To set up a GitHub Action workflow that integrates SonarQube with Maven for Java source code analysis, including code coverage and Quality Gate checks, you'll need the following:

1. **SonarQube server setup** (can be SonarQube Cloud or your own SonarQube instance).
2. **SonarQube scanner setup** for your GitHub Action.
3. **Maven build** setup for Java project, including test and code coverage generation (usually with tools like JaCoCo).
4. **SonarQube Quality Gate** check to ensure that the code meets specific quality standards (e.g., coverage percentage, issues count).

Steps Overview:

1. **SonarQube Setup**:
   - You need a SonarQube instance (either self-hosted or SonarCloud).
   - Obtain the `SONAR_TOKEN` for authentication.
2. **Maven Setup**:
   - Make sure your `pom.xml` includes the necessary plugins for code coverage and SonarQube integration.
3. **GitHub Actions Workflow**:
   - Define a workflow YAML file in your GitHub repository that includes steps for Maven build, code coverage generation, and SonarQube analysis.

Step-by-Step Guide:

1. SonarQube Setup:

Make sure you have access to a SonarQube instance, and obtain a token from the instance for authentication. This token will be used in the GitHub Actions workflow to authenticate with SonarQube.

If using SonarCloud, sign up and create a project to get the necessary token.

2. Maven Setup (pom.xml):

Ensure your `pom.xml` is configured to generate code coverage using **JaCoCo** and integrate with **SonarQube**. Here's a sample configuration:

Example `pom.xml` for Maven with JaCoCo and SonarQube:

```
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-app</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <properties>
        <sonar.projectKey>my-java-app</sonar.projectKey>
        <sonar.organization>my-organization</sonar.organization>
        <sonar.host.url>https://sonarcloud.io</sonar.host.url>  <!-- or
your self-hosted SonarQube instance -->
    </properties>

    <dependencies>
        <!-- Your dependencies here -->
    </dependencies>

    <build>
        <plugins>
            <!-- JaCoCo plugin to generate code coverage -->
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>0.8.7</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-agent</goal> <!-- Prepare agent
for code coverage -->
                        </goals>
                    </execution>
                    <execution>
                        <goals>
                            <goal>report</goal> <!-- Generate code coverage
report -->
                        </goals>
                    </execution>
                </executions>
            </plugin>

            <!-- SonarQube plugin to integrate with SonarQube -->
            <plugin>
                <groupId>org.sonarsource.scanner.maven</groupId>
                <artifactId>sonar-maven-plugin</artifactId>
                <version>3.9.0.1103</version>
            </plugin>
        </plugins>
    </build>
</project>
```

3. GitHub Actions Workflow (`.github/workflows/sonar.yml`):

The GitHub Action YAML file will define the steps for:

- Checking out the code.
- Setting up Maven.
- Running tests and generating code coverage.
- Running SonarQube analysis.
- Checking if the Quality Gate passes.

Here's a sample GitHub Actions YAML file:

Example `sonar.yml`:

```
yaml
Copy code
name: Java CI with Maven and SonarQube

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  sonar:
    name: SonarQube Analysis
    runs-on: ubuntu-latest

    steps:
    # Step 1: Checkout the code from the repository
    - name: Checkout code
      uses: actions/checkout@v2

    # Step 2: Set up JDK 11 (or whichever version is required for your
project)
    - name: Set up JDK 11
      uses: actions/setup-java@v2
      with:
        java-version: '11'
        distribution: 'adoptopenjdk'

    # Step 3: Cache Maven dependencies for faster builds
    - name: Cache Maven dependencies
      uses: actions/cache@v2
      with:
        path: ~/.m2/repository
        key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
        restore-keys: |
          ${{ runner.os }}-maven-

    # Step 4: Build the project and run tests to generate code coverage
    - name: Build with Maven
      run: mvn clean install

    # Step 5: Run SonarQube analysis
    - name: SonarQube Scan
      env:
        SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}  # Store the SonarQube
token in GitHub Secrets
      run: mvn sonar:sonar

    # Step 6: Wait for the Quality Gate to be successful
    - name: Check Quality Gate status
      uses: sonarsource/sonarcloud-github-action@v1
      with:
        token: ${{ secrets.SONAR_TOKEN }}  # Use the token from GitHub
Secrets
```

Explanation:

1. **`on`**: Defines the trigger for the workflow. This will trigger on `push` or `pull_request` events to the `main` branch.
2. **Job `sonar`**:
    - **Checkout code**: Uses `actions/checkout` to get the code from your repository.
    - **Set up JDK**: Configures the correct Java version needed for your build (Java 11 in this case).
    - **Cache Maven dependencies**: Caches the Maven dependencies to speed up the builds.
    - **Build with Maven**: Runs `mvn clean install` to build the project and run tests, generating code coverage via JaCoCo.
    - **SonarQube Scan**: Runs `mvn sonar:sonar` to start the SonarQube analysis. The `SONAR_TOKEN` is securely passed from GitHub Secrets.
    - **Quality Gate Check**: Uses the `sonarsource/sonarcloud-github-action` to check if the SonarQube Quality Gate passed or failed.

4. GitHub Secrets:

You need to store your SonarQube token as a GitHub secret to keep it secure. To do this:

1. Go to your GitHub repository.
2. Navigate to **Settings** > **Secrets** > **New repository secret**.
3. Add a new secret with the name `SONAR_TOKEN` and the value as your SonarQube token.

5. Quality Gate:

In SonarQube, you can define **Quality Gates**, which specify thresholds for various quality metrics (like code coverage, number of issues, duplications, etc.). The quality gate is checked after the analysis is completed.

- Go to your SonarQube instance (or SonarCloud).
- Define a **Quality Gate** that enforces the desired thresholds for metrics like:
    - **Coverage**: e.g., 80% or higher.
    - **New Bugs**: No new bugs allowed.
    - **Duplicated Lines**: Less than 3%.

The GitHub Action will automatically fail if the Quality Gate fails (i.e., if your code doesn't meet the quality standards).

6. Run the Workflow:

Once you commit and push the `.github/workflows/sonar.yml` file, the GitHub Action will trigger on every `push` or `pull_request` to the `main` branch. It will build the project, run tests, perform the SonarQube analysis, and check the Quality Gate.

Conclusion:

This setup integrates SonarQube with a Maven-based Java project on GitHub, running code analysis, generating code coverage, and verifying if the project meets the defined Quality Gate standards. With this GitHub Actions workflow, you can ensure continuous integration with code quality checks in place for your Java projects.