02

O processo de build de um software e introdução a ANT

Por que usar uma ferramenta de build?

O desenvolvimento de software passa por diversas fases até chegar ao ambiente de produção. É preciso compilar, testar, empacotar e executar outras tarefas durante o ciclo de vida do software. Muito pior, essas fases se repetem todo dia, ou até mesmo a cada hora.

Se fossemos definir essas fases em um projeto Java, deveríamos começar com a criação dos nossos códigos-fonte em algum diretório específico, normalmente o src/ e após essa etapa compilaríamos nosso código e testes. Com nossos testes validados, prepararíamos para o deploy um JAR (Java Archive) e colocaríamos o mesmo em produção.

Algumas dessas tarefas podem requerer uma sequência grande de passos, que manualmente pode dar muito trabalho, inclusive acarretar erros. O uso da IDE facilita bastante algumas dessas etapas, embora ainda seja necessário seguir algumas sequências manualmente, como é o caso da verificação dos testes e o processo de gerar um JAR e deployar no servidor, mas não é sempre que temos a IDE facilmente disponível e bem configurada.

Como tudo que se repete e é feito manualmente, a presença do erro humano é rotineira. Uma situação em que o desenvolvedor envia para produção um projeto cujo testes não estão validados, pode vir a acontecer e colocar em risco toda a aplicação. Tirar do desenvolvedor a responsabilidade de fazer todo esse trabalho repetitivo e cansativo é uma das tarefas dos builds automatizados.

Existem milhares de ferramentas de build, sendo o make uma das mais famosas, muito usada para construir e executar tarefas de projetos geralmente escritos em C e C++. ANT e MAVEN são ferramentas bastante difundidas no mundo Java, ambas possuem suporte das IDEs, e são facilmente executadas em qualquer sistema operacional. Nesse curso veremos como utilizar o ANT nos nossos projetos para automatizar tarefas recorrentes.

Build automatizados e entrega contínua

O processo de build é o coração das práticas de engenharia ágil: é através dele que automatizamos todos os passos necessários para garantir a qualidade mínima esperada. Esse nível de garantias varia entre cada projeto e todos os passos que uma equipe madura seria capaz de executar com quase perfeição são automatizados para minimizar falhas e atingir tal objetivo.

Diversas práticas são utilizadas para garantir a qualidade necessária que nos permite executar entregas contínuas, e a maior parte delas estão presentes ou são referenciadas dentro do processo automatizado de build. É fundamental para o desenvolvimento ágil automatizar o processo de construção e implantação do projeto através do build automatizado.

O ANT como ferramenta de build

A ideia do Ant (http://ant.apache.org/) é definir uma série de tarefas, que são invocadas através de tags XML, basicamente usamos o XML como uma linguagem de programação: em vez de possuir apenas dados estruturados, ele também possui comandos. Muitas pessoas criticam o uso do XML nesse sentido. Como é uma das ferramentas mais populares, as grandes IDEs, como Eclipse e Netbeans, já vem com suporte a ele.

Definição de tasks e targets

Queremos usar o Ant para ajudar nas tarefas mais comuns como compilar ou empacotar. Para cada uma das tarefas, ou tasks, o Ant possui uma tag que deve ser colocada em um arquivo XML, normalmente chamado de `build.xml` .

Por exemplo, o Ant define uma task `javac` para compilar o código fonte de um diretório para um destino:

```
<javac srcdir="src" destdir="build" />
```

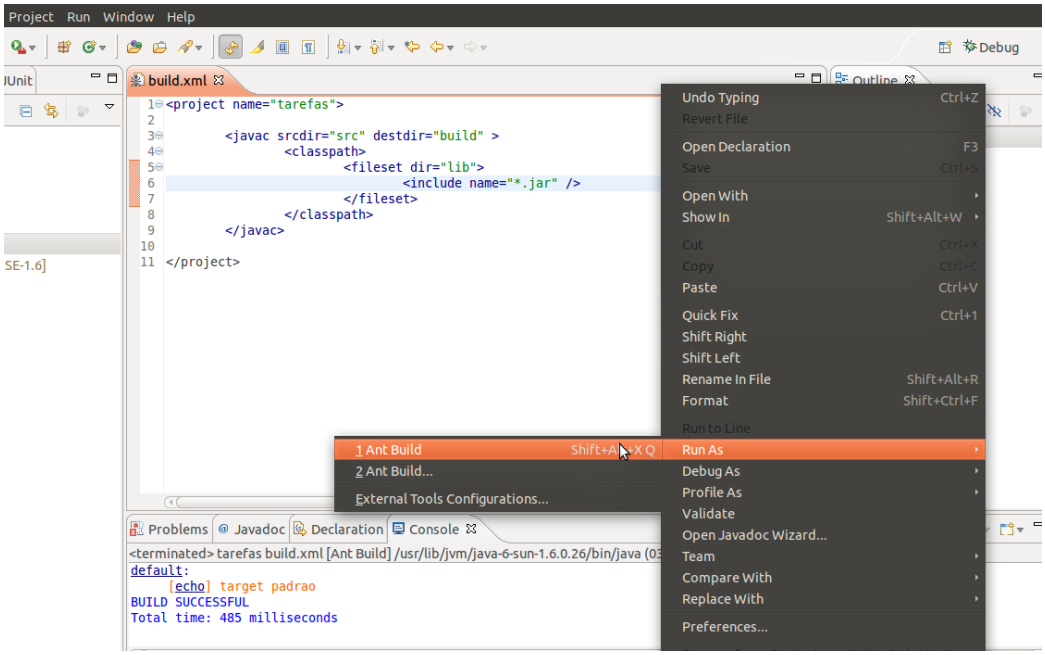
Porém muitas vezes as classes no diretório `src` dependem de outras bibliotecas (JARs). É preciso definir o classpath correto para a task `javac` funcionar. A tag `classpath` dentro da tag `javac` representa o classpath e define o tipo de arquivo que estamos procurando, normalmente um JAR:

```
<javac srcdir="src" destdir="build" >
    <classpath>
        <fileset dir="lib">
            <include name="*.jar" />
        </fileset>
    </classpath>
</javac>
```

A nossa tag `javac` faz parte do projeto como várias outras tarefas também. Todas as tarefas definem o projeto. Por isso as tarefas no arquivo `build.xml` devem ter uma tag mãe `project` na raiz do arquivo.

```
<project name="tarefas">

  <javac srcdir="src" destdir="build" >
    <classpath>
      <fileset dir="lib">
        <include name="*.jar" />
      </fileset>
    </classpath>
    </javac>

</project>
```

Já podemos executar o `build.xml` no Eclipse (Run as -> Ant build).



Porém a execução mostra um erro:

```
Buildfile: /workspace/tarefas/build.xml


BUILD FAILED /workspace/tarefas/build.xml:20: destination directory "/workspace/tarefas/build" does not exist or
is not a directory

Total time: 428 millisecon
```

No nosso exemplo as classes são compiladas no diretório `build`. Esse diretório deve existir antes de compilar. E para isso o Ant possui uma tarefa para a criação de diretórios, a task `mkdir`:

```
<mkdir dir="build" />
```

Também faz sentido garantir que o diretório esteja limpo, ou seja antes de criar, vamos apagar o antigo através da task `delete`:

```
<delete dir="build" />
```

Desse maneira o `build.xml` fica:

```
<project name="tarefas">

    <delete dir="build" />

    <mkdir dir="build" />

    <javac srcdir="src" destdir="build" >
      <classpath>
          <fileset dir="lib">
                <include name="*.jar" />
```

```
            </fileset>
        </classpath>
    </javac>

</project>
```

Ao chamar o `build.xml` no Eclipse será impresso no console:

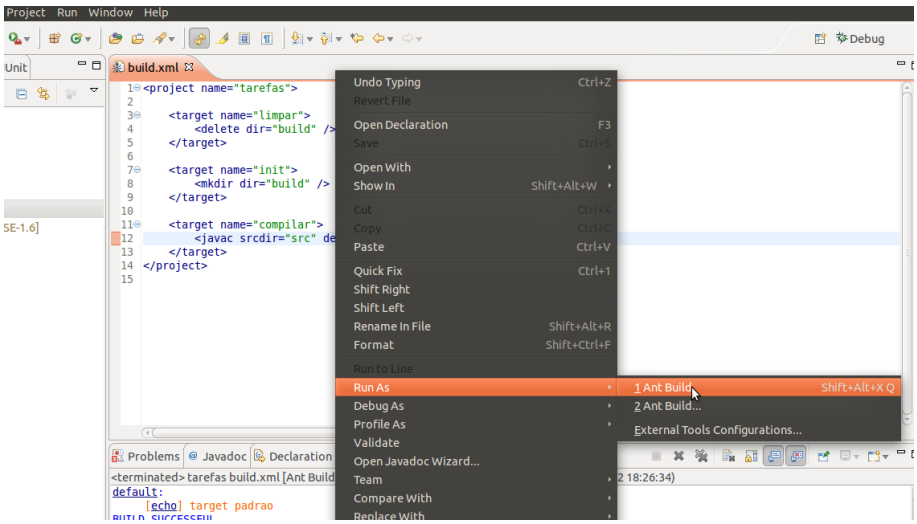
```
Buildfile: /workspace/tarefas/build.xml [delete] Deleting directory /workspace/tarefas/build [mkdir] Created dir: /workspace/tarefas/build [javac] Compiling 3 source files to /workspace/tarefas/build BUILD SUCCESSFUL Total time: 1 second
```

Vimos que Ant sempre executa as três tarefas `delete -> mkdir -> javac` , mas pode fazer todo sentido de executar uma tarefa apenas. Por exemplo, antes de fazer um backup do projeto inteiro é preciso apagar o diretório `build` . Como isolar uma tarefa?

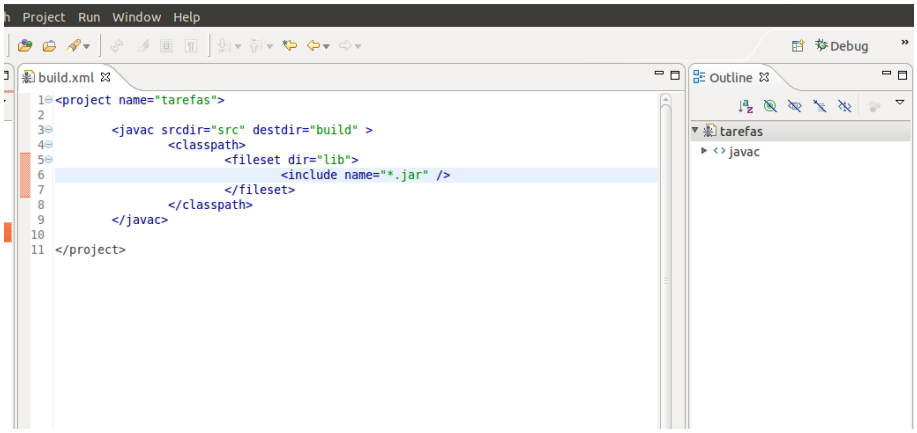
No Ant podemos organizar e agrupar as tarefas com targets. Esses `target` s funcionam como um contêiner de uma ou mais tarefas. Vamos organizar cada tarefa dentro de um target:

```
<project name="tarefas">

    <target name="limpar">
        <delete dir="build" />
    </target>

    <target name="init">
        <mkdir dir="build" />
    </target>

    <target name="compilar">
        <javac srcdir="src" destdir="build" >...</javac>
    </target>
</project>
```

Assim posso chamar um target isolado pelo Eclipse. Para tal, é preciso selecionar o target e chamar Run as:



Uma alternativa é a view `Outline` do Eclipse para a execução do target `compilar` .



Também posso definir um target padrão pela tag `project` caso não seja chamada um target específico, será sempre executado o padrão:

```
<project name="tarefas" default="compilar">
     ...
</project>
```

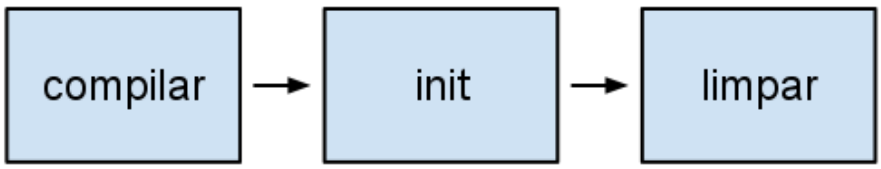
O target `compilar` é então o padrão, mas como já vimos, depende da existência do diretório `build` ou seja depende do target `init` . O target `init` por sua vez depende do target `limpar` . É muito comum que um target dependa de outro, e por isso a tag possui um atributo chamado `depends` para definir uma ou mais dependências:

```
<target name="compilar" depends="init">
    <javac srcdir="src" destdir="build" >...</javac>
</target>
```

O mesmo atributo podemos utilizar na target `init` :

```
<target name="init" depends="limpar">
    <mkdir dir="build" />
</target>
```

Assim, ao executar o target `compilar` será chamado anteriormente o `init` que chama antes `limpar` :



Gerando um JAR

Com nossas classes sendo compiladas, poderíamos disponibilizar o nosso projeto através de um JAR. Quando geramos um devemos primeiro dizer qual o destino e o nome dele, e o principal, a localização dos nossos `.class` . Para isso existe uma tarefa `jar` que recebe como atributos um `destfile` que representa o local onde o JAR será criado e outro atributo `basedir` representando o diretório de nossos arquivos compilados. Lembrando da necessidade de já ter executado o `target` compilar antes dessa de gerar um JAR, ou seja, declarar o atributo `depends` indicando que o target `empacotar` depende do target `compilar` . Teríamos um `target` como este:

```
    <target name="empacotar" depends="compilar" >
        <jar destfile="build/treinamento.jar" basedir="build" />
    </target>
```

Logo, para a execução do arquivo fazer a criação de um JAR deveríamos mudar o default para a `target` empacotar. Desse modo nosso arquivo de build ficaria como este:

```
<project name="tarefas" default="empacotar">

    <target name="limpar">
        <delete dir="build" />
    </target>

    <target name="init" depends="limpar">
        <mkdir dir="build" />
    </target>

    <target name="compilar" depends="init">
        <javac srcdir="src" destdir="build">
            <classpath>
                <fileset dir="lib">
                    <include name="*.jar" />
                </fileset>
            </classpath>
        </javac>
    </target>

    <target name="empacotar" depends="compilar">
        <jar destfile="build/treinamento.jar" basedir="build" />
    </target>

</project>
```

Dessa maneira o ANT vai tentar executar limpar->init->compilar->empacotar pois já criamos uma dependência entre elas.

```
Buildfile: /workspace/tarefas/build.xml limpar: [delete] Deleting directory /workspace/tarefas/build init:
[mkdir] Created dir: /workspace/tarefas/build compilar: [javac] Compiling 3 source files to
/workspace/tarefas/build empacotar: [jar] Building jar: /workspace/tarefas/build/treinamento.jar BUILD SUCCESSFUL
Total time: 1 second
```