Sending Jobs to Helix

The primary reason to use Helix is to leverage its scalability to run tests. Arcade and the Arcade SDK provide out-of-the-box functionality to interface with Helix.

Understanding Helix

This document assumes you have familiarity with Helix. If that is not the case, it is recommended that you start here.

Getting Started

First, you have to import the SDK. Everything that follows requires dotnet-cli 2.1.300 and needs the following files in a directory at or above the project's directory.

global.json

```
{
  "msbuild-sdks": {
    "Microsoft.DotNet.Helix.Sdk": "<version of helix sdk package from package feed>"
  }
}
```

Example: `"Microsoft.DotNet.Helix.Sdk": "1.0.0-beta.18502.3"`

NuGet.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <clear />
    <add key="dotnet-core" value="https://dotnetfeed.blob.core.windows.net/dotnet-core/index
  </packageSources>
</configuration>
```

Helix Access Token

Helix access tokens are used to authenticate users sending jobs to Helix.

External builds (Public CI)

For external builds, you don't need to specify an access token (indeed, doing so is prohibited). Simply specify a *Creator* in order to send the jobs to Helix anonymously. The "creator" should be an identifiable username which is clearly related to your build. For example, Arcade might specify a creator of `arcade`.

Please note that external jobs may only be submitted to queues which end with the value `IsInternalOnly` set to false. In general, these queues end with **.Open**; however, this is not necessarily true. To determine this value for a particular queue, see the list of available queues here.

Example:

```
steps:
- template: /eng/common/templates/steps/send-to-helix.yml
  displayName: Send to Helix
  parameters:
    ## more variables go here
    Creator: # specify your creator here
```

Internal builds

In the dev.azure.com/dnceng/internal project, you can use the `DotNet-HelixApi-Access` variable group to provide this secret to your build and then specify the `HelixApiAccessToken` secret for the `HelixAccessToken` parameter.

Please note that authorized jobs *cannot* be submitted to queues with `IsInternalOnly` set to false. To determine this value for a particular queue, see the list of available queues here.

Example:

```
variables:
- group: DotNet-HelixApi-Access

steps:
# $HelixAccessToken is automatically injected into the environment
- template: /eng/common/templates/steps/send-to-helix.yml
  displayName: Send to Helix
  parameters:
    HelixAccessToken: $(HelixApiAccessToken)
    # other parameters here
```

The Simple Case

The simplest Helix use-case is zipping up a single folder containing your project's tests and a batch file which runs those tests. To accomplish this, reference Arcade's `send-to-helix` template in `eng/common/templates/steps/send-to-helix.yml` from your `azure-pipelines.yml` file.

Simply specify the xUnit project(s) you wish to run (semicolon delimited) with the `XUnitProjects` parameter. Then, specify: * the `XUnitPublishTargetFramework` – this is the framework your **test projects are targeting**, e.g. `netcoreapp3.1`. * the `XUnitRuntimeTargetFramework` – this is the framework version of xUnit you want to use from the xUnit

NuGet package, e.g. `netcoreapp2.0`. Notably, the xUnit console runner only supports up to netcoreapp2.0 as of 14 March 2018, so this is the target that should be specified for running against any higher version test projects. * the `XUnitRunnerVersion` (the version of the xUnit nuget package you want to use, e.g. `2.4.2`).

Finally, set `IncludeDotNetCli` to true and specify which `DotNetCliPackageType` (`sdk`, `runtime` or `aspnetcore-runtime`) and `DotNetCliVersion` you wish to use. (For a full list of .NET CLI versions/package types, see these links: 3.0, 2.1, 2.2.)

The list of available Helix queues can be found on the Helix homepage.

```
- template: /eng/common/templates/steps/send-to-helix.yml
  parameters:
    HelixSource: pr/your/helix/source # sources must start with pr/, official/, prodcon/,
    HelixType: type/tests
    # HelixBuild: $(Build.BuildNumber) -- This property is set by default
    HelixTargetQueues: Windows.10.Amd64.Open;Windows.81.Amd64.Open;Windows.7.Amd64.Open #
    # HelixAccessToken: $(HelixAccessToken) this token is only for internal builds
    # HelixConfiguration: '' -- any property that you would like to attached to a job
    # HelixPreCommands: '' -- any commands that you would like to run prior to running yo
    # HelixPostCommands: '' -- any commands that you would like to run after running your
    XUnitProjects: $(Build.SourcesDirectory)/HelloTests/HelloTests.csproj # specify your
    # XUnitWorkItemTimeout: '00:05:00' -- a timeout (specified as a System.TimeSpan strin
    XUnitPublishTargetFramework: netcoreapp3.1 # specify your publish target framework he
    XUnitRuntimeTargetFramework: netcoreapp2.0 # specify the framework you want to use fo
    XUnitRunnerVersion: 2.4.2 # specify the version of xUnit runner you wish to use here
    # WorkItemDirectory: '' -- payload directory to zip up and send to Helix; requires Wo
    # WorkItemCommand: '' -- a command to execute on the payload; requires WorkItemDirect
    # WorkItemTimeout: '' -- a timeout (specified as a System.TimeSpan string) for the wo
    IncludeDotNetCli: true
    DotNetCliPackageType: sdk
    DotNetCliVersion: 2.1.403 # full list of versions here: https://raw.githubusercontent
    # WaitForWorkItemCompletion: true -- defaults to true
    Creator: arcade # specify an appropriate Creator here -- required for external builds
    # DisplayNamePrefix: 'Send job to Helix' -- the Helix task's display name in AzDO. De
    # condition: succeeded() - defaults to succeeded()
```

The More Complex Case

For anything more complex than the above example, you'll want to create your own MSBuild proj file to specify the work items and correlation payloads you want to send up to Helix. Full documentation on how to do this can be found in the SDK's readme.

Viewing test results

All test results will be downloaded to the Azure DevOps build and viewable through the **Tests** tab.

External test results

Tests results for “public” projects are accessible via the link which is provided in the build output.

Example build output:

```
Sending Job to Debian.9.Amd64.Arcade.Open...
Sent Helix Job; see work items at https://helix.dot.net/api/jobs/38f272c2-7999-44e4-953b-d5c
Waiting for completion of job 38f272c2-7999-44e4-953b-d5d003b614ce on Debian.9.Amd64.Arcade.
Job 38f272c2-7999-44e4-953b-d5d003b614ce on Debian.9.Amd64.Arcade.Open is completed with 14
```

Internal test results

Test results for “internal” projects are accessible via the link which is provided in the build output or via Azure DevOps Tests tab.

Helix Work Item Exit statuses:

As surfaced by the Helix API and backing Kusto (Azure Data Explorer) database, here’s an explanation of the statuses you may see from Helix work items.

- Pass - Work item did everything as expected, and its exit code was 0.
- Fail – Work item did everything as expected including reporting results, but had a non-zero exit code.
- BadExit – A non-zero exit code, typically a crash, where also no results were reported
- None – No network problems to Azure components, and stuff worked, but we were unable to start your work item. Often this will be caused by problems like no space left on the device or a payload that is so large it cannot be downloaded within 10 minutes, but can also be indicative of a misconfigured machine. We strive for 0 of these from machine configuration, but given we intentionally do not set limits on payloads this can and does happen for the former reasons.
- Error – Same as None except that the Helix client catches it and sends telemetry saying it knew it was in error.
- DeadLetter – Multiple Helix machines tried to run the work item “Max Delivery Count” times (typically configured as 3), and were unable to complete it, OR the queue is deprecated (end-of life: happening more in these core-reduction times). No output is registered, and the console log is redirected to a page explaining what DeadLetter means.

- InfraRetry – Work item completed as expected, but on the 2nd-Nth attempt; this can be a requested-by-the-workitem retry, machine being rebooted or deleted during execution, or any number of random Azure components being flaky. Typically ignoreable for test runs.
- PassOnRetry – Special legacy retry functionality which is purposefully obsoleted as it does not play well with Azure DevOps test reporting (reporting the same facts twice causes issues)
- Timeout – Work Item did not complete within its specified timeout and was forcibly killed. Corresponds to exit code -3 (made up value since the process never exited)

Was this helpful? 👍 👎