***Project Object Model*** (literalmente "projeto modelo de objeto"), ou POM, é a peça fundamental de um projeto do [Apache Maven](https://pt.wikipedia.org/wiki/Apache_Maven).

Um POM possui as infomações básicas de um projeto, bem como as diretivas de como o artefato final deste projeto deve ser construido.

A versão 1.0 do [Maven](https://pt.wikipedia.org/wiki/Maven) utiliza o arquivo project.xml para definição do POM. Na versão 2.0, este arquivo passa a se chamar pom.xml

Link: <https://www.devmedia.com.br/introducao-ao-maven/25128>

**O Apache Maven é uma excelente ferramenta de apoio a qualquer equipe que trabalhe com projetos Java** (outras tecnologias também são suportadas), fornecendo aos desenvolvedores uma forma de automatizar e padronizar a construção e publicação de suas aplicações.

**Guia do artigo:**

* [**O que é o Maven?**](https://www.devmedia.com.br/introducao-ao-maven/25128#1)
* [**Instalando o Maven**](https://www.devmedia.com.br/introducao-ao-maven/25128#2)
* [**Criando um projeto simples**](https://www.devmedia.com.br/introducao-ao-maven/25128#3)
* [**Gerenciamento de Dependências**](https://www.devmedia.com.br/introducao-ao-maven/25128#4)
* [**O arquivo pom.xml**](https://www.devmedia.com.br/introducao-ao-maven/25128#5)
* [**Expandindo nosso exemplo**](https://www.devmedia.com.br/introducao-ao-maven/25128#6)
* [**Ciclo de Vida de Construção**](https://www.devmedia.com.br/introducao-ao-maven/25128#7)
* [**Plugins**](https://www.devmedia.com.br/introducao-ao-maven/25128#9)
* [**Criando um projeto web**](https://www.devmedia.com.br/introducao-ao-maven/25128#10)
* [**Dicas**](https://www.devmedia.com.br/introducao-ao-maven/25128#11)

**O Apache Maven é uma ferramenta de automação e gerenciamento de projetos Java**, embora também possa ser utilizada com outras linguagens. Ela fornece às equipes de desenvolvimento uma forma padronizada de automação, construção e publicação de suas aplicações, agregando agilidade e qualidade ao produto final. Por ser extremamente flexível, permite que sejam adicionados plugins a si, para estender suas funcionalidades nativas.

O processo de criação de um projeto [**Java EE**](https://www.devmedia.com.br/guia/java-enterprise-edition-java-ee/34474) em geral envolve a criação de um diretório principal com vários subdiretórios, a configuração de diversos arquivos XML, a obtenção (via cópia ou download) de bibliotecas para o projeto e, posteriormente, a execução dos testes unitários, a criação dos pacotes de publicação, a geração de documentação javadoc, entre outras etapas. Normalmente, até algum tempo atrás, cada projeto tinha sua própria estrutura, seu próprio jeito de gerar pacotes, de efetuar cada um destes passos. Projetos complexos, com vários módulos, ainda podem precisar que estes sejam compilados em determinada ordem, para que o pacote final seja criado.

Como se pode imaginar, facilmente isso poderia se tornar um pesadelo para os projetos em manutenção ou com um desenvolvimento mais extenso, em equipes que envolvam alguns desenvolvedores, porque as mudanças estruturais realizadas por um desenvolvedor têm que ser comunicadas a cada um dos demais colegas, para que o projeto continue a compilar no computador de cada um. Considere, por exemplo, que o desenvolvedor A precisou incluir uma biblioteca. Antes de colocar no controle de versão o código que a utiliza, ele precisaria avisar a cada um de seus colegas que adicionou algo novo no projeto, dizer de onde baixar o novo componente (ou passá-lo de computador em computador) e se assegurar que todos equalizaram seus ambientes de desenvolvimento, para que o código continue compilando nas máquinas de seus colegas. Tempos depois, o desenvolvedor B precisa efetuar uma manutenção no projeto e descobre que uma versão nova da biblioteca supracitada fornece uma funcionalidade nova, que facilitará o seu trabalho. Mais uma vez, o ciclo de aviso e cópia do novo componente começa, com perda de tempo da equipe e correndo o risco de algum computador ficar desatualizado e um desenvolvedor perder tempo tentando achar um erro acarretado pelo uso da biblioteca antiga.

Outra situação possível, em um projeto de vários módulos: um desenvolvedor modifica um dos módulos básicos do sistema e não avisa os demais colegas, ou estes não deram atenção ao e-mail de alerta. Como resultado o código ou para de compilar ou passa a ter comportamento instável, porque o módulo alterado não foi recompilado nos computadores do restante da equipe.

Agora, imagine os cenários acima em uma fábrica de software, em que o ritmo de trabalho é intenso e os prazos impõem pressão às equipes. Facilmente podemos prever que bibliotecas serão adicionadas sem que o restante do time seja avisado, acarretando impedimento de compilação do código mais recente do projeto. Pessoas modificariam módulos e se esqueceriam de avisar as demais para pegarem a versão mais recente de código e recompilar as alterações; mudanças no processo de geração de pacotes que não são comunicadas, e daí em diante.

Finalmente, se forem consideradas equipes distribuídas, com seus membros trabalhando em home office, a complexidade de propagar mudanças estruturais do projeto aumenta muito. Nesses casos, até mesmo iniciar o projeto podia trazer muitas dores de cabeça, até toda a equipe conseguir estabilizar o ambiente de desenvolvimento de cada participante.

**Saiba mais:**

[**Curso de Maven Framework**](https://www.devmedia.com.br/curso/curso-de-maven-framework/419)

Visando solucionar estes e outros problemas, algumas ferramentas foram desenvolvidas para ajudar na organização dos projetos e foram bem sucedidas em seu intento, sendo a mais famosa delas o Apache Ant. O trabalho manual diminuiu bastante, mas ainda havia questões pendentes, como a distribuição de bibliotecas entre a equipe, falta de aderência a um padrão estrutural único entre os projetos, uma forma padronizada de compilar e gerar pacotes, entre outras. Outro ponto importante era a necessidade de configurar minuciosamente essas ferramentas a partir de arquivos de configuração, o que gerava certo trabalho tanto na criação quanto na evolução do projeto.

**O Maven surgiu**, então, como uma melhoria das ferramentas existentes e também como uma resposta para as demais questões. Neste artigo, veremos como o Maven faz isso, através de um exemplo prático e muito simples, cujas instruções de criação, com poucas adaptações, poderão ser usadas como base para a maioria dos seus projetos.

**O que é o Maven?**

Em seu cerne, **o Maven é uma ferramenta de gerenciamento e automação de construção (*build*) de projetos**. Entretanto, por fornecer diversas funcionalidades adicionais através do uso de plugins e estimular o emprego de melhores práticas de organização, desenvolvimento e manutenção de projetos, é muito mais do que apenas uma ferramenta auxiliar.

Um desenvolvedor que seja alocado em um projeto [**Java EE**](https://www.devmedia.com.br/guia/java-enterprise-edition-java-ee/34474) que **utilize o Maven corretamente** não terá que saber de imediato quais dependências (bibliotecas) o projeto necessita para compilar e executar, não precisará descobrir onde obtê-las e nem irá se preocupar em como realizar a construção do pacote do aplicativo. Com um comando simples, como mvn install, na raiz do código-fonte do projeto, instruirá o Maven a gerar o código extra necessário (cliente de um web service, por exemplo), validar e compilar o projeto, testá-lo através de seus testes unitários e gerar o pacote com o código compilado. Outras etapas poderiam incluir auditoria de qualidade de código, documentação, geração de estatísticas, entre diversas possibilidades.

Outra **característica do Maven** é estimular a adoção de boas práticas, porque uma das formas utilizadas por ele para reduzir o esforço de configuração do projeto é a utilização do conceito de *programação por convenção* (do inglês *convention over configuration*), em que a ferramenta assume que o seu usuário fará as coisas da forma como ela preconiza como ideais (estrutura de diretórios padrão, por exemplo), e o livra de ter que declarar algo que se repetirá em todo projeto. o incorporar as práticas aceitas pela comunidade Java como as mais indicadas para projetos Java EE, o **Maven acaba não só disseminando-as para novos desenvolvedores**, como também as padroniza entre os projetos em que ele é empregado, permitindo que novatos se localizem muito mais rapidamente dentro de projetos novos. Obviamente, pode-se definir manualmente o que é assumido como padrão, ao preço do aumento na carga de trabalho para a configuração inicial do projeto.

O Maven hoje fornece suporte a outras tecnologias, inclusive .NET, mas neste artigo focaremos somente nos projetos Java EE.

**Instalando o Maven**

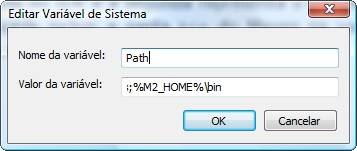
**A instalação do Maven** é bastante simples. Neste artigo, usaremos o Windows, porém estes podem ser facilmente adaptados para outros sistemas operacionais.

O único pré-requisito para a instalação do Maven é ter o [**Java Development Kit**](https://www.devmedia.com.br/introducao-ao-java-jdk/28896) (JDK) instalado, que pode ser obtido em *www.oracle.com/us/technologies/java/*, caso necessário. A versão do JDK utilizada para este artigo será a 1.6.0\_20.

O Maven propriamente dito é disponibilizado pela Apache Software Foundation em formato de arquivo compactado no endereço *http://maven.apache.org/download.html*, de onde baixaremos o arquivo zip. O artigo adotará como base a versão estável mais recente durante a sua escrita, a 3.0.4.

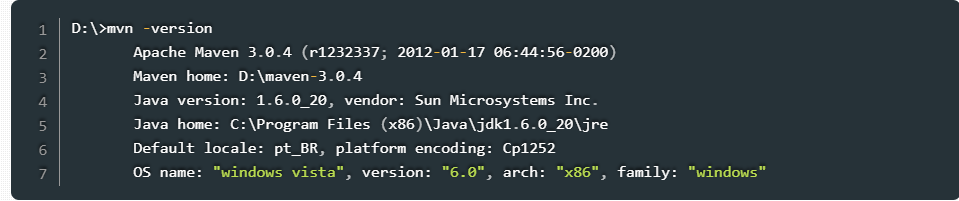
Após a conclusão do download, o arquivo deve ser descompactado em um diretório. Por uma questão de comodidade e padronização, recomenda-se que o nome deste diretório fique no formato maven-[versão]. Utilizando a nossa versão como exemplo, o nome que sugerimos é maven-3.0.4. Além disso, também é aconselhável que não haja espaços em nenhuma parte do nome físico completo da ferramenta. Com isso em mente, colocaremos o conteúdo do arquivo compactado no diretório D:\maven-3.0.4.

A próxima etapa é a criação de duas variáveis de ambiente no sistema operacional: JAVA\_HOME e M2\_HOME. A primeira indica a localização do diretório do JDK, e a segunda representa o mesmo para o Maven. Também é necessário acrescentar ao conteúdo da variável Path do sistema a pasta bin do Maven, conforme as **Figuras 1**, **2** e **3**.

Variável de ambiente que aponta para a
localização do JDK no sistema operacional  
**Figura 1.** Variável de ambiente que aponta para a localização do JDK no sistema operacional.Variável de ambiente que aponta para a
localização do Maven no sistema operacional  
**Figura 2.** Variável de ambiente que aponta para a localização do Maven no sistema operacional.  
**Figura 3.** Inclusão da pasta bin do Maven no Path do sistema operacional.

Para testar se a instalação foi realizada com sucesso, basta executarmos o comando mvn -version na linha de comando, conforme indica a **Listagem 1**.

**Listagem 1.** Teste de instalação do Maven.

D:\>

Se ocorrer algum erro nesta etapa, devemos verificar se os passos anteriores foram realizados e se as variáveis apontam para os devidos lugares.

**Criando um projeto simples**

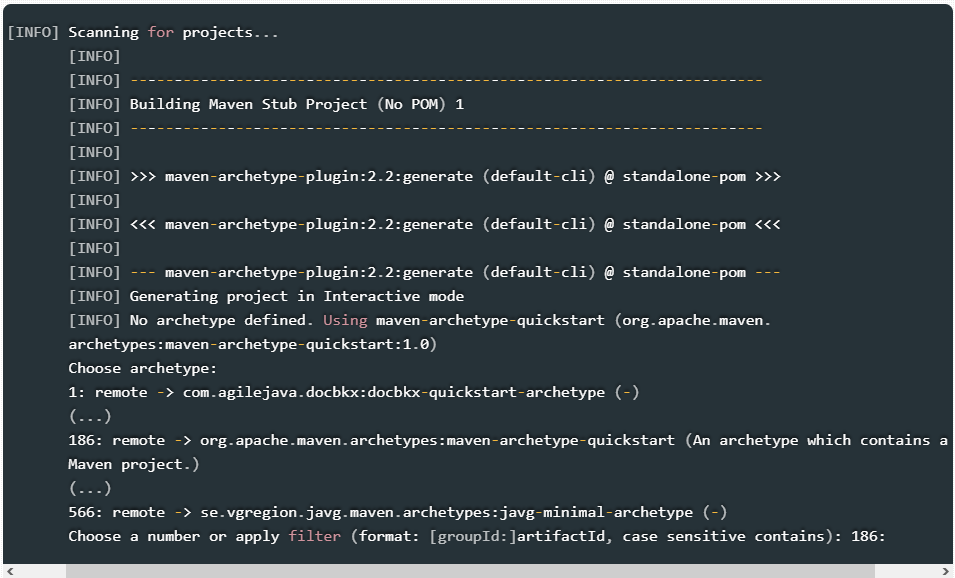
O Maven provê uma excelente funcionalidade de criação automática de projetos através de arquétipos. Os arquétipos são “esqueletos” que podem ser usados como base para projetos. Muitas vezes, criar um projeto Java EE do zero é um trabalho maçante, porque se deve gerar uma estrutura de diretórios, vários [**arquivos XML**](https://www.devmedia.com.br/introducao-a-xml-regras-de-sintaxe/24971) de configuração, definição de bibliotecas, e outros passos que podem (e devem!) ser automatizados. Com o uso de arquétipos, o Maven pode ser instruído a criar toda a estrutura necessária para se começar a desenvolver projetos dos mais diversos tipos, agilizando e muito o início do desenvolvimento.

Para nosso exemplo, criaremos um projeto utilizando o arquétipo mais simples, para analisarmos a estrutura que o Maven gera.

Dentro do diretório onde se deseja criar o projeto, executaremos o comando do Maven que nos permite manipular seus arquétipos. Se não receber parâmetros, o comando lista todos os arquétipos existentes para escolha do usuário. Neste artigo, optamos por utilizar como base para nossa demonstração o diretório D:\projetos, e sugerimos que o leitor faça o mesmo, para facilitar o acompanhamento dos exemplos. Doravante, nossos exemplos irão se referenciar a este diretório.

No prompt de comando, acesse o diretório onde deseja criar o projeto (*cd projetos*). Em seguida, digitando mvn archetype:generate, o Maven listará os arquétipos disponíveis, como exibido na **Listagem 2**.

**Listagem 2.** Listagem dos arquétipos.

[

Caso esta seja a primeira vez em que este comando esteja sendo executado, o Maven irá baixar os plugins que necessita para gerar a listagem de arquétipos. Este mesmo procedimento acontecerá cada vez que o usuário invocar uma função do Maven pela primeira vez, porque a ferramenta possui inteligência para acrescentar funcionalidades a si, por meio do download de plugins adicionais. Falaremos mais sobre plugins mais a frente.

Ao final da listagem, o Maven solicitará o número correspondente ao arquétipo de projeto a ser criado, sugerindo como padrão o arquétipo básico maven-archetype-quickstart, número 186, que utilizaremos. Vale ressaltar que a lista de arquétipos pode sofrer atualizações com o tempo, de modo que o número do arquétipo básico pode mudar. Em caso de dúvida, devemos aceitar o padrão sugerido pela ferramenta.

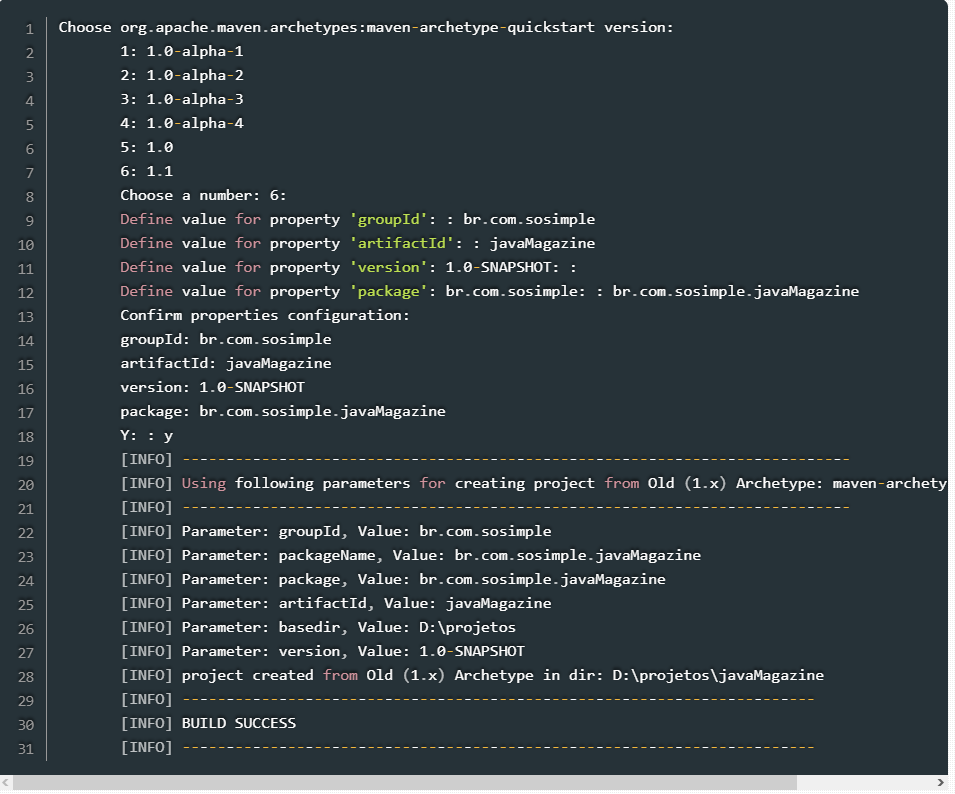
Depois de selecionarmos o arquétipo, escolhemos sua versão, sendo o padrão a mais recente, 1.1, que será nossa escolha.

O próximo passo é fornecermos os valores de groupId, artifactId, version e package.

GroupId é o nome que será usado como ponto de partida da hierarquia (grupo) do projeto no repositório Maven. Comumente assume o nome do pacote principal. Informaremos br.com.sosimple como groupId.

ArtifactId é o nome do projeto em si. Utilizaremos javaMagazine para este campo. Version é simplesmente a versão do seu projeto, o que é muito importante à medida que o projeto amadurece e novas versões são lançadas. Pode-se usar o valor padrão sugerido, 1.0-snapshot. O Maven utiliza essas três informações para definir a identidade de um artefato em particular, seja ele um projeto ou uma dependência. Por fim, package é o nome do pacote principal, que será br.com.sosimple.javaMagazine no nosso caso. Após entrarmos esses valores, basta confirmarmos com ‘Y’ e verificar o resultado. A **Listagem 3** mostra as escolhas que fizemos para nosso projeto.

**Listagem 3.** Definições do nosso projeto.



A partir do diretório onde o comando do Maven foi executado, um novo diretório será criado, com o nome que foi informado para o artifactId. Dentro dele, o Maven criou a estrutura padrão de diretórios que sugere como mais apropriado para melhor organização de projetos, conforme podemos ver na **Figura 4**.

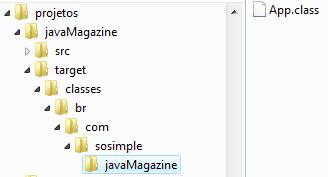
Estrutura padrão de diretórios, criada
automaticamente pelo Maven  
**Figura 4.** Estrutura padrão de diretórios, criada automaticamente pelo Maven.

Depois de criar o diretório javaMagazine, o Maven criou a pasta src, onde todo código-fonte deve ser mantido. Dentro de src, main e test também são criadas. A pasta main conterá o código-fonte propriamente dito do projeto, e test terá o código-fonte dos testes unitários. Finalmente, dentro destes, será criada a estrutura de pastas determinada pelo package, informado na criação do arquétipo, precedidas por um diretório pai chamado java.

Além desses diretórios, há outro criado pelo Maven sempre que é necessário compilar, construir ou gerar informações sobre o projeto: o diretório target. Para fins de exemplo, vamos pedir ao Maven para compilar nosso projeto neste momento, o que acarretará a sua criação. Portanto, no diretório D:\projetos\javaMagazine, execute o comando mvn compile para ver a criação da pasta target, exibida na **Figura 5**.

Diretório target, conforme criado pelo
Maven  
**Figura 5.** Diretório target, conforme criado pelo Maven.

Após a execução do comando, todo código do projeto foi compilado e o seu resultado colocado dentro do diretório target, conforme podemos notar por meio da criação do arquivo App.class, correspondente à compilação da classe br.com.sosimple.javaMagazine.App, no local exibido na **Figura 6**.

  
**Figura 6.** Diretório target expandido.

É muito importante não criarmos nem editarmos nenhum arquivo que esteja dentro de target, pois este diretório tem seu conteúdo rotineiramente alterado pelo Maven. Isso acontece porque, para refletir o código presente no diretório src, os seus comandos de compilação e construção vão sempre sobrescrever o conteúdo de target com base no conteúdo de src.

**Gerenciamento de Dependências**

Antes de prosseguirmos, precisamos mencionar como o Maven lida com as dependências. Praticamente todo projeto utiliza bibliotecas nativas ou disponibilizadas por terceiros para reuso de código ou para obter funcionalidades extras. Gerenciar essas bibliotecas, chamadas de dependências no jargão do Maven, pode se tornar uma dor de cabeça em projetos com várias delas.

O Maven fornece uma maneira fácil de administrar as bibliotecas a partir do arquivo pom.xml que, entre outras coisas, contém a lista de dependências que um projeto utiliza. Com esta lista, a ferramenta as analisa e tenta localizá-las para disponibilizar para o projeto. Os lugares onde o Maven procura por dependências chamam-se repositórios, cujos tipos principais são local e público.

O repositório local fica no computador onde se está rodando o Maven. Ele está localizado na pasta .m2/repository dentro da pasta home do usuário (no computador onde este artigo foi escrito, C:\Users\ottero\.m2\repository). Este é o primeiro lugar onde a ferramenta procura toda vez que precisa localizar uma dependência. Caso não a encontre, irá verificar o repositório público oficial, localizado em *http://repo.maven.apache.org/maven2/*. Se encontrar a dependência que precisa, o Maven primeiro a copia para o repositório local e depois a disponibiliza para o projeto, de modo que não será mais necessária uma consulta ao repositório público quando precisar desta mesma dependência no futuro. Além destes repositórios, podemos informar que utilizaremos dependências de outros repositórios, bastando, para isso, registrá-los no pom.xml. Em certos ambientes corporativos, onde diretivas de segurança vetem o acesso livre à internet, é comum a existência de repositórios na rede local que contêm as bibliotecas homologadas pela corporação para uso em seus projetos.

Depois de localizadas as bibliotecas necessárias no repositório local, o Maven as utiliza durante a execução de seus comandos. Assim, se o comando for mvn compile, o Maven irá utilizar as dependências para compilar o projeto; se for mvn package em um projeto web, as dependências serão inseridas dentro do arquivo WAR gerado pelo empacotamento do projeto, e o WAR resultante será criado dentro do diretório target, que, conforme vimos anteriormente, é criado e gerenciado pela ferramenta; se for mvn dependency:copy-dependencies, o Maven disponibilizará as bibliotecas para o projeto, colocando-as dentro do diretório target.

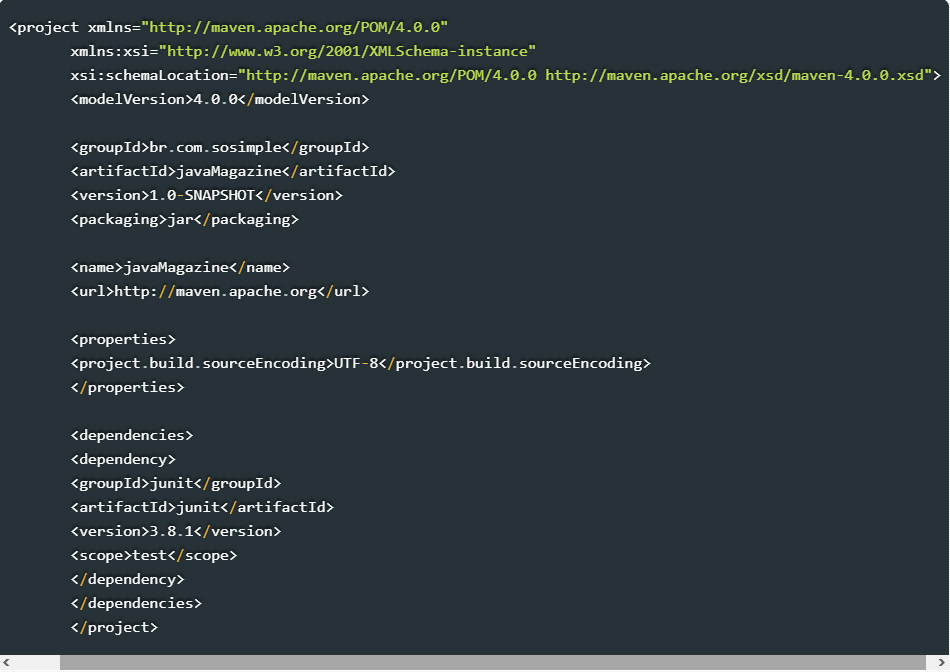
Outra capacidade da ferramenta é identificar as dependências das dependências do projeto. Ou seja, se uma biblioteca depender de outras, o Maven irá procurar por elas também e assim sucessivamente, até todas as dependências necessárias estarem à disposição do projeto.

**O arquivo pom.xml**

Este arquivo, presente no diretório-raiz do projeto, contém todas as configurações que o Maven necessita para interagir corretamente com o projeto. Ele pode ser simples, somente possuindo as coordenadas do projeto, ou extremamente complexo, relacionando dependências, repositórios, repositórios de plugins, plugins específicos, estratégias de construção do projeto, perfis, entre outros. Na medida em que formos prosseguindo em nosso artigo, veremos alguns desses itens.

Neste momento, o arquivo pom.xml de nosso exemplo está conforme foi criado pelo Maven (**Listagem 4**).

**Listagem 4.** pom.xml, conforme criado pelo Maven.



Há três informações muito importantes neste arquivo: as coordenadas do projeto, ou seja, os dados que o identificam (groupId, artifactId e version); o formato de publicação gerado ao se construir o projeto (jar, war, ear, etc.); e a declaração de que este depende da biblioteca JUnit versão 3.8.1, com a definição de que o escopo da mesma é test, ou seja, que estará disponível para o projeto durante a compilação e na fase de testes, mas não estará presente no pacote do projeto.

**Expandindo nosso exemplo**

Para explorarmos alguns detalhes do pom.xml, vamos criar um serviço no projeto para recuperar os números sorteados na mega-sena. Iremos usar uma biblioteca da Fundação Apache chamada HttpClient (*http://hc.apache.org/httpcomponents-client-ga/index.html*) para se conectar ao site da Caixa Econômica Federal como se fosse um cliente HTTP (simulando um navegador web) e recuperar essas informações.

Para isso, acrescentaremos a biblioteca que usaremos na seção dependency do pom.xml, conforme a **Listagem 5**.

**Listagem 5.** Incluindo uma dependência no pom.xml.

(...)

A seguir, no diretório D:\projetos\javaMagazine, execute o comando mvn eclipse:eclipse para gerar um projeto para a IDE Eclipse (para criar os exemplos, baixamos a versão *Eclipse IDE for Java EE Developers* em *http://www.eclipse.org/downloads/*).

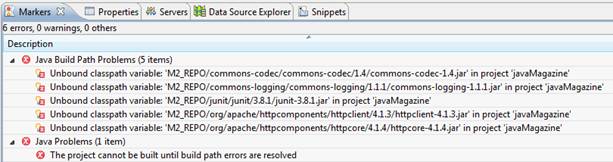
Após iniciarmos a IDE, devemos clicar em *File > Import*, escolhendo a opção *Existing Projects into Workspace* seguido de *Next*, como mostrado na **Figura 7**.

Iniciando wizard de importação de
projetos no Eclipse  
**Figura 7.** Iniciando wizard de importação de projetos no Eclipse.

Feito este procedimento, o Eclipse solicitará a localização do projeto a ser importado. A **Figura 8** ilustra como devemos preencher a janela. Note que não marcamos *Copy projects into workspace* porque continuaremos aplicando comandos do Maven em cima do diretório onde o projeto foi criado.

Importando o projeto javaMagazine no
Eclipse  
**Figura 8.** Importando o projeto javaMagazine no Eclipse.

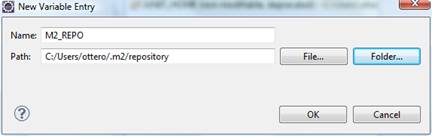
Depois da importação do projeto, o Eclipse detectará alguns erros de compilação, como demonstra a **Figura 9**.

  
**Figura 9.** Erros de compilação no Eclipse.

Estes erros ocorrem porque o Maven criou o projeto para o Eclipse usando uma variável chamada M2\_REPO, que aponta para o repositório local do usuário, que mencionamos anteriormente. É a forma com a qual a ferramenta informa onde estão as bibliotecas utilizadas pelo projeto. Caso a variável não exista na IDE, deverá ser criada, o que faremos agora. Para isso, clique com o botão direito no nome do projeto e acesse *Build Path > Configure Build Path*. A **Figura 10** ilustra o procedimento.

Configurando o build path do projeto
no Eclipse  
**Figura 10.** Configurando o build path do projeto no Eclipse.

Vamos criar a variável M2P\_REPO. Na aba *Libraries*, podemos ver as diversas formas que a IDE disponibiliza para incluirmos bibliotecas no projeto. Podemos incluir as que existem dentro do diretório do projeto, ou que estejam em locais externos, entre outras opções. No nosso caso, que precisamos criar uma variável, clicaremos no botão *Add Variable*. Na janela seguinte, que exibe as variáveis presentes na IDE, escolheremos *Configure Variables...* e, finalmente, na próxima janela, a criaremos, clicando em *New*. As **Figuras 11** a **14** ilustram os passos a serem efetuados.

Configuração da localização das
bibliotecas utilizadas pelo Eclipse  
**Figura 11.** Configuração da localização das bibliotecas utilizadas pelo Eclipse.Lista de variáveis que o Eclipse
utiliza para resolver a localização de diretórios  
**Figura 12.** Lista de variáveis que o Eclipse utiliza para resolver a localização de diretórios.Administração das variáveis do
Eclipse  
**Figura 13.** Administração das variáveis do Eclipse.  
**Figura 14.** Inclusão da variável M2\_REPO.

Podemos notar que, apesar de termos adicionado apenas uma nova dependência ao pom.xml, httpclient, o Maven acrescentou ao projeto, em seu build path, as dependências desta nova biblioteca: commons-codec, commons-logging e httpcore. Consulte novamente a **Figura 11** e observe que elas foram adicionadas ao projeto.

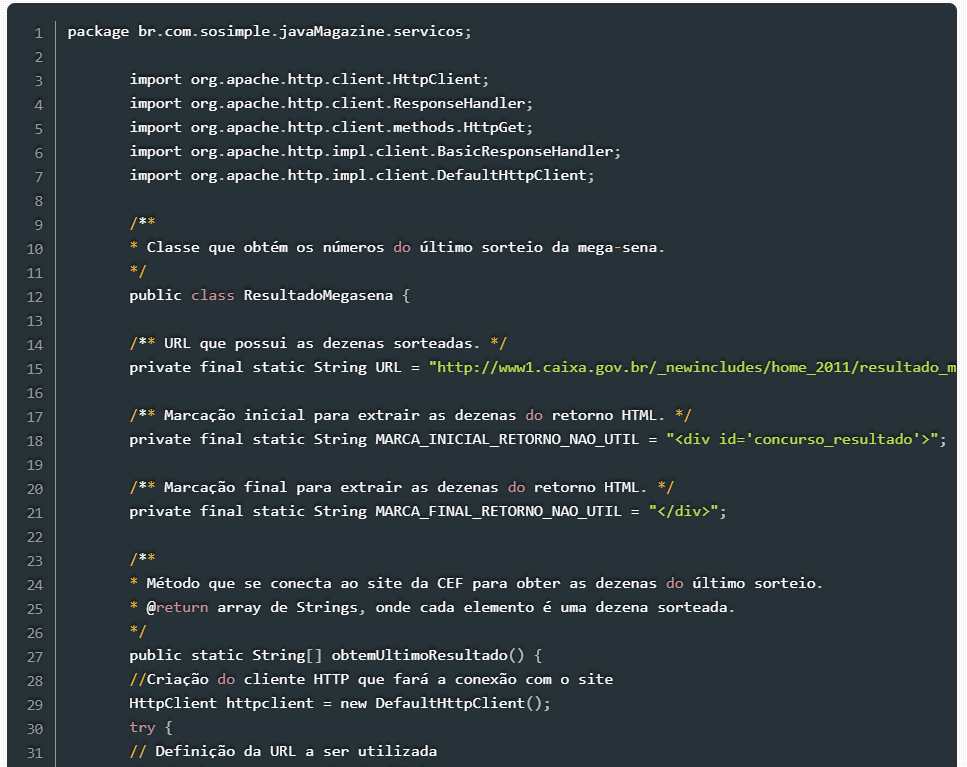
Agora o projeto do Eclipse não possui mais erros de compilação, de modo que podemos prosseguir na nossa codificação para recuperarmos o resultado do último sorteio da mega-sena.

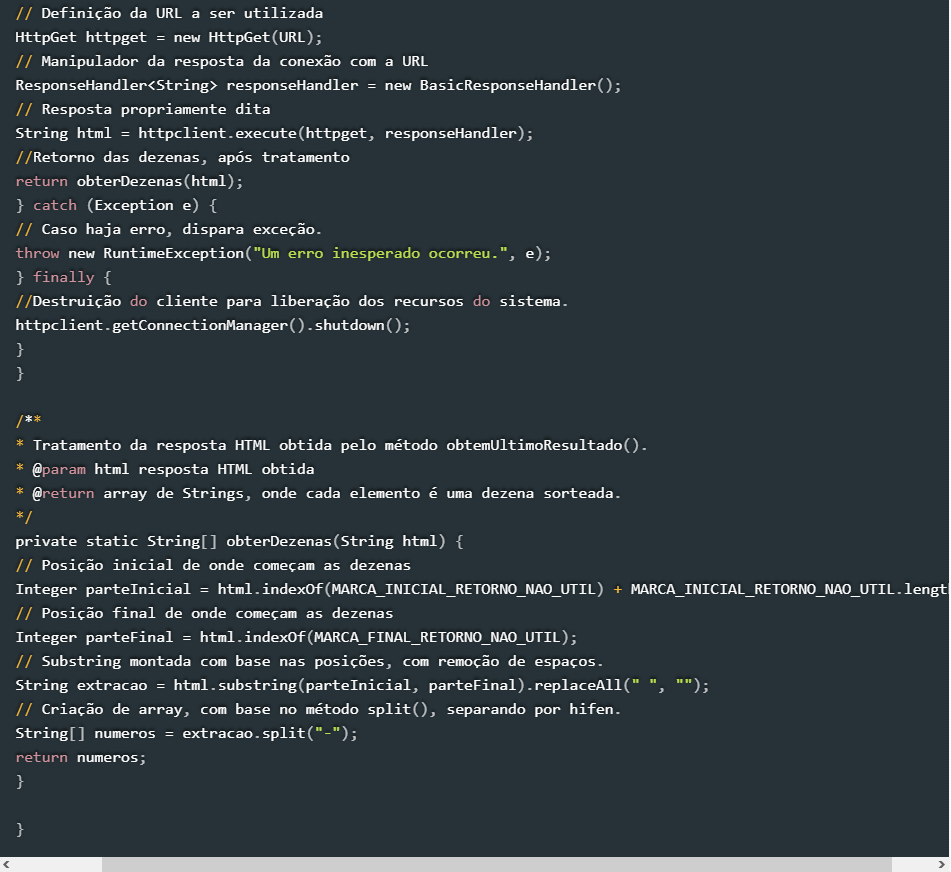
Como o propósito do artigo é demonstrar as funcionalidades do Maven, o código do projeto não leva em consideração as boas práticas, para focar apenas na ferramenta.

O site da Caixa Econômica Federal, http://www.caixa.gov.br/, utiliza uma URL para exibir os números sorteados: http://www1.caixa.gov.br/\_newincludes/home\_2011/resultado\_megasena.asp.

Iniciando nosso desenvolvimento, vamos criar, dentro do diretório src/main/java, uma classe chamada ResultadoMegasena, no pacote br.com.sosimple.javaMagazine.servicos, com o código exibido na **Listagem 6**.

**Listagem 6.** Código da classe ResultadoMegasena.

 }

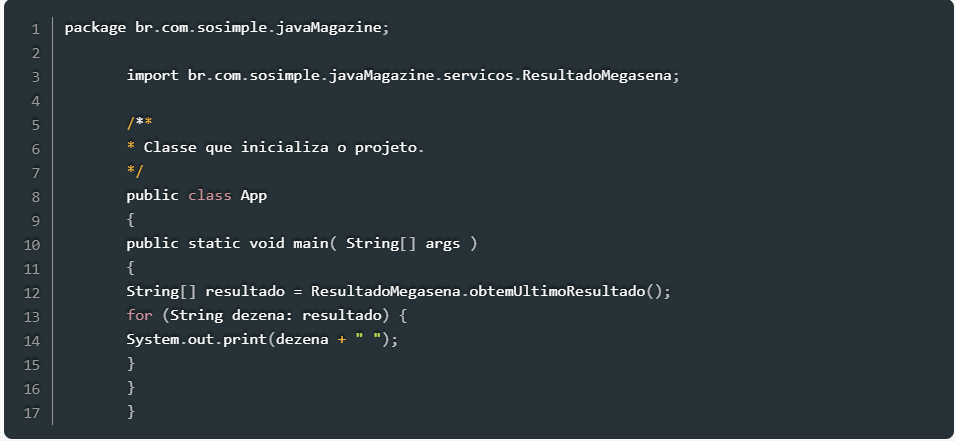


}

Basicamente, o código acima instancia um cliente HTTP e recupera o conteúdo da URL com o resultado da mega-sena sob a forma de String. Isto é feito no método obtemUltimoResultado(). A seguir, a String é enviada para o método obterDezenas(), que extrai as dezenas que queremos.

Precisamos agora instanciar ResultadoMegasena para que possamos recuperar as dezenas, e, para isso, alteraremos a classe criada pelo Maven, App, para ficar com o conteúdo da **Listagem 7**.

**Listagem 7.** Código da classe App atualizado.

 }

}

}

Por último, dentro da pasta src/test/java, vamos criar uma simples classe de teste unitário para ResultadoMegasena, que se chamará ResultadoMegasenaTest, no pacote br.com.sosimple.javaMagazine.servicos, cujo conteúdo está na **Listagem 8**. Basicamente, como queremos testar a classe ResultadoMegasena, precisamos chamar os seus métodos que desejamos testar e verificar se os resultados que esperamos aconteceram. No nosso caso, como apenas queremos ilustrar a integração dos testes unitários com o Maven e não demonstrar melhores práticas no desenvolvimento de testes, estes são muito simples: apenas verificam se o retorno do método foi diferente de nulo e posteriormente se o array retornado é composto de seis elementos.

**Listagem 8.** Código da classe de teste unitário para a classe ResultadoMegasena.



}

Quando executarmos a maioria dos comandos do Maven, todos os testes unitários do projeto serão executados automaticamente, e os seus resultados serão exibidos na tela e armazenados dentro de arquivos texto e XML, presentes no diretório target/surefire-reports. Outra forma que podemos utilizar para testar nosso projeto, mas desta vez manualmente, é rodando a classe App, e o retorno desta execução aparecerá no console do Eclipse. Caso ocorra algum problema na execução do projeto por alguma dessas formas, devemos rever se todos os passos anteriores foram corretamente realizados.

**Ciclo de Vida de Construção**

Um ponto central do Maven é o conceito de ciclo de vida de construção (*build lifecycle*), que significa que os procedimentos de construção são definidos em etapas chamadas de estágios do ciclo de vida. Esses estágios ocorrem de forma cumulativa. Deste modo, se, digamos, o estágio 4 de um ciclo de vida for invocado, os estágios anteriores também serão executados ordenadamente, ou seja, os estágios 1, 2 e 3 serão acionados antes do Maven ativar o estágio 4.

Os três ciclos de vida de construção que são nativos da ferramenta são o padrão (default), o clean e o site. O primeiro gerencia a implementação do projeto, o segundo a limpeza do projeto e o terceiro compreende a criação automatizada da documentação do projeto. Destes três, o principal ciclo é o padrão, que examinaremos com mais detalhes neste momento. Os outros serão exemplificados mais a frente.

O ciclo de vida padrão possui mais de 20 estágios, dos quais os mais importantes são:

* validate: valida se todas as informações obrigatórias do projeto estão preenchidas e são válidas;
* compile: compila o código-fonte;
* test: executa os testes unitários presentes no projeto, desde que o framework de testes utilizado seja compatível com o Maven. Por padrão, é usado o jUnit;
* package: empacota o código compilado no formato definido pela tag packaging do pom.xml;
* integration-test: caso exista, o pacote gerado no estágio anterior é instalado em um ambiente de teste de integração;
* verify: executa checagens para verificar se o pacote é válido e se atende aos critérios de qualidade;
* install: instala o pacote no repositório local, de modo que outros projetos locais possam utilizá-lo como dependência
* deploy: instala o pacote em repositórios externos, efetivamente disponibilizando-o em ambientes remotos.

Como dissemos anteriormente, se o estágio test for invocado, primeiro os estágios validate e compile serão executados. Rotineiramente, costumamos invocar o estágio install, que engloba todos os outros anteriores e, ao final de sua execução, temos o código fonte todo compilado, pronto para execução e disponibilizado como dependência para outros projetos locais.

**Conclusão**

O Maven é uma ferramenta extremamente poderosa, fantasticamente versátil e flexível. Ele agrega valor aos projetos onde é empregado, por causa de sua maturidade, filosofia organizacional, praticidade e, graças aos vários plugins existentes, de suas inúmeras funcionalidades, que estão sempre sendo expandidas.

Por adotar o conceito de “*programação por convenção*”, a ferramenta abstrai diversos detalhes que poderiam atrapalhar o seu uso cotidiano, ao mesmo tempo em que permanece aberta a configurações específicas, personalizadas para cada necessidade.

No próximo artigo, veremos como utilizar nossas próprias dependências, como construir o esqueleto de um projeto web a partir do Maven, como utilizar as opções de auditoria e analisar em mais detalhes o seu funcionamento e suas potencialidades!

Como dissemos na primeira parte, ele, em si, é uma ferramenta de gerenciamento e automação de construção (build) de projetos. No entanto, com a disponibilidade de inúmeros plugins para os mais variados fins, o Maven transcende sua proposta original, tornando-se uma potente fonte de funcionalidades auxiliares para o projeto.

Mostramos ainda a estratégia da ferramenta para facilitar o controle das bibliotecas utilizadas pelo projeto, ao assumir a responsabilidade por obtê-las e as instalar automaticamente. Isso, claro, presumindo que o Maven esteja bem configurado e que as bibliotecas sejam públicas ou estejam disponíveis no ambiente em que a ferramenta está sendo empregada.

Por fim, vimos também como a ferramenta estimula o emprego de boas práticas na implementação de soluções, ao adotar o conceito de programação por convenção (do inglês convention over configuration), no qual o Maven assume automaticamente que o seu usuário irá empregar a mesma metodologia e padrão de organização que ele considera ideal, poupando-o do trabalho tedioso de sempre ter que descrever as mesmas informações em cada projeto, como quais diretórios contêm código a ser compilado, o destino dos artefatos compilados, etc. Ao empregar inerentemente as práticas sugeridas pela comunidade, a ferramenta acaba as divulgando e estimulando que os projetos também as utilizem, aumentando a padronização e diminuindo a curva de aprendizado para novos desenvolvedores. Obviamente, qualquer convenção destas pode ser sobrescrita pelo usuário, que então deverá informar como quer que a ferramenta se comporte para seu projeto, prática somente aconselhada para casos de exceção, como projetos legados.

Apresentamos alguns princípios básicos do Maven, como o uso de arquétipos para a criação de projetos Java, o conceito de dependências (bibliotecas utilizadas pelo projeto), o arquivo pom.xml (o coração da ferramenta, responsável por sua configuração e controle de todas as suas funcionalidades) e explicamos seu ciclo de vida de construção. Ao mesmo tempo em que explanávamos os princípios, os aplicávamos na prática, o que culminou na criação de um pequeno projeto exemplo, cujo propósito era exibir as últimas dezenas sorteadas pela Mega-Sena.

Prosseguindo com o nosso estudo, vamos rodar o estágio install do ciclo de vida padrão, que é um dos comandos mais básicos do Maven, para que possamos analisar as etapas de construção do pacote de publicação. Como vimos no artigo prévio, o estágio **install**, ao executar, automaticamente invoca os estágios anteriores a si, instruindo a ferramenta a validar, compilar, testar e empacotar o projeto.

Para isso, vamos executar, dentro do diretório do projeto (caso tenha sido adotada a sugestão do artigo prévio, será D:\projetos\javamagazine), o comando: mvn clean install. Utilizamos o clean antes do install para instruir a ferramenta a descartar todo o código já gerado por ela para o projeto antes de começar o processamento dos demais ciclos de vida. Na prática, o clean apaga o diretório target.

O resultado da execução deste comando está na **Listagem 1**.

**Listagem 1.** Resultado do comando mvn clean install.

D:\projetos\javaMagazine>mv

(...)

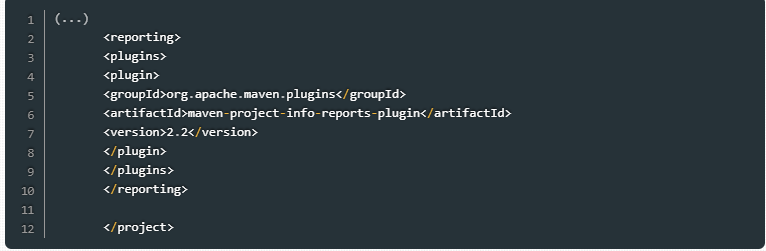
Com base no retorno obtido, podemos perceber alguns pontos importantes para a compreensão do funcionamento do Maven, marcados em vermelho:

1. O ciclo de vida clean é executado e o diretório target é removido. Se ocorrer algum problema neste ponto, provavelmente é porque existe algum arquivo desta pasta aberto, ou algum processo do sistema operacional está utilizando arquivos dentro deste diretório. Esse último caso costuma ocorrer com pacotes web (WAR) que estão instalados em servidores web ativos;
2. Geração do código compilado do projeto, e sua disponibilização dentro do diretório target/classes;
3. Criação do diretório target\surefire-reports, onde são publicados os resultados dos testes automatizados do projeto. Cada classe de teste executada gera dois arquivos dentro deste diretório, um em formato texto e outro em formato XML, com o nome da classe. O arquivo texto contém o resultado da execução do teste e o XML guarda as propriedades do ambiente onde os testes foram realizados;
4. Apresenta o resultado dos testes. Caso ocorra alguma falha em algum dos testes unitários, devemos olhar o resultado do teste falho no diretório citado no item 3;
5. Empacotamento do projeto, no formato definido pela tag packaging do pom.xml;
6. Disponibilização do pacote gerado no repositório local.

Após o término, encontraremos o pacote gerado dentro da raiz do diretório target: javaMagazine-1.0-SNAPSHOT.jar. Podemos notar que o nome padrão para o pacote se constitui do artifactId e da versão do artefato, separados por um hífen.

Vamos agora usar o ciclo de vida site para gerar a documentação do projeto. Se executarmos mvn clean site neste momento, o comando irá funcionar, porém o Maven irá nos alertar de que não definimos a versão que queremos utilizar do plugin que é empregado pela ferramenta para a documentação. Então, antes de seguirmos em frente, precisamos acrescentar mais uma seção ao nosso pom.xml, conforme mostrado na **Listagem 2**.

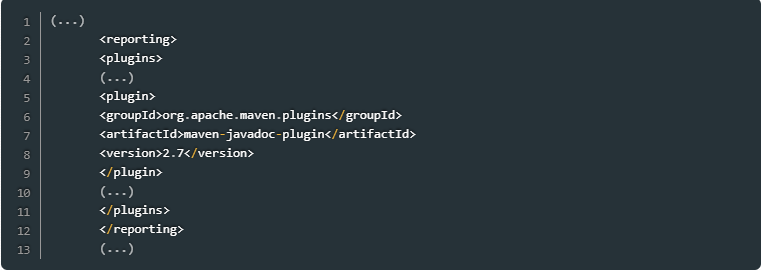
**Listagem 2.** Acrescentando o plugin de relatório.

(...)

A seção apresentada nesta listagem pode ser inserida em qualquer ponto do pom.xml. Por comodidade, sugerimos colocá-la logo antes da tag final do arquivo, ou seja, </project>. Feito isso, ao executarmos o comando mvn clean site, o diretório site será criado dentro de target e nele haverá um arquivo chamado index.html. Ao abrimo-lo em um navegador, veremos que foi montado um pequeno site informativo do projeto, com diversas informações úteis, como a árvore de dependências do projeto.

Outra funcionalidade útil do Maven é a geração simplificada do Javadoc do projeto. Para tanto, basta incluir o plugin do Javadoc na seção reporting, conforme exemplo mostrado na **Listagem 3**. Esta seção do pom.xml é destinada aos plugins que geram relatórios sobre o código-fonte.

**Listagem 3.** Acrescentando o plugin Javadoc.

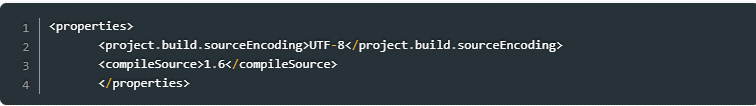


Para acionar o plugin, basta usar o comando mvn clean javadoc:jar, que cria a documentação empacotada sob a forma de um JAR, de nome javaMagazine-1.0-SNAPSHOT-javadoc.jar, e o diretório target\apidocs. O arquivo index.html é o ponto de entrada para a documentação da API. Para criarmos somente o diretório com a documentação, sem a criação do jar supracitado, podemos utilizar o comando mvn clean javadoc:javadoc.

Com a documentação gerada, vamos então testar a qualidade do código-fonte. Para isso, lançaremos mão de uma ferramenta open source já consolidada no mercado, o PMD, que verifica código Java em busca de problemas, como código que nunca será utilizado, complexidade desnecessária, más práticas, entre outros. Para mais informações sobre o PMD, consulte o site da ferramenta, em: *http://pmd.sourceforge.net/*.

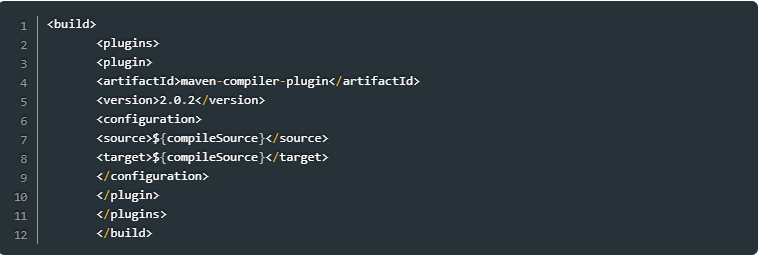
Antes, precisaremos fazer algumas configurações no arquivo pom.xml. Por padrão, o código gerado pela compilação do Maven é compatível com o JDK 1.5, enquanto o PMD analisa o código com base no JDK 1.4. Por causa disso, teremos que garantir que tanto o compilador do Maven quanto o PMD utilizem a mesma versão de JDK nas suas execuções.

Assim, iremos declarar uma propriedade dentro da seção properties do pom.xml. O resultado final será:



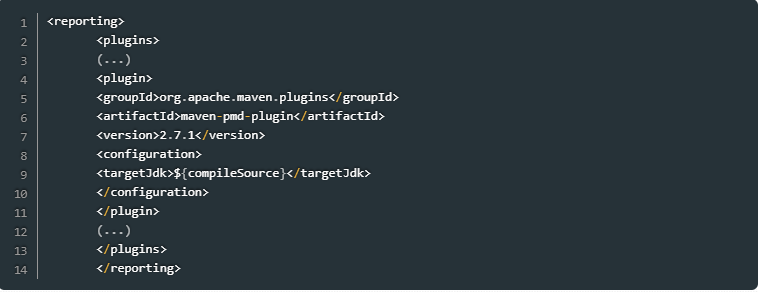
Em seguida, precisamos declarar explicitamente qual a versão do Maven compiler a ser utilizada, bem como a do PMD. No caso do compiler, acrescentaremos a seção build, que é a seção onde definimos os plugins utilizados para geração de código – ver Listagem 4.

**Listagem 4.** Acrescentando o plugin maven-compiler.



Para o plugin do PMD, iremos acrescentar sua definição dentro da seção reporting, para que fique de acordo com a**Listagem 5.**

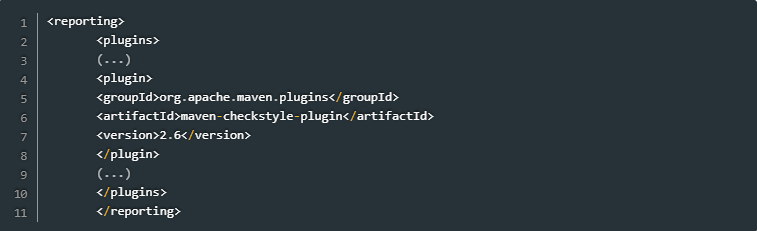
**Listagem 5.** Acrescentando o plugin PMD.



Podemos notar que a propriedade ${compileSource} é usada tanto para o plugin compiler quanto para o pmd. Com esta preparação, podemos executar o ciclo de vida site, através de mvn clean site, que incluirá o plugin PMD. Logo após, dentro do diretório target/site, encontraremos dois arquivos gerados pelo PMD: pmd.xml e pmd.html. No entanto, ambos estarão vazios, porque o projeto não possui nenhum problema. Sugerimos experimentar este plugin em um projeto profissional, para avaliação de um código-fonte real.

Ainda no quesito qualidade, podemos também utilizar outro plugin; desta vez da ferramenta Checkstyle, que é empregada para verificar a aderência aos padrões de codificação, como indentação, uso de boas práticas, formatação de código, presença de javadoc nos métodos, etc. Acrescentemos então o plugin do Checkstyle na seção reporting, conforme a **Listagem 6.**

**Listagem 6.** Acrescentando o plugin Checkstyle.

<

Em seguida, basta executarmos novamente o comando mvn clean site e checar o resultado em target/site, no arquivo checkstyle.html. Agora, nosso exemplo apresentou várias infrações (de acordo com as recomendações da ferramenta), como linhas com mais de 80 caracteres, parâmetros sem o modificador final, entre outras.

Tanto o PMD quanto o Checkstyle fornecem a possibilidade de configuração personalizada de suas regras e de suas validações, através de arquivos XML. Para mais informações, sugerimos consultar os endereços: *http://maven.apache.org/plugins/maven-pmd-plugin/* e *http://maven.apache.org/plugins/maven-checkstyle-plugin/*.

Por fim, vale a pena ressaltar que o Maven também pode gerar todas essas informações de auditoria de forma gráfica, através do comando mvn clean site dashboard:dashboard, que irá gerar o arquivo dashboard-report.html, dentro de target/site.

**Plugins**

Todas as funcionalidades do Maven são propiciadas através de plugins. Cada comando que nós damos é, na verdade, uma invocação a um plugin diferente, como clean, install, site, etc. Na página *http://maven.apache.org/plugins/*, temos a listagem dos plugins oficiais existentes. Outro site com diversos plugins úteis é *http://mojo.codehaus.org/plugins.html*, como um que integra o Maven com a ferramenta Hibernate, outro que executa instruções SQL e um que permite ao Maven se comunicar com o servidor de aplicação JBoss.

Por causa de sua arquitetura orientada a plugins, é possível perceber que o Maven permite a adição transparente de funcionalidades conforme a evolução do projeto, além de permitir que qualquer um crie um plugin para atender os requisitos que não estejam cobertos pelos plugins existentes. Por exemplo, pode-se querer que o Maven publique a documentação gerada automaticamente em um servidor web da empresa, de acordo com certos padrões; pode-se desejar que o Maven automatize procedimentos de projetos legados, substituindo trabalho manual. Como pode ser observado, as possibilidades são infinitas.

**Criando um projeto web**

Para demonstrar a criação de um projeto web através do Maven, desenvolveremos um aplicativo que irá exibir, em ambiente web, as dezenas sorteadas da Mega-Sena e, para obter essa informação, utilizaremos nosso projeto javaMagazine.

Assim, utilizaremos um arquétipo para criar nosso projeto web e posteriormente incluiremos o projeto javaMagazine como dependência dele. Desta vez, entretanto, mostraremos outra forma de utilizar a criação de projetos por arquétipo, passando as coordenadas do projeto como parâmetros. Portanto, dentro do diretório onde se deseja criar o projeto, execute: mvn archetype:generate -DarchetypeArtifactId=maven-archetype-webapp -DartifactId=javaMagazineWeb -DgroupId=br.com.sosimple -Dversion=1.0-SNAPSHOT -Dpackage=br.com.sosimple.javaMagazineWeb (no nosso caso, foi D:\projetos\javaMagazineWeb).

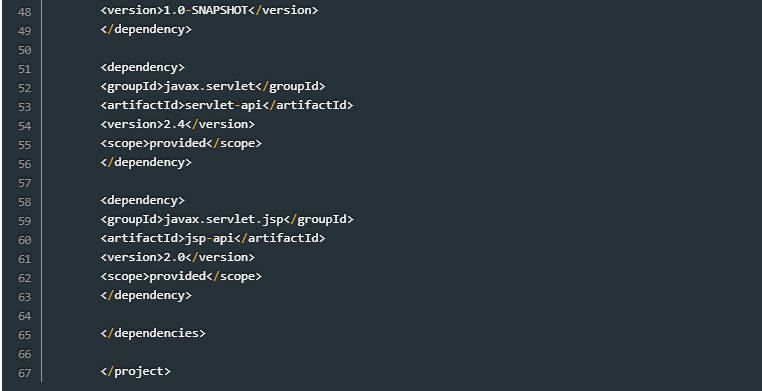
Este comando já instruiu ao Maven qual projeto queremos que seja criado, e especificamos suas coordenadas, isto é, seus dados de identificação única: groupId, artifactId e version. Antes de efetivamente criar o projeto, a ferramenta solicitará confirmação dos dados informados. Confirme pressionando <Enter>.

A seguir, no diretório onde o projeto foi criado, execute o comando mvn eclipse:eclipse para criar um projeto para o Eclipse, e em seguida o importaremos para analisarmos os arquivos criados pelo Maven.

Uma vez importado o projeto no Eclipse, podemos incluir nosso projeto prévio como dependência do novo. Dessa forma, primeiro vamos garantir que ambos os projetos sejam gerados com a mesma versão do JDK, através da configuração do plugin Maven compiler, conforme fizemos previamente. Depois, incluiremos a dependência para o projeto javaMagazine na seção dependencies, e também as dependências necessárias para o suporte a servlets, que são as classes Java que fornecem capacidades web a um projeto. O pom.xml do projeto ficará como mostrado na **Listagem 7**.

**Listagem 7.** pom.xml do projeto web.

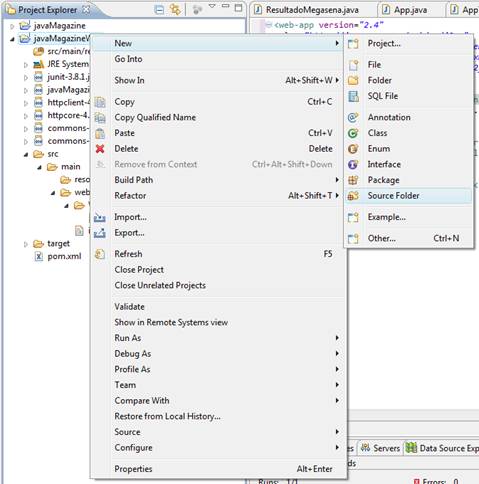
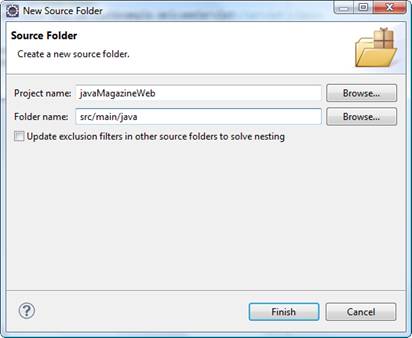




Como incluímos novas dependências, precisamos executar mvn eclipse:eclipse novamente, para que o projeto do Eclipse reflita as inclusões. Depois de executado o comando, será necessário atualizar o projeto no Eclipse (clicando com o botão direito no nome do projeto e depois na opção *Refresh*). Veremos que o Maven automaticamente também incluiu as dependências utilizadas pelo projeto anterior.

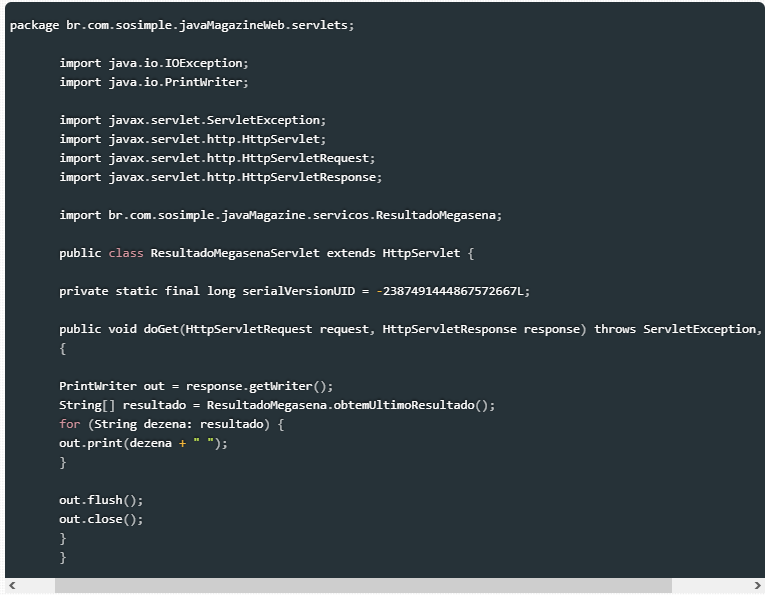
É importante notar nas dependências de servlet que elas possuem escopo provided, o que significa que o projeto precisa delas para compilar, mas que elas serão fornecidas pelo ambiente de execução, de modo que essas dependências não precisarão ser incluídas no empacotamento do WAR. O ambiente de execução, nesse caso, será um servidor web com suporte a Servlet, ***[Jetty](https://webtide.com/" \o "Jetty" \t "_blank)***. Como rodaremos o aplicativo web a partir da linha de comando do Maven, então também devemos declarar, dentro da seção build, o plugin do Jetty.

Com o intuito de especificar a localização do código-fonte para a IDE, criaremos um *Source Folder*, onde ficará nosso código Java. Deste modo, pressionando com o botão direito sobre o nome do projeto javaMagazineWeb, selecionaremos *New > Source Folder*. Em seguida, informaremos src/main/java como *Folder name*, de acordo com as **Figuras 1** e **2**.

  
**Figura 1.** Selecionando criação de source folder.  
**Figura 2.** Definindo o source folder.

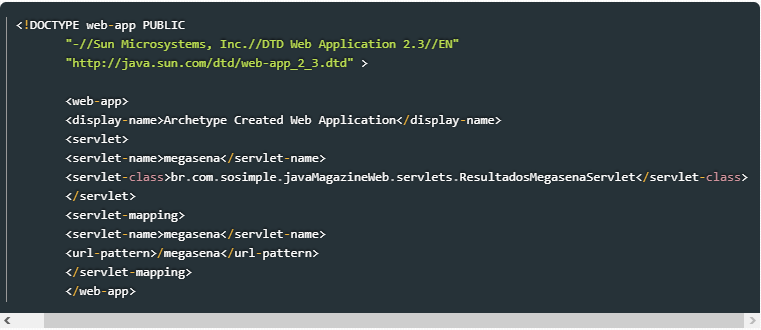
No source folder que acabamos de configurar, iremos começar o desenvolvimento propriamente dito do nosso projeto web. Como se trata apenas de um exemplo, utilizaremos apenas uma **[Servlet](https://www.devmedia.com.br/servlet-tutorial/27841" \o "O que são Servlets?" \t "_blank)**, que, quando chamada a partir de um navegador, irá invocar a classe ResultadoMegasena, do nosso projeto anterior, e obterá as dezenas para depois exibi-las na web. Para esse fim, criaremos o pacote br.com.sosimple.javaMagazineWeb.servlets e, dentro dele, a classe ResultadoMegasenaServlet, cujo conteúdo está expresso na **Listagem 8**.

**Listagem 8.** Código-fonte da classe ResultadoMegasenaServlet.

 }

Depois da classe pronta, precisaremos definir seus dados no arquivo /javaMagazineWeb/src/main/webapp/WEB-INF/web.xml, onde lhe forneceremos um nome (megasena) e depois iremos definir qual será a URL que a invocará, para que o servidor web saiba para onde direcionar requisições que apontem para a servlet. O conteúdo do arquivo web.xml, que fornece as configurações do nosso projeto web, deverá ser o mesmo da **Listagem 9**.

**Listagem 9.** Conteúdo do arquivo web.xml.

<

O próximo passo é compilar o código, empacotá-lo e o rodar no servidor web Jetty. Isso tudo será realizado com o comando mvn clean install jetty:run. Note que a linha de comando fica presa, indicando que o servidor está no ar. Enquanto não pressionarmos *Ctrl + C*, o Jetty continuará servindo requisições.

A URL para testarmos nossa servlet é http://localhost:8080/javaMagazineWeb/megasena. Se tudo ocorreu como esperado, visualizaremos na tela os números do último sorteio da Mega-Sena.

**Dicas**

A seguir, apresentamos uma compilação de dicas para o uso cotidiano do Maven:

* **Maven em modo offline:** uma vez que tenhamos todas as dependências do projeto em nosso repositório local, podemos economizar tempo explicitando que os comandos do Maven somente consultarão o repositório local. Para isso, acrescente o parâmetro –o, como no exemplo mvn install –o;
* **Pular o estágio de testes no ciclo de vida padrão:** como vimos na [**explicação dos ciclos de vida do Maven**](https://www.devmedia.com.br/gerenciando-projetos-com-maven/10823), grande parte dos estágios do ciclo de vida padrão ocorrem após o estágio de testes, implicando que os testes unitários do projeto são executados com uma frequência que pode ser desnecessária, sobretudo em projetos com grande quantidade de testes unitários. O parâmetro -Dmaven.test.skip=true indica que o estágio de testes não deve ser executado, como no exemplo mvn install -Dmaven.test.skip=true. Obviamente, essa dica deve ser empregada com moderação, para não eliminar as vantagens da existência dos [**testes unitários**](https://www.devmedia.com.br/e-ai-como-voce-testa-seus-codigos/39478);
* **Clean:** sempre que possível, devemos incluir o ciclo clean nos comandos que executamos, para termos certeza de que não teremos código espúrio no diretório target. Um exemplo disso ocorre quando removemos arquivos do diretório src: caso eles já existam no target, não serão apagados;
* **mvnrepository.com:** este site é muito útil para quem usa o Maven, porque informa as coordenadas de todas as versões das bibliotecas comumente utilizadas em projetos Java, além de fornecer o trecho de código necessário para registrar a dependência dentro da seção dependencies do pom.xml. A biblioteca que usamos no nosso exemplo, httpclient, está cadastrada no site: *http://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient*. Podemos ver que existe uma entrada para a versão que utilizamos, 4.1.3, que é a mais recente com status de release da listagem. Clicando em cima do número da versão, temos acesso a várias informações da dependência, como link para download, homepage, snippet para incluir no pom.xml, dependências utilizadas, etc.;
* **Adicionando suporte ao Maven em sua IDE preferida:** existem plugins para as IDEs mais famosas do mercado, o que permite que estas tenham as funcionalidades da ferramenta integradas a elas. Consulte a seção de **Links** para mais informações;
* **Incluindo uma dependência manualmente no repositório local:** em algumas vezes, não é possível encontrar as coordenadas de uma dependência pública. Em outras, temos dependências que foram criadas para um projeto específico e não são públicas, como ocorreu no nosso exemplo. Como a dependência não está publicada no repositório público do Maven, obviamente estará indisponível para projetos que queiram utilizá-la. Uma solução para esse problema é instalarmos o pacote da biblioteca em nosso repositório local, através do comando mvn install:install-file -Dfile=<caminho para o arquivo> -DgroupId=<id do grupo> -DartifactId=<id do artefato> -Dversion=<versao> -Dpackaging=<formato de empacotamento (jar, war, etc)> -DgeneratePom=true. Com isso, estamos instruindo o Maven a incluir a dependência, informando todas as coordenadas dela como parâmetros do comando. Como exemplo, caso precisássemos instalar o projeto que criamos, javaMagazine, no repositório, teríamos que executar o comando: mvn install:install-file -Dfile=D:\projetos\javaMagazine\target\javaMagazine-1.0-SNAPSHOT.jar -DgroupId=br.com.sosimple -DartifactId=javaMagazine -Dversion=1.0-SNAPSHOT -Dpackaging=jar -DgeneratePom=true.

**Conclusão**

Prosseguindo com as informações do artigo anterior, demonstramos outras possibilidades de utilização do Maven. Conforme vimos, a ferramenta fornece ainda mais velocidade na codificação, automatiza uma série de etapas (compilação, testes, empacotamento, distribuição, aferição de qualidade, geração de documentação, entre outras), ajuda a manter o projeto mais organizado, gerencia as dependências de forma centralizada e transparente e facilita a manutenção dos aplicativos.

Este artigo tem o propósito apenas de fornecer uma pequena visão da potencialidade do Maven. Recomendamos que este texto seja usado apenas para guiar os pequenos passos, e que os usuários desta ferramenta a estudem a fundo, para aproveitá-la cada vez mais.

Com as vantagens que oferece, fica fácil entender porque o Maven é hoje amplamente utilizado para [**gerenciar projetos**](https://www.devmedia.com.br/guia/gestao-de-projeto/34455). Do ponto de vista das equipes de implementação, o trabalho fica muito mais organizado e o desenvolvimento mais rápido e eficiente. Da perspectiva gerencial do projeto, além do ganho de produtividade e em manutenibilidade, torna-se possível executar auditorias no código produzido e gerar documentação automaticamente. Por todos esses fatores, recomendamos enfaticamente sua adoção em projetos Java EE!

O autor gostaria de agradecer a seu amigo e colega, Jorge Herrera Robert, pelo seu criterioso peer review e por suas inestimáveis sugestões.

**Links Úteis**

* [**Preparando o ambiente para programar em Java**](https://www.devmedia.com.br/curso/programar-java/2117):  
  Neste curso você aprenderá a preparar seu ambiente para programar em Java. Veremos aqui o que é necessário instalar e como proceder para desenvolver aplicações com essa linguagem.
* [**Aplicativos híbridos com Ionic**](https://www.devmedia.com.br/aplicativos-hibridos-com-ionic/39149):  
  Veja como criar aplicativos de forma rápida, flexível e produtiva para a plataforma do Ionic.
* [**Transformando layouts em código**](https://www.devmedia.com.br/transformando-layouts-em-codigo/39148):  
  Neste DevCast conheceremos algumas ferramentas que são utilizadas pelas equipes de design e programação para a criação e compartilhamento de layouts/protótipos, facilitando a comunicação entre as áreas.

**Saiba mais sobre Java ;)**

* [**Web services RESTful: CRUD 1:N com Jersey e Hibernate**](https://www.devmedia.com.br/curso/web-services-restful-java-jersey-hibernate/2113):  
  Neste curso você aprenderá a implementar uma Web API RESTful quando seu sistema possuir relacionamentos do tipo 1:N no banco de dados.
* [**JPQL: Escrevendo consultas na JPA**](https://www.devmedia.com.br/curso/curso-jpql/2108):  
  Aprender a consultar dados armazenados em bancos de dados relacionais é um importante passo na construção de sistemas.
* [**Criando meu primeiro projeto no Java**](https://www.devmedia.com.br/curso/curso-java/2105):  
  Neste curso você aprenderá a criar o seu primeiro programa com Java, e não, ele não será um simples “Hello, World!”. :) Para isso, vamos começar ensinando como instalar o Java e preparar o ambiente de desenvolvimento.

**Referências:**

* [**Site oficial do Maven.**](https://maven.apache.org/)
* [**Site oficial do Java.**](https://www.oracle.com/java/)
* [**Site com plugin do Maven para o Eclipse.**](https://maven.apache.org/plugins/maven-eclipse-plugin/usage.html)
* [**Site com plugin do Maven para o NetBeans.**](http://wiki.netbeans.org/Maven)
* [**Site com plugin do Maven para o IntelliJ IDEA.**](https://www.jetbrains.com/idea/features/ant_maven.htm)

Tecnologias:

* [Java](https://www.devmedia.com.br/java/)

* Maven

* XML