@criticalmanufacturing/ cli Documentation

Table of contents

1. About cmf-cli

cmf-cli is a Command Line Interface used for Critical Manufacturing developments.

1.1 Use cmf-cli to . . .

- Scaffold a new repository
- Generate new package structures
- Adapt packages of code for Critical Manufacturing MES
- Manage multiple versions of packages and package dependencies
- Create packages that can be used by any developer or customer
- View the package tree
- Restore packages for local development
- Assemble a release bundle

and a lot more!

1.2 Getting started

To get started with cmf-cli, you need to use the command line interface (CLI) to install cmf-cli. We look forward to seeing what you create!

2. Installation

To be able to install cfm-cli, you must install Node.js and the npm command line interface using either a Node version manager or a Node installer. **We strongly recommend using a Node version manager like nvm to install Node.js and npm.** We do not recommend using a Node installer, since the Node installation process installs npm in a directory with local permissions and can cause permissions errors when you run npm packages globally.

```
npm install --global @criticalmanufacturing/cli
```

2.1 Checking your version of cmf-cli and Node.js

To see if you already have Node.js and npm installed and check the installed version, run the following commands:

```
node -v
cmf -v
```

3. Plugins

The Critical Manufacturing cli is designed with a plugin system for extensibility. In the future, it will be possible to search for plugins straight from cli. Check issue #11 for progress.

In the meanwhile, some plugins are already in development. Here follows a non-exaustive plugin list:

• Portal SDK - command line tools to interact with the Critical Manufacturing Customer Portal

4. Commands

4.1 assemble

4.1.1 Usage

```
cmf assemble [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default: .]

Options

Name	Description
`-o, --outputDir <outputDir>`	Output directory for assembled package [default: Assemble]
`--cirepo <cirepo>`	Repository where Continuous Integration packages are located (url or folder)
`-r, --repo, --repos <repos>`	Repository or repositories where published dependencies are located (url or folder)
`--includeTestPackages`	Include test packages on assemble
`-?, -h, --help`	Show help and usage information

4.2 build

4.2.1 Usage

```
cmf build [options] [<packagePath>] [command]
```

Arguments

Name	Description
`<packagePath>`	Package Path [default: .]

Options

Name	Description
`-?, -h, --help`	Show help and usage information

Commands

Name	Description
`help`	

4.3 build help

4.3.1 Usage

```
cmf build [<packagePath>] help [options] [command]
```

Arguments

Name	Description
`<packagePath>`	Package Path [default: .]

Options

Name	Description
`-?, -h, --help`	Show help and usage information

Commands

Name	Description
`generateBasedOnTemplates`	
`generateMenuItems`	

4.4 build help generateBasedOnTemplates

4.4.1 Usage

```
cmf build [<packagePath>] help generateBasedOnTemplates [options]
```

Options

Name	Description
-?, -h, --help	Show help and usage information

4.5 build help generateMenuItems

4.5.1 Usage

```
cmf build [<packagePath>] help generateMenuItems [options]
```

Options

Name	Description
-?, -h, --help	Show help and usage information

4.6 bump

4.6.1 Usage

```
cmf bump [options] [<packagePath>] [command]
```

Arguments

Name	Description
`<packagePath>`	Package path [default: .]

Options

Name	Description
`-v, --version <version>`	Will bump all versions to the version specified
`-b, --buildNr <buildNr>`	Will add this version next to the version (v-b)
`-r, --root <root>`	Will bump only versions under a specific root folder (i.e. 1.0.0)
`-?, -h, --help`	Show help and usage information

Commands

Name	Description
`iot`	

4.7 bump iot

4.7.1 Usage

```
cmf bump [<packagePath>] iot [options] [command]
```

Arguments

Name	Description
`<packagePath>`	Package path [default: .]

Options

Name	Description
`-?, -h, --help`	Show help and usage information

Commands

Name	Description
`configuration <path>`	[default: .]
`customization <packagePath>`	[default: .]

4.8 bump iot configuration

4.8.1 Usage

```
cmf bump [<packagePath>] iot configuration [options] [<path>]
```

Arguments

Name	Description
`<path>`	Working Directory [default: .]

Options

Name	Description
`-v, --version <version>`	Will bump all versions to the version specified
`-b, --buildNrVersion <buildNrVersion>`	Will add this version next to the version (v-b)
`-md, --masterData`	Will bump IoT MasterData version (only applies to .json) [default: False]
`-iot`	Will bump IoT Automation Workflows [default: True]
`-pckNames, --packageNames <packageNames>`	Packages to be bumped
`-r, --root <root>`	Specify root to specify version where we want to apply the bump
`-g, --group <group>`	Group of workflows to change, typically they are grouped by Automation Manager
`-wkflName, --workflowName <workflowName>`	Specific workflow to be bumped
`-isToTag`	Instead of replacing the version will add -$version [default: False]
`-mdCustomization`	Instead of replacing the version will add -$version [default: False]
`-?, -h, --help`	Show help and usage information

4.9 bump iot customization

4.9.1 Usage

```
cmf bump [<packagePath>] iot customization [options] [<packagePath>]
```

Arguments

Name	Description
`<packagePath>`	Package Path [default: .]

Options

Name	Description
`-v, --version <version>`	Will bump all versions to the version specified
`-b, --buildNrVersion <buildNrVersion>`	Will add this version next to the version (v-b)
`-pckNames, --packageNames <packageNames>`	Packages to be bumped
`-isToTag`	Instead of replacing the version will add -$version [default: False]
`-?, -h, --help`	Show help and usage information

4.10 init

4.10.1 Usage

```
cmf init [options] <projectName> [<rootPackageName> [<workingDir>]]
```

Arguments

Name	Description
`<projectName>`	
`<rootPackageName>`	[default: Cmf.Custom.Package]
`<workingDir>`	Working Directory [default: .]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-c, --config <config>`	Configuration file exported from Setup [default:]
`--repositoryUrl <repositoryUrl>`	Git repository URL
`--deploymentDir <deploymentDir>`	Deployments directory (for releases). Don't specify if not using CI-Release. [default:]
`--MESVersion <MESVersion>`	Target MES version
`--DevTasksVersion <DevTasksVersion>`	Critical Manufacturing dev-tasks version
`--HTMLStarterVersion <HTMLStarterVersion>`	HTML Starter version
`--yoGeneratorVersion <yoGeneratorVersion>`	@criticalmanufacturing/html Yeoman generator version
`--nugetVersion <nugetVersion>`	NuGet versions to target. This is usually the MES version
`--testScenariosNugetVersion <testScenariosNugetVersion>`	Test Scenarios Nuget Version
`--infra, --infrastructure <infrastructure>`	Infrastructure JSON file [default:]
`--nugetRegistry <nugetRegistry>`	NuGet registry that contains the MES packages
`--npmRegistry <npmRegistry>`	NPM registry that contains the MES packages
`--azureDevOpsCollectionUrl <azureDevOpsCollectionUrl>`	The Azure DevOps collection address
`--agentPool <agentPool>`	Azure DevOps agent pool
`--agentType <Cloud|Hosted>`	Type of Azure DevOps agents: Cloud or Hosted
`--ISOLocation <ISOLocation>`	MES ISO file [default:]
`--nugetRegistryUsername <nugetRegistryUsername>`	NuGet registry username
`--nugetRegistryPassword <nugetRegistryPassword>`	NuGet registry password
`--releaseCustomerEnvironment <releaseCustomerEnvironment>`	Customer Environment Name defined in DevOpsCenter
`--releaseSite <releaseSite>`	Site defined in DevOpsCenter
`--releaseDeploymentPackage <releaseDeploymentPackage>`	DeploymentPackage defined in DevOpsCenter
`--releaseLicense <releaseLicense>`	License defined in DevOpsCenter
`--releaseDeploymentTarget <releaseDeploymentTarget>`	DeploymentTarget defined in DevOpsCenter
`-?, -h, --help`	Show help and usage information

4.11 ls

4.11.1 Usage

```
cmf ls [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default: .]

Options

Name	Description
`-r, --repo, --repos <repos>`	Repositories where dependencies are located (folder)
`-?, -h, --help`	Show help and usage information

4.12 new

4.12.1 Usage

```
cmf new [options] [command]
```

Options

Name	Description
`--reset`	Reset template engine. Use this if after an upgrade the templates are not working correctly.
`-?, -h, --help`	Show help and usage information

Commands

Name	Description
`business <workingDir>`	[default:]
`database <workingDir>`	[default:]
`data <workingDir>`	[default:]
`feature <packageName> <workingDir>`	[default:]
`help <workingDir>`	[default:]
`html <workingDir>`	[default:]
`iot <workingDir>`	[default:]
`test`	

4.13 new business

4.13.1 Usage

```
cmf new business [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-?, -h, --help`	Show help and usage information

4.14 new data

4.14.1 Usage

```
cmf new data [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`--businessPackage <businessPackage>`	Business package where the Process Rules project should be built [default:]
`-?, -h, --help`	Show help and usage information

4.15 new database

4.15.1 Usage

```
cmf new database [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-?, -h, --help`	Show help and usage information

4.16 new feature

4.16.1 Usage

```
cmf new feature [options] <packageName> [<workingDir>]
```

Arguments

Name	Description
`<packageName>`	The Feature package name
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-?, -h, --help`	Show help and usage information

4.17 new help

4.17.1 Usage

```
cmf new help [options] [<workingDir>] [command]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`--docPkg, --documentationPackage <documentationPackage> (REQUIRED)`	Path to the MES documentation package
`-?, -h, --help`	Show help and usage information

4.18 new html

4.18.1 Usage

```
cmf new html [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`--htmlPackage, --htmlPkg <htmlPackage>` (REQUIRED)	Path to the MES Presentation HTML package
`-?, -h, --help`	Show help and usage information

4.19 new iot

4.19.1 Usage

```
cmf new iot [options] [<workingDir>] [command]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default:]

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-?, -h, --help`	Show help and usage information

4.20 new test

4.20.1 Usage

```
cmf new test [options]
```

Options

Name	Description
`--version <version>`	Package Version [default: 1.0.0]
`-?, -h, --help`	Show help and usage information

4.21 pack

Description

cmf pack is a package creator for the CM MES developments. It puts files and folders in place so that CM Deployment Framework is able to install them.

It is extremely configurable to support a variety of use cases. Most commonly, we use it to pack the developments of CM MES customizations.

Run `cmf pack -h` to get a list of available arguments and options.

Important

cmf pack comes with preconfigured Steps per PackageType to run during the installation. This pre defined steps are assuming a restrict structure during the installation, this can be disabled using the flag `isToSetDefaultSteps:false` in your `cmfpackage.json`.

How it works

When the cmf pack is executed it will search in the working directory, for a `cmfpackage.json` file, that then is serialized to the CmfPackage this will guarantee that the `cmfpackage.json` has all the valid and needed fields. Then it will get which is the PackageType, and based on that will generate the package.

4.21.1 Usage

```
cmf pack [options] [<workingDir>]
```

Arguments

Name	Description
`<workingDir>`	Working Directory [default: .]

Options

Name	Description
`-o, --outputDir <outputDir>`	Output directory for created package [default: Package]
`-f, --force`	Overwrite all packages even if they already exists
`-?, -h, --help`	Show help and usage information

4.22 restore

Description

`cmf restore` allows fetching development dependencies from Deployment Framework (DF) packages, as an alternative to the stricter NuGet and NPM packages.

How it works

Running this command, any dependencies defined in `cmfpackage.json` will be obtained from the configured repositories (either specified via command option or registed in the `repositories.json` file) and are then unpacked to the `Dependencies` folder inside the package. Then each solution can add references/link packages from the Dependencies folder.

4.22.1 Usage

```
cmf restore [options] <packagePath>
```

Arguments

Name	Description
`<packagePath>`	Package path

Options

Name	Description
`-r, --repo, --repos <repos>`	Repositories where dependencies are located (folder)
`-?, -h, --help`	Show help and usage information

5. New

5.1 Scaffolding

5.1.1 Pre-requisites

Though `@criticalmanufacturing/cli` runs with the latest `node` version, to run scaffolding commands the versions required by the target MES version are **mandatory**.

For MES v8, the recommended versions are:

• latest node 12 (Erbium)

• latest npm 6 (should come with node)

Apart from those, scaffolding also needs the following dependencies:

```
npm install -g windows-build-tools
npm install -g gulp@3.9.1
npm install -g yo@3.1.1
```

Infrastructure

Rarely changing information, possibly sensitive, like NuGet or NPM repositories and respective access credentials are considered infrastructure. More information on how to set up your own is available at Infrastructure

Environment Config

A valid MES installation is required to initialize your repository, either installed via Setup or via DevOps Center. For the Setup: - in the final step of the Setup, click Export to obtain a JSON file with your environment settings For DevOps Center: - Open your environment and click Export Parameters

Both these files contain sensitive information, such as user accounts and authentication tokens. They need to be provided to the `init` command with:

```
cmf init --config <config file.json> --infra...
```

5.1.2 Scaffolding a repository

Let's start by cloning the empty repository.

```
git clone https://git.example/repo.git
```

Move into the repository folder

```
cd repo
```

For a classic project example, check the traditional structure documentation.

For more advanced structures, you'll probably be using Features.

5.1.3 Continuous Integration and Delivery

The scaffolding templates provide a few pipelines designed for Azure DevOps. They work both in Azure DevOps Server and Azure DevOps Services.

IMPORTANT: Only the pipelines for Pull Request and for Package generation (CI-Changes and CI-Package) are designed to run outside of Critical Manufacturing infrastructure. We currently do not support running the Continuous Delivery part of the pipelines in a client infrastructure.

The YAML files are available in the Builds folder at the repository root. Next to them are some JSON files which contain the metadata for the pipelines in Azure DevOps format, which can be used by directly invoking the Azure DevOps API. These files are in git ignore and they should **not** be committed, as they can contain secrets in plain text, such as Nuget credentials.

For CMFers: you can use an internal tool to import the pipeline metadata, as well as the branch policies. Check "How To Import Builds" in the COMMON wiki at Docs/Pipelines.

The **CD-Containers** pipeline requires a secret to be created into the Azure DevOps library. Check more details in the pipeline import document.

5.1.4 Manual Pipeline import

For non-CMFs, it's simple to import the pipelines. Check out this document.

5.1.5 External Users

There is more available information for non-CMFers at External Users.

5.2 Scaffolding and Pipelines for non-CMF Users

This document details a base approach on how to use `@criticalmanufacturing/cli` to scaffold a Product Customization solution if you are not in the CMF infrastructure. This guide assumes it is being executed in a Windows machine, but can be run in Linux with PowerShell Core if needed. Adapt as required.

5.2.1 Notes on the Local Repository

In this document, it is assumed that the user has got a copy of the necessary packages from CMF. However, it is also possible that the user simply has access to the needed repositories. If this is the case, skip the repository setup steps.

If the user is setting up their own repository, this guide is using the Sonatype Nexus v3 repo as an example. This repository provides hosting for NuGet, NPM and raw (i.e. Zip file) packages. Other repositories can be used, hosted together or not. However, it may be left to the user to determine the correct endpoints. In this document, it is assumed the local repository live at `example.local` and is an OSS Nexus v3.

For hosting Nexus, the easiest way is to use the official Docker image. The setup of the repository is not detailed in this guide but information can be found in Sonatype's help page.

The Nexus repository used in this guide needs to:

1. Have a NuGet repository. In this guide it is called `nuget-hosted`. We do not use the proxy or grouped repository types, as the public packages are directly referenced from nuget.org

2. Have an NPM repository. In this guide it is called `npm` and is of type `npm-grouped`. Unlike NuGet, here we do setup proxy and grouped repositories. The configuration is as follows:

a. Hosted npm repository `npm-hosted`: this will contain the private CMF packages

b. Proxy npm repository `npm-proxy`: this repository only proxies public package request to npmjs.org

c. Grouped npm repository `npm`: this repository is a group of the `npm-hosted` and `npm-proxy` repositories.

d. In Settings > Realms, have NPM Bearer Token and NuGet API Key realms active. The API Key can be obtained from the user profile.

Authentication

By default, NuGet and NPM repositories operate with anonymous access, requiring only credentials to publish. If this does not meet your requirements, you can opt to require authentication to all feeds.

NUGET

1. In Nexus, go to Settings > Security > Anonymous Access and disable anonymous access.

2. When running `init`, provide an infrastructure file that contains the credentials like so:

"NuGetRegistryUsername": "", "NuGetRegistryPassword": ""

If provided, the NuGet restore calls will be authenticated. Note that currently there is no way to do the restore using NuGet ApiKeys.

NPM

1. login to your NPM repo with the build account (This is also done when publishing the packages but for the hosted repo. In this case the grouped repo must be used):

   ```
   npm adduser --registry=http://example.local/repository/npm/
   ```

2. in the .npmrc file in your home folder, a new line should be added, in the format `//example.local/repository/npm/:_authToken=NpmToken.<GUID>`

3. take this line and add it to the .npmrc in both the Help and HTML solutions. The files are always in the `apps/<web solution>` path

4. commit these files

   > NOTE: this file will be visible to anyone with source code access! If push permissions are not desired here, use an account that can only download packages!

   In some cases `npm adduser` may fail. A known case is when using Personal Access Tokens (PATs) instead of passwords in Windows systems. In these cases, it's possible to directly use the credentials without this authentication step:

1. edit the `.npmrc` file in the HTML and Help package directories with your text editor

2. fill in the repository: `registry=https://example.io/repository/npm/`

3. fill in the credentials. The `_auth` string is obtained by doing a Base 64 encode of `<user>:<password>` with the respective username and password of the repository account: `//example.io/repository/npm/:_auth="<base64 string>"`

4. tell NPM to always authenticate: `//example.io/repository/npm/:always_auth=true`

   So if your username is `user1` and your password is `secret2` the .npmrc files would look like

   ```
   registry=https://example.io/repository/npm/
   //example.io/repository/npm/:_auth="dQBzAGUAcgAxADoAcwBlAGMAcgBlAHQAMgA="
   //example.io/repository/npm/:always_auth=true
   ```

   where the auth string can be generated in Powershell using

   ```
   [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes("user1:secret2"))
   ```

5.2.2 Process

Host packages in local repository

If using a local repository, the dependency packages need to be hosted there. CMF will have provided a package with the NuGet packages. The NPM packages are extracted from the MES ISO file. The instructions are available either on your Help website or in CMF's general Help repository.

However, currently the script provided in the Help tutorial only uploads the HTML solution dependencies, not the Help solution ones. As such, an updated copy of the script can be provided if requested.

It is recommended that you follow the Help tutorial, but a few quick steps will be detailed here

1. Authenticate to your local repo:

```
npm adduser --registry=http://example.local/repository/npm-hosted/
```

a. NOTE: If using Nexus OSS, you will have to upload to the `npm-hosted` repo, not to `npm`

2. Run

You only need to specify the Username if you don't have anonymous access to your repository

```
.\LoadNPMPackagesToLocalRepository.ps1 -PathToISO "<ISO path>" "http://example.local/repository/npm-hosted/" -ExtractionFolder "C:\_CMTemp\packages" [-
Username <User>]
```

a. the script will automatically mount the ISO, extract the packages and upload them to the repository

3. OPTIONAL: tag the loaded packages with the dist-tags used by the solutions. If this is not desired, the scaffolded solutions will have to be changed to point to the exact versions that are uploaded into the repository. Adding the dist-tags is generally easier. The dist-tags always follow the format `release-<version>` with no dots. As an example, for MES version 8.1.0 the dist-tag commands would be:

```
npm dist-tag add @criticalmanufacturing/mes-ui-web@8.1.0-202103302 release-810 --registry=http://example.local/repository/npm-hosted/

npm dist-tag add cmf.docs.web@8.1.0-20210329.4 release-810 --registry=http://example.local/repository/npm-hosted/
```

5.2.3 Loading the scaffolding into the target project

Using a local repository

If using a local repository, the first step is to load the NuGet dependencies. Until this is done the Business and Test solutions will not compile.

Place the packages in a folder. Open a powershell session in `Tools` and run the `Deploy_LoadAllNuGetPackages.ps1` script. Don't forget to authenticate first.

```
cd <folder>
nuget setapikey <nuget key> -source http://example.local/repository/nuget-hosted

# for each nuget file:
nuget push <file.nupkg> -source http://local.example/repository/nuget-hosted/
```

Commiting the code

If you did not run `init` and `new` inside a cloned git repository folder, you can add it as a remote in place. For the second approach, open a PS session where you ran the `init` and run:

```
git remote add origin https://<user>@dev.azure.com/<organization>/<project>/_git/<repository>
git push -u origin --all
```

Loading pipelines and policies

To run the pipelines, some tasks need to be installed in Azure DevOps. These are:

- MSPremier.PostBuildCleanup.PostBuildCleanup-task.PostBuildCleanup
- qetza.replacetokens

You can now load the Pipelines and Policies into your project.

5.3 Feature package scaffolding

A traditional project structure targets the scenario where one team does all of the development for a project with a single target, e.g. a customization team deploying MES to a single factory or a set of factories with the exact same customization.

However, some projects are more complex. A single team can target a multi-site deployment in which the sites target a set of common features, but complement them with site-specific features, or the common features need site-specific data packages to configure them.

In these cases, projects are usually structured in a **Feature Packages** arrangement, in that some packages are self-contained, or in a **Layered Project** arrangement, in which e.g. a Site package, which is specific to a given site/plant, depends on a Core/Baseline package, which is common among all sites.

`cmf new` allows assembling these package in whatever structure is necessary, with a few limitations: - the test solution is always at the repository level, i.e. we do not have feature-level test packages - you cannot mix traditional packages, which exist in the repository root, with these packages - a feature package must always exist inside a global repository

`cmf new` will automatically register each package as a dependency of its parent (as per the folder structure).

5.3.1 Initialize the Repository

In case we are creating the first package in the repository, we need to initialize it in the same way as we do for a traditional project. Check the **Initialize the repository** section of the traditional scaffolding instructions.

5.3.2 Creating a Feature Package

A **Feature Package** is a metapackage that can include one or more layer packages, e.g. we can have a Feature package that includes only a Business package.

A Feature package should include all necessary layer packages (business, UI, Help) needed for the Feature to work.

To create a new Feature Package, run the `new feature` command at your repository root, specifying the new package name: e.g. to create the Baseline package for your project, run:

```
cmf new feature Cmf.Custom.Baseline
```

The new feature package will be available in your repository at `Features\Cmf.Custom.Baseline`.

After creating a Feature package, it is not longer possible to create new layer packages in the repository root. Any `new` command for a layer package will fail:

```
> cmf new html
  Cannot create a root-level layer package when features already exist.
```

5.3.3 Creating Layer packages in a Feature package

Creating layer packages works exactly in the same way as for a traditional project, but the `cmf new` command should run from inside the feature package:

```
> cd Features/Cmf.Custom.Baseline
> cmf new business
The template "Business Package" was created successfully.
```

The layer package name will include the feature package name fragment to distinguish it from other packages in the same repository.

5.4 Infrastructure config file

5.4.1 Structure

Information regarding repositories and Azure projects usually don't change very often. For this set of information, `init` accepts a file with a few keys that specify:

1. The NPM repository URL

2. The NuGet repository URL

3. The Azure DevOps collection URL

4. The Azure DevOps project name

5. The type of Azure build agents used, Cloud or Hosted

a. Cloud means Microsoft hosted agents with can run a multitude of VMs

b. Hosted means self-hosted agents. Using self-hosted, for now, requires Windows agents with a set of pre-requisites installed.

5.4.2 Usage

The infrastructure file must be passed to the `init` command as an argument, e.g.:

```
cmf init --infrastructure <file> --config...
```

5.4.3 Example

```
{
    "ISOLocation": "path/to/MES/isos",
    "NPMRegistry": "http://local.example/repository/npm",
    "NuGetRegistry": "https://local.example/repository/nuget-hosted",
    "NuGetRegistryUsername": "user",
    "NuGetRegistryPassword": "password",
    "AzureDevopsCollectionURL": "https://azure.example.com/Projects",
    "AgentPool": "Projects",
    "AgentType": "Cloud"
}
```

5.5 Pipeline Import

To import the Continuous Integration pipelines, first make sure you have scaffolded your project successfully and can see a folder in the project root named Builds with several YAML files in it:
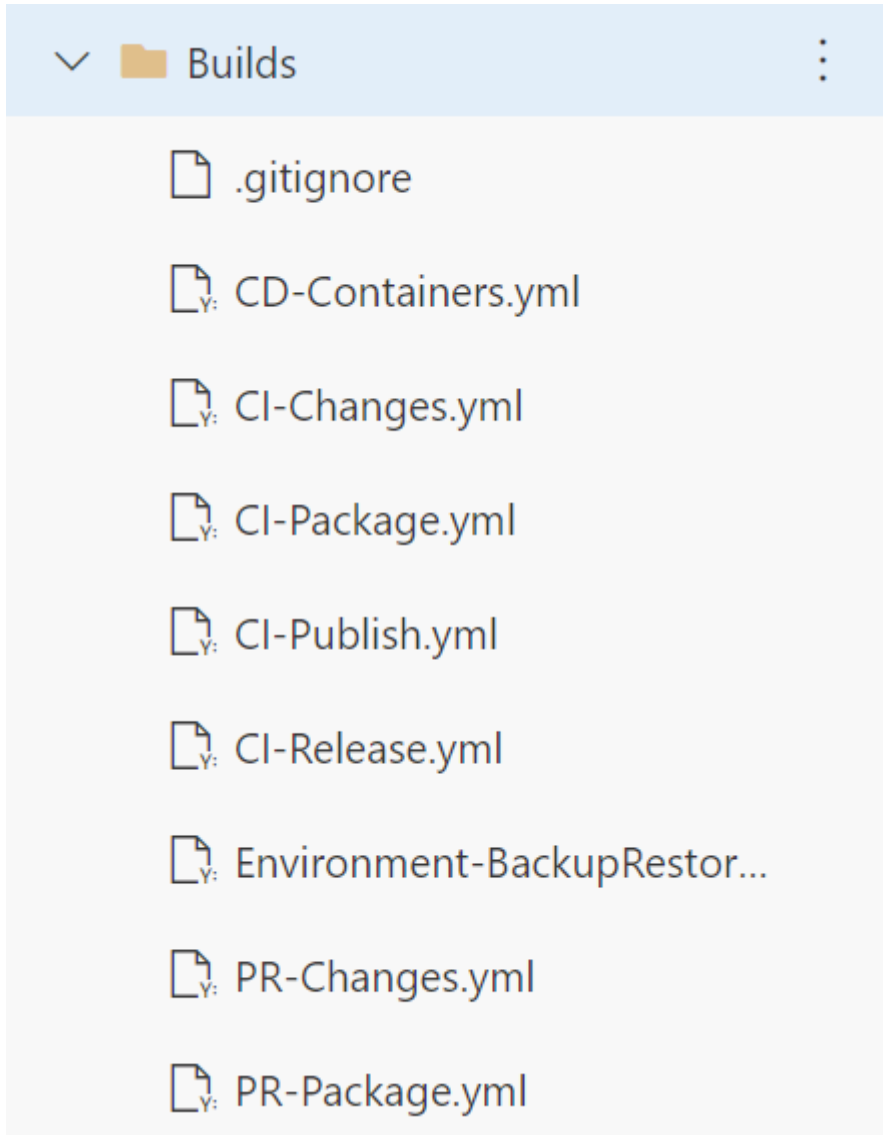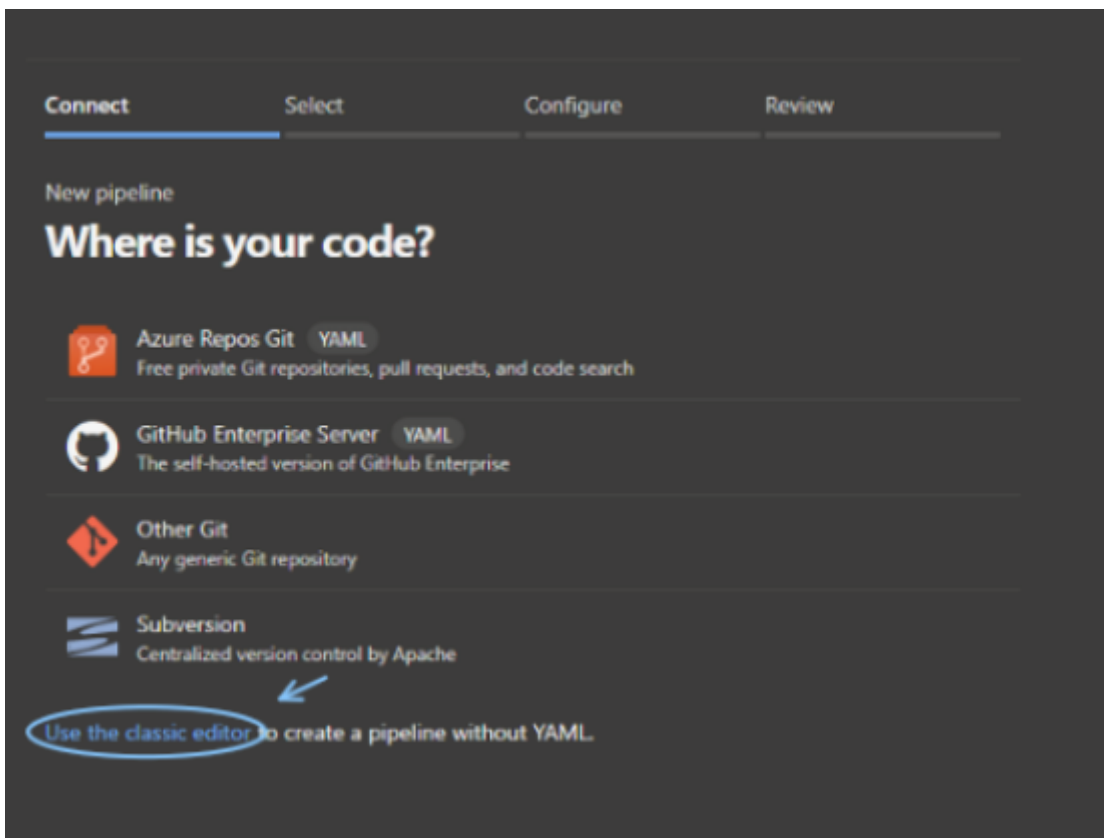

If so, we can proceed. Push your repository if you haven't yet. It is important that the default branch contains the pipeline YAML files. In our setup, the default branch is always **development**.

For each of **PR-Changes**, **PR-Package**, **CI-Changes** and **CI-Package**, repeat these steps:
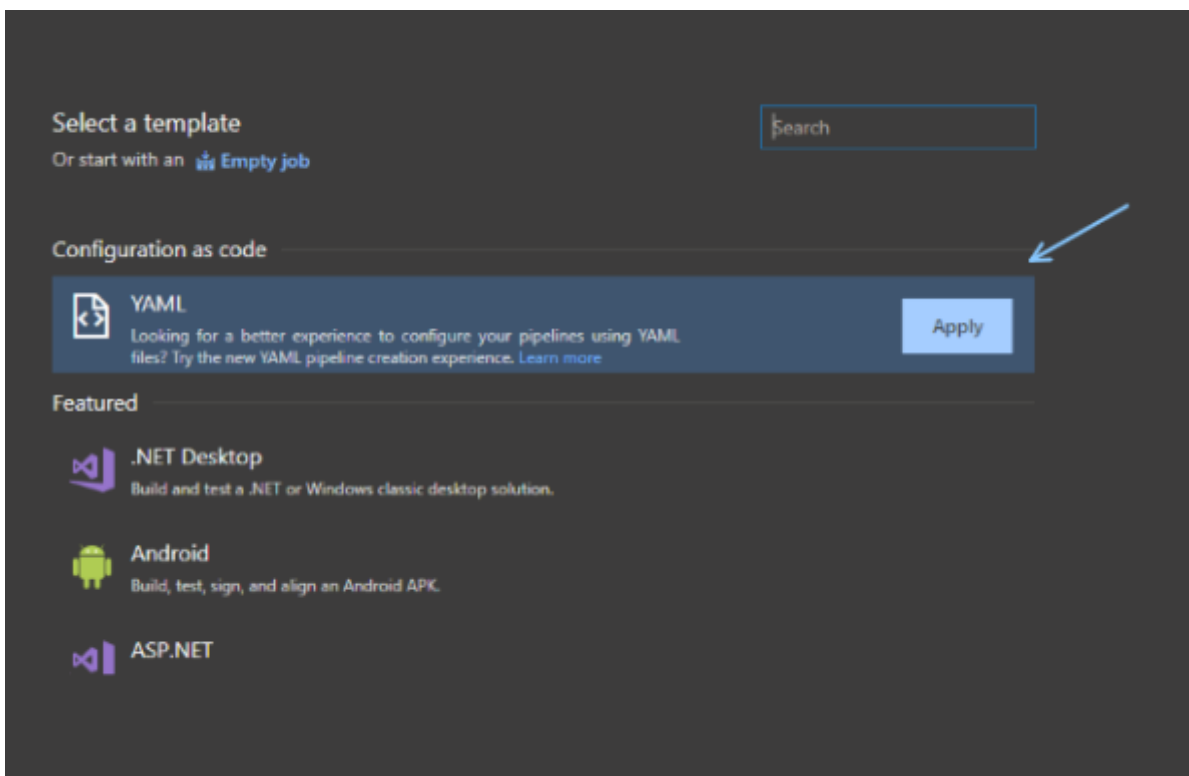
create a new pipeline

1.
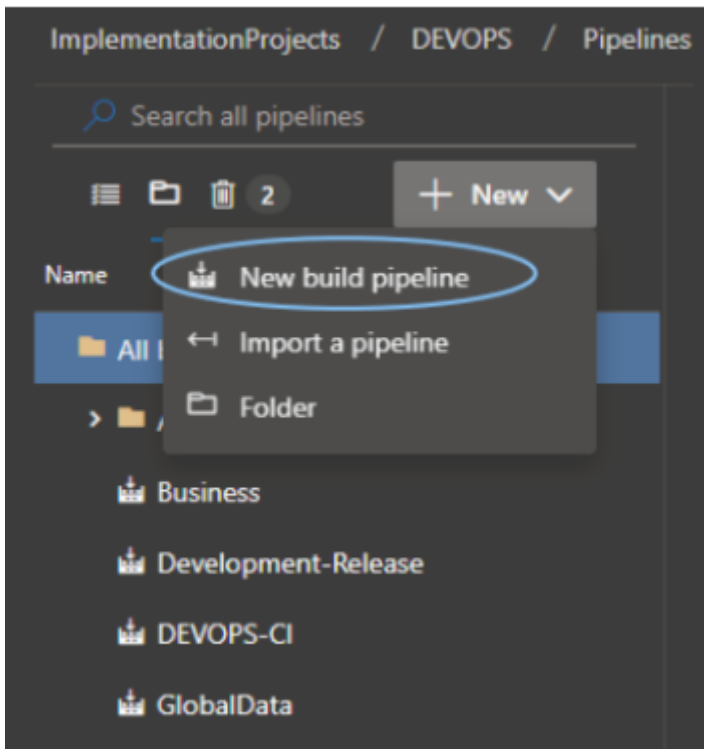
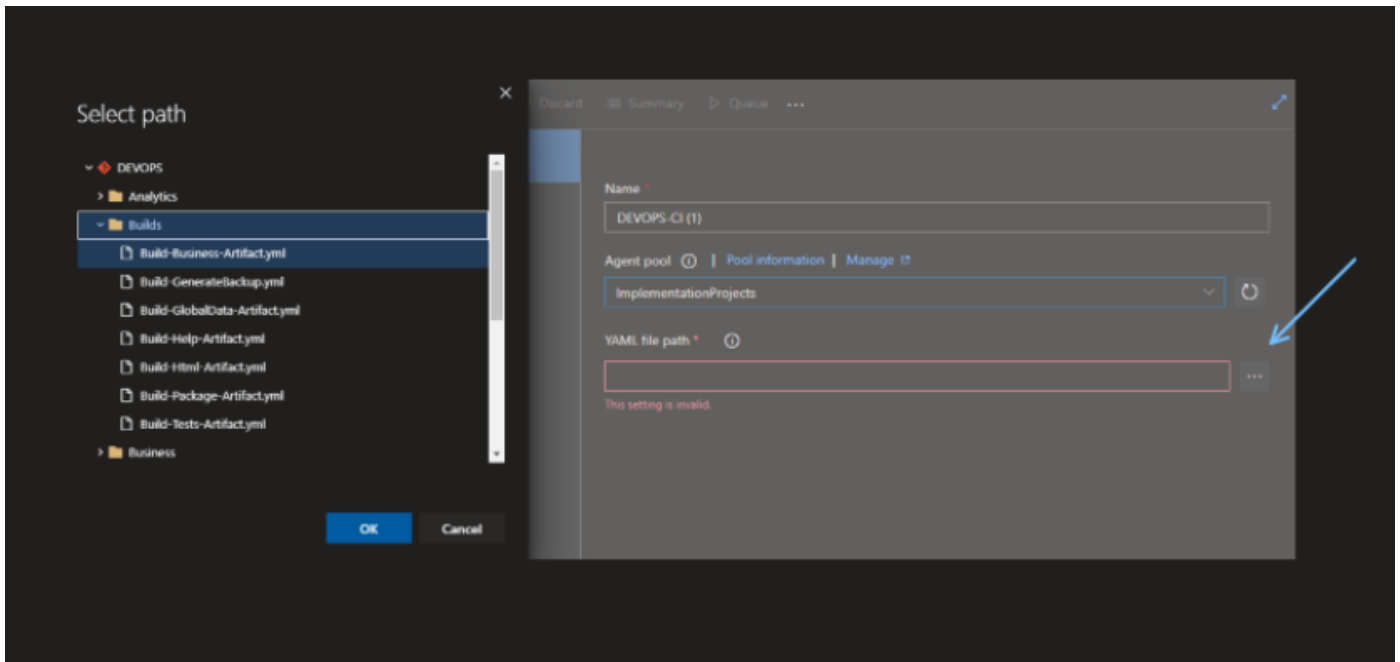2. Choose **Use the classic editor**

3. Choose your Team Project, repository and default branch. Our generated pipelines are expecting the default branch to be
   development
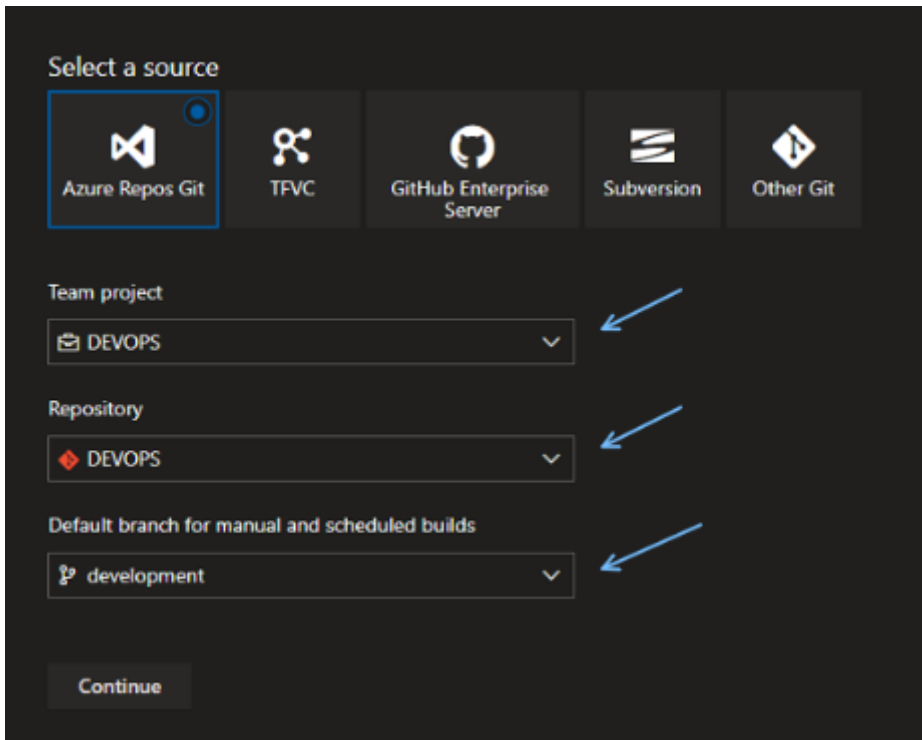
4. After that, choose Configuration as code > YAML

5. and choose the corresponding YAML file in the Builds folder

2009-present Critical Manufacturing SA

5.5.1 Pull Request policies

To effectively leverage the Pull Request flow our pipelines enable, Azure DevOps must be configured to protect the default (in our case, **development**) branch.

Go to Repos > Branches, on the **development** row, click the More button on the far right and select Branch Policies
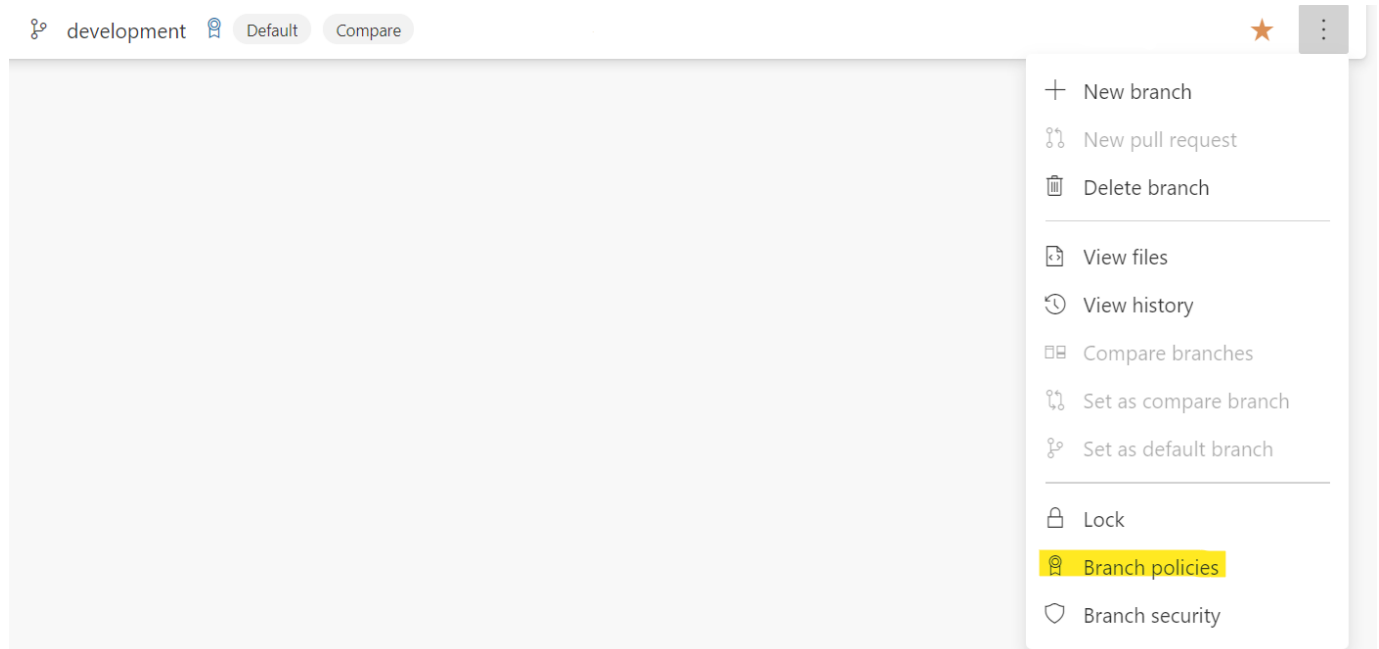
The policy options per-se are dependent on the team configuration and working mode. Here is an example of our policies

Protect this branch

- Setting a Required policy will enforce the use of pull requests when updating the branch
- Setting a Required policy will prevent branch deletion
- Manage permissions for this branch on the Security page

☑ **Require a minimum number of reviewers**
Require approval from a specified number of reviewers on pull requests.

Minimum number of reviewers   1

☐ Allow users to approve their own changes.

☐ Allow completion even if some reviewers vote "Waiting" or "Reject".

☐ Reset code reviewer votes when there are new changes.

☑ **Check for linked work items**
Encourage traceability by checking for linked work items on pull requests.

Policy requirement

⦿ Required
Block pull requests from being completed unless they have at least one linked work item.

◯ Optional
Warn if there are no linked work items, but allow pull requests to be completed.

☑ **Check for comment resolution**
Check to see that all comments have been resolved on pull requests.

Policy requirement

⦿ Required
Block pull requests from being completed while any comments are active.

◯ Optional
Warn if any comments are active, but allow pull requests to be completed.

☐ **Limit merge types**
Control branch history by limiting the available types of merge when pull requests are completed.

The important point here is to set the **PR-Changes** pipeline to run whenever a Pull Request is opened. To do so, add a new row in **Build validation** that looks like this

Build pipeline	Requirement	Path filter	Expiration	Trigger	
🏗 PR-Changes	Required	No filter	Strict expiration	Automatic	⬤ Enabled

This will ensure the PR pipelines will run each time code is pushed to a Pull Request, and will block merging if the build is failing.

5.5.2 Remaining pipelines

We do not support running the Continuous Delivery pipelines in a client setting. These pipelines are strongly opinionated on the environment in which they run and require virtual machines configured specifically for them. They are still provided as an example and guide so that each client or partner can create their own pipelines. However, they will not work out of the box and will hang on permissions issues if attempted to run.

5.5.3 Secrets

The **CD-Containers** pipeline requires that a secret is specified in a variable group to run successfully. The most important one is the Customer Portal PAT (Personal Access Token).

Library > 🔳 Docker Variables*

Variable group | 💾 Save | 📋 Clone | 🛡 Security | ❓ Help
6

Properties

Variable group name
Docker Variables   2

Description

🔵 Allow access to all pipelines   3
⚫ Link secrets from an Azure key vault as variables ⓘ

Variables

Name ↑		Value	🔒
CustomerPortalPAT	🗑	myPAT	🔒
+ Add		4	5

1. In Azure DevOps, go to Pipelines > Library and create a new Variable Group

2. Name the variable group "Docker Variables"

3. Check "Allow access to all pipelines"

4. Add a new variable named **CustomerPortalPAT** and paste your Customer Portal PAT as the value

5. Click the lock icon to turn the variable into a secret

6. Save the variable group

The pipeline will automatically use the variable group.

It is possible to add extra secrets or varibles into this group, to preserve some secrets from the environment settings. Any token in the format e.g. `#{aToken}#` will be replaced with the value of the **aToken** variable.

5.6 Post-scaffolding package tailoring

The packages generated by `cmf new` are as neutral as possible, so to be compatible with as many deployment scenarios as possible. However, some tailoring is advised for specific targets. This tailoring allows the packages to better adapt to their deployment target, eliminating the need of manual steps when installing.

Below are some tailoring options for specific targets. It's recommended that these changes are applied according to your target environment.

5.6.1 Containers:

- **Root Package**
- Add to dependencies

```
"dependencies": [
    { "id": "Cmf.Environment", "version": "8.3.0" }
]
```

- **Business Package**
- Add to steps:

```
"steps": [
    { "order": "1", "type": "DeployFiles", "ContentPath": "**/!(Cmf.Custom.*.BusinessObjects*).dll" }
]
```

5.6.2 Deployment Framework:

- **Root and IoT Root Package**
- Add to dependencies

```
"dependencies": [
    { "id": "CriticalManufacturing.DeploymentMetadata", "version": "8.3.0" }
]
```

- **Business Package**
- Add to steps:

```
"steps": [
    { "order": "1", "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/stop_host.ps1" },
    { "order": "2", "type": "DeployFiles", "ContentPath": "**/!(Cmf.Custom.*.BusinessObjects*).dll" },
    { "order": "3", "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/start_host.ps1" }
]
```

- **Data, IoTData and Tests MasterData Package**
- Add to steps:

```
"steps": [
    { "order": "1", "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/stop_host.ps1" },
    { "order": "2", "type": "TransformFile", "file": "Cmf.Foundation.Services.HostService.dll.config", "tagFile": true },
    { "order": "3", "type": "Generic", "onExecute": "$(Agent.Root)/agent/scripts/start_host.ps1" },
    { "order": "4", "type": "Generic", "onExecute": "$(Package[Cmf.Custom.Package].TargetDirectory)/GenerateLBOs.ps1" }
]
```

NOTE: Make sure that the order of the steps is the same referenced in this document.

5.7 Traditional scaffolding

A "traditional project" does not contain feature packages, is developed entirely by one team in one repository and is delivered directly to one customer.

These projects are usually composed of Business, UI, Help and Master Data customization, with optionally Exported Objects and IoT.

The objective is to obtain a structure equivalent to what `solgen` provided.

Please consult each commands help page for details of what each switch does.

5.7.1 Initialize the repository

These types of projects usually fully own their git repository and as such need to be initialized to obtain the base repo structure, as well as the build pipelines if we are targeting Azure DevOps.

This is done with the `cmf init` command:

Classic   **Containers**

```
cmf init Example `
    --infra ..\COMMON\infrastructure\CMF-internal.json `
    -c Example.json `
    --repositoryUrl https://tfs.criticalmanufacturing.com/Projects/Test/_git/Test `
    --MESVersion 8.2.1 `
    --DevTasksVersion 8.2.0 `
    --HTMLStarterVersion 8.0.0 `
    --yoGeneratorVersion 8.1.1 `
    --nugetVersion 8.2.1 `
    --testScenariosNugetVersion 8.2.1 `
    --deploymentDir \\vm-project.criticalmanufacturing.com\Deployments `
    --ISOLocation \\setups\CriticalManufacturing.iso `
    --version 1.0.0

cmf init Example `
    --infra ..\COMMON\infrastructure\CMF-internal.json `
    -c Example.json `
    --repositoryUrl https://tfs.criticalmanufacturing.com/Projects/Test/_git/Test `
    --MESVersion 8.2.1 `
    --DevTasksVersion 8.2.0 `
    --HTMLStarterVersion 8.0.0 `
    --yoGeneratorVersion 8.1.1 `
    --nugetVersion 8.2.1 `
    --testScenariosNugetVersion 8.2.1 `
    --deploymentDir \\vm-project.criticalmanufacturing.com\Deployments `
    --ISOLocation \\setups\CriticalManufacturing.iso `
    --version 1.0.0 `
    --releaseCustomerEnvironment EnvironmentName `
    --releaseSite EnvironmentSite `
    --releaseDeploymentPackage \@criticalmanufacturing\mes:8.3.1 `
    --releaseLicense EnvironmentLicense `
    --releaseDeploymentTarget EnvironmentTarget
```

Note: The `` ` `` character escapes multiline commands in `powershell`. For bash, the `\` character does the same thing.

The infrastructure file specifies the repositories to be used to get the project dependencies. If you are scaffolding a Deployment Services project, there is a CMF-internal.json infra file which specifies our internal infrastructure. It's in the **COMMON** project, **Tools** repository, at `/infrastructure`. If scaffolding a customer or partner project, you will need to create this infrastructure file first. Check Infrastructure for more details.

As in previous scenarios, the versions for the various input options must be previously determined. Unlike with `solgen`, `cmf init` will not assume default/current values for these options.

This will also create a root package, which may or may not be shipped to the customer. Unlike with `solgen`, this root package has no dependencies, initially. Each time a layer package is created, it will be registered in the higher level package found. For a traditional repository, this will be the root package.

After this, follow the instructions defined in the Post-scaffolding package tailoring. You will not be able to generate the layer packages before doing this.

5.7.2 Layer packages

Each application layer is deployed in a different package. This allows the team to deliver only what was actively modified during a sprint, and keep the previous versions of the unchanged layers in an installed system.

Business

The business package is straightforward and is generated with the `cmf new business` command:

```
cmf new business --version 1.0.0
```

This creates a .NET solution for backend development. Please note that unlike with `solgen`, the Actions project is not included in the business solution.

Master Data

The Master Data package includes also the Exported Objects. Unlike in a `solgen` solution, Exported Objects are loaded via Master Data and not using a specific ExportedObjects sub-package.

As the Master Data package also includes the Process Rules, it can optionally register the Actions package in a specific Business solution. For the traditional scenario, the command would be:

```
cmf new data --version 1.0.0 --businessPackage .\Cmf.Custom.Business\
```

UI

The UI and Help packages are also scaffolded differently from a `solgen` project. To fully scaffold these solutions, the corresponding Deployment Framework package needs to be specified. These can be found in the MES ISO/disk. Make sure you use the correct version: a mismatch may cause all kinds of problems when running.

The corresponding commands are:

HTML

```
cmf new html --version 1.0.0 --htmlPackage H:\packages\Cmf.Presentation.HTML.8.2.1.zip
```

If you require NPM registry authentication, the current procedure is to include the auth information in the apps\customization.web.npmrc file as is standard.

HELP

```
cmf new help --version 1.0.0 --documentationPackage H:\packages\Cmf.Documentation.8.2.1.zip
```

IoT

The IoT package contains both IoTData and IoTPackages as sub-packages. They are always created together.

```
cmf new iot --version 1.0.0
```

Database

The database package contains both Pre, Post and Reporting sub-packages.

```
cmf new database --version 1.0.0
```

Tests

The tests package is generated and built like any other layer package, but it is not installable. It is also not usually delivered to customers, unless requested.

```
cmf new test --version 1.0.0
```

5.7.3 Demo

This demo show the usual initial setup for a new project. For the first sprints, which focus heavily on modelling, the Data package is of the most importance. Obviously a Tests package is also needed. As an extra, the Business package is also initialized. This allows the Process Rules in the Data package to have a .NET solution where they can be compiled for checking.

Note that the GIF is quite large and can take a bit to load.

- 48/48 -