# Documentation: Setting Up Automated Build Tasks for Various Programming Languages

## 1. Setting Up Maven for Java Projects

Maven is a build automation tool used primarily for Java projects. It manages project dependencies, builds, and deployment.

### Step 1: Install Maven

Before setting up Maven, ensure it's installed on your machine:

- Download Maven from the official website.
- Follow the instructions for installation:
  - For Windows: Add the Maven `bin` directory to your `PATH`.
  - For macOS/Linux: Use a package manager like Homebrew (`brew install maven`) or download it manually.

### Step 2: Create a `pom.xml` File

The `pom.xml` (Project Object Model) file defines project dependencies, plugins, goals, and other settings.

Example basic `pom.xml`:

xml
Copy code

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-java-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Add your project dependencies here -->
    </dependencies>

    <build>
        <plugins>
            <plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

**Step 3: Create Build Commands**

In your CI/CD pipeline configuration file (e.g., Jenkins, GitLab CI), set up the command to execute Maven:

bash
Copy code

```
# Build command
mvn clean install

# Run tests
mvn test

# Package the application (e.g., for deployment)
mvn package
```

#### Step 4: Integrate with CI/CD Pipeline

- For **Jenkins**, use the Maven plugin to set up build steps.
- For **GitLab CI**, your `.gitlab-ci.yml` file might include:

yaml
Copy code

```
stages:
  - build
  - test
  - deploy

build:
  script:
```

```
    - mvn clean install
```

### 2. Setting Up npm for Node.js Projects

npm (Node Package Manager) is used for automating the build process of JavaScript/Node.js applications.

#### Step 1: Install Node.js and npm

If you haven't installed Node.js and npm, download and install them from the official Node.js website.

#### Step 2: Initialize the Node.js Project

Create a `package.json` file by running:

bash
Copy code

```
npm init -y
```

This generates a default `package.json` that includes project metadata and dependencies.

#### Step 3: Add Build and Test Scripts

In your `package.json` file, you can define custom build tasks under the `scripts` section.

Example `package.json`:

json
Copy code

```
{
```

```
  "name": "my-node-app",
  "version": "1.0.0",
  "scripts": {
    "test": "mocha",
    "build": "webpack --config webpack.config.js",
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "webpack": "^5.0.0"
  }
}
```

**Step 4: Create the Build Command**

**In the CI/CD pipeline, specify the npm build command to execute:**

**bash**
**Copy code**

```
# Install dependencies
npm install

# Run tests
npm test

# Build the project
npm run build
```

#### Step 5: Integrate with CI/CD Pipeline

For **Jenkins** or **GitLab CI**, the configuration might look like this:

yaml
Copy code

```yaml
stages:
  - install
  - build
  - test

install:
  script:
    - npm install

build:
  script:
    - npm run build

test:
  script:
    - npm test
```

### 3. Setting Up Gradle for Android Projects

Gradle is a build automation tool widely used for Android projects and Java applications.

#### Step 1: Install Gradle

Download and install Gradle from the official website.

You can also use Android Studio, which comes with Gradle pre-installed.

**Step 2: Configure `build.gradle`**

In your Android project, the `build. gradle` file configures build tasks, dependencies, and project settings.

Example `build.gradle` (Project-level):

gradle
Copy code

```
buildscript {
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath
'com.android.tools.build:gradle:4.1.2'
    }
}
```

Example `build.gradle` (App-level):

gradle
Copy code

```
apply plugin: 'com.android.application'
```

```
android {
    compileSdkVersion 30
    defaultConfig {
        applicationId "com.example.myandroidapp"
        minSdkVersion 21
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"
    }
}

dependencies {
    implementation
'com.android.support:appcompat-v7:28.0.0'
}
```

#### Step 3: Create Build and Test Tasks

**Gradle automatically generates tasks for building and testing the Android project. The most common commands are:**

**bash**
**Copy code**

```
# Build the project
./gradlew build

# Run unit tests
./gradlew test

# Build APK for release
```

```
./gradlew assembleRelease
```

#### Step 4: Integrate with CI/CD Pipeline

In your CI/CD pipeline configuration file (e.g., Jenkins or GitLab CI), use the following commands:

yaml
Copy code

```yaml
stages:
  - install
  - build
  - test

install:
  script:
    - ./gradlew clean

build:
  script:
    - ./gradlew assembleRelease

test:
  script:
    - ./gradlew test
```

### 4. Setting Up Webpack for Front-End Projects

Webpack is a popular tool for bundling JavaScript, CSS, and other assets for front-end web projects.

### Step 1: Install Webpack

First, initialize a Node.js project with npm:

bash
Copy code
```
npm init -y
```

Install Webpack and necessary loaders:

bash
Copy code
```
npm install --save-dev webpack webpack-cli
webpack-dev-server
```

### Step 2: Configure `webpack.config.js`

Create a `webpack.config.js` file that specifies how assets should be bundled.

Example `webpack.config.js`:

javascript
Copy code
```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
```

```
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        }
      }
    ]
  },
  devServer: {
    contentBase: './dist',
    hot: true
  }
};
```

#### Step 3: Create Build and Test Tasks

**In the `scripts` section of `package.json`, add the following tasks:**

**json**
**Copy code**

```
{
  "scripts": {
    "build": "webpack --config webpack.config.js",
    "dev": "webpack serve --config webpack.config.js",
    "test": "jest"
  }
}
```

**Step 4: Integrate with CI/CD Pipeline**

**In your GitLab CI or Jenkins, configure the CI/CD pipeline like this:**

**yaml**
**Copy code**

```yaml
stages:
  - install
  - build
  - test

install:
  script:
    - npm install

build:
  script:
    - npm run build

test:
  script:
    - npm test
```