The flowchart is broken down into several stages:

1. **User Registration and Verification**
2. **User Login and JWT Token Retrieval**
3. **Accessing Protected API Endpoints Using API Gateway and Lambda**

┌─────────────────────────────────────────────────────────────┐

│ User Interface │

└─────────────────────────────────────────────────────────────┘

│

┌───────────────────▼────────────────────┐

│ Sign Up Page (Frontend) │

└────────────────────────────────────────┘

│

│ User submits email, username, and password

▼

┌────────────────────────────────────────┐

│ Signup API Endpoint │

│ (Express.js + AWS Cognito) │

└────────────────────────────────────────┘

│

│ Stores user info and sends a verification code to user's email

▼

┌──────────────────────────────────────┐

│ AWS Cognito (User Pool) │

│ - Stores user details │

│ - Sends verification code │

└──────────────────────────────────────┘

│

│

▼

┌────────────────────────────────────────┐

│ Verify Code Page (Frontend) │

└────────────────────────────────────────┘

│

│ User enters verification code received in email

▼

┌────────────────────────────────────────┐

│ Verification API Endpoint │

│ (Express.js + AWS Cognito) │

└────────────────────────────────────────┘

│

│ Confirms user in Cognito

▼

┌──────────────────────────────────────┐

│ AWS Cognito (User Pool) │

│ - Verifies and activates user │

└──────────────────────────────────────┘

│

│

▼

┌────────────────────────────────────────┐

│ Redirect to Login Page │

└────────────────────────────────────────┘

│

│ User submits email and password for login

▼

┌──────────────────────────────────────┐

│ Login API Endpoint │

│ (Express.js + AWS Cognito) │

└──────────────────────────────────────┘

│

│ Authenticates user and issues JWT token

▼

┌──────────────────────────────────────┐

│ AWS Cognito (User Pool) │

│ - Issues JWT Token │

└──────────────────────────────────────┘

│

▼

┌────────────────────────────────────────┐

│ Store JWT in Local Storage │

│ (Frontend Application) │

└────────────────────────────────────────┘

│

│

▼

┌──────────────────────────────────────────────────────────────────┐

│ Access Protected API Endpoint │

│ (Using API Gateway + AWS Lambda) │

└──────────────────────────────────────────────────────────────────┘

│

│ User includes JWT token in the "Authorization" header

▼

┌──────────────────────────────────────┐

│ API Gateway │

│ - Receives request │

│ - Validates JWT with Cognito │

└──────────────────────────────────────┘

│

│

▼

┌──────────────────────────────────────┐

│ AWS Lambda Function │

│ - Processes request │

│ - Retrieves/updates data │

└──────────────────────────────────────┘

│

│

▼

┌──────────────────────────────────────────────────────────────────┐

│ Return Data to User (Frontend Application) │

└──────────────────────────────────────────────────────────────────┘

**Process Outline and Key Steps**

**Step 1: Set up Cognito for User Authentication**

1. **Created a Cognito User Pool**:
   * Defined user attributes such as username, email, and sub (Cognito’s unique user identifier).
   * Set up email verification for users.
2. **Post Confirmation Lambda Trigger**:
   * Configured a Lambda function as a post-confirmation trigger in Cognito.
   * Purpose: To save user metadata in DynamoDB once the user is confirmed.

**Step 2: Configure API Gateway**

1. **Set Up API Gateway**:
   * Created routes for user signup, verification, and login.
   * Enabled CORS to allow access from the frontend (localhost:3000).
2. **Integration with Lambda**:
   * Integrated each route in API Gateway with a corresponding Lambda function.
   * Configured appropriate HTTP methods (POST for signup, login, etc.).

**Step 3: Set up DynamoDB for User Metadata Storage**

1. **Created a DynamoDB Table**:
   * Defined UserID as the partition key to store unique user data.
   * Stored username, email, and other relevant user attributes.
2. **Configured Lambda Permissions**:
   * Added dynamodb:PutItem permissions to Lambda’s IAM role to allow it to write to the DynamoDB table.

**Step 4: Store Media Files in S3**

1. **Configured S3 Buckets**:
   * Created S3 buckets for storing movies, posters, and trailers.
   * Set bucket policies to allow controlled access to media files.
2. **Media Fetch Lambda Functions**:
   * Created Lambda functions to interact with S3 and retrieve media data as needed by the frontend.

**Step 5: Frontend Implementation**

1. **Developed React Components**:
   * Created signup, login, and verification components.
   * Configured Axios requests to interact with the API Gateway endpoints.
2. **Store JWT for User Sessions**:
   * Configured the login flow to store JWT tokens in local storage to manage sessions.

**Flowchart Representation**

Here’s a description of the flowchart that can be used to visualize this process:

1. **User Signup on Frontend**:
   * User enters signup details ➔ API Gateway ➔ Lambda (signup function) ➔ Cognito ➔ Sends confirmation code to user email.
2. **User Confirmation and Metadata Storage**:
   * User confirms code ➔ API Gateway ➔ Lambda (verification function) ➔ Cognito triggers Post Confirmation Lambda ➔ Lambda stores user metadata in DynamoDB.
3. **User Login**:
   * User logs in ➔ API Gateway ➔ Lambda (login function) ➔ Cognito authenticates ➔ Returns JWT ➔ Frontend stores JWT for session.
4. **Media Retrieval**:
   * User requests media ➔ API Gateway ➔ Lambda fetches media from S3 ➔ Returns data to frontend.

**Problems and Errors Encountered**

**Cognito Errors**

1. **Username Cannot Be in Email Format**:
   * This error occurred because Cognito was configured to not accept email formats as usernames.
2. **Preferred Username Error**:
   * Error message about Preferred username configuration; fixed by adjusting Cognito settings to allow usernames as preferred identifiers.

**Lambda Errors**

1. **require Not Defined Error**:
   * The Lambda function was using require syntax, which caused errors due to ES module settings. Switched to import syntax.
2. **DynamoDB Permission Denied**:
   * Encountered AccessDeniedException when Lambda tried to access DynamoDB due to insufficient permissions. Resolved by adding dynamodb:PutItem to Lambda’s IAM role.

**API Gateway Errors**

1. **CORS Policy Error**:
   * Initial CORS setup issues prevented API Gateway from accepting requests from localhost:3000. Resolved by configuring CORS settings in API Gateway.
2. **Invalid Lambda Response**:
   * Error messages like Unrecognizable Lambda output appeared due to incorrect return structure in the Lambda functions. Fixed by adjusting the JSON structure in Lambda responses.

**S3 Errors**

1. **Access Denied**:
   * Configuring access permissions for the S3 bucket was challenging, especially for allowing the correct policies to permit frontend access to media files.

**General Debugging Issues**

1. **Axios Network Errors**:
   * Encountered Network Error issues in Axios due to API Gateway settings and CORS misconfigurations. Resolved by correctly configuring API Gateway endpoints and enabling CORS.
2. **Lambda Console Testing Issues**:
   * Faced challenges with simulating real events in the Lambda console. This required creating custom test events with attributes similar to actual requests from API Gateway and Cognito.

**Next Steps and Recommendations**

1. **Automation**:
   * Consider using AWS CloudFormation or the AWS CDK to automate the setup of these resources in a repeatable manner.
2. **Error Handling**:
   * Add more comprehensive error handling on both frontend and backend to capture and display more specific error messages.
3. **Monitoring**:
   * Enable CloudWatch monitoring for all Lambda functions and API Gateway endpoints to track performance and debug issues in production.

It sounds like you're looking for a classic flowchart with arrows that indicate each step's direction and flow, similar to the examples you've shown. I'll outline a detailed flowchart that visually represents the serverless architecture workflow you've been implementing. Here's a structured step-by-step flow with labeled arrows and explanations for each step.

Since I can't directly draw it, you can follow this guide to create the flowchart, or you can use tools like Lucidchart, Draw.io, or even PowerPoint for a clean, professional look.

**Serverless Application Flowchart Outline**

**1. Client Interaction**

* **Step 1**: **Start**
* **Arrow**: User initiates an action from the web browser.

**2. API Gateway Routes**

* **Step 2**: **API Gateway**
  + Routes requests based on the endpoint.
* **Arrows**:
  + **/login** (POST): API Gateway → **Auth Lambda** for login.
  + **/signup** (POST): API Gateway → **Auth Lambda** for signup.
  + **/movies/getMovies** (GET): API Gateway → **Get Movies Lambda**
  + **/movies/getMovieDetails/{id}** (GET): API Gateway → **Get Movie Details Lambda** with **Movie ID** parameter.
  + **/movies/getMovieTrailers/{id}** (GET): API Gateway → **Get Movie Trailers Lambda** with **Movie ID** parameter.
  + **/search/history** (GET): API Gateway → **Search History Lambda**.
  + **/search/history/{id}** (DELETE): API Gateway → **Delete History Lambda** with **Item ID**.

**3. Lambda Functions**

* **Step 3**: **Authentication Lambda (Login, Signup)**
  + **Actions**: Handles user authentication using Cognito for login and signup.
  + **Arrow**: Auth Lambda → **Cognito** for token validation and user management.
  + **Response**: Returns **JWT token** and success message back to the client through API Gateway.
* **Step 4**: **Get Movies Lambda**
  + **Actions**: Retrieves a list of movies.
  + **Arrow**: Get Movies Lambda → **DynamoDB** to query movie data.
  + **Response**: Sends movie list back to the client through API Gateway.
* **Step 5**: **Get Movie Details Lambda**
  + **Actions**: Retrieves detailed information about a specific movie.
  + **Parameter**: Accepts **Movie ID** from the API Gateway request.
  + **Arrow**: Get Movie Details Lambda → **DynamoDB** to retrieve specific movie details.
  + **Response**: Returns movie details to the client through API Gateway.
* **Step 6**: **Get Movie Trailers Lambda**
  + **Actions**: Retrieves trailers for a specific movie.
  + **Parameter**: Accepts **Movie ID**.
  + **Arrow**: Get Movie Trailers Lambda → **DynamoDB** to fetch trailer links.
  + **Response**: Sends trailer URLs back to the client.
* **Step 7**: **Search History Lambda**
  + **Actions**: Retrieves or deletes user search history.
  + **Arrows**:
    - Retrieve History → DynamoDB → **Client**
    - Delete Item (with ID) → **DynamoDB** for deletion → **Client**
* **Step 8**: **Track Last Pause Position**
  + **Description**: Lambda function saves video playback progress.
  + **Arrows**: Save current position → **DynamoDB**.
  + **Response**: Fetches video from the saved point next time.

**4. Database (DynamoDB)**

* **Actions**: Stores user data, search history, movies, trailers, and playback position.
* **Arrows**: Lambda Functions interact with DynamoDB for storing and retrieving data.

**5. Cognito (For Authentication)**

* **Description**: Authenticates users for login and signup.
* **Arrows**:
  + Login Request → Cognito (Auth token returned).
  + Signup Request → Cognito (Account created and verified).

**6. Return Path**

* **Step 9**: **API Gateway → Client**
  + After each Lambda function finishes, the response goes back through API Gateway.
  + **Final Arrows**: Each Lambda Function → API Gateway → Client

With this structure, you can draw boxes for each service (e.g., Client, API Gateway, Lambda Functions, DynamoDB, Cognito) and connect them with arrows labeled according to the actions and parameters described above. This flowchart will illustrate both the data flow and the function of each service within your serverless application.

Let me know if you'd like me to clarify any specific step or interaction!