Introdução ao SonarQube

by Mauda 2 Comments

Conteúdo do Post:

Olá Pessoal, tudo bom?

O artigo de hoje irá apresentar um pouco sobre o SonarQube, ferramenta de qualidade de código de software para diversas linguagens de programação. Veja na continuação…

SonarQube

É uma ferramenta de análise de código que auxilia o gerenciamento da qualidade interna do software. Isso é importantíssimo se queremos que o software possua uma boa qualidade e que possamos melhorá-la. Assim o *SonarQube* mais conhecido como Sonar é uma plataforma de código aberto desenvolvida pela *SonarSource* para inspeção contínua da qualidade de código com as seguintes características:

- **Multilinguagem:** *ABAP, C/C++, C#, COBOL, Flex, Go, Java, JavaScript, Objective-C, PL/SQL, PL/I,* ***PHP****, Python, RPG, Swift, T-SQL, TypeScript, VB .NET, VB6, Web, XML*
- **Integração Devops:** Build: *Maven, Ant, Gradle, MSBuild, Makefiles*
- **Integração Contínua**: *Jenkins, VSTS, TFS, Travis-CI*
- **Qualidade Centralizada:** visão integrada *“cross-projects”*

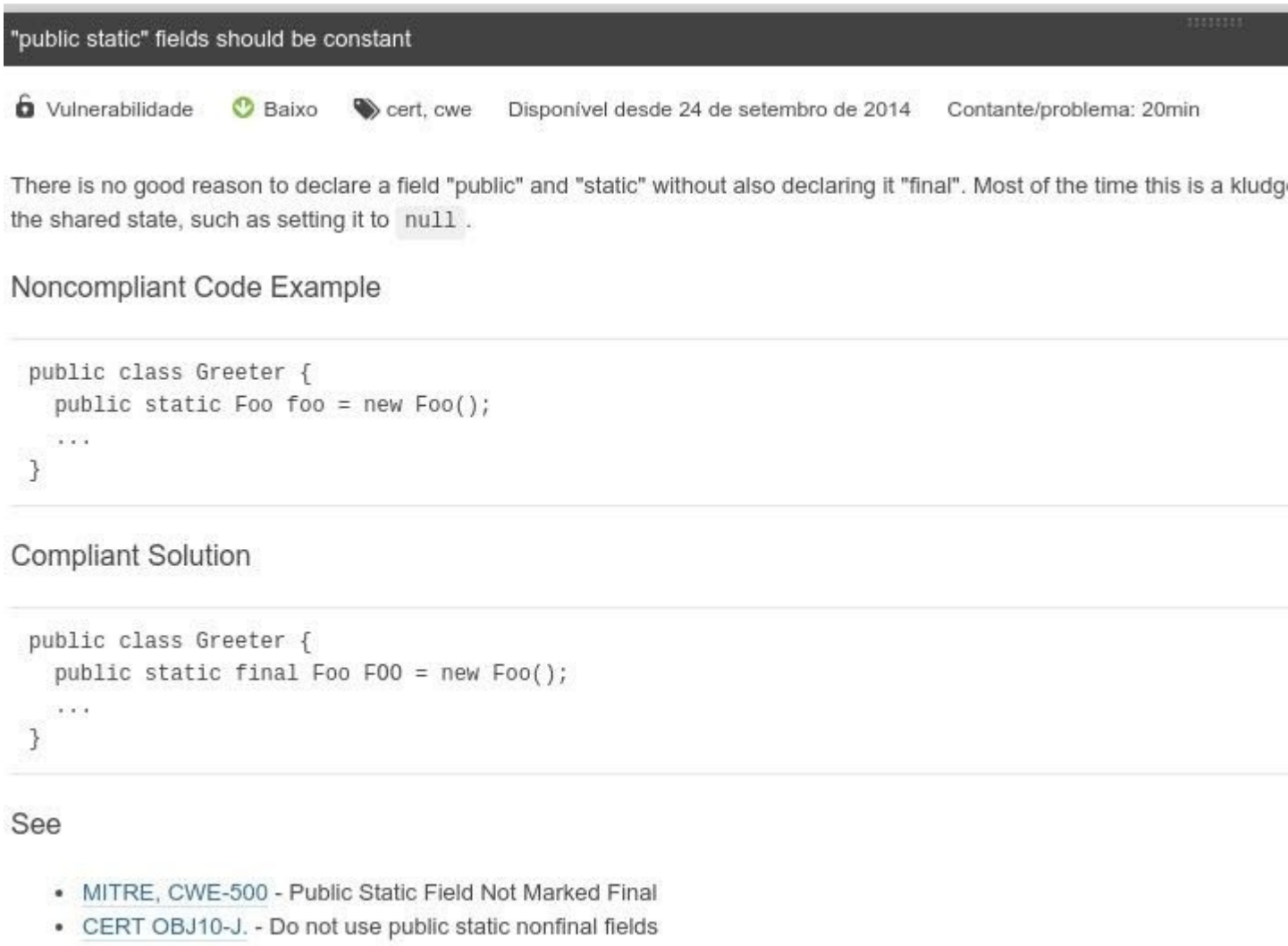
Métricas

O SonarQube oferece uma série de métricas sobre a qualidade do código, entre elas estão:

- Complexidade (ciclomática e cognitiva)
- Código duplicado
- Quantidade de Problemas
- Tamanho (quantidade de linhas de código, número de *classes*, etc…)
- Cobertura de testes
- Índice de Manutenibilidade, Confiabilidade e Segurança

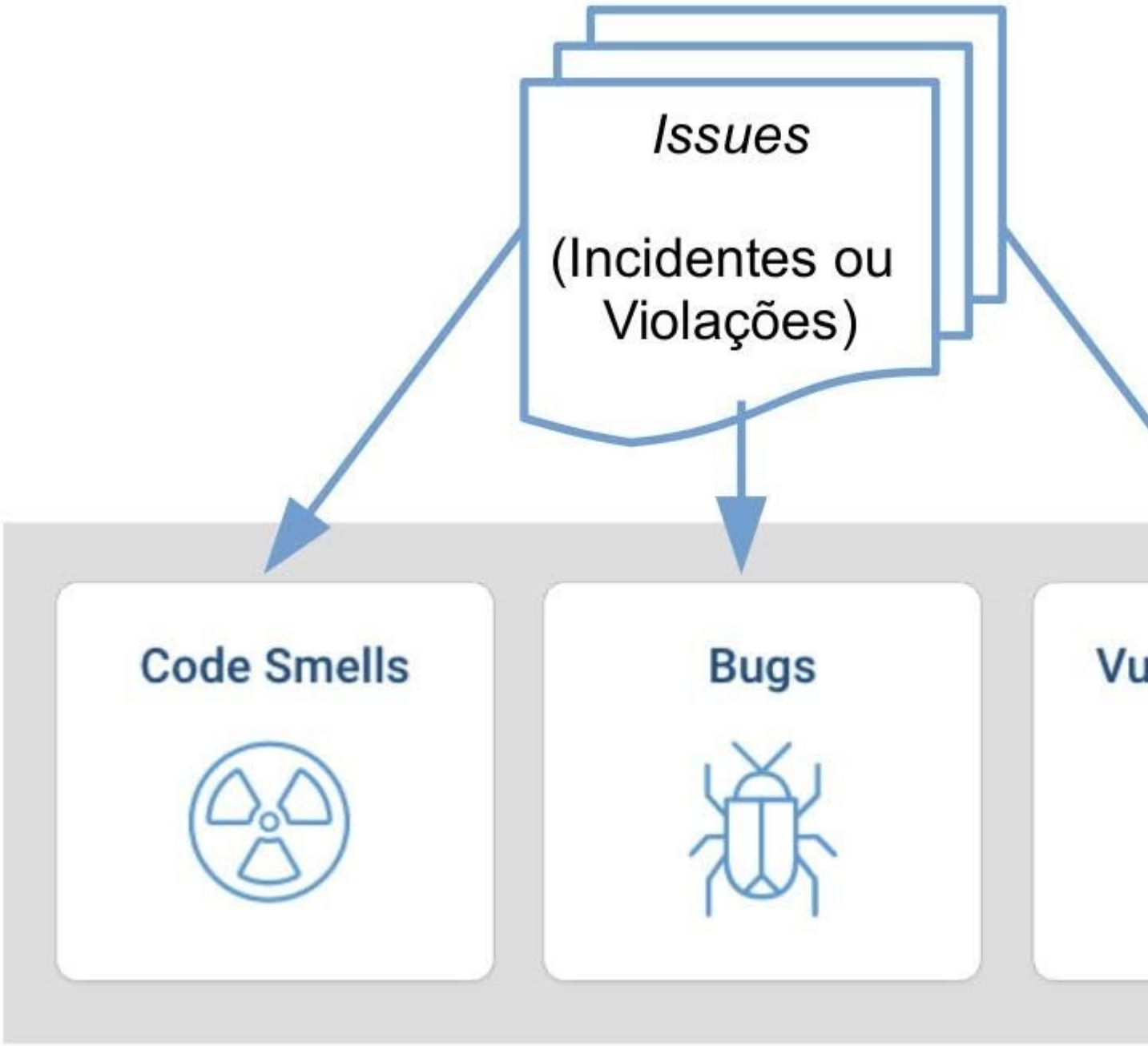
Regras

O SonarQube se baseia em regras pré-definidas para analisar o código. Uma regra é uma boa prática e cada linguagem possui um grupo de regras relacionadas. Toda a regra possui uma descrição, normalmente, com exemplo de código e links para descrições mais detalhadas, o que ajuda o desenvolvedor a entender e resolver o problema relacionado, como mostra a Figura 01.



Issues (Problemas)

Toda vez que um código quebra uma regra, uma *Issue* é gerada. As *Issues* estão divididas nas seguintes categorias:



- **Code Smells:** problemas relacionados a manutenibilidade do código. Deixar isso como está significa que, na melhor das hipóteses, os desenvolvedores terão mais dificuldade do que deveriam para fazer alterações no código. Na pior das hipóteses, eles ficarão tão confusos com o estado do código que introduzirão mais erros à medida que fizerem alterações.
- **Bugs:** itens que representam erros no código que, se ainda não aconteceram em produção, provavelmente acontecerão, e no pior momento possível. Isso precisa ser corrigido.
- **Vulnerabilities (Vulnerabilidades):** são fraquezas ou brechas de segurança na aplicação.

Além de categorias, as *Issues* também são classificadas por severidade, conforme abaixo:

Severidade	Descrição	Ação Indicada
***BLOCKER* (Impeditivo)**	*Bug* com alta probabilidade de impactar o comportamento do aplicativo em produção	Corrigir imediatamente
***CRITICAL* (Crítico)**	Um erro com baixa probabilidade de afetar o comportamento do aplicativo na produção ou um problema que representa uma falha de segurança	Corrigir imediatamente
***MAJOR* (Alto)**	Falha de qualidade que pode impactar muito a produtividade do desenvolvedor	
***MINOR* (Baixo)**	Falha de qualidade que pode afetar levemente a produtividade do desenvolvedor	
***INFO* (Informativo)**	Nem um *bug* nem uma falha de qualidade, apenas uma descoberta	

Existem sete coisas diferentes que você pode fazer com uma *Issue* (além de corrigi-la!)

Revisão Técnica	Disposição	Geral
Confirm (Confirmar)	*Assign* (Atribuir)	*Comment* (Registrar comentário)
False Positive (Falso Positivo)		
Won't Fix (Não será corrigido)		
Change Severity (Mudar severidade)		
Resolve (Resolver)		

- ***Confirm* (Confirmar):** ao confirmar um problema, você está basicamente dizendo "Sim, isso é um problema". Ao fazê-lo, ele sai do status "Aberto" para "Confirmado".
- ***False Positive* (Falso Positivo):** olhando para a questão em contexto, você percebe que, por qualquer motivo, esse problema não é realmente um problema. Então você marca como falso positivo e segue em frente.
- ***Won't Fix* (Não será corrigido):** olhando para o problema no contexto, você percebe que, embora seja um problema válido, não é um problema que realmente precisa ser corrigido. Em outras palavras, representa dívida técnica aceita.
- ***Change Severity* (Mudar severidade):** este é o meio termo entre as duas opções anteriores. Sim, é um problema, mas não é um problema tão ruim quanto a gravidade padrão da regra diz. Ou talvez seja muito pior. De qualquer forma, você ajusta a gravidade do problema para adequá-lo ao que você acha correto.
- ***Resolve* (Resolver):** se você acha que resolveu um problema em aberto, pode escolher esta opção. Se você está certo, a próxima análise irá movê-lo para o status fechado. Se você estiver errado, o status será reaberto.
- ***Assign* (Atribuir):** depois que os problemas passarem pela revisão técnica, é hora de decidir quem vai lidar com eles. Por padrão, eles são atribuídos ao último *committer* no momento em que o problema é levantado, mas você pode reatribuí-lo a si mesmo ou a outra pessoa. O responsável receberá uma notificação por e-mail da tarefa se ele se inscreveu para notificações, e a atribuição será exibida em todos os lugares em que o problema for exibido.
- ***Comment* (Registrar comentário):** a qualquer momento durante o ciclo de vida de um problema, você pode registrar um comentário sobre ele. Comentários são exibidos nos detalhes do problema. Você pode editar ou excluir os comentários que você fez.

Quality Profile (Perfil de Qualidade)

As *Issues* são geradas a partir das regras pré-definidas, que, por sua vez, estão relacionadas a um *Quality Profile* (Perfil de Qualidade). Cada linguagem possui um perfil de qualidade associado. Além disso, o *SonarQube* permite extensão de perfis através de herança.

Somente usuários com permissão podem editar um perfil de qualidade

Quality Gate (Meta de Qualidade)

Um *Quality Gate* (Meta de Qualidade) é uma meta que deve ser atingida antes que uma build possa ser liberada. Por exemplo:

Posso liberar uma build hoje? Sim, desde que:

- Os índices de Confiabilidade, Segurança e Manutenibilidade do novo código sejam = A
- 90% dos testes unitário passarem com sucesso

Somente usuários com permissão podem editar uma meta de qualidade

finally {

Duvidas ou sugestões? Deixe seu feedback! Isso ajuda a saber a sua opinião sobre os artigos e melhorá-los para o futuro! Isso é muito importante!

Até a próxima!

Filed Under: Qualidade de Código Tagged With: bugs, code smells, sonar, vulnerabilities

http://www.mauda.com.br/?p=2248

Analisar projeto Laravel 5 (PHP) com o SonarQube

- Escrito por Luís Cruz em
- This article is available in English

No artigo anterior instalámos e configurámos o SonarQube e o SonarQube Runner. Agora que temos as ferramentas instaladas, vamos analisar o projeto Laravel 5 que criámos no segundo artigo da série "Como criar um servidor de integração contínua".

Esta série de artigos sobre Integração Contínua começou com o artigo Ferramentas para servidor de integração continua (CI) e mostra como criar um servidor de integração contínua para projetos PHP.

Este artigo encontra-se dividido nas seguintes secções:

1. Instalar PHPUnit e configurar o `phpunit.xml`
2. Adicionar o ficheiro `sonar-project.properties` ao projeto
3. Executar análise do projeto com o SonarQube Runner

1. Instalar PHPUnit e configurar o `phpunit.xml`

Para que o SonarQube seja capaz de interpretar a nossa aplicação, nomeadamente a cobertura de testes ao código, vamos usar o PHPUnit. O PHPUnit vem incluído por defeito com o Laravel, mas vamos ter de o instalar no servidor e alterar as definições do `phpunit.xml`. Para isso, segue os seguintes passos.

1. Vamos começar por instalar o PHPUnit globalmente, através dos comandos:

   ```
   cd ~/Downloads
   wget https://phar.phpunit.de/phpunit.phar
   chmod +x phpunit.phar
   sudo mv phpunit.phar /usr/local/bin/phpunit
   ```

2. Para que o PHPUnit consiga exportar e guardar os resultados dos testes é necessário que tenhas a extensão XDebug ativa no PHP. Para instalares a extensão deves executar o comando abaixo.

   ```
   sudo apt-get install php5-xdebug
   ```

   Todos os comandos especificados neste artigo foram executados num Linux Mint baseado em Ubunto. Caso utilizes outra distribuição é provável que apenas tenhas de alterar o `apt-get` para o gestor de dependências correto. No entanto, é possível que os nomes das dependências sejam diferentes.
3. Para que o resultado dos testes seja interpretado pelo SonarQube, como já indiquei, é necessário ajustar a configuração do PHPUnit. Por defeito o Laravel já inclui o ficheiro de configuração `phpunit.xml`, na raiz do projeto, e é esse ficheiro que vamos alterar para o seguinte conteúdo:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <phpunit backupGlobals="false"
       backupStaticAttributes="false"
   ```

```
    bootstrap="bootstrap/autoload.php"
    colors="true"
    convertErrorsToExceptions="true"
    convertNoticesToExceptions="true"
    convertWarningsToExceptions="true"
    processIsolation="false"
    stopOnFailure="false"
    syntaxCheck="false">
  <testsuites>
    <testsuite name="Application Test Suite">
      <directory>./tests/</directory>
    </testsuite>
  </testsuites>
  <php>
    <env name="APP_ENV" value="testing"/>
    <env name="CACHE_DRIVER" value="array"/>
    <env name="SESSION_DRIVER" value="array"/>
    <ini name="memory_limit" value="2048M"/>
  </php>

  <logging>
    <log type="coverage-html" target="./ci/codeCoverage/" charset="UTF-8"
yui="true" highlight="false" lowUpperBound="35" highLowerBound="70"/>
    <log type="coverage-clover"
target="./ci/codeCoverage/codeCoverage.xml"/>
    <log type="metrics-xml" target="./ci/codeCoverage/metrics.xml"/>
    <log type="test-xml" target="./ci/codeCoverage/logfile.xml"
logIncompleteSkipped="false"/>
  </logging>
</phpunit>
```

As configurações que alterei, em relação à verão original do ficheiro, foram:

1. No grupo `<php>` incluí a propriedade `<ini name="memory_limit" value="2048M"/>` que permite aumentar o limite da memória do PHP quando os testes estão a ser executados. Embora esta propriedade não seja obrigatória eu recomendo que a adiciones para que o processo PHP tenha memória suficiente quando os testes são executados.
2. Adicionei o grupo `<logging>` que é o que realmente interessa. Esta configuração permite registar o resultado dos testes que será lido pelo SonarRunner.

   É importante registar a informação resultante do PHPUnit para que seja possível saber qual a percentagem de cobertura de testes do teu código. Ou seja, com este output do PHPUnit o SonarQube consegue detetar qual a percentagem do teu código que está testado.

   Para efeitos desta demonstração o resultado dos testes é guardado num novo diretório, em «`project_root`»`/ci/codeCoverage`.

4. Executa o comando `phpunit` na raiz do teu projeto para garantir que está tudo a funcionar corretamente. Deves ver a informação da criação do output e os diretórios serão criados automaticamente:

5. Agora faz `push` das alterações ao ficheiro de configurações para o servidor.

Antes de enviares as alterações, adiciona o diretório `ci` ao ficheiro `.gitignore`. Os ficheiros apenas são úteis para a análise do SonarQube e não existe qualquer necessidade de os colocares no repositório.

6. Depois de enviares as alterações, será executada a configuração no TeamCity e, caso acedas à aplicação, deves ver qualquer coisa como:

Aguarda que a execução termine antes de procederes ao passo seguinte.

7. Vamos verificar que os testes são executados no servidor e que o resultado é armazenado no diretório `ci`. Para isso, acede ao servidor através do terminal, ao diretório que contem o código no Teamcity Agent, como apresentado de seguida. Lembra-te de substituir a **{hash}** pelo nome correto do diretório.

```
cd /opt/TeamcityAgent/work/{hash}
phpunit
```

Se tudo correu bem, deves ver uma informação semelhante à da imagem seguinte e deverás ter o diretório `ci` criado.

2. Adicionar o ficheiro `sonar-project.properties` ao projeto

O teu projeto ainda não é visível no SonarQube porque ainda não foi executada nenhuma análise. Para que a análise seja feita corretamente, deves adicionar um novo ficheiro, com nome `sonar-project.properties` à raiz do teu projeto, com o seguinte conteúdo:

```
# Required metadata
sonar.projectKey=testproject
sonar.projectName=testproject
sonar.projectVersion=1.0.0

# Path to the parent source code directory.
sonar.sources=app

# Language
# We've commented this out, because we want to analyse both PHP and Javascript
#sonar.language=php

# Encoding of the source code
sonar.sourceEncoding=UTF-8

# Reusing PHPUnit reports
sonar.php.coverage.reportPath=ci/codeCoverage/codeCoverage.xml
sonar.php.tests.reportPath=ci/testResults.xml

# Here, you can exclude all the directories that you don't want to analyse.
# As an example, I'm excluding the Providers directory
sonar.exclusions=app/Providers/**

# Additional parameters
```

```
#sonar.my.property=value
```

Este ficheiro contem a informação necessária para a análise do projeto. Segue uma breve descrição das configurações do ficheiro.

- `sonar.projectKey`, `sonar.projectName` e `sonar.projectVersion`: são valores obrigatórios e permitem identificar o projeto no SonarQube (nomeadamente o *projectKey*).
- `sonar.sources`: Indica qual o diretório que contém o código fonte da tua aplicação. No nosso caso apontamos para o diretório `app` porque aí estará todo o nosso código.

  Não faz sentido incluir o diretório `vendor`, `node_components` ou `bower_components`, uma vez que estes diretórios já contém código testado e o código não é mantido por ti, pelo que a análise a este código causaria uma perda de tempo e poderia causar análises irrealistas. Apenas interessa o código da tua aplicação e é esse que deve ser testado e analisado.

- `sonar.language`: Indica qual a linguagem de programação a usar na análise. Como podes ver, temos esta propriedade comentada (com o caracter #) porque pretendemos ter uma análise a mais que uma linguagem (PHP e Javascript).
- `sonar.php.coverage.reportPath` e `sonar.php.tests.reportPath`: Indicam quais os caminhos do output dos testes do PHPUnit. Deves definir os mesmo caminhos já definidos no ficheiro `phpunit.xml`, no ponto anterior.
- `sonar.exclusions`: Permite definir exclusões de ficheiros ou diretórios existentes que pertencam ao diretório definido em `sonar.sources`. No nosso caso e especificamente para efeitos de testes excluimos o diretório `Providers`. Num caso real deves remover esta propriedade ou definir diretórios ou ficheiros que não devam ser usados na análise.

  Como nota, esta propriedade era útil com projetos Laravel 4, porque o diretório `app` continha ficheiros relativos a base de dados (`database`), a traduções (`langs`), ao armazenamento (`storage`) e a vistas (`views`), todos eles irrelevantes para a análise que pretendemos com o SonarQube.

- `sonar.my.property`: Esta propriedade está comentada e serve apenas para perceberes que podes definir propriedades especificas do teu projeto, que poderão depois ser usadas no SonarQube.

Existe mais informação passível de colocar neste ficheiro de configurações e existe informação específica mediante a linguagem e o tipo de projetos. Podes ver na página do GitHub exemplos do ficheiro de configuração, bastando

1. Selecionar a linguagem que pretendes
2. Procurar pelo diretório que termina em *sonar-runner*
3. Visualizar o conteúdo do ficheiro `sonar-project.properties`

Depois de adicionares o ficheiro ao projeto e o preencheres corretamente, envia as alterações para o repositório Git (`git push`).

3. Executar análise do projeto com o SonarQube Runner

Vamos executar o SonarQube Runner pela primeira vez, para que o projeto seja adicionado ao SonarQube. Para o efeito acede ao diretório da aplicação, no servidor, e executa o comando para efetuar a análise

```
cd /opt/TeamcityAgent/work/{hash}
sudo /opt/sonar-runner-2.4/bin/sonar-runner -e -X
```

Este comando deve terminar com a informação *EXECUTION SUCCESS*. Agora, quando acederes ao SonarQube, já deverás ver o teu projeto:

Se carregares no nome do projeto, és direcionado para uma página que contem os seus detalhes. No entanto, a informação apresentada ainda não é real

Isto acontece porque a análise feita ao nosso projeto foi Java (lembra-te que comentámos a propriedade `sonar.language` no ponto anterior e, como tal, o SonarQube Runner não sabe a linguagem que deve usar na analise.

Temos então de indicar ao SonarQube quais as linguagens que queremos analisar. Para isso, deves aceder a *Settings* » *System* » *Update Center*, como apresentado na imagem abaixo

Nesta página é possível instalar vários plugins e, para o nosso caso, vamos instalar três: *PHP*, *Javascript* (ambos dentro do grupo *Languages*) e *Redmine* (dentro do grupo *Integration*). Para instalares os plugins deves carregar sobre o nome do plugin e carregar no botão *Install*.

Depois de instalados os três plugins, reinicia o SonarQube com o comando

```
/opt/sonarqube-5.1.1/bin/linux-x86-64/sonar.sh restart
```

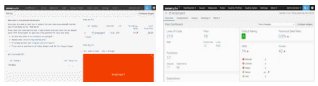
A análise ao projeto é feita através de Perfis de Qualidade. Estes perfis podem ser pré-configurados (como, por exemplo, o PSR-2) ou podem ser definidos por ti. Podes gerir os perfis no menu *Quality Profiles*.

Vamos agora associar os perfis de qualidade ao nosso projeto. Para o efeito, acede ao *Dashboard* e clica no nome do teu projeto. Dentro do teu projeto, acede ao menu *Settings* » *Quality Profiles*. Define, nessa página, o perfil *Sonar way* para Javascript e *PSR-2* para PHP.

Agora volta a executar manualmente a análise do projeto com o comando:

```
cd /opt/TeamcityAgent/work/{hash}
sudo /opt/sonar-runner-2.4/bin/sonar-runner -e -X
```

Neste momento já deves ver o número de linhas de código, a quantidade de problemas que tens para resolver e quais os ficheiros em que os tens de resolver. Existe uma panóplia de informação disponível e de formas de visualizar essa informação, que deixo para explorares.



Com isto consegues analisar o teu projeto e perceber a complexidade da tua aplicação. A minha recomendação é que faças um acompanhamento periódico à qualidade da tua aplicação e à Divida Técnica.

No próximo e último artigo desta série de Integração Contínua, vamos integrar o SonarQube com o TeamCity e com o Redmine.

Este artigo faz parte da série *Como criar um servidor de integração contínua*

- Ferramentas para servidor de integração continua (CI)
- Instalar SSH e Git em Linux e configuração da máquina de desenvolvimento Windows
- Instalação do Git e integração com o Redmine
- Instalação do TeamCity 9 em Linux
- Instalar e configurar TeamCity Agent em servidor Linux Mint
- Instalar SonarQube e SonarQube Runner em Linux Mint
- Analisar projeto Laravel 5 (PHP) com o SonarQube - Estás a ler este artigo
- Integrar SonarQube com TeamCity e Redmine

https://geekalicious.pt/pt/continuous-integration/analisar-projeto-php-laravel-5-multilingua-com-sonarqube/