

# Security for Microservices with Spring

<http://localhost:4000/decks/microservice-security.html>

Dave Syer, 2012

Twitter: @david\_syer

Email: [dsyer@pivotal.io](mailto:dsyer@pivotal.io)

(Security for Microservices with Spring)

# Agenda

---

- What is a Microservice?
- How would it be secure?
- If I was going to use Spring how would that look?
- What's the easiest way to get something working?

# Introduction

---

- There is a strong trend in distributed systems with lightweight architectures
- People have started to call them "microservices"



- So what are people doing about security in such systems?

# What is a Microservice?

---

- HTTP transport (usually).
- Text-based message content, usually JSON.
- Small, compact messages, and quick responses.
- REST-ful, or at least inspired by the REST
- Some degree of statelessness
- Interoperability.

# What Are the Security Requirements

---

*Stop bad guys from accessing your resources*

Identity and permissions:

- How is identity and permission information conveyed to a service?
- How is it decoded and interpreted?
- What data are needed to make the access decision (user accounts, roles, ACLs etc.)?
- How is the data managed: who is responsible for storing and retrieving it?
- How can you verify that the request hasn't been tampered with?

# HTTP Basic Authentication

---

- Something of a lowest common denominator
- Supported on practically all servers natively and out of the box
- Ubiquitous support on the client side in all languages
- Good support in Spring Security
- Spring Boot autoconfigures it out of the box

Example:

```
$ curl "https://$username:$password@myhost/resource"
```

# Simple Service

---

```
@Grab('spring-boot-starter-security')
@RestController
class Application {

    @RequestMapping("/")
    def home() {
        [status: 'OK']
    }
}
```

# So what's wrong with that?

---

- Nothing, but...
- Where do you get the credentials (the username and password)?
- Fine for systems where all participants can share secrets securely
- In practice that means small systems
- Only supports username/password
- Only covers authentication
- No distinction between users and machines



# Network Security

---

- Microservice endpoints not visible outside an internal network
- Very secure
- Very flexible (e.g. using virtual networks)
- Suits architecture based on "edge" server

# So what's wrong with that?

---

- Nothing, but...
- Annoying to debug and a little bit fiddly to maintain
- Configuration is out of the control of developers in many cases for organizational reasons
- There's no identity or authentication

# Certificate Based Security

---

- Set up SSL on server so that request has to contain certificate
- Spring Security (X.509) can turn that into an Authentication
- If SSL is in the service layer (i.e. not at the router/loadbalancer) then you have a keystore anyway

Example:

```
$ curl -k --cert rod.pem:password https://localhost:8443/hello
```

# So what's wrong with that?

---

- Nothing, but...
- There's no user identity (only at most the machine) unless browsers have certificates installed
- Requires keystores and certificates in all applications and services (significantly non-trivial if done properly, but some organizations require it anyway)
- No fine distinction between users and machines

# Custom Authentication Token

---

- Random identifier per authentication
- Grant them from a central service and/or store in a central database
- Can be exposed directly via developer UI
- Re-hydrate authentication in service
- Spring Security  
    `AbstractPreAuthenticatedProcessingFilter` and friends
- Even easier: Spring Session identifier
- Github example: token is temporary password in HTTP Basic

# So what's wrong with that?

---

- Nothing, but...
- No separation of client app from user authentication
- Coarse grained authorization (all tokens activate all resources)
- It's not a "standard" (but there are ready made implementations)
- For user authentication, need to collect user credentials in app

# OAuth2 Key Features

---

- Extremely simple for clients
- Access tokens carry information (beyond identity)
- Resources are free to interpret token content

# So what's wrong with that?

---

- Nothing, but...
- No standard (yet) for request signing



# Quick Introduction to OAuth2

---

*A Client application, often web application, acts on behalf of a User, but with the User's approval*

- Authorization Server
- Resource Server
- Client application

Common examples of Authorization Servers and Resource Servers on the internet:

- Facebook - Graph API
- Google - Google APIs
- Cloud Foundry - Cloud Controller

# OAuth2 Bearer Tokens

---

Centralizing account management and permissions:

- OAuth 2.0 adds an extra dimension - more information for the access decision
- Standards always help in security
- Lightweight - easy to curl
- Requires HTTPS for secure operation, but you can test with HTTP

# OAuth2 and the Microservice

---

Example command line Client:

```
$ curl -H "Authorization: Bearer $TOKEN" https://myhost/resource
```

- https://myhost is a Resource Server
- TOKEN is a Bearer Token
- it came from an Authorization Server

# Simple Authorization Server

---

```
@EnableAuthorizationServer
class AuthorizationServer extends
AuthorizationServerConfigurerAdapter {

    @Override
    void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
            .withClient("my-client-with-secret")...
    }
}
```

# Example token contents

---

- Client id
- Resource id (audience)
- Issuer
- User id
- Role assignments

# JWT Bearer Tokens

---

- OAuth 2.0 tokens are opaque to clients
- But they carry important information to Resource Servers
- Example of implementation (from Cloud Foundry UAA, JWT = signed, base64-encoded, JSON):

```
{  "client_id":"cf",
  "exp":1346325625,
  "scope":
  ["cloud_controller.read","openid","password.write"],
  "aud":["openid","cloud_controller","password"],
  "iss": "https://login.run.pivotal.io",
  "user_name":"tester@vmware.com",
  "user_id":"52147673-9d60-4674-a6d9-225b94d7a64e",
  "email":"tester@vmware.com",
  "jti":"f724ae9a-7c6f-41f2-9c4a-526cea84e614" }
```

# JWT Authorization Server

---

```
@EnableAuthorizationServer
class AuthorizationServer extends
AuthorizationServerConfigurerAdapter {

    @Bean
    public JwtAccessTokenConverter accessTokenConverter() {
        return new JwtAccessTokenConverter();
    }

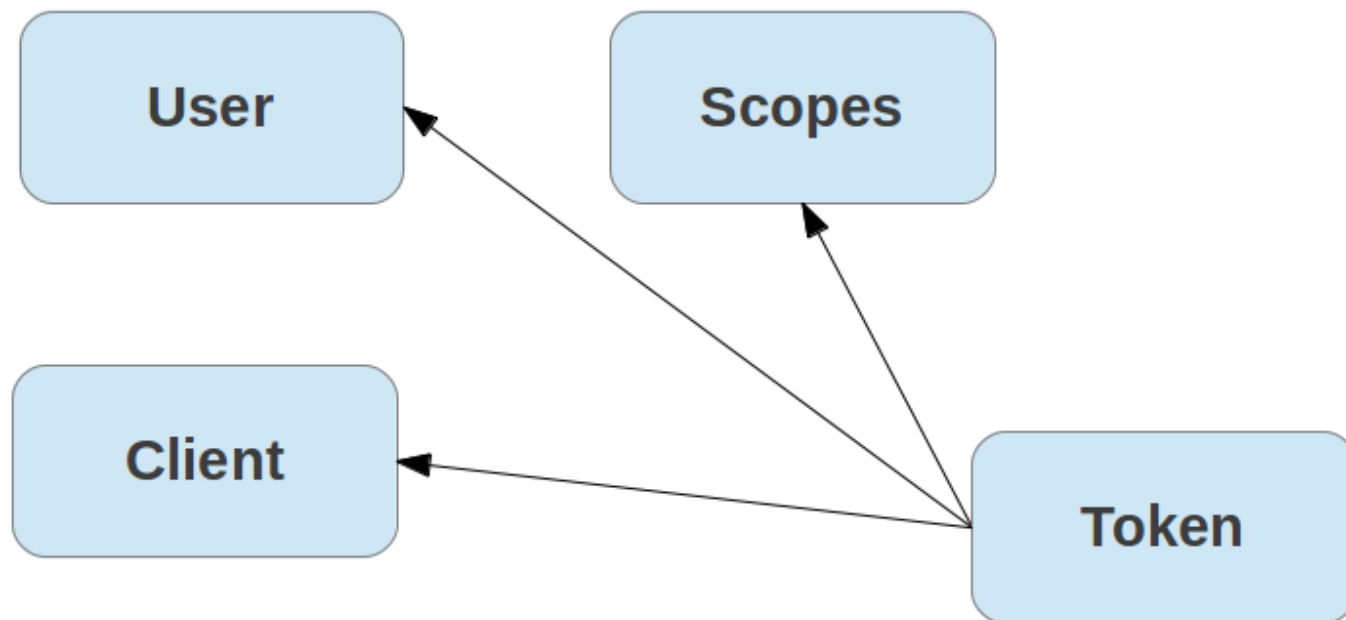
    @Override
    public void configure(AuthorizationServerEndpointsConfigurer
endpoints) throws Exception {
        endpoints.authenticationManager(authenticationManager)
            .accessTokenConverter(accessTokenConverter());
    }

    ... // client config
}
```

# User Approvals

---

An access token represents a user approval:



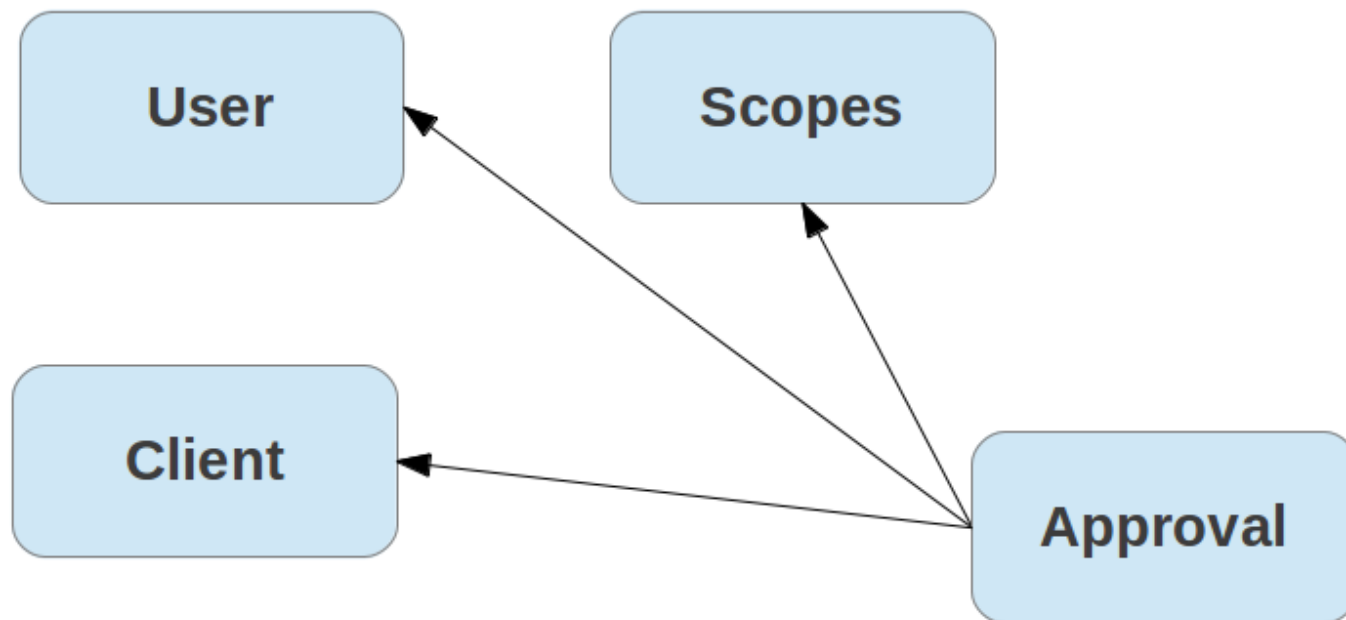




# User Approvals as Token

---

An access token represents a user approval:

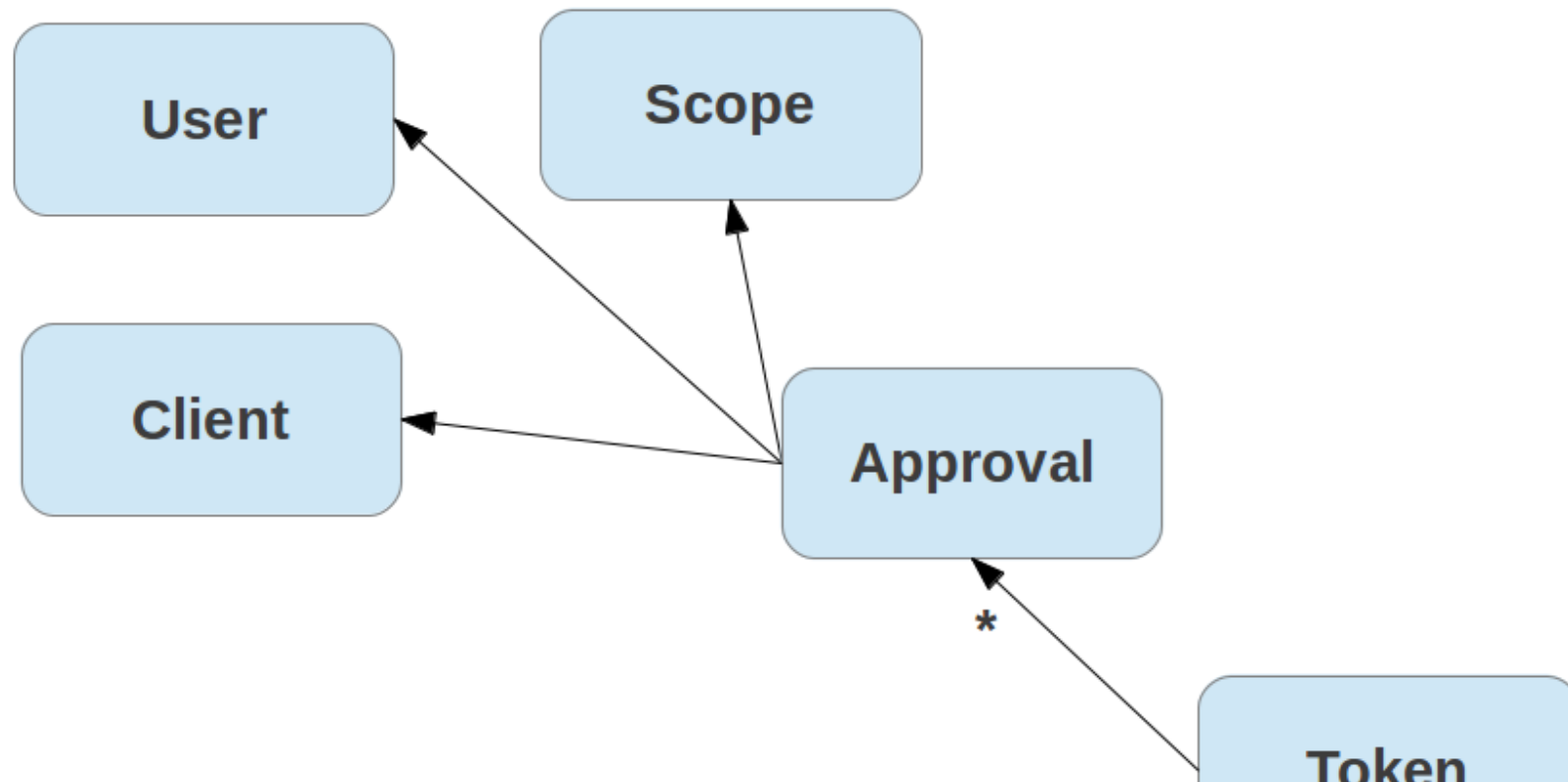




# Formal Model for User Approvals

---

It can be an advantage to store individual approvals independently (e.g. for explicit revokes of individual scopes):





# Spring OAuth Support

---

```
@EnableAuthorizationServer
class AuthorizationServer extends
AuthorizationServerConfigurerAdapter {

    @Override
    public void
configure(AuthorizationServerEndpointsConfigurer endpoints) throws
Exception {
        endpoints.tokenStore(tokenStore());
    }

    @Bean
    public ApprovalStore approvalStore() throws Exception {
        TokenApprovalStore store = new TokenApprovalStore();
        store.setTokenStore(tokenStore());
        return store;
    }

    ... // client config and tokenStore() defined here
}
```

# Simple Resource Server

---

```
@EnableResourceServer
class ResourceServer {
    @Bean
    TokenStore tokenStore() throws Exception {
        ...
    }
}
```

or

```
@EnableResourceServer
class ResourceServer {
    @Bean
    ResourceServerTokenServices tokenServices() throws Exception {
        ...
    }
}
```

# Simple Client Application

---

```
@EnableOAuth2Client
class ClientApplication {

    @Autowired // session-scoped
    private OAuth2ClientContext clientContext

    @Bean
    OAuth2RestOperations restTemplate() {
        new OAuth2RestTemplate(resource, clientContext)
    }

    ...
}
```



# Authorization Code Grant Summary

---

1. Authorization Server authenticates the User
2. Client starts the authorization flow and obtain User's approval
3. Authorization Server issues an authorization code (opaque one-time token)
4. Client exchanges the authorization code for an access token.

# Role of Resource Server

---

1. Extract token from request and decode it
2. Make access control decision
  - Scope
  - Audience
  - User account information (id, roles etc.)
  - Client information (id, roles etc.)
3. Send 403 (FORBIDDEN) if token not sufficient

# Role of the Authorization Server

---

1. Grant tokens
2. Interface for users to confirm that they authorize the Client to act on their behalf
3. Authenticate users (/authorize)
4. Authenticate clients (/token)

#1 and #4 are covered thoroughly by the spec; #2 and #3 not (for good reasons).

# Client Registration and Scopes

---

For secure channels a client has to authenticate itself to obtain a token, so it has to be known to the Authorization Server. Registration provides at a minimum:

- authentication (shared secret)
- registered redirect URI (optional but essential to prevent attacks)
- allowed scopes (clients are not permitted access to all resources)

Also useful:

- a way to identify which resources can be accessed
- ownership information (which user registered the client)

# More on Scopes

---

Per the spec they are arbitrary strings. The Authorization Server and the Resource Servers agree on the content and meanings.

Examples:

- Google: `https://www.googleapis.com/auth/userinfo.profile`
- Facebook: `email, read_stream, write_stream`
- UAA: `cloud_controller.read, cloud_controller.write, scim.read, openid`

Authorization Server has to decide whether to grant a token to a given client and user based on the requested scope (if any).

# Authentication and the Authorization Server

---

- Authentication (checking user credentials) is orthogonal to authorization (granting tokens)
- They don't have to be handled in the same component of a large system
- Authentication is often deferred to existing systems (SSO)
- Authorization Server has to be able to authenticate the OAuth endpoints (/authorize and /token)
- It *does not* have to collect credentials (except for grant\_type=password)

# OAuth2 and the Microservice

---

- Resource Servers might be microservices
- Web app clients: authorization code grant
- Browser clients (single page app): authorization code grant (better) or implicit grant
- Mobile and non-browser clients: password grant (maybe with mods for multifactor etc.)
- Service clients (intra-system): client credentials or relay user token

# Single Page Apps

---

With backend services CORS restrictions make reverse proxy useful (`@EnableZuulProxy`). Then you can acquire tokens in the client app and relay them to back end.

With no backend services, don't be shy, **use the session** (authorization code flow is vastly superior).

Spring Session helps a lot.



# Relaying User Tokens

---

*Front end app sends SSO token with user credentials to authenticate back end requests, back ends just relay it to each other as necessary.*

Simple but possibly flawed: the front end only needs access to user details to authenticate, but you need to give it permission to do other things to allow it access to the back ends.

Idea: exchange (with full authentication) the incoming token for an outgoing one with different permissions (client but not scope). Can use password grant (e.g. with the incoming token as a password).

# OAuth 1.0

---

- Another (slightly older) standard
- Includes request signing
- Common in early wave public APIs (e.g. Twitter)
- Spring Security OAuth is full solution at framework level

# So What's Wrong with That?

---

Nothing but...

- It's hard work for client app developers (crypto)
- (Notionally at least) superseded by OAuth2

# SAML Assertions

---

- Another standard with similar features to OAuth2
- XML based
- Common infrastructure in enterprise
- Spring Security SAML provides SP and Consumer roles (not IDP)
- Request signing is standardized

# So What's Wrong with That?

---

Nothing but...

- Painful to set up for servers and client
- Large amounts of XML data in HTTP headers
- Huge complexity for developers

N.B. exchanging SAML assertion for OAuth2 token is quote normal

# In Conclusion

---

- Lightweight services demand lightweight infrastructure
- Security is important, but should be unobtrusive
- Spring Security makes it all easier
- Special mention for Spring Session
- OAuth 2.0 is a standard, and has a lot of useful features
- Spring Security OAuth aims to be a complete OAuth2 solution at the framework level
- Cloudfoundry has an open source, OAuth2 identity service (UAA)

# Links

---

- <http://github.com/spring-projects/spring-security-oauth>
- <http://github.com/spring-projects/spring-security-oauth/tree/master/tests/annotation>
- <http://github.com/cloudfoundry/uaa>
- <http://blog.spring.io>
- <http://blog.cloudfoundry.org>
- <http://presos.dsyer.com/decks/microservice-security.html>
- Twitter: @david\_syer
- Email: [dsyer@pivotal.io](mailto:dsyer@pivotal.io)