The code so far is a basic CRUD (Create, Read, Update, Delete) API built using the Gin framework, which is a web framework for building APIs in Go. It connects to a SQLite database and allows you to perform basic database operations on a "users" table through a set of RESTful routes. You have also connected the code to Cloudinary, which is a cloud-based image and video management service.

Concretely, we have:

- Connected to a SQLite database
- Defined a User struct with GORM fields
- Created endpoints for CRUD operations on the User resource
- Created functions for handling the routing and database operations for each endpoint
- Discussed some error handling
- provided example of commands to test the endpoints using curl and postman

the code is connected to Cloudinary using the information from the link below:

https://dev.to/hackmamba/robust-media-upload-with-golang-and-cloudinary-gin-gonic-version-54ii

I added the Cloudinary library and used the provided API key and secret to connect the application to my Cloudinary account. The necessary routes and logic to handle file uploads and retrievals is also implemented.

Sample commands to input through the command-line (Windows):

```
curl -X POST -H "Content-Type: application/json" -d "{\"name\":\"Manuel
Cortes\",\"email\":\"manuel.cortes@example.com\"}"
http://localhost:8000/users

curl -X PUT -H "Content-Type: application/json" -d "{\"name\":\"Manuel
Cortes\",\"email\":\"manuel.cortes@example.com\"}"
http://localhost:8000/users/1

curl -X GET http://localhost:8000/users/1

curl -X DELETE http://localhost:8000/users/1
```

The front-end team will need to know the following information for each endpoint in order to properly write the front-end code:

- The endpoint's URL
- The request method (e.g. GET, POST, PUT, DELETE)
- The request parameters (if any) and their data types
- The request body (if any) and the expected format of the data in the body
- The response format, including the HTTP status code and any data that will be returned in the response body.
- The purpose of the endpoint and what it does in the overall functionality of the application.

Description of API endpoints:

1. GET /users: Retrieves a list of all users in the database.
2. GET /users/:id: Retrieves a specific user by their id.
3. POST /users: Creates a new user in the database. The request body should include the following fields: name, email, password.
4. PUT /users/:id: Updates an existing user in the database by their id. The request body should include the fields that you want to update.
5. DELETE /users/:id: Deletes a specific user by their id.
6. POST /file: Uploads an image file to the server and stores it on Cloudinary.
7. POST /remote: Uploads an image file to the server and stores it on Cloudinary by providing a URL.

Longer Description:

Endpoint: `GET /users`

- Description: Retrieves a list of all users in the database.
- Request: None
- Response:
  - 200 OK: Returns a JSON array of all users in the database. Each user is represented as a JSON object with the following properties: id (integer), name (string), email (string), and created_at (string).

```
[
{
"id": 1,
"name": "John Doe",
"email": "johndoe@example.com"
},
{
"id": 2,
"name": "Jane Smith",
"email": "janesmith@example.com"
},
...
]
```

Endpoint: `GET /users/:id`

- Description: GET /users/:id: Retrieves a specific user by their id.
- Request:
  - Parameter: The id of the user to retrieve.
- Response:
  - 200 OK: The user with the corresponding id in the following format:

```
{
"id": 1,
"name": "John Doe",
"email": "johndoe@example.com"
}
```

  - 404 Not Found: If there is no user with the given id in the database

Endpoint: `POST /users`

- Description: Creates a new user in the database.
- Request
  - Body: A JSON object containing the fields name, email, and imgurl.

-
  ```
  {
  "name": "John Doe",
  "email": "johndoe@example.com"
  }
  ```
- Response: A JSON object containing the fields id, name, email, and created_at.
  - 201 Created: Returns a JSON object representing the newly created user. The object has the following properties: id (integer), name (string), email (string), and created_at (string).

-
  ```
  {
  "id": 1,
  "name": "John Doe",
  "email": "johndoe@example.com"
  }
  ```

Endpoint: `PUT /users/:id`

- Description: Updates an existing user in the database. Similar to Post /Users
- Request:
  - Parameter: The id of the user to retrieve.
  - Body: A JSON object containing the fields name, email, and password.
- Response:
  - 200 OK: Returns a JSON object representing the updated user. The object has the following properties: id (integer), name (string), email (string), and created_at (string).
  - 404 Not Found: If no user exists with the specified id.

Endpoint: `DELETE /users/:id`

- Description: Deletes a specific user by their id.
- Request:
  - Parameters: id: the id of the user to delete.
- Response:
  - 200 OK: Returns a JSON object with the message "user deleted"
  - 404 Not Found: If no user exists with the specified id.

Endpoint: `POST /file`

- Description: Uploads an image to cloudinary.
- Request:
  - Body: Multipart/Form-data with a key-value pair where the key is "file" and the value is the file to upload.

```
{
"file": <file data>
}
```

- Response:
  - 200 OK: JSON object containing the url of the uploaded image on Cloudinary

Endpoint: `POST /remote`

- Description: Uploads a file from a remote url to the server
- Request:
  - Body: JSON object with the following properties: url (string).

```
{
"url": "<remote file url>"
}
```

- Response:
  - 200 OK: Returns a JSON object with the url of the uploaded file.

Note: All the endpoints are protected by the JWT token mechanism, so you need to send the token in the header of the request