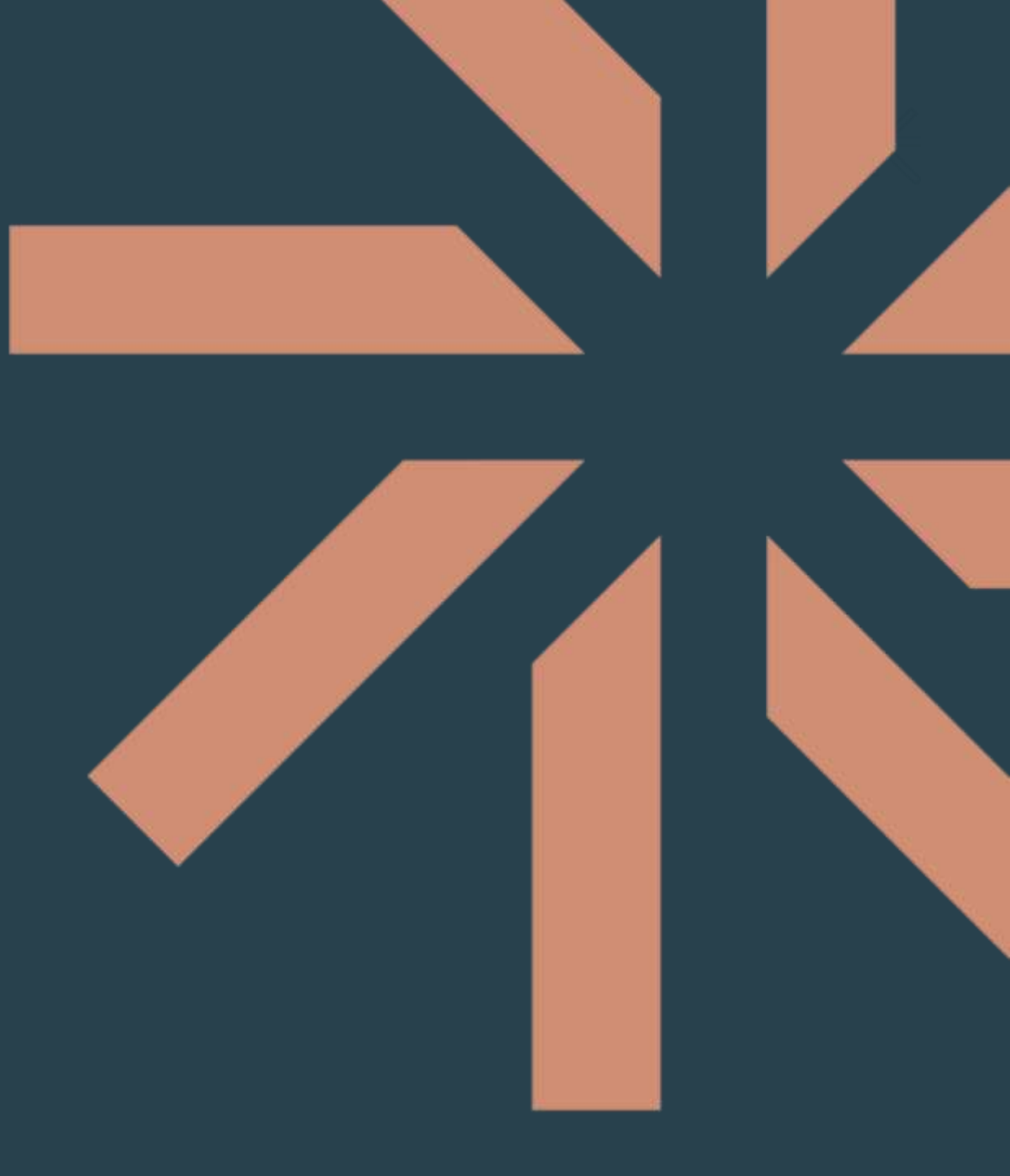
Authentication

Qinshift Academy

Authentication using Sessions

Session-based authentication is a classic method where a user's login credentials are used to authenticate once, and the server then creates a session that gets stored (usually in memory, Redis, or a DB). A session ID is sent back to the client in a cookie and used to identify the user on future requests.



How It Works – Step by Step

1. User Logs In

- Client sends a POST request to /login with username/email and password.
- Server verifies credentials (usually by checking the database).

2. Server Creates a Session

- If valid, the server creates a session and stores user info on the server (e.g., userId, role, etc.).
- It assigns a unique session ID to the session.

3. Session ID Sent to Client

- The session ID is sent back to the client as a secure HTTP-only cookie.

Set-Cookie: sessionId=abc123; HttpOnly; Secure;

4. Client Sends Session Cookie

- On every future request, the browser automatically sends the cookie back to the server.

Cookie: sessionId=abc123

5. Server Verifies the Session

- The server uses the session ID to find the session and determine the user's identity.
- If the session is valid, the request proceeds as an authenticated user.

🔐 Benefits of Session-Based Authentication

- Simple and widely supported
- No need to store tokens client-side
- Easy to implement role-based access
- Compatible with traditional server-rendered apps

⚠ Common Considerations

Concern	Solution
Memory leaks	Use external session store (e.g., Redis)
Cross-site attacks	Use HttpOnly, Secure, SameSite cookies
Horizontal scaling	Share session store across servers
Session expiration	Use cookie.maxAge or server TTL

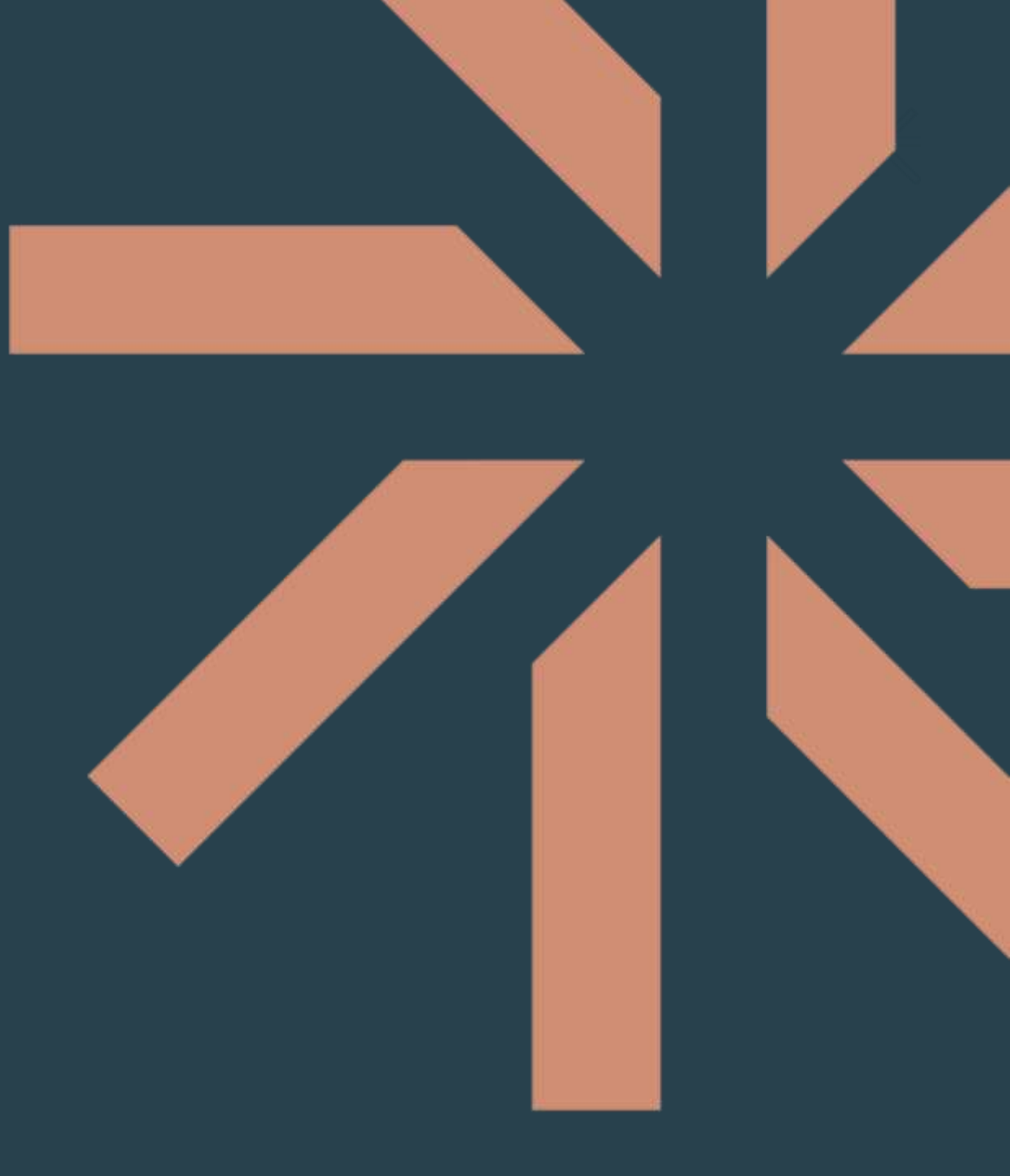
Summary

- Sessions store user data on the server
- A cookie with session ID identifies the session
- On each request, the server restores the user context
- Easy to implement, ideal for small to medium-sized apps

JWT Authentication

JWT (JSON Web Token) is a compact, self-contained way to represent user identity and claims securely.

It is commonly used for stateless authentication, where the server doesn’t store any session data.



How JWT Auth Works (No Refresh Tokens)

1. User logs in

- Sends POST /login with email & password

2. Server verifies credentials

- If valid, it creates a JWT access token that includes a payload like:

```
{
  "sub": 42,
  "email": "user@example.com",
  "role": "user",
  "iat": 1680000000,
  "exp": 1680003600
}
```

3. Token is sent back to client

- Usually in JSON response or a cookie (or header for SPA/mobile)

```
{
  "accessToken": "eyJhbGciOiJIUzI1NiIs..."
}
```

4. Client stores token

- In memory, localStorage, or cookie

5. Client sends token in each request

- Usually in Authorization header:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

6. Server verifies token

- If valid and not expired, request is authenticated.
- No need for server-side session storage.

Pros

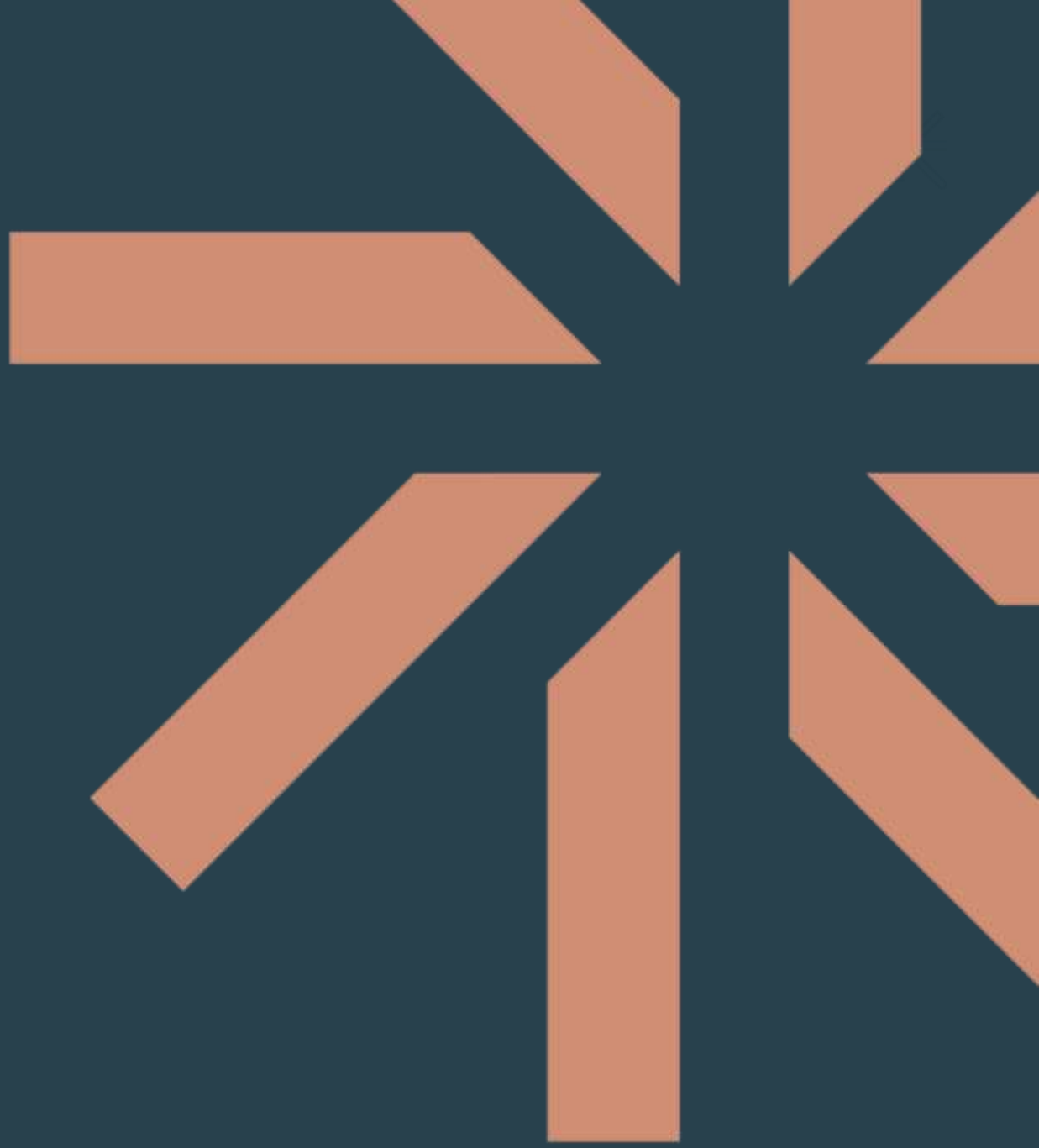
- Stateless and scalable (good for microservices & REST)
- Easy to use across mobile/web clients
- Tokens can include user roles or permissions

⚠ Cons

- When token expires, user must log in again
- No way to invalidate a token unless you maintain a blacklist
- Not ideal for sensitive systems without extra security layers

JWT Authentication with Refresh Tokens

To solve the problem of short-lived access tokens, we introduce refresh tokens.



Two Tokens Strategy

Token	Purpose	Lifetime
Access Token	Sent with each request	Short (e.g., 15m)
Refresh Token	Used to get new access tokens	Long (e.g., 7 days or more)

How It Works

1. User logs in
    - Server returns both tokens:
        - accessToken (short-lived)
        - refreshToken (long-lived)
2. Client stores tokens
    - accessToken is sent with each request
    - refreshToken is stored securely (e.g., HttpOnly cookie
3. When access token expires
    - Client calls POST /refresh-token with refresh token
    - Server verifies refresh token and issues a new access token
4. Logout
    - Server invalidates the refresh token (e.g., removes it from DB)

✓ Benefits of Refresh Tokens

- Improves security by keeping access tokens short-lived
- User stays logged in without having to re-enter credentials
- Tokens can be rotated for better protection (rotate on refresh)

⚠ Risks and Considerations

- Refresh tokens must be stored securely (prefer HttpOnly cookies)
- Should implement refresh token rotation and revocation
- Store and track refresh tokens in DB (or Redis) to invalidate on logout or compromise

Summary

Auth Style	Stateless	Needs DB	Scalable	Re-login Needed
JWT only	✓	✗	✓	✓ (on expiry)
JWT + RefreshToken	✓	✓	✓	✗ (token refresh)

Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

asistant@mail.com

Qinshift Academy