DevOps

Week 13

Murtaza Munawar Fazal

Dependencies in software

- Modern software development involves complex projects and solutions.
- Projects have dependencies on other projects, and solutions aren't single pieces of software.
- The solutions and software built consist of multiple parts and components and are often reused.
- As codebases are expanding and evolving, it needs to be componentized to be maintainable.
- A team that is writing software won't write every piece of code by itself but use existing code written by other teams or companies and open-source code that is readily available.
- A software engineer will need to identify the components that make up parts of the solution and decide whether to write the implementation or include an existing component.
- The latter approach introduces a dependency on other components.

What is dependency management?

- Software dependencies that are introduced in a project and solution must be appropriately declared and resolved.
- You need to manage the overall composition of the project code and the included dependencies.
- Without proper dependency management, it will be hard to keep the components in the solution controlled.
- Management of dependencies allows a software engineer and team to be more efficient working with dependencies.
- With all dependencies being managed, it's also possible to control the consumed dependencies, enabling governance and security scanning to use known vulnerabilities or exploits packages.

Elements of a dependency management strategy

- There are many aspects of a dependency management strategy.
  - **Standardization** Managing dependencies benefit from a standardized way of declaring and resolving them in your codebase. Standardization allows a repeatable, predictable process and usage that can be automated as well.
  - **Package formats and sources** The distribution of dependencies can be performed by a packaging method suited for your solution's dependency type. Each dependency is packaged using its usable format and stored in a centralized source. Your dependency management strategy should include the selection of package formats and corresponding sources where to store and retrieve packages.
  - **Versioning** Just like your own code and components, the dependencies in your solution usually evolve. It requires a versioning mechanism for the dependencies to be selective of the version of a dependency you want to use.

Identify dependencies

- It starts with identifying the dependencies in your codebase and deciding which dependencies will be formalized.
- It's common to use libraries and frameworks that are not written by yourself.
- Additionally, your existing codebase might have internal dependencies that aren't treated as such.
- For example,
  - Take a piece of code that implements a particular business domain model. It might be included as source code in your project and consumed in other projects and teams.
  - It would help if you investigated your codebase to identify pieces of code that can be considered dependencies and treat them as such.
  - It requires changes to how you organize your code and build the solution. It will bring your components.

Source and Package componentization

- There are two ways of componentization commonly used.
  1. **Source componentization** The first way of componentization is focused on source code. It refers to splitting the source code in the codebase into separate parts and organizing it around the identified components.

     It works if the source code isn't shared outside of the project. Once the components need to be shared, it requires distributing the source code or the produced binary artifacts created from it

  2. **Package componentization** The second way uses packages. Distributing software components is performed utilizing packages as a formal way of wrapping and handling the components.

     A shift to packages adds characteristics needed for proper dependency management, like tracking and versioning packages in your solution.

Scan your codebase for dependencies

- **Duplicate code** When certain pieces of code appear in several places, it's a good indication that this code can be reused. However, if the code can be made available properly, it does have benefits over copying code and must manage that. The first step to isolate these pieces of duplicate code is to centralize them in the codebase and componentize them in the appropriate way for the type of code.
- **High cohesion and low coupling** A second approach is to find code that might define components in your solution. You'll look for code elements that have high cohesion and low coupling with other parts of code.

Scan your codebase for dependencies

- **Individual lifecycle** Related to high cohesion, you can look for parts of the code that have a similar lifecycle and can be deployed and released individually. If such code can be maintained by a team separate from the codebase that it's currently in, it's an indication that it could be a component outside of the solution.
- **Stable parts** Some parts of your codebase might have a slow rate of change. That code is stable and isn't altered often.
- **Independent code and components** Whenever code and components are independent and unrelated to other parts of the system, they can be isolated to a separate component and dependency.

Package

- A package is a formalized way of creating a distributable unit of software artifacts that can be consumed from another software solution.
- The package describes the content it contains and usually provides extra metadata, and the information uniquely identifies the individual packages and is self-descriptive.
- It helps to better store packages in centralized locations and consume the contents of the package predictably.

Types of Packages

- The type of components you want to use in your codebase differ for the different parts and layers of the solution you're creating.
- Over the past years, the packaging formats have changed and evolved. Now there are a couple of de facto standard formats for packages.
  - Nuget
  - NPM
  - Maven
  - PyPi
  - Docker

Nuget

- NuGet packages (pronounced "new get") are a standard used for .NET code artifacts. It includes .NET assemblies and related files, tooling, and sometimes only metadata.
- NuGet defines the way packages are created, stored, and consumed. A NuGet package is essentially a compressed folder structure with files in ZIP format and has the .nupkg extension.

NPM

- NPM An NPM package is used for JavaScript development. It originates from node.js development, where it's the default packaging format. An NPM package is a file or folder containing JavaScript files and a package.json file describing the package's metadata.

Maven

- Maven is used for Java-based projects. Each package has a Project Object Model file describing the project's metadata and is the basic unit for defining a package and working with it.

PyPi

- PyPi The Python Package Index, abbreviated as PyPI and known as the Cheese Shop, is the official third-party software repository for Python.

Docker

- Docker packages are called images and contain complete and self-contained deployments of components. A Docker image commonly represents a software component that can be hosted and executed by itself without any dependencies on other images. Docker images are layered and might be dependent on other images as their basis. Such images are referred to as base images.

Understand package feeds

- Packages should be stored in a centralized place for distribution and consumption to take dependencies on the components it contains.
- The centralized storage for packages is commonly called a package feed. There are other names in use, such as repository or registry.
- One feed typically contains one type of packages. There are NuGet feeds, NPM feeds, Maven repositories, PyPi feed, and Docker registries.
- Package feeds offer versioned storage of packages. A particular package can exist in multiple versions in the feed, catering for consumption of a specific version.

Private and Public feeds

- Depending on the package, purpose, and origin, it might be generally available or to a select audience.
- The feeds can be exposed in public or private to distinguish in visibility.
- Typically, open-source projects for applications, libraries, and frameworks are shared with everyone and publicly available.
- Anyone can consume public feeds whereas Private feeds can only be consumed by those who are allowed access.

Private and Public feeds

- There might be reasons why you don't want your packages to be available publicly.
  - It could be because it contains intellectual property or doesn't make sense to share with other software developers.
  - Components developed for internal use might be available only to the project, team, or company that developed it.
- In such cases, you can still use packages for dependency management and choose to store the package in a private package feed.

Consume Package

The developer flow will follow this general pattern:

1. Identify a required dependency in your codebase.
2. Find a component that satisfies the requirements for the project.
3. Search the package sources for a package offering a correct component version.
4. Install the package into the codebase and development machine.
5. Create the software implementation that uses the new components from the package.

Upstream sources

- Part of package management involves keeping track of the various sources.
- It's possible to refer to multiple sources from a single software solution. However, when combining private and public sources, the order of resolution of the sources becomes essential.
- One way to specify multiple package sources is by choosing a primary source and an upstream source.
- The package manager will evaluate the primary source first and switch to the upstream source when the package isn't found there.
- The upstream source might be one of the official public or private sources.
- A typical scenario is to use a private package source referring to a public upstream source for one of the official feeds. It effectively enhances the packages in the upstream source with packages from the private feed, avoiding publishing private packages in a public feed.

Upstream sources

- A source with an upstream source defined may download and cache the requested packages if the source doesn't contain those packages themselves.
- The source will include these downloaded packages and starts to act as a cache for the upstream source. It also offers the ability to keep track of any packages from the external upstream source.
- An upstream source can be a way to avoid direct access of developers and build machines to external sources.
- The private feed uses the upstream source as a proxy for the external source. It will be your feed manager and private source that have the communication to the outside. Only privileged roles can add upstream sources to a private feed.

Azure Artifacts

- Microsoft Azure DevOps provides various features for application lifecycle management, including:
  - Work item tracking.
  - Source code repositories.
  - Build and release pipelines.
  - Artifact management.
- The artifact management is called Azure Artifacts and was previously known as Package management. It offers public and private feeds for software packages of various types.

Types of packages supported

- Azure Artifacts currently supports feeds that can store five different package types:
  - NuGet packages
  - NPM packages
  - Maven
  - Universal packages
  - Python
- Universal packages are an Azure Artifacts-specific package type. In essence, it's a versioned package containing multiple files and folders.
- A single Azure Artifacts feed can contain any combination of such packages.