# Purpose

We will use [Hystrix](http://steeltoe.io/docs/steeltoe-circuitbreaker/#1-0-netflix-hystrix) to add a circuit breaker when fetching a project.

The AllocationServer, BacklogServer, and TimesheetsServer all integrate with the RegistrationServer. They use it to verify that a project is active before creating a dependent resource. If the RegistrationServer is down then we cannot create any resources on the other servers.

At the end of the lab, if a project takes too long to load or fails to be retrieved, our application will display a cached project.

This allows our system to function smoothly even if there is a problem with our registration server.

# Discussion points

* Cascading failures in a distributed system.
* Circuit breaker pattern.
* Steeltoe Hystrix vs Metrics (SSE vs MQ) packages.
* Steeltoe package variants (\*Base vs \*Corevs\*Autofac).
* Rabbit MQ [spring.cloud.hystrix.stream](http://localhost:15672/#/exchanges/%2F/spring.cloud.hystrix.stream) exchange

# Add a circuit breaker

1. Add the Steeltoe.CircuitBreaker.HystrixCore, Steeltoe.CircuitBreaker.Hystrix.MetricsStreamCore, andRabbitMQ.Client packages to **Allocation**, **Backlog**, and **Timesheets** components. For example for Allocation:
2. dotnet add ~/workspace/pal-tracker-distributed/Components/Allocations \
3. package Steeltoe.CircuitBreaker.HystrixCore --version 2.1.1
4. dotnet add ~/workspace/pal-tracker-distributed/Components/Allocations \
5. package Steeltoe.CircuitBreaker.Hystrix.MetricsStreamCore --version 2.1.1
6. dotnet add ~/workspace/pal-tracker-distributed/Components/Allocations \
7. package RabbitMQ.Client --version 5.1.0
   * Steeltoe.CircuitBreaker.HystrixBase - main implementation of the circuit breaker, works on .NET Core or .NET Framework
   * Steeltoe.CircuitBreaker.HystrixCore - circuit breaker + .NET Core DI extension methods
   * Steeltoe.CircuitBreaker.HystrixAutofac - circuit breaker + Autofac DI hooks
   * Steeltoe.CircuitBreaker.Hystrix.MetricsStreamCore - streams hystrix events using server Rabbit MQ, works with PCF Hystrix dashboard, not OSS hystrix dashboard. (.NET Core variant)
   * Steeltoe.CircuitBreaker.Hystrix.MetricsStreamAutofac - stream hystrix events using server Rabbit MQ, works with PCF Hystrix dashboard, not OSS hystrix dashboard. (.NET Framework/Autofac variant)
   * Steeltoe.CircuitBreaker.Hystrix.MetricsEventsCore - stream hystrix events using Server-Sent Events (SSE), works with OSS Hystrix dashboard, but not PCF hystrix dashboard. (.NET Core variant)
   * Steeltoe.CircuitBreaker.Hystrix.MetricsEventsAutofac - stream hystrix events using Server-Sent Events (SSE), works with OSS Hystrix dashboard, but not PCF hystrix dashboard. (.NET Framework/Autofac variant)

**Circuit breaker can be used without event streaming by including one of the top packages.**

1. In the **Allocation**, **Backlog**, and **Timesheets** components create a GetProjectCommand, which inherits form the HystrixCommand base class. It will encapsulate the primary and fallback logic of the project retrieval operation, as well as the requested project's id.

[Hide GetProjectCommand.cs](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-cloud-developer/circuit-breakers/index.html" \l "pal-tracker-distributedfc76060a-802e-40cb-b04c-e4594e0a32c6)

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

﻿**using** System;

**using** System.Threading.Tasks;

**using** Steeltoe.CircuitBreaker.Hystrix;

**namespace** **Allocations**

{

**public** **class** **GetProjectCommand** : **HystrixCommand**<**ProjectInfo**>

{

**private** **readonly** Func<**long**, Task<ProjectInfo>> \_getProjectFn;

**private** **readonly** **long** \_projectId;

**private** **readonly** Func<**long**, Task<ProjectInfo>> \_getProjectFallbackFn;

**public** **GetProjectCommand**(

Func<**long**, Task<ProjectInfo>> getProjectFn,

Func<**long**, Task<ProjectInfo>> getProjectFallbackFn,

**long** projectId

) : **base**(HystrixCommandGroupKeyDefault.AsKey("ProjectClientGroup"))

{

\_getProjectFn = getProjectFn;

\_projectId = projectId;

\_getProjectFallbackFn = getProjectFallbackFn;

}

**protected** **override** **async** Task<ProjectInfo> **RunAsync**() => **await** \_getProjectFn(\_projectId);

**protected** **override** **async** Task<ProjectInfo> **RunFallbackAsync**() => **await** \_getProjectFallbackFn(\_projectId);

}

}

1. Update the Startup.cs for the **AllocationServer**, **BacklogServer**, and **TimesheetsServer** to:
   * Construct ProjectClient with a logger
   * Add Hystrix Metrics to container
   * Start Hystrix metrics stream service
   * Use Hystrix Request contexts
2. // ...
3. + using Steeltoe.CircuitBreaker.Hystrix;
4. namespace ...
5. {
6. public class Startup
7. {
8. // ...
9. public void ConfigureServices(IServiceCollection services)
10. {
11. // ...
12. services.AddSingleton<IProjectClient>(sp =>
13. {
14. var handler = new DiscoveryHttpClientHandler(sp.GetService<IDiscoveryClient>());
15. var httpClient = new HttpClient(handler, false)
16. {
17. BaseAddress = new Uri(Configuration.GetValue<string>("REGISTRATION\_SERVER\_ENDPOINT"))
18. };
19. - return new ProjectClient(httpClient);
20. + var logger = sp.GetService<ILogger<ProjectClient>>();
21. + return new ProjectClient(httpClient, logger);
22. });
23. +
24. + services.AddHystrixMetricsStream(Configuration);
25. }
26. public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
27. {
28. // ...
29. app.UseMvc();
30. app.UseDiscoveryClient();
31. + app.UseHystrixMetricsStream();
32. + app.UseHystrixRequestContext();
33. }
34. }
35. }
36. To simplify local testing, add the following configuration to the appsettings.json of **AllocationServer**, **BacklogServer**, and **TimesheetsServer**:
37. "hystrix": {
38. "command": {
39. "default": {
40. "execution": {
41. "isolation": {
42. "thread": {
43. "timeoutInMilliseconds": 5000
44. }
45. }
46. }
47. }
48. }
49. }

You can find more documentation on Hystrix configuration [on the official wiki](https://github.com/Netflix/Hystrix/wiki/Configuration#circuitBreaker.requestVolumeThreshold).

## Update ProjectClient

Update ProjectClient in the **Allocation**, **Backlog**, and **Timesheets** components as follows:

1. Move the logic for retrieving a project into another method (e.g. DoGet). Update it to store any fetched project in a local in-memory cache (using a Dictionary should be sufficient).
2. Implement another method (e.g. DoGetFromCache), which will return the project from our in-memory cache.
3. Finally, update the Get method to instantiate a GetProjectCommand (providing the other two functions and project id in the constructor). Execute the command asynchronously by calling its ExecuteAsync method.

Here is an example implementation if you would like some help:

[Hide ProjectClient.cs](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-cloud-developer/circuit-breakers/index.html" \l "pal-tracker-distributed8e4c282e-0fdf-4d0e-ac54-67fe5ab8d83e)

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

﻿**using** System.Collections.Generic;

**using** System.Net.Http;

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

**public** **ProjectClient**(HttpClient client, ILogger<ProjectClient> logger)

{

\_client = client;

\_logger = logger;

}

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

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

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

{

\_client.DefaultRequestHeaders.Accept.Clear();

**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]);

}

}

}

# Test locally

Follow the steps below to validate that the circuit breaker is working correctly.

1. Start all four servers and Eureka.
2. Create a user.
3. curl -i -XPOST -H"Content-Type: application/json" localhost:8883/registration -d'{"name": "Tom"}'
4. Find the account associated to the created user.
5. curl -i localhost:8883/accounts?ownerId=<USER\_ID>
6. Create Project A, and Project B using the registration server.
7. curl -i -XPOST -H"Content-Type: application/json" localhost:8883/projects -d'{"name": "Project A", "accountId": <ACCOUNT\_ID>}'
8. curl -i -XPOST -H"Content-Type: application/json" localhost:8883/projects -d'{"name": "Project B", "accountId": <ACCOUNT\_ID>}'
9. Create an allocation using Project A's id.
10. curl -i -XPOST -H"Content-Type: application/json" localhost:8881/allocations/ -d'{"projectId": <PROJECT\_A\_ID>, "userId": <USER\_ID>, "firstDay": "2015-05-17", "lastDay": "2015-05-18"}'
11. Stop the registration server.
12. Create an allocation using Project B's id. The call should fail because there is no cached result for Project B.
13. curl -i -XPOST -H"Content-Type: application/json" localhost:8881/allocations/ -d'{"projectId": <PROJECT\_B\_ID>, "userId": <USER\_ID>, "firstDay": "2015-05-17", "lastDay": "2015-05-18"}'
14. Create an allocation using Project A's id. The call should succeed because there is a cached result for Project A.
15. curl -i -XPOST -H"Content-Type: application/json" localhost:8881/allocations/ -d'{"projectId": <PROJECT\_A\_ID>, "userId": <USER\_ID>, "firstDay": "2015-05-17", "lastDay": "2015-05-18"}'
16. Start the registration server.
17. Create an allocation with Project B's id. The call should eventually succeed because the registration server is back up. It may take a bit for the circuit to recognize the registration server is back up.

# Deploy

1. Create Circuit Breaker Dashboard and AMQP service instances:
2. cf create-service p-circuit-breaker-dashboard standard tracker-circuit-breaker-dashboard
3. cf create-service p-rabbitmq standard tracker-amqp

Creation will take a few minutes and must finish before we bind the service.

1. Add the service bindings for both the dashboard and amqp to the manifests of **AllocationServer**, **BacklogServer**, and**TimesheetsServer**.
2. services:
3. ...
4. - tracker-circuit-breaker-dashboard
5. - tracker-amqp
6. Commit and push your code. CircleCI will deploy the applications to Cloud Foundry.
7. Verify that everything is working as expected by following the above set of curl commands against your PCF environment.
8. Confirm that the three applications (allocations, backlog, and timesheets) appear the Circuit Breaker dashboard:
   1. Go to the Apps Manager in your browser and log in with your CF CLI credentials. The URL is apps.sys.<FOUNDATION\_NAME>.pal.pivotal.io. For example, if you PCF API endpoint is api.sys.longs.pal.pivotal.iothen your Apps Manager URL is apps.sys.longs.pal.pivotal.io.
   2. Find your organization and space in the left navigation.
   3. Click the **Services** tab.
   4. Click the **Circuit Breaker** service in the list.
   5. Click the **Manage** link.

# Assignment

Submit the following assignment:

**cd** ~/workspace/assignment-submission

./gradlew dotnetCloudNativeDeveloperDistributedSystemWithCircuitBreaker \

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

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

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

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

# Extra

If you are finished with this assignment before the rest of the class is done, try using Redis to store cached projects rather than the in-memory data store.