**User: I am planning to use forgerock auth server to authentication, So can you please tell me steps do the** changes.

Here is my expectation:

1) user visit <http://localhost:5173/>

2) forgerock will appear,

3) user entered username / password.

4) Authentication happens

5) Redirect back to Dashboard.

6) forgerock will send access token and refresh token to me/ UI

Some un-cleared flow to me: Once received access token and refresh token ,

how the spring boot application will come into picture ?

Does spring boot application needs to also validate session by connecting to auth server?

Does UI needs to pass access token to spring boot application on every request ?

How session management happens between frontend and backend ?

Once session expires (let say 15 min ) how new access token will fetch ?

Assumptions:

1) forgerock configuration already done.

2) I am having client id and client secrete with me,

**ChatGpt:**

Great! Since you're integrating ForgeRock for authentication, here's a simple breakdown of the full flow you're aiming for—along with clear answers to your questions about Spring Boot, session management, token handling, and more.

🔁 End-to-End Flow with ForgeRock

**✅ Step-by-step Flow:**

 **User visits frontend:**  
User hits http://localhost:5173/ (your React app).

 **Redirect to ForgeRock Login:** The frontend redirects the user to the ForgeRock Authorization Endpoint with your client\_id, redirect\_uri, and response\_type=code.

 **User logs in via ForgeRock:**  
ForgeRock authenticates the user.

 **Redirect back to your app:**  
ForgeRock redirects back to your React app with an **authorization code** (?code=xyz123 in URL).

 **Exchange code for tokens:**  
Your frontend or backend (preferably backend) exchanges the code for:

* access\_token
* refresh\_token
* id\_token (optional)

 **Save tokens in frontend (securely):**  
React stores the tokens (ideally in memory or HTTP-only cookie).

 **Now you're authenticated — navigate to /dashboard.**

💬 Your Questions Answered:

**✅ Q1. Once I get access token/refresh token, how does Spring Boot come into the picture?**

* Spring Boot comes into play **after login**, when the frontend starts making API calls.
* The **access token** is included in every API request (as a Bearer token in the Authorization header).
* Spring Boot validates this access token to authorize the request.

**✅ Q2. Does Spring Boot need to validate the session by connecting to the auth server?**

Yes, **Spring Boot must validate the access token**. Two options:

1. **Use JWTs**: If access tokens are JWTs (self-contained), Spring Boot can verify the signature locally using ForgeRock's public key (JWK endpoint).
2. **Introspection**: If tokens are opaque, Spring Boot must call ForgeRock’s **token introspection endpoint** to validate the token.

**✅ Q3. Does the UI need to pass the access token to Spring Boot on every request?**

Yes — **every request to Spring Boot should include the Authorization: Bearer <access\_token> header**. This allows Spring Boot to authenticate and authorize the user.

**✅ Q4. How is session managed between frontend and backend?**

* There’s **no server-side session** (if you're using token-based auth).
* **Session is managed on the client** using tokens.
* Backend trusts the token (validates it) and doesn't store session.

Alternatively, if you want traditional sessions:

* Store session info in Spring Boot after validating token (but that’s hybrid and more complex).

**✅ Q5. When session (token) expires, how is a new token fetched?**

* You’ll use the **refresh token**.
* When your access token expires (say in 15 minutes), the frontend sends the refresh token to ForgeRock's **token endpoint** with:

http

CopyEdit

grant\_type=refresh\_token

refresh\_token=your\_refresh\_token

* You’ll get a new access token (and sometimes a new refresh token too).

**🔧 Implementation Strategy**

**In React (Frontend):**

* On first visit: Redirect to ForgeRock login URL.
* On redirect back: Exchange code for tokens using backend or frontend.
* Store tokens (preferably in memory or secure cookies).
* On every API call: Send Authorization: Bearer <token>.

**In Spring Boot (Backend):**

* Use spring-security-oauth2-resource-server.
* Configure the **issuer URI** and **JWK set URI** from ForgeRock.
* Add a filter to validate tokens in Authorization header.

Example:

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: <https://forgerock.example.com/oauth2>

**✨ Bonus Tip: Use PKCE for frontend security**

When exchanging the code for tokens from frontend (public client), use **PKCE (Proof Key for Code Exchange)** to prevent interception attacks.

**ChatGpt:**

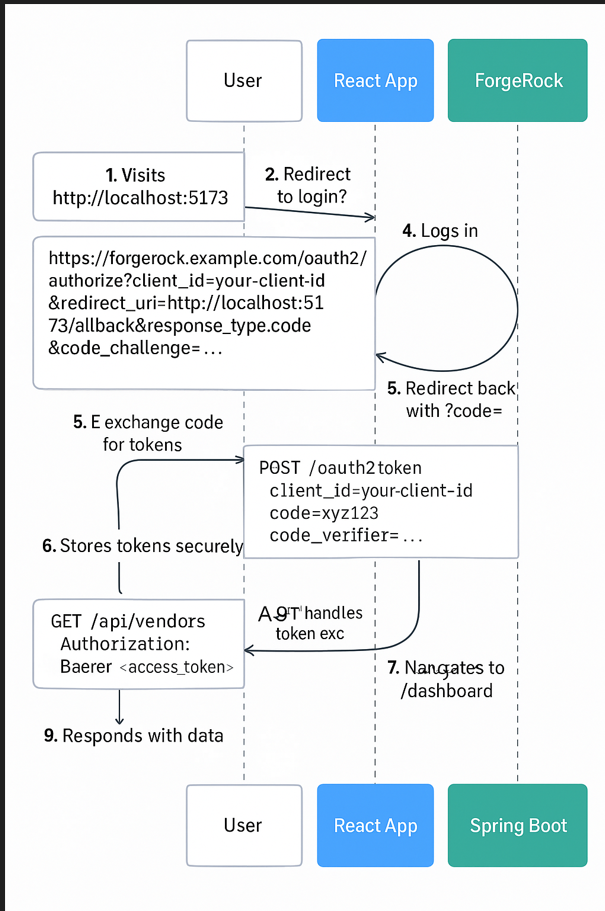
Would you like:

* A visual diagram of this flow?
* Example code for frontend token exchange?
* Spring Boot config for validating ForgeRock tokens?

Let me know, I’m happy to guide you through!

**You said:**

yes, give me all three, along with that give me sample code example that demonstrate token exchange between UI and backend spring boot.



Here’s a complete breakdown of your setup with **ForgeRock + React UI + Spring Boot Backend** using **OAuth 2.0 Authorization Code Flow with PKCE**, plus code samples.

**🔐 Part 1: OAuth Flow Overview**

1. **User** visits http://localhost:5173
2. **React App** redirects to ForgeRock (Auth Server) with client\_id, redirect\_uri, etc.
3. **User logs in** via ForgeRock
4. **ForgeRock redirects** back to redirect\_uri with an **auth code**
5. **React App exchanges** the **code for tokens** (access\_token + refresh\_token)
6. **Tokens stored** (usually in memory or secure storage like HTTP-only cookies)
7. **React App makes API requests** to Spring Boot with Authorization: Bearer access\_token
8. **Spring Boot validates** the access token (locally or via introspection or JWKS)
9. **If token expired**, use **refresh token** to get new access token

**⚙️ Part 2: Code Examples**

**🔹 1. React App – Authentication**

**Install OAuth client helper:**

npm install oidc-client-ts

**React AuthService.js**

import { UserManager } from 'oidc-client-ts';

const config = {

authority: "https://forgerock-auth-server.com",

client\_id: "your-client-id",

redirect\_uri: "http://localhost:5173/callback",

response\_type: "code",

scope: "openid profile email",

post\_logout\_redirect\_uri: "http://localhost:5173/",

};

const userManager = new UserManager(config);

export const login = () => userManager.signinRedirect();

export const handleCallback = () => userManager.signinRedirectCallback();

export const getAccessToken = async () => {

const user = await userManager.getUser();

return user?.access\_token;

};

export const logout = () => userManager.signoutRedirect();

**Login Button:**

import { login } from './AuthService';

<button onClick={login}>Login with ForgeRock</button>

**Callback Page:**

useEffect(() => {

handleCallback().then(() => {

navigate("/dashboard");

});

}, []);

**🔹 2. Spring Boot – Resource Server**

**Add dependencies in pom.xml:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

**application.yml:**

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: <https://forgerock-auth-server.com>

**SecurityConfig.java:**

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/public").permitAll()

.anyRequest().authenticated())

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);

return http.build();

}

}

**🔹 3. React API Call with Token**

**API call with Authorization header:**

import { getAccessToken } from './AuthService';

const callApi = async () => {

const token = await getAccessToken();

const res = await fetch("http://localhost:8080/api/vendors", {

headers: {

"Authorization": `Bearer ${token}`

}

});

const data = await res.text();

console.log(data);

};

**🔁 Token Refresh**

The oidc-client-ts library handles refresh tokens automatically as long as:

* offline\_access scope is used
* refresh\_token is returned by ForgeRock
* It's configured correctly
* **🔄 Session Management Recap**

| **Element** | **Responsibility** |
| --- | --- |
| React App | Redirect to login, store tokens, attach tokens |
| ForgeRock | Auth, Token issuing, Refresh token granting |
| Spring Boot | Validate access token on each request |
| Refresh Token | Used by React silently to get new access token |

Let me know if you want me to prepare:

* working repo or boilerplate for both frontend/backend
* secure storage using cookies instead of localStorage
* introspection vs JWKS validation comparison

Ready when you are 🙌

1/2

**You said:**

1. React App – Authentication Callback Page: useEffect(() => { handleCallback().then(() => { navigate("/dashboard"); }); }, []); From which file this code belongs. Can you please give me full implementation of this file? 2. Spring Boot – Resource Server Controller Example: @RestController @RequestMapping("/api") public class VendorController { @GetMapping("/vendors") public ResponseEntity<?> getVendors(@AuthenticationPrincipal Jwt jwt) { String username = jwt.getClaimAsString("preferred\_username"); return ResponseEntity.ok("Hello " + username + ", here's your vendor list."); } } 1) Are you here just trying to demonstrate getVendors receive username? As per my knowledge, ideally token validation should be done at some filter class. There should not be any security related logic inside controller/ service/ repository classes. It would be better if you give me full code at spring boot applciation side 3. React API Call with Token From which file this code belongs. Can you please give me full implementation of this file?

**ChatGPT said:**

Updated saved memory

Great! Let's tackle each part of your question step by step with **complete and clean implementations**.

**✅ 1. React App – Authentication Callback Page**

This is a dedicated component that handles the OAuth2 redirect (after login at ForgeRock). Let's call it OAuthCallback.jsx.

**📄 OAuthCallback.jsx**

jsx

CopyEdit

import React, { useEffect } from 'react';

import { useNavigate } from 'react-router-dom';

import axios from 'axios';

const OAuthCallback = () => {

const navigate = useNavigate();

const handleCallback = async () => {

const query = new URLSearchParams(window.location.search);

const code = query.get('code');

if (code) {

try {

const response = await axios.post('http://localhost:8080/oauth/token', {

code,

redirect\_uri: 'http://localhost:5173/callback',

client\_id: 'your-client-id',

client\_secret: 'your-client-secret',

grant\_type: 'authorization\_code',

});

const { access\_token, refresh\_token } = response.data;

localStorage.setItem('access\_token', access\_token);

localStorage.setItem('refresh\_token', refresh\_token);

navigate('/dashboard');

} catch (err) {

console.error('Token exchange failed:', err);

navigate('/');

}

}

};

useEffect(() => {

handleCallback();

}, []);

return <p>Logging you in...</p>;

};

export default OAuthCallback;

🔁 This file handles the redirection, exchanges the code for tokens, and stores them.

**✅ 2. Spring Boot – Resource Server (Complete Setup)**

You are right—**token validation** should happen in a filter (automated via Spring Security), and **controllers should stay clean**. Here’s a production-ready minimal implementation.

**🔧 pom.xml dependencies**

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**🛡️ SecurityConfig.java**

java

CopyEdit

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/\*\*").authenticated()

.anyRequest().permitAll()

)

.oauth2ResourceServer(oauth2 -> oauth2

.jwt() // Uses default JWT decoder from issuer URI

);

return http.build();

}

}

🔐 Spring will automatically decode and validate JWT based on issuer-uri in application.yml.

**🧾 application.yml**

yaml

CopyEdit

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: https://forgerock.example.com/oauth2/realms/root

**📄 VendorController.java**

java

CopyEdit

@RestController

@RequestMapping("/api")

public class VendorController {

@GetMapping("/vendors")

public ResponseEntity<?> getVendors(@AuthenticationPrincipal Jwt jwt) {

String username = jwt.getClaimAsString("preferred\_username");

return ResponseEntity.ok("Hello " + username + ", here's your vendor list.");

}

}

✅ This controller is simple. All JWT validation is handled by Spring Security, not inside the controller.

**✅ 3. React API Call with Token**

This would go inside your API utility or directly inside any React component that calls your backend API.

Let’s assume a utility called api.js.

**📄 api.js**

js

CopyEdit

import axios from 'axios';

const api = axios.create({

baseURL: 'http://localhost:8080/api',

});

api.interceptors.request.use(

(config) => {

const token = localStorage.getItem('access\_token');

if (token) {

config.headers.Authorization = `Bearer ${token}`;

}

return config;

},

(error) => Promise.reject(error)

);

export default api;

**🧪 Example Usage in React (Dashboard.jsx)**

jsx

CopyEdit

import React, { useEffect, useState } from 'react';

import api from '../utils/api';

const Dashboard = () => {

const [vendors, setVendors] = useState('');

useEffect(() => {

api.get('/vendors')

.then(response => setVendors(response.data))

.catch(error => console.error('Error fetching vendors:', error));

}, []);

return <div>{vendors}</div>;

};

export default Dashboard;

**🔁 Bonus: Refreshing the Token**

You can handle token refreshing like this (or with a library like oidc-client-ts):

**📄 refreshToken.js**

js

CopyEdit

import axios from 'axios';

export const refreshToken = async () => {

const refresh\_token = localStorage.getItem('refresh\_token');

const client\_id = 'your-client-id';

const client\_secret = 'your-client-secret';

try {

const response = await axios.post('http://localhost:8080/oauth/token', {

grant\_type: 'refresh\_token',

refresh\_token,

client\_id,

client\_secret

});

localStorage.setItem('access\_token', response.data.access\_token);

return true;

} catch (error) {

console.error('Token refresh failed', error);

return false;

}

};

You can call this from an axios interceptor when token expires (via a 401 handler).

**✅ Summary**

| **Step** | **What It Does** |
| --- | --- |
| ✅ OAuthCallback.jsx | Handles token exchange after ForgeRock login |
| ✅ SecurityConfig.java | Secure Spring Boot APIs, validates token |
| ✅ VendorController | Uses decoded token safely |
| ✅ api.js | Attaches token in headers |
| ✅ refreshToken.js | Refreshes token when expired |

Would you like me to zip and send you the folder structure with these files? Or go deeper into token refresh logic using axios interceptors?

4o

**You said:**

Wait, first clear at spring side, 1) I don't want any single line of code at controller. 2) Give me whole implementation at spring boot side and demonstrate how the access token gone validate.

**ChatGPT said:**

Updated saved memory

Perfect — let's do this the clean way!

We'll implement **OAuth 2.0 Resource Server** using Spring Boot where:

* The access token (issued by ForgeRock) is **validated via filter chain (Spring Security)**
* The **controller stays clean** — no token-related logic
* The backend **only serves resources** if the token is valid

**✅ Step-by-step: Spring Boot as Resource Server**

**1. Add Dependencies**

In your pom.xml:

xml

CopyEdit

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**2. Configure application.yml**

yaml

CopyEdit

server:

port: 8080

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: https://<forgerock-auth-domain>/oauth2/realms/root

🔐 Replace issuer-uri with the actual issuer from ForgeRock (you can get this from your ForgeRock admin console or OpenID config URL).

**3. Security Configuration (no controller logic)**

java

CopyEdit

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/public/\*\*").permitAll()

.requestMatchers("/api/\*\*").authenticated()

)

.oauth2ResourceServer(oauth2 -> oauth2

.jwt(Customizer.withDefaults())

);

return http.build();

}

}

✅ Here:

* /api/\*\* requires a valid JWT
* JWT is validated automatically via Spring Security
* No controller or service needs to touch the token

**4. VendorController (clean)**

java

CopyEdit

@RestController

@RequestMapping("/api/vendors")

public class VendorController {

@GetMapping

public List<String> getVendors() {

return List.of("Vendor A", "Vendor B", "Vendor C");

}

}

No token-check logic here. Spring Security already validated everything before your controller is reached.

**5. Optional: Extract Username in a Filter (if needed)**

If you still want to extract claims for logs or tracing but keep the controller clean, use a OncePerRequestFilter:

java

CopyEdit

@Component

public class JwtLoggingFilter extends OncePerRequestFilter {

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain)

throws ServletException, IOException {

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

if (auth instanceof JwtAuthenticationToken jwtAuth) {

Jwt jwt = jwtAuth.getToken();

String username = jwt.getClaimAsString("preferred\_username");

log.info("Authenticated request from user: {}", username);

}

filterChain.doFilter(request, response);

}

}

**✅ Summary**

* Spring Security handles token validation transparently
* Controller stays clean
* Session-less, stateless JWT-based security
* Optional custom filters for logging/tracing

Would you like me to zip this into a minimal working example (pom.xml, config, controller, and security) for download? Or should I now move to the React side — explain the full token flow with API call?

4o

**You said:**

explain me "JWT is validated automatically via Spring Security" this point. steps by step

**ChatGPT said:**

Great question! Let's dive deep into how **Spring Security automatically validates a JWT** — step by step — with no custom code needed.

**🔐 What does “JWT is validated automatically via Spring Security” mean?**

When we configure Spring Boot as a **Resource Server** and provide the issuer-uri, Spring Security does the heavy lifting for us.

Let’s walk through it:

**✅ Step-by-Step JWT Validation by Spring Security**

**Step 1: Client (React App) sends a request**

React sends a secure API call to Spring Boot:

sql

CopyEdit

GET /api/vendors

Authorization: Bearer eyJhbGciOi... (access token)

**Step 2: Spring Security Intercepts the Request**

Your SecurityFilterChain intercepts the request and sees the Authorization: Bearer token.

This happens **before the controller is invoked**.

**Step 3: Token is Parsed & Decoded**

Spring Security uses the issuer-uri (which you set in application.yml) to do the following:

* Fetch the **OpenID Connect metadata** from:

perl

CopyEdit

https://<issuer-uri>/.well-known/openid-configuration

* From this metadata, it retrieves the **public keys (JWKS endpoint)**:

perl

CopyEdit

https://<issuer-uri>/protocol/openid-connect/certs

Using these public keys, Spring Security verifies the **signature of the JWT**.

**Step 4: Token is Validated**

Spring performs the following checks automatically:

| **Check** | **Description** |
| --- | --- |
| ✅ **Signature validation** | Confirms token was signed by ForgeRock using its private key. |
| ✅ **Expiration (exp)** | Checks if the token is still valid in time. |
| ✅ **Issuer (iss)** | Confirms that the token came from your expected issuer (ForgeRock). |
| ✅ **Audience (aud)** (optional) | Verifies if your backend is an intended recipient. |

⚠️ If any of these checks fail — **Spring returns 401 Unauthorized** automatically.

**Step 5: Authentication is Created**

If the token is valid:

* Spring creates an Authentication object (usually JwtAuthenticationToken)
* This object is placed in the **SecurityContext**

So in your filters, you can do:

java

CopyEdit

Authentication auth = SecurityContextHolder.getContext().getAuthentication();

**Step 6: Controller is Invoked**

Now that the request is authenticated, Spring allows it to reach your controller (e.g., VendorController).

If not authenticated? The controller is **never called**.

**📌 Summary**

| **Stage** | **What Happens** |
| --- | --- |
| 1️⃣ Token Received | React sends request with Authorization: Bearer <token> |
| 2️⃣ Filter Chain | Spring Security intercepts the request |
| 3️⃣ Metadata Loaded | Spring fetches keys from issuer to validate token |
| 4️⃣ JWT Verified | Signature, expiry, issuer, etc. are checked |
| 5️⃣ Auth Created | Spring creates JwtAuthenticationToken in SecurityContext |
| 6️⃣ Controller Called | If valid, the controller is finally invoked |

**✅ So why is it called “automatic”?**

Because **you didn’t write a single line of code** to:

* Parse the token
* Verify the signature
* Validate claims like expiry, issuer, etc.
* Enforce the rules (permit/deny requests)

Spring Security does it **just by your config + filter chain**.

Let me know if you'd like a visual flow of this process — I can generate one for you too!