

Estensioni personalizzate per le pipeline di Azure DevOps e GitHub con .NET 5

Matteo Tumiati

Senior DevOps Engineer @icubedsrl @openloopit

Content Manager @aspitalia

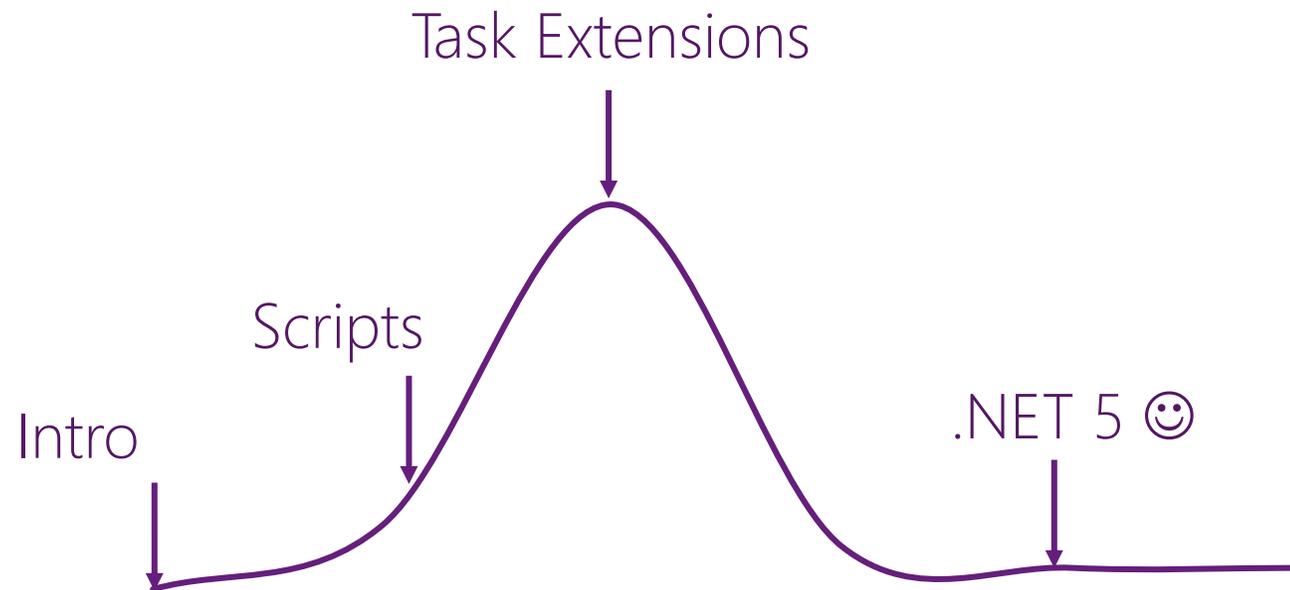
Microsoft WinDev/Dev Technologies MVP

matteot@aspitalia.com | @xtumiox

.NET Conference
Italia 2020

.NET

Agenda

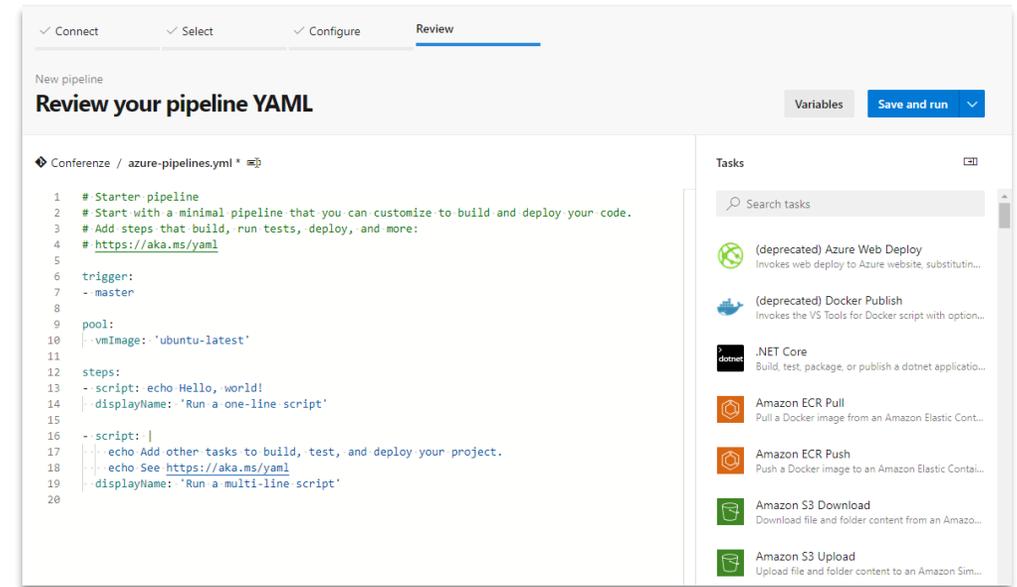
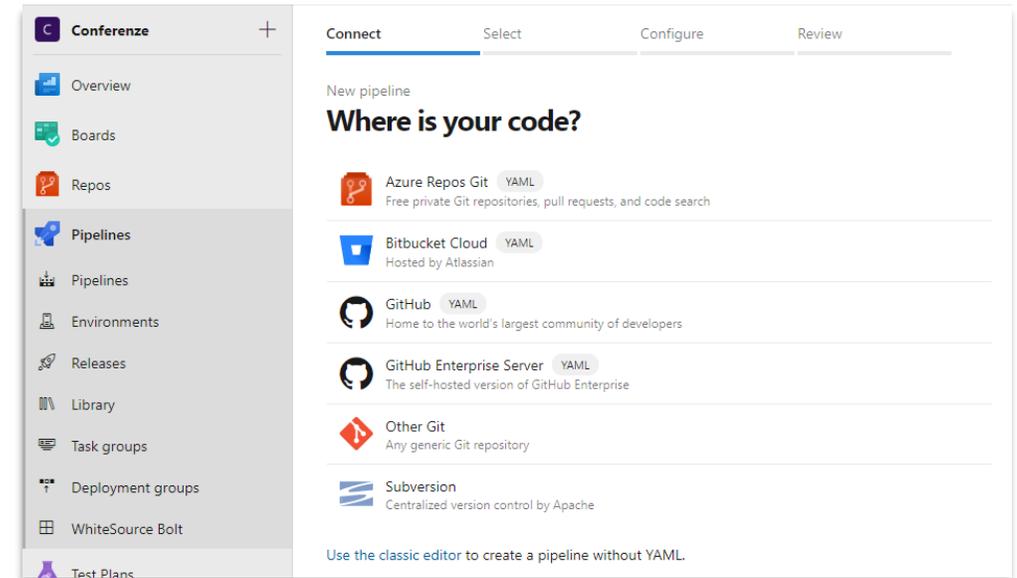


Build flow

The pipeline is created either from classic UI or from YAML

The pipeline definition (or process) is represented by a list of steps executed in sequence

Steps can also be grouped in jobs and jobs in stages to create context and run multiple operations in parallel (if needed)



What are steps?

Steps are a way to execute certain activities

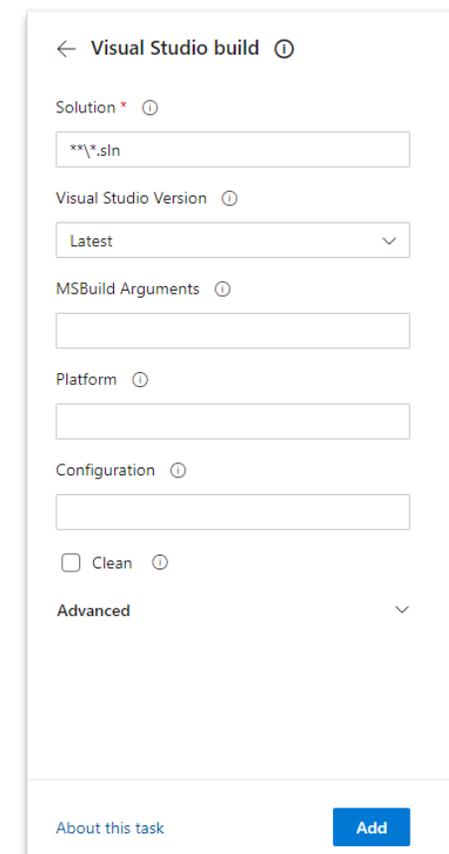
- Restore packages, build a solution, publish artifacts...

Steps can be tasks, scripts, or references to external templates

- Templates only apply when using YAML pipelines

Each step can be referenced in pipelines either from:

- A “predefined” list of built-in tasks in Azure DevOps or
- The Azure DevOps marketplace



The screenshot shows the configuration page for the 'Visual Studio build' task. It includes the following fields and options:

- Solution ***: A text input field containing the wildcard pattern `***.sln`.
- Visual Studio Version**: A dropdown menu currently set to `Latest`.
- MSBuild Arguments**: An empty text input field.
- Platform**: An empty text input field.
- Configuration**: An empty text input field.
- Clean**: A checkbox that is currently unchecked.
- Advanced**: A dropdown menu that is currently collapsed.

At the bottom of the configuration area, there is a link for [About this task](#) and a blue **Add** button.

What if I need a task that doesn't exist?

Suppose that:

- I need to deploy on my datacenter
- I must comply to a set of policies when executing steps
- I want to execute a task (for simplicity) but it doesn't provide all the information I need

Customizations!

- It's hard to personalize a world-wide multi-tenant tool...
- Azure DevOps has some extensibility points, not only API based...

There are two main options:

- Use PowerShell, Bash, CMD or whatever to script your requirements
- Create an extensions!

Execute custom scripts

```
[CmdletBinding()]  
param (  
    [Parameter(Mandatory = $True)]  
    [String]  
    $Name  
)  
  
Write-Host "Hello world, $Name!"
```

Step 1: write the script in the language you prefer based on requirements

In this case it's Hello-World.ps1 as it will run on Windows

```
- task: PowerShell@2  
  displayName: Execute Script  
  inputs:  
    filePath: './Hello-World.ps1'  
    arguments: '-Name Matteo'  
    failOnStderr: true
```

```
- name: Execute Script  
  run: ./Hello-World.ps1 -Name Matteo  
  shell: powershell
```

Step 2: execute the script in Azure DevOps (top) or GitHub (bottom)

Should I use custom scripts?

Pros

- You can write any code you need in the way you need to comply to requirements (i.e. PowerShell on Windows, Bash on Linux)
- Can re-use existing code, if any
- There is plenty of documentation available for any command

Cons

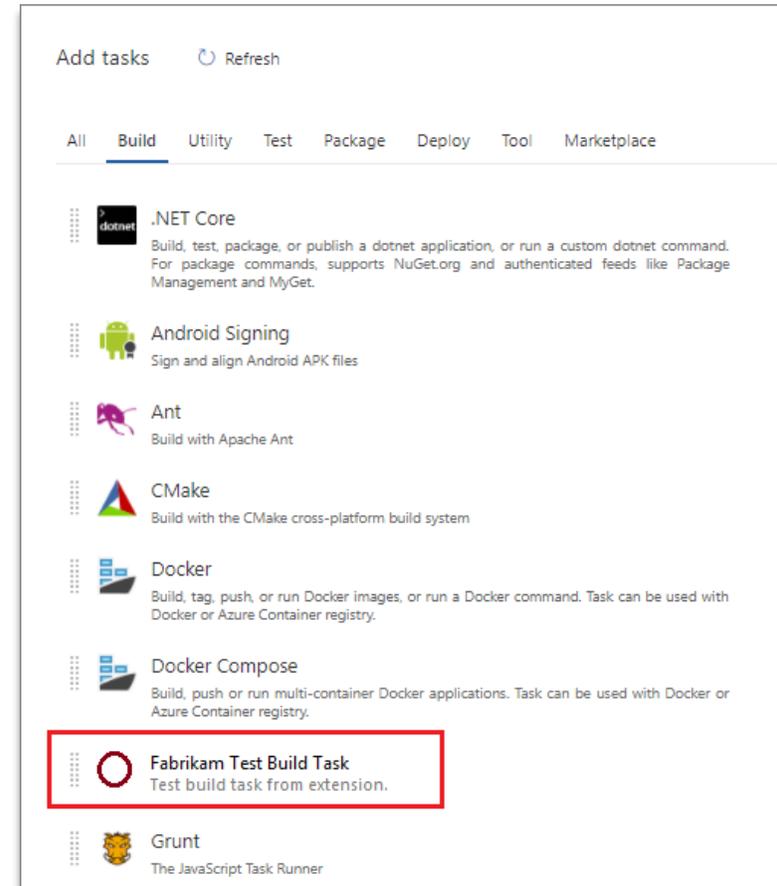
- Lack of user interface means that all the parameters must be passed in input
- Parameters must be validated at runtime (no compile-time checking)
- May be hard to get started if you are not skilled enough
- Can be tricky if the same script has to run against different OS

Custom pipeline task extensions

The best experience you can have OOB for your users
Share the same task with the community on the Azure DevOps marketplace

Some issues presented by scripts are solved:

- Immediately validate (partially) input parameters with the UI
- Can use an SDK to get started
- If a script doesn't run on a certain OS, you get notified when adding it in the pipeline with an unsupported agent



A lot of prerequisites...

An organization in Azure DevOps and an account for the marketplace

- Not that obvious, if you plan to use GitHub...

A text editor

- Visual Studio Code, which provides IntelliSense and debugging support is perfect

The latest version of Node.js

- The production environment uses only Node10 or Node6 (by using the "Node" in the "execution" object instead of Node10)

TypeScript Compiler 2.2.0 or greater

- Microsoft recommends version 4.0.2 or newer for tasks that use Node10

Cross-platform CLI for Azure DevOps (tfx-cli) to package extensions

- You can install tfx-cli by using by running `npm i -g tfx-cli`.

Author a task extension – Part 0

```
|--- README.md           // brief description on how to use the task
|--- images              // displayed in the pipeline, marketplace
  |--- extension-icon.png
|--- task                // where your task scripts are placed
|--- tests               // unit tests folder
|--- vss-extension.json  // extension's manifest
```

Author a task extension – Part 1

```
# Create a directory and package.json file
```

```
npm init
```

```
# Add azure-pipelines-task-lib, that should be used to create tasks
```

```
npm install azure-pipelines-task-lib --save
```

```
# Install and configure typescript
```

```
npm install @types/node --save-dev
```

```
npm install @types/q --save-dev
```

```
npm install typescript@4.0.2 --save-dev
```

```
tsc --init
```

```
# Configure git
```

```
echo node_modules > .gitignore
```

Author a task extension – Part 2

Create the *task.json* file that describes the build (or release) task.

This is what the build (or release) system in Azure DevOps uses to render configuration options to the user and know which scripts to execute at build/release time

```
{
  "$schema": "https://raw.githubusercontent.com/Microsoft/azure-pipelines-task-lib/master/tasks.schema.json",
  "id": "{{taskguid}}",
  "name": "{{taskname}}",
  "friendlyName": "{{taskfriendlyname}}",
  "description": "{{taskdescription}}",
  "category": "Utility",
  "author": "{{taskauthor}}",
  "version": {
    "Major": 0,
    "Minor": 1,
    "Patch": 0
  },
  "instanceNameFormat": "Echo $(samplestring)",
  "inputs": [
    {
      "name": "name",
      "type": "string",
      "label": "The name of the user running the task",
      "defaultValue": "World",
      "required": true,
    }
  ],
  "execution": {
    "Node10": {
      "target": "index.js"
    }
  }
}
```

Author a task extension – Part 3

Create an *index.ts* file. This code runs when the task is called.

```
import tl = require('azure-pipelines-task-lib/task');

async function run() {
  try {
    const inputString: string | undefined = tl.getInput('name', true);
    if (inputString == 'bad') {
      tl.setResult(tl.TaskResult.Failed, 'Bad input was given');
      return;
    }
    console.log('Hello', inputString);
  }
  catch (err) {
    tl.setResult(tl.TaskResult.Failed, err.message);
  }
}

run();
```

Author a task extension – Test

```
npm install mocha --save-dev -g  
npm install sync-request --save-dev  
npm install @types/mocha --save-dev
```

Step 1: include all the dependencies of your test framework (Mocha)

```
tsc  
$env:TASK_TEST_TRACE=1  
mocha tests/_suite.js
```

Step 3: compile and run tests

```
import * as path from 'path';  
import * as assert from 'assert';  
import * as ttm from 'azure-pipelines-task-lib/mock-test';  
  
describe('Sample task tests', function () {  
    before( function() {  
    });  
  
    after(() => {  
    });  
  
    it('should succeed with simple inputs', function(done: Mocha.Done) {  
        // Add success test here  
    });  
  
    it('it should fail if tool returns 1', function(done: Mocha.Done) {  
        // Add failure test here  
    });  
});
```

Step 2: write the test cases in *tests/_suite.js*

Author a task extension – Build and run

Enter "tsc" from the source folder to compile an *index.js* file from *index.ts*.

Run the same file to see the logs that are *simulating* the experience of the Azure DevOps pipeline

```
# Compile
tsc

# Prepare input parameter and execute
$env:INPUT_NAME="Matteo"
node index.js

# Logs
##vso[task.debug]agent.workFolder=undefined
##vso[task.debug]loading inputs and endpoints
##vso[task.debug]loading INPUT_NAME
##vso[task.debug]loaded 1
##vso[task.debug]Agent.ProxyUrl=undefined
##vso[task.debug]Agent.CAInfo=undefined
##vso[task.debug]Agent.ClientCert=undefined
##vso[task.debug]Agent.SkipCertValidation=undefined
##vso[task.debug]name=Matteo
Hello Matteo
```

Author a task extension – Sign

The extension manifest contains all the information about the extension.

Ensure you've created an images folder with *extension-icon.png*.

More info on the syntax are available here: <https://docs.microsoft.com/en-us/azure/devops/extend/develop/manifest?view=azure-devops>

```
{
  "manifestVersion": 1,
  "id": "build-release-task",
  "name": "Fabrikam Build and Release Tools",
  "version": "0.0.1",
  "publisher": "fabrikam",
  "targets": [ { "id": "Microsoft.VisualStudio.Services" } ],
  "description": "Tools for building/releasing with Fabrikam. Includes one build/release task.",
  "categories": [ "Azure Pipelines" ],
  "icons": {
    "default": "images/extension-icon.png"
  },
  "files": [ { "path": "task" } ],
  "contributions": [
    {
      "id": "custom-build-release-task",
      "type": "ms.vss-distributed-task.task",
      "targets": [
        "ms.vss-distributed-task.tasks"
      ],
      "properties": {
        "name": "buildAndReleaseTask"
      }
    }
  ]
}
```

Author a task extension - Distribute

All extensions are packaged as VSIX 2.0-compatible files

We can use the *tfx-cli* that Microsoft provides (works cross-platform) to package the extension in a VSIX format

Once the extension is ready, it can be published in the marketplace and then added as part of the organization for private testing

```
tfx extension create --manifest-globs vss-extension.json  
tfx extension publish --manifest-globs your-manifest.json --share-with {{organization}}
```

Should I use task extensions?

Pros

- Good UI and UX for end-user who are just getting started and/or want a unified experience built into the product
- Complete ALM under Azure DevOps with pipelines, tests and so on...
- Documentation is being renewed and contains references also to the UI components

Cons

- Even if we have CLI, it's a bit hard to get started
- You (probably) need to learn a lot of new tools and languages
- Doesn't work on GitHub

What about GitHub?

GitHub and Azure DevOps are both under heavy development in Microsoft and are both supported and suggested to get started with

- More info here: [ReBuild 2020 Live - Online | conferenza | ASPItalia.com](https://rebuild2020.com/conferenza/aspitalia.com)

GitHub has a different extension method that supports:

- JavaScript actions that run plain on the agent and represent the most similar option compared to what is available in Azure DevOps
- Dockerized actions that run in a container, slower compared to pure JS actions
- Composite actions that allow you to combine multiple workflow steps within one action

There is no way to convert a task extension into a custom action

.NET tools to the rescue

A .NET tool is a special NuGet package that contains an entire console application written in .NET

Why a .NET tool?

- Fits perfectly in between the scripts and task extensions
- You can choose your own language that is part of the .NET family
- No UI and no complexity in getting started
- Tasks can work on any OS given .NET 5 is cross-platform (since .NET Core)
- Takes advantage of CI/CD system for ALM and versioning as it's a "classic" console app
- Azure DevOps? Yes. GitHub? Yes. What about Jenkins? YES!

How to get started?

On the authoring side:

1. Write a console app
2. Extend it to be a .NET tool
3. Distribute it on NuGet (or similar)

On the consumer side:

1. Use [dotnet search](#) or [ToolGet](#) to search for packages (optional)
2. Install the tool using `dotnet tool install -g {{ package-name }}`
3. Invoke the tool using `{{ package-name }}` like a "classic" script

Demo

Create a .NET Tool and use it
in Azure DevOps

.NET Conference
Italia 2020



.NET

Use the task

Azure DevOps

```
# Install the .NET tool
- script: dotnet tool install --tool-path tools Hello --version 1.0.0
  displayName: 'Install tool'
  env:
    DOTNET_SKIP_FIRST_TIME_EXPERIENCE: true
    DOTNET_CLI_TELEMETRY_OPTOUT: true

# When the tool is installed, then we can use it
- pwsh: ./tools/hello-world-aspitalia ${{ parameters.speaker }}
  displayName: 'Print hello'
```

GitHub

```
# Install the .NET tool
- name: Install tool
  run: dotnet tool install --tool-path tools Hello --version 1.0.0

# When the tool is installed, then we can use it
- name: Print hello
  run: ./tools/hello-world-aspitalia $speaker
  env:
    SPEAKER: "Matteo"
```

Recap

	Scripts	Extensions	.NET Tools
Works on GitHub	Yes without rewriting	No requires rewrite	Yes without rewriting
Has UI interface	No	Yes	No
Flexibility	No You must choose a language dependent on OS for script and tests	No You must do whatever Azure DevOps/GitHub requires	Yes Any console app based on .NET will simply work
Easy to start with	Yes If you are skilled in writing scripts (Pwsh, Bash...)	No Even if you are experienced, there are a lot of prerequisites	Yes If you are skilled in .NET
Can reuse existing code	Yes	Possible, but difficult	Yes
Easy to test	No Depends on the language and on the framework (Pester?)	No Unit tests? Yes. Integration tests? You must install the extension. UI tests? No	Yes Can use any preferred framework (xUnit, NUnit...)
Language agnostic	No There are limits in PowerShell vs. Pwsh Core and so on...	No Must know JS or PowerShell	Yes (99%) Any console app written in .NET (C#, VB, F#...)
Documentation	Yes Depends on the language	Yes developer.microsoft.com/en-us/azure-devops	Yes docs.microsoft.com

Resources

Create .NET Tools

- <https://docs.microsoft.com/en-us/dotnet/core/tools/global-tools>
- <https://docs.microsoft.com/en-us/dotnet/core/tools/global-tools-how-to-create>

Create Azure DevOps custom tasks/extensions

- <https://docs.microsoft.com/en-us/azure/devops/extend/develop/add-build-task?view=azure-devops#optional-install-and-test-your-extension>

Create GitHub custom tasks

- <https://docs.github.com/en/free-pro-team@latest/actions/creating-actions>

Demo

- <https://github.com/aspitalia/net-conf-2020>

@xtumiox
matteot@aspitalia.com

Slide e materiale su
<https://aspit.co/netconfit-20>

.NET Conference
Italia 2020

.NET