International Journal of Engineering Research-Online
A Peer Reviewed International Journal
Email:editorijoer@gmail.com http://www.ijoer.in

Vol.4., Issue.3., 2016
(May-June)

REVIEW ARTICLE

ISSN: 2321-7758

SURVEY: SONARQUBE CUSTOM PLUG-INS WRITTING

MADHURA SUBRAY HEGDE[1],VASANTHA RAGHVAN[2]

[1]M. Tech in Computer Engineering, SJCE, Mysuru, Karnataka

[2]Professor in Computer Engineering, SJCE, Mysuru, Karnataka

ABSTRACT

A Sonarqube custom plugin is a collection of Java objects that implement extension points. These extension points are abstract classes or interfaces which model a portion of the system and define what needs to be implemented. With the help of various abstract interfaces and abstract classes' metrics, sensors, decorator and widget are developed. These modules form a complete custom plugin.

Keywords- Sonarqube; Custom plugin; Sensor; Metrics; Decorates; Maven;

I. INTRODUCTION

Sonarqube is a static and dynamic code analysis tool and dashboard, that lets user not only inspect code, but also track all metrics and publish it in a dashboard, manage code quality, and much more. Custom plug-ins intensify Sonarqube functionality in different ways: it support static analysis of various languages, addition of new metrics to the tool, or changingthe way of presenting information and managing.

Sonarqube plug-in projects can be developed using Maven, with the added advantage that leverage the project folder structure and dependency maintenance mechanism. A Sonarqubecustom plug-in has three requirements, in terms of Maven configuration, this makes them different from a regular Java project:

1. Anexclusive type of packaging, sonar-plugin. This way of packaging is used to fine tune the project life-cycle without compelling to tweak plug-in configuration in the pom.xml file.
2. A reliance with Sonarqube plug-in API. Artifact coordinates are: org.codehaus.sonar:sonar-plugin-api.
3. A build plug-in will take care of the settings of packaging the plug-in for releasing it into a Sonarqube installation. Settings for the Sonarqube packaging plug-in must include the plug-in key and the plug-in main class.

The POM file for the plug-in looks as follows[1] :

```
<?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">

    <groupId>com.wordpress.deors</groupId>

<artifactId>tools.sonarqube.idemetadata</artifactId>
    <packaging>sonar-plugin</packaging>
    <version>1.0-SNAPSHOT</version>
```

.....(CONTINUE)

```
<dependencies>
    <dependency>
        <groupId>org.codehaus.sonar</groupId>
        <artifactId>sonar-plugin-api</artifactId>
        <version>3.7.3</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.sonar</groupId>
            <artifactId>sonar-packaging-maven-
plugin</artifactId>
            <version>1.7</version>
            <extensions>true</extensions>
            <configuration>
                <pluginKey>idemetadata</pluginKey>

<pluginClass>deors.tools.sonarqube.idemetadata.IDEMeta
dataPlugin</pluginClass>
                <pluginName>Sonar IDE Metadata
plugin</pluginName>
                <pluginDescription>Gathers and
displays information from IDE metadata files,
including project type (based on natures/facets) and
dependencies.</pluginDescription>
            </configuration>
        </plugin>
```

II. THE CUSTOM PLUG-IN MAIN CLASS

The entry point of every Sonarqubecustom plug-in is a class that must extend the org.sonar.api.Sonar Plugin abstract class. A custom plug-in is designed as an implementations of the interface org.sonar.api. Extension.

The Sonar qube custom plug-in main class has unique purpose of defining which extensions will be contributed by the plug-in – sensors,metrics, widgets, decorators. The custom plug-in main class looks as follows:

```
public class IDEMetadataPlugin extends SonarPlugin
    public List<Class<? extends Extension>>
getExtensions() {
        return Arrays.asList(
            IDEMetadataMetrics.class,
            IDEMetadataSensor.class,
            IDEMetadataDashboardWidget.class);
    }
}
```

Metric classes introduces custom metrics, sensor scans the code, collects the metrics and stores all of them in Sonarqube, a widget is displayed on dashboard to view the collected information by the users and decorators are used for calculating derived values for a project resource.

When designing a custom Sonarqube [2] plug-in, it's important to distinguish between Sensors and Decorators. A Sensor of custom plugin is executed only once per analysis. The project folder structure is traversed recursively by sensor to compute metric values, i.e. with the help of an external tool, and stores the information in Sonarqube database. After all sensor are executed a decorator is executed for each resource once. Decorators can query existing metric values, compute derived values and store all of them in Sonarqube database. Decorators are essentially used to compute aggregated values or to compute metrics that depends on values coming from different sources.

I. Custom Metrics and Sensors DEFINATION

The custom plug-in collects all information from IDE configuration files, i.e. all the data about the project configuration.The custom plug-inrely on an EclipseAnalyzer class, which does the hard work.

A Metric class defines the set of metrics called custom metrics that the plug-in will gather. A Sensor class will execute the EclipseAnalyzer on behalf of the analysis, and the result is storedin the Sonarqube API.

II. Defining a Metric Class

A Metric class in Sonarqube API is very simple: Createa classthat implements the interface **org.sonar.api.measures.Metrics**. This interface specifies only one method to be implemented: **List<org.sonar.api.Metric>getMetrics ()**. That is nothing but a simple method which returns a list of Metric objects.

Each Metric object which is returned will represent a Metric entity in Sonarqube model. Each Metric can have value of a certain data type like Boolean, string, integer values, floating-point values, percentages, etc., bequalitativeor quantitative, aggregated automatically and other characteristics.

The simplest method to define a Metric is to make use of the Metric builder pattern, as shown below:

```
public static final Metric IDE_IS_JAVA =
    new Metric.Builder(
        "ide_is_java", // metric identifier
        "Java project", // metric name
        Metric.ValueType.BOOL) // metric data ty
    .setDescription("Whether the project is conf
as Java in the IDE")
    .setQualitative(false)
    .setDomain(CoreMetrics.DOMAIN_GENERAL)
    .create();
```

After defining needed metrics, the metrics must be implemented as follows:

```
public List<Metric> getMetrics() {
    return Arrays.asList(
        IDE_PRJ_NAME, IDE_IS_JAVA, IDE_IS_EAR,
IDE_IS_EJB, IDE_IS_WEB,
        IDE_IS_GWT, IDE_IS_GAE, IDE_IS_GROOVY,
IDE_IS_GRAILS,
        IDE_IS_PDE, IDE_IS_JET, IDE_DEPENDENCI
IDE_CLASSPATH);
}
```

III. Defining a Sensor Class

After defining the metric class a Sensor class must be defined. Sensor class in Sonarqube API is very simple to define:

Createa class which implementing the interface **org.sonar.api.batch.Sensor** and then implement the two methods.

The first one is **booleanshouldExecuteOnProject(org.sonar.api.resources.Project)**, this tells Sonarqubeanalyser whether it should execute on any specific project or not. The Project object provides all the information needed to take decision, like the project name, analysis type,language, the list of modules it contains, etc.

The second one is **voidanalyse(org.sonar.api.Resources.Project,org.sonar.api.batch.SensorContext)**. This is the method inside whichmetric values are computed,the project resources will be scanned, and saved in Sonarqube database.

To access the project directory structure, the appropriate way is to leverage the dependency is to injection mechanism of the Plexus container where SonarQube analysis runs. The constructor is defined as follows and Plexus will perform other tasks:

```
public
IDEMetadataSensor(org.sonar.api.scan.filesystem.Module
FileSystem fileSystem) {
    this.fileSystem = fileSystem;
}
```

Independent of the method used to compute metric values, once they are available, they will be store using the **saveMeasure on org.sonar.api.measures. Measure** method of interface **org.sonar.api.batch.Sensor Context**.

A Measure is created using a simple constructor with metric id and value. The sensor context basically will determine the resource to which the metric value belongs to as shown below:

```
Measure measure = new
Measure(IDEMetadataMetrics.IDE_PRJ_NAME,
projectInfo.getProjectName());
sensorContext.saveMeasure(measure);
measure = new Measure(IDEMetadataMetrics.IDE_IS_JAVA,
projectInfo.isJavaProject() ? 1d : 0d); // for boolean
values, 1 is true, 0 is false
sensorContext.saveMeasure(measure);
```

IV. Creating Widgets

The aim of the widget to be created is to simply showcase the collected and computed project-level information in project's dashboard. For example which type of project user has configured in the IDE, the dependencies that exist with other projects, etc.

Defining a widget class is very simple [4]:

- Create a class extending **org.sonar.api.web. AbstractRubyTemplate** and implementing **org.sonar.api.web.RubyRails Widget**.
- Annotate the class with **@org.sonar.api.web.UserRole**. Add as arguments the users that will have access to this widget: Admin, User,Code Viewer.
- User can also annotate the class with **@org.sonar.api.web.Description**.

Parameters must be added as the description for the widget, as it will be displayed in Sonarqube dashboard configuration view. This annotation also tells the other users what the widget meant to do.

- Implement **getId()** and **getTitle()** methods. These methods simply returns a string with the widget id and title. The id must be unique and the title will be showed later in Sonarqube dashboard configuration view.
- Implement **getTemplatePath()** method. This function will return a path where the Ruby page will be residing and can be found. The path must be relative to the custom plug-in project classpathand also it must start with a forward slash.

V. SonarQube Development Mode

Sonarqube provides with a development mode to prevent developer's sanity. This mode of development will automate the deployment process as a Maven plug-in, downloading a Sonarqube [3] instance from the internet and then running it as a child process of the Maven process. To launch Sonarqube'sdevelopment mode the following command is used:

Mvn install org.codehaus.sonar:sonar-dev-maven-plugin:"version":start-war -Dsonar.runtime

VI. Conclusion

Sonarqube custom plugin can be developed by using the various abstract classes and interfaces. Also the main modules that are necessary for the custom plugin are custom metrics, sensors, decorators and widget, which are simple java objects.

REFERENCE

[1] "Writing your own sonar plug-in", Dr. Macphail's Trance

[2] G. Ann Campbell and Patroklos P. Papapetrou, "Sonarqube in action"

[3] Rudolf Ferenc, Laszl ´ o Lang ´ o, Istv ´ an Siket, Tibor Gyim ´ othy, "SourceMeter SonarQube plug-in", IEEE International Working Conference on Source Code Analysis and Manipulation

[4] A page from Sonarqube documentation "http://www.sonarqube.org/tag/plugins/"