# Purpose

During this lab we will add security to our applications. We will use OAuth for service-to-service communication.

In order to achieve this, we will set up each application as a resource server requiring OAuth. PCF's [Single Sign-On (SSO) Service](http://docs.pivotal.io/p-identity/1-5/index.html)will serve as our identity service. [Steeltoe Security](http://steeltoe.io/docs/steeltoe-security/) will ease the integration between the identity service and our applications.

# Discussion points

* OAuth basics ([authorization code flow diagram](https://docs.pivotal.io/p-identity/images/oauth_auth_code.png), [client credentials flow diagram](https://docs.pivotal.io/p-identity/images/oauth_client_credentials.png))
* Pivotal UAA
* Custom scopes and UAA and implications for blue-green deploys

# Introduce Steeltoe Security

Add the Steeltoe Security package to **AllocationServer**, **BacklogServer**, **TimesheetsServer**, and **RegistrationServer** projects. For example:

dotnet add ~/workspace/pal-tracker-distributed/Applications/AllocationsServer \

package Steeltoe.Security.Authentication.CloudFoundryCore --version 2.1.1

Make the following changes to Startup.cs of **AllocationServer**, **BacklogServer**, **TimesheetsServer**, and **RegistrationServer** to configure JWT authentication and authorization.

1. Change the MVC configuration to secure all endpoints of the application. This will require a scope claim with a value of "uaa.resource" to be present in the JWT. This is a default scope provided by UAA server. It will also only happen if security is not disabled via the DISABLE\_AUTH environment variable.
2. + using Microsoft.AspNetCore.Mvc.Authorization;
3. + using Microsoft.AspNetCore.Authorization;
4. public class Startup
5. {
6. // ...
7. public void ConfigureServices(IServiceCollection services)
8. {
9. // ...
10. // Add framework services.
11. - services.AddMvc();
12. -
13. + services.AddMvc(mvcOptions =>
14. + {
15. + if (!Configuration.GetValue("DISABLE\_AUTH", false))
16. + {
17. + // Set Authorized as default policy
18. + var policy = new AuthorizationPolicyBuilder(JwtBearerDefaults.AuthenticationScheme)
19. + .RequireAuthenticatedUser()
20. + .RequireClaim("scope", "uaa.resource")
21. + .Build();
22. + mvcOptions.Filters.Add(new AuthorizeFilter(policy));
23. + }
24. + });
25. services.AddDbContext<AllocationContext>(options => options.UseMySql(Configuration));
26. }
27. }
28. Add the Cloud Foundry JWT authentication service to the DI container:
29. // ...
30. + using Microsoft.AspNetCore.Authentication.JwtBearer;
31. + using Steeltoe.Security.Authentication.CloudFoundry;
32. namespace ...
33. {
34. public class Startup
35. {
36. // ...
37. public void ConfigureServices(IServiceCollection services)
38. {
39. // ...
40. + services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
41. + .AddCloudFoundryJwtBearer(Configuration);
42. }
43. }
44. }

You can read more about claim-based authorization [here](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims).

# Update ProjectClient for OAuth

Because the registration service is now secured, its consumers will need to provide an OAuth2 [access\_token](https://www.oauth.com/oauth2-servers/access-tokens/) with each request. We will update ProjectClient to do just that.

1. Add an parameter of type Func<Task<string>> as the last argument of the ProjectClient constructor. This will be a function responsible fetching the access token from our OAuth server.

Provide the retrieved access token as a Bearer token in the Authorization header of each request to the registration service.

The end result should look something like this:

[Hide ProjectClient.cs](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-cloud-developer/security/index.html" \l "pal-tracker-distributeddb7729ed-3211-4ad9-85ca-98410a159d66)

pal-tracker-distributed/Components/Allocations/ProjectClient.cs

﻿**using** System;

**using** System.Collections.Generic;

**using** System.Net.Http;

**using** System.Net.Http.Headers;

**using** System.Runtime.Serialization.Json;

**using** System.Threading.Tasks;

**using** Microsoft.Extensions.Logging;

**namespace** **Allocations**

{

**public** **class** **ProjectClient** : **IProjectClient**

{

**private** **readonly** HttpClient \_client;

**private** **readonly** ILogger<ProjectClient> \_logger;

**private** **readonly** Func<Task<**string**>> \_accessTokenFn;

**private** **readonly** IDictionary<**long**, ProjectInfo> \_projectCache = **new** Dictionary<**long**, ProjectInfo>();

**public** **ProjectClient**(HttpClient client, ILogger<ProjectClient> logger, Func<Task<**string**>> accessTokenFn)

{

\_client = client;

\_logger = logger;

\_accessTokenFn = accessTokenFn;

}

**public** **async** Task<ProjectInfo> **Get**(**long** projectId) =>

**await** **new** GetProjectCommand(DoGet, DoGetFromCache, projectId).ExecuteAsync();

**private** **async** Task<ProjectInfo> **DoGet**(**long** projectId)

{

**var** token = **await** \_accessTokenFn();

\_client.DefaultRequestHeaders.Accept.Clear();

\_client.DefaultRequestHeaders.Authorization = **new** AuthenticationHeaderValue("Bearer", token);

**var** streamTask = \_client.GetStreamAsync($"project?projectId={projectId}");

\_logger.LogInformation($"Attempting to fetch projectId: {projectId}");

**var** serializer = **new** DataContractJsonSerializer(**typeof**(ProjectInfo));

**var** project = serializer.ReadObject(**await** streamTask) **as** ProjectInfo;

\_projectCache.Add(projectId, project);

\_logger.LogInformation($"Caching projectId: {projectId}");

**return** project;

}

**private** Task<ProjectInfo> **DoGetFromCache**(**long** projectId)

{

\_logger.LogInformation($"Retrieving from cache projectId: {projectId}");

**return** Task.FromResult(\_projectCache[projectId]);

}

}

}

1. Update how the ProjectClient is constructed in Startup.cs of **AllocationServer**, **BacklogServer**, and **TimesheetsServer**.

We will use the [GetTokenAsync](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.authenticationhttpcontextextensions.gettokenasync) function to retrieve the access token from the [UAA Server response](https://docs.cloudfoundry.org/api/uaa/version/4.8.0/index.html#authorization-code-grant-2).

Because we are outside the scope of a controller, we don't have direct access to the HttpContext where the GetTokenAsync method resides. Instead, we will use an HttpContextAccessor to get a handle on the HttpContext, which will in turn give us access to GetTokenAsync.

Add HttpContextAccessor to the service container and then use it to get a handle on GetTokenAsync.

// ...

+ using Microsoft.AspNetCore.Authentication;

namespace ...

{

public class Startup

{

// ...

public void ConfigureServices(IServiceCollection services)

{

// ...

+ services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

services.AddSingleton<IProjectClient>(sp =>

{

var handler = new DiscoveryHttpClientHandler(sp.GetService<IDiscoveryClient>());

var httpClient = new HttpClient(handler, false)

{

BaseAddress = new Uri(Configuration.GetValue<string>("REGISTRATION\_SERVER\_ENDPOINT"))

};

var logger = sp.GetService<ILogger<ProjectClient>>();

+ var contextAccessor = sp.GetService<IHttpContextAccessor>();

- return new ProjectClient(httpClient, logger);

+ return new ProjectClient(

+ httpClient, logger,

+ () => contextAccessor.HttpContext.GetTokenAsync("access\_token")

+ );

});

// ...

}

}

}

# Disable security locally

We have coded our applications to be able to turn off authentication, which makes testing them locally easier.

1. You now need to set the DISABLE\_AUTH environment variable when starting the servers. Here is an example:
2. $env:DISABLE\_AUTH = "true"
3. dotnet run --urls "http://\*:8883" --project Applications/RegistrationServer
4. Update the integration test to disable authentication.
5. public FlowTest()
6. {
7. \_registrationServer = TestAppServerBuilder()
8. .AppName("RegistrationServer")
9. .Port(8883)
10. .Database("tracker\_registration\_test")
11. + .SetEnvironmentVariable("DISABLE\_AUTH", "true")
12. .Build();
13. \_allocationsServer = TestAppServerBuilder()
14. .AppName("AllocationsServer")
15. .Port(8881)
16. .Database("tracker\_allocations\_test")
17. .SetEnvironmentVariable("REGISTRATION\_SERVER\_ENDPOINT", \_registrationServer.Url())
18. + .SetEnvironmentVariable("DISABLE\_AUTH", "true")
19. .Build();
20. \_backlogServer = TestAppServerBuilder()
21. .AppName("BacklogServer")
22. .Port(8882)
23. .Database("tracker\_backlog\_test")
24. .SetEnvironmentVariable("REGISTRATION\_SERVER\_ENDPOINT", \_registrationServer.Url())
25. + .SetEnvironmentVariable("DISABLE\_AUTH", "true")
26. .Build();
27. \_timesheetsServer = TestAppServerBuilder()
28. .AppName("TimesheetsServer")
29. .Port(8884)
30. .Database("tracker\_timesheets\_test")
31. .SetEnvironmentVariable("REGISTRATION\_SERVER\_ENDPOINT", \_registrationServer.Url())
32. + .SetEnvironmentVariable("DISABLE\_AUTH", "true")
33. .Build();
34. }
35. Rebuild the application and run the tests (unit and integration).

# Configure SSO in Cloud Foundry

Begin by creating an SSO service instance.

Create a service instance of type p-identity (check the marketplace for details) and call it tracker-sso.

cf create-service p-identity p-identity tracker-sso

Edit **all** of your manifest.yml files to integrate with the SSO service. For each application:

1. Set the GRANT\_TYPE environment variable to client\_credentials.
2. Bind the application to the tracker-sso service instance.

For example, our allocations manifest looks like:

[Hide manifest-allocations.yml](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-cloud-developer/security/index.html" \l "pal-tracker-distributeddaa42bf4-a726-461c-a880-44f4b96d712a)

pal-tracker-distributed/manifest-allocations.yml

**---**

applications:

- name: tracker-allocations

instances: 1

memory: 256M

routes:

- route: allocations-pal-INITIALS.apps.pikes.pal.pivotal.io

path: Applications/AllocationsServer/bin/Release/netcoreapp2.1/publish

env:

GRANT\_TYPE: client\_credentials

services:

- tracker-allocations-database

- tracker-service-registry

- tracker-circuit-breaker-dashboard

- tracker-amqp

- tracker-sso

Push your code changes to your GitHub and CircleCI will deploy.

Now test that everything is working.

1. Look at the environment for tracker-allocations to get your UAA endpoint, client id, and client secret.
2. cf env tracker-allocations
3. // ..
4. "p-identity": [
5. {
6. "credentials": {
7. "auth\_domain": "https://p-identity.login.YOUR\_CF\_SYS\_URL",
8. "client\_id": "YOUR\_CLIENT\_ID",
9. "client\_secret": "YOUR\_CLIENT\_SECRET"
10. },
11. "label": "p-identity",
12. "name": "tracker-sso",
13. "plan": "p-identity",
14. "provider": null,
15. "syslog\_drain\_url": null,
16. "tags": [],
17. "volume\_mounts": []
18. }
19. ]
20. // ..
21. Obtain an access token from the UAA server
22. curl -k 'https://p-identity.login.YOUR\_CF\_SYS\_URL/oauth/token' -i -u 'YOUR\_CLIENT\_ID:YOUR\_CLIENT\_SECRET' -X POST -H 'Accept: application/json' -H 'Content-Type: application/x-www-form-urlencoded' -d 'grant\_type=client\_credentials&response\_type=token'
23. {
24. "access\_token": "SOME\_VERY\_LONG\_TOKEN",
25. "token\_type": "bearer",
26. "expires\_in": 43199,
27. "scope":"uaa.resource",
28. "jti":"cb94c15dc6a0438d849168c18996a1ce"
29. }
30. Use the above access token to make some test calls to your application.

# Assignment

Submit the following assignment. You can find the uaaUrl, clientId, and clientSecret by running cf env on any of microservice that has a UAA service bound to it and inspecting the VCAP\_SERVICES.p-identity.credentials key. For example:

"VCAP\_SERVICES": {

"credentials": {

"auth\_domain": "<uaaUrl>",

"client\_id": "<clientId>",

"client\_secret": "<clientSecret>"

}

}

Keep in mind that **clientId and clientSecret change after each deployment**.

./gradlew dotnetCloudNativeDeveloperDistributedSystemWithSecurity \

-PregistrationServerUrl=https://<registration-app-url> \

-PbacklogServerUrl=https://<backlog-app-url> \

-PallocationsServerUrl=https://<allocations-app-url> \

-PtimesheetsServerUrl=https://<timesheets-app-url> \

-PuaaUrl=https://<uaa-url> \

-PclientId=<client-id> \

-PclientSecret=<client-secret>

# Extra

If you are finished with this assignment before the rest of the class is done, try adding a UI that talks to one of your applications. You may want to read up on OAuth [grant types](http://projects.spring.io/spring-security-oauth/1.x/docs/oauth2.html#grant-types).