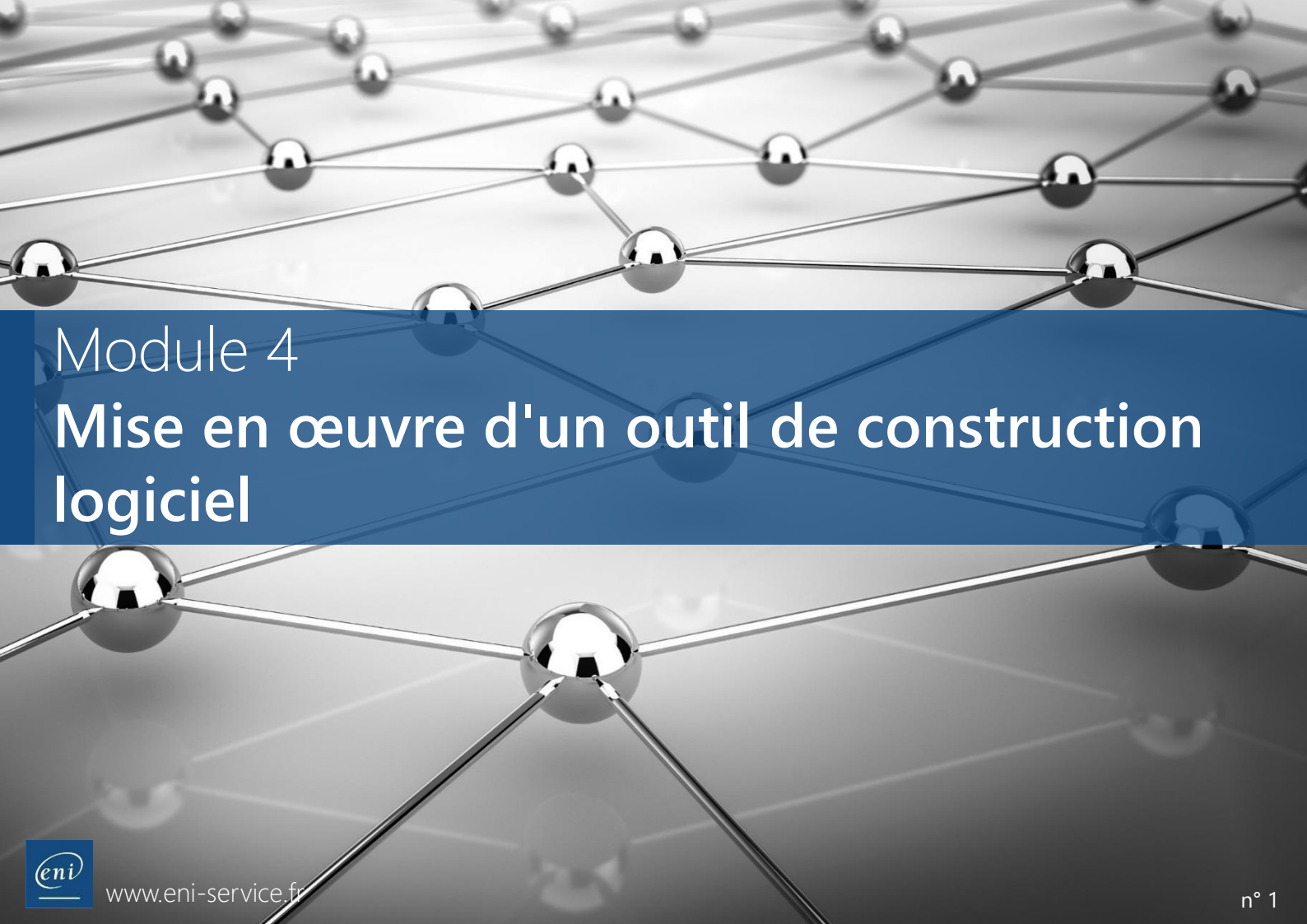


Module 4

Mise en œuvre d'un outil de construction logiciel

Les outils de construction logiciel

- Un outil de construction logiciel permet de faciliter et d'automatiser certaines tâches de la gestion d'un projet
  - On parle d'outils de « *build* »
- Il permet notamment :
  - d'automatiser certaines tâches :
    - Compilation
    - Exécution des tests
    - Packaging des applications
  - de gérer les dépendances (bibliothèques nécessaires au projet)
- Ils peuvent être utilisés :
  - de manière autonome par un développeur
    - En ligne de commande
    - Via l'intégration à un IDE
  - par une plateforme d'intégration continue

Perspectives pour le développeur

- Pour le développeur, le principal intérêt se situe dans le fait de n'avoir à lancer qu'une seule commande pour toutes ses tâches
  - Compilation
  - Exécution des tests
  - Packaging des livrables
- D'un point de vue des tests par exemple, cela assure que tous les tests sont exécutés
  - Même s'ils peuvent être omis…
- Ils gèrent les dépendances de projet
  - Cela permet d'inclure facilement les bibliothèques et frameworks nécessaires
- Autre avantage : l'uniformité des constructions
  - Chaque construction se fait avec les mêmes étapes à chaque fois
- L'intégration aux outils de développement (IDE) facilite l'usage de l'outil de construction

Perspectives pour l'intégration continue

- Globalement des avantages similaires à ceux apportés aux développeurs
  - Automatisation des tâches
  - Gestion de dépendances
  - Exécution des tâches
- Cela facilite également la mise en place des projets de construction dans l'ordonnanceur
  - Potentiellement une seule commande à exécuter dans le job !

Panorama des outils

- Un bon nombre de langages et technologies de développement permettent l'usage d'un outil de construction
  - Certains outils ont quelques fonctionnalités indépendantes des langages
- Quelques références :
  - MSBuild, NAnt, dotnet CLI
    - C#
  - Make
    - C++
  - Maven, Ant
    - Java
  - zc.buildout, PyBuilder
    - Python

Principes de base

Cycle de vie

- Le cycle de vie, d'un point de vue d'un outil de construction logicielle, représente la liste des tâches qu'il doit effectuer ainsi que leur ordonnancement logique
- Cela peut par exemple inclure :
  - La vérification d'une arborescence de projet
  - La compilation du code
  - L'exécution des tests
  - La création d'un livrable
  - …
- Ce cycle de vie peut être contrôlé le plus souvent par une configuration afin d'affiner les tâches à réaliser à chaque étape

La gestion des dépendances

- Le principe de gestion de dépendances consiste à déléguer à un outil la responsabilité de trouver toutes les bibliothèques et exécutables nécessaire au fonctionnement d'une application
  - L'objectif étant de décharger le programmeur d'avoir à localiser, télécharger et installer ces éléments qui peuvent eux même avoir des dépendances !
- Pour aller plus loin, l'outil peut télécharger et mettre automatiquement ces bibliothèques à disposition de l'application
  - En utilisant un dépôt en ligne référençant les bibliothèques
    - Ex : Maven Central en Java, NuGet en C#, PyPi en Python, …

En Java : Apache Maven

Objectifs de Maven

- Apache Maven est un outil logiciel libre pour la gestion et l'automatisation de production des projets logiciels Java en général et Java EE en particulier.
  - A l'instar de l'outil Make sous Unix
- Maven est géré par l'organisation Apache Software Foundation.
- Maven utilise le principe de « Project Object Model » (POM) afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production.
- Il est livré avec un grand nombre de tâches prédéfinies
  - compilation de code Java, lancement de test, génération de rapports…
- Maven est capable de fonctionner en réseau.
  - Une des motivations historiques de cet outil est de fournir un moyen de synchroniser des projets indépendants
  - Maven peut automatiquement télécharger des librairies sur des dépôts logiciels connus, distants (Internet) ou locaux (LAN).
- http://maven.apache.org

Principes de base

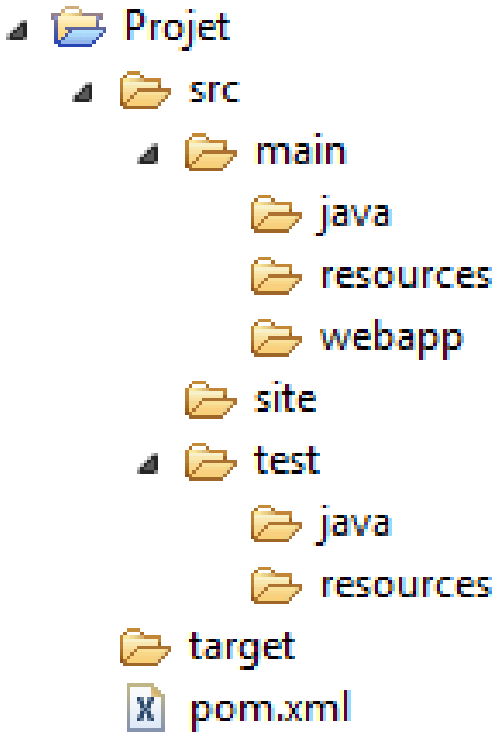
- Maven utilise le principe « Convention Over Configuration »
  - Moins de configuration Maven impose une arborescence et un nommage strict des fichiers d'un projet.
  - Ces conventions permettent de réduire la configuration
  - Si l'on s'écarte de la convention, on doit le préciser dans la configuration du projet.
- Chaque projet (et sous-projet) est configuré par un POM qui contient les informations de base pour permettre à Maven de travailler :
  - Nom du projet et numéro de version,
  - Dépendances éventuelles vers d'autres projets,
  - Bibliothèques nécessaires à la compilation,
  - Noms des développeurs,
  - …
- Un fichier `pom.xml` à la racine du projet (POM Parent)
- Approche permettant l'héritage des propriétés du projet parent.
  - Si une propriété est redéfinie dans le POM du projet, elle surcharge celle qui est définie dans le projet parent. (Réutilisation et allègement de configuration).

Installation et configuration de Maven

- Maven est logiciel écrit en Java pour gérer des projets Java
  - Une plateforme Java (JDK pour pouvoir compiler) est évidemment nécessaire…
  - De plus la variable d'environnement JAVA_HOME doit pointer vers le répertoire d'installation de la plateforme Java
- Télécharger une distribution binaire de Maven et la décompresser dans le répertoire souhaité pour l'installation
- Pour un usage en ligne de commande il faut référencer le sous répertoire `bin/` de l'installation dans le PATH du système

Arborescence

- Arborescence standard d'un projet Maven :
  - Deux répertoires de base, `src/` et `target/`



- `target/` est le répertoire de destination de tous les traitements Maven
- `src/` contient les sources de l'application
  - `site/` : les fichiers pour le site
  - `main/` : le logiciel…
  - `test/` : Code des tests unitaires
    - `resources/` : les ressources de l'application
    - `webapp/` : les fichiers de l'application Web
    - `java/` : les sources Java

Le fichier pom.xml

- Structure d'un fichier POM minimal :

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

        <modelVersion>4.0.0</modelVersion>
        <groupId>fr.eni</groupId>
        <artifactId>exemple-formation-maven</artifactId>
        <packaging>jar</packaging>
        <name>Projet d'exemple Maven</name>
        <version>1.0</version>

</project>
```

La commande mvn et le cycle de vie

- Maven s'utilise dans un interpréteur de commande : la commande `mvn`
  - Il peut également être intégré à un IDE…
- Maven gère le cycle de vie d'un projet en passant par différentes étapes :
  - Cycle de vie par défaut :
    - `validate` : Valide la structure du projet
    - `compile` : Compile les sources
    - `test-compile` : Compile le code source des tests
    - `test` : Lance les tests unitaires
    - `package` : Création des archives Jar, War, Ear...
    - `verify` : Test l'archive créée.
    - `install` : Installe l'archive localement, elle peut être utilisée comme dépendance.
    - `deploy` : Installe l'archive sur un serveur pour qu'elle puisse être réutilisé par tout le monde.
  - Cycle de vie « clean » (`mvn clean`)
    - `clean` : Nettoyage du répertoire `target/`
- Exemple : `mvn compile`
  - Compile le code mais fait également la validation de la structure du projet

Configuration des dépendances

- Exemple de configuration à ajouter : (Pour Struts 1.2.x)

```
<dependencies>
  ...
  <dependency>
    <groupId>struts</groupId>
    <artifactId>struts</artifactId>
    <version>1.2.9</version>
    <scope>compile</scope>
  </dependency>
  ...
</dependencies>
```

Syntaxe de la configuration

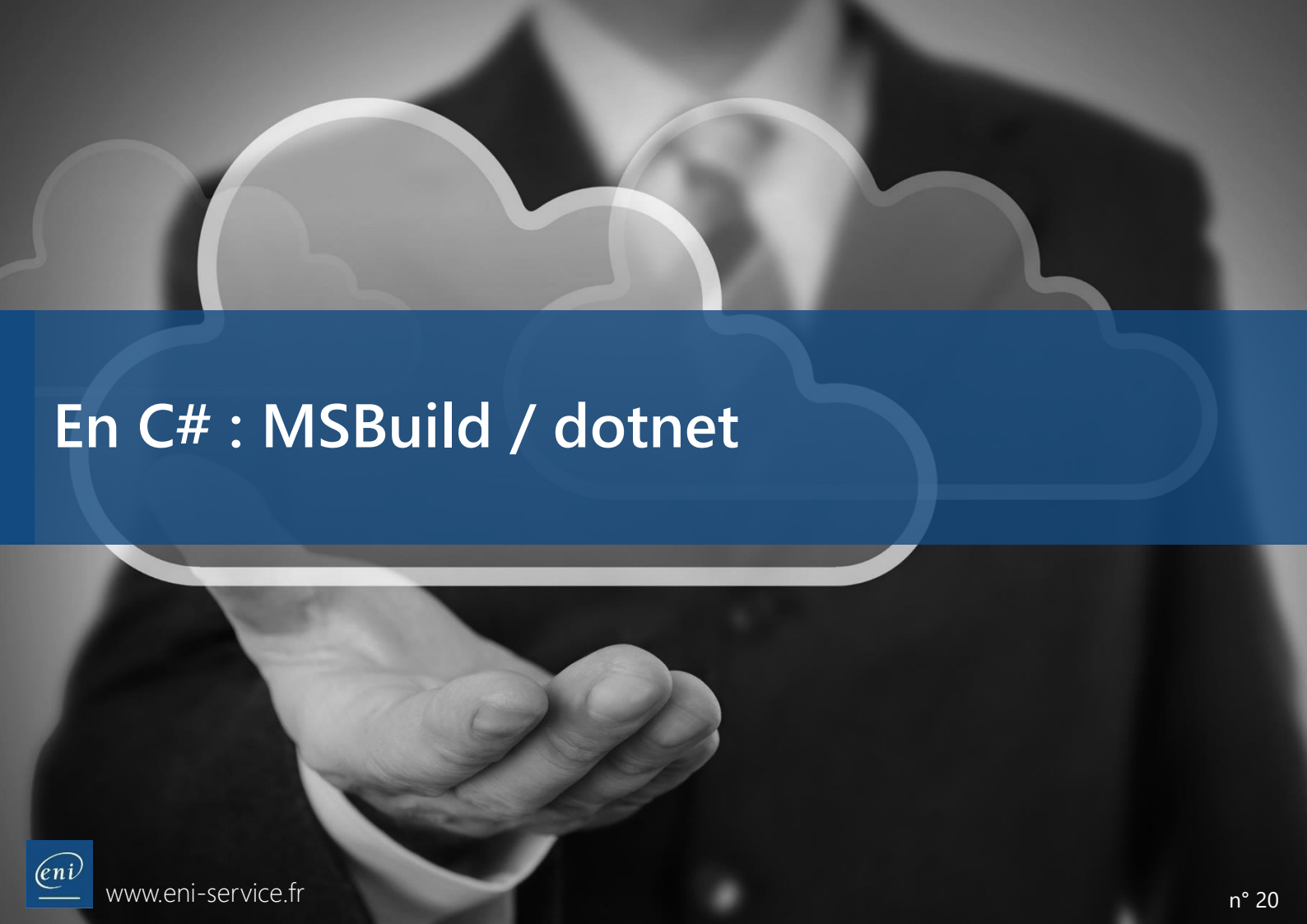
- Moteur de recherche de librairies pour Maven : http://search.maven.org
- Paramètres obligatoires sont le `groupId` et l'`artifactId` et la `version`.
  - Il est important de spécifier la version, sinon Maven utilise toujours la dernière version en date (alpha, beta...).
- Le paramètre `scope` permet de spécifier l'utilité de la librairie :
  - **compile** : Valeur par défaut, la dépendance sera toujours disponible dans le CLASSPATH.
  - **provided** : Indique que la dépendance est nécessaire pour la compilation mais sera fournie par le container ou le JDK et donc ne sera pas fournie dans le package.
  - **runtime** : Indique que la dépendance est nécessaire pour l'exécution mais pas pour la compilation.
  - **test** : Indique que la dépendance est nécessaire pour la compilation et l'exécution des tests unitaires.
- Le scope **provided** est très intéressant pour les servlet par exemple... (Nécessaire a la compilation mais pas a l'exécution)

Les tests unitaires

- JUnit est naturellement intégré à Maven grâce au plugin **Surefire**
- Le lancement des tests se fait simplement en lançant le goal « test » de Surefire
  - `mvn surefire:test`
- Ou bien encore plus simplement en invoquant le cycle « test »
  - `mvn test`
- Surefire inclut toutes les classes dont le nom commence par **Test** ou se termine par **Test**, **Tests** ou **TestCase**.

La configuration de test

- Dans le cadre d'un projet Maven, la configuration de test peut (doit ?) être spécifique
- Pour le code, **src/main/resources** contient habituellement la configuration
- Pour les tests, **src/test/resources** peut fournir une configuration alternative pour l'exécution des tests
  - La configuration de test pourra faire référence à une base de données mémoire par exemple
  - Cette configuration pourra prendre en charge l'initialisation et la réinitialisation d'un jeu de données.



En C# : MSBuild / dotnet

MSBuild

- MSBuild est un moteur de compilation utilisé par Visual Studio pour effectuer des opérations de compilation ou de nettoyage
  - Lorsque ces actions sont lancées via l'interface de Visual Studio, c'est l'exécutable `msbuild.exe` qui est utilisé
- A partir d'un fichier de solution ou de projets, MSBuild va ordonner et lancer la compilation en utilisant le compilateur `csc.exe` de façon à générer une application exécutable ou une assembly
  - MSBuild utilise les informations contenues dans le fichier solution ou dans les fichiers de projet savoir comment ordonner la compilation avec `csc.exe`
  - Ces fichiers constituent des espèces de scripts lisibles par MSBuild
- MSBuild est découplé de Visual Studio ce qui permet de :
  - L'utiliser à partir de la ligne de commandes sur des serveurs sans lancer Visual Studio,
  - Personnaliser le mécanisme de compilation pour effectuer des opérations particulières, par exemple, avant la compilation ou pour copier le résultat de compilation dans un répertoire particulier etc...
- Le point d'entrée de `msbuild.exe` est :
  - un fichier solution (`.sln`)
  - ou un fichier projet (`.csproj`, `.vbproj`, etc...)

Disponibilité de MSBuild

- Avant Visual Studio 2013, MsBuild était livré avec le Framework .NET et sa version était couplée avec celle du Framework
- Depuis Visual Studio 2013 et le Framework .NET 4.5.1, MSBuild est livré avec Visual Studio et sa version est désormais couplée avec celle de Visual Studio
  - L'exécutable msbuild.exe reste cependant découplé de Visual Studio et il est possible de l'exécuter sans que Visual Studio ne soit lancé
- Pour localiser les installations de MSBuild sur une machine, il faut aller dans la base de registres et consulter la valeur `MSBuildToolsPath` des clés se trouvant dans `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\ToolsVersions\`
- Pour installer MSBuild, il faut installer le Framework .NET

La commande msbuild.exe

- Pour lancer une construction, on précisera en paramètre le nom du fichier de solution ou de projet
  - Exemple, pour compiler une solution à partir du fichier `.sln` et en précisant la configuration de compilation et la plateforme cible :
    - `msbuild.exe "solutionName.sln" /p:configuration=Debug /p:platform="Any CPU"`
- Les actions de génération (*build*), de nettoyage (*clean*) ou de regénération (*rebuild*) correspondent à des cibles (“targets”) prédéfinies. Il est possible d’exécuter une cible particulière
  - `msbuild.exe "[fichier projet]" /t:[cibles séparées par ";"]`
  - Exemple :
    - `msbuild.exe "projectName.csproj" /t:build`

Le fichier de « build »

- Il définit les éléments qui seront lus par MSBuild pour effectuer les différentes actions.
  - En plus des actions de génération ou nettoyage, il est possible de définir d'autres actions : faire appel à un script PowerShell, faire des copies de fichiers, etc…
  - Ce sont des fichiers de solution (`.sln`) ou de projet (`.csproj`, …) au format XML !
- Les éléments principaux d'un fichier de « build » :
  - Target
    - Désigné par un nom qui pourra être appelé en argument de MSBuild pour indiquer l'action qui doit être réalisée.
    - Les « targets » prédéfinies sont *build*, *clean* et *rebuild* mais il est possible d'en définir d'autres
    - Les targets exécutent des « tasks »
  - Task
    - Actions à exécuter, comme par exemple faire une copie de fichiers ou lancer une compilation
  - Item
    - Une liste d'éléments, le plus souvent des fichiers
  - Property
    - Variable dont on définit la valeur et qui est utilisable dans le script

Structure de base du fichier

- Le fichier de build contient un élément racine Project qui contiendra des cibles et des propriétés
- Exemple :

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <OutputDirectory>bin\debug</OutputDirectory>
        <OutputAssembly>$(OutputDirectory)\HelloWorld.exe</OutputAssembly>
        <Optimize>false</Optimize>
    </PropertyGroup>

    <Target Name="Message">
        <Message Text="Hello world" />
    </Target>

</Project>
```

- Pour lancer la construction, il faudrait taper :
  - `msbuild.exe helloWorld.csproj /t:Message`

Les « items »

- Des listes dynamiques d'éléments, le plus souvent des listes de fichiers
  - NOTE : Les valeurs des « items » sont calculées avant exécution des « targets »
- Exemples :

```
<ItemGroup>
    <Compile Include = "file1.cs"/>
    <Compile Include = "file2.cs"/>
</ItemGroup>
```

```
<ItemGroup>
    <MyReleaseFiles Include=".\bin\debug\*.*" Exclude=".\bin\debug\*vshost.exe" />
</ItemGroup>
```

- Les « items » seront ensuite utilisable dans les « targets » via la syntaxe @(nom de la liste)
  - Exemple :

```
<Target Name="Release" >
    <Copy SourceFiles="@(MyReleaseFiles)" DestinationFolder="$(MyReleaseOutput)" />
</Target>
```

Les cibles

- Une cible (« *target* ») correspond à une liste d'actions qu'il est possible d'exécuter en indiquant le nom de la cible lors de l'appel à MSBuild
- Une cible est composée de plusieurs tâches (« *tasks* ») qui correspondent à des actions à exécuter
- Une cible se définit à l'intérieur d'un nœud **Target** et est identifiable en utilisant l'attribut **Name**

```
<Target Name="Release" >
    <Copy SourceFiles="@(MyReleaseFiles)" DestinationFolder="$(MyReleaseOutput)" />
</Target>
```

- Il est possible de définir des dépendances entre les cibles, ainsi qu'une cible par défaut :

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" DefaultTargets="Release">
  <Target Name="Build">
    <!-- ... -->
  </Target>
  <Target Name="Release" DependsOnTargets="Build">
    <!-- ... -->
  </Target>
</Project>
```

Les tâches

- Une tâches (« *task* ») est une action exécutée au sein d'une cible.
  - Une cible peut comporter plusieurs tâches

```
<Target Name="MakeBuildDirectory">
    <MakeDir Directories="$(BuildDir)" />
</Target>
```

- Les tâches les plus courantes :

Message	Affiche un message sur la console.
Copy	Permet de copier un ou plusieurs fichiers d'un répertoire à un autre
Delete	Supprimer un ou plusieurs fichiers
MakeDir	Crée un ou plusieurs répertoires
RemoveDir	Supprime un ou plusieurs répertoires
Exec	Permet d'exécuter un processus externe
MSBuild	Permet d'exécuter un ou plusieurs "targets" dans un fichier MSBuild externe
Csc	Permet d'exécuter le compilateur C# pour produire un exécutable ou une assembly

La gestion des dépendances

- Les dépendances d'une application sont regroupées dans le fichier `.csproj`
- Elle peuvent y être ajoutées :
  - Manuellement en éditant le fichier
  - Par les outils de Visual Studio
  - Par la commande `dotnet`
- Utilisation de la commande `dotnet`
  - La commande `dotnet add package` permet d'ajouter un package à un projet.
  - Exemples :
    - `dotnet add package Microsoft.NET.Test.Sdk`
    - `dotnet add package Nunit3TestAdapter`
    - `dotnet add package NUnit`
  - Le fichier `.csproj` sera mis à jour

Les dépendances dans le fichier

- Après usage de la commande `dotnet add package`, le fichier est mis à jour avec les informations suivantes :

```
<ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.8.3" />
    <PackageReference Include="Nunit" Version="3.12.0" />
    <PackageReference Include="Nunit3TestAdapter" Version="3.17.0" />
</ItemGroup>
```

- A noter que l'option `-v` de la commande `dotnet add package` permet de spécifier la version souhaitée d'une dépendance

La commande `dotnet`

- Cet utilitaire est construit par-dessus MSBuild, il permet donc de réaliser tout ce que MSBuild fait
  - Mais il va plus loin !
- `dotnet` ajoute des cibles par défaut supplémentaires pour :
  - Créer une nouvelle application à partir d'un modèle
  - Télécharger et installer les dépendances déclarées dans le fichier de projet
  - Lancer l'exécution des tests logiciels

Les commandes

- `dotnet new`
  - Cette commande permet de créer la structure d'un projet à partir d'un modèle
  - Elle s'utilise classiquement avec la syntaxe :
    - `dotnet new <type of project template> -o <name of directory>`
  - Par exemple :
    - `dotnet new console -o app`
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-new?tabs=netcore22&?wt.mc_id=personal-blog-chnoring
- `dotnet build`
  - Cette commande génère un projet et toutes ses dépendances.
    - Depuis le répertoire racine avec une solution, alors il construit la solution entière, donc tous les projets de la solution.
    - Depuis un répertoire de projet spécifique, il ne construira que ce répertoire
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-build?wt.mc_id=personal-blog-chnoring

Les commandes (suite…)

- `dotnet run`
  - Cette commande permet d'exécuter le projet, c'est à dire de lancer le point d'entrée
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-run?tabs=netcore21&wt.mc_id=personal-blog-chnoring
- `dotnet clean`
  - Cette commande nettoie un projet en supprimant le contenu des répertoires de génération
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-clean?wt.mc_id=personal-blog-chnoring
- `dotnet add/remove <package/reference>`
  - Permet d'ajouter/supprimer un projet/une librairie à son projet
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-add-reference?wt.mc_id=personal-blog-chnoring
  - https://docs.microsoft.com/dotnet/core/tools/dotnet-add-package?wt.mc_id=personal-blog-chnoring

Usage

- Restauration des dépendances déclarées dans un fichier de projet :
  - `dotnet restore`
- Exécution des tests :
  - `dotnet test`
- Les commandes qui lancent implicitement un « restore » :
  - `dotnet new`
  - `dotnet build`
  - `dotnet run`
  - `dotnet test`

Travaux Pratiques

Travaux Pratique

- En Java :
  - Création d'un projet Maven
  - Ajout des dépendances de tests nécessaires
  - Ajout du code de test précédemment créé dans l'arborescence du projet
  - Exécution des tests avec Maven
- En C#
  - Création d'un projet `dotnet`
  - Ajout des dépendances de tests nécessaires
  - Ajout du code de test précédemment créé dans l'arborescence du projet
  - Exécution des tests avec `dotnet`