Mstest xml coverage report

Thank you for your feedback, Daniel. Sorry, the sign mstest-plugin message is a bit wrong. It should specify something different: [MSTEST] XML coverage report file not found or format not supported [..] The problem you are having is the fact that the contents of the .coveragexml file may vary depending on the tool that produces them. The Mstest plug-in reads the format produced by Visual Studio (or by a handmade tool written according to the instructions given in MSDN). SonarQube reads the format produced by codecoverage.exe unique. Visual Studio can read both formats. I hope it's time to produce an XSLT .exe format in CodeCoverage to the format produced by Visual Studio. Until then, I recommend using a different tool to produce a .coveragexml file ... We updated the wiki for MSTest-plugin according to the text above. Wiki changes may take

some time to publish. Version history GitLab 11.2 introduced. GitLab Runner requires 11.2 and above. Renamed from JUnit test reports to Unit test reports in GitLab 13.4. It is very common for a CI/CD to contain a test job that will valide your consecutive pipeline code. If tests fail, pipelines fail and users are taken. The person working on the merge request must check the job logs and see where the tests failed before they can correct them. You can configure your business to use Unit test reports, and GitLab displays a report about the merge request so that it is easier and faster to identify the error without having to check the entire log. Unit test reports currently only support test reports in JUnit report format. If you are not using Defragmentation Requests but still want to see unit test report output without searching job logs, full Unit test reports are available in the pipeline detail view. Consider the following workflow: The major is rock solid, your project uses GitLab CI/CD, and your pipelines show that nothing is broken. A team sends a merge request, a test fails, and pipelines receive the known red icon. To do more research, you should review the work logs to find out the cause of the failed test, which typically contains thousands of rows. You configure unit test reports, and immediately GitLab collects and reveals them in the merge request. No more searches of the job records. The development and debugging workflow becomes easier, faster, and more efficient. How it works first, GitLab Runner GitLab loads XML files in all JUnit report format as artifacts. Then, when you visit a merge request, GitLab begins comparing the JUnit report format XML files of the head and base branch: The base branch is the target branch (usually the master page). The lead branch is the source branch (the most recent pipeline in each merge request). Reports there is a summary of how many tests failed, how many errors there were, and how many were corrected. If the comparison cannot be made, because If it is not available for the base branch, the panel shows only a list of failed tests for the head. There are four types of results: New failed tests: Test cases that pass in the base branch and fail in newly encountered errors in the head branch: Test cases that pass in the base branch and fail due to a test error in the head branch Existing errors: Test cases that failed in the base branch and failed in the head branch Resolved errors: Test samples that failed in the base branch and passed in the head branch show the test name and type from the list above. When you click the test name, a modal window opens with details of the execution time and error output. In merge requests, you must add structures, how to set up unit test reports:reports:junit in .gitlab-ci.yml, and specify the path(s) to the generated test reports. Reports must .xml files, otherwise GitLab returns an Error 500. In the following examples, the work in the test phase runs, and GitLab collects the Unit test report from each business unit. After each job is executed, XML reports are stored as structures in gitlab and shown in the merge results request widget. To make unit test report output files browseable, add them with the keyword builds:paths, as shown in the Ruby example. To load the report even if the job fails (for example, if the tests do not pass), use the builds:when:always keyword. You cannot run multiple tests with the same name and class in your JUnit report format XML file. Use the following job in the ruby sample .gitlab-ci.yml. Structures to link to a unit test report output file:paths contain the keyword. ## Use to create XML file in JUnit report format with RSPEC ruby: phase: test script: - package installation - package exec rspec --format progress --format RspecJunitFormatter --out rspec.xml artifacts: time: always paths: - rspec.xml reports: junit: rspec.xml Go sample .gitlab-ci.yml use the following job, and -make sure you use set-exit-code, otherwise the pipeline will succeed, even if tests fail: ## Use to create XML file in JUnit report format with Golang: stage: test script: - go -u github.com/jstemmer/go-junit-report - test -v 2>&1 | go-junit-report -set-exit-code > report.xml artifacts: time: always reports: junit: report.xml Java examples There are several tools that can produce Java JUnit report format XML file. Gradle In the following example, the gradient is used to create test reports. If there are multiple test tests defined, the gradient creates multiple indexes under build/test results/. In this case, you can have glob matching by defining the following path: build/test-results/test/**/TEST-*.xml: java: stage: test script: - gradient test structures: when: always reports: junit: In GitLab Runner 13.0 and later, after, **' can be used. To dedicate maven Surefire and Failsafe test reports,.gitlab-ci.yml: java: stage: test script: - confirm mvn works: when: always reports: junit: - target/surefire-reports/TEST-*.xml - target/failsafe-reports/TEST-*.xml Python example This example is pytest Uses -junitxml=report.xml flag JUnit report to format output in XML format: pytest: stage: test script: - pytest --junitxml=report.xml artifacts: time: always reports: junit: report.xml C/C++ sample C/C++ There are several tools that can produce JUnit report format XML files. GoogleTest In the following example, gtest is used to create test reports. If you have multiple gtest executables created for different architectures (x86, x64, or arm), it will be necessary to run each test that provides a unique file name. The results are then collected together. cpp: phase: test script: - gtest.exe --gtest_output=xml:report.xml artifacts: time: always reports: junit: report.xml CUnit CUnit cunit cunit report format xml files can be made to produce automatically when run using cUnitCI.h macros: cunit: phase: test script: - ./my-cunit-test file: when: always reports: junit: ./my-cunit-test.xml .NET sample JunitXML.TestLogger NuGet package can create test reports for .Net Framework and .Net Core applications. The following example expects a solution with one or more project files in the root folder of the store, in the subfolders. One result file is generated per test project, and each file is placed in a new build folder. This example includes optional formatting arguments that improve the readability of test data in the test widget. It's a complete... A Net Core sample is available. ## Source code and documentation here: Test: stage: test script: - 'dotnet test --test-adapter-path:. --logger:junit; LogFilePath=.. \artefakt\{build}-test result.xml; MethodFormat=Class; FailureBodyFormat=Verbose' works: when: always ways: . /**/*test-result.xml reports: junit: - ./**/*test-result.xml There are several tools in JavaScript that can produce XML files in JUnit report format. Jest-junit npm package can create test reports for JavaScript applications. In the following example .gitlab-ci.yml, javascript uses Gesture to create business test reports: javascript: scene: test script: - 'gesture --ci -reporters=default --reporters=gesture-junit' artifacts: time: always reports: junit: - junit.xml Karma-junit-reporter npm package you can create test reports for JavaScript applications. In the following .gitlab-ci.yml example, javascript uses Hash to create business test reports: javascript: stage: test script: - hybrid startup -reporters junit artifacts: when: always reports: junit: - junit.xml GitLab Version history Display Unit test reports GitLab 12.5 behind a property flag (junit_pipeline_view), disabled by default. Property flag became and the feature is usually available in GitLab 13.3. If JUnit report format XML files are created and loaded as part of a pipeline, those reports can be displayed within the consecutive pages page. The Tests tab on this page displays a list of test packages and cases reported from the XML file. You can view all known test suites and click on each of them to see more details, including the situations that make up the package. You can also get reports through the GitLab API. GitLab 13.0 introduced GitLab Version history JUnit screenshots viewing. By default, a disabled property is deployed behind the flag. To use GitLab in self-managed instances, ask for a GitLab administrator to enable it. If the JUnit report format XML files contain an additional tag, GitLab deeers the attachment. Upload your screenshots as objects to GitLab. The attachment label must contain the absolute path to the screenshots you upload. <testcase time=1.00 name=Test> <system-out>[[EK|/absolute/path/to/some/file]]</system-out> </testcase> When this problem is complete, the attached file will be visible on the pipeline details page. Enabling the JUnit screenshots feature This feature comes with a :junit_pipeline_screenshots_view feature flag that is disabled by default. To enable this feature, ask a GitLab administrator who has access to the Rails console to run the following command: Feature.enable(:junit_pipeline_screenshots_view) Feature.enable.enable(:junit_pipeline_screenshots_view)