Pivotal

Security with OAuth2

Addressing Security Requirements
for Distributed System

spring by Pivotal

Objectives

After completing this lesson, you should be able to

- Explain OAuth2 grant types and flow
- Implement authorization server
- Implement resource server
- Implement client application

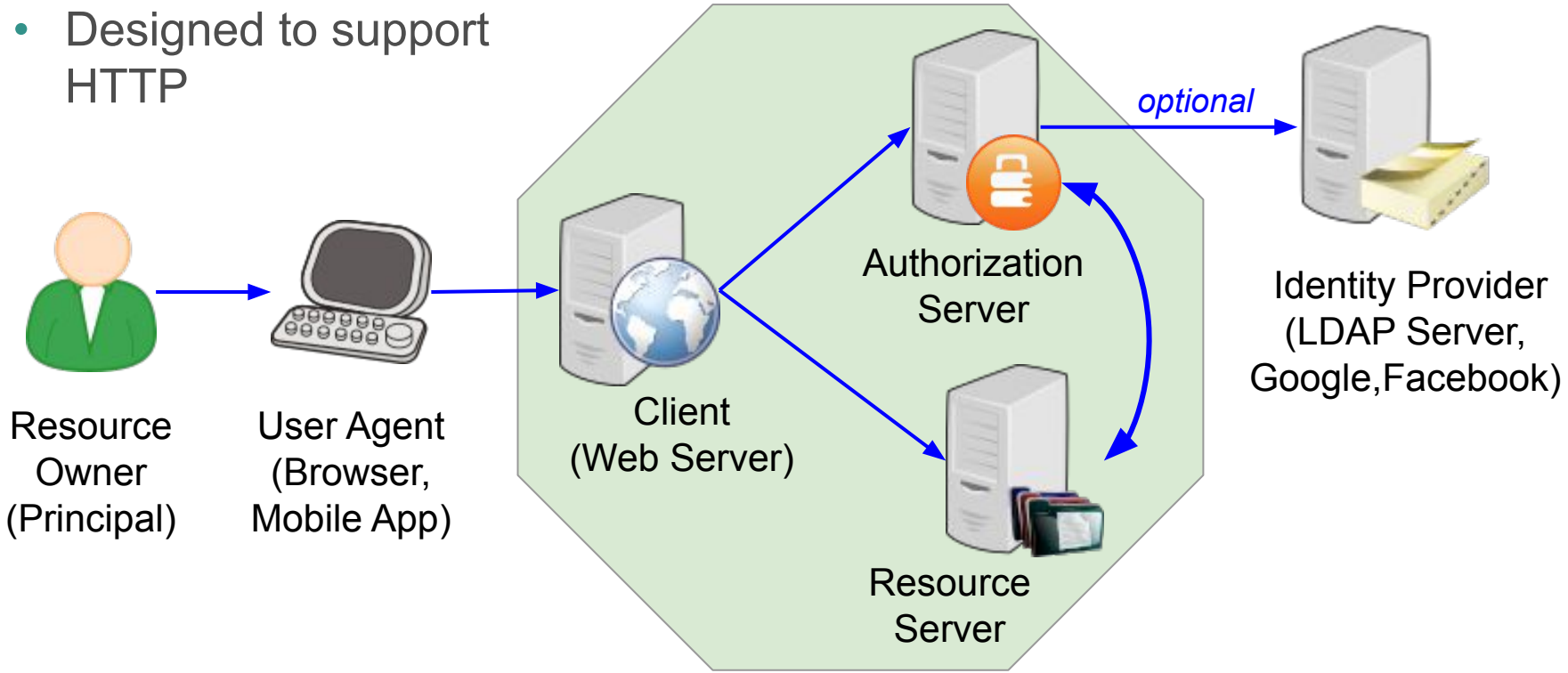
See: Spring Security OAuth
https://spring.io/projects/spring-security-oauth

Agenda

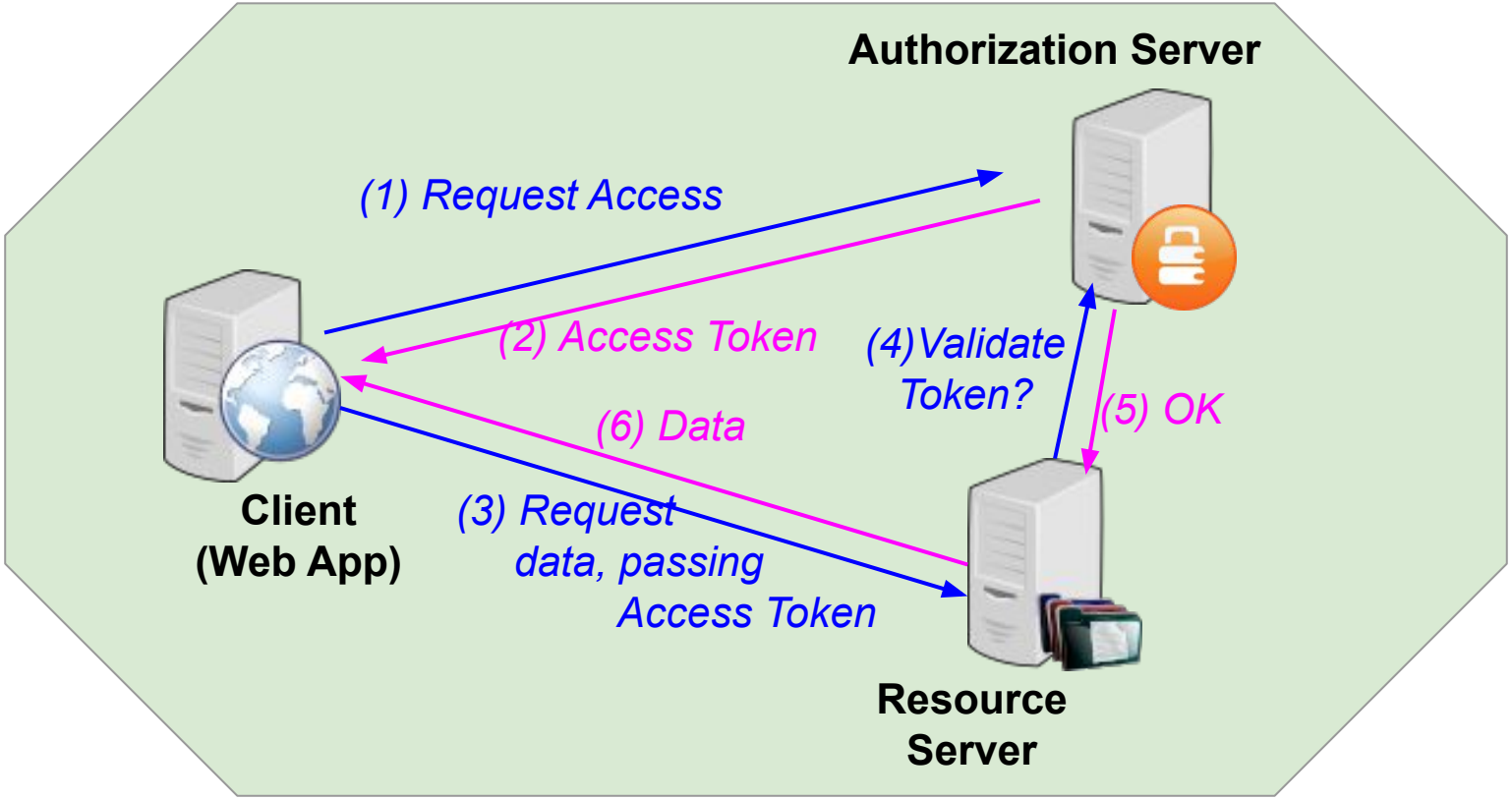
- **OAuth2 Overview**
- Implementing OAuth2
- Lab

Distributed *Authorization*

- Designed to support HTTP



The OAuth Flow - *the "Dance"*



Grant Types - Controlling the Flow

- Web App (**`grant_type=authorization_code`**)
  - *Example:* login to web-site using Facebook/Google credentials
- Single Page JavaScript App (**`grant_type=implicit`**)
  - As *Web App* but where client app not trusted to store credentials
- Native Mobile App (**`grant_type=password`**)
  - For clients capable of obtaining the user's credentials
- Service-to-Service App (**`grant_type=client_credentials`**)
  - *Example:* microservices interacting

https://alexbilbie.com/guide-to-oauth-2-grants

Agenda

- OAuth2 Overview
- **Implementing OAuth2**
- Lab

Step 1: Implement Authorization Server

- Take a Spring Boot Application
  - Add **`@EnableAuthorizationServer`**
  - Define an **`AuthorizationServerConfigurer`**

- **`AuthorizationServerConfigurer`** defines
  - Which tokens do we recognize?
  - Who can we authorize? (uses *Client Details Service*)

Authorization Server - 1

```
@SpringBootApplication
@EnableAuthorizationServer
public class AuthorizationServer {
   @Bean
    AuthorizationServerConfigurer authServerConfig() {
        return new AuthorizationServerConfigurerAdapter() {
            @Override public void configure(AuthorizationServerSecurityConfigurer security) {
                security.checkTokenAccess("hasAuthority('ROLE_TRUSTED_CLIENT')");
            }

            @Override
            public void configure(ClientDetailsServiceConfigurer c) throws Exception { … }
        };
    }

    public static void main(String[] args) {
        SpringApplication.run(AuthorizationServer.class, args);
    }
}
```

Allow tokens from clients with Trusted-Client authority

Define valid clients - *next slide*

SpEL Expression

Authorization Server - 2

Register OAuth server clients

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    PasswordEncoder passwordEncoder =
            PasswordEncoderFactories.createDelegatingPasswordEncoder();
    clients.inMemory() // in-memory ClientDetailsService
        .withClient("resource-server")   // Define the Resource Server username
            .secret(passwordEncoder.encode("secret"))                  // Password
            .authorizedGrantTypes("client_credentials")   // OAuth2 grant type
            .authorities("ROLE_TRUSTED_CLIENT")      // Authority (role)
    .and() //
        .withClient("web-server")        // Define the Client username
            .secret(passwordEncoder.encode("secret"))                  // Password
            .authorizedGrantTypes("client_credentials")  // OAuth2 grant type
            .scopes("resource.read");                                  // Authority
}
```

Equivalent to *UserDetailsService*

Step 2: Implement a Resource Server

- Take a Spring Boot Application
  - Add **`@EnableResourceServer`**
  - Configure using properties

```
# Username
security.oauth2.client.client-id=resource-server
# Password
security.oauth2.client.client-secret=secret
# “Check token” URL on Authentication Server (running on port 1111)
security.oauth2.resource.token-info-uri=http://localhost:1111/oauth/check_token
```

Add a Resource Service Configurer

```
@Bean
public ResourceServerConfigurer resourceServerConfigurer() {
    return new ResourceServerConfigurer() {
        @Override public void configure(ResourceServerSecurityConfigurer cfg) {
            cfg.resourceId("accounts");
        }

        @Override public void configure(HttpSecurity http) throws Exception {
            http.authorizeRequests()          // Specify URL restrictions
                .mvcMatchers(HttpMethod.GET, "/data/**")
                .access("#oauth2.hasScope('resource.read')") ;
        }
    };
}
```

Step 3: Implement a Client

- Use an **`OAuth2RestTemplate`** to make requests of the Resource Server
- Configure with properties

```
# Client identification: username, password, grant-type, authority (scope)
security.oauth2.client.client-id=web-server
security.oauth2.client.client-secret=secret
security.oauth2.client.grant-type=client_credentials
security.oauth2.client.scope=resource.read

# URL on Authentication Server to get token (running on port 1111)
security.oauth2.client.access-token-uri=http://localhost:1111/oauth/token
```

How It Works

Lab shows this interaction - do lab or run the solution to see

- Client makes a request for a token to the Auth Server
  - Passing its client details in request
  - Data returned contains the token
- Client uses **`OAuth2RestTemplate`** to get data from Resource Server
  - Passes token in *Authorization* request header
  - Resource server validates token with Auth Server
  - Returns data to Client if token validates OK

Summary

- OAuth provides authorization protocol for distributed system based on access token
- Spring Framework supports the implementations of
  - Authorization server
  - Resource server
  - Client application

Lab: OAuth2 Security

Lab project:
41-security-oauth2

Anticipated Lab time:
25 Minutes