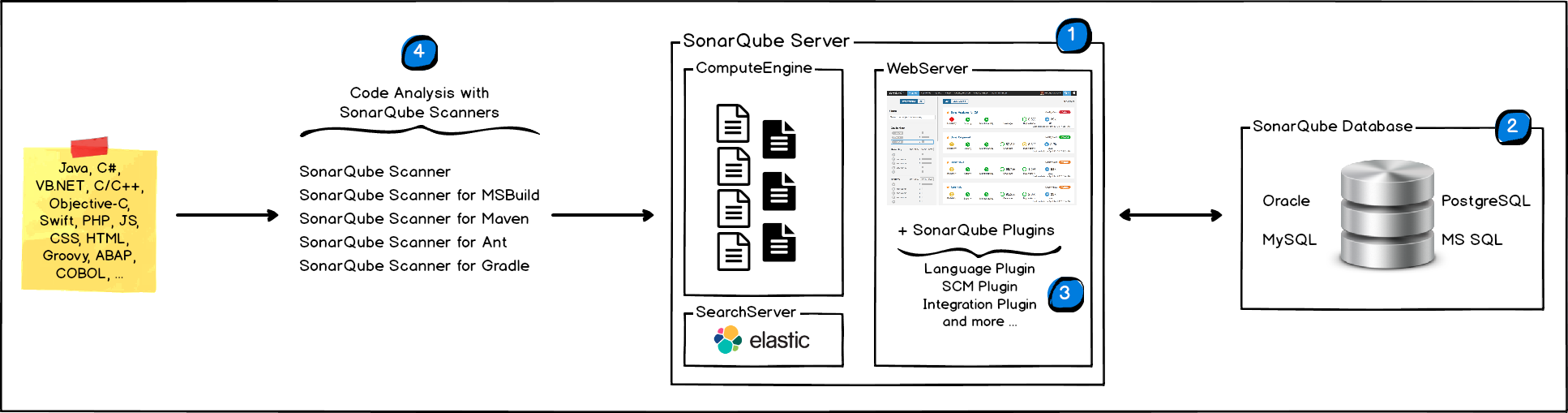
**Architecture**

The SonarQube Platform is made of 4 components:

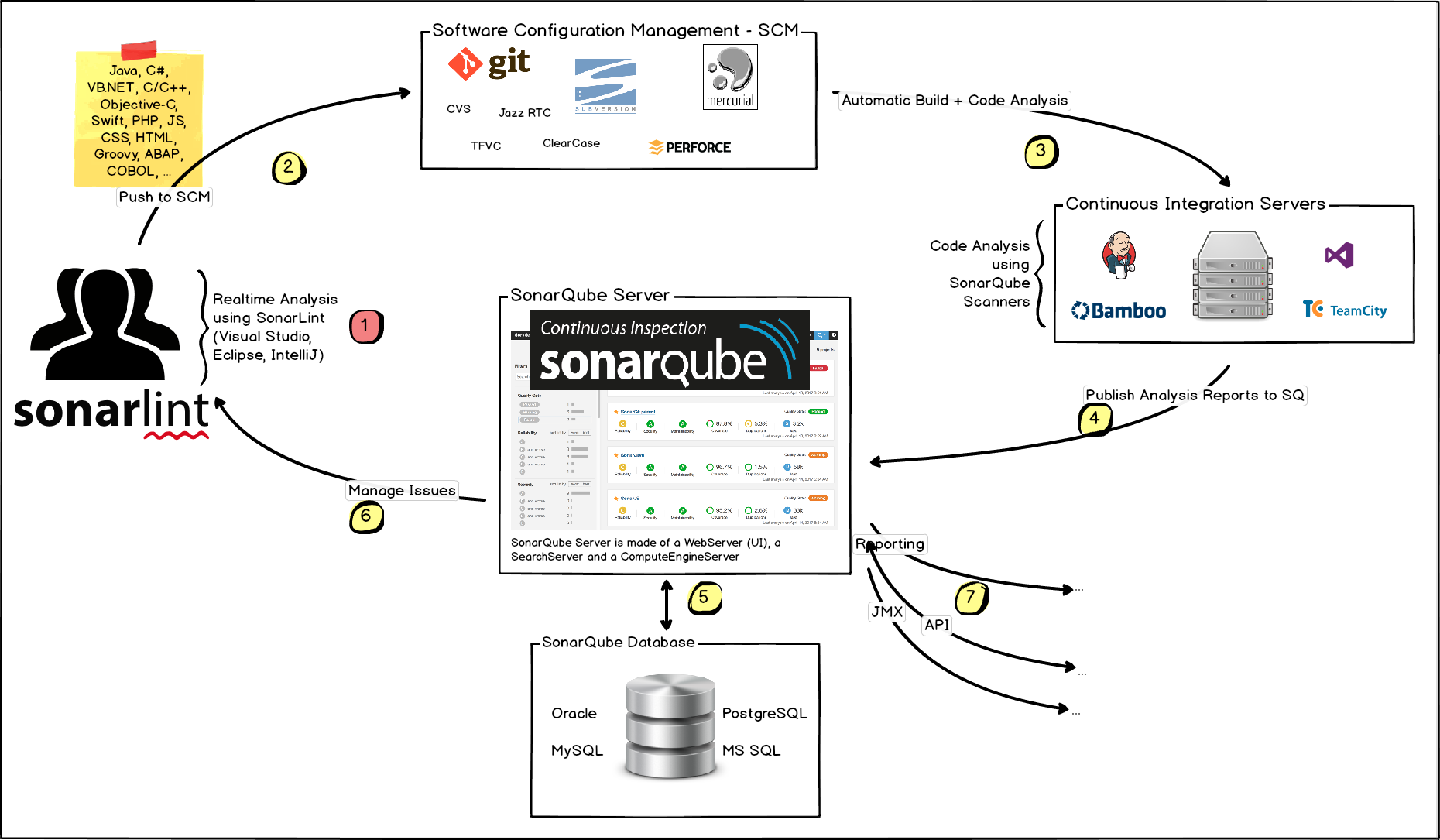
1. One **SonarQube Server** starting 3 main processes:
   1. a **Web Server** for developers, managers to browse quality snapshots and configure the SonarQube instance
   2. a **Search Server**based on Elasticsearch to back searches from the UI
   3. a **Compute Engine Server** in charge of processing code analysis reports and saving them in the SonarQube Database
2. One **SonarQube Database**to store:
   1. the configuration of the SonarQube instance (security, plugins settings, etc.)
   2. the quality snapshots of projects, views, etc.
3. Multiple **SonarQube Plugins** installed on the server, possibly including language, SCM, integration, authentication, and governance plugins
4. One or more**SonarQube Scanners** running on your Build / Continuous Integration Servers to analyze projects



# Integration

The following schema shows how SonarQube integrates with other ALM tools and where the various components of SonarQube are used.

1. Developers code in their IDEs and use SonarLint to run local analysis.
2. Developers push their code into their favorite SCM: git, SVN, TFVC, ...
3. The Continuous Integration Server triggers an automatic build, and the execution of the SonarQube Scanner required to run the SonarQube analysis.
4. The analysis report is sent to the SonarQube Server for processing.
5. SonarQube Server processes and stores the analysis report results in the SonarQube Database, and displays the results in the UI.
6. Developers review, comment, challenge their Issues to manage and reduce their Technical Debt through the SonarQube UI.
7. Managers receive Reports from the analysis.  
   Ops use APIs to automate configuration and extract data from SonarQube.  
   Ops use JMX to monitor SonarQube Server.



## About Machines and Locations

* The SonarQube Platform cannot have more than one SonarQube Server and one SonarQube Database.
* For optimal performance, each component (server, database, scanners) should be installed on a separate machine, and the server machine should be dedicated.
* SonarQube Scanners scale by adding machines.
* All machines must be time synchronized.
* The SonarQube Server and the SonarQube Database must be located in the same network
* SonarQube Scanners don't need to be on the same network as the SonarQube Server.
* There is **no communication** between **SonarQube Scanners** and the **SonarQube Database**.

**Requirements and Installation:**

**Analysis of Code:**

Once the SonarQube platform has been installed, you're ready to install an analyzer and begin creating projects. A project is created in the platform automatically on its first analysis. However, if you need to set some configuration on your project before its first analysis, you have the option of provisioning it.

**Scope of Analysis: Types of Files and Data**

SonarQube can perform analysis on 20+ different languages. The outcome of this analysis will be quality measures and issues (instances where coding rules were broken). However, what gets analyzed will vary depending on the language:

* On all languages, "blame" data will automatically be imported from supported SCM providers. Git and SVN are supported automatically. Other providers require additional plugins.
* On all languages, a static analysis of source code is performed (Java files, COBOL programs, etc.)
* A static analysis of compiled code can be performed for certain languages (.class files in Java, .dll files in C#, etc.)
* A dynamic analysis of code can be performed on certain languages.

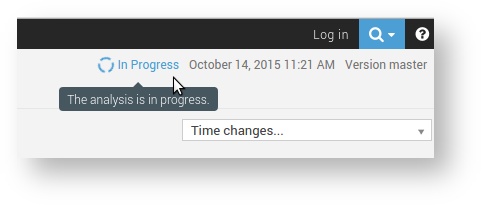
**Unrecognized files**

By default, only files that are recognized by a language plugin are loaded into the project during analysis. For example, if your SonarQube instance has the Java and JavaScript plugins on board, all .java and .js files will be loaded, but .xml files will be ignored. However, it is possible to import all text files in the analysis encoding in a project by setting Settings > Exclusions > Files > Import unknown files to true.

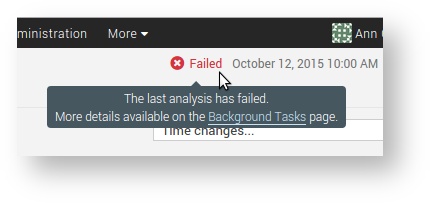
**During Analysis**

During analysis, data is requested from the server, the files provided to the analysis are analyzed, and the resulting data is sent back to the server at the end in the form of a report, which is then analyzed asynchronously server-side.

Analysis reports are queued, and processed sequentially, so it is quite possible that for a brief period after your analysis log shows completion, the updated values are not visible in your SonarQube project. However, you will be able to tell what's going on because an icon will be added next to the project name. Mouse over it for more detail (and links if you're logged in with the proper permissions.)



The icon goes away once processing is complete, but if analysis report processing fails for some reason, the icon will change:



**Running Analysis**

First, you should install the plugin(s) for the language(s) of the project to be analyzed, either by a direct download or through the update center.

Then, you need to choose an analysis method. The following are available:

* SonarQube Scanner for MSBuild: Launch analysis of .Net projects
* SonarQube Scanner for Maven: Launch analysis from Maven with minimal configuration
* SonarQube Scanner for Gradle: Launch Gradle analysis
* SonarQube Scanner for Ant: Launch analysis from Ant
* SonarQube Scanner for Jenkins: Launch analysis from Jenkins
* SonarQube Scanner: Launch analysis from the command line when none of the other analyzers is appropriate

Note that we do not recommend running an antivirus scanner on the machine where a SonarQube analysis runs, it could result in unpredictable behavior.

**Project Page:**

Your first logged-in view of the SonarQube platform will be the Projects space, which defaults to showing your favorite projects.

From there, you may want to head to the Project Page of your current project, where you'll see all the primary indicators of quality gathered together. First, and most important, is the Quality Gate status. Does the project pass or not? From there, you can decide what needs to be worked on.

Or you could head to the Issues space, to see your personal leak, i.e. the Issues you've introduced recently. Some of them can be hard to find ahead of time, but many can be headed off by doing a Local and Branch Analysis.

What are issues: They mark spots in the Code where violations of Rules were found.

**Quality Gates:**

A quality gate is the best way to enforce a quality policy in your organization. It's there to answer ONE question: can I deliver my project to production today or not?

In order to answer this question, you define a set of Boolean conditions based on measure thresholds against which projects are measured. For example:

* No new blocker issues
* Code coverage on new code greater than 80%
* Etc.

Ideally, all projects will be verified against the same quality gate, but that's not always practical. For instance, you may find that:

* Technological implementation differs from one application to another (you might not require the same code coverage on new code for Web or Java applications).
* You want to ensure stronger requirements on some of your applications (internal frameworks for example).
* Etc.

Which is why you can define as many quality gates as you wish. Quality Gates are defined and managed in the Quality Gates page found in the top menu.

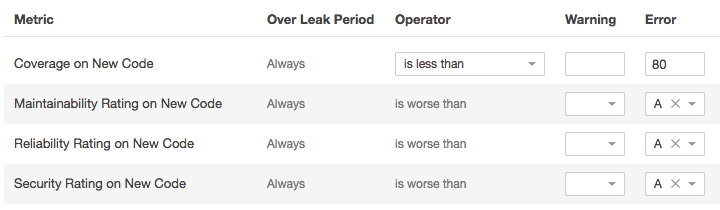
**Use the Best Quality Gate Configuration**

The quality gate "SonarQube way" is provided by SonarSource and activated by default. It represents our view of the best way to implement the Fixing the Water Leak concept. At each SonarQube release, we adjust this default quality gate according to SonarQube's capabilities.

With SonarQube 6.2 comes three new metrics allowing you to enforce a given Rating of Reliability, Security and Maintainability, not just overall but also on new code. These new metrics are now recommended and come as part of the default quality gate. We strongly advise you to adjust your own quality gates to use them to make feedback more clear to your developers looking at their quality gate on their project page.

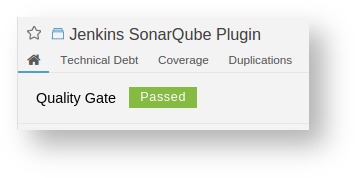
Don't forget also that quality gate conditions must use differential values. There is no point for example to check an absolute value such as: Number of Lines of Code is greater than 1000.

**Recommended Quality Gate:**



# Quality Gate Status

The current status is displayed prominently at the top of the Project page :



**Getting Notified When a Quality Gate Fails**

Users can be notified when a quality gate fails. To do so, subscribe to the New quality gate status notification either for all projects or a set of projects you're interested in. Security

**Security**

Quality Gates can be accessed by any user (even anonymous users). All users can view every aspect of a quality gate.

To make changes (create, edit or delete) users must be granted the **Administer Quality Profiles** **and Gates permission.**

A project administrator can choose which quality gates his/her project is associated with.

**Defining Quality Gates**

To manage quality gates, go to Quality Gates (top menu bar).

Each Quality Gate condition is a combination of:

* measure
* period: Value (to date) or Leak (differential value over the Leak period)
* comparison operator
* warning value (optional)
* error value (optional)

**User Account**

**Home page:**

* your groups
* your SCM accounts

**Security:**

In addition to being able to change your password, if your instance is not using a 3rd party authentication mechanism such as LDAP or any OAuth provider (GitHub, Google Account), you can manage your own authentication tokens.

You can create as many Token as you want. Once a Token is created, you can use it to publish analysis to a project where you have the execute analysis permission.

**Notifications**

The notifications feature allows any logged in user to subscribe to the following email notifications:

* Changes in issues assigned to me or reported by me
* Issues resolved as false positive or won't fix
* My New Issues
* New issues
* New quality gate status

... globally or for specific projects.

To subscribe to notifications, tick the events you want to subscribe to.

# Local and Branch Analysis

There are several workflows where you want to analyze a project without updating the canonical SonarQube project:

* **You want to check for new issues before committing your code** - SonarLint is the answer for this. It raises issues in your IDE as you type.
* **You want to check for new issues before a Pull Request is merged** - Pull request (PR) analysis allows you to configure your CI engine to perform "preview" analysis (i.e. analysis without updating SonarQube) on PR's. Any new issues are annotated in the PR itself. If none are found, an "all clear" message is added instead. PR analysis is available for GitHub projects with the GitHub plugin, and on some other providers with Other Plugins.
* **You want to analyze a long-lived branch in SonarQube** -  In this case, you want to maintain analysis of the canonical project in SonarQube, say the "master" branch, and also publish in SonarQube ongoing analysis of a secondary branch. The typical case is a release branch. In this scenario, you add the sonar.branch=[branch key] analysis property to the release branch to create a second, independent project in SonarQube.

# Code Viewer

# The code viewer is the heart of SonarQube: it displays the source code of a file (both source and test files), and its high-level statistics:

* **Lines**
* **Issues** (generated by the rules activated on the quality profile)
* **Test coverage** by unit or integration tests
* **Duplications** within the same file or in other files
* **SCM information** like who last committed a specific line and when
* All **tests** of a given test file, along with their execution time and their status

You will land on the code viewer:

* when drilling down from the Measures and Code pages.
* when reviewing issues on the Issues page.
* when searching for a particular file using the search input at the top-right.

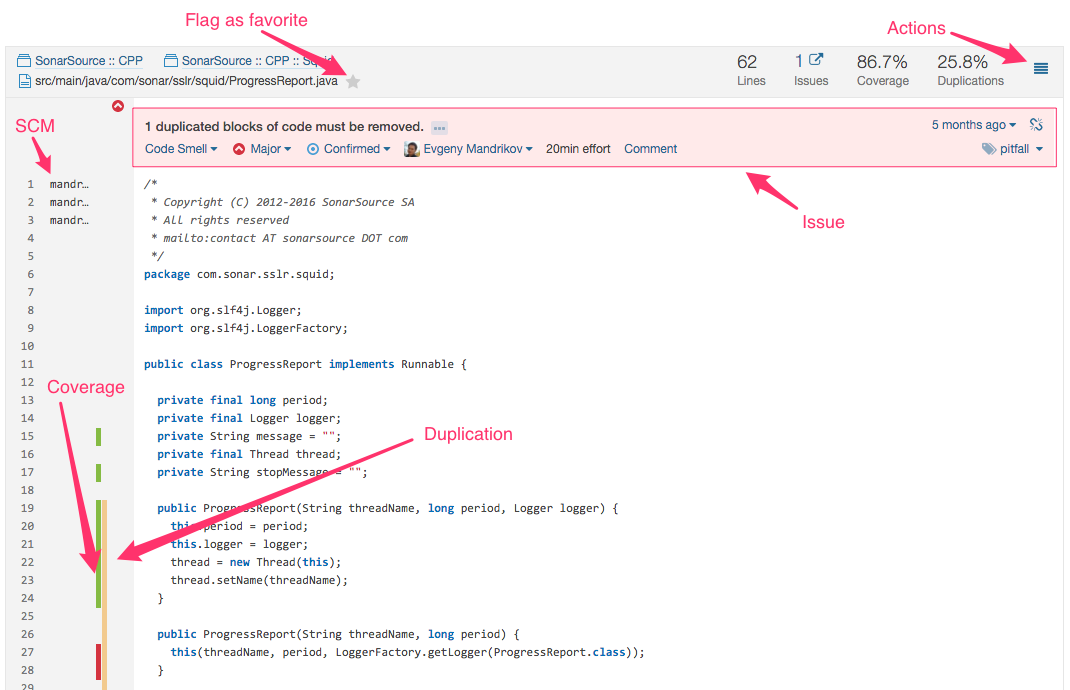
The code viewer has two aspects, the current file and pinned files.

# Current File

## **Layout**

The code viewer is composed of 2 parts:

* The **header** lies across the top of the file. It displays useful information and offers decoration and filtering actions.
* The **source code** is in the center, decorated with additional information based on the options chosen in the header.

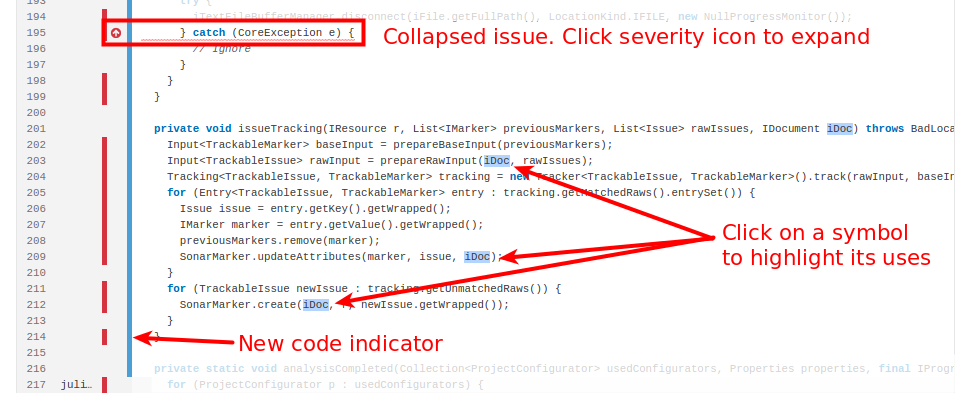


## **Header**

The header can contain up to four data blocks, one per main axis: Lines, Issues, Coverage (for source files) or Tests (for test files), and Duplications. Data blocks which aren't relevant to the current file won't be shown. For instance, if the project has no tests, the coverage number will be omitted. Similarly, the duplications block will be omitted if there are no duplications.

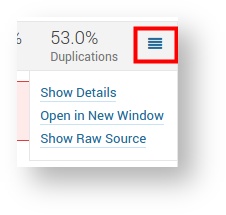
## **The source-code**

The main purpose of the code viewer is to show source code and its effort to fix it. For that reason, issue, duplication, and test decorations are always visible, but issues may be collapsed if you don't come to the code viewer from the Issues page, and the marker for new code only appears when a differential has been applied:

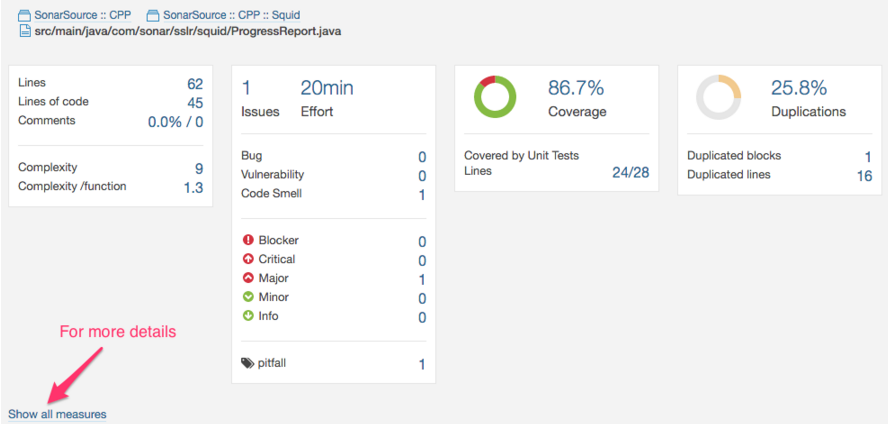


## **Additional actions**

The actions menu at the top-right of the code viewer header offers additional options.

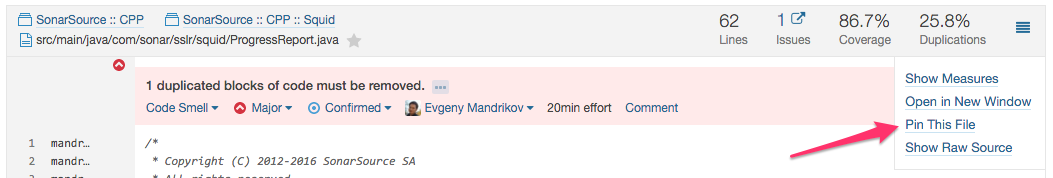


The most notable of these is "Show Details", which opens a model popup with additional data on the file:

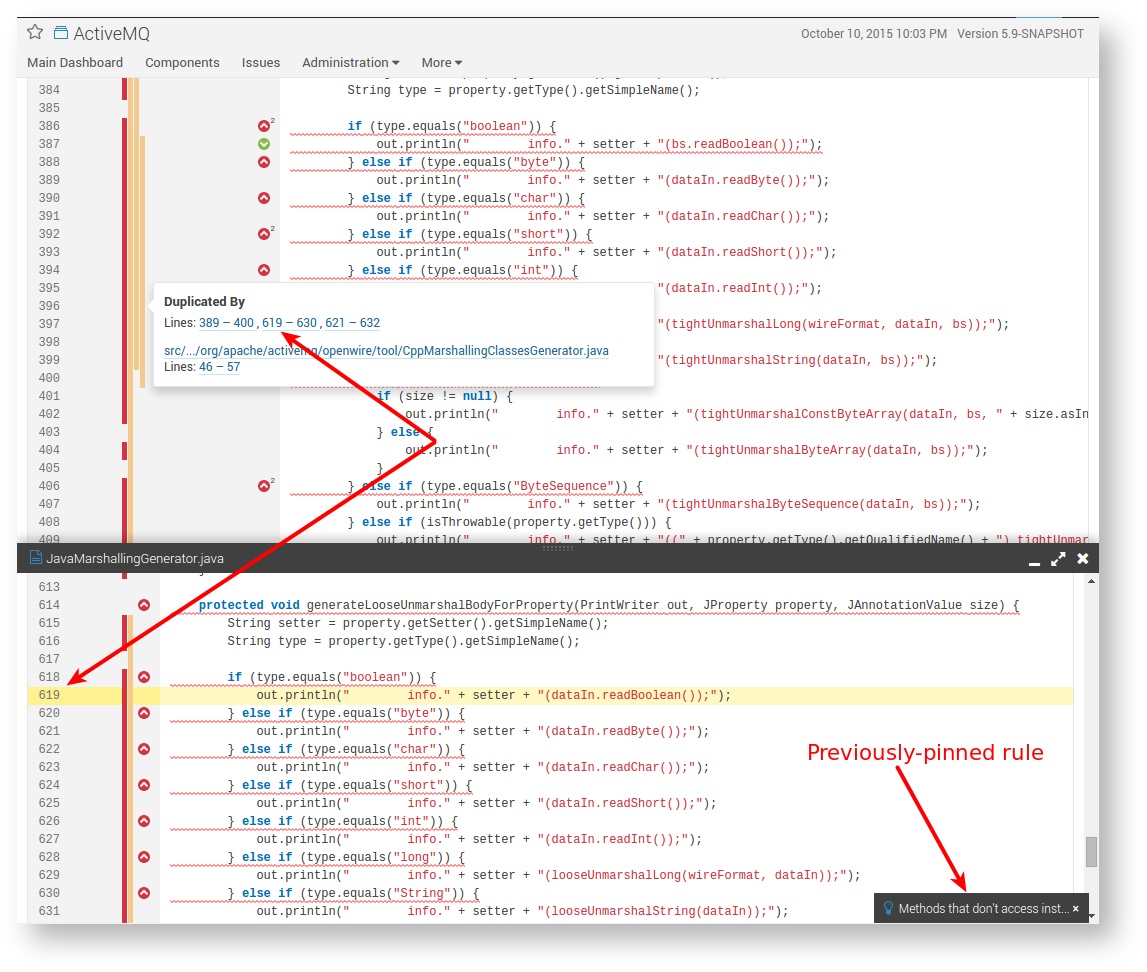


# Pinned Files

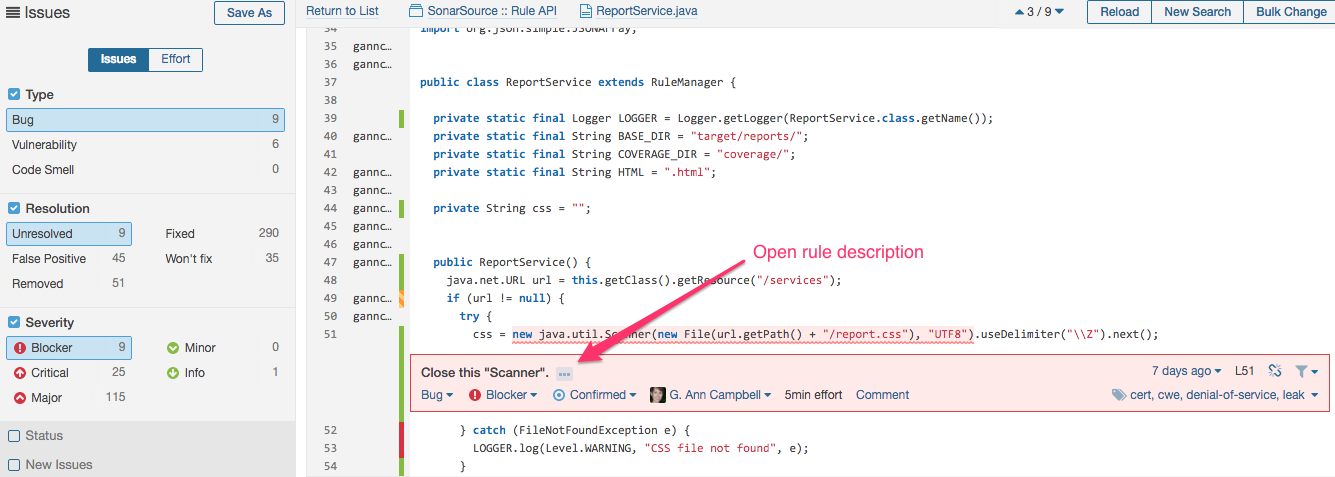
Both source files and rule descriptions can be pinned directly:



or from another file's duplication detail:



Similarly, test files can be pinned from coverage detail, and rule descriptions can be pinned from issue messages:

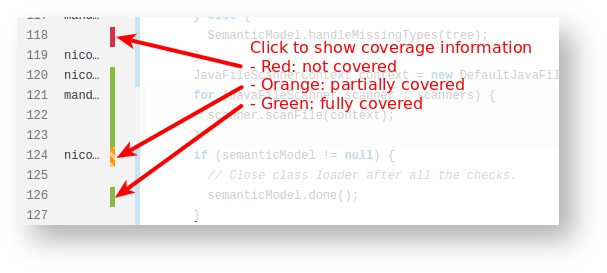


You can adjust the height of the currently-active pinned window, minimize it, maximize it, and close it. The currently-active window will automatically be minimized if you change contexts or pin another file.

**Seeing Coverage**

If coverage information is available on your project, then the component viewer can help you visualize where combined test coverage is missing.

Coverage information is conveyed via color:

See

## **Coverage per tests**

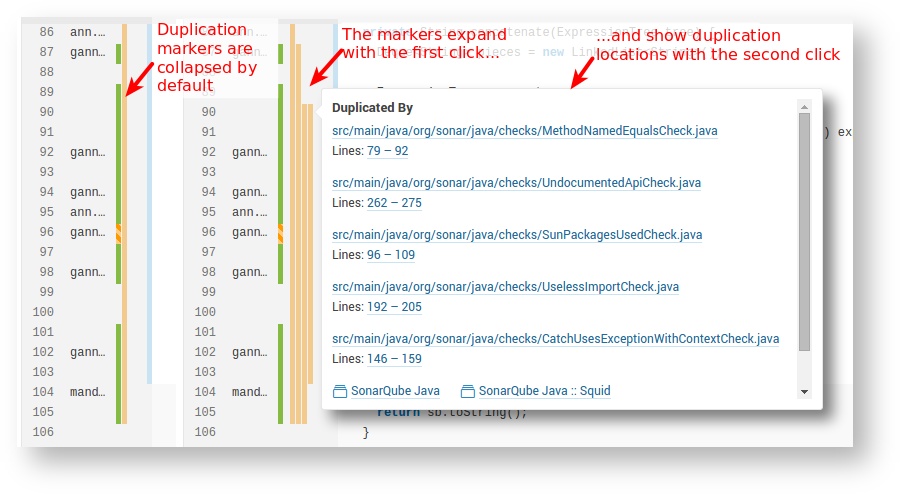
For Java projects using JaCoCo, it's possible to have "coverage per test" information. This includes:

* Displaying the number of different tests that cover a specific line
* Listing those tests
* Being able to navigate to the test files that define those tests

These things are shown in a popup when you click on the coverage indicator. Clicking on a link in the popup window opens the test file in a pinned window.

**Seeing Duplicates**

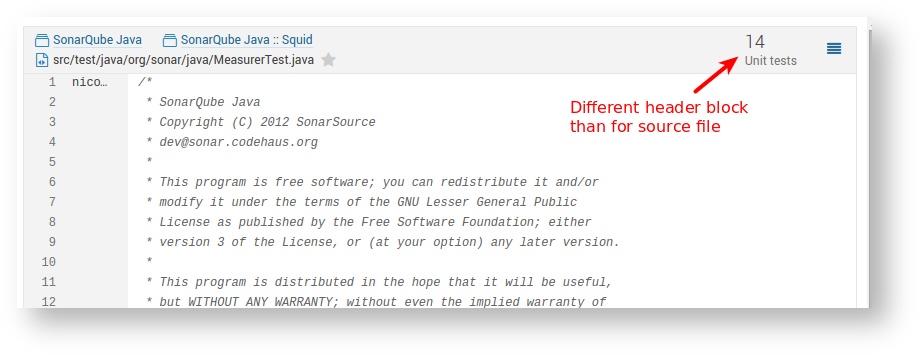
The duplications decoration shows which parts of the source code are duplicated. By default, the duplications markers are collapsed:



Once they're expanded, you can click the orange duplication indicator to see where a block of code is duplicated. Clicking on a link in the popup opens the file in a pinned window.

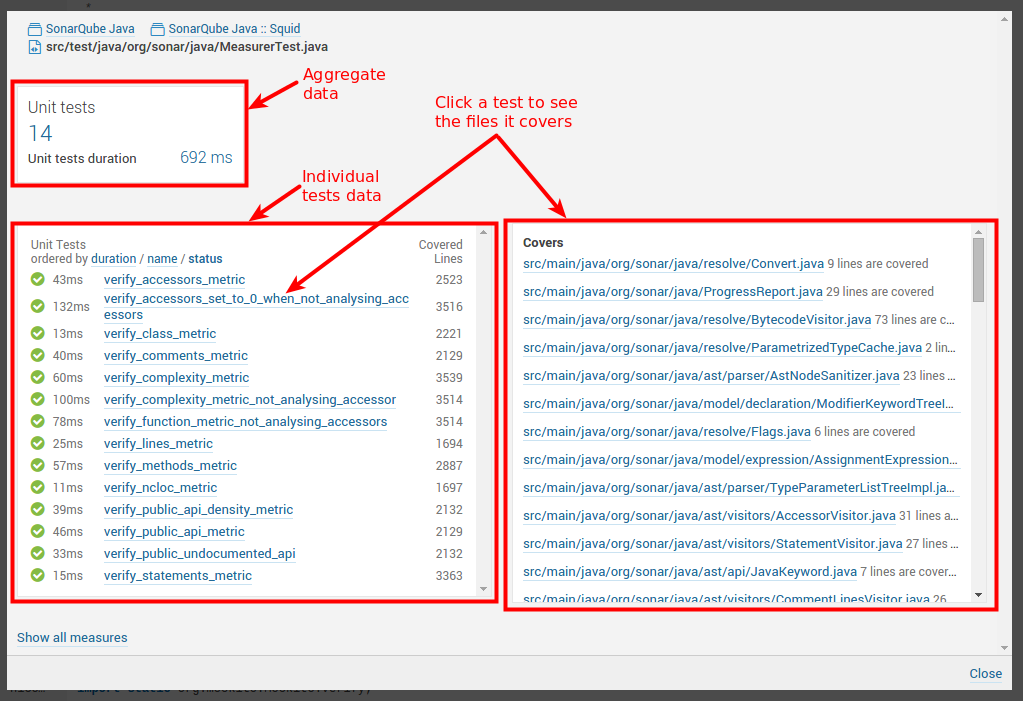
**Seeing Tests**

On test files, the Component viewer shows different data than for source files:



## **Coverage per tests**

Similarly, the **Show details** option offers If "coverage per test" information is available, the test tab can show additional information and allow you to navigate from the test file to the source files that it covers:



The objective of this tab is to answer the following questions:

* Which files are covered by a given unit test?
* How many lines of code are covered by a given test?

**Issues**

code breaks a coding rule. The set of coding rules is defined through the quality profile associated with the project.

Each issue has one of five severities:

1. **BLOCKER**  
   Bug with a high probability to impact the behavior of the application in production: memory leak, unclosed JDBC connection, .... The code MUST be immediately fixed.
2. **CRITICAL**  
   Either a bug with a low probability to impact the behavior of the application in production or an issue which represents a security flaw: empty catch block, SQL injection, ... The code MUST be immediately reviewed.
3. **MAJOR**  
   Quality flaw which can highly impact the developer productivity: uncovered piece of code, duplicated blocks, unused parameters, ...
4. **MINOR**  
   Quality flaw which can slightly impact the developer productivity: lines should not be too long, "switch" statements should have at least 3 cases, ...
5. **INFO**Neither a bug nor a quality flaw, just a finding.

Ideally, the team wouldn't introduce any new issues (any new technical debt). SonarLint for Eclipse, SonarLint for IntelliJ, and SonarLint for Visual-Studio can help developers because they provide the ability to perform local analyses to check their code before pushing it back to the SCM. But in real life, it's not always possible to code without any new technical debt, and sometimes it's not worth it. So new issues get introduced.

# Issue edits

SonarQube's issues workflow can help you manage your issues. There are seven different things you can do to an issue (other than fixing it in the code!): Comment, Assign, Confirm, Change Severity, Resolve, Won't Fix, and False Positive.

These actions break out into three different categories. First up is the "technical review" category.

## **Technical Review**

Confirm, False Positive, Won't Fix, Change Severity, and Resolve fall into this category, which presumes an initial review of an issue to verify its validity. Assume it's time to review the technical debt added in the last review period - whether that's a day, a week, or an entire sprint. You go through each new issue and do one:

* **Confirm** - By confirming an issue, you're basically saying "Yep, that's a problem." Doing so moves it out of "Open" status to "Confirmed".
* **False Positive** - Looking at the issue in context, you realize that for whatever reason, this issue isn't actually an ~~issue~~, erm... "problem." It's not actually a problem. So you mark it False Positive and move on. Requires Administer Issues permission on the project.
* **Won't Fix** - Looking at the issue in context, you realize that while it's a valid issue it's not one that actually needs fixing. In other words, it represents accepted technical debt. So you mark it Won't Fix and move on. Requires Administer Issues permission on the project.
* **Change Severity** - This is the middle ground between the first two options. Yes, it's a problem, but it's not as bad a problem as the rule's default severity makes it out to be. Or perhaps it's actually far worse. Either way, you adjust the severity of the issue to bring it in line with what you feel it deserves. The marker in the drilldown will change to show the new severity immediately, but the change won't be reflected in your issue counts until after the next analysis. Requires Administer Issues permission on the project.
* **Resolve** - If you think you've fixed an open issue, you can Resolve it. If you're right, the next analysis will move it to closed status. If you're wrong, its status will go to re-opened.

If you tend to mark a lot of issues false positive or won't fix, it means that some coding rules are not appropriate for your context. So, you can either completely deactivate them in the quality profile or use issue exclusions to narrow the focus of the rules so they are not used on specific parts (or types of object) of your application. Similarly, making a lot of severity changes should prompt you to consider updating the rule severities in your profiles.

## **Dispositioning**

Once issues have been through technical review, it's time to decide who's going to deal them. By default, they're assigned to the last committer on the issue line (at the time the issue is raised), but you can certainly reassign them to yourself or someone else. The assignee will receive email notification of the assignment if he signed up for notifications, and the assignment will show up everywhere the issue is displayed, including in the My Issues list in the My Account space.

## **General**

At any time during the lifecycle of an issue, you can log a comment on it. Comments are displayed in the issue detail in a running log. You have the ability to edit or delete the comments you made.

## **Bulk Change**

All of these changes and more can be made to multiple issues at once using the Bulk Change option in the issues search results pane.

# Other

Issues come from rules and rules are collected in profiles. Only certain users can edit profiles, but every user can view them.

**Issue Life Cycle**

Issues are detected automatically.

# Statuses

After creation, issues flow through a lifecycle, taking on one of five possible statuses:

* Open - set by SonarQube on new issues
* Confirmed - set manually to indicate that the issue is valid
* Resolved - set manually to indicate that the next analysis should Close the issue
* Reopened - set automatically by SonarQube when a Resolved issue hasn't actually been corrected
* Closed - set automatically by SonarQube for automatically created issues.

# Resolutions

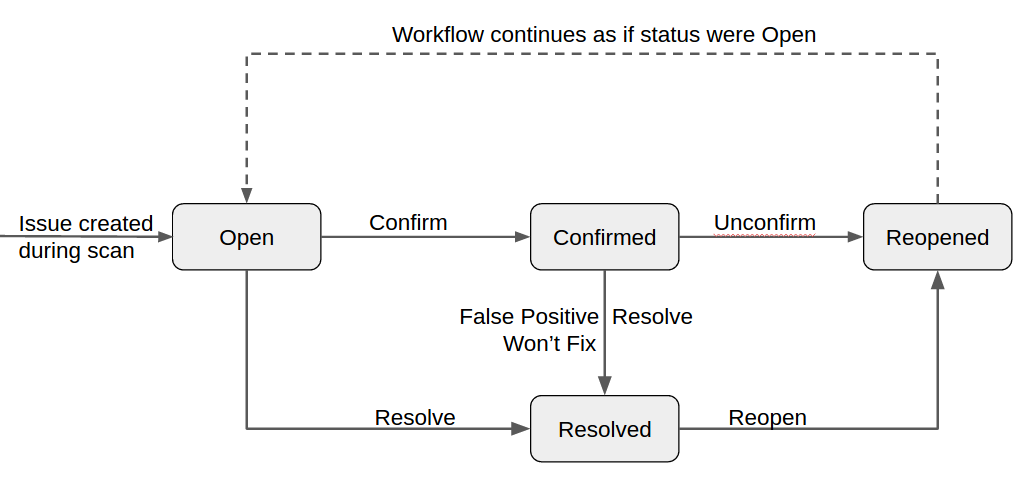
Closed issues will have one of two resolutions:

* Fixed - set automatically when a subsequent analysis shows that the issue has been corrected
* Removed - set automatically when either the related coding rule or the file is no longer available. The rule may not be available either because it has been removed from the profile or because the underlying plugin has been uninstalled. The file could be unavailable because it has been removed from the project, moved to a different location or renamed.

Resolved issues will have one of two resolutions:

* False Positive - set manually
* Won't Fix - set manually

# Issue workflow



Issues are automatically closed (status: Closed) when:

* an issue (of any status) has been properly fixed => Resolution: Fixed
* an issue no longer exists because the related coding rule has been deactived or is no longer available (ie: plugin has been removed) => Resolution: Removed

Issues are automatically reopened (status: Reopened) when:

* an issue that was Resolved (but Resolution is not False positive) is shown by a subsequent analysis to still exist

# Understanding which issues are "new"

To determine the creation date of an issue, during each analysis, the following algorithm is executed to determine if an issue is new or existed previously. For each issue, three of four criteria (rule; line number; line content or "hash"; and message) must match for a detected issue to be matched up with an existing one:

* on the same rule, with the same line number and with the same hash (but not necessarily with the same message) > MATCH
* on the same rule, with the same message and with the same hash (but not necessarily with the same line) > MATCH
* on the same rule, with the same message and with the same line number (but not necessarily with the same hash) > MATCH

In any other case, the issue is a new one.

# Purging Closed Issues

By default, Closed issues are kept for 30 days. For more details, browse the Housekeeping documentation.

**Issue Tags:**

**Automatic Assign Issues:**

**Rules**

In the SonarQube platform, plugins contribute rules which are executed on source code to generate issues. Those issues are used to compute remediation cost and technical debt. There are three basic types of rules: Reliability and Maintainability rules, from which zero false positives are expected, and Security rules, which may produce some false positives. The Rules page is the entry point where you can discover all the existing rules or create new ones based on provided templates.

# Finding Rules

Click on the top "Rules" menu item to enter the world of rules. By default, you will see all the available rules, with the ability to narrow the selection based on search criteria in the left pane:

* **Language**: the language to which a rule applies.
* **Type**: Bug, Vulnerability or Code Smell rules
* **Tag**: it is possible to add tags to rules in order to classify them and to help discover them more easily.
* **Repository**: the engine that contributes rules to SonarQube.
* **Default Severity**: the original severity of the rule - as defined by the plugin that contributes this rule.
* **Status**: rules can have 3 different statuses:
  + **Beta:** The rule has been recently implemented and we haven't gotten enough feedback from users yet, so there may be false positives or false negatives.
  + **Deprecated:** The rule should no longer be used because a similar, but more powerful and accurate rule exists.
  + **Ready:** The rule is ready to be used in production.
* **Available Since**: date when a rule was first added on the SonarQube instance. This is useful to list all the new rules since the last upgrade of a plugin for instance.
* **Template**: display rule templates that allow to create custom rules (see later on this page).
* **Quality Profile:** inclusion in or exclusion from a specific profile

If a quality profile is selected, it is also possible to check for its active severity and whether it is inherited or not. See the Quality Profile documentation for more.

# Rule Details

To see the details of a rule, either click on it, or use the right arrow key. Along with basic rule data, you'll also be able to see which, if any, profiles it's active in and how many open issues have been raised with it.

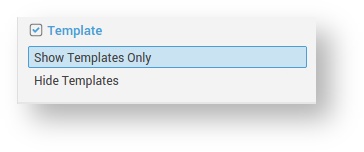
The 2 following actions are available only if you have the right permissions ("Administer Quality Profiles and Gates"):

* **Add/Remove Tags**:
  + It is possible to add existing tags on a rule, or to create new ones (just enter a new name while typing in the text field).
  + Note that some rules have built-in tags that you cannot remove - they are provided by the plugins which contribute the rules.
* **Extend Description**:
  + Extending rule descriptions is useful to let users know how your organization is using a particular rule for instance or to give more insight on a rule.
  + Note that the extension will be available to non-admin users as a normal part of the rule details.

# Rule Templates and Custom Rules

Rule Templates are provided by plugins to allow users to define their own rules in SonarQube. For instance, the template "Architectural Constraint" can be used to create any kind of rule that checks forbidden access from a set of files to another set of files.

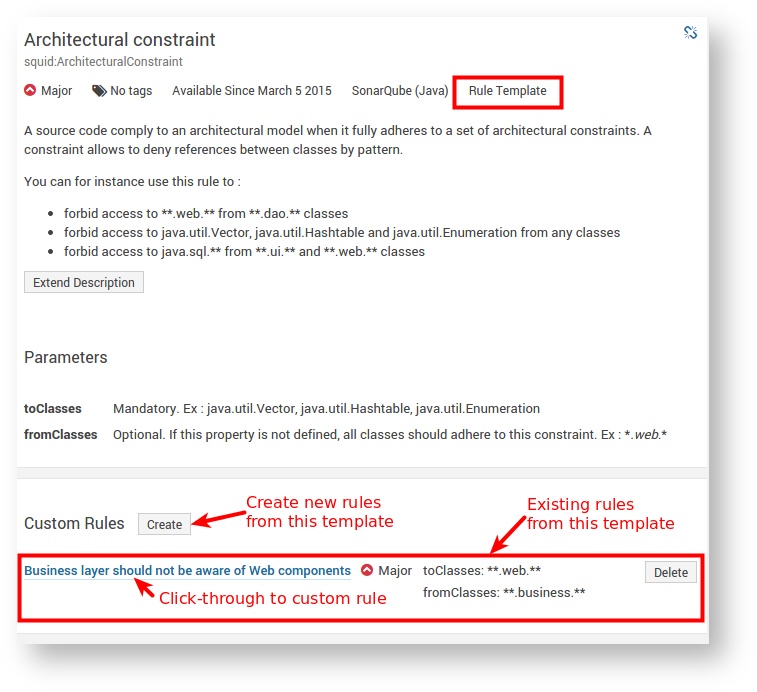
Rule templates are like cookie cutters from which you can stamp out new, "custom rules". To find templates, use the template facet:



To create a custom rule from a template, you will have to fill the following information:

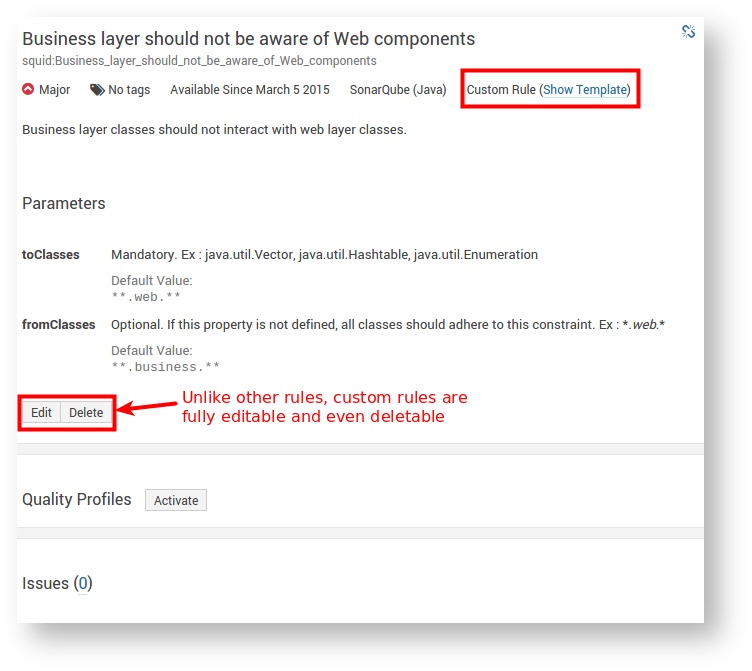
* Name
* Key (auto-suggested)
* Description (Markdown format is supported)
* Default Severity
* Status
* The parameters specified by the template

It's easy to navigate from a template to the custom rules defined from it: just click on the link in the "Custom Rules" section and you will end up on the details of the given rule.



## Custom Rules

Custom Rules are considered like any other rule, except that they can be fully edited or even deleted:



Note that when deleting a custom rule, it is not physically removed from the SonarQube instance but rather its status is set to "REMOVED". This allows current or old issues related to this rule to be displayed properly in SonarQube until they are fully removed.

# Extending Coding Rules

Custom coding rules can be added. See Adding Coding Rules for detailed information and tutorials.

**Metrics Definition:**

This is not a complete list if metrics, for the full list go through the web search for metrics in SonarQube.

**Complexity**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Key | Description | |
| **Complexity** | complexity | It is the complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each **function** has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do. | |
| **Cognitive Complexity** | cognitive\_complexity | | How hard it is to understand the code's control flow.. |
| **Complexity /class** | class\_complexity | Average **complexity** by **class**. | |
| **Complexity /file** | file\_complexity | Average **complexity**by **file**. | |
| **Complexity /method** | function\_complexity | Average **complexity**by **function**. | |

# Documentation

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Comment lines** | comment\_lines | Number of lines containing either comment or commented-out code.  Non-significant comment lines (empty comment lines, comment lines containing only special characters, etc.) do not increase the number of comment lines.  The following piece of code contains 9 comment lines:   |  | | --- | | /\*\*                                    +0 => empty comment line   \*                                     +0 => empty comment line   \* This is my documentation            +1 => significant comment   \* although I don't                    +1 => significant comment   \* have much                           +1 => significant comment   \* to say                              +1 => significant comment   \*                                     +0 => empty comment line   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*           +0 => non-significant comment   \*                                     +0 => empty comment line   \* blabla...                           +1 => significant comment   \*/                                    +0 => empty comment line    /\*\*                                    +0 => empty comment line   \* public String foo() {               +1 => commented-out code   \*   System.out.println(message);      +1 => commented-out code   \*   return message;                   +1 => commented-out code   \* }                                   +1 => commented-out code   \*/                                    +0 => empty comment line | |
| **Comments (%)** | comment\_lines\_density | Density of comment lines = **Comment lines** / (**Lines of code** + **Comment lines**) \* 100  With such a formula:   * 50% means that the number of lines of code equals the number of comment lines * 100% means that the file only contains comment lines |
| **Public documented API (%)** | public\_documented\_api\_density | Density of public documented API = (**Public API** - **Public undocumented API**) / **Public API** \* 100 |
| **Public undocumented API** | public\_undocumented\_api | **Public API** without comments header. |
| **Commented-out LOC** | commented\_out\_code\_lines | Commented lines of code |

# Duplications

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Duplicated blocks** | duplicated\_blocks | Number of duplicated blocks of lines.  For a block of code to be considered as duplicated:   * Non-Java projects:   + There should be at least 100 successive and duplicated tokens.   + Those tokens should be spread at least on:     - 30 lines of code for COBOL     - 20 lines of code for ABAP     - 10 lines of code for other languages * Java projects:   + There should be at least 10 successive and duplicated statements whatever the number of tokens and lines.   Differences in indentation as well as in string literals are ignored while detecting duplications. |
| **Duplicated files** | duplicated\_files | Number of files involved in duplications. |
| **Duplicated lines** | duplicated\_lines | Number of lines involved in duplications. |
| **Duplicated lines (%)** | duplicated\_lines\_density | Density of duplication = **Duplicated lines** / **Lines** \* 100 |

# Issues

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **New issues** | new\_violations | Number of new issues. |
| **New xxxxx issues** | new\_xxxxx\_violations | Number of new issues with severity xxxxx, xxxxx being blocker, critical, major, minor or info. |
| **Issues** | violations | Number of issues. |
| **xxxxx issues** | xxxxx\_violations | Number of issues with severity xxxxx, xxxxx being blocker, critical, major, minor or info. |
| **False positive issues** | false\_positive\_issues | Number of false positive issues |
| **Open issues** | open\_issues | Number of issues whose status is Open |
| **Confirmed issues** | confirmed\_issues | Number of issues whose status is Confirmed |
| **Reopened issues** | reopened\_issues | Number of issues whose status is Reopened |

## **Severity**

|  |  |
| --- | --- |
| **Blocker** | Operational/security **risk**: This issue might make the whole application unstable in production. Ex: calling garbage collector, not closing a socket, etc. |
| **Critical** | Operational/security **risk**: This issue might lead to an unexpected behavior in production without impacting the integrity of the whole application. Ex: NullPointerException, badly caught exceptions, lack of unit tests, etc. |
| **Major** | This issue might have a substantial impact on **productivity**. Ex: too complex methods, package cycles, etc. |
| **Minor** | This issue might have a potential and minor impact on **productivity**. Ex: naming conventions, Finalizer does nothing but call superclass finalizer, etc. |
| **Info** | Unknown or not yet well-defined security risk or impact on productivity. |

# Maintainability

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Code Smells** | code\_smells | Number of code smells. |
| **New Code Smells** | new\_code\_smells | Number of new code smells. |
| **Maintainability Rating** (formerly SQALE Rating) | sqale\_rating | Rating given to your project related to the value of your Technical Debt Ratio. The default Maintainability Rating grid is:  A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1  The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is:   * <=5% of the time that has already gone into the application, the rating is A * between 6 to 10% the rating is a B * between 11 to 20% the rating is a C * between 21 to 50% the rating is a D * anything over 50% is an E |
| **Technical Debt** | sqale\_index | Effort to fix all maintainability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days. |
| **Technical Debt on new code** | new\_technical\_debt | Technical Debt of new code |
| **Technical Debt Ratio** | sqale\_debt\_ratio | Ratio between the cost to develop the software and the cost to fix it. The Technical Debt Ratio formula is:  Remediation cost / Development cost  Which can be restated as:  Remediation cost / (Cost to develop 1 line of code \* Number of lines of code)  The value of the cost to develop a line of code is 0.06 days. |
| **Technical Debt Ratio on new code** | new\_sqale\_debt\_ratio | Ratio between the cost to develop the code changed in the leak period and the cost of the issues linked to it. |

# Quality Gates

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Quality Gate Status** | alert\_status | State of the Quality Gate associated to your Project. Possible values are: ERROR, WARN, OK |
| **Quality Gates Details** | quality\_gate\_details | For all the conditions of your Quality Gate, you know which condition is failing and which is not. |

# Reliability

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Bugs** | bugs | Number of bugs. |
| **New Bugs** | new\_bugs | Number of new bugs. |
| **Reliability Rating** | reliability\_rating | A = 0 Bug B = at least 1 Minor Bug C = at least 1 Major Bug D = at least 1 Critical Bug E = at least 1 Blocker Bug |
| **Reliability remediation effort** | reliability\_remediation\_effort | Effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days. |
| **Reliability remediation effort on new code** | new\_reliability\_remediation\_effort | Same as Reliability remediation effort by on the code changed in the leak period. |

# Security

|  |  |  |
| --- | --- | --- |
| Name | Key | Description |
| **Vulnerabilities** | vulnerabilities | Number of vulnerabilities. |
| **New Vulnerabilities** | new\_vulnerabilities | Number of new vulnerabilities. |
| **Security Rating** | security\_rating | A = 0 Vulnerability B = at least 1 Minor Vulnerability C = at least 1 Major Vulnerability D = at least 1 Critical Vulnerability E = at least 1 Blocker Vulnerability |
| **Security remediation effort** | security\_remediation\_effort | Effort to fix all vulnerability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days. |
| **Security remediation effort on new code** | new\_security\_remediation\_effort | Same as Security remediation effort by on the code changed in the leak period. |

|  |  |  |
| --- | --- | --- |
| **Classes** | classes | Number of classes (including nested classes, interfaces, enums and annotations). |
| **Directories** | directories | Number of directories. |
| **Files** | files | Number of files. |
| **Lines** | lines | Number of physical lines (number of carriage returns). |
| **Lines of code** | ncloc | Number of physical lines that contain at least one character which is neither a whitespace or a tabulation or part of a comment. |
| **Lines of code per language** | ncloc\_language\_distribution | Non-Commenting Lines of Code Distributed by Language |
| **Methods** | functions | Number of functions. Depending on the language, a function is either a function or a method or a paragraph. |
| **Projects** | projects | Number of projects in a view. |
| **Public API** | public\_api | Number of public **Classes** + number of public **Functions** + number of public Properties |
| **Statements** | statements | Number of statements. |

# Tests

|  |  |  |  |
| --- | --- | --- | --- |
| **Condition coverage** | branch\_coverage | On each line of code containing some boolean expressions, the condition coverage simply answers the following question: 'Has each boolean expression been evaluated both to true and false?'. This is the density of possible conditions in flow control structures that have been followed during unit tests execution.   |  | | --- | | Condition coverage = (CT + CF) / (2\*B)    where  CT = conditions that have been evaluated to 'true' at least once  CF = conditions that have been evaluated to 'false' at least once    B = total number of conditions | |
| **Condition coverage on new code** | new\_branch\_coverage | Identical to **Condition coverage** but restricted to new / updated source code. |
| **Condition coverage hits** | branch\_coverage\_hits\_data | List of covered conditions. |
| **Conditions by line** | conditions\_by\_line | Number of conditions by line. |
| **Covered conditions by line** | covered\_conditions\_by\_line | Number of covered conditions by line. |
| **Coverage** | coverage | It is a mix of **Line coverage** and **Condition coverage**. Its goal is to provide an even more accurate answer to the following question: How much of the source code has been covered by the unit tests?   |  | | --- | | Coverage = (CT + CF + LC)/(2\*B + EL)    where    CT = conditions that have been evaluated to 'true' at least once  CF = conditions that have been evaluated to 'false' at least once  LC = covered lines = lines\_to\_cover - uncovered\_lines    B = total number of conditions  EL = total number of executable lines (lines\_to\_cover) | |
| **Coverage on new code** | new\_coverage | Identical to **Coverage** but restricted to new / updated source code. |
| **Line coverage** | line\_coverage | On a given line of code, Line coverage simply answers the following question: Has this line of code been executed during the execution of the unit tests? It is the density of covered lines by unit tests:   |  | | --- | | Line coverage = LC / EL    where    LC = covered lines (lines\_to\_cover - uncovered\_lines)  EL = total number of executable lines (lines\_to\_cover) | |
| **Line coverage on new code** | new\_line\_coverage | Identical to **Line coverage** but restricted to new / updated source code. |
| **Line coverage hits** | coverage\_line\_hits\_data | List of covered lines. |
| **Lines to cover** | lines\_to\_cover | Number of lines of code which could be covered by unit tests (for example, blank lines or full comments lines are not considered as lines to cover). |
| **Lines to cover on new code** | new\_lines\_to\_cover | Identical to **Lines to cover** but restricted to new / updated source code. |
| **Skipped unit tests** | skipped\_tests | Number of skipped unit tests. |
| **Uncovered conditions** | uncovered\_conditions | Number of conditions which are not covered by unit tests. |
| **Uncovered conditions on new code** | new\_uncovered\_conditions | Identical to **Uncovered conditions** but restricted to new / updated source code. |
| **Uncovered lines** | uncovered\_lines | Number of lines of code which are not covered by unit tests. |
| **Uncovered lines on new code** | new\_uncovered\_lines | Identical to **Uncovered lines** but restricted to new / updated source code. |
| **Unit tests** | tests | Number of unit tests. |
| **Unit tests duration** | test\_execution\_time | Time required to execute all the unit tests. |
| **Unit test errors** | test\_errors | Number of unit tests that have failed. |
| **Unit test failures** | test\_failures | Number of unit tests that have failed with an unexpected exception. |
| **Unit test success density (%)** | test\_success\_density | Test success density = (**Unit tests** - (**Unit test errors** + **Unit test failures**)) / **Unit tests** \* 100 |

The same kinds of metrics exist for Integration tests coverage and Overall tests coverage (Units tests + Integration tests).

Metrics on test execution do not exist for Integration tests and Overall tests.

**Concepts**

# Architecture

|  |  |  |
| --- | --- | --- |
| **Analyzer** | A client application that analyzes the source code to compute **snapshots**. | See the SonarQube Platform Overview. |
| **Database** | Stores:   * configuration * **snapshots** | See the SonarQube Platform Overview. |
| **Server** | Web interface that is used to browse **snapshot** data and make configuration changes | See the SonarQube Platform Overview. |

# Quality

|  |  |  |
| --- | --- | --- |
| **Bug** | An issue that represents something wrong in the code. If this has not broken yet, it will, and probably at the worst possible moment. This needs to be fixed. Yesterday. |  |
| **Check** | Check = **Coding Rule**. |  |
| **Code Smell** | A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they'll be so confused by the state of the code that they'll introduce additional errors as they make changes. |  |
| **Coding Rule** | A good coding practice, Not complying to coding rules leads to quality flaws and creation of issues in SonarQube.  Coding rules can check quality on files, unit tests or packages. | See Viewing Unit Tests > Issues Perspective |
| **Component** | A piece of software (project, module/package, file) or a view or a developer. |  |
| **Cost** | See Remediation Cost |  |
| **Debt** | See Technical Debt |  |
| **Issue** | When a **component** does not comply with a **coding rule**, an issue is logged (was violation prior to SonarQube 3.6) on the **snapshot**.  An issue can be logged on a source file or a unit test file. There are 3 types of issue:   * Code Smell: an issue affecting your maintainability rating, preventing you to inject changes as fast as when you start from scratch * Bug: an issue highlighting a real or potential point of failure in your software * Vulnerability: an issue highlighting a security hole that can be used to attack your software |  |
| **Leak Period** | The period for which you're keeping a close watch on the introduction of new problems in the code. Typically, this is since the previous\_version, but if you don't use a Maven-like versioning scheme you may need to set a relatively arbitrary time period such as 21 days or since a specific date. |  |
| **Measure** | The value of a **metric** for a given **component** at a given time.  Examples:   * 125 lines of code on class MyClass * Density of duplicated lines of 30.5% on project myProject |  |
| **Metric** | A type of measurement. Metrics can have varying values, or **measures**, over time. Examples: number of lines of code, complexity, etc.  A metric may be either:   * qualitative: gives a quality indication on the **component** (ex: density of duplicated lines, line coverage by tests, etc.) * or quantitative: does not give a quality indication on the **component** (ex: number of lines of code, complexity, etc.) | See detailed documentation on metrics. |
| **Non-functional requirement** | Non-functional requirement = **coding rule** |  |
| **Quality Profile** | A set of **coding rules**.  Each **snapshot** is based on a single quality profile. | See Quality Profiles. |
| **Remediation Cost** | The estimated time required to fix Vulnerability and Reliability Issues. |  |
| **Snapshot** | A set of **measures** and **issues** on a given **component** at a given time.  A snapshot is generated for each analysis. |  |
| **Technical Debt** | The estimated time required to fix all Maintainability Issues / code smells |  |
| **Vulnerability** | A security-related issue which represents a potential backdoor for attackers. See also Security-related rules. |  |

**Project Administration**

**Background Tasks**

A Background Task can be:

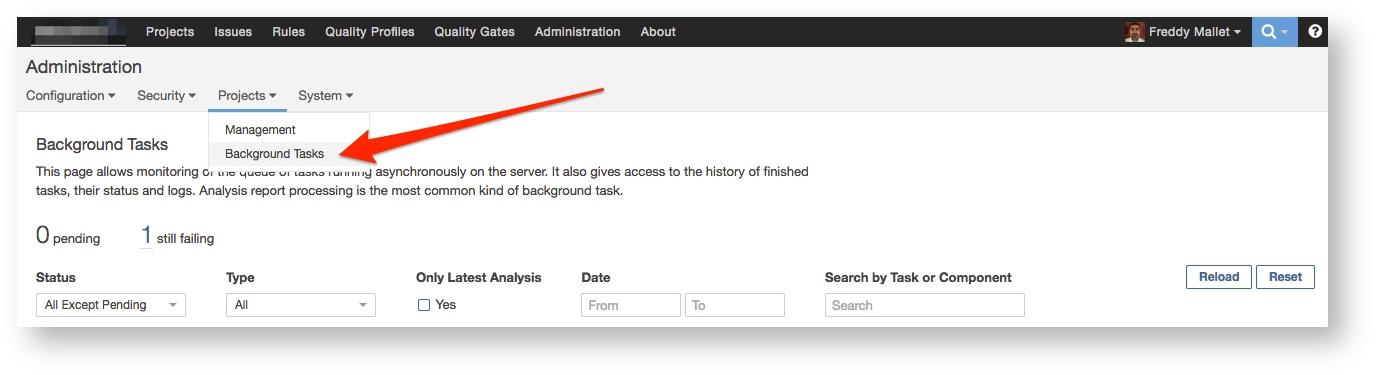
* the import of an Analysis Report
* the execution of the computation of Views

# Analysis Reports

After a SonarQube Scanner has finished analyzing your code, the result of the analysis (Sources, Issues, Metrics) -  the Analysis Report - is sent to SonarQube Server for final processing by the Compute Engine.

Those Analysis Reports are queued and processed serially. You can control the number of Analysis Reports that can be processed at a time in $SQ\_HOME/conf/sonar.properties (see sonar.ce.workerCount - Default is 1).

Administrators can view the current queue at Administration > Projects > Background Tasks.



## **Filters**

You can filter Background Tasks according to their Status: Pending, Success, Failed or Canceled

The "Only Latest Analysis" button will filter the Background Tasks by showing only the last import for each project.

You can also filter Background Tasks according to the starting date of import.

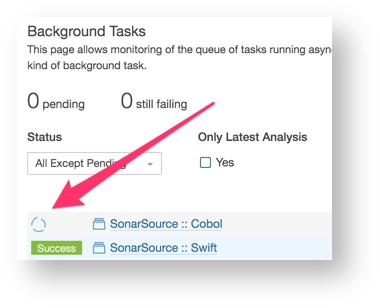
At the top of the page, are there are two to three counters:

* **pending** - shows the number of Analysis Reports queued and waiting to be processed.
* **Failures** - shows the number of projects where the processing of a project's most recent analysis report failed.
* the third counter is only present during the processing of an analysis report. It shows the duration-to-date of the task currently being processed.

At the Project level, when there is a pending Analysis Report waiting to be consumed, you have a "Pending" notification in the header.

https://docs.sonarqube.org/download/attachments/9012240/image2015-10-16%2022%3A47%3A23.png?version=1&modificationDate=1445020952000&api=v2

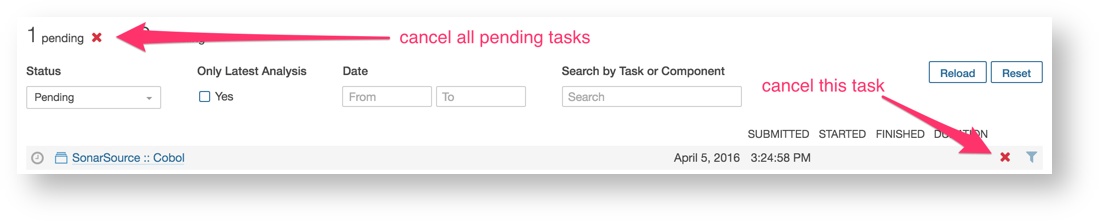
The Background Task being processing is marked with a rolling "wait" icon:



## **Cancel Import**

You can cancel the handling of a task by clicking:

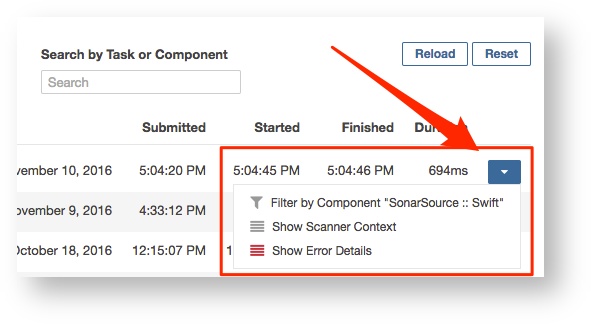
* on the red 'x' available on each line of a pending task
* on the red "bulk cancel" option next to the pending jobs count. This button cancels all pending tasks.



# Logs

For each Analysis Report you have a dropdown menu allowing you to access to the "Scanner Context" showing you the configuration of the Scanner at the moment when the code scan has been run.

In case of failure, you will have another option available on this dropdown menu: "Show Error Details", to get the technical details why the processing of the Background Task failed.



# Other Tasks

The Background Tasks page allows you also to follow other task executions such as, for example, Views Computation triggered by the Governance Plugin.

**Managing the project history**

One of the most powerful features of SonarQube is that it shows you not just your project health today, but how it has changed over time. It does that by selectively keeping data from previous analyses (see Housekeeping). It doesn't keep all previous analyses - that would bloat the database. Similarly, for the analyses it does keep, SonarQube doesn't keep all the data. Once a project snapshot moves from the "Last analysis" (i.e. the most recent) to being part of the project's history, data below the project level is purged - again to keep from bloating the database.

Typically, these aren't things you need to even think about; SonarQube just handles them for you. But occasionally you may need to remove a bad snapshot from a project's history or change the housekeeping algorithms.

# Managing History

Occasionally, you may need to manually delete a project snapshot, whether because the wrong quality profile was used, or because there was a problem with analysis, and so on. Note that the most recent snapshot (labeled "Last snapshot") can never be deleted.

About deleting snapshots

Deleting a snapshot is a 2-step process:

* The snapshot must first be removed from the project history by clicking on **Delete snapshot**. It won't be displayed anymore on this **History** page but will still be present in the database.
* The snapshot is actually deleted during the next project analysis.

At project level, from the front-page **Activity** list, choose **Show More** to see the full activity list.

For every snapshot, it is possible to manually:

* Add, rename or remove a version
* Add, rename or remove an event
* Delete the snapshot

**Notifications**

# Notification Mechanism

During each analysis, notifications are computed for each subscribed user. Then, asynchronously, these notifications are sent via email.

To set the delay between processing of the notification queue, set the sonar.notifications.delay property (in seconds) in SONAR\_HOME/conf/sonar.properties. The server must be restarted for the new value to be taken into account.

**Note**

Note that sometimes there could be some discrepancy between values displayed in the email and values displayed on the web interface. This happens if another analysis has been run between the sending of the email and your clicking on the link to the Issues drill-down in this email.

# Who gets notifications

Only users who subscribe themselves will get notifications. There is no admin functionality to proactively subscribe another user. If you believe a user should be receiving notifications, then it's time to practice the gentle art of persuasion.

# Email Configuration

To configure the email server, go to **Administration > General Settings > Email**.

Check also the **Server base URL** property at **Administration > General Settings > General** to make sure that links in those notification emails will redirect to the right SonarQube server URL.