varias operaciones de unión.

La primera consulta determina el número de interacciones de usuarios descontentos sobre las recetas, es decir, las puntuaciones que varían en el rango de 1 a 3.

1. **SELECT COUNT(*)**
2. **FROM** puntuacion
3. **WHERE** (nota > 0 **AND** nota <= 3);

La siguiente consulta indica el top 10 de usuarios con más recetas aportadas a la plataforma, esto se realiza gracias a la tabla ranking que recoge el número de recetas que ha subido cada usuario a la plataforma.

1. **SELECT** username
2. **FROM** ranking
3. **GROUP BY** username, n_recetas
4. **ORDER BY** n_recetas **DESC**
5. **LIMIT** 10;

Esta consulta permite determinar el tiempo medio de las recetas asociadas a la categoría *Ensaladas*. Es una consulta bastante natural y determinamos que puede llegar a ser un caso real de consulta.

1. **SELECT AVG(tiempo)**
2. **FROM** receta **JOIN** categoria
3. **ON** receta.categoria_id = categoria.categoria_id
4. **GROUP BY** categoria.nombre
5. **HAVING** categoria.nombre = 'Ensaladas';

La consulta que aparece a continuación especifica el nombre de las recetas que contengan *sal* como ingrediente y que pertenezcan a la categoría *Desayunos*. También vimos esta consulta como algo natural, podemos poner como ejemplo el caso en el que una persona quiere hacerle el desayuno a otra persona pero esta última es hipertensa.

1. **SELECT** receta.titulo
2. **FROM** receta
3. **JOIN** receta_ingrediente **ON** receta.receta_id = receta_ingrediente.receta_id
4. **JOIN** ingrediente **ON** receta_ingrediente.ingrediente_id = ingrediente.ingrediente_id
5. **JOIN** categoria **ON** receta.categoria_id = categoria.categoria_id
6. **WHERE** categoria.nombre = 'Desayunos'
7. **GROUP BY** receta.receta_id, ingrediente.ingrediente_id
8. **HAVING** ingrediente.nombre = 'Sal';

Esta consulta determina la lista de usuarios que han realizado recetas difíciles, esto quiere decir que han realizado recetas cuya dificultad tiene el valor de 5.

1. **SELECT DISTINCT** usuario.username
2. **FROM** usuario **JOIN** usuario_receta
3. **ON** usuario.username = usuario_receta.username
4. **JOIN** receta
5. **ON** receta.receta_id = usuario_receta.receta_id
6. **GROUP BY** usuario.username, receta.receta_id
7. **HAVING** receta.dificultad = 5;

La consulta que se encuentra a continuación determina la lista de usuarios que han aportado a la plataforma alguna receta que se asocia con la categoría *Vegetarianas*.

1. **SELECT DISTINCT** usuario.username
2. **FROM** usuario **JOIN** usuario_receta
3. **ON** usuario.username = usuario_receta.username
4. **JOIN** receta
5. **ON** receta.receta_id = usuario_receta.receta_id
6. **JOIN** categoria
7. **ON** receta.categoria_id = categoria.categoria_id
8. **GROUP BY** categoria.nombre, usuario.username, receta.receta_id
9. **HAVING** categoria.nombre = 'Vegetarianas';

La siguiente consulta determina aquellas recetas con más de 7 ingredientes.

1. **SELECT** receta_id, count(*) **AS** cantidad_ingredientes
2. **FROM** receta_ingrediente
3. **GROUP BY** receta_id
4. **HAVING** count(*) > 7;

A continuación aparece la consulta que imprime por pantalla el nombre de los ingredientes que aparecen en recetas asociadas a la categoría *Gourmet*.

1. **SELECT DISTINCT** ingrediente.nombre
2. **FROM** receta_ingrediente
3. **JOIN** ingrediente
4. **ON** receta_ingrediente.ingrediente_id = ingrediente.ingrediente_id
5. **JOIN** receta
6. **ON** receta_ingrediente.receta_id = receta.receta_id
7. **JOIN** categoria
8. **ON** receta.categoria_id = categoria.categoria_id
9. **WHERE** categoria.nombre = 'Gourmet';

La consulta que aparece a continuación también nos pareció interesante añadirla, ya que muestra la lista de usuarios que han realizado y aportado a la plataforma recetas de más de una hora de duración.

1. **SELECT DISTINCT** username
2. **FROM** usuario_receta **JOIN** receta
3. **ON** usuario_receta.receta_id = receta.receta_id
4. **WHERE** tiempo >= 60;

La siguiente consulta es una de las consultas más típicas que puede darse en nuestra plataforma, ya que muestra los títulos de las recetas que ha hecho un usuario en específico, más concretamente, el usuario con el username *oscar*.

1. **SELECT** titulo **FROM** usuario_receta
2. **JOIN** receta
3. **ON** usuario_receta.receta_id = receta.receta_id
4. **WHERE** username = 'oscar';

Esta consulta determina los amigos de una persona en concreto, en este caso se trata de los amigos de *juan*, nuevamente una consulta muy concurrida en nuestra base de datos.

1. **SELECT ***
2. **FROM amigo**
3. **WHERE username1 = 'juan' OR username2 = 'juan';**

La siguiente consulta muestra todas las recetas que están asociadas a una categoría en concreto. Es una de las consultas que más se repiten en la plataforma por ejemplo para filtrar en un buscador. Hay que resaltar que se busca por el identificador de la categoría, creemos que es más conveniente en caso de estar utilizando índices, buscar por un tipo de datos numérico en vez de uno cualitativo.

1. **SELECT receta.receta_id, receta.titulo, receta.instrucciones, receta.tiempo, receta.raciones, receta.dificultad, receta.categoría_id, categoría.nombre AS categoría**
2. **FROM categoría**
3. **JOIN receta ON categoría.categoría_id = receta.categoría_id**
4. **WHERE categoría.categoría_id = 8**
5. **GROUP BY receta.receta_id, categoría.nombre**
6. **ORDER BY receta.receta_id DESC;**

La siguiente consulta muestra los ingredientes cuya unidad de medida sea gramos.

1. **SELECT ***
2. **FROM ingrediente**
3. **WHERE unidad_medida = 'g';**

La consulta de a continuación muestra el top 5 de los ingredientes que más se repiten en las recetas.

1. **SELECT ingrediente_id, COUNT(ingrediente_id) AS cantidad**
2. **FROM receta_ingrediente**
3. **GROUP BY ingrediente_id**
4. **ORDER BY cantidad DESC**
5. **LIMIT 5;**

La consulta que se realiza muestra aquellos patrocinadores ordenados de mayor a menor atendiendo a la cantidad total que han donado a la plataforma.

1. **SELECT patrocinadores.patrocinador_id, patrocinadores.cantidad_donada**
2. **FROM (SELECT patrocinador_id, SUM(cantidad_donada) AS cantidad_donada**
3. **FROM patrocinador**
4. **GROUP BY patrocinador_id) AS patrocinadores**
5. **ORDER BY patrocinadores.cantidad_donada DESC;**

La siguiente consulta devuelve la información referente a los comentarios cuyo contenido tiene más de cien caracteres. Esto también nos puede servir para determinar aquellos usuarios que realmente se toman el tiempo de escribir un comentario.

- ```
1. SELECT *
2. FROM comentario
3. WHERE length(contenido) > 100;
```

La consulta que se presenta ahora determina aquellos comentarios que contienen alguna palabra de las siguientes: "rico", "delicioso", "sabroso", "bueno", "recomendable".

- ```
1. SELECT *
2. FROM comentario
3. WHERE contenido LIKE '%rico%' OR
4. contenido LIKE '%delicioso%' OR
5. contenido LIKE '%sabroso%' OR
6. contenido LIKE '%bueno%' OR
7. contenido LIKE '%recomendable%';
```

La siguiente consulta representa los usuarios que más interacciones han realizado en la plataforma, es decir, la suma de comentarios y puntuaciones.

- ```
1. SELECT username, COUNT(*) AS total
2. FROM interaccion
3. GROUP BY username
4. ORDER BY total DESC
5. LIMIT 10;
```

La consulta de a continuación permite determinar el número de comentarios que realizó una persona en concreto sobre una receta.

- ```
1. SELECT COUNT(*)
2. FROM comentario
3. JOIN interaccion
4. ON comentario.comentario_id = interaccion.comentario_id
5. WHERE username = 'ana';
```

La consulta de a continuación permite determinar el número de puntuaciones que agregó una persona en concreto a una receta.

- ```
1. SELECT COUNT(*)
2. FROM puntuacion
3. JOIN interaccion
4. ON puntuacion.puntuacion_id = interaccion.puntuacion_id
5. WHERE username = 'ana';
```