

# **@criticalmanufacturing/ cli Documentation**

---

# Table of contents

---

1. About cmf-cli	3
1.1 Use cmf-cli to . . .	3
1.2 Getting started	3
2. Installation	4
2.1 Checking your version of cmf-cli and Node.js	4
3. Plugins	5
4. Telemetry	6
4.1 Telemetry implementation	6
5. Reference	7
5.1 Commands	7
5.2 API	29
6. Scaffolding	34
6.1 Pre-requisites	34
6.2 Scaffolding a repository	34
6.3 Pipelines	35

# 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.

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

- [Portal SDK](#) - command line tools to interact with the Critical Manufacturing Customer Portal

## 4. Telemetry

---

### 4.1 Telemetry implementation

---

Basic telemetry currently only tracks the CLI startup and logs:

- CLI name and version
- latest version available in NPM
- if the CLI version is stable or testing
- if the CLI is outdated

#### **no identifiable information is collected in basic telemetry**

However, any user can optionally enable extended telemetry, which might help with troubleshooting. **Extended telemetry includes identifiable information** and as such should be used with care. This includes the basic telemetry, plus:

- for the version (startup) log, it also includes
- current working directory
- hostname
- IP
- username

It also tracks and logs the package tree if, for any command, the tree must be computed. This includes all of the above information plus the package name, for each package in the tree.

Enabling telemetry can be done via environment variables:

- `cmf_cli_enable_telemetry` - enable basic telemetry. If this is off (the default), no telemetry will be collected, even if extended telemetry is on. To enable, set to `true` or `1`, do not set or set to `false` or `0` to disable.
- `cmf_cli_enable_extended_telemetry` - enable extended telemetry. Note the above warnings regarding the impact of keeping this on. To enable, set to `true` or `1`, do not set or set to `false` or `0` to disable.
- `cmf_cli_telemetry_enable_console_exporter` - also print the telemetry information to the console. This is for auditing or troubleshooting as it makes the console output extremely verbose
- `cmf_cli_telemetry_host` - specify an alternate telemetry endpoint (if you're hosting your own)

## 5. Reference

---

### 5.1 Commands

---

#### 5.1.1 assemble

##### DESCRIPTION

cmf assemble is a command that will read all the dependencies of a given package of type Root, and assemble it together.

If a `repositories.json` file is available in the working directory, the repos will be also read from that file.

The command will copy all defined dependencies from the repos, and paste it on the defined outputDir. If the package is not found, and error is thrown.

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

##### 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

## 5.1.2 build

---

### 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	



## 5.1.3 build help

---

### 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	

## 5.1.4 build help generateBasedOnTemplates

---

### Usage

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

### OPTIONS

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

## 5.1.5 build help generateMenuItems

---

### Usage

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

### OPTIONS

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

## 5.1.6 bump

---

### 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	

## 5.1.7 bump iot

---

### 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: .]

## 5.1.8 bump iot configuration

### 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

## 5.1.9 bump iot customization

### 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

## 5.1.10 init

### 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> (REQUIRED)	Configuration file exported from Setup [default: ]
--deploymentDir <deploymentDir> (REQUIRED)	Deployments directory
--MESVersion <MESVersion> (REQUIRED)	Target MES version
--DevTasksVersion <DevTasksVersion> (REQUIRED)	Critical Manufacturing dev-tasks version
--HTMLStarterVersion <HTMLStarterVersion> (REQUIRED)	HTML Starter version
--yoGeneratorVersion <yoGeneratorVersion> (REQUIRED)	@criticalmanufacturing/html Yeoman generator version
--nugetVersion <nugetVersion> (REQUIRED)	NuGet versions to target. This is usually the MES version
--testScenariosNugetVersion <testScenariosNugetVersion> (REQUIRED)	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
--ISOLocation <ISOLocation>	MES ISO file [default: ]
--nugetRegistryUsername <nugetRegistryUsername>	NuGet registry username
--nugetRegistryPassword <nugetRegistryPassword>	NuGet registry password
-, -h, --help	Show help and usage information



## 5.1.11 ls

---

### 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

## 5.1.12 new

### 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: ]
securityPortal <workingDir>	[default: ]
test	

### 5.1.13 new business

---

**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

## 5.1.14 new data

---

### 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

## 5.1.15 new database

---

### 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

## 5.1.16 new feature

---

### 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

## 5.1.17 new help

---

### 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

## 5.1.18 new html

---

### 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



## 5.1.19 new iot

---

### 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

### COMMANDS

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

## 5.1.20 new test

---

### Usage

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

### OPTIONS

Name	Description
<code>--version &lt;version&gt;</code>	Package Version [default: 1.0.0]
<code>-, -h, --help</code>	Show help and usage information

## 5.1.21 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.

### Usage

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

### ARGUMENTS

Name	Description
<code>&lt;workingDir&gt;</code>	Working Directory [default: .]

### OPTIONS

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

## 5.1.22 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 registered 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.

### Usage

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

### ARGUMENTS

Name	Description
<packagePath>	Package path

### OPTIONS

Name	Description
<code>-r, --repo, --repos &lt;repos&gt;</code>	Repositories where dependencies are located (folder)
<code>-?, -h, --help</code>	Show help and usage information

## 5.2 API

---

### 5.2.1 cmfpackage.json

This document is all you need to know about what's required in your **cmfpackage.json** file.

#### PACKAGEID

The most important things in your cmfpackage.json are the `packageId` and `version` fields as they will be required. The `packageId` and `version` together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version.

The `packageId` is automatically generated when the package is created via the `cmf new` and by default contains the `packageType` (eg: `Cmf.Feature.Business`).

#### VERSION

The most important things in your cmfpackage.json are the `packageId` and `version` fields as they will be required. The `packageId` and `version` together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version.

Version must be parseable by [node-semver](#).

#### DESCRIPTION

Put a description in it. It's a string.

#### PACKAGETYPE

The `packageType` is defined via an Enum, check all the valid values [here](#).

#### ISINSTALLABLE

Boolean value (default true). A value of true indicates that the package is prepared to be installed. This value is usually only false for packages of type Tests.

#### ISUNIQUEINSTALL

Boolean value (default false). A value of true indicates that the package is only installed once per environment, a second run of this package doesn't re-install it. This value is usually only true for packages of type Data, IoTData and Database.

#### ISTOSETDEFAULTSTEPS

Boolean value (default true). A value of true indicates that a set of predefined steps (each `packageType` has a set of steps) will be used. This value is usually only false when some debug is needed or for some reason the default steps are not working as expected.

#### STEPS

The `steps` field is an array of steps that will be executed during the package deployment. This is by default empty, because each package type has a set of predefined steps. This field is very useful when you want to add steps to be executed after the predefined or when combined with the property `isToSetDefaultSteps` as false you can override all the predefined steps.

Example:

```
{
  "packageId": "Cmf.Custom.Data",
  "version": "1.1.0",
  "description": "Cmf Custom Data Package",
  "packageType": "Data",
  "isInstallable": true,
  "isUniqueInstall": true,
  "steps": [
    {
      "type": "Generic",
```

```

    "onExecute": "scriptToRun.ps1"
  }
}

```

## CONTENTTOPACK

The `contentToPack` field is an array of file patterns (source and target, check below) that describes the files and folders to be included when your package is packed. File patterns follow a similar syntax to `.gitignore`, but reversed: including a file, directory, or glob pattern (\*, \*\*/\*, and such) will make it so that file is included in the zip when it's packed.

Some special files and directories are also included or excluded regardless of whether they exist in the files array (see [here](#)).

You can also provide a `.cmfpackageignore` file, which will keep files from being included.

Properties:

### source

File pattern where the files/directories to pack are located (relative to the `cmfpackage.json`).

### target

File pattern where the files/directories should be placed in the zip.

### contentType

Usually used in packages of Type Data. Defined via an Enum, check all the valid values [here](#).

### ignoreFiles

File pattern that should point to `.cmfpackageignore` files.

### action

Action that will occur during the packing. Defined via an Enum, check all the valid values [here](#).

The properties **source** and **target** have support for token replacement of any property of the `cmfpackage`. Example:

```

{
  "packageId": "Cmf.Custom.Database.Post",
  "version": "1.1.0",
  "description": "Cmf Custom Database Post Scripts Package",
  "packageType": "Database",
  "isInstallable": true,
  "isUniqueInstall": false,
  "contentToPack": [
    {
      "source": "Online/${version}/**",
      "target": "Online/${version}"
    }
  ]
}

```

This means that the `contentToPack` will look in to the folder `/Online/1.1.0`.

## DEPENDENCIES

Dependencies are specified in an array of a simple object that maps a `packageId` to a version. It can point to a local dependency (in the same repo) or to a remote dependency (in a remote repo).

Remote dependencies depend on remote repos (currently we only support folders), these repos are defined in the file [repositories.json](#)

Example:

```

{
  "packageId": "Cmf.Custom.Data",
  "version": "1.1.0",
  "description": "Cmf Custom Data Package",
  "packageType": "Data",
  "isInstallable": true,
  "isUniqueInstall": true,
  "dependencies": [
    {

```

```

    "id": "Cmf.Custom.Business",
    "version": "1.0.0"
  }
}

```

#### RELATEDPACKAGES

In some cases, you want to guarantee that a set packages are built or packed together. To do this you just need to add this property, point to the `relativePath` of the package and define when it should be build or packed. Example:

```

{
  "packageId": "Cmf.Custom.Data",
  "version": "1.1.0",
  "description": "Cmf Custom Data Package",
  "packageType": "Data",
  "isInstallable": true,
  "isUniqueInstall": true,
  "relatedPackages": [
    {
      "path": "../Cmf.Custom.Business",
      "preBuild": true,
      "postBuild": false,
      "prePack": false,
      "postPack": false
    }
  ]
}

```

#### TESTPACKAGES

Just like the `dependencies` property, the `testPackages` are specified in an array of a simple object that maps a `packageId` to a version. This is useful to link packages of type Tests, to any other packages. This allow the `cmf assemble` command to assemble it together with the relative package.

### Generic Type Packages

This type doesn't have any predefined BuildStep, Step or ContentToPack, so it will completely rely on what is defined to know how it should be built, packed and deployed. Check the above properties that are only available for this PackageType.

#### BUILDSTEPS

Array of terminal commands (similar to [package.json scripts](#)) that will be used to build the package during the `cmf build` command execution.

Example:

```

"buildSteps": [
  {
    "args": ["build -c Release"],
    "command": "dotnet",
    "workingDirectory": "."
  }
]

```

#### DFPACKAGE TYPE

The `dfPackageType` is defined via an Enum, check all the valid values [here](#).

#### TARGETLAYER

String value that should match a container layer from CM Framework. Valid values should be checked in the official documentation [here](#).

#### EXAMPLE

```

{
  "packageId": "Cmf.Custom.Generic.Package",
  "version": "1.0.0",
  "description": "Generic Package",
  "packageType": "Generic",
  "dfPackageType": "Business",
  "targetLayer": "host",
  "isInstallable": true,
  "isUniqueInstall": false,
  "buildSteps": [
    {
      "args": [

```

```
    "build -c Release"
  ],
  "command": "dotnet",
  "workingDirectory": ".",
},
"contentToPack": [
  {
    "source": "Release/netcoreapp3.1/*.*",
    "target": ""
  }
],
"steps": [
  {
    "type": "DeployFiles",
    "contentPath": "**/*"
  }
]
}
```



## 5.2.2 repositories.json

---

This document is all you need to know about what's required in your **repositories.json** file.

### CIREPOSITORY

Path that points to a folder that contain packages that are treat as Continuous Integration packages.

### REPOSITORIES

Array of paths that point to folders that contain package that are treat as official (i.e. upstream dependencies or already releases packages).

Example:

```
{
  "CIRepository": "\\fileshare\my-continuous-integration\packages\repository",
  "Repositories": [
    "\\fileshare\my-official\packages\repository",
    "\\fileshare\my-released\packages\repository"
  ]
}
```

## 6. Scaffolding

---

### 6.1 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 v10**, the recommended versions are:

- latest node 18 (Hydrogen)
- latest npm 9 (should come with node)

For MES v8 and v9, 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
```

For **MES v10**, you will also need [angular cli](#)

```
npm install -g @angular/cli
```

#### 6.1.1 NuGet and NPM repositories

---

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](#)

#### 6.1.2 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...
```

## 6.2 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](#).

## 6.3 Pipelines

---

By default, our scaffolding doesn't provide any built-in CI/CD pipelines, giving you the flexibility to choose any tool/platform that suits your needs.

However, we can share as a reference our internal process:

### 6.3.1 Pull Requests (PRs)

---

For each changed package, we run the command `cmf build --test`, which compiles the package and runs unit tests if available, comparing with the target branch.

We consider a package as "changed" when any file is modified inside a folder with a `cmfpackage.json` file.

An alternative is to run `cmf build --test` for all packages.

### 6.3.2 Continuous Integration (CI)

---

After merging code into the main branch, we perform the following steps:

1. Run `cmf build --test` to ensure successful building of all packages and passing of unit tests.
2. Run `cmf pack` to generate a package that can be installed via DevOps Center or Critical Manufacturing Setup.

### 6.3.3 Continuous Deployment (CD)

---

#### Traditional (Windows VMs)

1. Follow the instructions in the [documentation](#)
2. In the Package Sources step, add the path where your packages are located.

#### Containers

1. Follow the instructions in the [documentation](#)
2. Copy the generated packages to the folder defined in volume **Boot Packages**.
3. In the Configuration > General Data step, set the Package to Install as `RootPackageId@PackageVersion`