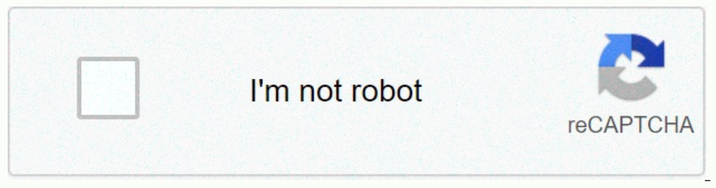


Continue

# Allure report jenkins pipeline

Allure is designed to be highly customizable, because practice shows that many teams around the world may have different measurements, or perhaps developed unique approaches to testing, their products might require something completely different from the initial requirements Allure was built to comply. To deal with this kind of challenge Allure has a plugin system, which gives a lot of flexibility to the representation of the report. To be precise, all the basic features are modular in plugins, and it would be too long to list them all here. But almost all of the features described in the features section [link to allure features section] is implemented internally as a plugin. Let's take a look at the upper class diagram in the Allure plugin system. Plugin classes can extend 3 basic interfaces that provide different aspects of functionality: Reader allows you to implement a readResults method, which defines the logic of reading results with the ResultsVisitor instance from the directory with test results. Aggregator implements the overall method, which defines the aggregation logic of the results processed in all results files, and writes the resulting data to the report directory. Widget this interface allows to implement the getData method that again defines the logic of aggregating processed results, but this time the resulting data is stored in the widget.json file to be used by the named widget as a result of the value provided in getName. The contexturing interface with the T getValue method() allows you to create a utilitarian class to use for all plugins via a Configuration.requireContext(Class) method, available from a configuration instance, provided as a parameter in all 3 methods above. Front side of the Allure is built using the BackboneJS frame. Thus, some basic understanding of its internal mechanisms may sometimes be necessary. Api is accessible from the global object allure.api. Let's take a look at the list of features it provides: addTab (tabName, title, icon, route, onEnter - notFound) - can be used to set a new tab for the report that will appear on the left flap menu with tabName name, its icon will be defined by a cs style provided in the icon chain, route will set an address for a new page. onEnter must be a function, which instantiated a class Show the representation management of your new tab. addTranslation (lang, json) - gives you the ability to support several languages to name in the widgets or test case blocks you've created. lang is a language key, and json is a json object that contains maps for string values in the specified language. translate (name, options) - is necessary if you generate html code in your plugin and do not use existing components, provided in allure.components. In the tab example above, you need to wrap the strings in the model in this function call to activate the tab. translation to be taken in the global register. See more in docs for i18next. addWidget (name, Widget) - is a way to create a new widget on the Overview page of the report. Name will set its name displayed, and Widget is a view to add to the widgets grid. Api provides you with a basic class for a speed.components.WidgetStatusView widget, which we'll review later in the Behavior Plugin section. But you can design a widget for your own needs extending from Backbone.Marionette.View, just keep in mind that this widget definition is designed to pop up the data to fill the template for that view from the widgets.json file, by the key you provided in the name setting. addTestcaseBlock (view, 'position') - allows you to add a View class to the Test Case page, in one of the 3 possible block groups, determined by a position argument. The position can be one of these values: mark, after or before. To understand what kind of information you can attach to the test case page, jump to the section with related features [jump to the feature list] Here we'll cover the steps it usually takes to build a new plugin. Basically, any plugin will consist of two main parts: the Java classes that will process the report data and produce certain results in the report folder. Script JS that takes the stored results and creates a representation for them on the front side of the report, such as a widget or an additional tab. Typical structure of a plugin module would look like this: /my-plugin /src /dist /static allure-plugin.yml /hand /java /my.company.plugin build.gradle Here in src/dist/static all static files .js and .css are stored, and everything on src/hand/java is a Java code of data processing. allure-plugin.yml - is a configuration file. File content look-plugin.yml This file contains guidelines in human-readable format that the plugin charger will use more to locate resources and connect the plugin. allure-plugin.yml id: my-plugin name: Plugin name goes here description: More detailed explanation of what this plugin does. extensions: - my.company.allure.CustomPlugin // - Fully qualified class names that implement the Extension interface and include data processing features. - my.company.allure.CustomWidget jsFiles: - index.js cssFiles: - styles.css Adding allure-plugin-api dependency To be able to use the API, you should simply download the look-plugin-api dependency from the jcenter repository. To do this add to your project construction script: outbuildings - compileOnly 'Dependence', 'io.qameta.allure.allure'/groupId's 'adid',api-lt;/artifactId; Version-lt-paceVersion/version will contain captured test case arguments in the parameters. uuid:0edd28b1-3c7f-4593-8dda-db9aa04891f, fullName:io.qameta.allure.animals.AnimalsTest .angryCat, name:angryCat, status:p, stage:finished, start:1495467840415, stop:1495467840416, parameters:[name:arg0, value:Hiss! We are preparing to write a new plugin in its own right that adds a new tab with some representation of the test results and creates a widget to place on the Overview tab with some digested data. For example, consider a plugin that extracts past and failed settings from these set-up tests, creates a new tab, and a widget where only recent failures are displayed. We should start by writing a Java class that implements the Aggregator and Widget interfaces. MyPlugin.java classe publique MyPlugin implémente agrégateur, widget { @Override agrégat de vide public (configuration finale configuration, lancements de liste<LaunchResults> finale, final Path outputDirectory) lance IOException { final JacksonContext jacksonContext = configuration .requireContext (JacksonContext.class); final Path dataFolder = Files.createDirectories (outputDirectory.resolve(« data »)); final Path dataFile = dataFolder.resolve (« myplDirectories(outputDirectory.resolve »)); final Path dataFile = dataFolder.resolve (« myplugindata.json »); résultats finaux du<TestResult> fluxStream = launches.stream() .flatMap (lancement -> launch.getAllResults().stream()); essayez (OutputStream os = Files.newOutputStream(dataFile)) { jacksonContext.getValue().writeValue(os, extractData(resultsStream)); } } extrait<Map> de collection privéData (test final<TestResult> streamResults) { } @Override objet public getData (configuration configuration, lancements de<LaunchResults> liste) { Flux<TestResult> filtréResults = launches.stream().flatMap (lancement -> launch.getAllResults().stream()) .filter(result -> result.getStatus().equals(Status.FAILED)); return extractData(filteredResults); } @Override public String getName() { return « mywidget »; } } } What's going on in the code above? In the overall method, data extracted from test results in the extractData method are written on the myplugindata.json file that is stored in the report data folder. To create an appropriate .json file, a JacksonContext is used to get a map instance. This data will be displayed on the new tab. getData implementing the method creates data to use in the new widget, and the getName method defines the entry name for the widgets.json file where that data will be stored. myplugindata.json [ sounds: [Growl! and Hiss! ], name: angryCat, sounds: [Oink!, Meow!,, name: hungryCat - sounds: Bark!, Woof!, Moo! ], name: bigDog - ] widgets.json ... mywidget: [sounds: [Oink!), name: hungryCat, sounds: [Moo!), name: bigDog], ... Your plugins may need to share some common utilities that would be wise to make available on demand. A quick example of such a utility class would be JacksonContext, /TestResults/LaunchResults can be used to get a map to serialize Java objects with data in the JSON report files. JacksonContext implements Context (objectMapper - private final ObjectMapper mapper; public JacksonContext)) -this.mapper - new ObjectMapper() .configure (MapperFeature.USE_WRAPPER_NAME_AS_PROPERTY_NAME, true) .setAnnotationIntrospector (new JaxbAnnotationIntrospector (TypeFactory.defaultInstance()))).enable(SerializationFeature.INDENT_OUTPUT.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES.setSerializationInclusion (JsonInclude.Include.NON_NULL); @Override public ObjectMapper getValue() Then, from a plugin class, it can be viewed from the configuration instance as in step 2. Here, we move to the front side of the Allure report and start by adding JavaScript code to the index.js file. Backbone manages the data with models or collections, on the previous step, we have recorded the data from our page as a collection, so that the tab model should extend Backbone.Collection. This object will contain the file data, specified in the url. Then, for your new tab, you need to extend a View class from the basic AppLayout class that already contains the report's left navigation menu. It is provided in the global look object: var MyTabModel - Backbone.Collection.extend(url: 'data/myplugindata.json' ) class MyLayout extends allure.components.AppLayout initialize() - this.model - new MyTabModel(); 'loadData'; ' ' return this.model.fetch('); 'getContentView' 'return new MyView' ('items: this.model.') models); In the MyLayout class, you can replace a getContentView method to set another View class that will manage the content of your tab. Here is some simplistic implementation of the Class Show, model is a model function that returns html model with added data. model const - function (data) - html - "H3 class'pane__title my tab'/h3'; for (var item of data.items) - html - item.attributes.name ' says: 'item.attributes.$el.html sounds' ' ' ' ' After all that addTab function call would look like this: allure.api.addTab ('mytab', 'My Tab', icon: 'fa fa-trophy', route: 'mytab', onEnter: (function () 'new return MyLayout') This will finally give you a new tab: To create a new widget, you need to implement a small View class that handles the data you put in widgets.json on step 2. Note, that if you return getData data as a it will eventually be provided to the widget as a table, which can be obtained as this.model.get (articles). In the code below the model function defines the actual html to display in the widget. index.js MyWidget expands to the data - return widgetTemplate (data) - serializeData() - return - items: this.model.get ('items'), 'look.api.addWidget('mywidget', MyWidget); This finally gives us a new widget on the Overview dashboard. Going back to the tab example, it's very easy to activate the strings translated into it. In the templates, you need to replace simple text strings with reserved spaces and use the translate function, and you also need to save translations via addTranslation. model const - function (data) - html - "lt;h3 class'pane__title'- pace.api.translate (mytab.name) ' lt;/h3'; for (var item of data.items) - html - 'item.attributes.name' says: 'item.attributes.sounds' ' ' ' ' ' ' ' ' ' return html; 'allure.api.addTranslation('en', 'mytab': 'My Tab', '' allure.api.addTranslation ('ru', 'mytab': 'name: "" " ', ', ', ', '); Internally, many Allure features are implemented using api plugin, let's see how for example links are added to the test case page. With the addTestcaseBlock method, you can set a view that you assume will have a test case object as a model available at this.model. LinksView.js './styles.css'; import 'View' from 'backbone.marionette'; import 'className' from '.. /.. /decorators'; './LinksView.hbs' import model; @className ('pane__section') class LinksView Extends View 'Model', Model; serializeData') - back - links: this.model.get ('links') Handlebars is used as a model engine: LinksView.hbs -#if links - 'lt;h3 class'pane__section-title 'testCase.links.links'./h #each 3 '#if (eq type 'issue'#if) 'a class'link href',this.url' target_blank'lt'/at;/span'; '/each' index.js import LinksView from './LinksView'; allure.api.addTestcaseBlock (LinksView, 'position: 'before'); This adds a links section to the test case: When you build a plugin, you need to find the next structure, which can then be copied into the plugins folder of the command line distribution. /my-plugin allure-plugin.yml plugin.jar /lib dependence.jar /static styles.css index.js plugin.jar - is a pot with your plugin classes compiled /lib - all defies of your pugin must be placed here /static - a folder containing all static .js and .css files. Here is a gradle construction script template for a plugin project that uses Java Library Distribution Plugin to pack plugin classes and copy files and dependencies into a .zip archive. repositories - jcenter() apply plugin: 'java-library-distribution' jar 'archiveName' 'plugin.jar' 'dependence' compileOnly ('io.qameta.allure:allure-plugin-api:2.0-BETA8') - command Allureline distribution has a following folder structure: /allure-commandline /bin /bin allure.yml /lib /plugins /behaviors-plugin /junit-plugin /screen-diff-plugin Here, in plugins folder plugins distributions to use to the resident report generation. By default, several plugins are already added to the Allure. Their use is managed by default build profile configuration file'/config/allure.yml'. In this file, under the plugin plugins folders to use are listed, so its content should look like this: allure.yml plugins: - behaviors-plugin - junit-plugin - screen-diff-plugin To activate your own plugin, copy the folder with plugins file distribution, then add the folder name to the corresponding build profile configuration: /allure-commandline/bin /config allure.yml/lib /plugins/behaviors-plugin /junit-plugin /screen-diff-plugin /my -plugin allure.yml plugins: - behaviors-plugin - junit-plugin - screen-diff-plugin - my-plugin Several important look features are implemented as decoupled plugins that are stored independently under the folder plugins of the Allure Commandline distribution. Their use can be managed by build profile features (jump to configuration (link to section)[Order line configuration]). Let's start with studuying how one of the simplest plugins works. In this section, we'll quickly review a plugin available in the Allure distribution that allows you to edit a logo image displayed in the top left corner of the report. plugin sources structure directory: /src /dist allure-plugin.yml /static custom-logo.svg styles.css build.gradle allure-plugin.yml id: custom-logo name: Custom logo aggregator description: The aggregator replaces the Allure logo default by a custom cssFiles: - styles.css custom logo.svg - is a vector graphic file with a logo to use styles.css - a css file that adds style, which will replace the default logotype. styles.css .side-nav__brand - background: url ('custom-logo.svg') left center without repetition; margin-left: 10px; Plugin Behaviors is created to support the behavioral approach in testing with the Allure report. Test cases must have feature and history labels, this plugin will aggregate and create a widget showing the statistics of the results of the stories by each feature, and a new tab, where all test results are grouped by their features and stories. In Java-based adapters, you can mark your tests with @Feature and @Story annotations. In JS-based adapters, you can use allure.feature (featureName) and allure.storyName methods In CucumberJVM Features and stories are extracted exactly as the tests are features and scenarios. Once your tests are properly labeled, you can start using the Behaviors plugin immediately since it is included in the default Allure distribution. You can find plugin sources in the folder plugins of the Allure 2 behaviors plugin structure: /src / dist allure-plugin.yml / static index.js/ main / java / io.qameta.allure.behaviors BehaviorsPlugin.java BehaviorsPlugin.java This plugin creates a different representation for the test results tree, which is why its BehaviorsPlugin class inherits a basic class providing tree aggregation - AbstractTreeAggregator and implements a Widget interface to prepare data from a widget on the Overview report dashboard. To provide an aggregation classifier, plugin must implement a ListGroupGroups (TestResult final result) method @Override-TreeGroup Protected List getGroups (final result of TestResult) - back Arrays.asList (TreeGroup.allByLabel (result, LabelName.FEATURE, DEFAULT_FEATURE), TreeGroup.allByLabel (result, LabelName DEFAULT_STORY. These groups will determine a placement for each TestResult when building a clustered results tree to store in the behaviors.json file. Another part of the BehaviorsPlugin class is a getData method that includes the implementation of the Widget interface. This method prepares aggregated information on the amount of stories spent by each feature, which will be put into the widget.json file. After that, in the index.js api calls for the creation of a new tab and the creation of a new widget are: allure.api.addTab (behaviours, title: tab.behaviors.name, icon: 'fa fa-list', route: 'behaviors(/:testcaseId)', onEnter: (function () - var routeParams - Array.prototype.slice.call (arguments); return new allure.components.TreeLay tab.behaviors.name out }); allure.api.addWidget ('behaviors', allure.components.WidgetStatusView.extend(title: 'widget.behaviors.name', baseUrl: 'behaviors', showLinks: false, showAllText: 'widget.behaviors.showAll'); Note a special TreeLayout component that can be used to display all types of data, produced by AbstractTreeAggregator implementations. In addition, a route attribute defines a template for links to test case pages. Junit plugin works without a front part, it is created to allow the Allure generator to process the junit xml ratio format in the Allure test results, this plugin is enabled by default, so when you use allure generate the command for a folder with junit test results, a report will be generated. Generated. TreeGroup/TreeGroup