Oracle® JavaScript Extension Toolkit (Oracle JET)

Using and Extending the Oracle JET Audit Framework

17.1.0
G12878-01
November 2024

Oracle JavaScript Extension Toolkit (Oracle JET) Using and Extending the Oracle JET Audit Framework, 17.1.0

G12878-01

Primary Author: Oracle Corporation

Contents

15 Reference: Custom Audit Rule Utility Libraries

Preface

Using and Extending the Oracle JET Audit Framework describes how to use and extend audits with the Oracle JET Audit Framework.

Topics:

Audience

Using and Extending the Oracle JET Audit Framework is intended for Oracle JET application developers who want to use and extend the Oracle JET Audit Framework to audit their applications.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Resources

For more information, see these Oracle resources:

- *Developing Oracle JET Apps Using MVVM Architecture*
- Oracle JET Web Site
- API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET)
- Oracle® JavaScript Extension Toolkit (JET) Keyboard and Touch Reference
- Oracle® JavaScript Extension Toolkit (JET) Styling Reference

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Get Started with the Oracle JET Audit Framework

The Oracle JET Audit Framework (JAF) is a command-line utility and supporting API that allows you to audit JET applications by using a rich set of built-in audit diagnostic rules. The API supports extending JAF with custom audit rules that you write to meet specific diagnostic requirements of your application.

About Auditing Oracle JET Applications

Oracle JET Audit Framework (JAF) is dynamic and performs an audit of Oracle JET project files based on runtime options it finds in a configuration file.

To get started with JAF, the initial creation of the configuration file is automated to help you get up and running quickly. After a default configuration file is created, you have many options to tailor the audit to your project needs.

With Oracle JAF installed, audits that you perform against your JET project run in a command-line interface, similar to the Oracle JET CLI. The installation of JAF provides a lint-style command-line utility named `ojaf` that audits JET application files (currently HTML, JS, TS, TSX, CSS, and JSON) by applying rules that perform a static analysis from an Oracle JET perspective.

The audit diagnostic messages returned by invocation of the `ojaf` utility result from built-in rule sets that are specific to the Oracle JET release version of the application. As new versions of Oracle JET are released, you can update JAF and keep current with the latest applicable audit rules. For this reason, you will want to update JAF regularly.

The audit rules that are specific to each version of Oracle JET are called the *built-in rules*. Additionally, you can configure the audit to run with custom, *user-defined rules*. Both the built-in rules and rules that you may write yourself, are logically and physically grouped together in a rule pack.

Like an Oracle JET installation, installation of Oracle JAF requires that Node.js is installed as a prerequisite. The `ojaf` utility will report if the Node.js version does not meet the minimum node version requirement.

Before you run the audit, you use the JET tooling to initialize Oracle JAF and to scaffold a default JAF configuration file, `oraclejafconfig.json`. You can customize the properties of the Oracle JAF configuration to control many aspects of both the rule set (the set of active rules) and the file set (the set of files to be audited) for a specific audit run:

- Any rule may be disabled.
- One or more rules may be designated by name to be run and all others excluded.
- Rule groups may be specified (for example, only run rules related to HTML).
- Rules may selected by severity of the issue they report.
- All built-in rules may be suppressed (allowing only user-defined rules to run).
- All rules specified in a defined rule pack may be disabled.

- The target file set can be defined easily, including by using glob support for both inclusion and overriding exclusion.
- Multiple configuration files can created for specific runtime criteria or projects. The configuration files are JSON format, but JavaScript/TypeScript style comments are permitted for documentation purposes. The configuration file to be used can be specified on the command-line.

If the built-in audit rules provided with the JAF installation do not meet all the diagnostic requirements of your application, you can write custom audit rules to extend JAF. You implement user-defined audit rules as JavaScript files. The JAF API allows you to register event listeners and handle the audit context created by JAF on the file set of your JET projects. Custom audit rules can be assembled into distributable rule packs and invoked by developers on any Oracle JET application.

Typical Workflow for Auditing an Oracle JET Application

Understand auditing options for Oracle JET web and hybrid mobile applications.

To audit the source files in an Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Initialize the Oracle JET Audit Framework (JAF) and perform an audit dry-run	Create a default `oraclejafconfig.json` file and run a dryrun report to verify the files that will be audited.	Initialize Oracle JAF and Run an Audit
Configure JAF	Edit the `oraclejafconfig.json` file to limit the scope of the audit to the desired file set or rule severity level, for example.	Configure the JET Audit Framework
Run an audit	Report the diagnostic messages discovered by the rule set for a specific Oracle JET version.	Run Audits on Oracle JET Applications
Fine-tune audit results	Hide diagnostic messages emitted by specified rules or comment source code with JAF comment commands to limit the scope.	Fine Tune the Audit
Customize audit reports	Format the optional JSON output of the audit in a custom report.	Work with the Output of Audits

Typical Workflow for Writing Custom Audit Rules

Understand how to interact with the Oracle JAF API and work the various utility libraries to create user-defined, custom audit rules that extend JAF.

To write custom audit rules and assemble your JavaScript implementation files as a rule pack that you can share with JET application developers, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand the JAF concepts for writing custom rules	Learn about the various JAF audit context events and the properties that you can access on the context object passed at runtime to your audit rules by the JAF audit engine.	About the JAF Audit Engine
Understand the custom audit rule implementation	Learn how you can handle the audit context object by registering a listener to respond to context events and learn about the miscellaneous helper functions provided by JAF utility libraries that you can access on the context object.	Understand the Structure of Custom Audit Rules and Reference: Custom Audit Rule Utility Libraries
Understand the requirements to create and to distribute custom rules	Set up a JET development environment to implement your custom rules before you distribute the rules as a rule pack.	Get Started Writing Custom Audit Rules
Write custom rules that audit file data parsed by JAF	Review sample custom (node) rules that show how to audit for data nodes returned on your application's target file set, including HTML, JSON, JavaScript/TypeScript, and CSS files.	Implement Custom Node Rules
Write custom rules that work with the phases of the JAF lifecycle	Review sample custom (hook) rules that show how you can invoke an audit at various stages of the JAF audit lifecycle, such as upon audit startup.	Implement Custom Hook Rules

Install the Oracle JET Audit Framework

Use npm to install the Oracle JET Audit Framework (JAF).

- Like an Oracle JET installation, installation of Oracle JAF requires that Node.js is installed as a prerequisite. For additional information, see Install Node.js.
- At the command prompt, enter the following command as Administrator on Windows or use `sudo` on Macintosh and Linux machines:

```
[sudo] npm install -g @oracle/oraclejet-audit
```

  It may not be obvious that the installation succeeded. Enter `ojaf -v` to verify that the installation succeeded. If you do not see the Oracle JAF version, scroll through the install command output to locate the source of the failure.

  - If you receive an error that your version of Node.js is outdated, download and install the recommended version.
  - If you receive an error related to a network failure, verify that you have set up your proxy correctly if needed.
  - If you receive an error that your version of `npm` is outdated, type the following to update the version: `[sudo] npm install -g npm`.

Initialize Oracle JAF and Run an Audit

If you have an Oracle JET application you created using the JET command-line interface, then you can auto-configure an audit for the application and run the audit in the command prompt window.

In the JET application, open a command prompt and, from the root directory, use the Oracle JET Audit Framework (JAF) command to initialize a default JAF configuration for the application.

```
ojaf --init
```

When you initialize Oracle JAF, the JET tooling scaffolds a default JAF configuration file named `oraclejafconfig.json`. The tooling creates the JAF configuration file in the root directory and defines the default configuration settings based on the JET application configuration file `oraclejetconfig.json`, also located in the application root directory.

Before you audit the application for the first time, you can confirm the default configuration for the application files that JAF will audit. This command is called a *dry-run* because it does not audit the application but confirms the files to be audited based on current JAF configuration settings.

```
ojaf --dryrun
```

To perform an audit of your application, enter the command to invoke the JAF audit utility.

```
ojaf
```

When you run the audit, Oracle JAF searches the directory in which you initiated the audit for the JAF configuration file `oraclejafconfig.json`. If no configuration file is found there, then JAF processes only HTML files found in the current directory and will always use the default JAF configuration for the audit.

On subsequent invocations of `ojaf`, a check is made to see if the Oracle JET configuration file (`oraclejetconfig.json`) has changed since the last `ojaf` invocation. If changes are detected, such as might occur when you migrate your application to a new JET version, then JAF updates certain settings in the JAF configuration file automatically. The configuration property settings that JAF monitors and updates based on JET configuration file changes are:

- **jetVer** specifies the JET version to be audited against.
- **files** specifies the file paths or URLs used to determine the input file set to be audited.
- **exclude** specifies the files paths which should be excluded from the audit.
- **components** specifies paths to folders where user-defined custom web component metadata (`component.json`) can be found. This provides the ability for rules to inspect the component metadata and to validate custom element attributes.
- **theme** specifies the Oracle theme (`redwood`, `alta`, `stable`, or `none`) and affects rules that are theme dependent. In particular, the attribute `label-edge` if not used will have a default value that is dependent on the theme. The Redwood theme is the default if the theme property is not specified, and `stable` is considered a synonym for `redwood`.

Note that you may freeze a JAF configuration and prevent further automatic updates by editing the `oraclejafconfig.json` file and setting the property **update** as follows.

```
"ojet": {
           "update": false,
           ...
        }
```

All other configuration properties remain unchanged, however, you may customize your application audit, by updating the configuration file `oraclejafconfig.json` in a text editor and adding or amending JAF properties. A full description of the configuration options are found in About the Oracle JAF Configuration File Properties.

Additionally, you can obtain a complete list of `ojaf` command line flags by entering the following command.

```
ojaf --help
```

See also Configure the JET Audit Framework and Run Audits on Oracle JET Applications.

Part I

Use the Oracle JET Audit Framework

Run Oracle JAF audits against the application files of your JET project and perform a static analysis of the source code from an Oracle JET perspective.

Topics:

2

Configure the JET Audit Framework

You can modify some basic settings of the default Oracle JAF configuration to set up an audit for your application.

About the Oracle JAF Configuration

Oracle JET Audit Framework (JAF) relies on the configuration file created by the JET tooling when you invoke the JAF initialization command `ojaf --init` in a Command Prompt window on the JET application.

The `oraclejafconfig.json` file that you create when you initialize Oracle JAF the first time defines the properties that you can use to control many aspects of your JET application audit. For example, by configuring the JAF audit, you can perform the following.

- Specify the JET version when you want to use audit rules that are specific to a JET version. This is configured by default as the JET version of the application to be audited.
- Specify the file set when you want to exclude application directories and file types. This is configured by default to include all files of the application to be audited.
- Invoke custom audit rules that are user-defined and assembled as a JAF rule pack for distribution.
- Prevent specific audit rules from running in the audit or limiting the audit to only rules of a certain severity level.
- Include the metadata of Oracle JET Web Components to audit the HTML files of your application's custom components.
- Control the JavaScript/TypeScript source code to audit based on JAF comments that you embed in your source files.
- Work with the output of the audit to customize the presentation of audit messages or to suppress audit messages.

The properties in the `oraclejafconfig.json` file configuration settings are up to you to specify. By doing so, you can fine-tune the audit to focus audit results on only the source that you intend. Multiple configuration files can created for specific runtime criteria or projects. The configuration files are JSON format, but JavaScript style comments are permitted for documentation purposes. The configuration file to be used can be specified on the command-line.

Each time you run the audit from a Command Prompt window, Oracle JAF searches the directory in which you initiated the audit for the JAF configuration file `oraclejafconfig.json`. If no configuration file is found there, then JAF processes only HTML files found in the current directory. In that case, the default JAF configuration settings are used for the audit.

About the Oracle JAF Configuration File Properties

Oracle JET Audit Framework (JAF) configuration file is a JSON format document with properties that define the runtime behavior of the JAF audit.

The `oraclejafconfig.json` file that you create when you initialize JAF the first time contains the properties that you can use to control many aspects of the JET application audit. The file lets you define the following properties.

Configuration Property	Description
rulePacks	Optional. Specifies sets of user-defined rules to include into an audit. The property specifies the zip files or folders for the rules that make up a custom rule pack. For an example, see Audit with Custom Rule Packs.
builtinJetRules	Optional. Default is **true**. If **false**, all the built-in JET rules in the internal rule set with prefix **JET** are disabled. This is a convenience property since it obviates having to specify all the JET rules and individually marking them disabled if only user-defined rules in custom rule packs are to be run. For an example, see Audit with Custom Rule Packs.
builtinJetWcRules	Optional. Default is **false**. If **true**, all the built-in rules in the internal rule set with prefix **JETWC** for audits by custom Web Component authors are enabled. This is a convenience property since it obviates having to specify all the JET Web Component rules and individually marking them enabled. For an example, see Audit JET Custom Web Component Projects.
ruleMods	Optional. Enables/disables rules and can also override or define any rule's options in any rulePack, such as the built-in JET rules in the internal rulePack **JET**. Rules may be enabled/disabled by use of the **enable**/**disable** properties, whose values are the rule names or **group** names. The override options are grouped by the configuration property **rulePacks** sub-property **prefix**. Each entry is a name/value pair, where the property name is a **rule** name and the property value object supplies the option(s) for the rule. See JAF configuration property **rulePacks**. For an example, see Configure Audit Rule Runtime Properties . Note that rules marked with the property **$required** set to `true` cannot be disabled via configuration properties.
severity	Optional. Specifies the rule severity levels to which issues will be restricted. If omitted, the default is **all** (all issues found are reported). The default severity levels accepted are (in descending priority order): **blocker**, **critical**, **major**, **minor**, **info**. (If these have been redefined by using the configuration property **sevMap**, then a redefined severity may also be used here.) For an example, Restrict Audit Rule Severity Level.
sevMap	Optional. Allows reassignment of the severity level of any audit rule message from any rule pack. It can also be used to redefine the set of severity levels used by Oracle JAF when security level identifiers other than the default ones are preferred. For an example, Alter the Severity Level of an Audit Rule .
groups	Optional. Specifies one or more groups that will be used to form the active rule set. If omitted the default is **all**. If this property is used, only the rule groups defined in it are active. Note that this property is mutually exclusive with the JAF configuration property **ruleNames**. For an example, Audit with Specific Rules.
defGroups	Optional. Allows custom groups to be defined and used in the JAF configuration property **groups** definition. For an example, see Audit with Specific Rules.
ruleNames	Optional. Specifies the specific rules and, optionally, rule groups that will form the active rule set. This property is mutually exclusive with JAF configuration property **groups**. If this property is used, only the rules or rule groups defined in it are active. For an example, see Audit with Specific Rules.
base	Optional. Specifies the base directory used to resolve any relative file paths found in the JAF configuration file. If omitted, the location of the configuration file is used. The macro **$jafcwd** may be used, and its value is the directory in which the audit is invoked. For an example, see Configure the Project Scope for the Audit.

Configuration Property	Description
files	Optional. Specifies an array of directory paths, file paths, or URLs used to determine the input file set to be audited. Globs may be specified in a file path. If any file path is relative, it is considered to be relative to the location specified by the JAF configuration property **base**. Note that forward slashes may be used to specify file paths regardless of the platform (Microsoft Windows users can avoid using double backslashes, making the file paths more readable). See also the JAF configuration property **exclude**. For an example, see Configure the Project Scope for the Audit.
exclude	Optional. Specifies an array of file paths which should be excluded from the audit. Globs may be specified. If any file path is relative, it is considered to be relative to the JAF configuration property **base**. Note that forward slashes may be used to specify file paths regardless of the platform (Microsoft Windows users can avoid using double backslashes, making the file paths more readable). For an example, see Configure the Project Scope for the Audit.
jetPagesOnly	Optional. If **true**, this property suppresses auditing of an HTML file if the page does not contain any JET custom elements. Default is **false**. For an example, see Audit Only HTML Files that Contain Oracle JET Components.
theme	Optional. Specifies the Oracle theme - may be **redwood**, **alta**, **stable**, or **none**. If the property is omitted, **redwood** is assumed as the default. Note that **Stable** is treated as a synonym for **Redwood**. The theme setting affects rules that are theme dependent. In particular, the attribute `label-edge` if unspecified has a default value that is dependent on the theme. Refer to the JET API, for description of this attribute in `<oj-input-*>`, `<oj-select-*>`, and `<oj-combobox-*>` elements.
format	Optional. Specifies the type of audit output generated. May be **prose** for standard report style text output, **line** for a flattened-out text style, or **json** for JSON output format. If not specified, the default output is prose style. For an example, see Toggle the Default Format of Audit Messages.
proseFormat	Optional. If **format** has been specified as **prose**, this property defines a custom presentation format (template) for the displayed issues. May use a list of template tokens that may be specified in any order. For examples, see Customize the Presentation of the Audit Messages.
lineFormat	Optional. If JAF configuration property **format** has been specified as **line**, this property defines a custom presentation format (template) for the displayed issue. May use a list of template tokens that may be specified in any order. This property is useful for reconfiguring the output when used in a Microsoft Visual Code terminal window. For an example, see Customize the Presentation of the Audit Messages.
outPath	Optional. Specifies the file path to which the audit output (specified as either prose or JSON) will be written. If the file path is relative, it is considered to be relative to the location specified by the JAF configuration property **base**. If omitted, the output is written to stdout. For an example, see Output Audit Messages in JSON Format.
tabs	Optional. Specifies the tab settings to be used when reporting line/column issues found during an audit. If omitted, the default assumes that each tab character represents 4 spaces. For an example, see Adjust the Tab Value Used to Report Line and Column Issues.
messages	Optional. Controls which messages are reported. It can specify the message IDs to reject, or alternatively those to be accepted only. For an example, see Suppress Audit Messages.

Configuration Property	Description
markupOptions	Optional. This property can be used to modify the reporting of links found in markup files. If **true** is specified (or the option is omitted), **ojaf** looks for URL-like text that has not been marked-up as a link. This can be useful when analyzing `.md` text for compliance. It is possible for false positives to be generated, since text such as `foo.html` or `foo.in` are considered links, and the rule should inspect the link for validity. If **false** is specified, links not formally declared using markup syntax will not be reported as links.
typescript	Optional. This property must be set to true to enable the auditing of TSX files. Otherwise, the files will be skipped. For more information, see Audit JET Web Component Projects Containing VComponents.
addFileList	Optional. If set to **true**, and the JAF configuration property **format** is set to **json**, this property causes the audited file set to appear in an additional **fileset** section (as an array of full file path strings) of the JSON output. This can be useful when creating custom reports from the audit's output JSON, since it allows access to the full file set that was audited. For an example, see Output Audit Messages in JSON Format.
title	Optional. Specifies one or more strings used to customize a title for an audit report. The strings will be displayed when the audit is run with prose format, or the strings will be inserted into JSON output for the audit with JSON format. Macros are available to insert values such as the Oracle JET version, or date and time into the title strings. For an example, see Format a Title for the Audit Report.
jetVer	Optional. Specifies the Oracle JET version to be audited against. Can be a full or partial semantic version (semver) with the format "`[x [.y[. z]]`", such as "8.2.0" (a quoted string). For an example, see Audit with Specific JET and ECMA Script Versions.
components	Optional. Specifies paths to folders where metadata in the `component.json` file of user-defined Oracle JET Web Components can be found. This provides the ability for rules to inspect the Web Component metadata to validate custom element attributes. (Note that this is for user-defined Web Components only; it is not used for Oracle JET HTML components.) For an example, see Audit JET Custom Web Component Usages.
componentOptions	Optional. Specifies controlling options for the associated **components** and **componentUrls** properties. If the neither property is specified, **componentOptions** is ignored. This can be used to disable the application of metadata schema extracted by JAF from `component.json` files and used to validate web components. This may be useful to prevent JAF-INIT messages from being displayed if the schema pass identifies component metadata issues. For an example, see Audit JET Custom Web Component Usages.
nameSpaces	Optional. Specifies the list of namespaces that are not allocated by Oracle and that are user defined to allow JAF to successfully audit user-defined custom Web Components. For an example, see Audit JET Custom Web Component Usages.
stylesets	Optional. Specifies an allowed list of valid user-defined Web Component style names. This permits Oracle JAF to report on invalid style names in HTML and CSS and to distinguish those styles from valid JET core styles. For an example, see Audit CSS Styles and Web Components Styles.
comments	Optional. If **true**, this property enables JavaScript commenting with JAF comment commands when you want to limit audit scope at the level of the source code. Default value is **false**. For an example, see Comment Source Code for Fine-Grained Audit Control.

Configuration Property	Description
followLinks	Optional. If **true**, `<link>` and `<script>` elements in HTML that refer to external stylesheet and JavaScript files are followed, and the files are audited. Specify **false** to prevent externally linked files from being audited. The default value if not specified is **true**. For an example, see Suppress Auditing Linked Content.
ecmaVer	Optional. Specifies the ECMA script version for JavaScript auditing. May be **5**, **6**, **7**, **8**, **9**, **10**, or **11** (a number, without quotes) or the corresponding ES *version year* (for example **2015**, **2016**, **2017**, and so on). Support for ES version **11** (**2020**) requires JAF version 2.9.20 or later. If omitted, the default is ES version **11** (**2020**) as of Oracle JAF version 2.9.48 and ES version **10** (**2019**) is the default starting in Oracle JAF version 2.9.11. For an example, see Audit with Specific JET and ECMA Script Versions.
tempDir	Optional. Specifies the file path to the directory in which Oracle JAF should create its internal work folder (`jaftmp@`). Intermediate directories are created if necessary. If omitted, the default value is the current working directory. This can be used when the default value cannot be used due to permission restrictions on file creation. The path can be absolute or relative. If relative, the path is considered to be relative to the current working directory
ruleDescriptions	Optional. This property takes the following values: **none**, **all**, **short**, or **long**, and causes an additional **descriptions** section to appear in the output JSON. This property applies only if JSON output format has been specified (using JAF configuration property **"format" : "json"**). For an example, see Output Audit Messages in JSON Format.
userDefs	Optional. Specifies a user-defined property (typically an object) that is passed in `context.userDefs` to a fired rule. (This property is not examined nor used by the audit.) The **userDefs** property should be limited to specific, simple cases only. Generally, and especially when multiple non-JET rulepacks are used, run-time data should be maintained in a rulepack extension (see **startupRP** and **closedownRP** in Implement Custom Rules Using the Audit Lifecycle.
options	Optional. Defines miscellaneous runtime options. The property can be used to reduce clutter on the command line when running an audit using the Command Line Interface. For an example, see Audit the Application with Predefined Runtime Options.
extends	Optional. Specify a configuration file whose properties are to be inherited. If the path is relative, it is resolved via the **base** property or the location of the configuration file in which it is specified. A configuration may also extend a JAF standard profile, a list of which can be viewed by using the command `ojaf -prof`. For an example, see Specify Configuration Inheritance.
extendOptions	Optional. Specify one or more properties that should not be inherited from the configuration specified by the property **extends**. For an example, see Specify Configuration Inheritance.
@include()	Optional. This directive allows a configuration file to include text from another file. This can be useful when the application file set to be audited is very large, and permits the audit parameters to be separated from the file list. The directive specifies the path to a text file which is be included in the configuration in place of the directive line. The included files may also use **@include()**. For an example, see Audit CSS Styles and Web Components Styles.

Configure the Project Scope for the Audit

Use the optional Oracle JAF configuration properties **files** and **excludes** to limit audit scope based on the file set to audit. If the properties are omitted, the default file set is based on the JET application configuration and includes by default all HTML files in the root directory.

The **files** property specifies an array of directory paths, file paths, or URLs that the JAF tooling will use to determine the input file set to audit. The **excludes** property specifies an array of file paths which should be excluded from the audit. Here is a basic sample:

```
/*  Comments are supported  */
{
   "files" : [ <path>/*.html, <path_to_specific_file>, ...]  // comments are
supported
}
```

Globs may be specified in a file path to allow filepath expansion and matching using wildcard characters. For example, glob matching in the following sample matches any number of directories at that level in the path hierarchy, and any number of their subdirectories.

```
{
   "files" : ["D:/apps/components/public_html/js/views/**/*.html"]
}
```

Both the **files** and the **excludes** properties can use relative paths. A relative path is considered to be relative to the **base** property if defined, or to the location of the configuration file if the **base** property is not defined. If the **base** property is itself a relative path, it is considered to be relative to the configuration file location.

```
{
   "base"    :  ["./some/filepath"],
   "files"   :  ["./html/*.html"],
   "exclude" :  ["./html/*test[1-9].html"],
 ...
}
```

In this sample, the **files** declaration is considered relative to the **base** property. Here, the **base** property itself is also declared as relative, so the **base** is considered relative to the location of the configuration file. If **base** is omitted, the **files** declarations would be considered to be relative to the configuration file location.

The top-level **base** property can also be specified using the macro value **$jafcwd**. This macro takes on the value of the directory in which the audit was invoked.

To specify the audit scope based on application paths:

1. To set the scope as a list to include, edit the **files** property:

```
   "files" : [
              "./pages/html/*.html",
              "./mobile/html/*.html",
              "./pages/static/**",
```

```
          "http://server:1234/test/app.html
        ]
```

Windows users might prefer to use the forward slash file separator as shown to avoid having to escape the backslashes (by using double backslashes). Glob matching is supported by wildcards *, **, ? and [].

> **Note:**
>
> When a complex specification of globs is used in the **files** and **excludes** properties of the JAF configuration, it is useful to be able to verify the files that will be audited. JAF offers the ability to be able to perform a dry-run without actually invoking the audit rules selected by the configuration. On the command line, specify the flag **--dryrun**. This will verify that the configuration file contains no errors, and will then display only the file paths to the files that would have been audited, thus obviating the need to inspect the full output.

2. To set the scope as a list to include with reference to a base folder, edit the **files** property.

```
"files" : [
            "someFilePath",
            "someFilePath2",
            {
              "base" : "D:/git/trunk/built/apps/components/public_html/
demo",
              "files" : [
                          "**/recipe.html",
                          "/**/description.html",
                          "/**/customHeader.html",
                          "demo-dataVisualizations-filtering_element.html",
                          "demo-dataVisualizations-
highlighting_element.html",
                          "demo-accordion-basicAccordion.html",
                          "demo-accordion-events.html",
                          "demo-accordion-multiExpandAccordion.html",
                          . . .
                        ]
            },

... // multiple file objects can be specified

          ]
```

For convenience and clarity where a large number of long file paths need to be specified, this alternative format is available. This allows an object definition to be used in place of a simple string file path and defines a base folder for use with the set of files defined in the object.

3. To set the scope as a list to exclude, edit the **files** and **excludes** properties.

```
{
   "files" :    [
                  "D:/git/trunk/built/apps/components/public_html/js/
```

```
views/**/*.html",
                    ...
                ],
   "excludes" : [
                  "D:/git/trunk/built/apps/components/public_html/js/
views/**/Test[1-9].html",
                    ...
                ]
}
```

Sometimes it is more convenient to use generic globs without extremely detailed regular expressions, and then exclude the exceptions. This can be achieved using the **excludes** property as the sample shows.

> **Note:**
>
> Specification of files that should not be audited may cause false positive issues to be reported by JAF. For example, *angular* file snippets might be construed with the JET Custom Web Component attribute expression syntax. The file set should be tailored to meet the needs of a particular audit. Generally, if you have files that need to be run through a pre-processor, you should not submit these raw files for audit. You may check the actual runtime file set generated by a JAF configuration by using the **--dryrun** command line flag.

Specify Configuration Inheritance

Use the **extends** configuration property to allow parent configurations to inherit properties and property values from child configurations, and use the **extendOptions** property to define the actions taken during inheritance.

When multiple configurations need to be maintained by an organization that has different auditing requirements, some aspects of the configurations may be common and need to be repeated. To avoid duplication and related synchronization issues, JAF permits a hierarchical structure to be used, where one configuration can inherit properties or property values from another. This is performed by use of the **extends** property.

```
Primary config_1
         --------------
         {    "extends" : "/path/to/config_2.json"
               . . .
         }

         Child config_2
          -------------
          {    "extends" : "/path/to/config_3.json"
                . . .
          }

          Child config_3
          -------------
          {    . . .   // no "extends"
          }
```

In the above example, the final active configuration is a modified copy of the primary configuration, **config_1**, and is the result of a merge of the child configurations and the primary configuration. The merge sequence starts at the last child, **config_3**, and moves upwards to **config_1**. The sequence is as follows:

1. **config_3** is a child configuration and is thus merged into its direct parent, **config_2**.
2. The result of that merge is a child configuration to its direct parent, **config_1** (the primary configuration), and is merged into **config_1**.
3. A copy of the updated **config_1** becomes the final active configuration for the audit.

Note that circular references are not permitted and are detected by the command **ojaf**, resulting in termination of the audit.

If a configuration uses relative file references, these are resolved during the merge with its direct parent. This resolution is performed in the standard JAF documented sequence:

1. If no **base** property is declared, the relative paths are considered relative to the location of their containing configuration file.
2. If a **base** property is declared, the relative paths are considered relative to the **base** property path.
3. If the **base** property is itself relative, the **base** property is considered relative to the location of the containing configuration file.

When two configurations are merged, a parent property either takes precedence over the child property, or the child property values are merged into the parent property.

1. If the child property does not exist in the parent configuration, the property is transferred directly to the parent, unless this is overridden by the optional property **extendOptions**.
2. If the parent property exists, but the child property cannot be merged or is a single value (such as **severity**, **theme**, and **jetver**), the parent property takes precedence.
3. During an inheritance merge, the **base** and **extends** properties of a child are not inherited by the parent.
4. If a property is considered inheritable, the action taken depends on the specific property. As illustrated in the following example, the **options** property result is a direct merge, assuming that **Config2** extends **Config1**:

```
Active (final) Config    <---     Config2              <---        Config1
--------------------              ------------------
--------------------

"options" : {                      options" :
{                       "options" : {
              "msgid" : true,                    "msgid" :
true,                     "msgid" : true,
              "color" : false,                   "color" :
false                     "color" : true,
              "verbose" :
true                                                        "verbose" : true
            }                                  }
 }
```

Optionally, the action taken during inheritance can be controlled by the property **extendOptions**. This property is examined only for a parent configuration and is never

inherited. **extendOptions** declares the action that applies to each property in the direct child that it refers to.

```
"extendOptions" : {
  <property>  :  "ignore" ,
   . . .
}
```

A property that is declared in a child configuration can be prevented from propagating into the parent configuration. For example, a parent configuration can specify that the child configuration property **ruleDescriptions** should be ignored.

```
"extendOptions" : {
  "ruleDescriptions"  :  "ignore" ,
   . . .
}
```

The configuration properties **files** and **exclude** together define the scope of the fileset that will be audited. As noted in the table below, these properties are inheritable and are merged by default (unless vetoed by **extendOptions**). The action taken for these two properties is as follows:

1. The parent and child **files** and **exclude** properties are examined for relative paths, which are resolved.
2. If a **files** property contains **files** or **base** objects, they are expanded and resolved.
3. The resulting flattened/resolved arrays (with duplicates removed) are merged, and the result is their union.

Inherited Properties Merged By Default	Properties Never Inherited or Propagated
components, componentUrls, exclude, files, groups, messages, nameSpaces, options, ruleMods, ruleNames, rulePacks, sevMap	base, extends, tokens

Use the command-line flag `-dac` (display active configuration) to evaluate the result of inheritance without actually running the audit life-cycle.

Inheritance from built-in *standard profiles* is available when using the **extends** property while setting up a configuration. JAF provides some standard profiles to enable/disable standard sets of rules and, in some cases, elevate the severity of errors. You can view the available profiles by using the `-prof` command option (see Audit the Application Using the Command Line).

```
$ ojaf -prof

--- JAF Profiles ---
* best-practice
   JAF Standard profile configuration enforcing best practice for JET
Application Development.
* redwood-strict (extends 'best-practice')
   JAF Standard profile configuration for strict audit checking of Redwood
applications.
```

To use one of these profiles as a base for your own configuration, use the **extends** option but specify the *name* of the profile required rather than a *path* to a specific configuration JSON file.

```
{
  "extends": {"profile":"redwood-strict"}
  "jetVer": "11.0.0",
  "ecmaVer":"2019",
  "base": "$jafcwd",
  "files": [
     "./src/**/*.html",
     "./src/**/*.js",
     "./src/**/*.css",
     "./src/**/component.json"
  ],
  ...
}
```

For example, in the case of the profile **redwood-strict** above, the severity of rule `oj-html-alta-deprecated` is upgraded to **blocker** because a Redwood application should not use the Alta theme. You can view specific profiles by using `ojaf -prof <profile name>` (e.g.,, `ojaf -prof best-practice`). To view the full set of rules that are enabled and their severity, you can use the `-dac` command option to display the final active configuration that has been evaluated.

Configure Audit Rule Runtime Properties

Use the Oracle JAF configuration property **ruleMods** to change the auditing behavior of any audit rule.

You can customize the audit behavior of individual audit rules by using the **ruleMods** property to enable/disable rules and to override or define the auditing options defined by the rule pack to which the rule belongs. Each entry in the **ruleMods** property that you define is a name/value pair, where the property name is a rule name and the property value object supplies the option(s) for the rule. The override options are grouped by the **rulePacks** property **prefix** (**ABCD** in the following example). In the case of the built-in JET rule pack, which is enabled by default, specify the rule pack prefix **JET**.

```
"ruleMods": {
  "ABCD": {
      "my-rule": {
          "pageType": "index",
          "companyCode": "acd",
          // user rule with arbitrary runtime properties
          "isMulti": true, . . .
        }
  }
}
```

Each rule name/value that you define is merged over the properties defined for the rule in its respective rule pack and overrides the run-time rule behavior. Any rule option may be set. For more information, see Reference the Custom Audit Rules in an Audit

The properties that you can customize for individual rules are shown in the table.

Rule Property	Description
enabled	Enables or disables the audit rule. All rules are enabled by default. See example below.
severity	Classifies the severity level of the audit rule. You may change the level to **info**, **minor**, **major**, **critical** (default), **blocker** when you want to restrict the audit by rule severity level. Note that these severity levels may be replaced by user-defined severity levels as required by your organization. See also Alter the Severity Level of an Audit Rule .
status	Associates a development status with the audit rule. You may change the status to **production**, **alpha**, **beta**, or **deprecated**.
filetype	Specify the file types for which the audit rule will be invoked. You may change the file type to **html** and/or **css**, and/or **js** and/or **json**. For example: `"filetype : "html"` or `"filetype" : ["html", "css"]` The **filetype** property is ignored by hook rules declared for **startup**/ **closedown** phases, since these are not file related.
group	Specify the group or groups to which the audit rule is assigned. You may change the rule group when you want to restrict the audit by rule group. For example: `"group" : "jet-html"` or `"group" : [ "jet-html", "jet-aria"]` See also Audit with Specific Rules.
jetver	Specifies the Oracle JET release version or versions to be audited against. You may change the JET release version required to invoke the rule. The format supports semantic versioning, as used in programs like **npm**. For example: `"jetver" : ">=7.1.0"` or `"jetver" : "~7.1.0"` For more information about this property and semantic versioning, see Audit with Specific JET and ECMA Script Versions.

Rule Property	Description
issueTag	Defines a string which is passed through to the output JSON, or custom **proseFormat** / **lineFormat**, whenever the rule to which it is applied fires an audit issue. The string may be applied to any rulepack and rule. It is not inspected by JAF, and could, for example be formatted as a colon-separated key:value pair. When the rule to which it is applied fires, the **issueTag** property is passed through to the output. For JSON formatted output, it appears as Issue property **issueTag**, and similarly in the Issue object for API/AMD modes. For CLI default output, the **issueTag** string is not displayed, unless a custom format is defined via the configuration **proseFormat** or **lineFormat** properties. In this case the **issueTag** string is available as substitution `%symbol %itag`. `. . .` `"format": "prose",` `"proseFormat": "%itag",` `. . .` `"ruleMods": {` `"JET": {` `"oj-ts-noconsole": {` `"issueTag": "Do not include console statements"` `}` `}` `. . .`

Rules may be enabled/disabled by using the **enable**/**disable** properties.

```
"ruleMods" : {
  // can specify rule names and group names
  "enable" :   [ "rulename1", "rulename2, "groupname1"],
  "disable" :  [ "rulename3", "rulename4, "groupname2" ]
}
```

Note that since a **group** name represents a set of rules, **group** names can be declared in addition to rule names.

If both **enable** and **disable** properties are declared, the **enable** set is processed first, followed by the **disable** set. If a conflict is found, a notification message is generated and the run abandoned.

To set rule options:
Configurable rules inspect the rule options at start-up time and configure themselves accordingly. Redefine rule options for these rules, such as their severity level and enabled status, by editing the `oraclejafconfig.json` file **ruleMods** property. Each rule option entry is a name/value pair, where the property name is a rule name and the property value object supplies the option(s) for the rule. For instance, to set a desired value for the **severity** sub-property and **enabled** sub-property:

```
"ruleMods" : {
  "JET" : {
    "oj-html-ojattr" : {"severity": "critical"},
```

```
    "oj-html-lib" : {"enabled": "false"}
  }
}
```

This example illustrates that any rule option can be set, where the built-in rule **oj-html-ojattr** has been overridden as severity level **critical** and the rule **oj-html-lib** has been disabled. However, it is recommended that those particular options are set through the configuration **sevMap** and **ruleMods enable**/**disable** properties.

3

Run Audits on Oracle JET Applications

You can run an audit without making changes to the default Oracle JAF configuration or you can customize a variety of configuration properties to change audit behavior.

Audit the Application Using the Command Line

If you have configured Oracle JAF using the `ojaf --init` command, you can run an audit in the Command Prompt window.

In the JET application, open a Command Prompt window and from the root directory, use the JAF command without any arguments to run an audit based on the current JAF configuration for the entire application, against all HTML, JS, CSS, and JSON files.

```
ojaf
```

The full command line syntax is, with optional flags and an optional space-separated list of directory and/or file path arguments:

```
ojaf [ <flags> ] [ <files>]
```

The audit runs starting at the directory where you invoke the command in the Command-Line Interface. To audit a subset of the application files, change to the desired starting directory and invoke the audit command. Alternatively, for a simple audit, you can specify the files and file paths on the command line, relative to the current working directory:

```
ojaf myFile1[, myFile2, myFilex]
```

For example, you can audit two files in the current working directory and also audit all files in the subfolders of only the test directory like this:

```
ojaf file1 file2 ./test/**/*
```

Note that some platforms perform wildcard expansion of command line arguments, so you may need to quote `ojaf` argument with wildcard characters to prevent this.

The `ojaf` command accepts command line flags that you can use to override corresponding property settings in the JAF configuration file. For example, you can append the `--format` flag when you want to override the default flattened-out text output and display standard report style output for the audit.

```
ojaf --format prose
```

Other command line flags allow you to interact with the `ojaf` utility to return desired information. For example, if you want to confirm the Oracle JET version of your application is among the current versions supported by JAF, append the `--jetlist` flag.

```
ojaf --jetlist
```

You might also use the command line to get more details about a particular issue that your audit reported. This is supported by appending the `--help` flag and argument that provides either the ID of the message or the name of the rule that emitted the message.

```
ojaf --help msgID
```

The complete list of command line flags can be displayed using the `--help` flag without arguments.

```
ojaf --help
```

The `--help` command displays the command line flag options for the `ojaf` utility, as described in the table below. Note that Oracle JAF command line flags are case insensitive.

You can also use the `--help` flag to display the manual page (manpage) documentation for a command line flag by appending the flag as an argument to `--help` or `-h`, (such as `ojaf -h dr`).

Alternatively, you can display the manpage for a flag by ending the command with a `?` (such as `ojaf -rc?`). Note that for *nix shells, the question mark should be escaped or the argument enclosed in quotes (`ojaf -rc\?` or `ojaf "-rc?"`).

OJAF Command Flag (long/short)	Description
`--help` or `-h`	Displays the usage of the command line flags described in this table.
`--help` *`msgId`* | *`rulename`* | *`command`* or `-h` *`msgId`* | *`rulename`* | *`command`*	Displays the description for a rule identified by its message ID or name, or displays the manpage for a command. For example, the following displays a description for the JAF audit message ID `JET-0160`: `ojaf -h JET-0160` The following displays the description for the JAF audit rule `oj-html-binding-attr`: `ojaf -h oj-html-binding-attr` The following displays the description for the JAF command line flag `--retcode`: `ojaf -h -rc`
`--<command>?`	Displays the manpage for a command, as an alternative to using `-h <command>`. For example, `ojaf -h dr` and `ojaf -dr?` both bring up the manpage for the command option `--dryrun`. Note that for *nix shells, the question mark should be escaped or the argument enclosed in quotes (`ojaf -rc\?` or `ojaf "-rc?"`) to avoid command line expansion.

OJAF Command Flag (long/short)	Description
`--init` or `-i`	Scaffold the `oraclejafconfig.json` file in the current directory. The configuration file is required to run an audit and can run with default settings or the settings can be changed to customize audit behavior.
`--initRule` *`rulename`* or `-ir` *`rulename`*	Scaffolds a skeleton custom audit rule implementation file in the current directory. The argument is followed by the rule name without the `.js` file extension. For example, the following creates a file `my_new_rule.js` in the current directory: `ojaf --ir my-new-rule` For details about creating user-defined, custom audit rules, see Set up the Custom Audit Rules Test Project.
`--config` *`filepath`* or `-c` *`filepath`*	Runs the audit with a specified configuration when followed by a file path to a configuration file. If omitted, the current directory is searched for the `oraclejafconfig.json` file.
`--jetver` *`semver`* or `-jv` *`semver`*	Runs the audit for a JET release version when followed by a partial or full semantic version value representing the JET release version. Use to match the Oracle JAF rule set to the version of your application. Accepts a partial semantic version definition, such that *`semver`* `9` will be promoted by JAF to the latest available release: 9.*x*.*y*. Similarly, *`semver`* `9.1` will be promoted by JAF to the highest patch level 9.1.*y*. For more semantic value examples, see Audit with Specific JET and ECMA Script Versions. **Tip:** Use the command `ojaf --jetlist` (or `ojaf -jl`) to display the current JET release versions supported by the `ojaf` Command-Line utility. If omitted, the rule set is based on the version specified by the **jetVer** configuration property in the `oraclejafconfig.json` file.
`--groups` *`ruleGroupName1 ruleGroupName2 ...`* or `-g` *`ruleGroupName1 ruleGroupName2 ...`*	Runs the audit with the specified rule set groups. For the list of built-in rule set groups, see Audit with Specific Rules.
`--format prose | line | json` or `-t prose | line | json`	Formats audit output when followed by the output display format: `prose` for standard report style text output, `line` for flattened-out text output, or `json` for JSON format.
`--outPath` *`filepath`* or `-o` *`filepath`*	Writes the output of the audit to a specified directory when followed by a file path. If the path is relative, the output directory is resolved relative to the current directory. If omitted, the audit output directory is the current directory.

OJAF Command Flag (long/short)	Description
`--noout` or `-no`	Suppresses output to the file specified in the configuration file by the **outPath** configuration property, and causes the audit output to be directed to the console.
`--severity [ > | >= | < | <= ] blocker | critical | major | minor | info | all (default)` or `-s [ > | >= | < | <= ] blocker | critical | major | minor | info | all (default)` **Note:** If comparative operators are used, the severity condition *must* be enclosed by quotes.	Restricts the severity of the issues reported by the audit when followed by a severity level. For severity level descriptions, see Restrict Audit Rule Severity Level. The optional comparative operators >, >=, <, and <= may precede the severity level (where the condition must be enclosed by quotes). For example, the following will display issues of severity "critical" and "blocker" level. `--severity ">=critical"` If omitted, the audit reports issues of all severity levels. If these have been redefined by using the configuration property **sevMap**, then a redefined severity may also be used here.
`--followlinks` or `-fl`	Audit follows the stylesheet `<link>` elements and JAF will audit linked-to stylesheets and referenced JavaScript/TypeScript script files. If omitted, the audit follows the linked-to stylesheets by default.
`--nofollowlinks` or `-nfl`	Audit does not follow the stylesheet `<link>` elements and JAF will not audit linked-to stylesheets or referenced JavaScript/TypeScript script files.
`--extra` or `-e`	Shows extra details in the output. For example, it includes the rule that was used to create a particular issue, and the issue message ID.
`--msgid` or `-id`	When prose mode is configured, (see `--format`), the message ID of reported issues is appended to the displayed issue text. A custom format defined by the JAF configuration property **proseFormat** overrides this flag.
`--dryrun` or `-dr`	Performs a full start-up and analysis of the configuration file. Displays the files that would have been audited, but does not fire the rules on the file set (as determined by the JAF configuration property settings **files** and **exclude**). This is useful when you want to verify that the configuration file contains no errors and particularly useful when the settings for **files** and **exclude** properties are complex.
`--jetlist` or `-jl`	Displays the current JAF supported versions of JET. This is useful when you want to verify that the version of JET used to create your application is among the versions supported by the current JAF installation.
`--profiles` or `-prof [ profile name]`	Displays the available configuration profiles that can be inherited. If the command flag is followed by an optional profile name, the profile is displayed.

OJAF Command Flag (long/short)	Description
`--nslist` or `-nsl`	Displays the namespaces known to JAF. This is useful when you want to audit the file set for references to user-defined custom Web Components and need to suppress false audit reporting of valid Web Component references. For details, see Audit JET Custom Web Component Usages.
`--dslist` or `-dl`	Displays a list of the rules disabled by default in the JET built-in rule packs. May be immediately followed by an optional pack prefix to limit the output to the pack declared.
`--deplist` or `-dpl`	Displays a list of the rules that are deprecated. May be immediately followed by an optional pack prefix to limit the output to the pack declared.
`--grouplist` or `-gl`	Displays the built-in JET rule pack groups and their associated rules. May be immediately followed by an optional pack prefix name to filter the output to the specified pack only.
`--xgrouplist` or `-xgl`	Displays external (non-JET) rule pack groups and their associated rules. May be immediately followed by an optional pack prefix name to filter the output to the specified pack only. Note that this command requires a configuration file with the **rulePacks** property set (that is, use of `-c` on the command line). OJAF only refers to the **rulePacks** property, so no other properties need be present.
`--amdlist` or `-amd`	Displays the availability/unavailability of rules in AMD mode in JAF built-in rulepacks. An optional rulepack prefix may be specified to restrict output to just the specified pack.
`--loadorder` or `-rlo`	Displays the loading order of rules in JAF built-in rulepacks. An optional rulepack prefix may be specified to restrict output to just the specified pack.
`--betalist` or `-bl`	Displays the status of rules in JAF built-in rulepacks. An optional rulepack prefix may be specified to restrict output to just the specified pack.
`--metahist` or `-mh`	Displays the metadata history across all JET versions known to JAF, for deleted and renamed classes, and deleted class methods and members. This is useful for diagnosing problems resulting from deprecation or name changes. Note JAF reports these in messages JET-3070 and JET-3071.
`--dac` or `-dac`	Displays the active configuration file to evaluate the result of inheritance. No audit is performed.

OJAF Command Flag (long/short)	Description
`--deflist` or `--default` or `-def`	Displays the current JAF configuration default values for selected properties.
`--debug` or `-d`	Enables debug mode to allow very verbose output. This is useful for diagnosing problems.
`--rules` or `-r`	Displays the active rules information and their options as text, but no audit is performed. Can be used in combination with the `--outPath` flag to specify the file path to which rule data will be written. If omitted, rule data is written to the console.
`--rulesjson` or `-rj`	Displays the active rules information and their options as JSON, but no audit is performed. Can be used in combination with the `--outPath` flag to specify the file path to which rule data will be written. If omitted, rule data is written to the console.
`--rulessonar` or `-rs`	Displays the active rules information and their options as XML (SONAR format), but no audit is performed. Can be used in combination with the `--outPath` flag to specify the file path to which rule data will be written. If omitted, rule data is written to the console.
`--retcode auto (default) | errors | x (a number)` or `-rc auto (default) | errors | x (a number)`	Overrides the return code behavior of the OJAF command-line interface. Specify **auto** for the default behavior. Specify **errors** if the return code should be 0, except in the case of abnormal termination, where -1 will still be returned as in the default mode. Specify a number that will be used as the return code. Requires OJAF version 2.9.11 or later.
`--help` *`msgId | rulename`* or `-h` *`msgId | rulename`*	Displays the description for a rule identified by its message ID or name. For example, the following displays a description for the JAF audit message ID `JET-0160`: `ojaf -h JET-0160` The following displays the description for a JAF audit rule `oj-html-binding-attr`: `ojaf -h oj-html-binding-attr`
`--initRule` *`rulename`* or `-ir` *`rulename`*	Scaffolds a skeleton custom audit rule implementation file in the current directory. The argument is followed by the rule name without the `.js` file extension. For example, the following creates a file `my_new_rule.js` in the current directory: `ojaf --ir my-new-rule` For details about creating user-defined, custom audit rules, see Set up the Custom Audit Rules Test Project.

Audit the Application with Predefined Runtime Options

Use the optional JAF configuration property **options** to define runtime options to use each time you run the audit from the command line.

Instead of invoking the `ojaf` command with command line flags, you can pre-define various runtime options by configuring the corresponding settings in the JAF configuration file **options** property. For example, instead of appending `--extra --nocolor` to an `ojaf` command invocation each time, you can define the configuration options **debug** and **color**.

To configure miscellaneous runtime options:

- To configure runtime options to apply to `ojaf` invocations, edit the `oraclejafconfig.json` file **options** property.

```
"options": {
    "debug": true | false, // Set debug mode.
    "verbose": true | false, // Set verbose mode for additional output.
    "color": true | false, // Set color mode for messages and prose
output. (Works best on a black background.)
    "msgid": true | false, // Add message IDs to displayed issues.
(Requires 'format' : 'prose' | 'line'.)
    "ruleName": true | false // Add rule names and msgId's to displayed
issues. (Requires 'format' : 'prose' | 'line'.)
    "retCode" | "rc": "auto" | "errors" | number // Override OJAF CLI
return code. Default if omitted is "auto". See examples below.
}
```

The command line equivalents are as follows.

```
"options": {
    "debug": false, // Same as command line  -d   or -- debug
    "verbose": false, // Same as command line  -e   or --extra
    "color": false, // Same as command line  -nc  or --nocolor
    "msgid": true // Same as command line  -id  or --msgid (Requires
'format' : 'prose' | 'line'.)
}
```

Examples of the configuration option to override the return code behavior in the OJAF command-line interface. Requires OJAF version 2.9.11 or later. Note that `"rc"` and `"retcode"` are synonyms.

```
"options": {
    "rc": 0 // return code will always be zero
}
```

or

```
"options": {
    "rc": "errors
    // return code will always be zero, except
```

```
    // in the event of early/abnormal
}
```

or

```
"options": {
    "rc": "auto"
    // default behavior - 0 - nnn -> number of issues found,
    // -1 for early termination
}
```

Audit with Specific JET and ECMA Script Versions

During an audit, Oracle JAF refers to application-specific metadata to process the JAF rule set. Use the optional JAF configuration file **jetVer** property to set the metadata for a JET version and use the optional **ecmaVer** property to set the ECMA version. If the properties are omitted, then JAF processes the rule set by using metadata that is specific to the version of JET used to create the application.

The JAF built-in audit rule set works against metadata that is optimized for a specific version of JET and a specific JavaScript ECMA implementation. By default, JAF derives the rules metadata from the JET JavaScript API reference documentation that is specific to your application. However, you can customize the audit to process the built-in rule set for versions of JET and JavaScript/TypeScript other than your application.

- Oracle JET version "5.2.0" (a quoted string) and later are valid settings for JET metadata. Note, if you happen to need to run with a JET release candidate, trailing characters are permitted, such as "9.0.0-rc3".
- For the JavaScript ECMA, versions 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 (a number, without quotes) are valid and may be specified by the ECMA version number or ES version year, such as 2023 for ES14.

To change the application-specific metadata to use for an audit:

1. To set the metadata for an Oracle JET release version by specifying a semantic version value, edit the `oraclejafconfig.json` file **jetVer** property.

```
{
   ...
   "jetVer" :  "9.0.0"
   ...
}
```

   Must be a quoted string of JET release version "5.2.0" or later. The specification is treated by JAF as a semantic version value, where partial version specifications are valid. Generally speaking, only the *major* and *minor* release values are required (for example, "9" or "8.2"), since the *patch* number will be supplied by JAF. JAF promotes the specified **jetVer** value to the *latest* available JET release that corresponds to the semantic version (refer to the output of command `ojaf -jl` for the current default for the version of JAF you are using). For example, assume that JET release versions 8.1.1, 8.1.2, 8.2.1, and 8.2.2 are supported, then JAF promotes semantic version values for **jetVer** as follows.

jetVer Value	Promoted To JET Release
"8"	8.2.2
"8.1"	8.1.2
"8.2"	8.2.2
"8.2.0"	8.2.2

> **Note:**
>
> For more information about semantic versioning, visit the SemVer org website at https://semver.org.

2. To set the metadata for a specific JavaScript implementation, edit the `oraclejafconfig.json` file **ecmaVer** property.

```
{
   ...
   "ecmaVer" :  11
   ...
}
```

A version number with or without quotes in the version range `5` to `14` (`5`, `6`, and so on), or the year range `2015` to `2023` (`2015`, `2016`, and so on).

Starting in JAF version 2.9.15, the ECMA version can also be specified as an ES version year with or without quotes, such as 2020, 2019, and so on.

Official Name / Version	JAF ecmaVer Property	JAF Version Required
ES2023 / ES14	**14** or **"14"** or **2023** or **"2023"**	6.9.0 or later
ES2022 / ES13	**13** or **"13"** or **2022** or **"2022"**	5.9.0 or later
ES2021 / ES12	**12** or **"12"** or **2021** or **"2021"**	5.9.0 or later
ES2020 / ES11	**11** or **"11"** or **2020** or **"2020"**	2.9.20 or later
ES2019 / ES10	**10** or **"10"** or **2019** or **"2019"**	2.9.11 or later
ES2018 / ES9	**9** or **"9"** or **2018** or **"2018"**	2.8.21 or later
ES2017 / ES8	**8** or **"8"** or **2017** or "**2017"**	2.8.21 or later
ES2016 / ES7	**7** or "**7"** or **2016** or **"2016"**	2.8.21 or later
ES2015 / ES6	**6** or **"6"** or **2015** or **"2015"**	2.8.20 or earlier
na / ES5	**5** or **"5"**	na

Audit with Specific Rules

The audit rule set invoked by JAF can be optionally conditioned by the use of the configuration properties **groups**, **ruleNames**, and **ruleMods**. These properties provide different ways to enable and disable rules in the runtime rule set to specify a list of rules that will be run, and thus indirectly disable all other rules.

All JAF rules are defined in rule packs. Some rule packs are built-in to JAF and can simply be selected by setting the configuration properties starting with **builtin**. For example, **builtinJetRules** can be set to **true**, and all (enabled) rules defined in it now become available to JAF. (Note that **builtinJetRules** is **true** by default if omitted, and thus nothing needs to be declared to perform a standard audit using the JET built-in rules.) For rule packs that are not

among the JET built-in rule packs, the configuration property **rulePacks** can be used to declare which external rule packs are to be loaded.

So, by default, the built-in Oracle JAF rule pack is enabled and, optionally, external rule packs may be loaded, and all rules within these rule packs are enabled to run in an audit. It is possible to disable rules in the rule pack to prevent the rule from running, but if the requirement is to run only a few rules, you may find it easier to state which rules are to be run rather than disable all those not required.

Use the JAF configuration property **ruleNames** to specify a list of specific rules and rule groups to run, and thus indirectly disable all other rules. Alternatively, use the property **groups** to specify a list of rule groups to run, and thus indirectly disable all other rule groups. Only the named rules or rule groups will be invoked in the JAF audit.

Note that the **groups** and **ruleNames** configuration properties alone do not enable rules, they only declare rules that you intend to use. Whether these rules are run or not may depend on the audit configuration. If, for example, a group is specified, but its containing rule pack is not loaded, the group's rules will not be run. Similarly, if a rule that's declared in **ruleNames** is disabled by default (or is disabled elsewhere via the **ruleMods** property), the rule will not run.

Thus, the **ruleNames** and **groups** property give you a convenient way to temporarily override an existing configuration of loaded rule packs without resorting to editing or commenting out configuration entries. This is particularly useful during rule development or debugging. These properties are also useful sometimes to state the rules to be run, rather than disable the rules you don't want run.

To audit with specific rules:

1. To declare subsets of the rules to run, edit the `oraclejafconfig.json` file **groups** property.

   ```
   { "groups" : ["ruleGroup1", "ruleGroup2", ...],
   }
   ```

   In the case of the JET built-in rules, you can specify the rules to run by declaring the group name for the built-in JET rule pack subset as described below. For example, where the group name **jet-html** defines the HTML rules and the group name **jet-js** defines the JavaScript/TypeScript rules, then the sample enables only the rules belonging to the group **jet-html**.

   ```
   { "groups" : ["jet-html"],
   }
   ```

Built-in Rule Group Name	Usage
jet-html	Rules check HTML.
jet-css	Rules check CSS.
jet-js	Rules check JavaScript.
jet-ts	Rules check TypeScript.
jet-tsx	Rules check TypeScript with JSX.
jet-md	Rules that check Markdown.
jet-json	Rules check JSON.
jet-override	Rules that check for CSS overrides.
jet-perf	Rules that check performance.
jet-aria	Rule check accessibility compliance.

Built-in Rule Group Name	Usage
jet-deprecated	Rules check deprecated status.
jet-deleted	Rules checking deleted status.
jet-bp	Rules enforce JET development best practices.
jet-redwood-bp	Rules that enforce best practices for Redwood.
jet-csp	Rules check for Content Security Policy (CSP) violations.
html5	Rules check for obsolete tags and attributes for HTML5.
jet-cca	Enforces rules defined by Web Component metadata whether they exist in the application or on Oracle Component Exchange.

2. Optionally, to declare custom rule groups to run, edit the `oraclejafconfig.json` file **defGroups** property and add the custom group name to the **groups** property.

```
"groups": [
    "pages"
] // references a custom group

"defGroups": [
    {
        "pages": [
            "jet-html",
            "jet-css"
        ]
    },
                ...
]
```

In the sample, the custom group **pages** is defined in the **defGroups** property and enabled for auditing in the **groups** property.

3. To declare a list of specific rules and rule groups to run, edit the `oraclejafconfig.json` file **ruleNames** property.

```
{ "ruleNames" : ["ruleName1", "ruleName2, "ruleGroup1", ...],
}
```

A rule or rule group name may be declared using a regular expression string that includes wildcard characters (requires JAF version 2.9.29 or later).

```
{ "ruleNames" : ["oj-ruleName-*", ...],
}
```

The **ruleNames** property and the **groups** property are mutually exclusive. If the **ruleNames** property is configured, only the rules and rule groups defined in it are active.

4. To enable or disable specific rules and rule groups, edit the `oraclejafconfig.json` file **ruleMods** property and set the desired list of rules and rule groups on the **enable** and **disable** sub-properties.

```
"ruleMods": {
    "enable": [
        "rule1",
```

```
            "rule2",
            "rulegroup1"
        ],
        "disable": [
            "rule3",
            "rule4",
            "rulegroup2"
        ]
    }
```

Note that because a rule group name represents a set of rules, the group name can be declared. If both **enable** and **disable** properties are declared, the **enable** set is processed first, followed by the **disable** set.

Alternatively, if you are overriding rule options with **ruleMods**, then you can specify the **enabled** property of the desired rule as **true** or **false** on the rule definition. In this example, the rule is disabled.

```
"ruleMods": {
    "JET": {
        "oj-html-ko-databind": {
            "enabled": false
        },
    ...
    }
}
```

Audit with Custom Rule Packs

Use the Oracle JAF configuration property **rulePacks** to include sets of user-defined, custom audit rules in an Oracle JAF audit.

You can optionally enable sets of user-defined, custom audit rules by using the **rulePacks** property. The property specifies the zip file or folder containing the rules. For details about how to use Oracle JAF to create user-defined rules, see Extend the Oracle JET Audit Framework.

```
"rulePacks": [
    {
        "path": "path/to/myrulepack.zip",
        "enabled": [
            true (default) | false
        ]
                "status": [
            "all" (default),
            "production",
            "deprecated",
            "beta",
            "alpha"
        ]
    },
    {
        "path": "path/to/my/rulepack/folder",
        "enabled": [
            true (default) | false
        ]
```

```
        "status": [
            "all" (default),
            "production",
            "deprecated",
            "beta",
            "alpha"
        ]
    },
   ...
]
```

The **enabled** property is optional and provides the ability to easily disable a complete rule pack. If omitted, the default is enabled.

The specified **path** can be relative. If relative, it is considered to be relative to the location of the configuration file, or the configuration file's **base** property, if defined.

To audit with specific rule packs:

1. To enable custom rule packs to run, edit the `oraclejafconfig.json` file **rulePacks** property, specify **path** as location of the rule pack and set the desired value for the optional **status** sub-property.

   ```
   "rulePacks": [
       {
           "path": "path/to/myrulepack1.zip",
       },
       {
           "path": "path/to/myrulepack2.zip",
           "status": [
               "deprecated",
               "beta"
           ]
       }
   ]
   ```

   The sample enables the audit to run with custom rule packs specified by the **path** properties. The **enabled** property is by default true and can be omitted. The **status** property for `myrulepack2.zip` in this sample limits the audit to load only rules with the system property **status** defined as **deprecated** or **beta** in the rules declaration file `rules.json`. For more information about system properties defined by a custom rule pack that you may override, see Configure Audit Rule Runtime Properties .

2. To disable a custom rule pack from running, edit the `oraclejafconfig.json` file **rulePacks** property, specify **path** as the location of the rule pack and set the value **false** for the **enabled** sub-property.

   ```
   "rulePacks" : {
                       "path" : "path/to/myrulepack1.zip",
                       "enabled" : false
                   }


   "rulePacks": {
       "path": "path/to/myrulepack1.zip",
   ```

```
    "enabled": false
}
```

The sample overrides the **enabled** system property for a rule pack by disabling the rule pack specified by **path**.

3. To disable all built-in JET rules and run the audit with only custom rule packs, edit the `oraclejafconfig.json` file and set the **builtinJetRules** property to **false**, and then edit the **rulePacks** property, as described above to enable the desired custom rule packs.

```
"builtinJetRules" : false
```

The sample overrides the **builtinJetRules** property to disable all the JET rules provided with the Oracle JAF installation.

Audit Only HTML Files that Contain Oracle JET Components

Use the optional Oracle JAF configuration file property **jetPagesOnly** to surpress auditing of an HTML file if the page does not contain any Oracle JET custom elements. If the property is disabled, then JAF processes and reports issues on all HTML files in the application.

When enabling auditing of Oracle JET only pages, JAF examines all elements in the page after parsing and before firing any rules. Thus enabling the **jetPagesOnly** property is less performant than when the property is disabled.

To audit only pages with Oracle JET components:

- To change the audit to ignore pages without JET components and audit only pages with JET components, edit the `oraclejafconfig.json` file **jetPagesOnly** property.

```
{
   ...
   "jetPagesOnly" :  true
   ...
}
```

Audit JET Custom Web Component Usages

Because Oracle JET Web Components are user-defined, auditing their custom HTML depends upon processing the Web Component's metadata before auditing can begin. Use the optional JAF configuration properties **components** and **componentsUrls** to enable auditing support for such custom HTML elements by informing JAF where to find and extract Web Component metadata.

Oracle JET custom Web Components in your application rely on the `component.json` file to define metadata for their API, including their tag names and supported properties, methods, and events. When you want JAF to process rules specific to Web Components, you must set the JAF configuration **components** property for local Web Components to inform JAF where to find the `component.json` file.

During its pre-process phase, JAF recursively searches the specified locations to extract the component metadata. After this initial phase, the configuration properties are no longer needed and in subsequent phases of the audit, the previously extracted component metadata is used by the built-in JET rules defined by the `jet-cca` group to resolve Web Component references.

Although not a complete review of the JAF rules in the `jet-cca` group, once properly configured, JAF will enforce implementation details such as the following.

- Check that components don't use a reserved namespace prefix (such as, `oj-`).
- Check that `for` attributes are prefixed and point to an expression.
- Checks that hardcoded element IDs are not used within the component implementation.

Because user-defined Web Components are associated with a namespace to avoid naming collisions, JAF needs to be informed about the namespace to prevent reporting references in the audit file set as not valid. By convention, the prefix of the Web Component name identifies the namespace. To declare the namespace for Web Components not defined by the `oj` namespace, you set the JAF configuration property **nameSpaces**.

To support auditing of Web Components:

1. To set the `component.json` file path for Web Components that reside within the application, edit the `oraclejafconfig.json` file **components** property.

   ```
   "components" : ["path/to/my_components"]
   ```

   The property may be specified as a single string, or as an array of strings, where each folder is inspected, and a search is made for `component.json` files at the top level and recursively down through all child levels.

   ```
   "components": [
       "path/to/my_components/group1",
       "path/to/my_components/group2"
   ]
   ```

2. **Important:** Set the **file** property to include the folder paths of locally referenced Web Components.

   ```
   "files": [
       ...
       "./components/**"
   ]
   ```

3. Suppress false reporting of valid references to Web Components by editing the `oraclejafconfig.json` file **nameSpaces** property and declare the list of allowed namespaces.

   ```
   "nameSpaces" : [ "my-foo", "my-bar", ... ]
   ```

   The namespace of user-defined Web Components is by convention the prefix you assigned the component to avoid naming collisions. In this example, Web Components with the prefixes `my-foo` and `my-bar` will not be flagged by the JAF audit when referenced in the file set.

   Tip: You can list the namespaces known to the JAF version you are running by entering this `ojaf` command:

   ```
   ojaf --nslist
   ```

4. (Optionally) Suppress validation of Web Component metadata if you do not want the current `component.json` file schema to be applied during the pre-audit phase.

```
"components":  ... ,
"componentOptions": {
    "applySchema": false
}
```

This can be useful to prevent JAF-INIT messages from being displayed if the schema pass identifies component metadata issues.

5. (Optionally) To enforce a maximum number of Web Components to exist in an HTML page, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JET**, and customize the built-in **oj-html-cca-count** rule specification.

```
"ruleMods": {
    "JET": {
        "oj-html-cca-count": {
            "thresholds": {
                "oj-table": 5
            }
        }
    }
}
```

The threshold for allowed Web Components in an HTML page is any number. For example, this sample checks the number of tables on the page.

Component specifications can include wildcard characters. For example, this sample checks for all gauges and buttons, in addition to the oj-table component, and also limits the total number of components used by using the **$total** property.

```
"ruleMods": {
    "JET": {
        "oj-html-cca-count": {
            "thresholds": {
                "oj-table": 5,
                "oj-*-gauge": 10,
                "oj-buttonset-*": 10,
                $total: 20
            }
        }
    }
}
```

An invalid rule option will prevent Oracle JAF from running the **oj-html-cca-count** rule. Confirm that the option **thresholds** has been specified correctly.

Audit JET Custom Web Component Projects

You can run an extended audit over the Web Component project that you created using the Oracle JET Tooling by enabling the rule set **builtinJetWcRules** in the JAF configuration file.

Before you share a custom Web Component that you create, you can audit the standalone JET project that implements the Web Component. The rule set **builtinJetWcRules** is provided

specifically for component developers to use to validate best practices and other concerns that guarantee the validity of Web Component implementation details.

The **builtinJetWcRules** rule set is defined by a number of audit groups that you can enable individually for fine control over the rules to invoke. For example, you might enable the rule group **jetwc-pre-release** just before the cut to production. The rule set also contains rules that you can configure specific to your project. All rule groups and individual rules with the groups are defined by the prefix **jetwc**.

To run the audit on your Web Component project, the project must be a standard JET project that you created as the source for your custom components. Specifically, the rule set expects the organization of the project folders to be consistent with the organization created by Oracle JET CLI when you create a new component or pack of components, using the `ojet create component` and `create pack` commands.

The **builtinJetWcRules** rule set enforces implementation details such as the following.

- Checks that the folder structure and component names within the folders are as expected by the Oracle JET Tooling, and checks other such structure-related concerns.
- Checks the validity of dependency relationships between components, including semvar values, JET version values, requireJS paths, and component API implementations.
- Checks for consistency and correctness of component APIs in the namespace of your custom component, and that non-template slots are declared, events are declared, and that properties are nested only to a specified level—to identify just a few of the many API audit concerns.
- Checks for deprecated API styles, usages, and practices in JET.
- Checks for NLS support to verify the root language bundle location and that translation bundles have been provided for the specified locales.
- Checks for issues related to theme-able components, such as the presence of the ojcss! plugin.
- Checks for issues that can effect the usage of your component in Oracle Visual Builder, such as length limits for a display name that you define and the availability of a specified icon.

To enable auditing of Web Component projects and configure customizable rules:

1. To enable the **builtinJetWcRules**, edit the `oraclejafconfig.json` file. This is a basic configuration for a Web Components project:

```
{
  "jetVer": "17.1.0",
  "base": "$jafcwd",
  "files": [
    "./src/**/*.html",
    "./src/**/*.js",
    "./src/**/component.json"
  ],
  "components": [
    "./src/js/jet-composites"
  ],
  "builtinJetRules": true,
  "builtinJetWcRules": true,
  "format": "json",
  "outPath": "audit-report.json"
}
```

The files and components properties specify the paths found in the standard project folder structure, as generated by the JET Tooling. The **$jafcwd** macro sets the base directory to the one in which the audit is invoked.

2. To declare subsets of the rules to run, edit the `oraclejafconfig.json` file **groups** property.

```
{ "groups" : ["ruleGroup1", "ruleGroup2", ...],
}
```

In the case of the JET built-in rules, you can specify the rules to run by declaring the group name for the built-in JET rule pack subset as described below. For example, where the group name **jetwc-structure** defines the folder structure and **jetwc-api** defines the API usage rules, then the sample enables only the rules belonging to those groups.

```
{ "groups" : ["jetwc-structure", "jetwc-api"],
}
```

Built-in Web Component Rule Group Name	Usage
jetwc-structure	Rules that check that the overall disk layout is as expected by the Oracle JET Tooling. This group should be enabled for most component sets.
jetwc-dependencies	Rules that verify the various dependency relationships between components both within and across packs. This group should be enabled for all component sets.
jetwc-pre-release	Rules to be run before you cut a production release.
jetwc-api	Rules that verify the consistency and correctness of your component APIs. This group should be enabled for all component sets.
jetwc-vb	Rules that specifically check for things that effect the Visual Builder usage experience of your components.
jetwc-nls	Rules relating to NLS support.
jetwc-deprecations	Rules that check for known deprecated usages.
jetwc-theming	Rules to enable if you are trying to create theme-able components.

3. (Optionally) To verify a specified set of files exists in the project, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JETWC**, and customize the built-in **jetwc-standard-files** structure rule specification. Note that you must ensure that all files that you want to check for are listed.

```
"ruleMods": {
    "JETWC": {
        "jetwc-standard-files": {
            "customOpts": {
                "files": [
                    "README.md",
                    "changes.txt",
                    "version.mf"
                ]
            }
```

```
        }
    }
```

The **jetwc-standard-files** rule checks by default for the `README.md` and `CHANGELOG.md` files, but if you customize this rule, then you must list them.

4. (Optionally) To impose some length limit for the **displayName** property when the Web Component will be used with Visual Builder, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JETWC**, and customize the built-in **jetwc-displayname** Visual Builder rule specification.

```
"ruleMods": {
    "JETWC": {
        "jetwc-displayname": {
            "enabled": true,
            "customOpts": {
                "limits": {
                    "component": {
                        "length": 30,
                        "words": 4
                    },
                    "property": {
                        "length": 30,
                        "words": 3
                    },
                    "event": {
                        "length": 30,
                        "words": 3
                    },
                    "template": {
                        "length": 40,
                        "words": 8
                    }
                }
            }
        }
    }
```

Note that the number of letters and words can be configured for components that appear in either the Component palette (Composites and Patterns) or in Visual Builder Application templates, as well as for Events and Properties in the Properties pane.

5. (Optionally) To specify a list of requireJS paths used by `define()` or `require()` calls in your component that are to be assumed to be available, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JETWC**, and customize the built-in **jetwc-require-paths** dependency rule specification. For example to disallow JQuery and VB (Visual Builder) paths, remove them from the **standardPaths** list.

```
"ruleMods": {
    "JETWC": {
        "jetwc-require-paths": {
            "enabled": true,
            "customOpts": {
                "standardPaths": [
                    "ojs",
```

```
                    "knockout",
                    "hammerjs",
                    "persist"
                ]
            }
        }
    }
```

All other valid paths are derived from the declared dependencies of the component that owns the file being audited.

The path "./" is always considered to be valid. The path "../" is always considered to be invalid.

Take care when adding extra standard paths of your own and avoid adding a path that constitutes a dependency that is not shipped as a standard part of JET or Visual Builder. For such cases, your component needs to declare an explicit dependency to another component (for example, by using a reference or resource component).

6. (Optionally) To ensure that nesting of Web Component properties do exceed the desired level, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JETWC**, and customize the built-in **jetwc-property-nesting** API rule specification. By default the rule checks property nesting does not exceed two child levels (also called sub-properties). To check for nesting beyond three levels of sub-properties, set **depth** to 4, as shown.

```
"ruleMods": {
    "JETWC": {
        "jetwc-property-nesting": {
            "enabled": true,
            "customOpts": {
                "depth": 4
            }
        }
    }
```

Nesting of properties beyond two child levels is not recommended. To disallow sub-property use all together you would set **depth** to 1.

7. (Optionally) To ensure that translation bundles exist for a specified list of locales, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JETWC**, and customize the built-in **jetwc-nls-languages** NLS rule specification. JAF will invoke this rule only when you have configured the locales you want to check against.

```
"ruleMods": {
    "JETWC": {
        "jetwc-nls-languages": {
            "enabled": true,
            "customOpts": {
                "locales": [
                    "fr",
                    "fr-ca",
                    "de",
                    "zh-Hans|zh-CN"
                ],
                "strict": false
            }
```

```
        }
    }
```

Use the pipe (`|`) symbol to separate acceptable alternatives. For example, the sample shows either `zh-Hans` or `zh-CN` locale is acceptable.

By default, the checking code is lenient, with case matching and underscore v's hyphen. You can set **strict** to **true** to force exact matching.

8. To allow exceptions to the **jetwc-css-scoping** rule to include CSS in the component CSS without component tag scoping, include a configuration option with an array of scopes to ignore. For example, the following configuration option:

```
"ruleMods" : {
  "JETWC":{
    "jetwc-css-scoping":{
     "enabled":true,
       "customOpts":{
         "ignoreScopes":["oj-sp-color-invert-bg"]
    }
  }
}
```

Allows the inclusion of `oj-sp-color-invert-bg` in the component CSS without component tag scoping, assuming that the `oj-sp-color-invert-bg` class is in a parent of the component.

Audit JET Web Component Projects Containing VComponents

The **builtinJetWcRules** rule set supports the auditing of Web Component projects that use VComponent TSX files in addition to the JS and TS files used by conventional CCA components.

To enable the **builtinJetWcRules** and thus the custom component audits, you must edit the `oraclejafconfig.json` file and set **builtinJetWcRules** to `true`.

The **builtinJetWcRules** rule set expects that the web component is a standard JET project that was created as the source for your custom components. That is, the rule set expects that the context and folder organization of the project be consistent with projects created using the Oracle JET CLI tool.

In order to include TSX files in the project audit, a TypeScript compilation must first be run against the code. Therefore, there are some extra requirements for the JAF configuration `oraclejafconfig.json` file:

- You must be running JAF 3.3.0 or later.
- Your JAF configuration must include `.tsx` files in the **files** property array to ensure that VComponents are processed.
- You must set the `"typescript":{"compile":true}` option in your JAF configuration. Without this, `.tsx` files will be skipped.
- The project under audit needs to have populated `node_modules`, which includes `@oracle/oraclejet`, as the JET version used by the project is also used for compiling.
- Your project must compile successfully before the audit can be run.

For a mixed Typescript CCA and VComponent project, your **oraclejafconfig.json** file should look similar to the following:

```
{
  "jetVer": "17.1.0",
  "base": "$jafcwd",
  "typescript":{
    "compile":true,
  },
  "files": [
    "./src/**/*.html",
    "./src/**/*.ts",
    "./src/**/*.tsx",
    "./src/**/component.json"
  ],
  "components": [
    "./src/js/jet-composites"
  ],
  "builtinJetRules": true,
  "builtinJetWcRules": true,
  . . .
}
```

Audit CSS Styles and Web Components Styles

Use the optional Oracle JAF configuration property **stylesets** to create a whitelist of valid user-defined web component styles to enable auditing support for an undefined JET core style, a misspelled style name, or an unknown (to JAF) Web component style.

The configurable style rules in the built-in JAF rule set can emit accept or reject CSS style diagnostics. In order to allow JAF to accept valid user-defined styles for Web components, you can specify the list of supported Web component styles in the **stylesets** property. If Web component styles are not defined, JAF may return a false positive for a valid style.

To audit CSS styles and include valid Web component styles:

1. To enable auditing of CSS styles that do not return a false positive for valid Web component styles, edit the `oraclejafconfig.json` file **stylesets** property and define an array of style names for the namespace of the Web component.

   ```
   "stylesets": [
       // For namespace "oj-xxx"
       "oj-xxx-color",
       "oj-xxx-bgcolor",
                   . . .

       // For namespace "oj-yyy"
       "oj-yyy-foo",
       "oj-yyy-bar",
   . . .
   ]
   ```

   The sample enables JAF to distinguish valid Web component styles from an undefined JET built in style, a misspelled style name, or an unknown (to JAF) Web component style.

2. To enable auditing of CSS styles and specify the valid Web component styles in an include `.txt` file, edit the `oraclejafconfig.json` file **stylesets** property and add **@include()** with the file path of the style set lists.

```
"stylesets" :  [
   @include('./stylesets/styleset-oj-sample.txt') ,
   // trailing comma to terminate the included list (see below)
   @include('./stylesets/styleset-oj-foo-bar.txt')
 ],
```

The sample specifies the location of the style set list for multiple Web components to disassociate their style lists from the configuration for easier maintenance of the style lists.

A sample `styleset-oj-sample.txt` file might look like the following. Note that if no trailing comma terminates the style list, one can be added after the **@include()**, as shown above.

```
// oj-sample styles
"oj-sample-card-emp-image",
"oj-sample-card-emp-name",
"oj-sample-card-emp-initials",
"oj-sample-card-emp-title"    <-- no trailing comma
```

3. (Optionally) To report deprecation details on a style that has been deprecated, edit the `oraclejafconfig.json` file **styleset** property and replace the style name with an object containing **deprecated** and optional **since** and **note** properties.

```
// oj-sample styles
"oj-sample-card-emp-image",
{"deprecated" : "oj-sample-card-emp-name"},
{"deprecated" : "oj-sample-card-emp-initials", "since": "8.2.0", "note":
"Replace with oj-xxx"},
"oj-sample-card-emp-title"
```

The sample informs JAF to report a deprecated style diagnostic, where the **since** and **note** properties will be added to the message.

```
[10, 20] CSS class selector 'oj-sample-card-emp-name is deprecated.  //
since/note properties not defined
[10, 20] CSS class selector 'oj-sample-card-emp-initials' is deprecated
(since 8.2.0). Replace with 'oj-xxx' // props defined
```

4. (Optionally) To enforce validation of JET styles on HTML elements (including JET built-in components) and custom web component elements, edit the `oraclejafconfig.json` file and customize the built-in **oj-html-stylesel** rule specification.

```
"ruleMods": {
    "JET": {
        "oj-html-stylesel": {
            // these styles will not be checked in HTML and web component
elements
            "ignore": [
                "oj-ux-*, "oj-fwk-*"]
            }
```

```
        }
    }
```

> **Note:**
>
> Starting in JAF 2.10.0, **oj-html-stylesel** replaces audit rules **oj-html-style** and **oj-html-ojstyle** by validating the CSS styles used in HTML elements, including JET core components, and custom web components.

Checks for JET styles (begin with `oj-`) used in HTML `class` attribute styles, as well as in style metadata associated with custom web components, defined by JAF configuration properties `components` and `componentsUrls`. CSS styles must be valid JET style classes. You can configure the rule to ignore certain classes. The class strings are treated as regular expressions, thus wild card characters can be used.

5. (Optionally) To enforce checks for deprecated JET styles, edit the `oraclejafconfig.json` file and customize the built-in **oj-css-style-deprecated** rule specification.

```
"ruleMods": {
    "JET": {
        "oj-css-style-deprecated": {
            // all JET class selectors will be tested for deprecation,
except these
            "reject": [
                "oj-form-*"
            ]
        }
    }
}
```

Checks that CSS does not specify deprecated JET class selectors. The rule can be configured to accept or reject certain style selectors. The class strings are treated as regular expressions, thus wild card characters can be used. Property **accept** specifies that only the class selector(s) declared should be tested for deprecation. Property **reject** specifies that all JET class selectors should be checked for deprecation except those specified by it. The properties **accept** and **reject** are mutually exclusive.

6. (Optionally) To enforce verification that CSS does not specify overrides of JET styles, edit the `oraclejafconfig.json` file and customize the built-in **oj-css-style-override** rule specification.

```
"ruleMods": {
    "JET": {
        "oj-css-style-override": {
            // only these styles will be tested for overrides
            accept": ["oj-list*"]
        }
    }
}
```

or

```
"ruleMods": {
    "JET": {
        "oj-css-style-override": {
            // all styles will be tested for overrides, except these
            reject": ["oj-list*"]
        }
    }
}
```

Checks that CSS does not specify overrides of JET styles. The rule can be configured to accept or reject certain style classes. The class strings are treated as regular expressions, thus wild card characters can be used. Property **accept** specifies that only the class selector(s) declared should be tested for deprecation. Property **reject** specifies that all JET class selectors should be checked for deprecation except those specified by it. The properties **accept** and **reject** are mutually exclusive.

Note: If you require all `.oj-*` styles to be checked for overrides, then declare an empty **reject** array. Because all styles are checked, except for those defined in **reject**, the net effect is that all `.oj_*` styles are checked.

7. (Optionally) To enforce checks for the use of CSS variables that have been overridden, and avoid their use, as a best practice, edit the `oraclejafconfig.json` file and customize the built-in **oj-css-var-override** rule specification.

```
"ruleMods": {
    "JET": {
        "oj-css-var-override": {
            "var": {
                // 'ignore' is used to ignore any overridden CSS vars that
match
                "ignore": [
                    "^oj-foo"
                ] //  For example, ignore CSS vars starting with oj-foo
            }
        },
        "oj-css-style-bp-font": {
            "var": {
                // 'accept' is used to report only on overridden CSS vars
that match
                "accept": [
                    "^oj-foo"
                ] //  For example, report only CSS variables beginning
with oj-foo
            }
        },
    }
}
```

Optional CSS variable names using regular expression strings may be added to this rule's options to control the audit. For this rule, the rule option **var** can be overridden using configuration property **ruleMods**. The **var** property has two mutually exclusive sub-properties **ignore**, and **accept**.

8. (Optionally) To enforce checks for color references in standalone CSS or in embedded in HTML `<style>`, edit the `oraclejafconfig.json` file and customize the built-in **oj-html-style-bp-color** and **oj-css-style-bp-color** rule specifications.

```
"ruleMods": {
    "JET": {
        "oj-html-style-bp-color": {
            // only this CSS property audited for color references
            "accept": [
                "background-color"
            ]
        },
        "oj-css-style-bp-color": {
            // do not audit these properties for color references
            "reject": [
                "background-image",
                "border"
            ]
        }
    }
}
```

Check for use of color references in CSS (standalone or embedded in HTML `<style>`). They may be configured to accept or reject the color references found on specific CSS properties. Property **accept** specifies that only the properties declared by it should be audited. Property **reject** specifies that all properties should be audited except those specified by it. The properties **accept** and **reject** are mutually exclusive.

9. (Optionally) To enforce checks for the use of `font-size` and `font-weight` in standalone CSS or in embedded in HTML `<style>`, edit the `oraclejafconfig.json` file and customize the built-in **oj-html-style-bp-font** and **oj-css-style-bp-font** rule specifications.

```
"ruleMods": {
    "JET": {
        "oj-html-style-bp-font": {
            // only this CSS font property will be audited
            "accept": [
                "font-size"
            ]
        },
        "oj-css-style-bp-font": {
            // accept font-size, but reject font-weight audits
            "reject": [
                "font-weight"
            ]
        }
    }
}
```

Check for use of `font-size` and `font-weight` in CSS (standalone or embedded in HTML `<style>`). They may be configured to accept or reject the font references found on specific CSS properties. Property **accept** specifies that only the properties declared by it should be audited. Property **reject** specifies that all properties should be audited except those specified by it. The properties **accept** and **reject** are mutually exclusive.

10. (Optionally) To enforce checks for the use of certain `font-family` values in standalone CSS or in embedded in HTML `<style>`, edit the `oraclejafconfig.json` file and customize the built-in **oj-html-style-bp-font-family** and **oj-css-style-bp-font-family** rule specifications.

```
"ruleMods": {
    "JET": {
        "oj-html-style-bp-font-family": {
            // only this CSS font family will be audited
            "accept": [
                "Oracle Sans"
            ]
        },
        "oj-css-style-bp-font-family": {
            // ignore these
            "ignore": [
                "inherit",
                "initial"
            ]
        }
    }
}
```

These rules may be configured to accept or ignore font family values found on specific CSS properties. Property **accept** specifies that only the font families declared by it should be audited. Property **ignore** specifies that all font families should be audited except those specified by it. The properties **accept** and **ignore** are mutually exclusive.

11. (Optionally) To enforce checks for the use of CSS absolute length units in standalone CSS or in embedded in HTML `<style>` so that you may replace them with relative units, as a best practice, edit the `oraclejafconfig.json` file and customize the built-in **oj-html-style-abs-units** and **oj-css-style-abs-units** rule specifications.

```
"ruleMods": {
    "JET": {
        "oj-html-style-abs-units": {
            "absunits": {
                // only px quantities > 2 will be audited
                "px": 2,
                ...
            }
        },
        "oj-css-style-abs-units": {
            "absunits": {
                // all units other than px have a threshold
                "all": 2,
                // only px quantities > 2 will be audited
                "px": 2,
                ...
            }
        },
    }
}
```

Check for use of CSS absolute length units in CSS (standalone or embedded in HTML `<style>`). Optional quantity thresholds can be set for all or each units For example, if 2 is set as a threshold for "px", then only quantities whose absolute value is larger than the threshold 2 will emit diagnostics. For CSS rules, specific rule selectors may also be optionally configured via regular expressions. The two optional rule option properties are **absunits** and **selectors**. The **selector** property (for rule `oj-css-style-abs-units` only) has two mutually exclusive sub-properties: **accept** and **ignore**. Property **accept** specifies that only the properties in the rule block for the selector(s) declared should be tested. Property **ignore** specifies that any selectors that match should be ignored.

For rule **oj-css-style-abs-units** only, the **selectors** property may be specified separately, or in conjunction with **absunits** to set the scope based on the selector classes in the rule. A selector string is a regular expression string. When **absunits** and **selectors** are both specified, they are considered to be an AND condition. In that case, only the units exceeding the specified threshold and matching the **selectors** list are reported.

```
"ruleMods": {
    "JET": {
        "oj-css-style-abs-units": {
            "absunits": { ...
            },
            "selectors": {
                // ignore absolute units in rule blocks with
                // selector classes ending in "-image"
                "ignore": [
                    "-image$"
                ]
            }
        }
        "oj-css-style-abs-units": {
            "absunits": { ...
            },
            "selectors": {
                // ignore absolute units in rule blocks with
                // selector classes ending in "-image"
                "accept": [
                    "-image$"
                ]
            }
        },
    }
}
```

Audit for Oracle JET Deprecated Functionality

By default the Oracle JAF configuration will process all rules of the built-in JET rule pack, including rules for detecting deprecated Oracle JET functionality. Use the optional **groups** property of the JAF configuration file to customize the audit to process only the rules related to deprecated functionality by adding the **jet-deprecated** rule group to the JAF configuration.

The **jet-deprecated** rule group when configured for an audit will alert you to the presence of functionality that has been deprecated for the JET release version that you have configured for JAF, including deprecated JET API methods and members, as well as deprecated JET custom HTML components. Although the processing of rules in the **jet-deprecated** rule group is

dependent on the JET release version, you can expect rules similar to the following to apply to your application.

- **oj-js-ojcomp-deprecated** - Deprecated JET component classes should not be instantiated.
- **oj-html-ojtag-deprecated** - Deprecated JET custom components should not be used.
- **oj-js-comp-attr-deprecated** - Deprecated JET component class attributes should not be referenced. This rule's scope is configurable by the JAF configuration file **ruleMods** property, as described below.
- **oj-js-comp-meth-deprecated** - Deprecated JET component class methods should not be invoked. This rule's scope is configurable by the JAF configuration file **ruleMods** property, as described below.
- **oj-html-ojattr-deprecated** - Deprecated JET component attributes should not be used.
- **oj-html-ojslot** - Deprecated JET component **<oj-slot>** should not be used since it is a binding-only element, and not a full custom element with properties and methods that can be accessed.
- **oj-html-style-deprecated** - JET component **class** attribute for deprecated CSS styles should not be used.
- **oj-css-style-deprecated** - Deprecated JET CSS class selectors should not be used.

To audit deprecated functionality:

1. To enable processing exclusively the **jet-deprecated** rule group for all audits, edit the `oraclejafconfig.json` file **groups** property.

   ```
   { "groups" : ["jet-deprecated"]
   }
   ```

   JAF audits will process only the **jet-deprecated** rule group until the **groups** property setting is changed or until overridden for a single audit from the `ojaf` utility command line.

2. Alternatively, to enable processing exclusively the **jet-deprecated** rule group for a single audit (without editing the configuration file), enter the following `ojaf` command.

   ```
   ojaf --groups jet-deprecated
   ```

   A command that you enter in the command-line overrides the corresponding property setting in the JAF configuration file for the duration of the current audit.

3. To configure the scope of the audit rules that check for deprecated/deleted members and methods, edit the `oraclejafconfig.json` file **ruleMods** property, specify the prefix of the built-in rule pack **JET**, and set the **deprecated** and **deleted** sub-properties of the **scope** property by specifying the enabled status for the **sure** and **unsure** confidence level options.

   ```
   "ruleMods": {
       "JET": {
           "oj-js-comp-attr-deprecated": {
               "scope": {
                   "deprecated": {
                       "sure": true,
                       "unsure": false
                   },
   ```

```
                "deleted": {
                    "sure": true,
                    "unsure": false
                }
            }
        },
        "oj-js-comp-meth-deprecated": {
            "scope": {
                "deprecated": {
                    "sure": true,
                    "unsure": false
                },
                "deleted": {
                    "sure": true,
                    "unsure": false
                }
            }
        }
    }
}
```

The sample shows the default values for the confidence level options **sure** and **unsure** as specified by using the **ruleMods** property to configure the audit rules that detect deleted/deprecated members/functions. During static analysis of JavaScript/TypeScript, it may not be possible to determine with confidence the contents of `obj` in `obj.fn()` and `obj.mem`. This can cause the rules to be noisy in the case where a known deleted/deprecated member/function has been detected, but the rule has low confidence in the contents of `obj`. To reduce noise in the audit report, the scope of these rules can be configured by setting the confidence level for the audit. All properties are optional. For more details about the **ruleMods** property, see Configure Audit Rule Runtime Properties .

4. To view long descriptions of the rules in the **jet-deprecated** rule group, enter the following `ojaf` command.

```
ojaf -r
```

4

Fine Tune the Audit

You can customize the Oracle JAF configuration to narrow the focus of an audit by disabling rules, rule groups, or specific message IDs. You can also add Oracle JAF comments within source files for finer-grained control over what to audit.

Restrict Audit Rule Severity Level

Use the optional Oracle JAF configuration **groups** property to limit audit results to the desired rule severity level. If the property is omitted, all issues found are reported.

The **severity** property specifies the rule severity level to which audit issue reporting will be restricted. By default, the severity levels defined by Oracle JAF, in descending priority order, are **blocker**, **critical**, **major**, **minor**, and **info** as described below.

> **Note:**
>
> When your organization prefers to standardize on severity levels other then this list, you can redefine these levels using your own severity levels by editing the **sevMap** property in the `oraclejafconfig.json` file. Additionally, rules, such as those in the built-in JET rule set, have a default severity level that you may map to an alternate severity level. See Alter the Severity Level of an Audit Rule .

Severity Level	Description
blocker	A bug with a high probability to impact the behavior of the application in production. The code *must* be immediately fixed.
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. The code *must* be immediately reviewed.
major	A quality flaw which can highly impact developer productivity. For example, uncovered piece of code, duplicated blocks, or unused parameters.
minor	A quality flaw which can slightly impact developer productivity. For example, lines should not be too long or `switch` statements should have at least three cases.
info	A finding that is not a bug or a quality flaw.

To specify a rule severity filter or map custom severity levels:

1. To set the severity filter as a string, edit the `oraclejafconfig.json` file **severity** property:

```
"severity" : "critical",
```

The comparative operators >, >=, <, and <= may precede the severity level. For example, the following will display issues of severity "minor", "major", "critical", and "blocker" level.

```
"severity" : ">info",
```

This could also have been written:

```
"severity" : ">=minor",
```

2. To set the severity filter as a list, edit the `oraclejafconfig.json` file **severity** property.

```
"severity" : ["critical", "blocker"],
```

Alter the Severity Level of an Audit Rule

Use the Oracle JAF configuration property **ruleMods** and **severity** rule property to remap the default severity level of audit rules or use the configuration property **sevMap** to replace the default severity levels with ones used by your organization.

You can use the **ruleMods** configuration property to override the severity level assigned to an audit rule defined by configured rule packs, including any JET built-in rule. Additionally, when your organization prefers to standardize on severity levels other than those provided by Oracle JAF, you can replace the default severity levels by specifying user-defined levels in the **sevMap** property. For more information about severity levels, see Restrict Audit Rule Severity Level.

To customize audit rule severity levels:

1. To redefine the severity level assigned to individual audit rules, edit the `oraclejafconfig.json` file **ruleMods** property, specify the rule pack prefix of the rule (**JET** is the prefix of the JET built-in rules), and set the desired value for the **severity** sub-property.

```
"ruleMods" : {
               "JET" : {
                          "oj-html-ojattr" : {"severity": "critical"},
                          "oj-html-lib" : {"severity": "major"}
                        }
             }
```

The sample reclassifies the built-in rule **oj-html-ojattr** as severity-level **critical** and reclassifies the built-in rule **oj-hmtl-lib** as severity level **major**. The default severity levels that you can specify, in ascending order of restrictiveness, are **info**, **minor**, **major**, **critical**, and **blocker**.

2. To redefine the severity level of individual audit rule messages, edit the `oraclejafconfig.json` file **sevMap** property and specify the **sevMsg** sub-property.

```
"sevMap" : {
            "sevMsg" : {
                          "JET-xxxx", "minor",
                           ...
                        }
            }
```

If the severity level set is also redefined to user-defined levels (see sub-property **sevSet** below), then the new severity levels may be used in **sevMsg**.

```
"sevMap" : {
            "sevMsg" : {
                          "JET-xxxx", "sev4",
                          ...
                       },
            "sevSet" : {
                          // "sev4" defined here, see below
                       }
           }
```

3. To redefine the entire default set of severity levels, edit the `oraclejafconfig.json` file **sevMap** property and specify the **sevSet** sub-property.

```
"sevMap" : {
             "sevSet" : {
                          "blocker" :  "sev1",
                          "critical" : "sev2"
                          "major" :    "sev3"
                          "minor" :    "sev4"
                          "info" ;     "sev5"
                        }
           }
```

sevSet can also be used to reduce the number of severity levels used. For example reduce the number of severity levels to two levels.

```
"sevMap" : {
             "sevSet" : {
                          "blocker" :  "mustfix",
                          "critical" : "mustfix"
                          "major" :    "mustfix"
                          "minor" :    "warning"
                          "info" ;     "warning"
                        }
           }
```

Suppress Auditing Linked Content

Use the optional Oracle JAF configuration **followLinks** property to control whether `<link>` and `<script>` elements in HTML that refer to external stylesheet and JavaScript/TypeScript files are followed, and the files are audited.

By default, JAF enables auditing of externally linked files that include stylesheets and JavaScript/TypeScript files. To prevent externally linked files from being audited, you must disable the **followLinks** setting in the JAF configuration file.

To suppress auditing externally linked files:

- To suppress audit message resulting from externally linked files, edit the `oraclejafconfig.json` file **followLinks** property.

```
"followLinks" : false
```

Suppress Audit Messages

Use the optional Oracle JAF configuration **messages** property to control which messages are emitted in the audit report. If the property is omitted, all issues found are reported.

The **messages** property takes two sub-properties **reject** and **accept** that you can use to tailor the list of audit messages emitted in a report: either to suppress particular messages, or alternatively to return only desired messages. The sub-properties are specified by a list of message IDs to filter. These sub-properties are mutually exclusive, so that the message IDs in the specified lists must not overlap. Regular expressions and wildcard characters can be used to specify the message ID. For example, `"JET-20*"` and `"JET-3[0-9]+"` are valid.

To control the list of reported audit messages:

1. To suppress messages with particular messages IDs, edit the `oraclejafconfig.json` file **messages** property and specify the message IDs to filter out as a list in the **reject** sub-property.

```
"messages" : {
               "reject" : ["JET-3020", "JET-20*"]
             },
```

   This sample specifies the audit report will exclude the message with ID `JET-3020` and exclude the set of messages with IDs like `JET-2000`, `JET-2010`, `JET-2020` and so on.

2. To report only those messages with particular message IDs, edit the `oraclejafconfig.json` file **messages** property and specify the message IDs to report as a list in the **accept** sub-property.

```
"messages" : {
               "accept" : ["JET-3020", "JET-2[1-2]+"]
             },
```

   This sample specifies the audit report will only include the message with ID `JET-3020` and only include the set of messages with IDs like `JET-2100`, `JET-2120`, `JET-21`*`xx`*, `JET-2200`, `JET-2210`, `JET-22`*`xx`*.

Adjust the Tab Value Used to Report Line and Column Issues

Use the optional Oracle JAF configuration **tabs** property to control how tab characters are handled when encountered in the audit.

By default, JAF assumes that each tab character represents 4 spaces. When you need to adjust this value for your application files, you can specify settings for specific HTML, JS, CSS, and JSON file types. Each file type can define the number of spaces to use for a tab character and a list of column values to use for individual tab stops.

```
"tabs" : {
            "html" | "js" | "css" | "json" | "all" : {
```

```
                    <tab settting objects per file type>
                },
        }
```

Two configuration tab styles are available for advancing to a column when a tab character is encountered: either a tab is equated with *n* spaces, using the **spaces** sub-property or else the column advances to the next tab stop column, using the **stops** sub-property. If both **stops** and **spaces** are specified, the tab configuration style is *tab stops*, and JAF uses the **spaces** value to calculate the next tab stop column whenever a tab advances beyond the last **stops** position.

```
"tabs" : {
            "html" | "js" | "css" | "json" | "all" : {
                       "spaces" : n,              // declare tab spacing
                       "stops" : [i, j, k, ...]   // declare tab stop columns
                },
        }
```

To adjust tab settings:

1. To configure the tab style setting to use within *specific* application file types (including HTML, JS, CSS, or JSON), edit the `oraclejafconfig.json` file **tabs** property.

```
"tabs" : {
           "html" : {
                       "spaces" : n                // declare tab spacing
                       "stops" : [i, j, k, ...]    // declare tab stop
columns
                     },
           "js" : {
                       "spaces" : n
                       "stops" : [i, j, k, ...]
                     },
           "css" : {
                       "spaces" : n
                       "stops" : [i, j, k, ...]
                     },
           "json" : {
                       "spaces" : n
                       "stops" : [i, j, k, ...]
                     }
         }
```

   For example, the following tab style configuration sample generates tab stops at 8, 12, 16, 20, 24, and so on since both properties are specified.

```
"tabs" : {
           "html" : {
                      "spaces" : 4,
                      "stops" : [8, 12]  // generates tab stops at 8, 12,
16, 20, 24 ...
                     },
            ...
         }
```

2. To configure the same tab settings to use within *all* application file types, edit the `oraclejafconfig.json` file **tabs** property.

```
"tabs" : {
            "all" : {
                        "spaces" : n,              // declare tab spacing
                        "stops" : [i, j, k, ...]   // declare tab stop
columns
                     },
         }
```

The **all** sub-property can also be used in conjunction with any of the file type sub-properties (**html**, **js**, **css**, and **json**) to provide a default for other non-declared file types. For example the following entry would assume 5 spaces per tab for HTML, and 3 for all other file types.

```
"tabs" : {
           "html" : {"spaces" : 5}, // declares tab spacing for HTML files
only
           "all" :  {"spaces" : 3}  // declares tab spacing for all other
file types
         }
```

Comment Source Code for Fine-Grained Audit Control

Oracle JAF comment commands allow contextual audit suppression of specific lines or blocks of code within individual application files. Use JAF comment commands to refine audit results and to gain greater control over the reported issues.

The Oracle JAF configuration property **comments** set to **true** enables Oracle JAF to interpret comments that you insert into your source code. JAF recognizes comments with a JAF-specific command of the form:

`/* <JAFcommand> [optional data]`

or

`// <JAFcommand> [optional data]`

Note that chevrons (< >) do not appear in an actual command name and the use of square brackets ([ ]) when specifying optional data is optional.

All JAF comment commands have the prefix `jaf-`. The command name must immediately follow the opening `/*` or `//` and is specified as `/* jaf-xxx */` or `// jaf-xxx`, where a whitespace preceding the command name is permitted.

Note that no program text is permitted within a JAF comment.

The following table describes supported JAF comment commands.

Oracle JAF Comment Command	Description
`// jaf-disable-next-line` `/* jaf-disable-next-line */`	Disables all JAF audit rules for the next statement.
`// jaf-disable-next-line [rule1, rule2, ...]` `/* jaf-disable-next-line [rule1, rule2, ...] */`	Disables the specified JAF audit rules for the next statement.

Oracle JAF Comment Command	Description
`// jaf-disable-line` `/* jaf-disable-line */`	Disables all JAF audit rules for the current statement. `... some statement ; // jaf-disable-line`
`// jaf-disable-line [rule, rule2, ...]` `/* jaf-disable-line [rule, rule2, ...] */`	Disables the specified JAF audit rules for the current statement.
`// jaf-disable` `/* jaf-disable */`	Disables all JAF audit rules until the end of file, or until the next comment.
`/* jaf-disable [rule, rule2, ...] */` `// jaf-disable [rule, rule2, ...]`	Disables the specified JAF audit rules(s) until the end of file, or until the next comment. Note that the square brackets and commas are optional.
`// jaf enable` `/* jaf-enable */`	Enables all JAF audit rules.
`// jaf enable rule1, rule2, ...` `/* jaf-enable rule1, rule2, ... */`	Enables the specified JAF audit rules.

To comment source code to enable and disable JAF audit rules:

1. To enable commenting support, edit the `oraclejafconfig.json` file **comments** property.

   ```
   { "comments" : true }
   ```

2. In your target source file, use one or more `jaf-disable` comment commands to disable specific JAF audit rules until the end of the file, or until the next JAF comment command.

   ```
   // jaf-disable rule1
   // jaf-disable rule2
   // jaf-disable rule3
   ```

   Which is functionally the same as:

   ```
   // jaf-disable rule1, rule2, rule3
   ```

3. In your source file, use one or more `jaf-enable` comment commands to enable specific JAF audit rules until the end of the file, or until the next JAF comment command.

   ```
   // jaf-enable rule1
   // jaf-enable rule2
   // jaf-enable rule3
   ```

   Which is functionally the same as:

   ```
   // jaf-enable rule1, rule2, rule3
   ```

4. In your source file, combine JAF comment commands in an additive or subtractive manner to enable or disable all JAF audit rules, except those specified.

```
// jaf-disable        all rules are disabled after this
...
// jaf-enable rule1        all rules except rule1 are disabled after this
```

5. In your source file, combine JAF comment commands in an additive or subtractive manner to enable or disable all JAF audit rules, except those specified only for the current or next line.

```
// jaf-disable
...
// jaf-enable rule1, rule2
...
// jaf-disable-next-line rule3
...                                  <--  for this code statement, only
rule3 is disabled

...                                  <-- all rules disabled except rule1 and
rule2
```

5

Work with the Output of Audits

You can customize the Oracle JAF configuration to tailor the output for your needs.

About Audit Output

Apart from specifying the scope of the audit by using Oracle JAF configuration properties such as **files**, **excludes**, **severity**, **groups**, **ruleNames**, **ruleMods**, you can also customize the format of the output.

Oracle JAF provides these ways to customize the output of the audit report:

- The format of reported issues can be customized by using the **proseFormat** configuration property. You can also add a custom report title by using the **title** configuration property.
- You can specify that the output format be JSON by using the **format** and **outPath** configuration properties, and then you can process this JSON to create any desired output. For example, HTML could be generated for web use, or perhaps other information could be injected into the output and the resulting file distributed in email.

Display Details About a Rule

You can use the rule name or rule message ID from the audit results to obtain a description of the rule.

By default the JAF configuration file **format** property is set to **prose** and the rule name will display with the audit message. The rule name is appended to the prose output after the message ID:

```
[71, 41]  <oj-list-view> attribute 'selection' is deprecated!
Use selected attribute instead. [JET-0080 : oj-html-ojattr-deprecated]
```

With the rule name, you can obtain a description of the rule by using `ojaf --help`.

```
ojaf --help oj-html-ojattr-deprecated

Rule: oj-html-ojattr-deprecated  Severity: major
JET component deprecated attributes should not be used.
```

Alternatively, if you know the message ID, you can obtain the rule name and rule description:

```
ojaf --help JET-0080

Rule: oj-html-ojattr-deprecated  Severity: major
JET component deprecated attributes should not be used.
```

> **Note:**
>
> To get rule help, you must run `ojaf --help` from the directory that contains your application's `oraclejafconfig.json` file. By default this is the root of the application.

For more information about enabling rule names to display in audit results, see Display Rule Names with Audit Messages.

Toggle the Default Format of Audit Messages

Use the optional Oracle JAF configuration file property **format** to specify the default display format for audit messages.

You can set the JAF configuration file **format** property to **prose** or to **line** to toggle the default presentation of audit messages between these two styles. The prose format displays audit messages in a report style, while the line format flattens out audit messages into single lines.

You can also customize the presentation for audit messages in either style by applying custom templates that you create, as describe in Customize the Presentation of the Audit Messages.

To toggle the default display style:

1. To enable the display of audit messages in the report presentation, edit the `oraclejafconfig.json` file **format** property.

   ```
   "format" : "prose"
   ```

2. To enable the display of audit messages as single lines, edit the `oraclejafconfig.json` file **format** property.

   ```
   "format" : "line"
   ```

   Note that for certain environments, like Microsoft Visual Studio Code, line format supports hyperlinks on the file paths displayed within Oracle JAF audit messages.

Display Rule Names with Audit Messages

You can enable Oracle JAF to append the corresponding rule name to reported audit messages and you can use the rule name to return a description of the rule.

If the JAF configuration file **format** property is set to **prose** (the default), the rule name displays with the audit message. The rule name will be appended to the prose output after the message ID:

```
[71, 41]  <oj-list-view> attribute 'selection' is deprecated!
Use selected attribute instead. [JET-0080 : oj-html-ojattr-deprecated]
```

With the rule name, you can obtain a description of the rule by using `ojaf --help`.

```
ojaf --help oj-html-ojattr-deprecated
```

```
Rule: oj-html-ojattr-deprecated  Severity: major
JET component deprecated attributes should not be used.
```

Alternatively, if you know the message ID, you can view the rule name with the rule description:

```
ojaf --help JET-0080

Rule: oj-html-ojattr-deprecated  Severity: major
JET component deprecated attributes should not be used.
```

> **Note:**
>
> To get rule help, you must run `ojaf --help` from the directory that contains your application's `oraclejafconfig.json` file. By default this is the root of the application.

To configure rule names to display with audit messages:

1. To enable support for appending rule names, edit the `oraclejafconfig.json` file **format** property to ensure that either **prose** has been specified or that the property has been omitted (which specifies prose output is the default).

   ```
   "format" : "prose"
   ```

   Note that if the configuration file output format is set to **json**, the rule name is always included in the output, and configuration options are not required. For more information, see Output Audit Messages in JSON Format.

2. To enable appending the rule name for a specific rule, edit the `oraclejafconfig.json` file **options.*ruleName*** property.

   ```
   "options" : {
                 "ruleName" : true,
                 ...
               }
   ```

3. Alternatively, to enable appending the rule name for all rules, edit the `oraclejafconfig.json` file **options.verbose** property to enable verbose output mode.

   ```
   "options" : {
                 "verbose" : true,
                 ...
               }
   ```

   You can also enable verbose output each time your run the audit from the command line.

   ```
   ojaf -e ...
   ```

Customize the Presentation of the Audit Messages

Use the optional Oracle JAF configuration file properties **proseFormat** and **lineFormat** to define templates to redefine the presentation of reported audit issues.

If the JAF configuration file **format** property is set to **prose** or to **line**, you can use the respective properties **proseFormat** and **lineFormat** to define a custom presentation template to format the displayed audit issues.

A prose-formatted audit message begins with the text `Audit for`.

```
** Audit for D:/git/trunk/built/apps/components/public_html/content/redwood-loadmorelist/demo.html
[46, 25] blocker : <oj-bind-if> 'test' attribute : read-only '[[...]]' expression expected [JET-0163]
```

Here is a line-formatted audit message with different template.

```
● blocker  :  <oj-bind-if> 'test' attribute : read-only '[[...]]' expression expected  [JET-0163]
   → D:/git/trunk/built/apps/components/public_html/content/redwood-loadmorelist/demo.html: 66:25
```

If you omit **proseFormat**, Oracle JAF uses the default template, `"%pos1 %s : %m"`, which displays as follows:

```
[46, 25] blocker : <oj-bind-if> 'test' attribute : read-only '[[...]]' expression expected
```

Specify the template for the properties as a string, containing any of the following tokens in any order. JAF replaces the tokens in the audit message output at runtime.

Audit Message Tokens	Replacement Value
%l	The line number only.
%c	The column number only.
%pos1	The comma-separated line and column number with square brackets. For example: `[46, 25]`
%pos2	The comma-separated line and column number with parens. For example: `(46, 25)`
%s	The severity level of the processed rule.
%m	The entire audit issue message.
%mid	The full audit message ID. For example, `JET-1234`
%p	The prefix of the rule set. For example, `JET` for the built-in JET rule set.
%n	The audit message number only. For example, `1234`
%r	The rule name.
%f	The file path.

To define a custom presentation template:

1. To enable support for customizing the presentation of audit message, edit the `oraclejafconfig.json` file **format** property to ensure that either **prose** or **line** has been specified. Note that if the **format** property has been omitted, the default specifies prose output.

   ```
   "format" : "prose"
   ```

or

```
"format" : "line"
```

2. If the output format has been specified as prose, edit the `oraclejafconfig.json` file **proseFormat** property to define a custom presentation template to format displayed audit issues. The value of **proseFormat** is a string containing tokens shown in the table above.

   ```
   "proseFormat" : "%pos1 %s : %m [%mid]"
   ```

   In the sample, the tokens specify the first four replacement values for the audit message: including the line/column number format followed by the processed rule's severity level, and then a colon followed by the audit issue message and the full audit message ID within square brackets.

   Output based on this template looks similar to this audit message:

   ```
   ** Audit for D:/myapp/public_html/content/demo.html
   [46,25] blocker : <oj-bind-if> 'test' attribute : read-only '[[...]]'
   expression expected [JET-0163]
   ```

3. If the output format has been specified as line, edit the `oraclejafconfig.json` file **lineFormat** property to define a custom presentation template to format displayed audit issues. The value of **lineFormat** is a string containing tokens shown in the table above.

   ```
   "lineFormat" : "%s  :  %m [%mid]\n → %f: %l:%c"
   ```

   In the sample, the tokens specify the first four replacement values for the audit message: including the processed rule's severity level followed by the audit issue message, a line break, and then an right arrow character, the file path, and then the line and column numbers.

   Output based on this template looks similar to this audit message:

   ```
   blocker : <oj-bind-if> 'test' attribute : read-only '[[...]]' expression
   expected [JET-0163]
   -> D:/myapp/public_html/content/demo.html: 46, 25
   ```

Format a Title for the Audit Report

You can use the optional Oracle JAF configuration **title** property create a title for audit reports.

If the JAF configuration file **format** property is set to **prose** (the default), you can use the **title** property to format a title header to display with the audit output. The title definition may include tokens to insert values, such as the Oracle JET version into the title.

Audit Title Tokens	Replacement Values
$jafdate	The current date, like "Friday Feb 14, 2020".
$jaftime	The current time, like "8:05am EDT".
$jetver	The Oracle JET version, like "8.1.0".
%jafver	The Oracle JAF version, like "2.4.0".

Audit Title Tokens	Replacement Values
%jafconfig	The file path for the Oracle JAF configuration file used in the audit, like "D:\myproject\oraclejafconfig.json".

To configure a title:

1. To format a title for audit reports, edit the `oraclejafconfig.json` file **format** property to ensure that either **prose** has been specified or that the property has been omitted (which specifies prose output is the default).

   ```
   "format" : "prose"
   ```

   Note that if the configuration file output format is set to **json**, the report title is also included in the output. For more information, see Output Audit Messages in JSON Format.

2. To format the report title, edit the `oraclejafconfig.json` file **title** property. Macros are available to insert values such as the Oracle JET version, or date and time into the title strings.

   ```
   "title" : [
              "+------------------------------------------------------------+",
              "|                  Some Title for the Audit                  |",
              "+------------------------------------------------------------+",
              "JET : $jetver  ($jafdate, $jaftime)\n"
             ]
   ```

   This sample specifies the audit report will exclude the message with ID `JET-3020` and exclude the set of messages with IDs like `JET-2000`, `JET-2010`, `JET-2020` and so on.

   ```
   +------------------------------------------------------------+
   |                  Some Title for the Audit                  |
   +------------------------------------------------------------+
   JET : 8.1.0  (Friday Feb 14, 2020, 8:05am EDT)
   ```

Output Audit Messages in JSON Format

Use the optional Oracle JAF configuration file property **format** to specify that the output format of the audit to be a JSON document. You can then process the JSON to create any desired output.

If the JAF configuration file property **format** is set to **json**, you can generate the output of the audit in JSON format. You can direct JAF to output the JSON document to the desired directory by defining the JAF configuration property **outPath**.

The output of the JSON is structured as follows.

JSON Section	Description
reported	An array object for each audited file containing an array of reported issue objects.

JSON Section	Description
summary	An object containing summary data such as the number of issues found or the number of files processed, for example.
run	An object containing run data such as run date/time and JET version number.
descriptions	An optional object containing rule descriptions for the issues in the **reported** section. Must be configured by using the JAF configuration property **ruleDescriptions**, as described below.
fileset	An optional object containing the file set processed by the audit. Must be configured by using the JAF configuration property **addFileList**, as described below.

This is an abbreviated sample of typical JSON output for reported issue.

```
{
  "reported": [
                {
                  "file": "/tests/rules/oj-html-binding-attr/binding-
foreach_FAIL_2.html",
                   "issues": [
                                {
                                  "severity": "minor",
                                  "msg": "Use of attribute 'id' is meaningless
for binding element <oj-bind-for-each>",
                                  "msgId": "JET-0015",
                                  "position": {
                                               "row": 14,
                                               "col": 12,
                                               "start": 467,
                                               "end": 482
                                             },
                                  "rule": "oj-html-binding-attr"
                                }
                            ]
                },
                . . .
              ],
  "summary":  {
                "severities": {
                                "blocker": 0,
                                "severity": 0,
                                "major": 0,
                                "minor": 0,
                                "info": 1
                              },
                "issues": 1,
                "issueFiles": 1,
                "errorFiles": 0,
                "parseErrors": 0,
                "errors": 0,
                "warnings": 1,
```

```
                "asserts": 0,
                "files": 1,
                "urls": 0,
                "rulesActive": 64,
                "rulesFired": 41,
                "hooksFired": 15
              },
  "run": {
           "date": "Tues, 14 April 2020 15:52:39 GMT",
           "jetversion": "8.1.0",
           "title" : "Testing",
           "config": "d:\\runAudit\\testing_config.json"
         }
}
```

To specify JSON as the output format:

1. To format a title for audit reports, edit the `oraclejafconfig.json` file **format** property to specify **json**.

   ```
   "format" : "json"
   ```

2. Optionally, to include custom rule descriptions in the JSON output, edit the `oraclejafconfig.json` file **ruleDescription** property and set the value to **short** or to **long**.

   ```
   "descriptions": {
         "JET-0290": {
                       "long": "...a rule long description.",
                     },
         "JET-3172": {
                       "long": "...a rule long description.",
                     },
         . . .
         }
   ```

   This property takes the following values : **none**, **all**, **short**, or **long** and causes an additional **descriptions** section to appear in the output JSON document.

3. Optionally, to format a title to include in the JSON document, edit the `oraclejafconfig.json` file **title** property.

   ```
   "title" : [
              "+------------------------------------------------------------+",
              "|                  Some Title for the Audit                  |",
              "+------------------------------------------------------------+",
              "JET : $jetver  ($jafdate, $jaftime)\n"
             ]
   ```

   Macros are available to insert values such as the Oracle JET version, or date and time into the title strings. For more information, see Format a Title for the Audit Report.

4. Optionally, to specify the output path for the JSON document, edit the `oraclejafconfig.json` file **outPath** property.

   ```
   "outPath" : "myfolder/myreport.json"
   ```

   Note that alternatively you can omit **outPath** and redirect the output to a file. This will also redirect other information that was written to stdout by JAF.

5. Optionally, to append the file set list to the JSON document, edit the `oraclejafconfig.json` file **addFileList** property.

   ```
   "addFileList" : true
   ```

   This optional property causes an additional JSON section **fileset** to appear in the JSON document as an array of full path name strings. This can be useful when creating custom reports from the output JSON, since it allows access to the full file set that was audited.

Part II

Extend the Oracle JET Audit Framework

Use the API and supporting utility libraries provided by JAF to write user-defined, custom audit rules to extend the JAF built-in rule sets.

Topics:

6

Understand the JAF Audit Engine

The JAF audit engine invokes custom audit rules when Oracle JAF audits an Oracle JET app.

About the JAF Audit Engine

There are two types of custom audit rules that the Oracle JAF audit engine supports: *standard* rules and *hook* rules. The difference is that standard rules are invoked in response to the parsing by JAF of file data and hook rules are invoked in response to phases of the JAF audit lifecycle (for example, at startup, close-down, or when a file is first read).

At runtime, when Oracle JAF performs an audit, each file in the target file set is parsed by the JAF audit engine and an abstract syntax tree (AST) is created. The AST is then walked by JAF and data node events are passed to listener functions that you register in your custom audit rules.

You implement custom audit rules as JavaScript files which the JAF audit engine loads based on a configuration file that you define. At runtime, when the JAF audit engine generates the AST of the target file set, it passes context objects to the loaded rules and triggers the AST node event listeners that you implement in the rule's `.js` file. This allows your audit rule to respond to specific data from the audited file set.

The following reference topics on the audit engine list the available node listener types that you can use to write standard, node audit rules. The listener types correspond to AST data nodes that are specific to the file types of the JET application source.

- Listener Types for HTML and JSON Rules
- Listener Types for CSS Rules
- Listener Types for JavaScript/TypeScript Rules

The JAF audit engine gives your invoked custom audit rules access to a `Rule` context object that it passes to the rule's registered listener so you can test data and execute functionality. At the start of the audit, the audit engine passes a `Register` context object to the entry-point of all rules so you can get information about the audit. For details about these context objects, see these audit engine reference topics.

- Context Object Members Passed to the Register Function
- Context Object Properties Available to Registered Listeners
- Context Object Properties Available to CSS Rule Listeners

For details about the audit engine hook points that you can use to create hook rules, see About Hook Rule Invocation.

Understand the Structure of Custom Audit Rules

The Oracle JET Audit Framework (JAF) can be extended by the addition of custom rules that you implement. A rule is a JavaScript file that exports certain public functions.

When an audit is performed, each file in the target file set is parsed and an abstract syntax tree (AST) is created. The AST is then walked and the nodes are passed to one or more registered

listener functions in the rules. Rules are implemented as JavaScript files and loaded by JAF at runtime as node.js module. JAF passes the loaded rules a context as it analyzes the AST and invokes the rule listeners.

To qualify as a valid rule, a rule must export the following four methods:

Method	Description
getDescription()	Returns a full detailed rule description. The description may contain HTML markup.
getName()	Returns the rule name.
getShortDescription()	Returns a short description/summary of the rule.
register(*Object* **context)**	Called during JAF startup, this is the main entry point in the rule implementation. Declares the type of data that the rule wants to listen for together with the event handler functions. Returns an object that contains the events for the specified parsed AST data types or JAF audit lifecycle phases.

Here is a skeleton outline of a rule that you can implement to audit HTML or JSON files:

Skeleton Rule

```
function getName()
{
   return "my-rule-name" ;
};

function  getShortDescription()
{
   return "This a short description of the rule" ;
};

function  getDescription()
{
   return "This a much more detailed explanation of the rule, and can include
markup." ;
};

function  register(context)
{
    // Here the rule registers the type of data that it wants to listen for,
together with event handler function(s).
};

module.exports = {getName, getDescription, getShortDescription, register};
```

> **Note:**
>
> The rule description returned by `getDescription()` can contain HTML markup.

For the list of available events that your rule can listen for see:

- Listener Types for HTML and JSON Rules

- Listener Types for CSS Rules
- Listener Types for JavaScript/TypeScript Rules

See also:

- Audit Rule Entry Point Method Structure
- Audit Rule Listener Function Structure

> **Tip:**
>
> A skeleton rule can be easily scaffolded in the current directory using
>
> `ojaf --initrule myRuleName`

If preferred, ES6 class syntax can be used.

Returning an ES6 class

```
class Rule {                                  // (name can be anything)
     getName()               {...}
     getDescription()        {...}
     getShortDescription()   {...}
     register(regCtx)        {...}
}

module.exports = Rule;
```

If preferred, the following prototype inheritance format for creating a class is also acceptable, and Oracle JAF will automatically perform a `new` on the function:

Returning a class

```
var anyName = function()  {};  // (name is "internal" and can be anything)
anyName.prototype.getName = function()  {...};
anyName.prototype.getShortDescription = function()  {...};
anyName.prototype.getDescription = function()   {...};
anyName.prototype.register(context) {...};
module.exports = anyName ;
```

Audit Rule Entry Point Method Structure

The audit rule's main entry point is the `register()` function. For node rules that you define, this function is called during JAF startup and declares listener functions for specific types of data found during file set auditing. When you need to define a hook rule, use this function to declare listeners for events triggered by JAF on the audit lifecycle.

In the case of node rules, the basic purpose of a registered entry-point method implementation is to examine the data passed to it and to return one or more `Issue` objects, where each contains a description of the problem found. You can then choose which issues to report by using a `Reporter` instance. If no issues are found, the rule just returns. The method gets its data from the passed-in `Register` context object.

The following pseudo code sample registers an event listener for the registered listener type `ojtag`. The `ojtag` type is an example of one of many listener types that you can register specifically for HTML and JSON files. For more details about the listener function, see Audit Rule Listener Function Structure.

```
function register(regContext)
{

 return {
    ojtag : function(ruleContext, tagName)  // "ojtag" is an example of a
registered type - it causes the
    {                                       // function to be called for each
DOM element of the form <oj-xxx>
      var issue ;


      // analyze the data passed in the Rule context, and any other supplied
args
      . . .

      if (found_a_problem)
      {
        issue =  new  ruleContext.Issue( "describe the problem found" ) ; //
create new Issue object
        ruleContext.reporter.addIssue(issue, ruleContext) ;              //
pass Issue to the Reporter instance
        }
    };
 }
};
```

The `register()` function sample shows that the `Rule` context object provides an `Issue` class which can be used to create an `Issue` instance. The `Issue` instance is then passed to the `Reporter` instance (also available from the context) where you choose one or more issues to report.

> **Tip:**
>
> Generally, it is best to limit the custom audit rule to listen for and to report a single issue per rule. This permits a specific diagnostic to be disabled, if required, in the JAF configuration file.

When the current file has been completely audited, JAF emits the issues in the format that you specified in the JAF configuration file.

Audit Rule Listener Function Structure

Listener functions for audit events are defined in your rule's `register()` function. You can declare listener functions for specific types of data found during auditing of a file to define a node rule. Alternatively, you can declare listener functions for events in the audit engine lifecycle to define hook rules.

The listener function has the following signature for a node-type rule, where some arguments are also properties of the `ruleContext` object.

```
function _fnHandler(ruleContext, arg1, arg2) { . . .
      };
```

In the case of a hook rule, where the registered type is an audit engine signaled event (for example, **endselector**), the arguments are not used.

For node rules, the arguments depend on the registered listener type.

- where *arg1* is a string representing the data node token. For example, if `ruleContext.type` is **ojtag** or **tag**, this would represent a string such as `oj-button` or `div`.
- where *arg2* is a string that is an optional value supplementing *arg1*. For example, if `context.type` is **attr**, this would represent the attribute's value, and *arg1* will contain the attribute name.

> **Note:**
>
> For the complete list of registered listener types and a description of their arguments, see Listener Types for HTML and JSON Rules, Listener Types for JavaScript/TypeScript Rules, and Listener Types for CSS Rules. For the registered types that you can define for hook type rules, see About Hook Rule Invocation.

7

Get Started Writing Custom Audit Rules

You use an Oracle JET application that you want to audit as the project for creating and testing custom audit rules. You can configure the application to run Oracle JAF and test the custom rules against the files of the target application. You can zip the application folder containing the implemented custom audit rules for use by other developers to audit their JET applications as a custom rule pack.

Set up the Custom Audit Rules Test Project

Writing custom audit rules is an iterative development process that ideally starts with an existing Oracle JET application project that you can use to test your custom audit rules against.

Before you start writing custom audit rules, choose an Oracle JET application that contains the actual files that you intend your custom rules to audit. This application will become a kind of development environment for writing and testing of the custom rules. You can then implement custom rules as JavaScript files within a folder that you add to the root of the test application. Once you configure the test application to run Oracle JAF and invoke the audit rules in your custom rules folder, you can easily iterate over the target file set of the test application in a test/debug audit cycle.

> **Tip:**
>
> By default, Oracle JAF audits the application files located in the `src` folder of the JET application. To avoid auditing the source code of your custom audit rules, create the custom rules folder at the root level of your test application.

The custom rules folder that you add to the test application will have the following contents, including the JavaScript (`.js`) files that implement your custom audit rules:

```
rule-1.js        }
rule-2.js         }  these are your custom rule files
. . .            }
rules.json           mandatory file describing the rule properties
msgid.json           optional file associating rules with message ID's
```

The `rules.json` file is a single rules definition file that you must define within the custom rules folder to describe the properties of your custom audit rules. The rules definition file can include comments and has the following structure.

```
/*-------------------------------------------------------------*/
/*  Test 'rulePack' definition                                 */
/*-------------------------------------------------------------*/
{
  "title" :  "A descriptive title for the rule pack",
  "prefix" : "ABCD",                    <--  the prefix prepended to message
ids
```

```
  "version" : "1.1.0",                    <--  the rule pack version
  "rules" : {
              "rule-1" :  {
                 // Standard rule options
                 "severity": "major",
                   // Additional optional user rule options
                     "customOpts": {
                        "maxLevel": 3
                      }
                     },
              "rule-2" :  { ... },
               . . .
            }
}
```

The **prefix** property identifies custom rules as belonging to a common rule set. At runtime, Oracle JAF will prepend the prefix you specify to the message IDs of emitted diagnostic messages. The prefix you specify helps users to identify the invoked audit rules.

Before You Begin:

- Choose an Oracle JET application that you can use to test your custom audit rules against. This application will serve as the project where you will implement custom audit rules.
- Install Oracle JAF from npm. For details, see Install the Oracle JET Audit Framework.

To set up the custom audit rules project:

1. Open a Command Prompt window and run the JAF initialize command from the application root.

   ```
   ojaf --init
   ```

   When you run the command, the tooling will scaffold a default JAF configuration file named `oraclejafconfig.json` at the application root. You will edit this file to configure JAF to run the custom audit rules during testing.

2. In the root of your custom audit rules project, create a folder to contain the custom rules and rule definition file. The folder name can be any name that you choose.
3. Edit the generated `oraclejafconfig.json` file at the root of the application and configure the **rulePacks** property value to point to the custom rules folder.

   ```
   "rulePacks" : [
                   {
                     "path" : "path/to/my/customrulepack/folder",
                     "enabled" : true
                     "status" : "all"
                   }
                 ]
   ```

   The **enabled** and **status** properties are optional and provide the ability to easily disable a complete rule pack or to report only rules of a particular status. If omitted, the default enables and reports all rules in the rule pack.

4. Optionally, disable audit reporting for the built-in JET rule set. Edit the generated `oraclejafconfig.json` file and set the **builtinJetRules** property value to **false**.

   ```
   "builtinJetRules" : false
   ```

   When you want to test only custom audit rules, the **builtinJetRules** property is a convenience property that obviates having to individually disable built-in JET rules to prevent them from running during your test/debug audit cycle.

5. Create the mandatory rule definition file `rules.json` in the custom rules folder that you added to the JET application.

   ```
   {
     "title" :  "My Custom Audit Rules",
     "prefix" : "CUSTOM",
     "version" : "1.0.0",
     ...
   }
   ```

   The prefix you assign will be prefixed to the audit diagnostic messages to help you identify diagnostics that result from your custom audit rules. The title and version are arbitrary and help you identify a rule pack version.

6. In the rule definition file, add the **rules** property with the list of custom audit rules that you will implement in this project and any standard or user-defined property values that you want to pass in the case of configurable audit rules.

   ```
   {
     "title" :  "My Custom Audit Rules",
     "prefix" : "CUSTOM",
     "version" : "1.0.0",
     "rules": {
       "custom-check-heading-levels-1" : {},
       "custom-check-heading-levels-2" : {},
       "custom-check-heading-levels-3": {
            "filetype": "html",
            "customOpts": {
                   "maxLevel": 4
                 }
            }
     }
   }
   ```

   The `rules.json` file defines the rule pack and identifies the audit rules and optionally their configurable properties that JAF will load at runtime for the registered rule pack. By convention, rule names include the rule pack prefix.

   In this sample, the rule name prefix `custom` helps to identify the rules as belonging to the same rule pack. The first two rules declare no runtime properties and the third rule declares a default property value that can be optionally configured by the user in the `oraclejafconfig.json` file of the target application. Additionally, for the list of system properties that you can optionally define for individual audit rules, see Define the Runtime Properties of Custom Audit Rules.

7. Optionally, designate a rule that must not be disabled at runtime by setting the **$required** property to `true`.

```
{
  "title" :  "My Custom Audit Rules",
  "prefix" : "CUSTOM",
  "version" : "1.0.0",
  "rules": {
    "custom-check-heading-levels-1" : {},
    "custom-check-heading-levels-2" : {},
    "custom-check-heading-levels-3": {
         "$required": "true",
         "filetype": "html",
         "customOpts": {
                "maxLevel": 4
              }
         }
  }
}
```

The **ruleMods** configuration property (see Configure Audit Rule Runtime Properties) or, indirectly, the **ruleName** property can disable rules from running. You typically use the **$required** property for rules that perform rulepack setup or other non-audit related functions and whose execution is mandatory. It also ensures that these rules are loaded/registered before all other rules, in the order they are specified.

8. Optionally, create a rule message ID file `msgsid.json` in the custom rules folder that you added to the JET application.

```
{
    "custom-check-heading-levels-1" : "1234",
    "custom-check-heading-levels-2" : "1235",
    "custom-check-heading-levels-3" : "1236"
}
```

When JAF reports an issue, it includes a unique message ID of the format *`prefix-nnnn`*, where *`prefix`* is the prefix of the rule pack and *`nnnn`* is a message number defined for the rule. Alternatively, you can hardcode the message ID in your custom audit rule, as described in Define the Message ID of Custom Audit Rules.

9. You are now ready to begin writing rules that you implement as `.js` files added to the custom rules folder, as described in Implement the Custom Audit Rules.

> **Tip:**
>
> To quickly scaffold a skeleton audit rule, in the current directory run the `ojaf` command with the `--initrule` command line flag.
>
> ```
> ojaf --initrule myRuleName
> ```

For an introduction to audit rule JavaScript, see Understand the Structure of Custom Audit Rules.

As you implement custom audit rules, you'll want to get started testing custom audit rules in your project:

- Reference the Custom Audit Rules in an Audit
- Audit the Application Using the Command Line

Define the Runtime Properties of Custom Audit Rules

Use the **rules** property of the `rules.json` file to declare the rules in a rule pack, including the properties of individual custom audit rules.

All custom audit rules in the rule pack must be declared in the **rules** property of the `rules.json` file. Properties that you can define include standard system properties when you want to override a default value defined by JAF. You can also include optional properties when you want to pass property values to the custom rule at runtime, but these properties must be enclosed in an additional **customOpts** property.

Here is a basic example of a user-defined rule definition:

```
"rules": {
 "my-rule": {
  // standard system properties
  "$required" : "true",
  "severity": "info",
  "filetype": "html",
  // optional properties
  "customOpts": {
    "maxLevel": 1
    }
  }
}
```

This declaration specifies that a custom rule exists that is referred to as "**my-rule**" and that it is implemented in the file `my-rule.js`, in the same folder as the `rules.json` file. It includes a number of standard system properties (**$required**, **severity**, and **filetype**). Additionally it declares the rule-specific property **maxLevel**. This property is not inspected by JAF, and will be passed to the rule in a `Rule` context object when it is invoked. The custom rule implementation handles the passed values to achieve the desired audit result.

Some property names are reserved by JAF and cannot be re-purposed by the custom audit rule. The following JAF system properties are reserved and all properties are optional on the rule declaration. If you do not add these properties to the custom audit rule declaration, JAF will assign a default value. For example, unless you specifically define the **severity** property, the custom audit rule will be associated with the severity level **critical**.

JAF system properties	Description
inservice	Rules are assumed to be in service, unless this property is set to **false**. This setting overrides the **enabled** property and suppresses the use of the configuration **ruleMods** property to attempt to enable the rule. Rules not in service do not participate in an audit. The default value is **true**.
enabled	Enables or disables the custom audit rule. All custom rules are enabled by default.

JAF system properties	Description
severity	Classifies the severity level of the custom audit rule. By default Oracle JAF defines a set of levels that you can assign: **info**, **minor**, **major**, **critical** (default), **blocker**. Use this property to specify the severity of the custom rule so that users can restrict the audit by rule severity level. For example, see Restrict Audit Rule Severity Level.
status	Associates a development status with the custom audit rule. May be **production** (default), **alpha**, **beta**, or **deprecated**.
filetype	Specify the file types for which the custom audit rule will be invoked. By default the custom rule is not restricted to a file type. May be **html** and/or **css**, and/or **js** and/or **json**. For example: `"filetype : "html"` or `"filetype" : ["html", "css"]` The **filetype** property is ignored by custom hook rules declared for **startup**/**closedown** phases, since these are not file related. For all other audit rules, you should specify this property.
group	Specify the group or groups to which the custom audit rule is assigned. Use this property to assign the custom rule to a group of any name so that users can restrict the audit by rule group. For example: `"group" : "jet-html"` or `"group" : [ "jet-html", "jet-aria"]` For example, see Audit with Specific Rules.
jetver	Specifies the Oracle JET release version or versions required to invoke the custom audit rule. It the property is omitted, the custom rule will operate across all JET versions. The format supports semantic versioning, as used in programs like **npm**. For example: `"jetver" : ">=9.1.0"` or `"jetver" : "~9.1.0"` For more information about this property and semantic versioning, see Audit with Specific JET and ECMA Script Versions.
theme	Specify a JET theme if the rule is theme dependent. The value is compared with the configuration property `theme` (or its default), and the rule is disabled if there is no match. Can be specified as a string or an array of theme strings. For example, `"theme": "Redwood"` or `"theme": ["Redwood", "Alta"]`
amd	Specifies that an audit rule cannot be used in AMD mode if the rule performs any I/O. It can be omitted in all other cases. The property is ignored if not running in AMD mode. It is recommended that you set **amd : false** if the rule performs any file I/O to prevent failures for future AMD usage.

JAF system properties	Description
issueTag	Specify a string that is to be passed through to the output Issue object for an issue flagged by this rule. This string is not inspected by JAF, and its value may be encoded as required by the processing routine examining the string in the output JSON (or object if API/AMD mode).
customOpts	Declares an object container for user custom rule option properties. Prior to JAF 10.0.0, custom properties were not permitted to be prefixed by the values shown below, since they were reserved for internal JAF use only, and use by custom written rules was prohibited. This restriction was removed in JAF 5.11.0 when **customOpts** was introduced. Top-level user options were deprecated in 5.11.0, and **customOpts** became mandatory in JAF 10.0.0. Option properties may not be prefixed with the following if not contained in **customOpts**. `$` `jet` `jaf` `oj` `ojc` `jetwc` `jetwco` `vdom` `jetvdom` `spoc` `wdt` `csp`
$required	Designates that this rule cannot be disabled by the configuration property **ruleMods** and, indirectly, by the **ruleNames** property. This is typically used for rules that perform pack set-up or other non-audit related functions. Rules marked **$required** are loaded/registered (in the order found in `rules.json`) ahead of all other rules. Specify **true** to make running of this rule during pack set-up mandatory. The default value is **false**.
$...	Other properties starting with **$** are for internal JAF use only, and may not be referenced in a configuration file.

The custom audit rule's properties may be overridden at runtime by users though the `oraclejafconfig.json` file configuration property **ruleMods**, as described in Configure Audit Rule Runtime Properties . Note also that a rule can be designated as one that must not be disabled at runtime by setting the `$required` property to `true`.

Define the Message ID of Custom Audit Rules

The message ID that Oracle JAF uses to report an issue can be generated by default by JAF or you can optionally define the IDs to better document custom audit rules.

When JAF reports an issue, it includes a unique message ID of the format `ppp-`*`nnnn`*, where `ppp` is the prefix of the rule pack, and *`nnnn`* is the message ID. The custom audit rule can supply the message number in a number of ways.

The message ID can be either hardcoded, or it can be obtained from some user-defined custom mechanism (for example, by using a rule pack extension), and specified in the `Issue` constructor.

Alternatively, you may use the optional `msgid.json` definition file to associate a rule name and message ID within a rule pack. The format of a `msgid.json` file is shown below:

```
{
   "rulename1" : "1234",
   "rulename2" : "1235"
   . . .
}
```

You can annotate a `msgid.json` file with `//` and `/* */` comments.

At runtime, if no ID is specified for an `Issue` when it is added to the `Reporter` instance, JAF will attempt to resolve it by looking for a file named `msgid.json` within the same folder as the rule `.js` files and the mandatory `rules.json` file. In this case, JAF uses the rule name as the message lookup key to obtain the message number

If a `msgid.json` file is used, for flexibility, it is also possible to change the default lookup key from the rule name to a unique key that you specify in your audit rule handler by using `Issue.setMsgKey()`.

```
var issue = new ruleContext.Issue(". . .") ;
. . .
issue.setMsgKey("some key value") ;
```

To hardcode the message ID, your custom audit rule may supply the number (for example, `1234`) directly on the `Issue` object your audit rule handler function creates.

```
var issue = newruleContext.Issue("some rule message", "1234");
```

Your handler function may also set the message ID subsequently on an `Issue` object

```
var issue = newruleContxt.Issue("some rule message");
 . . .
issue.setMsgId("1234");
```

Here is an example of how to obtain the message ID through some custom mechanism. In this example, a custom rule pack extension is used.

```
var RPExtension     = ruleContext.rulePack.getExtension() ;  // get the
rulepack's extension object
var myMsgIdAssigner = RPExtension.assignMsgId ;          // assumes the rule
pack has created a routine for assigning message ID's

var issue = new ruleContext.Issue("some rule message",
myMsgIdAssigner(ruleContext)) ;  // (myMsgIdAssigner could use
ruleContext.ruleName)
```

Refer to Rule Issue Class Methods for a description of the `Issue` constructor and available methods.

Refer to Implement Custom Rules Using the Audit Lifecycle for an example of a custom rule pack extension that you might create for use in a startup hook rule.

Implement the Custom Audit Rules

A custom audit rule is a JavaScript file that you implement and that exports certain public functions.

When you implement custom audit rules in your project, you add a `.js` file with the same name as the rule you declare in the project's `rules.json` file. To illustrate how to implement audit rules, we'll describe three rules of increasing complexity that audit HTML files for excessive levels of HTML heading nesting:

- `custom-check-heading-levels-1.js`
- `custom-check-heading-levels-2.js`
- `custom-check-heading-levels-3.js`

The `rules.json` file for these rules declares the `CUSTOM` rule pack like this:

```
{
  "title": "Example Custom Audits",
  "prefix": "CUSTOM",
  "version": "1.0.0",
  "rules": {
    "custom-check-heading-levels-1" : {},
    "custom-check-heading-levels-2" : {},
    "custom-check-heading-levels-3": {
    "filetype": "html",
    "customOpts": {
        "maxLevel": 4
    }
     }
  }
}
```

The implementation of each rule will use the same JavaScript regular expression to match and extract the numerical part of an HTML heading tag passed from the target audit files. At runtime, JAF processes the HTML files in the JET application and a rule listener that we register in the audit rule passes each HTML tag to an event handler function that our audit rules implement. We will vary the rule handler function implementation to illustrate ways it might use the results of the regular expression matching. Additionally, as the `rules.json` file sample shows, the third rule declaration differs since it defines a default value for the **maxLevel** property. The third audit rule illustrates how to make an audit rule configurable by the end-user of the JAF audit.

Version 1 - Report Heading Levels Greater Than H4

To qualify as a valid rule, a custom audit rule must export the following four methods:

```
/**
  * Copyright (c) 2018, 2022, Oracle and/or its affiliates.
  * Licensed under The Universal Permissive License (UPL), Version 1.0
  * as shown at https://oss.oracle.com/licenses/upl/
  */

/
*--------------------------------------------------------------------------
```

```
---*/
/*   JAF Rule:
'CustomHeadingLevelsAuditBasic'                                        */
/*
Purpose :
*/
/
*-------------------------------------------------------------------------
---*/


const RULENAME = "CustomHeadingLevelsAuditBasic";
const DESCRIPTION = "This rule checks that excessive levels of header nesting
have
                     not been used on HTML pages by raising an error whenever
a
                     heading tag greater than H4 is used";
const SHORT_DESCRIPTION = "'Checks HTML files for any use of tags <h5> and
above";

class Rule {
  getName() {
    return RULENAME;
  }

  getDescription() {
    return DESCRIPTION;
  }

  getShortDescription() {
    return SHORT_DESCRIPTION;
  }

  register(regContext) {
    return ({
      tag: this._doHeaderLevelAudit
    }
    )
  }
. . .
}

module.exports = Rule;
```

The first three methods in our custom audit rule implementation return usage information that you supply about the audit rule. This information will be passed to Oracle JAF whenever the end-user interacts with the `ojaf` command line interface to request additional details about the audit rule that emitted a particular diagnostic message.

The fourth method `register()` is the required entry point to every custom audit rule. This method is called during JAF startup, and you will use it to declare a node listener for specific types of data found during the file set audit. The method returns a context object that contains the events for the specified node listener type. Rules that you write to audit file data are called *node audit rules* because the `register()` method returns node data on the context object created by JAF from the Abstract Syntax Tree (AST) it generates on the target file.

> **Note:**
>
> The `register()` method that you implement in your rule's `.js` file can also declare listeners for events triggered by JAF on the different phases of the JAF audit lifecycle. This set of listener types provides you with hooks into the audit engine and any rules that you write for these hooks do not rely on file data. For more information about writing hook rules, see About Hook Rule Invocation.

In this version of the heading level audit rule, the `register()` method specifies the `tag` listener type to check all HTML tags in the audited file set. To handle events triggered by the processed tags, the rule needs to implement the audit handler function for the `ruleContext` object and other arguments passed into our handler function. Our implementation invokes the `doHeaderLevelAudit` handler function in response to the listener event. In the case of the registered `tag` listener, a `ruleContext` object and a `tagElementName` string get passed in as arguments to our function.

```
/**
  * Copyright (c) 2018, 2022, Oracle and/or its affiliates.
  * Licensed under The Universal Permissive License (UPL), Version 1.0
  * as shown at https://oss.oracle.com/licenses/upl/
  */

. . .
class Rule {

 . . .
  register(regContext) {
    return ({
      tag: this._doHeaderLevelAudit
    }
    )
  }
_doHeaderLevelAudit(ruleContext, tagElementName) {
  . . .
   }
 }

module.exports = Rule;
```

In this version of the heading level audit rule, we hardcode the heading level so the rule reports a heading level that exceeds H4 in the rule diagnostic message. Then in JavaScript we define a regular expression that allows us to match and extract the numerical part of an HTML `<H*>` tag from the passed in `tagElementName` string. If a match is found, we check the number portion extracted by the regular expression to see if it is greater than the hardcoded limit of 4. Finally, our implementation needs to report the issue by creating an instance of the `Issue` object with a diagnostic message and an optional message ID for the audit rule. Then a `Reporter` instance allows us to call `addIssue()` to allow JAF to output the audit results.

> **Note:**
>
> Hardcoding a unique audit rule message ID in your audit rule handler function is one way to document your custom audit rule. The ID you define will appear in the audit output as *ppp*-*nnnn*, where *ppp* is the rule pack prefix and *nnnn* is the message ID. If your audit rule does not define a message ID and one cannot be found in the optional `msgid.json` file, JAF will generate the rule message ID at runtime for you. For more information, see Define the Message ID of Custom Audit Rules.

```
/**
  * Copyright (c) 2018, 2022, Oracle and/or its affiliates.
  * Licensed under The Universal Permissive License (UPL), Version 1.0
  * as shown at https://oss.oracle.com/licenses/upl/
  */

. . .
class Rule {

 . . .
  _doHeaderLevelAudit(ruleContext, tagElementName) {
    //Define a regular expression that will allow us to match extract the
numerical part of an HTML <H*> tag
    const matchHeader = new RegExp(/^[h](\d*)$/, 'i');
    //Check the tag being processed against the Regular Expression
    const matches = tagElementName.match(matchHeader);

    //A not-null result means it's some kind of header tag, so now we check
the number portion extracted by the
    //regular expression to see if it is greater than the hardcoded limit of
4 in this case
    if (matches !== null) {
      const headerLevel = parseInt(matches[1]);
      if (headerLevel > 4) {
        //Report the issue
        const issue = new ruleContext.Issue("Header level 4 exceeded", "001");
        ruleContext.reporter.addIssue(issue, ruleContext, 'minor');
      }
    }
  }
}

module.exports = Rule;
```

Next let's modify this sample rule to improve our rule's diagnostic message.

Version 2 - Include Heading Tag Information in Report

In this sample, our revised heading level audit rule continues to register the `tag` listener type to trigger the `doHeaderLevelAudit` audit handler function. However, in this version we enhance the diagnostic message to include the heading text and heading tags. The audit handler function logic tests node data on the `children.length` and `children.type` properties of the `ruleContext.node` object passed to our handler. If the content is a simple header string, we

assign the node `ruleContext.data` to the variable `headerText`, formatted with the heading tags in `problemHeader` and passed to the `Issue` instance that we create. Finally, the call to `addIssue()` to output the audit result on the `Reporter` object remains unchanged.

> **Tip:**
>
> Test your audit rules in a development tool that can invoke the Oracle JET `ojaf` utility, such as VS Code, to more easily visualize the runtime context object properties and their data.

```
/**
  * Copyright (c) 2018, 2022, Oracle and/or its affiliates.
  * Licensed under The Universal Permissive License (UPL), Version 1.0
  * as shown at https://oss.oracle.com/licenses/upl/
  */

class Rule {

  . . .

  _doHeaderLevelAudit(ruleContext, tagElementName) {
    //Define a regular expression that will allow us to match extract the
numerical part of an HTML <H*> tag
    const matchHeader = new RegExp(/^[h](\d*)$/, 'i');
    //Check the tag being processed against the Regular Expression
    const matches = tagElementName.match(matchHeader);

    //A not-null result means it's some kind of header tag, so now we check
the number portion extracted by the
    //regular expression to see if it is greater than the hardcoded limit of
4 in this case
    if (matches !== null) {
      const headerLevel = parseInt(matches[1]);
      if (headerLevel > 4) {
        //In this enhanced version, before we report the issue let's get the
actual
        //tag information to add to the report
        //Only report the actual content for the simple case though otherwise
use ellipsis
        let headerText = '...';
        if (ruleContext.node.children.length === 1 &&
ruleContext.node.children[0].type === 'text') {
          headerText = ruleContext.node.children[0].data;
        }

        const problemHeader = `<${tagElementName}>${headerText}</$
{tagElementName}>`;
        const issue = new ruleContext.Issue(`Header level 4 exceeded for
element: ${problemHeader}`, "002");
        ruleContext.reporter.addIssue(issue, ruleContext, 'minor');
      }
    }
  }
```

```
}

module.exports = Rule;
```

Notice also that our custom audit rules pass in a severity level as an argument to `addIssue()`. If you do not hardcode the severity level or define the **severity** system property in the rule declaration in your project's `rules.json` file, then JAF will assign the custom audit rule the default severity level **critical**. In our samples, we hardcode the severity level **minor** for all three custom audit rules. For details about **severity** and other system properties that your custom audit rules can define, see Define the Runtime Properties of Custom Audit Rules.

Next let's modify this sample rule to illustrate a configurable audit rule that will allow end-users to configure the audit heading level before they run the audit.

Version 3 - Configure the Audit Rule for a Heading Level

Every rule pack must contain a `rules.json` file with the list of audit rules that Oracle JAF loads at audit startup. If the rule is configurable, then the `rules.json` file specifies the property on the declaration line like this **maxlevel** property we declare in this final version of our heading level audit rule that checks against a configurable heading level.

```
{
    "title": "My Custom Audit Rules",
    "prefix": "CUSTOM",
    "version": "1.0.0",
    "rules": {
        "custom-check-heading-levels-3": {
            "customOpts": {
                "maxLevel": 4
            }
        }
    }
}
```

To use the configurable property **maxLevel**, our audit rule sample calls `getRuleOption()` to query rule pack information on the `Register` context object passed in when the rule pack is loaded at startup. We assign the value to `configuredLevel` and the audit handler function tests the value using the same logic described for the previous version of the rule. If the node data for the heading tag exceeds the configured level, then we report the issue and output the message for the offending heading tag together with the `configuredLevel` value.

```
/**
  * Copyright (c) 2018, 2022, Oracle and/or its affiliates.
  * Licensed under The Universal Permissive License (UPL), Version 1.0
  * as shown at https://oss.oracle.com/licenses/upl/
  */

. . .
class Rule {

. . .

_doHeaderLevelAudit(ruleContext, tagElementName) {
```

```
    //Before we start, in this version, find out what the configured max
level is from
    //the rules.json declaration for our custom rule
    const configuredLevel =
ruleContext.rulePack.getRuleCustomOptions().maxLevel;

    //Define a regular expression that will allow us to match extract the
numerical part of an HTML <H*> tag
    const matchHeader = new RegExp(/^[h](\d*)$/, 'i');
    //Check the tag being processed against the Regular Expression
    const matches = tagElementName.match(matchHeader);

    //A not-null result means it's some kind of header tag, so now we check
the number portion extracted by the
    //regular expression to see if it is greater than the hardcoded limit of
4 in this case
    if (matches !== null) {
      const headerLevel = parseInt(matches[1]);
      //This time we check against the configured level passed in with the
options
      if (headerLevel > configuredLevel) {
        //In this enhanced version, before we report the issue let's get the
actual tag information to add to the report
        //Only report the actual content for the simple case though otherwise
use ellipsis
        let headerText = '...';
        if (ruleContext.node.children.length === 1 &&
ruleContext.node.children[0].type === 'text') {
          headerText = ruleContext.node.children[0].data;
        }

        const problemHeader = `<${tagElementName}>${headerText}</$
{tagElementName}>`;
        const issue = new ruleContext.Issue(`Header level ${configuredLevel}
exceeded for element: ${problemHeader}`, "003");
        ruleContext.reporter.addIssue(issue, ruleContext, 'minor');
      }
    }
  }
}

module.exports = Rule;
```

This concludes our walkthrough of a basic node rule. As an exercise, you may reuse the sample code of the three heading level audit rules to create a custom rule pack to audit the HTML source files of your JET application. The rule pack you create will contain a `.js` implementation file for each audit rule and a `rules.json` file to declare the rules. In each implementation file, be sure to include the required methods shown in the sample described for rule version 1 that for brevity the samples omit in rule versions 2 and 3. For more information about creating a custom rule pack that you can reference in an Oracle JAF audit, see Reference the Custom Audit Rules in an Audit.

> **Note:**
>
> End users register your custom rule pack by editing the `oraclejafconfig.json` file in their JET application to define the **rulePacks** property. They can also define the **ruleMods** property to override default values declared within your custom rule pack rule definitions. For details about how end users enable custom rule packs to audit their JET application, see Audit with Custom Rule Packs, and for details about how end users may override properties of configurable audit rules in their audit runs, see Configure Audit Rule Runtime Properties .

Reference the Custom Audit Rules in an Audit

Use the **rulePacks** property of the `oraclejafconfig.json` file to register the custom audit rules in your project to be loaded by JAF at audit runtime.

An audit rule is a JavaScript file that exports certain public functions. Rules with a common diagnostic purpose, for example, specific to a group of user-defined Web Components, can be placed in a folder and that folder's location referenced in the configuration file. A group of associated rules is referred to in JAF as a rule pack. A rule pack may also be zipped for distribution to other users. The Oracle JAF configuration file will reference the location of the zip file in this case.

The zip file or folder should have the contents as described in Set up the Custom Audit Rules Test Project.

To declare the rule pack, edit the generated `oraclejafconfig.json` file **rulePacks** property to specify the path to the custom rule pack folder or zip file.

```
"rulePacks" : [
                {
                  "path" : "path/to/myrulepack.zip",
                  "enabled" : [true (default) | false]
                  "status" : ["all" (default), "production", "deprecated",
"beta", "alpha"]
                },
                {
                  "path" : "path/to/my/rulepack/folder",
                  "enabled" : [true (default) | false]
                  "status" : ["all" (default), "production", "deprecated",
"beta", "alpha"]
                },
                ...
              ]
```

The **enabled** property is optional and provides the ability to easily disable a complete rule pack. If omitted, the default is enabled.

The specified **path** can be relative. If relative, it is considered to be relative to the location of the configuration file, or the configuration file's **base** property, if defined.

8

Implement Custom Node Rules

Node rules are standard audit rules that you write in response to the parsing of application files, including HTML, JSON, CSS, and JavaScript. Oracle JAF handles file parsing by creating data nodes that the Oracle JAF audit engine walks in the form of an Abstract Syntax Tree (AST) and exposes to you through node event listeners that you can register in your custom node rule.

About AST Rule Nodes in CSS Auditing

Rules that audit CSS files or the **<style>** section of HTML files are implemented as JavaScript/TypeScript files which are loaded at runtime as `node.js` modules, and are passed a context from Oracle JAF as it analyzes the abstract syntax tree (AST) of the audited content and invokes the node type listeners that you have registered with your audit rules.

Overview of Rule Nodes in CSS

Consider the following CSS rule:

```
body,html {
  margin:0; padding:0;
}
```

The CSS rule is represented in the AST as a **Rule** node. Here is a skeleton view of the **Rule** node:

```
{
   "type": "Rule",
    . . .
   "prelude" : {},              // see below
   "block" : {                  // contains the property/value pairs
      "children" : [
              {
                "type" : "Declaration",
                "property" : "margin"
                "value" : {
                     "children" : [
                             {
                               "type" : "number",
                               "value" : "0"
                             }
                      ]
                }
            },
            {
              "type" : "Declaration",
              "property" : "padding"
              "value" : {
                   "children" : [
```

```
                                {
                                  "type" : "number",
                                  "value" : "0"
                                }
                            ]
                        }
                    }
                ]
            }
}
```

From this sample it would be a simple task to extract the property/value pairs from this **Rule** node.

For clarity, some content above has been omitted. For example, throughout the **Rule** node there are **loc** sub-properties which contain positional information:

```
"loc": {
    "source": " ",
       "start": {
           "offset": 18,
           "line": 3,
           "column": 5
       },
       "end": {
           "offset": 26,
           "line": 3,
           "column": 13
       }
}
```

> **Note:**
>
> The **loc** position information is relative to the start of the CSS text. Since CSS may also be embedded in an HTML **<style>**, the rule context provides the **offset** property which provides the actual origin of the text, and that can be used to adjust the position information when reporting an issue. See the **offset** property and helper utility method **CssUtils.getPosition()** description in Context Object Properties Available to CSS Rule Listeners.

In this CSS rule example, the property **prelude** was shown. This contains a *higher* view of the structure of the rule, and introduces node types **SelectorList** and **Selector**. Here is a skeleton example.

```
"prelude": {
    "type": "SelectorList",
    "children": [
        {
          "type": "Selector",
          "children": [
               {
                 "type": "TypeSelector",
```

```
                  "name": "body"
               }
            ]
         },
         {
            "type": "Selector",
            "children": [
               {
                  "type": "TypeSelector",
                  "name": "html"
               }
            ]
         }
      ]
}
```

In the sample above, the **type** property has value **TypeSelector** since it refers to the elements **<body>** and **<html>**. For other selector types, **ClassSelector**, **IdSelector**, and **PsuedoSelector** are used. Note that **SelectorList** contains two **Selector** nodes; this is because **body** and **html** were *grouped* in the CSS using a *comma*. A more detailed discussion of the **SelectorList** node can be found below.

Overview of the SelectorList Node

In the sample above, the **SelectorList** property of the **prelude** node was introduced for a simple case using *grouping*. In that example, the **SelectorList** contains two **Selector** nodes of type **TypeSelector**. There were two Selector nodes generated because of the use of the *grouping comma*. This section goes into greater depth when *combinators* and *pseudo* selectors are used. When selectors are combined, only one compound **Selector** is generated and contains multiple child nodes.

Combinator Examples

Consider the following:

```
.foo.bar  { ... }
```

This will produce a skeleton **SelectorList** and **Selector** node as follows. Note that the **Selector** node contains two child **ClassSelector** nodes:

```
"prelude": {
  "type": "SelectorList",
  "children": [
     {
        "type": "Selector",
        "children": [
           {
              "type": "ClassSelector",
              "name": "foo"
           },
           {
              "type": "ClassSelector",
              "name": "bar"
           }
        ]
     }
```

```
  ]
}
```

Consider the following:

```
.foo .bar {...}
```

This will produce a skeleton **SelectorList** and **Selector** node as follows:

```
"prelude" : {
  "type": "SelectorList",
  "children": [
     {
        "type": "Selector",
        "children": [
           {
             "type": "ClassSelector",
             "name": "foo"
           },
           {
             "type": "WhiteSpace",
             "value": " "
           },
           {
             "type": "ClassSelector",
             "name": "h2"
           }
        ]
     }
  ]
}
```

Consider the following:

```
div > p  { ... }
```

This generates the following **SelectorList** node:

```
"prelude" : {
  "type": "SelectorList",
  "children": [
     {
        "type": "Selector",
        "children": [
           {
             "type": "TypeSelector",
             "name": "div"
           },
           {
             "type": "Combinator",
             "name": ">"
           },
           {
```

```
            "type": "TypeSelector",
            "name": "p"
          }
        ]
      }
  ]
}
```

Note that a **Combinator** node appears between the two type selectors, per the CSS.

Consider this slightly more complex example using an attribute selector:

```
a[href^="https"]  { ... }
```

This generates the following **SelectorList** node as follows:

```
"prelude": {
   "type": "SelectorList",
   "children": [
      {
        "type": "Selector",
        "children": [
           {
             "type": "TypeSelector",
             "name": "a"
           },
           {
             "type": "AttributeSelector",
             "name": {
                "type": "Identifier",
                "name": "href"
             },
             "matcher": "^=",
             "value": {
                "type": "String",
                "value": "\"https\""
             }
           }
        ]
      }
   ]
}
```

In the sample above, an **AttributeSelector** node has been generated with a **matcher** property.

Pseudo Class Selector Examples

Consider the following:

```
.foo:focus  { . . . }
```

This will produce a skeleton **SelectorList** and **Selector** node as follows:

```
"prelude": {
  "type": "SelectorList",
  "children": [
     {
        "type": "Selector",
        "children": [
           {
             "type": "ClassSelector",
             "name": "foo"
           },
           {
             "type": "PseudoClassSelector",
             "name": "focus"
           }
        ]
     }
  ]
}
```

Note that the **Selector** node reflects the class selector followed by the pseudo class selector.

Consider the following:

```
p:nth-last-child(2) {}
```

This generates a more complex **Selector** node as follows:

```
"prelude": {
   "type": "SelectorList",
   "children": [
      {
        "type": "Selector",
        "children": [
           {
             "type": "TypeSelector",
             "name": "p"
           },
           {
             "type": "PseudoClassSelector",
             "name": "nth-last-child",
             "children": [
                {
                   "type": "Nth",
                   "nth": {
                      "type": "AnPlusB",
                      "a": null,
                      "b": "2"
                   }
                }
             ]
           }
        ]
      }
```

```
    ]
}
```

Note that the **PseudoClassSelector** now has an expanded **children** node.

Walkthrough of Sample HTML and JSON Audit Rules

Rules that audit HTML or JSON files are passed a context from Oracle JAF as it analyzes the abstract syntax tree (AST) of the audited file and invokes the node type listeners that you have registered with your HTML/JSON audit rules.

In this walkthrough, the first audit rule shows how easy it is to get started writing a rule that audits HTML. Subsequent rule samples illustrate greater complexity and the power of Oracle JAF for writing custom rules. Overall, Oracle JAF gives you the ability to look forwards or backwards within a file from the current position, and the various JAF utility functions that are available simplify the task of writing a rule.

> **Note:**
>
> For clarity, the samples in this section omit `getName()`, `getDescription()`, and `getShortDescription()` methods. To understand the basics of node rule implementation, see Understand the Structure of Custom Audit Rules.

Version 1 - Validating id attributes

In this simple introductory rule, the requirement is to inspect all element **id** attributes to ensure that they begin with a common prefix (`acv-`) for the project.

```
... // for clarity, the getName(), getDescription(), and
getShortDescription() methods have been omitted

function register(regContext)
{
  return { attr : _fnAttrs };
};

function _fnAttrs(ruleContext, attrName, attrValue)
{
  let issue;

  if ((attrName === "id") && (! attrValue.startsWith("acv-")))
  {
    issue = new ruleContext.Issue(`'id' attribute ('${attrValue}') is not
prefixed with project prefix \"acv\"`);
    ruleContext.reporter.addIssue(issue, ruleContext);
  }
};
```

Version 2 - Validating id attributes

In general, you can look into the context for additional information, so let's assume that for this rule, you only want to look at particular project files in the file set that begin with `ACV`. The

`ruleContext` object has the member **filepath** that you can use. Note that **filepath** always uses forward slashes, regardless of the platform, so the test for /`ACV` will succeed on all platforms.

```
function _fnAttrs(ruleContext, attrName, attrValue)
{
   let issue;

   if (ruleContext.filepath.includes("/ACV") && (attrName === "id") && (!
attrValue.startsWith("acv-")))
   {
     issue = new ruleContext.Issue(`'id' attribute ('${attrValue}') is not
prefixed with project prefix \"acv\"`);
     ruleContext.reporter.addIssue(issue, ruleContext);
   }
};
```

Version 3 - Validating id attributes

How can the rule be improved? Because JAF is very efficient at file processing, you could seek to improve performance if very large numbers of files are involved. To do that, let's use the context object **node** property, and the **attribs** property of the node. The **node** property is the current node in the file, so you can navigate forwards or backwards from it. Secondly, from the performance aspect, you can reduce the number of invocations of the rule by only listening for HTML elements instead of attributes. Let's assume that, on average, the DOM elements have 5 attributes, then you would reduce the number of rule invocations by 80%. In this version of the rule, the attributes of each element are examined directly.

```
function register(regContext)
{
  // Listen for DOM elements instead of attribute
  return { tag : _fnTags };
};

function _fnTags(ruleContext, tagName)
{
  // Look at the element's attributes
  let attribs, attrValue, issue;

  // 'attribs' is an object of attribute name/value properties for the tag
  attribs   = ruleContext.node.attribs;
  // Get the 'id' value if it exists
  attrValue = attribs.id;

  if (attrValue && (! attrValue.startsWith("acv-")))
  {
    issue = new ruleContext.Issue(`'id' attribute ('${attrValue}') is not
prefixed with project prefix \"acv\"`);
    ruleContext.reporter.addIssue(issue, ruleContext);
  }
};
```

Version 4 - Validating id attributes

At this point, it is worth noting that the `ruleContext` object provides access to **DomUtils**, a collection of useful DOM utility functions. For example, the function **_fnTags()** in the above example could be rewritten as follows.

```
function _fnTags(ruleContext, tagName)
{
  let attrValue, issue;

  // Returns the 'id' attribute's value if found
  attrValue = ruleContext.DomUtils.getAttribValue(context.node, "id");
  if (attrValue && (! attrValue.startsWith("acv-")))
  {
    issue = new ruleContext.Issue(`'id' attribute ('${attrValue}') is not
prefixed with project prefix \"acv\"`);
    ruleContext.reporter.addIssue(issue, ruleContext);
  }
};
```

Version 5 - Validating id attributes

While JAF is efficient, audit rules can always be improved upon. To listen for a file invocation, the rule must register a listener for the **file** type.

> **Note:**
>
> It is necessary to understand that performance and rule complexity/maintainability is a tradeoff. For example, it is possible to reduce this rule's invocation count to once per file by converting the rule to a hook type rule, as described by Implement Custom Hook Rules. Essentially, hook rules make it possible to request that a rule to be invoked (once only) when the file is first read, and prior to any other rules. This means that the rule must then examine the parsed nodes looking for elements and their attributes.

```
function register(regContext)
{
   // Listen for files instead of elements or attributes
   return { file : _fnFiles };
};

function _fnFiles(ruleContext)
{
  let tagNodes, node, attrValue, i;
  const DomUtils = ruleContext.utils.DomUtils;

  // Get elem nodes only (ignore text, comments, directives, etc)
  tagNodes = DomUtils.getElems() ;
  for (i = 0; i < tagNodes.length; i++)
  {
    node = tagNodes[i] ;
    // Get the id" attribute value
```

```
    attrValue = DomUtils.getAttribValue(node, "id");
    if (attrValue && (! attrValue.startsWith("acv-")))
    {
       issue = new ruleContext.Issue(`'id' attribute ('${attrValue}') is not
prefixed with project prefix \"acv\"`);
       ruleContext.reporter.addIssue(issue, ruleContext);
    }
  }
};
```

Walkthrough of a Sample CSS Audit Rule

Rules that audit CSS files or the **<style>** section of HTML files are passed a context from Oracle JAF as it analyzes the abstract syntax tree (AST) of the audited file and invokes the node type listeners that you have registered with your CSS audit rules.

The CSS in this CSS rule walkthrough is as follows.

```
p ...  {
          color : "#112233",
          ...
        }
```

Note that **p** can be decorated with additional CSS syntax, and the audit rule must ignore any such decoration.

The audit rule starts by listening for CSS rules and then looks for a **p** type selector. For more information about determining a type selector, see About AST Rule Nodes in CSS Auditing.

Here is the basic framework for the audit rule for CSS:

```
var  CssUtils ;

function  register(regCtx)
{
   // See setPosition() below
   CssUtils.regCtx.utils.CssUtils;
   return { "css-rule" : _onRule }
};


function  _onRule(ruleCtx, rule)
{
   // If the rule has a p type selector
   if (_hasParaTypeSelector(rule))
   {
      // and the rule sets the 'color' property
      let loc = _getColorPropPos(rule);
      if (loc)
      {
         // report the issue.
         _emitIssue(ruleCtx, loc);
      }
   }
};
```

```
function  _emitIssue(ruleCtx, loc)
{
   var issue = new ruleCtx.Issue("p type selector must not override the
'color' property");

   issue.setPosition(CssUtils.getPosition(ruleCtx, loc));
   ruleCtx.reporter.addIssue(issue, ruleCtx);
};
```

The next step is to analyze the rule node to find the **p** type selectors:

```
function  _hasParaTypeSelector(rule)
{
  var sels, sel, a, ch, i, j;

  if (rule.prelude.type === "SelectorList")
  {
    a = rule.prelude.children;

    for (i = 0; i < a.length; i++)
    {
       sels = a[i];
       if (sels.type === "Selector")
       {
         ch = sels.children;
         for (j = 0; j < ch.length; j++)
         {
           sel = ch[j];
           if (sel.type === "TypeSelector" && sel.name === "p")
           {
             return true;
           }
         }
       }
    }
  }
};
```

Finally, we need to search the rule to see if it specifies the **color** property:

```
function  _getColorPropPos(rule)
{
   var block, decl, i;

   // Process the rule's block of property/value pairs
   block = rule.block.children;
   for (i = 0; i < block.length; i++)
   {
      decl = block[i];
      if (decl.type === "Declaration" && decl.property === "color")
      {
         // Return the 'color' property position for the Issue
         return decl.loc;
```

```
        }
    }
};
```

Walkthrough of a Sample Markdown Audit Rule

Oracle JAF parses a Markdown file into an abstract syntax tree (AST). This AST is subsequently analyzed, and the summarized data objects are presented to rules via their registered rule listeners.

For Markdown processing, a rule can listen for file events such as when a .md file is first read, or for specific Markdown events such as when a particular type of markup is found.

For a list of Markdown rule listeners and a description of their arguments, see Listener Types for Markdown Rules.

In either case, when a rule listener is invoked, it is passed a `context` object. In addition to the many properties available on the `context` object for all rule types (see Context Object Properties Available to Registered Listeners), `context` contains the supplementary data property **suppData**, which is of particular interest when auditing a Markdown file. The property provides easy access to summarized data (links, images, paragraphs, headings, code blocks, etc.) through methods available on its `utils` object. For more information, see Context Object Properties Available to Markdown Rule Listeners.

> **Note:**
>
> Use of the file event in conjunction with the utility methods available via `suppData.utils` will be the most straightforward approach to accessing summarized data, rather than walking the AST, since all the summarized data will be available at that point and the AST will not need to be inspected.

Listen for Markdown Events

The Markdown events are of the form **md-xxxxx**, where **xxxxx** represents the type of Markdown data required (e.g., **md-link** for link events, used in the following rule class).

```
register(regCtx)
{
   return {
       // want markup URL references
       md-link : this._onLink,
         . . .
    }
}

_onLink(ruleCtx, link)
{
   // Process the link object passed as the second argument
   . . .

   // Access to all the parsed data is also available
   // through the supplementary data object
   var utils  = ruleCtx.suppData.utils ;
   // Returns an array of image objects
```

```
   var images = utils.getImages() ;

   // Process the links array
   . . .
}
```

Note how the supplementary data property **suppData** is used to get the Markdown **utils** object. This is then used to acquire the image data from the markup text.

A file hook can also be used, and this permits a rule to access all of the Markdown data at the same time.

```
register(regCtx)
{
   return {
             file : this._onFile,      // Listen for files being read
             . . .
          }
}

_onFile(ruleCtx)
{
   var utils, links, images, paras ;

   utils  = ruleCtx.suppData.utils ;
   links  = utils.getLinks() ;         // Array of link objects
   images = utils.getImages() ;        // Array of image objects
   paras  = utils.getParas() ;         // Array of paragraph objects

   // Process the links, images, and paragraphs
   . . .
}
```

Example Rules

The following rule checks that the first paragraph of Markdown contains a copyright.

```
class  Rule
{
  // For clarity, getRule(), getDescription(), and getShortDescription() have
been omitted.

  register(regCtx)
  {
    return { file : _onFile }
  }

 // file hook listener
  _onFile(ruleCtx)
  {
    var  utils  = ruleCtx.suppData.utils ;
    var  paras  = utils.getParas() ;

    // Get the first paragraph.
    var para = paras[0] ;
```

```
    if (! /[Oo]racle/.test(para.text))
    {
      let issue = new ruleCtx.Issue("Copyright must be declared in first
paragraph") ;

      // Supply start and end indices so that the paragraph can be
highlighted.
      // JAF will compute the line/col from the start index
      issue.setPosition(null, null, paras.pos[0].start,
paras.pos[1].end);
      ruleCXtx.reporter.addIssue(issue, ruleCtx) ;
    }
  }

}
```

The following rule finds all references to URLs containing "Oracle".

```
class Rule {
  // for clarity getRule(), getDescription(), and getShortDescriptiomn() have
been omitted

  // listen for files
  register(regCtx) {
    return { file: _onFile }
  }
  // file hook listener
  _onFile(ruleCtx)
  {
    var utils = ruleCtx.suppData.utils;
    var links = utils.getLinks();
    var refLinks = utils.getRefLinks();
    var images = utils.getImages();

    // URL's are found in inline-links, inline-images, and reference links

    // Inspect inline links
    links.forEach((link) => {
      if (_checkUrl(link.inline)) {
        // found URL containing 'Oracle'
      }
    });

    // Inspect reference links
    for (const ref in refLinks) {
      link = refLinks[ref]
      if (_checkUrl(link, true))
      {
        // found URL containing 'Oracle'
      }
    }

    // Inspect inline-image links
    images.forEach((link) => {
```

```
      if (_checkUrl(link.inline)) {
        // found URL containing 'Oracle'
      }
    });

  }

  _checkUrl(link, refLink) {
    return (link.inline || refLink) ? link.link.includes("Oracle") : false;
  }

};
```

Walkthrough of a Sample JavaScript/TypeScript Audit Rule

Rules that audit JavaScript/TypeScript files are passed a context from Oracle JAF as it analyzes the abstract syntax tree (AST) of the audited file and invokes the node type listeners that you have registered with your JavaScript/TypeScript audit rules.

A JavaScript/TypeScript file is parsed by JAF into an Abstract Syntax Tree (AST), and as the tree is subsequently walked, any node with a type registered by a rule is passed to the rule in **context.node**. The **node.type** property (string) specifies what the node represents. For example, a **node.type** of **AssignmentExpression** indicates a typical statement form such as `myVariable = 42`. As an example, in JavaScript, the portion of the AST representing this statement is:

```
myVariable = 42;
```

The above statement parses into the following node, where some additional properties have been removed for clarity:

```
{
   "type":  "ExpressionStatement",
   "expression":  {
                     "type":  "AssignmentExpression",
                     "operator": "=",
                     "left":   {
                                 "type": "Identifier",
                                 "name":  "myVariable",
                               },
                     "right":  {
                                  "type": "Literal",
                                  "value": 42,
                                  "rawValue": 42,
                                },
                  },
}
```

Thus a simple rule to flag number assignments to variables that are greater than 42, could be:

```
function register(regContext)
{
   return {
             AssignmentExpression : _fnAssign
```

```
          };
};

function _fnAssign(ruleCtx, node)
{
  if (node.left && (node.left.type === "Identifier"))
  {
    if (node.right && (node.right.type === "Literal") &&
(parseInt(node.right.value) > 42))
    {
       let issue = new ruleCtx.Issue(`${node.left.name} assignment is greater
than 42`);
       ruleCtx.reporter.addIssue(issue, ruleCtx);
    }
  }
};
```

> **Tip:**
>
> When writing a JavaScript rule, it is helpful to be able to look at the syntax tree for the particular case being audited. The AST Explorer tool can be very helpful by allowing you to generate syntax trees for arbitrary pieces of JavaScript.

Walkthrough of a Sample Virtual DOM TSX Audit Rule

Rules that audit virtual DOM TSX files are passed a context from Oracle JAF as it analyzes the abstract syntax tree (AST) of the audited file and invokes the node type listeners that you have registered with your audit rules.

User interface markup can be declared directly in a TSX file, as in the following example where a variable declaration references UI markup:

```
const elemTitle = (
 <div class="foo">
  <h1>Title</h1>
   <h2>sub-Title</h2>
  </div>
 );
```

When Oracle JAF parses the TSX file that contains the previous declaration, it creates a section of the AST for the initializer of the variable declaration as a collection of **JSXElement** nodes (together with other non-JSX related nodes). This section can be tedious to process, and for many TSX rules where tags and attributes or properties are being analyzed, a boilerplate and simpler view of the node structure is more convenient. To assist, Oracle JAF gathers related markup entries in the AST into objects of type **TsxComponent** and **TsxProperty** and aggregates them into a structure of type **TsxRenderComponent**. Additional registered listener types are available to deliver these TSX objects to rules. The AST nodes are also available to rule listeners. The TSX objects supply a synopsis structure for

examination and processing of the renderable content, as the following skeleton view of the `elemTitle` variable as a **TsxRenderComponent** demonstrates.

```
{
  "type" : "TsxRenderComponent,
  "components" : [
    {
      "type": "TsxComponent",
      "name": "div",
      "properties": [
        {
          "type": "TsxProperty",
          "name": "class",
          "valueRaw": "\"Foo\""
        }
      ],
      "children": [
        {
          "type": "TsxComponent",
          "name": "h1",
          "valueRaw": "<h1>Title</h1>"
        },
        {
          "type": "TsxComponent",
          "name": "h2",
          "valueRaw": "<h2>Title</h2>"
        }
      ]
    }
  ]
}
```

This structure contains boilerplate information that many TSX rules need for examination of the markup, and avoids direct processing of numerous AST nodes which, even for the above case, is more complex.

If, for example, you want a rule to check that none of the elements declared in a TSX file are deprecated, the following skeleton code could be used which iterates over the **TsxRenderComponent** structure.

```
class Rule {

  getName() {
    return RULENAME;
  }

  getDescription() {
    return DESCRIPTION;
  }

  getShortDescription() {
    return SHORT_DESCRIPTION;
  }

  /**
```

```
    * Registration - declare listeners
    * @param {Object}   regCtx  the registration context
    */

  register(regCtx) {
    return { "TsxRenderComponent": this._onMarkup }
  }

  /**
    *  Process markup for deprecated tags
    *  @param {Object}  ruleCtx  the rule context
    *  @param {Object}  node     the TsxRenderComponent node (see above)
    */
  _onMarkup(ruleCtx, node) {
    node.tags.forEach(tsxcomponent) => {
      // recursive check for tags and children
      this._checkDeprecatedTags(ruleCtx, tsxComponent);
    }
  }


  /**
    *  Recursively check a tag and its children
    *  @param {Object}        ruleCtx   the rule context
    *  @param {Tsxcomponent}  tsxComp   a TsxComponent node
    */
  _checkDeprecatedTags(ruleCtx, tsxComp) {
    var compName = tsxComp.name;
    if (tsxComponent.isWCTag(compName) &&
ruleCtx.libs.metaLib.isTagDeprecated(compName)) {
      let issue = new ruleCtx.Issue(`Tag <${compName}> is deprecated . . .`);
      ruleCtx.utils.tsxUtils.setIssuePosition(issue, tsxComp);
      ruleCtx.reporter.addIssue(issue, ruleCtx);
    }

    if (tsxComp.children) {
      tsxComp.children.forEach((child) =>
{ this._checkDeprecatedTags(ruleCtx, child); })
     }
  }

}

module.exports = Rule;
```

Report Position Information in an Issue for a TSX Audit

Oracle JAF automatically adds position information to a new issue based on the information in the node presented to the listener function, but in the case of the TsxRenderComponent object, the structure can aggregate multiple component and property nodes.

When each of these tags is extracted and made the subject of a new issue, the position information in that node should be used rather than letting Oracle JAF use the default position

information for the **TsxRenderComponent** as a whole. The following example processes the **TsxRenderComponent** aggregate.

```
register(regCtx)
{
    this._tsxUtils = regCtx.utils.tsxLib;
    return { "TsxRenderComponent": _onTsxRC };
}

/**
  * Listener for TsxRenderComponents
  * @param {Object} ruleCtx  the rule context
  * @param {Object} tsxRC     the TsxRenderComponent node
  */

_onTsxRC(ruleCtx, tsxRC)
{
    // Process the tags
    tsxRC.components.forEach((tsxComponent) => {
        this._checkSomething(ruleCtx, tsxComponent);
    });
}

// Recursively extract the tags
_checkSomething(ruleCtx, tsxComponent)
{
    if (this._isThereAnIssue((tsxComponent)
  {
        this._emitIssue(ruleCtx, tsxComponent);
    }

  // Check the component's children
  if (tsxComponent.children) {
            tsxcomponent.children.forEach((child) => {
                if (this._isThereAnIssue(child)) {
                    this._emitIssue(ruleCtx, child);
                }
            });
        }
   }
}

// Report an issue
_emitIssue(ruleCtx, tsxComponent)
{
    var issue = new ruleCtx.Issue(`<${tsxComponent.name}> . . . `);
    this._tsxUtils.setIssuePosition(issue, tsxComponent);
    ruleCtx.reporter.addIssue(issue, ruleCtx);
}
```

9

Implement Custom Hook Rules

Hook rules are effectively hooks into the Oracle JAF audit engine, and are called at specific phases of an audit, as opposed to a response to parsed file data for standard, node rules.

About Hook Rule Invocation

Rules that you write in response to specific phases in the lifecycle of the Oracle JAF audit are called hook rules. Unlike node rules, hook rules are not invoked in response to parsing of application file data.

Rules that you register with a type of **startup**, **closedown**, **startupRP**, **startaudit**, **endaudit**, **closedownRP**, or **file** correspond to specific phases in the lifecycle of the JAF audit. The lifecycle of the JAF audit is described below as an aid to understanding the sequence of hook rule invocation.

1. **Startup Phase**

   Initialization, configuration analysis, and file set expansion are performed.

2. **Rule Packs Loaded**

   Rules are loaded (instantiated) and evaluated by the audit engine:

   → first `register()` is called on all rules in a rule pack to execute rule listeners.

3. **Pre-Audit Phase**

   → **startup** hook rules are fired.

   → **startupRP** hook rules are fired for each rule pack.

4. **Audit Phase**

   → **startaudit** hook rules are fired for each file in the file set.

   {

   → **file** hook rules are fired.

   → now non-hook (node) rules are fired.

   }

5. **Closedown Phase**

   → **endaudit** hook rules are fired.

   → **closedownRP** hook rules are fired for each rule pack.

   → **closedown** hook rules are fired.

Implement Custom Rules on the File Context

Use a Oracle JAF audit engine **file** hook when you need to create an audit rule that is invoked for each file before any other rules are fired for that file or after all rules have been fired for the file.

A file-type hook rule has the ability to conditionally terminate the audit of the file that it is invoked on. If the rule returns a boolean `false`, the file audit will be vetoed. Note that returning `true` is the same as omitting the return statement and the audit will continue.

The **context.filename** property contains the full file path to the file for which the hook rule was invoked. The file path for the file is always maintained with forward slashes regardless of the platform.

The **context.node** property specifies the first node of the DOM (for HTML and CSS) or JavaScript AST or JSON AST.

Hook Listener Type	When Invoked
file	Invoked after a file has been read and before any non-hook rules are fired.
endfile	Invoked after all non-hook rules have been fired for the file.
startscript	Invoked after an embedded JavaScript `<script>` has been read and before any non-hook rules are fired.
endscript	Invoked after all non-hook rules have been fired for the embedded JavaScript text.

Implement Custom Rules Using the Audit Lifecycle

Use any of the various Oracle JAF audit lifecycle hooks when you need to create an audit rule that is invoked during a specific phase of the audit lifecycle.

Hook Rules for startup, closedown, startaudit, and endaudit Phases

The **startup** rules are called after completion of audit initialization and before any data files are read, and the **closedown** rules just prior to audit completion and after all files have been audited. The register context is passed to the rule's `register()` listener method(s). These rules may be used to initialize user data or load user support packages (but see also the **startupRP** hook).

If a rule registers **startup**, **closedown**, **startaudit**, or **endaudit**, the **context.phase** property value will reflect the respective value.

The **startaudit** rules are called after all **startupRP** rules have been fired on all the enabled rule packs, and just prior to the auditing of the file set. The **context.phase** property will reflect **startaudit**.

The **endaudit** rules are called on completion of the file set audit, and prior to the firing of the **closedownRP** rules. The **context.phase** property will reflect **endaudit**.

The **startup** and **startaudit** hook rules have the ability to conditionally terminate an audit. If the rule returns a boolean `false`, the audit is vetoed. Note that returning `true` is the same as omitting the return statement, and the audit will continue.

Hook Rules for startupRP and closedownRP Phases

These rules are called just before and just after the file auditing phase of the lifecycle. The **startupRP** hooks are called after any **startup** hooks have been fired, and the **closedownRP** hooks are called before the final closedown hooks are fired. These registered listeners are typically defined once in a rule pack rule to permit startup activity, such as initialization of common rule pack data, and, if required, closedown of the rule pack data.

```
function  register(regCtx)
{
   return {
              startupRP   : _fn,
              closedownRP : _fn
           };
}

function _fn(ruleCtx)
{
    // ruleCtx.phase contains "startupRP" or "closedownRP"
}
```

The rule pack hooks are invoked as follows.

Hook Listener Type	When Invoked
startupRP	After completion of audit initialization and after the **startup** hook is fired, and before any data files are read. The rule is called once for each enabled rule pack (configuration file property **rulePack**). This rule may be used to initialize any user data or to load custom user support packages/services (referred to as a *rule pack extension*) needed by the rules, and may be associated with the rule pack by passing back the created extension by calling `context.rulePack.setExtension()`. `// Here is a skeleton rule pack extention:` `context.utils.setExtension({` `lib : get_some_lib(),` `package : require('some package'),` `table :  [ . . . ]` `. . .` `});` Any standard rule in the rule pack may retrieve the extension declared for the rule pack by calling `context.rulePack.getExtension()`. A **startupRP** hook rule has the ability to conditionally terminate an audit. If it returns a boolean `false`, the audit is vetoed. Note that returning `true` is the same as omitting the return statement and the audit will continue. If execution of the rule is mandatory and the pack cannot tolerate this rule being disabled by the user configuration file, the **$required** property should be set to true in the pack's `rules.json` file for the rule.
closedownRP	After all files have been audited and before the **closedown** hook is fired. This complementary rule to **startupRP** is called once for each enabled rule pack, and may be optionally used to handle any necessary close-down of custom support services generated via **startedRP**. If execution of the rule is mandatory and the pack cannot tolerate this rule being disabled by the user configuration file, the **$required** property should be set to true in the pack's `rules.json` file for the rule.

Walkthrough of a Sample Audit Hook Rule

Audit rules that register with a listener type of **startup**, **closedown**, **startupRP**, **startaudit**, **endaudit**, **closedownRP**, or **file**, are called at specific phases in the Oracle JAF audit engine lifecycle. These rules are distinct from node-type rules which are called in response to parsed file data.

This walkthrough demonstrates a simple use of two audit rules that work together to classify the usage of `<oj-xxx>` elements in the HTML files of an application. The first rule is a standard HTML data node rule that makes use of a rulepack extension object to save the number of references to the various `<oj-xxx>` elements across all audited files. The second rule uses the audit engine **startupRP** hook to set up the counters in a rulepack extension object, and uses the **closedownRP** hook to display the accumulated element counts on the console upon completion of the audit.

First, we create the node rule `ojtag-counter` to maintain the count of `<oj-xxx>` element usages by registering the **ojtag** event listener on the parsed HTML files. To maintain the count, this rule relies on the `tagStats` rulepack extension object that you'll need to create just before file auditing begins by using a hook rule.

```
... // for clarity, the getName(), getDescription(), and
getShortDescription() methods have been omitted

function register(regContext)
 {
   return { ojtag : function(ruleCtx, tagName)
                    {
                      let tagStats = ruleCtx.rulePack.getExtension().tagStats;
                      if (tagStats[tagName] === undefined)
                      {
                        // first use - create an entry for the tag
                        tagStats[tagName] = 0;
                      }
                      tagStats[tagName]++;
                    }
          };
};
```

Then our hook rule `ojtag-count-display` sets up the count at the start of the audit and displays the accumulated stats to the console at the end of the audit. This hook rule relies on the **startupRP** hook to create the rulepack extension object and **closedownRP** to display the accumulated count. These two hooks are triggered just before and just after the file auditing phase of the audit engine lifecycle.

```
... // for clarity, the getName(), getDescription(), and
getShortDescription() methods have been omitted

function register(regCtx)
{
   return { startupRP : function(ruleCtx)
                        {
                            // Get or create an extension
                            let RPExt = ruleCtx.rulePack.getExtension() || {};
                            // Add a stats collection object to it
```

```
                        RPExt.tagStats = {};
                        // Update the extension
                        ruleCtx.rulePack.setExtension(RPExt);
                      },
            closedownRP : function(ruleCtx)
                      {
                        let ojTag, tagStats =
context.rulePack.getExtension().tagStats;

                        console.log("\n");
                        for (ojTag in tagStats)
                        {
                          console.log(`<${ojTag}> : ${tagStats[ojTag]}`);
                        }
                        console.log("\n");
                      }
        };
};
```

Here is typical sample output returned by these rules:

```
<oj-option> :     1824
<oj-label> :      1413
<oj-input-text> :  545
<oj-button>     :  447
 . . .
<oj-chart-item> :    1
<oj-chart-group> :   1
<oj-bind-dom> :      1
```

10

Access Oracle JET Metadata

Use the **metaLib** utility library when you need to access Oracle JET audit metadata.

The utility methods in **metaLib** (metadata access library) provided by Oracle JAF insulate the rule writer from changes to the format of the metadata.

This audit rule example from the built-in JAF rule set illustrates the usage of **metaLib** by checking for deprecated components.

```
function register()
{
  // 'ojtag' signifies that the element name starts with 'oj-'
  return { ojtag : function(ruleCtx, tagName)
                   {
                     let issue, suggestion;
                     const metaLib = context.utils.metaLib;

                     // true if the <oj-xx> name represents a JET built-in
component
                     if (! ruleCtx.ojTag) { return ; }
                     // method returns the suggested alternative if deprecated
                     suggestion = metaLib.isTagDeprecated(tagName);
                     if (suggestion !== null)
                     {
                       issue = new ruleCtx.Issue(`<${tagName}> is
DEPRECATED! : ${suggestion}`);
                       ruleCtx.reporter.addIssue(issue, ruleCtx);
                     }
                   }
         };
};
```

11

Create the Audit File Set at Runtime

Use the **JafLib** API when your rule pack needs to audit a dynamically derived file set, for example, by inspecting the current state of other data sets.

First, the configuration file should be setup to specify no files.

```
{
   "files"   : [],
   "exclude" : [],     // if needed


   "rulePacks" : {
                        // your rulepack reference
                        // this pack will contain the fileset generating rule
                 }
}
```

Next you need to create the rule that will generate the file lists. The rule needs to listen for the JAF lifecycle phase **startaudit**. This phase occurs immediately before the general auditing phase begins.

This rule uses `setFileset()` in `ruleCtx.utils.jafLib` to set the file set. Note that full file paths (not relative) must be used.

```
let  verboseMode;

function  register(regCtx)
{
   // optional feedback
   verboseMode = regCtx.sysOpts.verboseMode;

   return { "startaudit" : _fn };
};


function  _fn(ruleCtx)
{
   // create an array of full filepaths
   let fileset = _computeFileSet(ruleCtx);
   let exclude = _computeExcludeSet(ruleCtx);


  if (fileset.length)
   {
      if (verboseMode) { console.log(`Rule 'my-fileset-generator': injecting ${fileset.length} files into configuration 'files'`); }

      // param exclude may be omitted
      ruleCtx.utils.jafLib.setFileset(ruleCtx, fileset, exclude);
```

```
        }
    };
```

12

Reference: Custom Audit Rule Listener Types

Use this reference to learn about the listener types that you can register in custom audit rules. The available listener types are specific to the file types of the Oracle JET application.

Listener Types	Description
Listener Types for HTML and JSON Rules	The Oracle JAF audit engine supports a long list of listener types specific to HTML and JSON files.
Listener Types for CSS Rules	The Oracle JAF audit engine supports listener types specific to CSS source files.
Listener Types for JavaScript/ TypeScript Rules	The Oracle JAF audit engine supports listener types specific to JavaScript source files.
Listener Types for Markdown Rules	The Oracle JAF audit engine supports listener types specific to Markdown source files.
Listener Types for TSX Rules	The Oracle JAF audit engine supports listener types specific to renderable JSX content in TSX source files.

Listener Types for HTML and JSON Rules

The Oracle JAF audit engine supports a long list of listener types specific to HTML and JSON files.

The following table describes the listener event types for the various data nodes that JAF may encounter when parsing the contents of HTML and JSON files and generating an abstract syntax tree (AST) for the parsed files. You implement audit rules for the HTML and JSON files of your application by registering an event listener on the desired node. The table also list the expected arguments that you supply in the registered listener and the context properties that your listener can access on the event for a particular node.

Listener Type	Listener Arguments	Description, Including Available Context Properties
webcomp	**(***Object* **context,** *string* **elementName)**	Called for any DOM element recognized as a web component. Your **webcomp** event listener can access the following context properties. • **context.type**—a string `tag`. • **context.tag—**a string that is the element name • **context.ojTag**—a boolean: `true` if element is an Oracle JET Web Component. • **context.tagNode**—an object that is the current node in the Abstract Syntax Tree (AST). • **context.node**—an object that is the current node in the AST. • **context.position**—an object whose members **row** and **col** represent the position of the tag. Members **startIndex** and **endIndex** represent the displacements into the file. • **context.rawData**—a string that is the data from the resource (either a file or a URL). This is not used for zip file data. • **context.data**—a string that is the data being audited. This is typically the same as **rawData**, but can be different in the case of an HTML file in a zip. The HTML data can be found in **rawData**, but if the HTML contains embedded JavaScript or CSS, and that data is being audited, then **data** will contain the JavaScript or CSS data. • **context.suppData**—an object that is Null, except in the case of an audit involving an Oracle Component Exchange zip file. If the zip contains a `component.json` file, then **suppData** an object that contains supplementary information about the `component.json`. This property will be present for all other files processed in the zip, and allows the files to be evaluated in the context of the `component.json`. The object contains the following properties: – **suppData.obj**—an object that contains the parsed JSON, or Null if the parse failed. – **suppData.msg**—an error message if the parse failed, else Null. • **context.elemStack**—an array of parent element names to the current element. The final entry (the one with the highest index) is the current element name. For example, `["html", "body", "div", oj-button"]`.
jetcomp	**(***Object* **context,** *string* **elementName)**	Called for any JET web component. Your **jetcomp** event listener can access the following context properties. • **context.type**—a string `jetcomp`. • **context.ojTag**—a boolean: `true` if element is an Oracle JET Web Component. • **context.ojNs**—a boolean: `true`. For all other **context** contents, refer to the previous **webcomp** entry.

Listener Type	Listener Arguments	Description, Including Available Context Properties
extcomp	**(***Object* **context,** *string* **elementName)**	Called for any external (not-JET) web component. Your **extcomp** event listener can access the following context properties. • **context.type**—a string `extcomp`. • **context.ojTag**—a boolean: `false`. • **context.ojNs**—a boolean: `true`, if the element name is defined in the OJ namespace. For all other **context** contents, refer to the previous **webcomp** entry.
elem	**(***Object* **context,** *string* **elementName)**	Called for any DOM element. Your **elem** event listener can access the following context properties. • **context.type**—a string `elem`. • **context.ojTag**—a boolean: `true` if element is an Oracle JET Web Component. • **context.ojNS**—a boolean: `true` if element name is defined in the OJ namespace.
globtag	**(***Object* **context,** *string* **elementName)**	Called for any global/common DOM element (including standard HTML5 element names). Your **globtag** event listener can access the following context properties. • **context.type**—a string `tag`. • **context.ojTag**—a boolean: `false`. • **context.ojNS**—a boolean: `false`.
notglob	**(***Object* **context,** *string* **elementName)**	Called for any element that is not a common HTML name nor an SVG tag name. Your **notglob** event listener can access the following context properties. • **context.type**—a string `notglob`. • **context.ojTag**—a boolean: `true` if element is a JET component. • **context.ojNS**—a boolean: `true` if tag uses a registered namespace, else `false`.
tag	**(***Object* **context,** *string* **elementName)**	Called for any element name NOT starting with `oj-`. Your **tag** event listener can access the following properties. • **context.type**—a string `tag`. • **context.ojTag**—a boolean: `false`. • **context.ojNS**—a boolean: `false`.
bindingtag	**(***Object* **context,** *string* **elementName)**	Called for a JET binding tag such as `oj-bind-if` or `oj-bind-for-each`. Your **bindingtag** event listener can access the following properties. • **context.type**—a string `bindingtag`. • **context.ojTag**—a boolean: `false`. • **context.ojNS**—a boolean: `true` if element name is defined in the OJ namespace.

Listener Type	Listener Arguments	Description, Including Available Context Properties
ojtag	**(***Object* **context,** *string* **elementName)**	Called for any element name starting with `oj-`. Your **ojtag** event listener can access the following properties. • **context.type**—a string `ojtag`. • **context.ojTag**—a boolean: `false`. • **context.ojNS**—a boolean: `true` if element name is defined in the OJ namespace.
<component>	**(***Object* **context,** *string* **elementName)**	Called for any non-global HTML element with the name declared between angle brackets (e.g., `<oj-messages>`). A regular expression may be used (e.g., `<^oj-combobox>` to match all elements starting with "oj-combobox"), and whitespace is permitted on either side of the component name. Your **<component>** event listener can access the following properties. • **context.type**—a string `<component>`. • **context.ojTag**—a boolean: `false`. • **context.ojNS**—a boolean: `true` if element name is defined in the OJ namespace.
<component attrname=>	**(***Object* **context,** *string* **attribName,** *string* **attribValue,** *string* **rawAttribValue)**	Called for the specified attribute if found on the component element (e.g., `<oj-messages display-options.category=>`). Whitespace is permitted on either side of the component name and before the ending chevron. The component declaration may not be a regular expression. Your **<component attrname=>** event listener can access the following properties. • **context.type**—a string `<component|attrname>`. **Note:** See the note in the `<attrname=>` entry for additional information regarding the `class` and `style` attributes.

Listener Type	Listener Arguments	Description, Including Available Context Properties
<attrname=>	**(***Object* **context,** *string* **attribName,** *string* **attribValue,** *string* **rawAttribValue)**	Called for the specified attribute found on any component element (e.g., `<display-options.category=>`). This is a special case of the component/attribute combination listed above, with the component omitted. Whitespace is permitted on either side of the attribute declaration. Your **<attrname=>** event listener can access the following properties. • **context.type**—a string `<attrname=>`. **Note:** This syntax should not be used for `style` or `class` attributes. These special cases are handled by their own registered types of style and class respectively. See the subsequent entries in this table.
script	**(***Object* **context,** *string* **elementName)**	Called for `<script>` elements. Note that rules with registered type **tag** are also called.
link	**(***Object* **context,** *string* **elementName)**	Called for `<link>` elements. Note that rules with registered type **tag** are also called.
attr	**(***Object* **context,** *string* **attribName,** *string* **attribValue,** *string* **rawAttribValue)**	Called for each attribute of an element (excluding Oracle JET `event`, `type` and `class` attributes). The **rawAttrValue** attribute value contains the complete text including the string delimiting quotes. Your **attr** event listener can access the following property. • **context.type**—a string `attr`.
attrexpr	**(***Object* **context,** *string* **attribName,** *string* **attribValue,** *string* **rawAttribValue)**	Called for each attribute of an element if the attribute value represents a `[[. . .]]` or `{{. . .}}` expression.
attrexpr-$props	**(***Object* **context,** *string* **attribName,** *string* **attribValue,** *string* **rawAttribValue)**	Called for each attribute whose value represents a `[[. . .]]` or `{{. . .}}` expression and contains at least one reference to **$properties**.
class	**(***Object* **context,** *string* **elemName,** *string* **classValue)**	Called for each `class` attribute of an element. Your **class** event listener can access the following property. • **context.type**—a string `class`.
style	**(***Object* **context,** *string* **attribName,** *string[]* **attribValue,** *string* **rawAttribValue)**	Called for each `style` attribute of an element. The **attrValue** is an array of styles extracted from the **style** attribute. **rawAttrValue** is the attribute string. Your **style** event listener can access the following property. • **context.type**—a string `style`.
type	**(***Object* **context,** *string* **elemName,** *string* **typeValue)**	Called for each `type` attribute of an element. Your **type** event listener can access the following property. • **context.type**—a string `type`.

Listener Type	Listener Arguments	Description, Including Available Context Properties
event	**(***Object* **context,** *string* **eventName,** *Object* **eventValue)**	Called for each Oracle JET event attribute (starting with `on-oj-`). The **eventValue** attribute is an object with the following properties: • **val**—a string that is the event attribute value. • **rawval**—a string that is the full event attribute value, including containing quotes (as specified). Your **event** event listener can access the following property. • **context.type**—a string `event`.
comment	**(***Object* **context,** *string* **comment)**	Called for HTML comments of the form `<!-- -->`. For example, for the comment **<!-- This is a comment -->**, the **comment** argument will contain the string `This is a comment`.
doctype	**(***Object* **context,** *string* **comment,** *string* **value)**	Called for directives of the form `<!DOCTYPE ...>`. For example, for the comment **<!DOCTYPE html>**, the **doctype** argument will contain the full data, including the string `!DOCTYPE html` and **value** argument will contain the rest of the statement, the string `html`.
procstmt	**(***Object* **context,** *string* **procinstr,** *string* **value)**	Called for processing instructions of type `<?x.........x?>`. For example, the **procinstr** argument might be the string `?robots index="yes" follow="no"?` and the **value**argument will contain the rest of the statement, in this case, the string `index="yes" follow="no"`.
directive	**(***Object* **context,** *string* **directive,** *string* **value)**	Called for processing instructions of the form `<!xxxx>` that are not DOCTYPE instructions. For example, for the directive **<!ABCD xxxx>**, the **directive** argument will contain the full data, including the string `!ABCD xxxx` and the **value** argument will contain the rest of the statement, the string `xxxx`.
json	**(***Object* **context)**	Called for a file of type `json`. Your **json** event listener can access the following properties. • **context.type**—a string `json`. • **context.suppData**—an object with the following members. – **suppData.obj**—the parsed JSON object, or Null if the parse failed. – **suppData.ast**—the Abstract Syntax Tree (AST) for the JSON file, or Null if the parse failed. – **suppData.msg**—an error message if the parse failed, else Null.

Listener Type	Listener Arguments	Description, Including Available Context Properties
compjson	**(***Object* **context)**	Called for a file of type `json`. Your **compjson** event listener can access the following properties. • **context.type**—a string `json`. • **context.suppData**—an object with the following members. – **suppData.obj**—the parsed JSON object, or Null if the parse failed. – **suppData.ast**—the Abstract Syntax Tree (AST) for the JSON file, or Null if the parse failed. – **suppData.msg**—an error message if the parse failed, else Null.
file	**(***Object* **context)**	Called after reading a file, and prior to any standard rules (node rules, not hook rules) being fired on the file. Your **file** event listener can access the following properties. • **context.phase**—a string `file`. • **context.rawData**—a string that is the file contents. • **context.data**—a string that is the data being audited. This is typically the same as **rawData**, but can be different in the case of an HTML file in a zip. The HTML data can be found in **rawData**, but if the HTML contains embedded JavaScript or CSS, and that data is being audited, then **data** will contain the JavaScript or CSS data. • **context.suppData**—a string that with supplementary file data. • **context.zipContent**—an array of file names in the component zip, or an error string if there was an error during expansion of the zip. For file type `.zip` only.
endfile	**(***Object* **context)**	Called after all standard rules (node rules, not hook rules) have been fired on the file. Your **endfile** event listener can access the following properties. • **context.phase**—a string `file`. • **context.rawData**—a string that is the file contents. • **context.data**—a string that is the data being audited. This is typically the same as **rawData**, but can be different in the case of an HTML file in a zip. The HTML data can be found in **rawData**, but if the HTML contains embedded JavaScript or CSS, and that data is being audited, then **data** will contain the JavaScript or CSS data. • **context.suppData**—a string that with supplementary file data. • **context.zipContent**—an array of file names in the component zip, or an error string if there was an error during expansion of the zip. For file type `.zip` only.

Listener Types for CSS Rules

The Oracle JAF audit engine supports listener types specific to CSS source files.

The following table describes the listener event types for the various data nodes that JAF may encounter when parsing the contents of CSS files and generating an abstract syntax tree (AST) for the parsed files. You implement audit rules for the CSS files of your application by registering an event listener on the desired CSS node. The table also list the expected arguments that you supply in the registered listener.

Listener Type	Listener Arguments	Description
css-sheet	**(***Object* **context,** *node* **sheet)**	Called for a stylesheet. **sheet** is a node from the AST representing all the **Rule** nodes in the sheet. The complete AST can be found in **context.ast**.
css-rule	**(***Object* **context,** *Object* **rule)**	Called for a CSS rule declaration. **rule** is a node from the AST. A rule node contains the following properties and sub-properties: **SelectorList**, **Selector**, **Block**. The complete AST can be found in **context.ast**.
css-atrule	**(***Object* **context,** *Object* **rule,** *string* **identifier)**	Called for any CSS `@rule` statement. For example, `@keyframes`. **rule** is a node from the AST. **identifier** is the rule identifier. For example, for `@keyframes`, the identifier will be **keyframes**. A rule node contains the following properties and sub-properties: **SelectorList**, **Selector**, **Block**. The complete AST can be found in **context.ast**.
css-@xxxx	**(***Object* **context,** *Object* **rule,** *string* **identifier)**	Called for a CSS statement of the name ***xxxx***. For example, for `@media`, the type is **css-@media**. **rule** is a node from the AST. **identifier** is the rule identifier. For example, for `@media`, the identifier will be **media**. A rule node contains the following properties and sub-properties: **SelectorList**, **Selector**, **Block**. The complete AST can be found in **context.ast**.
css-selector	**(***Object* **context,** *node* **selector)**	Called for a selector of some type. Refer to **node.type** for the actual selector type.
css-sel-type	**(***Object* **context,** *node* **selector)**	Called for a ***type*** selector.
css-sel-id	**(***Object* **context,** *node* **selector)**	Called for an ***ID*** selector.
css-sel-class	**(***Object* **context,** *node* **selector)**	Called for a ***class*** selector.
css-sel-pseudo	**(***Object* **context,** *node* **selector)**	Called for a ***pseudo*** selector.
css-var	**(***Object* **context,** *node* **var**, *Object* **rule**)	Called for a CSS variable declaration. ***var*** is the variable declaration node in the AST and contains all information about the variable. Two of the prominent members of var are: • **var.property** is of type `String` and references the CSS variable's name. • **var.value** is of type `Object` and references the CSS variable's value. The parsed value details can be found in the elements of *array* ***var.value.children*** Argument ***rule*** is the Rule object that contains the variable declaration.

Listener Types for Markdown Rules

The Oracle JAF audit engine supports listener types specific to Markdown source files.

The following table describes the listener event types for the data nodes that JAF may encounter when parsing the contents of Markdown files and generating an abstract syntax tree (AST) for the parsed files. You implement audit rules for the Markdown files of your application by registering an event listener on the desired Markdown node. The table also lists the expected arguments that you supply in the registered listener.

For descriptions of the format of returned objects and their properties, see Context Object Properties Available to Markdown Rule Listeners.

Listener Type	Listener Arguments	Description
md-link	**(***Object* **context,** *Object* **link)**	Invoked for direct or indirect link references. For example, `[Click Here](https://some/link/foo.html)` It is also invoked for URLs found in paragraph text without markup. The `link` object has the following properties: • **inline**: a boolean, default true, which indicates a complete link including a URL • **link**: the URL string • **text**: a string containing the link text: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key If the link declaration uses a reference link, the link object will have the property **inline** declared to be `false`, and the **refKey** property will replace the **link** property.
md-ref	**(***Object* **context,** *Object* **ref)**	Invoked for indirect link references. For example, `[some ref]: <the url>` The `ref` object has the following properties: • **refKey**: a string with the reference key (e.g., `some ref`) • **link**: a string containing the link with URL (e.g., `www.github.com`) • **pos**: an object containing location information, with the following properties: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key

Listener Type	Listener Arguments	Description
md-para	**(***Object* **context,** *Object* **para)**	Invoked for paragraph text, and headings using `#`. The `para` object has the following properties: • **text**: a string with paragraph or heading text • **level**: the number of the heading level • **pos**: an object containing two objects that hold the paragraph/heading start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 Note: For standard paragraph text, **level** is 0. A value greater than zero represents the number of markup heading `#` characters (e.g., `##` represents a heading level h2, where **level** will be 2).
md-image	**(***Object* **context,** *Object* **image)**	Invoked for all image references. For example, `![Click Here](https://somelink/foo.png)` The `image` object has the following properties: • **inline**: a boolean, default `true`, which indicates a complete link including a URL • **link**: the image URL string • **alt**: a string containing the alt text • **pos**: an object containing location information, with the following properties: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key If the image declaration uses a reference link, the image object will have the property **inline** declared to be `false`, and the **refKey** property will replace the **link** property.

Listener Type	Listener Arguments	Description
md-code	**(***Object* **context,** *Object* **code)**	Invoked for all fenced code blocks (using backticks or tildes). For example, ``` // Installation instructions npm install -g @oracle/oraclejet-audit ``` The `code` object has the following properties: • **code**: a string containing the code block, with `/n` newline characters used as separators • **pos**: an object containing two objects that hold the start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of code block, relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of code block, relative to 0
md-list	**(***Object* **context,** *Object* **list)**	Invoked for ordered or unordered lists using the markup `*` or `n.` notation. The `list` object has the following properties: • **ordered**: a boolean, with `false` used for unordered lists and `true` for ordered lists • **items**: an array of list item objects, each with an **item** property referring to a string value • **pos**: an array containing two objects that hold the list start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of line relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 If a list item represents a sub-list, the **children** property is used within the list item. The **children** property is an array containing sub-list objects; each object is formatted the same as the `list` object, with **ordered**, **items**, and **pos** properties.

Listener Types for JavaScript/TypeScript Rules

The Oracle JAF audit engine supports listener types specific to JavaScript/TypeScript source files.

The following table describes the listener event types for the various data nodes that JAF may encounter when parsing the contents of JavaScript/TypeScript files and generating an abstract syntax tree (AST) for the parsed files. You implement audit rules for the JavaScript/TypeScript

files of your application by registering an event listener on the desired JavaScript/TypeScript node.

Listener Type	Listener Type	Listener Type
AssignmentExpression	Identifier	UpdateExpression
AssignmentPattern	IfStatement	VariableDeclaration
ArrayExpression	Literal	VariableDeclarator
ArrayPattern	LabeledStatement	WhileStatement
ArrowFunctionExpression	LogicalExpression	WithStatement
AwaitExpression	MemberExpression	YieldExpression
BlockStatement	MetaProperty	JSXIdentifier
BinaryExpression	MethodDefinition	JSXNamespacedName
BreakStatement	NewExpression	JSXMemberExpression
CallExpression	ObjectExpression	JSXEmptyExpression
CatchClause	ObjectPattern	JSXExpressionContainer
ClassBody	Program	JSXElement
ClassDeclaration	Property	JSXClosingElement
ClassExpression	RestElement	JSXOpeningElement
ConditionalExpression	ReturnStatement	JSXAttribute
ContinueStatement	SequenceExpression	JSXSpreadAttribute
DoWhileStatement	SpreadElement	JSXText
DebuggerStatement	Super	ExportDefaultDeclaration
EmptyStatement	SwitchCase	ExportNamedDeclaration
ExperimentalRestProperty	SwitchStatement	ExportAllDeclaration
ExperimentalSpreadProperty	TaggedTemplateExpression	ExportSpecifier
ExpressionStatement	TemplateElement	ImportDeclaration
ForStatement	TemplateLiteral	ImportSpecifier
ForInStatement	ThisExpression	ImportDefaultSpecifier
ForOfStatement	ThrowStatement	ImportNamespaceSpecifier
FunctionDeclaration	TryStatement	
FunctionExpression	UnaryExpression	

As an alternative to the AST node types defined above, an object of abbreviated string constants can be found in the rule context. For example:

```
var NT = ruleCtx.utils.AstUtils.getNodeTypes() ;

if (node.type === NT.CALL_EXPR || node.type === NT.MEMBER_EXPR)  {
   . . .
}
```

In the register context, the node types object can be found in **regCtx.jsNodeTypes()** or **regCtx.tsNodeTypes()**, depending on whether the rule handles JavaScript or TypeScript. The following example caches the node types for JavaScript.

```
var NT ;

register(regXtx)  {
   NT = regCtx.jsNodeTypes ;     // cache the node types
```

```
    . . .
}
```

Here is a list of available constants:

Constant	Node Type
ARRAY_EXPR	"ArrayExpression"
ARROW_FUNC_EXPR	"ArrowFunctionExpression"
ASSIGN_PATTERN	"AssignmentPattern"
ASSIGN_EXPR	"AssignmentExpression"
ASSIGNMENT_EXPR	"AssignmentExpression"
AWAIT_EXPR	"AwaitExpression"
BINARY_EXPR	"BinaryExpression"
BLOCK_STMT	"BlockStatement"
BREAK_STMT	"BreakStatement"
CALL_EXPR	"CallExpression"
CATCH_CLAUSE	"CatchClause"
CLASS_BODY	"ClassBody"
CLASS_DECLARATION	"ClassDeclaration"
CONDITIONAL_EXPR	"ConditionalExpression"
CONTINUE_STMT	"ContinueStatement
CLASS_EXPR	"ClassExpression"
DO_WHILE_STMT	"DoWhileStatement"
DEBUG_STMT	"DebuggerStatement"
EMPTY_STMT	"EmptyStatement"
EX_REST_PROP	"ExperimentalRestProperty"
EXPR_STMT	"ExpressionStatement"
EX_SPREAD_STMT	"ExperimentalSpreadProperty"
FOR_STMT	"ForStatement"
FOR_IN_STMT	"ForInStatement"
FOR_OF_STMT	"ForOfStatement"
FUNC_DECLARATION	"FunctionDeclaration"
FUNC_EXPR	"FunctionExpression"
IDENTIFIER	"Identifier"
IF_STMT	"IfStatement"
LOGIC_EXPR	"LogicalExpression"
LABELED_STMT	"LabeledStatement"
LITERAL	"Literal"
MEMBER_EXPR	"MemberExpression"
META_PROP	"MetaProperty"
METH_DEF	"MethodDefinition"
NEW_EXPR	"NewExpression"
OBJ_EXPR	"ObjectExpression"
OBJ_PATTERN	"ObjectPattern"
PROGRAM	"Program"
PROPERTY	"Property"

Constant	Node Type
PROPERTY_DEF	"PropertyDefintion"

> **Note:**
>
> The parent node of an **ArrowFunctionExpression** used to define a class method may be changed when using the newer AST parser library from **ClassProperty** to this **PropertyDefintion** node type

REST_ELEM	"RestElement"
RETURN_STMT	"ReturnStatement"
SEQUENCE_EXPR	"SequenceExpression"
SPREAD_ELEM	"SpreadElement"
SUPER	"Super"
SWITCH_CASE	"SwitchCase"
SWITCH_STMT	"SwitchStatement"
TAGGED_TEMPLATE_EXPR	"TaggedTemplateExpression"
TEMPLATE_ELEM	"TemplateElement"
TEMPLATE_LIT	"TemplateLiteral"
THIS_EXPR	"ThisExpression"
THROW_EXPR	"ThrowExpression"
TRY_STMT	"TryStatement"
UNARY_EXPR	"UnaryExpression"
UPDATE_EXPR	"UpdateExpression"
VAR_DECLARATION	"VariableDeclaration"
VAR_DECLARATOR	"VariableDeclarator"
WHILE_STMT	"WhileStatement"
WITH_STMT	"WithStatementt"
YIELD_EXPR	"YieldExpression"
JSX_ID	"JSXIdentifier"
JSX_NS_NAME	"JSXNamespacedName"
JSX_EMPTY_EXPR	"JSXEmptyExpression"
JSX_EXPR_CONTAINER	"JSXExpressionContainer"
JSX_ELEM	"JSXElement"
JSX_CLOSING_ELEM	"JSXClosingElement"
JSX_OPENING_ELEM	"JSXOpeningElement"
JSX_ATTRIB	"JSXAttribute"
SX_SPREAD_ATTRIB	"JSXSpreadAttribute"
JSX_TEXT	"JSXText"

Constant	Node Type
EXPORT_DEFAULT_DECL	"ExportDefaultDeclaration"
EXPORT_NAMED_DECL	"ExportNamedDeclaration"
EXPORT_ALL_DECL	"ExportAllDeclaration"
EXPORT_SPECIFIER	"ExportSpecifier"
IMPORT_DECL	"ImportDeclaration"
IMPORT_EQUALS_DECL	"TSImportEqualsDeclaration"
IMPORT_SPECIFIER	"ImportSpecifier"
IMPORT_DEFAULT_SPECIFIER	"ImportDefaultSpecifier"
IMPORT_NS_SPECIFIER	"ImportNamespaceSpecifier"
EXTERN_MOD_REF	"TSExternalModuleReference"

Listener Types for TSX Rules

The Oracle JAF audit engine supports listener types specific to TSX files.

The following table describes the listener event types for the various data nodes that JAF may encounter when parsing the contents of TSX files and generating an abstract syntax tree (AST) for the parsed files. You implement audit rules for the TSX files of your application by registering an event listener on the desired node. The table also list the expected arguments that you supply in the registered listener and the context properties that your listener can access on the event for a particular node.

Listener Type	Listener Signature	Description, Including Available Context Properties
TsxRenderComponent	(*Object* **ruleContext,** *Object* **tsxRenderComponent**)	Called for an associated group of TSX renderable content (that is, multiple component/HTML markup elements in a single declaration, such as a *render()* call, or a variable declaration, and so on.)
TsxComponent	(*Object* **ruleContext,** *Object* **tsxComponent**)	Called for any component/HTML element/Preact function found in the TSX renderable content.
TsxWebComponent	(*Object* **ruleContext,** *Object* **tsxComponent**)	Called for any web component known to JAF found in the TSX renderable content.
TsxJetComponent	(*Object* **ruleContext,** *Object* **tsxComponent**)	Called for any JET web component (legacy or Jet Core pack)
TsxElem	(*Object* **ruleContext,** *Object* **tsxComponent**)	Called for any standard HTML element found in the TSX renderable content.
TsxEvent	(*Object* **ruleContext,** *Object* **tsxProperty**)	Called for any event property found in TSX renderable content.
TsxProperty	(*Object* **ruleContext,** *Object* **tsxProperty**)	Called for any property found in the TSX renderable content.
Tsx<component_name>	(*Object* **ruleContext,** *Object* **tsxComponent**)	Called for the named component/HTML element element found in the TSX renderable content.
Tsx<component_name propName=>	(*Object* **ruleContext,** *Object* **tsxProperty**)	Called for the named component/HTML element and named property found in the TSX renderable content.
Tsx<propName=>	(*Object* **ruleContext,** *Object* **tsxProperty**)	Called for the named property found in the TSX renderable markup.

Listener Type	Listener Signature	Description, Including Available Context Properties
TsxFunction	**(Object ruleContext, Object tsxFunction)**	Called for each function declaration in the .TSX file before any of the other TsxXxx listeners are called. After parsing the file into an AST, JAF makes a pass through the tree accumulating the general details of the function(s) and particularly the `return` statement(s). `TsxFunction` listeners are called first before JAF walks the tree and any subsequent invocation of the other TsxXxx listeners. This permits these other listeners to easily peek at the functions referenced in the JSX (for example, `foo={MyFunc}`.

Additionally, all file hook listener types (such as `file` and `endfile`) and any AST node types are permitted.

> **Note:**
>
> For named properties and events, the TSX form shown above must be used. That is, an event type of the form `<oj-foo>` cannot be used. It must be declared as `Tsx<oj-foo>`.

13

Reference: Custom Audit Rule Context Object Properties

Use this reference to learn about the properties and functionality available to custom audit rules on passed in context objects. The context objects contain AST node data generated by the Oracle JAF audit engine in response to the audited file set.

Context Objects	Description
Context Object Members Passed to the Register Function	A rule's `register()` function receives a `Register` context object when it is invoked by the JAF audit engine during the audit startup.
Context Object Properties Available to Registered Listeners	A rule's registered listeners receive a `Rule` context object when the listener is triggered by the JAF audit engine for specific data in the target files of the audit.
Context Object Properties Available to CSS Rule Listeners	A CSS audit rule's registered listeners receive a `Rule` context object with information that is specific to CSS processing.
Context Object Properties Available to Markdown Rule Listeners	A Markdown audit rule's registered listeners receive a `Rule` context object with information that is specific to MArkdown processing.

Context Object Members Passed to the Register Function

A rule's `register()` function receives a `Register` context object when it is invoked by the JAF audit engine during the audit startup.

A `Register` context object is passed during JAF audit startup to the audit rules specified by your JAF configuration. The context object generated at JAF startup contains miscellaneous supporting data and functionality that is available to the `register()` function of all rule types.

> **Note:**
>
> You must not cache the `Register` context object. You may cache contained values, for example, like `context.utils.msgLib`.

Members	Description
rulePack	The rule pack manager instance. This provides methods to access rule pack data. For details, see RulePack Class Methods.
ruleOpts	A read-only copy of the rule options object from the rule pack's `rules.json` file, and any override in the configuration property **ruleMods**.
runMode	The run mode value will be **cli** to reflect the audit invocation from the command line. Other options may be supported in a future release.
config	The configuration object which is a read-only copy of the effective runtime configuration properties.
ojetConfig	The **oraclejetconfig** configuration object (if the Oracle JAF audit is run in the root of a project maintained by the Oracle JET Tooling). This property is only available in the **startupRP** and **closedownRP** phase of hook rules.

Members	Description
utils	Miscellaneous utility libraries. Refer to **utils** in Context Object Properties Available to Registered Listeners.
sysOpts	An object containing boolean properties **verboseMode** and **debugMode** reflecting the runtime options.
jsNodeTypes	An object containing JavaScript Abstract Syntax Tree (AST) enumerated node listener type strings (if the rule handles JavaScript). See also, Listener Types for JavaScript/ TypeScript Rules.

Context Object Properties Available to Registered Listeners

A rule's registered listeners receive a `Rule` context object when the listener is triggered by the JAF audit engine for specific data in the target files of the audit.

The JAF audit engine passes a `Rule` context object to all rules with a listener that your rule registers to handle specific node types in the target file set. The generated context object contains the following miscellaneous supporting data and functionality available to the registered listener of any rule type.

> **Note:**
>
> You must not cache the `Rule` context object. You may cache contained values, for example, like `context.utils.msgLib`.

Property	Description
type	Designates the type of data passed in the arguments to a rule, or signals an event. For HTML, JSON, and CSS: refer to Listener Types for HTML and JSON Rules. For JavaScript/TypeScript: refer to Listener Types for JavaScript/TypeScript Rules. The registered type can also specify that the rule is a hook rule. Hook rules are effectively hooks into the audit engine, and are called at specific phases of an audit, as opposed to a response to parsed file data for a general node rule. For details, see About Hook Rule Invocation. The hook types are **startup**, **closedown**, **startupRP**, **startaudit**, **closedownRP** and **file**.
node	A node object containing details from the current node in the DOM, JavaScript/TypeScript (Abtract Syntax Tree - AST), JSON (AST), or selector trees.
tagNode	The current tag element node object containing details about the containing tag element. In the case of type `tag`, the **node** and **tagNode** properties are the same.
tag	The HTML element name, for example `div` or `oj-avatar`.
ojTag	`true` if the HTML element starts with `oj-` and is a defined Oracle JET element tag. Note that a `context.type` of `ojtag` reflects only that the element name starts with `oj-`.
ojNS	`true` if the element name is defined in the `OJ` namespace.
elemStack	For HTML pages, an array of node objects providing the positional context of the current element. The last entry (with the highest index) in the array is the current element.
sysOpts	Miscellaneous system options including the boolean options **verboseMode** and **debugMode**. Refer also to context property **msgLib**.
userDefs	The configuration file **userDefs** property (if defined). This property is not examined by Oracle JAF.

Property	Description
rulePack	The rule pack manager instance. This provides methods to access rule pack data. For a description of the available methods, see RulePack Class Methods.
Issue	The `Issue` class. Use this to create a new `Issue` object for each issue that is to be reported and query the Issue details. For a description of the available methods, see Rule Issue Class Methods.
reporter	The `Reporter` instance. Constructed `Issue` objects are passed to this instance by an audit rule for inclusion in the audit output. Use this to add the Issue object to a Reporter instance and query Reporter details. For a description of the available methods, see Rule Reporter Class Methods.
ruleName	The rule name.
filepath	The full file path to the file currently being processed. Note that for consistency, **filepath** always uses forward slashes, regardless of the platform. Audit rules are platform independent in this respect.
filetype	The file type (in lower case)
phase	If the rule is a hook rule, this string represents the phase in which it was invoked, including **startup**, **closedown**, **startupRP**, **startaudit**, **closedownRP**, or **file**.
utils	Contains Oracle JAF utility libraries: • **utils** - miscellaneous non-file related utilities. Refer to Utils: General Non-File System Functions. • **fsUtils** - miscellaneous file related utilities. Refer to FsUtils: File System Functions. • **DomUtils**—a parsed DOM tree utility library. Provides methods to traverse and inspect the parsed tree of nodes. Available only when `context.filetype` is "`html`". Refer to DomUtils: Node Object Functions. • **semVerUtils**—a utility library providing semantic versioning support. This is available when the `context.filetype` is `json`. Refer to SemVerUtils: Semantic Version Functions. • **AstUtils**—miscellaneous JavaScript/TypeScript Abstract Syntax Tree (AST) methods. Available only when `context.filetype` is `js` or `ts/tsx`. Refer to AstUtils: JavaScript File Helper Functions. • **metaLib** —an Oracle JAF audit metadata access library. Refer to MetaLib: JET Metadata Access Functions. • **jafLib** —available only in phase **auditstart**. Provides the ability to dynamically define the file set to be audited. Refer to JafLib: JAF Core Access Methods. • **msgLib** —miscellaneous message display routines. Refer to MsgLib: Message Display Functions. • **CssUtils**—CSS rule processing functions. Refer to CssUtils: CSS Utility Functions. • **sevLib**—rule severity level utility methods. Refer to SevLib: Severity Support Helper Functions. A library may be omitted depending on the registration type. For example, for type `html`, the **AstUtils** library (for JavaScript/TypeScript) will not be present. (See Utils: General Non-File System Functions.)
userDefs	The value defined (optionally) by the configuration **userDefs** property.
NodeTypes	A set of enumerated type definitions. For example, **NodeTypes.TAG** or **NodeTypes.SCRIPT**, and so on.

Context Object Properties Available to Markdown Rule Listeners

A Markdown rule's registered listeners receive a `Rule` context object with information that is specific to Markdown processing.

For Markdown processing, a rule can listen for `file` events (when a .md file is first read) or for specific Markdown events (when a particular type of markup is found). In either case, the JAF

audit engine passes the `Rule` context object to rule listeners that your rule registers to handle specific node types of the Markdown files. The `context` object generated for Markdown files contains supporting data and functionality that are specific to Markdown processing.

To access summarized data, the `context` object's supplementary data property **suppData** provides the following methods available on its `utils` object to obtain images, paragraphs, headings, code blocks, and so on. These are in addition to the base properties available on the `context` object for all rule types, as described in Context Object Properties Available to Registered Listeners.

> **Note:**
>
> You must not cache the `Rule` context object. You may cache contained values such as, for example, `context.utils.msgLib`.

The supplementary data property **suppdata** contains the following subproperties:

- **ast**: the abstract syntax tree (AST)
- **utils**: a utility object (methods in the table below)

Method	Arguments	Description
getLinks()	None	Scans the markup for URL references, and returns an array of `link` objects. The `link` object has the following properties: • **inline**: a boolean, default true, which indicates a complete link including a URL • **link**: the URL string • **text**: a string containing the link text • **pos**: an object containing location information, with the following properties: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key If the link declaration uses a reference link, the link object will have the property **inline** declared to be `false`, and the **refKey** property will replace the **link** property. The **refKey** property can be used to look up the associated URL via the reference links object. See **getRefLinks()** below.

Method	Arguments	Description
getRefLinks()	None	Scans the markup for reference links. A non-inline link provides a reference to a link defined elsewhere in the markup. This method provides an object containing the referenced link objects by reference key. Consider the following markup text with an indirect reference to a URL: `Link to the [resource file](some ref key)` This resolves to the following reference link: `[some ref key]: www./some/url` This returns an object containing the referenced link objects, each named by their reference key (e.g., `some ref key`). Each referenced link object has the following properties: • **link**: the URL string • **title**: the title string, if specified in the MD construct • **pos**: an object containing location information, with the following properties: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key
getImages()	None	Scans the markup for image declarations and returns an array of `image` objects. The `image` object has the following properties: • **inline**: a boolean, default `true`, which indicates a complete link including a URL • **link**: the image URL string • **text**: a string containing the alt text • **pos**: an object containing location information, with the following properties: – **line**: line number to reference key, relative to 1 – **col**: column number to reference key, relative to 1 – **start**: index number to start of reference key – **end**: index number to end of reference key If the image declaration uses a reference link, the image object will have the property **inline** declared to be `false`, and the **refKey** property will replace the **link** property.
getCode()	None	Scans the markup for fenced code blocks (using backticks or tildes) and returns an array of `code` objects. The `code` object has the following properties: • **code**: a string containing the code block, with `/n` newline characters used as separators • **pos**: an object containing two objects that hold the start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of code block, relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of code block, relative to 0

Method	Arguments	Description
getParas()	None	Scans the markup for paragraphs and headings. Returns an array of `para` objects. The `para` object has the following properties: • **text**: a string with paragraph or heading text • **level**: the number of the heading level • **pos**: an object containing two objects that hold the paragraph/ heading start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 Note: For standard paragraph text, **level** is 0. A value greater than zero represents the number of markup heading `#` characters (e.g., `##` represents a heading level h2, where **level** will be 2).
getLists()		Scans the markup for ordered and unordered lists and returns an array of objects. Each object has the following properties. The `list` object has the following properties: • **ordered**: a boolean, with `false` used for unordered lists and `true` for ordered lists • **items**: an array of list item objects, each with an **item** property referring to a string value • **pos**: an array containing two objects that hold the list start and end location info, respectively, in the following format: – **line**: starting line number, relative to 1 – **start**: index number to start of line relative to 0 – **end**: index number to end of line, relative to 0 – **line**: ending line number, relative to 1 – **start**: index number to start of line, relative to 0 – **end**: index number to end of line, relative to 0 If a list entry represents a sub-list, the **children** property is used within the list item. It contains an array containing each object sub-list, each using the same formatting as the `list` object, with **ordered**, **items**, and **pos** properties.
testParas()	**(***regexp* **RegExp**, [*boolean* **firstmatch**])	Takes a regular expression as an argument and applies it to each paragraph found in the markup. An optional second boolean argument specifies whether all matching paragraphs should be returned, or just the first matching paragraph (the default, if omitted. Returns an array of **para** objects. See **getParas()** above for **para** object properties. In the following example, **paras** will contain the matching paragraph objects: `var paras = utils.testParas(/Oracle/);`

Method	Arguments	Description
getLineMap()	None	Returns a `Map` object with line numbers (relative to 1) as the map keys. The return value from `get()` on the map is an object with properties **start** and **end**, which represent the start and end indices (relative to 0) into the file for the line specified. In this example, the **map** variable stores the `Map` object, and **pos** stores the position info for line 3 (as an object with properties **start** and **end**) `var map  = getLineMap();` `var pos  = map.get(3);`
getLine()	(*number* **line**)	Returns a line from the .md file, as a string. **line** is specified relative to 1. In this example, the **map** variable stores the `Map` object, and **line** stores line 3 as a string. `var map  = getLineMap();` `var line  = getLine(3);`
getLineDisp()	**(***number* **line**, [*number* **col**])	Returns the index number (relative to zero) to the start of the line specified. If optional **col** column number (relative to 1) is specified, the returned index will point to that column.

Context Object Properties Available to CSS Rule Listeners

A CSS rule's registered listeners receive a `Rule` context object with information that is specific to CSS processing.

The JAF audit engine passes a `Rule` context object to CSS rule listeners that your rule registers to handle specific node types of the target CSS files. The context object generated for CSS files contains the following supporting data and functionality, specific to CSS processing. This is in addition to the base properties available on the context object for all rule types, as describe in Context Object Properties Available to Registered Listeners.

> **Note:**
>
> You must not cache the `Rule` context object. You may cache contained values, for example, like `context.utils.msgLib`.

Property	Type	Description
context.ast	node	The full AST for the stylesheet.

Property	Type	Description
context.offset	Object	Contains the origin of the stylesheet relative to the containing file. If the stylesheet text is a standalone `.css` file, the values will be {**row** : 1, **col**: 1, **index**: 0}. However, if the stylesheet is an embedded HTML **<style>**, the **row**, **col**, and **index** values will reflect the position of the first character of the stylesheet in the containing file. This property can be used in conjunction with the **loc** property in a CSS node to reflect the true position of a particular property or value. (The **loc** position information is always relative to the start of the stylesheet.)
context.utils.CssUtils	Object	A utility library with miscellaneous functions available on the register context when the rule is invoked, as well as the rule context. Refer to CssUtils: CSS Utility Functions.

14

Reference: Custom Audit Rule Context Object Methods

Use this reference to obtain details about functionality that you can access through an instance of a rule pack manager, rule issue, and rule reporter that you create in custom audit rules.

Audit Rule Classes	Description
RulePack Class Methods	The rule pack manager instance you create provides methods to access rule pack data.
Rule Issue Class Methods	The rule issue instance you create provides methods to set properties of the reported issue.
Rule Reporter Class Methods	The rule reporter instance you create provides methods to handle reported issues.

RulePack Class Methods

The rule pack manager instance provides methods to access rule pack data.

The Oracle JAF rule pack manager provides the following methods that you can call on an instance of a `RulePack` object that you obtain from the `Rule` context object.

Method	Description
getPackInfo()	Returns an object that contains a read-only copy of the rule pack summary information. Properties include **path**, **prefix**, **title**, **version**, **enabled**, **status**.
getRuleCustomOptions(*string* **ruleName,** *string* **packPrefix)**	Returns an object that contains read-only copy of the custom options for a rule. That is, those specified by the user, and not JAF system rule options. Both arguments are optional: if **packPrefix** is omitted, the current rulepack is assumed. If **ruleName** is omitted the calling rule's options are returned.
getRuleCustomOption(*string* **property,** *string* **ruleName,** *string* **packPrefix)**	Returns the specified custom option property value for the rule, or *null* if not found. The **ruleName** and **packPrefix** arguments are optional. If **packPrefix** is omitted, the current rulepack is assumed. If **ruleName** is omitted the calling rule's option property is returned.
getRuleOptions(*string* **ruleName,** *string* **prefix)**	Returns an object that contains a read-only copy of the options for a rule. Both arguments are optional: if **prefix** is omitted, the current rule pack is assumed. If **ruleName** is omitted the calling rule's options are returned.
getRuleOption(*string* **property,** *string* **name,** *string* **prefix)**	Returns the specified rule option property for the rule. Arguments **name** and **prefix** are optional: if **prefix** is omitted, the current rule pack is assumed. If **ruleName** is omitted the calling rule's option property is returned.
getPrefix()	Returns a string that is the rule pack prefix.

Method	Description
setRuleOptions(*string* **ruleName,** *Object* **options)**	Sets the options for the rule in the caller's rule pack. The **rulename** argument is optional - if omitted, the calling rule's options are updated. **options** is an object containing the property/values to apply. Returns `true` if the options are successfully applied, else `false`.
isRuleEnabled(*string* **ruleName,** *string* **prefix)**	Returns `true` if the specified rule is currently enabled. The **prefix** argument is optional, and if omitted, the rule is assumed to be in the caller's rule pack.
isRuleDisabled(*string* **ruleName,** *string* **prefix)**	Returns `true` if the specified rule is currently disabled. The **prefix** is optional, and if omitted, the rule is assumed to be in the caller's rule pack.
disableRule(*string* **ruleName)**	Disables the specified rule in the caller's rule pack. Returns `true` if the rule was disabled (or was already disabled) - `false` if the **rulename** is invalid, or the rule was never registered.
enableRule(*string* **ruleName)**	Enables the specified rule in the caller's rule pack. Returns `true` if the rule was enabled (or was already enabled) - `false` if the **rulename** is invalid, or the rule was never registered. Note: In the current release, only a previously loaded rule (at startup) can be re-enabled. That is, `enableRule()` will not be successful, if the rule was discarded during initialization for any reason, and its `register()` method was thereby never called.
getRuleCount()	Returns the number of rules defined in the calling rule's rule pack.
getEnabledRuleCount()	Returns the number of rules currently enabled in the calling rule's rule pack.
getRuleList()	Returns a list of the rule names in the calling rule's rule pack.
getEnabledRuleList()	Returns an array list of the names of the currently enabled rules in the calling rule's rule pack.
getRule()	Returns the instance of a rule in the caller's rule pack to allow its exported methods to be called.
getExtension()	Returns the rule pack specific data supplied by `setExtension()`.
setExtension(* **data)**	Sets rule pack specific data that can be retrieved via `getExtension()`. `setExtension()` is typically used in a rule pack's **startupRP** hook rules. **data** can be any data type, and can be used to refer to any data, libraries, and son on that is useful to the rule.

Method	Description
sendMsg(*string* **ruleName, ...data)**	Sends a data message to the `onMsg()` method of a specific rule in the same pack as the caller. **data** represents any number of arguments - these arguments are presented to `onMsg()` as an array : **onMsg(***Object* **sender, *[] data);** where **sender** is an object of the following format: `{` `  sender : <string>, // the sending ruleName` `  regType : <string> // the register() type of the calling rule` `}` Returns the value returned by `onMsg()`. Note that a rule cannot send a data message to itself (to mitigate race conditions).
broadcastMsg(*string[]* **ruleNames, ...data)**	Send a data message to the `onMsg()` method of a list of rules in the same pack as the caller. If **ruleNames** is null, the data is broadcast to all currently enabled rules with an exported `onMsg()` method. Returns the number of rules whose `onMsg()` was invoked. Refer to `sendMsg()` for the format of **data**.
getInfo()	Returns miscellaneous information about the JAF instance as an object that has the following properties as strings. • **version**: the version formatted as "major:minor:patch" • **description**: a description of the JAF instance • **packageVer**: the JET code base that JAF was built on, formatted as "major:minor:patch" • **platform**: the platform the JAF instance was invoked on

Rule Issue Class Methods

You can create an `Issue` object and use the provided methods of the Oracle JAF `Issue` class to set properties of the reported issue.

The Oracle JAF `Issue` class provides the following constructor and setter methods that you can use to set the properties of the `Issue` object that you create within the registered event listener function of your audit rule.

The typical usage of the `Issue` class is to call the constructor method to create an `Issue` instance with details like the issue description and optional message ID:

```
var myIssue = new context.Issue(msg [, messageID]);
```

The severity for the rule is obtained from the rule definition, but this may be overridden within a rule using `Issue.setSeverity()`.

Method	Description
constructor(*string* **msg [,** string **id])**	**msg**—(optional) a string that is the issue description. **id**—(optional) a string that is an optional message ID. This will appear in the audit output as `ppp-nnnn`, where `ppp` is the rule pack prefix and `nnnn` is the message ID.
setId(*string* **id)**	Sets the id portion of a message id. **id**—a string that is the trailing ID in a message number. For example, if the **id** is `1234` and the rule's prefix is `ABC`, then the message number will appear as `ABC-1234`.
setPosition(*number* **line,** *number* **column,** *number* **startIndex,** *number* **endIndex)** or **setPosition(***Object* **pos)**	Overrides the line/column numbers in the audited file that appear in the reported message or JSON. **line**—the line number in the audited data. **column**—the column number in the audited data. **startIndex**—(optional) the starting position in the audited data. **endIndex**—(optional) the ending position in the audited data. Note that **line**/**column** may be specified as null, in which case the line and column will be computed by JAF from **startIndex**. May also be specified as an object, for example: `issue.setPosition(pos);` where **pos** is an object with the following format: **line**—the line number. **column**—the column number. **startIndex** or **start**—the start index. **endIndex** or **end**—the end index. The **pos** object may either be constructed by a rule or can be the return value from `DomUtils.getAttribPosition()`, `DomUtils.getAttribValuePosition()`, or `CssUtils.getPosition()`.
setSeverity(*string* **severity)**	Sets the severity of the issue (overriding the rule severity defined in the `rules.json` file, or the modified severity in configuration property **ruleMods**. May be **info**, **minor**, **major**, **critical**, or **blocker**.
setMsgKey(*string* **id)**	Sets the message ID for the issue. This overrides the message key obtained from the rule pack's `msgid.json` file if it exists.
setMsgEx(*Object***)**	Sets an extended message information object on the issue. Note that this is only meaningful when the output is not prose. It is typically used to supply information extracted from the audit message string in an easy to access form, to avoid the use of the output object from having to parse the audit message. It can also be used to augment the issue with any additional details the rule might want to offer. The Object supplied by this method will be added to the output JSON or message object, as property **msgEx**. JAF does not use or inspect the Object's contents.
setReportedFilePath(*string* **filepath)**	Sets the specified file path as the reported file path for the issue. This is designed for use where an issue is being reported by a rulepack lifecycle listener (e.g., `closedownRP`) for a file that was audited earlier. Since no file is actually audited when the issue is created, the issue cannot automatically report a file path. This method permits the rulepack to associate a specific file path to the issue for reporting purposes. Note: This is not required for normal audit cycle rules.

Rule Reporter Class Methods

You can create an `Reporter` object and use the provided methods of the Oracle JAF `Reporter` class to handle reported issues.

The Oracle JAF `Reporter` class provides the following methods that you can use to handle reporting of the `Issue` object that you pass to the `Reporter` instance in the registered event listener function of your audit rule.

The typical usage of a `Reporter` instance is to call the `addIssue()` method:

```
context.reporter.addIssue(myIssue, context);
```

Method	Description
addIssue(*Object* **issue, *Object* context [, *string* severity])**	Adds an `Issue` object to be reported. **issue**—an `Issue` object created via `new context.Issue(...)` . **context**—the context object passed to the rule. **severity**—(optional) allows a rule to override the rule severity. May be **infor**, **minor**, **major**, **critical**, or **blocker**.
clearIssues()	Removes all issues added to the `Reporter` instance for the current file.
getCount()	Returns the current number of issues added to the `Reporter` instance for the current file.
getFormat()	Returns the current output format: `prose` or `json`.

15

Reference: Custom Audit Rule Utility Libraries

Use this reference to obtain details about the utility libraries provided by Oracle JAF. These libraries provide miscellaneous helper functions that can be useful when you write custom audit rules.

Utility Library	Description
DomUtils: Node Object Functions	DomUtils is a collection of Document Object Model (DOM) utility functions and helper functions.
MetaLib: JET Metadata Access Functions	MetaLib provides Oracle JAF audit metadata access functionality that can be used in a rule.
Utils: General Non-File System Functions	Utils is a collection of non-file system utility functions.
FsUtils: File System Functions	FsUtils is a collection of file system utility functions.
SemVerUtils: Semantic Version Functions	SemVerUtils is a collection of semantic version (SemVer) utility functions.
JafLib: JAF Core Access Methods	JafLib is a library of exposed core Oracle JAF lifecycle functions.
MsgLib: Message Display Functions	MsgLib is a namespace property providing access to Oracle JAF internal messaging routines.
CssUtils: CSS Utility Functions	CssUtils is a library of CSS rule processing functions.
AstUtils: JavaScript File Helper Functions	AstUtils is a collection of Abstract Syntax Tree (AST) helper functions.
SevLib: Severity Support Helper Functions	SevLib is a library of helper functions that support processing severities that have been remapped via the Oracle JAF configuration property **sevMap**.
TsxUtils: TSX Utility Functions	TsxUtils is a library of TSX rule processing functions.

MetaLib: JET Metadata Access Functions

MetaLib provides Oracle JAF audit metadata access functionality that can be used in a rule.

Access these Oracle JET audit metadata interface library functions through rule context object **context.utils.metaLib**. The methods in this library insulate your rules from changes to the format of Oracle JAF metadata. Use these methods when writing audit rules that require access to metadata.

Oracle JET Audit Metadata Interface Library Metadata Methods

The Oracle JET audit metadata interface library provides the following metadata methods.

Method	Returns	Description
getTagMetadata()(*string* **tag)**	Object | null	**tag** is the HTML element name. Returns the `component.json` metadata as an object for the specified user-custom component.

Method	Returns	Description
getTagPropMetadata(*string* **tag,** *string* **propName)**	Object | null	**tag** is the HTML element name (specified with/without the surrounding chevrons). **propName** is the property/sub-property name. Returns the user web component property metadata as an object from the `component.json` metadata for the specified component property. The property name may specify dot-separated sub-properties. For example: `metaLib.getTagAttrMetadata('component-name', 'prop.subprop1.subprop2')` This is a convenience method to reduce the rule code required to check for the existence of the property.
getTagSlotMetadata(*string* **tag,** *string* **slotName)**	Object | null	Returns the 'slot' object from the component metadata
getAllCustomMetadata()	Object	Returns an object containing all user custom component metadata. The object keys are the component names: `{ component_name : { folder : string, // the containing folder (in native platform format) of: // 1) the component.json, or // 2) the VComponent source from which the metadata was // compiled. json : object, // the component.json as an object VCType: 'func' | 'class' // if a VComponent (JAF 8.9.0 or later) }, . . . }`
getTagStatus(*string* **tag,** *string* **statusType)**	Array<Object> | null	Returns an array of metadata `status` objects of the type specified by statusType. **tag** is the web component name **statusType** is an optional metadata `status` type. Possible values are `deprecated`, `maintenance`, `antiPattern` or `supersedes`. This parameter is optional - if omitted, all status objects at the component level are returned. `null` is returned in all other cases if no status objects can be found, or `tag` or `statusType` are invalid.

Method	Returns	Description
getPropStatus(*string* **tag,** *string* **prop,** *string* **statusType)**	Array<Object> | null	Returns an array of metadata `status` objects of the type specified by statusType. **tag** is the web component name **prop** is a component property name **statusType** is an optional metadata `status` type. Possible values are `deprecated`, `maintenance`, `antiPattern` or `supersedes`. This parameter is optional - if omitted, all status objects at the component level are returned. `null` is returned in all other cases if no status objects can be found, or `tag` or `statusType` are invalid.
isTagStatus(*string* **tag,** *string* **statusType)**	boolean	Returns `true` if the component has the status type specified by `statusType` at the component level. **tag** is the web component name **statusType** is an optional metadata `status` type. Possible values are `deprecated`, `maintenance`, `antiPattern` or `supersedes`.

Oracle JET Audit Metadata Interface Library Tag Methods

The Oracle JET audit metadata interface library provides the following tag methods.

Method	Returns	Description
isWCTag(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if an HTML tag element is a known JET or user-defined custom web component.
isJetTag(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if an HTML tag element is a known JET element (such as `<oj-avatar>`).
isJetLegacyTag(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if an HTML tag element is a known JET legacy element (`oj-*`).
isComponentTag(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if an HTML element is a known user-defined, custom Web Component (also called a composite component). The `component.json` files found via the **component** property of the configuration file are examined. See also **isWCTag()** .
isTagDeprecated(*string* **tag)**	string | null	**tag** is the HTML element name (specified with/without the surrounding chevrons). Tests a JET component for deprecated status, and returns the suggested alternative from the JET JS Doc if it is. Returns null if the element is not deprecated.

Method	Returns	Description
isTagAttrDeprecated(*string* **tag,** *string* **attrName)**	Array.<Object> | null Object | string | null	**tag** is the HTML element name. **attrName** is the attribute name. Tests the web component attribute for deprecated status and returns information including a suggested alternative and the version when deprecated. Multiple entries might be returned, and each object's `target` property should be checked for relevance. Each object is of the `component.json` deprecated type `status` object format. *Prior to JAF 3.1.0:* Tests the web component attribute for deprecated status, and returns the suggested alternative. If there is no alternative, the string is empty. Returns null if the attribute is not deprecated. If an object is returned, it will contain one or both of the properties `since` and `description` (both strings). Returns null if the attribute is not deprecated.
isTagAttr(*string* **tag,** *string* **attrName)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **attrName** is the attribute name. Returns `true` if the attribute is defined for the specified custom component, else `false`.
isTagAttrValue(*string* **tag,** *string* **attrName,** *string* **value)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **attrName** is the attribute name. **value** is the attribute value. Returns `true` if the attribute value is defined for the specified custom component name, else `false`.
hasTagAttrValues(*string* **tag,** *string* **attrName)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **attrName** is the attribute name. Returns `true` if the custom element attribute has at least one defined value, else `false`.
hasTagSlot(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if the JET component supports the `slot` attribute.
hasTagDefaultSlot(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if the JET component has a default slot.
hasTagDynamicSlot(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if the JET legacy (`oj-*`), JET Core Pack (`oj-c-*`) or use-defined component supports dynamic slot.
isTagSlotName(*string* **tag,** *string* **slotName)**	boolean	Returns `true` if the web component provides the specified slot name. Since JAF 6.1.7, `""` may be used to represent the default slot for the **slotName** argument.

Method	Returns	Description
getPreferredSlotContent()	Array<string>	Returns the preferred content (interfaces) for a specified component **tag** and **slot**. **slot** may be defined as `""` for the default slot. `getPreferredSlotContent("oj-chart","groupTemplate")  --> ["ChartGroupElement"]`
getTagAttrType(*string* **tagName,** *string* **attrName)**	string | null	**tagName** is the web component name (specified without the surrounding chevrons). **attrName** is the name of the attribute. Returns the web component element property type, for the specified attribute, from the component metadata for the property.
isTagSlotValue(*string* **tag,** *string* **slotValue)**	boolean	Deprecated in JAF 3.1.0. Use `isTagSlotName()` instead.
isTagEvent(*string* **tag,** *string* **attrName)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **attrName** is the attribute name. Returns `true` if a custom component attribute is a defined event. For example: `on-oj-action` for that component.
isTagSupportedInTheme(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if the specified web component name is supported in the current theme.
isAttrSupportedInTheme(*string* **tag,** *string* **attrName)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **attrName** is the attribute name. Returns `true` if the specified attribute in the web component is supported in the current theme.
isTagStyleDeprecated(*string* **tag,** *string* **styleName)**	string | null	**tag** is the HTML element name (specified with/without the surrounding chevrons). **styleName** is a `class` attribute style name. Tests the JET component `class` attribute has deprecated status, and returns the suggested alternative. If there is no alternative, the string is empty. Returns null if the style is not deprecated.
isTagStyle(*string* **tag,** *string* **styleName)**	string | null	**tag** is the HTML element name (specified with/without the surrounding chevrons). **styleName** is a `class` attribute style name. Returns `true` if the specified CSS `class` style is defined for the specified JET element name.
isStyle(*string* **styleName)**	boolean	**styleName** is a `class` attribute style name. Returns `true` if the CSS style is a defined JET style, else `false`.
isGenStyle(*string* **styleName)**	boolean	**styleName** is a `class` attribute style name. Returns `true` if the CSS style is a generic JET (non-component specific) style, else `false`.

Method	Returns	Description
isClassStyle(*string* **styleName)**	boolean	**styleName** is a `class` attribute style name. Returns `true` if the CSS style is a known JET component style, else `false`.
isJetStylePrefix(*string* **styleName)**	boolean	**styleName** is a `class` attribute style name. Returns `true` if the CSS style begins with a known JET style prefix, else `false`. Note that `false` can also be returned if the loaded metadata is for an older JET version and there is no prefix data available. In such cases **hasJetStylePrefixes()** can be used to determine if the `false` value is meaningful. .
hasJetStylePrefixes()	boolean	**styleName** is a `class` attribute style name. Returns `true` if the loaded metadata contains JET style prefix information.
getTagsFromStyle(*string* **styleName)**	string[] | null	**styleName** is a `class` attribute style name. Returns the JET element names using the specified style name. For example, for the style `oj-form-control-text-align-right`, the returned value would be [`oj-input-number`, `oj-input-text`].
getClassesFromStyle(*string* **styleName)**	string[] | null	**styleName** is a `class` attribute style name. Returns the classes using the specified style name. For example, for the style `oj-button-sm`, the returned value would be [`oj.ojButton`, `oj.ojMenuButton`].
getClassFromTag(*string* **tagName)**	string | null	**tagName** is the HTML element name (specified with/ without the surrounding chevrons). Returns the class (e.g.,`oj.ojTable`) for a given JET element name (such as `oj-table`). Returns null if **tagName** is not a defined JET element name.
getTagFromClass(*string* **className)**	string | null	**className** is a JET class name. Returns the JET element name (e.g., `oj-table`) for a given class name (such as `oj.ojTable`). Returns null if **className** is not a known JET class.
getClassFromModule(*string* **module)**	string | null	**module** is a specification such as `ojs/arraydataprovider`. Returns the class (e.g., `oj.ArrayDataProvider`) for a given module path (such as `ojs/arraydataprovider`).
getTagPropType(*string* **tagName,** *string* **propName)**	string | null	This method is deprecated as of JAF release 7.0.0. You should now use **getTagAttrType()**. **tagName** is the web component name (specified without the surrounding chevrons). **propName** is the name of the property specified in the `component.json` **properties**. For web components that have been inspected by JAF during startup because they are defined in the Oracle JAF configuration property **components**. Returns the property type (**type** sub-property) defined for a property in the Oracle JAF `component.json` file.

Method	Returns	Description
getTagPropertyFromEvent(*string* **tagName,** *string* **eventAttrName)**	string | null	Returns a property name from a tag event attribute. For example: `getTagPropertyFromEvent('oj-pop-up', 'on-oj-before-close') --> 'ojBeforeClose'` `getTagPropertyFromEvent('oj-pop-up', 'onOjBeforeClose') --> 'ojBeforeClose'`
isBindingTag(*string* **tag)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). Returns `true` if the element tag is defined with `@ojbindingelement` (such as `oj-bind-if`, or `oj-bind-for-each`).
isClass()	boolean	Returns `true` if component/class is a JET class name (e.g., `oj.ArrayDataProvider`).
isClassRenamed(*string* **className)**	Object | null	**className** is a JET class name (e.g., `oj.ArrayDataProvider`). Returns an object with the new name and the JET version in which it was renamed. For example: `{ to : "new class name", since : "x.y.z" }` Returns null if the class is not renamed.
isClassDeleted(*string* **className)**	string | null	**className** is a JET class name (e.g., `oj.ojAccordian`). Test if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) is deleted from a version, and if so returns the (semantic) version in which it was deleted. Returns `null` if not deleted.
isClassDeprecated(*string* **className)**	string | null	**className** is a JET class name (e.g., `oj.ojAccordian`). Test if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) has deprecated status, and returns the suggested alternative if it is. The suggested alternative is a string supplied by the developer in the class's API doc. Returns `null` if not deprecated.
isClassMethodDeleted(*string* **className,** *string* **methodName)**	string | null	**className** is a JET class name (e.g., `oj.ojAccordian`). **methodName** is a method of the class. Test if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) method is deleted from a version, and if so returns the version in which it was deleted. Returns `null` if the method is not deleted.

Method	Returns	Description
isClassMethodDeprecated(*string* **className,** *string* **methodName)**	Array.<Object> | null string | null	**className** is a JET class (e.g., `oj.ojAccordion`). **methodName** is a method of the class. Tests if a JET component/class (e.g., `oj.ojDialog`) method has deprecated status and returns information including a suggested alternative and the version when deprecated. Multiple entries might be returned, and each object's `target` property should be checked for relevance. Each object is of the `component.json` deprecated type `status` object format. *Prior to JAF 3.1.0*: Tests if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) method has deprecated status, and returns the suggested alternative if it is. The suggested alternative is a string supplied by the developer in the class's API doc. Returns `null` if the method is not deprecated.
isClassMemberDeleted(*string* **className,** *string* **memName)**	string | null	**className** is a JET class name (e.g., `oj.ojAccordian`). **memName** is an attribute name. Test if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) member is deleted from a version, and if so returns the version in which it was deleted. Returns `null` if the member is not deleted.
isClassMemberDeprecated(*string* **className,** *string* **memName)**	Array.<Object> | null string | null	**className** is a JET class (e.g., `oj.ojAccordion`). **memName** is an attribute name. Tests if a JET component/class (e.g., `oj.ojDialog`) method has deprecated status and returns information including a suggested alternative and the version when deprecated. Multiple entries might be returned, and each object's target property should be checked for relevance. Each object is of the `component.json` deprecated type `status` object format. *Prior to JAF 3.1.0*: Tests if a component/class (e.g., `oj.RouterState` or `oj.ojTree`) member has deprecated status, and returns the suggested alternative if it is. The suggested alternative is a string supplied by the developer in the class's API doc. Returns `null` if the member is not deprecated.
isClassFinal(*string* **className)**	boolean	**className** is a JET class name (e.g., `oj.Module`). Returns `true` if the class is `final`.
isClassMethodFinal(*string* **className,** *string* **methodName)**	boolean | null	**className** is a JET class name (e.g., `oj.Module`). **methodName** is a method of the class. Returns `true` if the method is static, else `false`. Returns null if the **className** or **methodName** is undefined.
getClassMethodScope(*string* **className,** *string* **methodName)**	string | null	**className** is a JET class name (e.g., `oj.Module`). **methodName** is a method of the class. Returns `instance` or `static`. Returns null if the **className** or **methodName** is undefined.

Method	Returns	Description
getDeletedMethodList()	Object	Make a list of deleted methods as a rule helper. Returns `{Array.<Object>}` with an object with method names as the properties. Each property value is an array of class names in which it was deleted, followed by the version in which it was deleted.
getDeletedMemberList()	Object	Make a list of deleted members as a rule helper. Returns `{Array.<Object>}` with an object with member names as the properties. Each property value is an array of class names in which it was deleted, followed by the version in which it was deleted.
isRuleIgnored(*string* **tag,** *string* **ruleName)**	boolean	**tag** is the HTML element name (specified with/without the surrounding chevrons). **ruleName** is the name of the rule to be tested for exclusion. Tests if the rule is excluded for the component by its appearance in the component's optional `extension` metadata. For example: `{` `  . . .` `  "extension" : {` `  "audit" : {` `  "ignore" : [rulename1, rulename2, ...]` `  }` `  }` `  . . .` `}` The method returns `true` if the rulename is declared in the component's metadata, else `false`.
getRenamedClassList()	Object	Get a list of JET classes that have been renamed. The following object is returned: `{` `  "old_classname":` `{  // e.g. "oj.ArrayDataProvider"` `  to  : "new_classname",  // e.g. "ArrayDataProvider"` `  since : "a.b.c"  // e.g.  "8.2.0"` `  },` `  . . .` `}`

Method	Returns	Description
getRevisionInfo()	Object	Returns an object with the following properties: • **jetVersion** - The JET version of the metadata. (string) • **revision** - The JET source revision info, currently from GIT. (string) • **tag** - The JET source GIT tag. (string)
getMetaVers()	Object	Returns an object with the following properties: • **versions** - The JET version supported, and the build dates. (Object) `{` `"6.2.2" : {` `"date" : "ddddddd",` `"time" : "ttttttt"` `},` `. . .` `}` • **default** - The default JET version, if configuration property **metaVer** is omitted. (String)
getSubcomponentType()	string	Returns the component **subcomponentType** metadata property as a string (e.g., **packPrivate**, **data**, etc.).
isInterfaceImplemented()	(string tagName, string interfaceName)	Returns true if the specified interface name is implemented by the tagName component
getInterfaces()	Array<string>	Returns an array containing the names of the interface(s) implemented by the referenced component **tagName** (e.g., `oj-button`).
getImplementers()	Array<string>	Returns an array of the names of components implementing the specified interface.
hasJetWCInterfaces()	boolean	Tests whether the selected JET metadata (via the config `jetVer` property) contains web component interface information.
hasWCInterfaces()	boolean	Tests whether any web components known to JAF (JET or non-JET) have implemented interfaces.
getWCInterfaces()	Object	If any web component known to JAF declares an implemented interface, an object is returned of the form: `{` `interface_name_1 : [component_tag_1, . . .],` `interface_name_2 : [component_tag_3, . . .],` `. . .` `}`

Method	Returns	Description
walkDomStackForOJTag(*Object* **context [,** *boolean* **excludeBindIf])**	string | null	**context** is the context object passed to the rule. Convenience method. Walks back up the DOM from the current element to see if there is a containing JET parent element. If found, the name of the JET element is returned. Optional argument **excludeBindIf** can be used to ignore `<oj-bind-*>` elements during the walk.
getStyleOrigin()	string	Returns a code denoting if the style is a JET component style, a JET generic style class, or a non-JET user custom style.
isTagPackPrivate()	boolean	Returns **true** if the web component is marked as `packPrivate` in its metadata.

Utils: General Non-File System Functions

Utils is a collection of non-file system utility functions.

Access these non-file system utility functions through the rule context object property **context.utils.utils** as an instance of a library object. They may be helpful when writing audit rules.

Method	Returns	Description
hasAnyProps(*Object* **object)**	boolean	Returns `true` if the object has any non-inherited properties
isProperty(*Object* **object,** *string* **name)**	boolean	Returns `true` if the object has the named property. The property name can be a compound dot-separated name, such as `extension.catalog.readme`.
getProperty(*Object* **object,** *string* **name)**	*	Returns the object's named property value. The property name can be a compound dot-separated name, such as `extension.catalog.readme`.
getType(* **varName)**	string	Returns the type of a variable. This function returns a `true` type value. For example: the `typeof` operator returns `object` for an array, whereas this function returns `array`. The following values can be returned (all lower-case) : `object`, `array`, `string`, `number`, `boolean`, `null`, `undefined`.
isArrayContentsType(*array[]* **array,** *string* **type)**	boolean	Traverses an array, matching each element's type to the supplied (lower-case) type string. The **type** string can be one of the following : `object`, `array`, `string`, `number`, `boolean`, `null`, or `undefined`
eatWhitespace(*string* **string,** *number* **startIndex)**	number	Traverses a string, optionally starting at the **startIndex** index, and returns the index to the first non-whitespace character. Returns `1` if no non-whitespace is found. If **startIndex** is omitted or invalid, it defaults to zero.
getIndexToWhitespace(*string* **string,** *number* **startIndex)**	number	Traverses a string, optionally starting at the **startIndex** index, and returns the index to the first whitespace character. Returns `-1` if no whitespace is found. If **startIndex** is omitted or invalid, it defaults to zero.
decommentJson()	string	Replaces the `//` and `/* */` sequences in a JSON string with blanks to preserve the relative line/column positions of the content tokens.

Method	Returns	Description
parseJson()	*	Parses a JSON string and returns the parsed object, or an error object containing a syntax error message together with supplemental position information that may be used to augment the message, if required.

FsUtils: File System Functions

FsUtils is a collection of file system utility functions.

Access these file system utility functions through the rule context object property **context.utils.fsUtils** as an instance of a library object. They may be helpful when writing audit rules.

Method	Returns	Description
getFileTypeSync(*string* **filePath)**	string | Error	Checks the file at **filePath** and returns: • **f** - file • **d** - directory If an error occurs, an Error exception object is returned. Note if the path is a symbolic link, the file type of the link target is returned (for example, "f" or "d"). Use **isSymLink()** to determine if the path represents a symlink.
getFileExtSync(*string* **filePath)**	string	Returns the file extension for the supplied path. If the path is a symbolic link, the file extension for the link's target file is returned. If the file path cannot be accessed, an empty string is returned.
isSymLinkSync(*string* **filePath)**	boolean	Returns `true` if the file path is a symbolic link, else `false`. In case there is an error processing the file path, `false` is returned.
pathExistsSync(*string* **filePath)**	boolean	Returns `true` if the file path exists.
fileExistsSync(*string* **filePath** [, *boolean* **dir**] **)**	boolean	Returns `true` if file/directory exists, else `false`.
readFileSync(*string* **filePath** [, *boolean* **error** | *Function* **error**(string **message**)]	string | null	Reads a text file and returns the contents as a string, or null if unsuccessful. If the file read is unsuccessful, the action taken depends on optional argument `error`: • If boolean `true`, an exception is thrown. • If boolean `false`, an error string is returned. • If omitted, the function returns null. • If a function is defined, it will be called with a string argument representing the error message, and the return value of `readFileSync()` will be null.

Method	Returns	Description
readFileBufSync(*string* **filePath** [, *boolean* **error** | *Function* **error**(string **message**)]	Buffer | null	Reads a text file and returns the contents as a Buffer, or null if unsuccessful. If the file read is unsuccessful, the action taken depends on optional argument `error`: • If boolean `true`, an exception is thrown. • If boolean `false`, an error string is returned. • If omitted, the function returns null. • If a function is defined, it will be called with a string argument representing the error message, and the return value of `readFileBufSync()` will be null.
readJsonSync(*string* **filePath** [, *boolean* **comments** [, *boolean* | *Function* **error**]] **)**	Object | string | null	Reads a JSON file and returns the object. If optional argument **comments** is set to `true`, comments are permitted in JSON. Optional argument **error** can be used to indicate how errors should be handled: • If `true`, an exception is thrown. • If `false`, an error string is returned. • If omitted, the function returns null. • If a function is defined, it will be called with a string argument representing the error message, and the return value of `readJsonSync()` will be null. Error strings are prefixed with `SYNTAX:` or `NOFILE:`.
writeJsonSync(*Object* **obj**, *string* **filePath** [, *Object* **opts** [, *boolean* | *Function* **error**]] **)**	boolean | string	Writes an object to the specified file path as a JSON file, and returns `true`. Optional object **opts** has the following optional properties: • **spaces** can be a number to indent each line, or a string to prefix each line. • **replacer** a JSON `stringifer()` replacer. Optional argument **error** can be used to indicate how errors should be handled: • If `true`, no return value (error thrown) • If `false`, an error string is returned. • If a function is defined, it will be called with a message string argument, and `writeJsonSync()` returns `false`. If **error** is omitted, `false` is returned.

Method	Returns	Description
readDirSync(*string* **filePath)**	Array | null	Reads a directory and returns a hierarchical array of objects of the following form: `[` `  {` `  name : "filename1",  // for file in a folder` `  isFile : true` `  },` `  {` `  name : "directory1",  // for each sub-folder` `  isFile : false,` `  files : [` `  {` `  name : "filename1",` `  isFile : true` `  },` `  ...` `  ]` `  },` `  ...` `]`

Method	Returns	Description
walkDirSync(*string* **dirPath**, *Function* **cb**, [*boolean* **fwdSlash**])	none	Recursively walk a directory tree invoking a callback function with the full path to the file or sub-directory. **dirPath** is the initial directory path to walk. **cb** is a callback function that receives the following args: `(string fpath, string ftype)` `fpath` is the full file or sub-directory path `ftype` = `'f'` for a file, `'d'` for a sub-directory, `'l'` for a symbolic link. Optional arg **fwdSlash** set to *true* forces the *fpath* string in the callback to use the forward slash in place of the Windows escaped "\\" if found. Normally the callback function returns nothing - however it can optionally return `"exit"` to terminate the walk, `"direxit"` to terminate further walking of the current sub-directory For example, to find all .JS files in a directory tree: `walkDirSync("some/path", (fpath, ftype) => {` `if (ftype === "f" && fpath.endsWith(".js")) {` `// do something` `}` `}) ;` When reading the files in a directory, if a sub-directory is found, that sub-directory is examined next, and so on.
createFolderSync(*string* **filePath)**	boolean	Create a folder. The file path can contain multiple folders that also need to be created. For example: `createFolder("./folder1/folder2/folder3") ;` Note: folder3, folder2 and folder1 are created if they don't exist.
deleteFolderSync(*string* **filePath [,** *boolean* **deleteTarget])**	none	Deletes the contents (files/directories/sub-directories) of the directory specified by **filePath**. If optional argument **deleteTarget** is specified, and is `true`, the directory specified by **filePath** is also deleted.

Method	Returns	Description
getUniqueFilenameSync(*string* **template)**	string	Returns a unique file name based on a template. All instances of the letter X are replaced. For example: `getUniqueFilename('@@users-XXXXXX.tmp')` could return `@@users-wKFMN5`.

SemVerUtils: Semantic Version Functions

SemVerUtils is a collection of semantic version (SemVer) utility functions.

Access these semantic version utility functions through the rule context object property **context.utils.semVerUtils**. They may be useful when writing audit rules. All method arguments are strings.

Method	Returns	Description
isValid(*string* **semver)**	boolean	Returns `true` if the **semver** string is valid semantic version syntax.
isValidRange(*string* **semver)**	boolean	Returns `true` if the **semver** string is a valid range of semver comparators.
satisfiesRange(*string* **semver,** *string* **range)**	boolean	Returns `true` if **semver** satisfies the **range**.
minVersion(*string* **range)**	string | null	Returns a *major.minor.patch* string representing the lowest version that can possibly match the given range. Null is returned if **range** is invalid. For example: `minVersion('>=1.0.0')` returns `'1.0.0'`. Requires JAF version 2.9.9 or later.
prerelease(*string* **semver)**	Array | null	Returns an array of prerelease tags extracted from **semver**. If **semver** does not contain any prerelease tags (or **semver** is invalid), null is returned. For example: `prerelease('1.2.3-alpha.1')` returns `['alpha', 1 ]`. Requires JAF version 2.9.9 or later.
eq(*string* **semver1,** *string* **semver2)**	boolean	Compares two **semvers** and returns `true` if they are logically equivalent, even if they are not exactly same string.
lt(*string* **semver1,** *string* **semver2)**	boolean	Compares two semvers and returns `true` if **semver1** is less than **semver2**.
lte(*string* **semver1,** *string* **semver2)**	boolean	Compares two semvers and returns `true` if **semver1** is less than or equal to **semver2**.
gt(*string* **semver1,** *string* **semver2)**	boolean	Compares two semvers and returns `true` if **semver1** is greater than **semver2**.
gte(*string* **semver1,** *string* **semver2)**	boolean	Compares two semvers and returns `true` if **semver1** is greater than or equal to **semver2**.

Method	Returns	Description
major(*string* **semver)**	number	Returns the major version number from the **semver**.
minor(*string* **semver)**	number	Returns the minor version number from the **semver**.
patch(*string* **semver)**	number	Returns the patch version number from the **semver**.
parse(*string* **semver)**	Object | null	Parses **semver** into an object. Null is returned if **semver** is invalid. For example, `parse('1.2.3-alpha.1')` returns: `{` `"options" : {` `"loose" : false,` `"includePrerelease" : false` `},` `"loose" : false,` `"raw" : "1.2.3-alpha.1",` `"major" : 1,` `"minor" : 2,` `"patch" : 3,` `"prerelease" : [ "alpha", 1 ],` `"build" : [],` `"version" : "1.2.3-alpha.1"` `}` Require JAF version 2.9.9 or later.

DomUtils: Node Object Functions

DomUtils is a collection of Document Object Model (DOM) utility functions and helper functions.

Access these DOM utility function through rule context object property **context.utils.DomUtils**. Most methods take a **node** object, which is found in **context.node**, and some methods require an attribute name or element name string.

Method	Returns	Description
getName(*Object* **node)**	string	Returns the element name of the supplied node.
getType(*Object* **node)**	string	Returns the type of the node (`tag`, `directive`, or `comment`).
getChildren(*Object* **node)**	nodes[]	Returns an array of child node objects for the supplied node.
getParent(*Object* **node)**	node	Returns the parent node object of the supplied node.
getSiblings(*Object* **node)**	nodes[]	Returns an array of sibling node objects of the supplied node.
getAttribs(*Object* **node)**	Object	Returns an object containing attribute name/value properties.
getAttribValue(*Object* **node,** *string[]* **attrName)**	string	Returns the specified attributes value.

Method	Returns	Description
hasAttrib(*Object* **node,** *string* **attrName)**	boolean	Returns `true` if the node has the named attribute, else `false`.
hasChildren(*Object* **node)**	boolean	Returns `true` if the node has children, else `false`.
hasNext(*Object* **node)**	boolean	Returns `true` if there is a next node, else `false`.
getNext(*Object* **node)**	node	Returns the next node, else null.
getFirst()	node	Returns the first (i.e top) node.
getFirstElem()	node	Returns the first (i.e. top) tag node.
getLast()	node	Returns the last node.
getElemsByName(*string* **elemName)**	nodes[]	Returns an array of node objects with the specified element name (such as `div`).
getElemById(*string* **id)**	node	Returns the node for the element with specified `id` attribute value.
getBody()	node	Returns the `<body>` node (or null if not found).
getHead()	node	Returns the `<head>` node (or null if not found).
getLinks()	node[]	Returns an array of `<link>` tag nodes in the `<head>` section (or an empty array if none found).
getScripts()	node[]	Returns an array of `<script>` tag nodes in the `<head>` section (or an empty array if none found).
getElems()	nodes[]	Returns an array of node objects for all elements. (Nodes such as text, comments, directives, etc. are ignored.)

DomUtils also contains additional non-DOM tree helper functions that are useful when writing rules.

Method	Returns	Description
extractAttribsFromDataBind(*string* **attrValue)**	Object | null	Given a `data-bind` with an `attr` property, returns an object where each property is the name/value extracted from the `data-bind` string.
extractComponentFromDataBind(*string* **attrValue)**	string | null	Given a `data-bind` attribute value, returns the JET component name instantiated, or null if not found or not a JET component.
getAttribIndex(*string* **data,** *Object* **node,** *string* **attrName)**	number	Returns the position (relative to zero) in the data of the named attribute for the node.
getAttribPosition(*string* **data,** *Object* **node,** *string* **attrName)**	Object	Returns the position of the named attribute for the node in the data. The object has the following members: • **row** : number • **col**: number • **start**: number // index (relative to zero) to the attribute's value in the file data • **end**: number // index to the last character of the attribute's value in the file data This method is useful when a rule listens to tag node events but needs to report on the position of a particular tag attribute's value. This information can be applied to an `Issue` object via its `setPosition()` method.

Method	Returns	Description
getAttribValuePosition(*string* **data,** *Object* **node,** *string* **attrName)**	Object	Returns the position of the named attribute's value in the data for the tag node. The object has the following members: • **row**: number • **col**: number • **start**: number // index (relative to zero) to the attribute into the file data • **end**: number // index to the last character of the attribute in the file data This method is useful when a rule listens to **tag** or **ojtag** events, but needs to report on a particular attribute in the tag. This information can be applied to an `Issue` object via its `setPosition()` method.
getLineCol(*string* **data,** *number* **index number)**	Object	Returns the line and column for a given index into file data. The object has the following members: • **row**: number • **col**: number
getComponentElems()	nodes[]	Returns an array of nodes for JET custom elements.
getDataBindAttrs(*Object* **context)**	Object | `null`	Checks the current **node** in **context**, and if it contains a `data-bind` attribute with a `attr` property, returns an object where each property is the name/value extracted from the data-bind string.
getDataBindComponent(*Object* **context)**	string | null	Checks the current **node** in **context**, and if it contains a `data-bind` attribute, extracts the JET component name if defined in the attribute value,
getElemById(*string* **id [,** *boolean* **labelId])**	Object | null	Searches the DOM for a node with the specified `id` attribute value and returns the DOM tree node if found. If optional argument **labelId** is set to `true`, the search also includes the `label-id` attribute.
hasBody(*Object* **context)**	boolean	Returns `true` if the HTML page has a `<body>` element.
isChildOfElem(*string* **elemName,** *Object* **node)**	boolean	Returns `true` if the DOM tree node is a child of the element named `elemName`.
isCommonAttr(*string* **attrName)**	boolean	Returns `true` if the specified attribute is a standard defined xxx. (Note: if **attrName** is prefixed with '`:`', the prefix is ignored).
isCommonElem(*string* **elemName)**	boolean	Returns `true` if the specified element is a standard defined xxx.
isCommonEventAttr(*string* **attrName)**	boolean	Returns `true` if the specified attribute is a standard xxx defined xxx. (Note: if **attrName** is prefixed with '`:`', the prefix is ignored).
isHtml5ObsoleteElem(*string* **elemName)**	boolean	Returns `true` if the specified element is obsolete in HTML5
isHtml5ObsoleteAttr(*string* **elemName,** *string* **attrName)**	boolean	Returns `true` if the attribute is obsolete in HTML5 for the specified element. Note: the return value is unconditional. There may be additional exception conditions that need to be evaluated (for example, the `border` attribute on `<img>`) - refer to xxx.
isJetPage(*Object* **context)**	boolean	Returns `true` if there is at least element with a name starting with `oj-`

Method	Returns	Description
isNamespaceTag(*string* **tagName)**	boolean	Given a Web Component element name starting with `oj-` (e.g. `oj-ext-foo`), this returns `true` , if the prefix (`oj-ext` in this example) is defined in the xxx.
isNamespacePrefix(*string* **tagPrefix)**	boolean	Given a Web Component name prefix starting with `oj-` (for example `oj-ext`), this returns `true` , if the prefix is defined in the xxx .
isNonFragmentJetPage(*Object* **context)**	boolean	Returns `true` if the HTML page has a `<body>` and also has at least one JET component element.
isSvgElem(*string* **elemName)**	boolean	Returns true if the element name is an SVG element
isValidJson(*string* **json)**	boolean | string	Checks a string to see if it is JSON, and if so, validates it. If the string is valid JSON, or is not JSON, `true` is returned. If it is JSON and is not well formed a message string is returned. (Note: the JSON can contain // and /* . . . */ style comments - these are ignored.)
isValidSvgPath(*string* **svg)**	boolean	If the string appears to be a valid SVG path, `true` is returned. Note this does not validate the path for being syntactically correct SVG.
isExpression(*string* **)**	boolean	Returns `true` if the string has valid expression delimiters {{. . .}} and [[. . .]] and they are matched correctly.
getExpression(*string* **)**	string|null	Returns the expression contained within the expression delimiters {{. . .}} and [[. . .]]. If not an expression or the expression is not validly delimited, null is returned.
camelCase(*string* **prop)**	boolean	Returns a camel-cased string from a kebab-cased property string.
kebabCase(*string* **prop)**	boolean	Returns a kebab-cased string from a camel-cased property string.
kebabCaseEvent(*string* **prop)**	boolean	Returns a kebab-cased string from a camel-cased event name.

ConfigLib: Configuration Library

ConfigLib is a context member that provides convenient access to some configuration information.

Access these methods through the rule context object property **context.utils.configLib**.

Method	Returns	Description
getConfig()	Object	Returns a copy of the currently active configuration. See also **JafLib.getConfig()**.
getOrigConfig()	Object	The original configuration supplied to JAF before processing by JAF into the currently active configuration
getExtendsProfileName()	string | undefined	The profile name extended by the initial configuration (from configuration property `extends`).
isBuiltinJetRules()	boolean	`true` if JET built-in rules are enabled (from configuration property `builtinJetRules`).

Method	Returns	Description
isBuiltinCspRules()	boolean	`true` if JET built-in CSP rules are enabled (from configuration property `builtinCspRules`).
isBuiltinJetWcRules()	boolean	`true` if extended audits (JETWC) for customer component authors are enabled (from configuration property `builtinJetWcRules`).
isBuiltinJetWcOracleRules()	boolean	`true` if extended Oracle audits (JETWCO) for customer component authors are enabled(from configuration property `builtinJetWcOracleRules`).
isBuiltinOjcMigrationRules()	boolean	`true` if the built-in component migration rules are enabled (from configuration property `builtinJOjcMigrationRules`).
isBuiltinWebDriverTestRules()	boolean	`true`, if the built-in WebDriver test audit rules (from configuration property `builtinWebDriverTestRules`).

JafLib: JAF Core Access Methods

JafLib is a library of exposed core Oracle JAF lifecycle methods.

Access these methods through the rule context object property **context.utils.jafLib**.

Method	Returns	Description
setFileset(*Object***context,** *string[]* **fileset[,** *string[]* **exclude])**	boolean	Sets the file set to audit. This is a dynamic replacement for the **files** and **exclude** properties of the configuration file. **fileset** and optional **exclude** specify an array of full file paths. Globs may be used. **context** is the rule context object. Returns `true` if successful, else `false`. This method may be useful when writing hook rules. The method can be used only during the **startaudit** phase of the audit lifecycle.
getFileset(*Object* **context)**	Object	Returns the **files** and **exclude** configuration properties in an object with the same property names. **context** is the rule context object. **Do not modify the object property values!** To modify either **files** and/or **exclude** properties use **setFileset()**. Note that this can be used in conjunction with **setFileset()** to merge or remove entries. This method may be useful when writing hook rules. The method can be used only during the **startaudit** phase of the audit lifecycle.
getJafReleaseCount()	number	Returns the estimated number of JET major public releases since the JET version of the JAF package used.

Method	Returns	Description
getMajorReleaseCount(*string* **semver |** *number* **major)**	number	Returns the estimated number of JET major public releases since the JET major value in the supplied *semver* argument. The `semver` string can be a full or partial semver string (only the major value is used/ required), or the major value can be supplied as a number. If the `major` value is invalid or omitted, its default value is the JET major value of the version of JAF used.
getConfig()	Object	Returns an object with getter methods that return configuration property values. Refer to JafLib: Configuration Object Property Getter Methods.
getTsConfig()	Object	Returns the tsconfig.json as an object.
getRunMode()	string	Returns a string representing the JAF run mode. Values are **cli**, **api**, or **amd**. Currently, running in CLI mode only is possible.
isCLI()	boolean	Returns `true` if running in CLI mode (the only mode currently available).
isAPI()	boolean	Currently returns `false`. Note that API mode is reserved for future support.
isAMD()	boolean	Currently returns `false`. Note that AMD mode is reserved for future support.
getJafVer()	string	Returns the JAF version semver, for example: 3.12.4.
getJetVer()	string	Returns the JET metadata version semver, for example: 9.2.3.

JafLib: Configuration Object Property Getter Methods

JafLib exposes the **getConfig()** method to access JAF configuration property values.

The configuration object returned by **JafLib.getConfig()** contains methods returning configuration property values. Note that due to configuration analysis during startup, property values returned do not necessarily reflect the same values. For example, if the configuration property **theme** is not defined, **Config.getTheme()** will return the default value **alta** and not ***undefined***.

Method	Configuration Property Returned
getComponents()	**components**
getComponentsBase()	**componentsBase**
getComponentsBaseUrl()	**componentsBaseUrl**
getComponentOptions()	**componentOptions**
getDisable()	**ruleMods.disable**
getEcmaVer()	**cmaVer**
getEnable()	**ruleMods.enable**
getExclude()	**exclude**
getFiles()	**files**
getGroups()	**groups**

Method	Configuration Property Returned
getJetVer	**jetVer**
getMessages()	**messages**
getOptions()	**options**
getOutPath()	**outPath**
getRuleDescriptions()	**ruleDescriptions**
getRuleMods()	**ruleMods**
getRuleNames()	**ruleNames**
getRulePacks()	**rulePacks**
getTheme()	**theme**
getTypescript()	**typescript**
getTsConfig()	**typescript.tsconfig**
getTsConfigObj()	Returns the **tsconfig.json** found on the path defined by the **typescript.tsconfig** sub-property, as an object.
isBuiltinJetRules()	**builtinJetRules**
isBuiltinJetWcRules()	**builtinJetWcRules**
isBuiltinSpocRules()	**bultinSpocRules**

MsgLib: Message Display Functions

MsgLib is a namespace property providing access to Oracle JAF internal messaging routines.

Access these internal messaging routines through the rule context object property **context.utils.msgLib** as an instance of a library object. They may be helpful when you need to provide the ability for an audit rule to write conditional and unconditional messages to the output. Refer also to the context property **sysOpts**.

Method	Description
msg()	Writes a general message string to the console. For example: `msgLib.msg("this is a message")`.
info()	Writes the message string to the console only if verbose mode is on. For example: `msgLib.msg("this is only displayed in verbose mode")`. The message is preceded by `[info]:`.
debug()	Writes the message string to the console only if debug mode is on. For example:`msgLib.debug("this is only displayed in debug mode")`. The message is preceded by `[debug]:`.
error()	Writes the message to the console. For example: `msgLib.error("this is an error message")`. The message is preceded by `[error]:`.
assert()	Writes a message string to the console. For example: `msgLib.assert("this is an assertion message")`. The message is preceded by `[ASSERT]:`.

CssUtils: CSS Utility Functions

CssUtils is a library of CSS rule processing functions.

Access these functions through the context property **context.utils.CssUtils** on the register context object when the rule is invoked or on the rule context object passed to the registered listener. These functions may be useful when writing CSS rules.

Method	Returns	Description
isColorName(*string* **name)**	boolean	Returns `true` if the string is a CSS defined color (for example, "`coral`"). Accepts all color names to CSS Color Module Level 4.
isMarkerstyle(*string* **style)**	boolean	Returns `true` if the style string is an Oracle JET marker style.
getPosition(*Object* **context**, *Object* **loc)**	Object	Converts a **loc** position node in the CSS Abstract Syntax Tree (AST) to a position object suitable to pass to `Issue.setPosition()`. The resulting position object is adjusted for the origin of the CSS text.

AstUtils: JavaScript File Helper Functions

AstUtils is a collection of Abstract Syntax Tree (AST) helper functions.

Access these functions through the rule context object property **context.utils.AstUtils**. They may be useful when writing audit rules for JavaScript files. For example:

```
var node = ruleContext.utils.AstUtils.getBody();  // get the array of program
body nodes
```

Name	Returns	Description
getBlock(*Object* **node)**	node	Returns the block node in which the specified node resides. If node is not specified, the outer program block is returned.
getBody(*Object* **node)**	Array[]	Returns the `body` array of nodes containing the specified node. If node is not specified, the body of the program is returned.
getProgram()	node	Returns the Program node.
isFuncArg(*Object* **node,** *string* **var)**	`true` if the variable is an argument to the function, else `false`.	Tests if a variable found in an expression in any node in the body of a function is declared as an argument to that function.
isCommonDocApi(*string* **methodName)**	`true` if the method is a common `Document` method.	Tests if a method is a common `Document` method (for example, `getElementById()`).
parseDefine(*Object* **node)**	An an object with members of the form: `<function arg> : <module path>`	Parses a KO `define` statement.
getNodeTypes()	An an object with members of the form: `<function arg> : <module path>`	Returns an object mapping abbreviated node types for JavaScript. For details, see the list of node type constants in Listener Types for JavaScript/TypeScript Rules.

SevLib: Severity Support Helper Functions

SevLib is a collection of helper functions that provides support to process severity levels that have been remapped via the Oracle JAF configuration property **sevMap**.

Access these functions through the rule context object property **context.utils.SevLib**. They may be useful when writing audit.

Name	Returns	Description
map(*string* **sev)**	string	Maps a severity. If severity is not mapped, the unmapped severity is returned. For example, `map("major")` will return the mapped value, or **major** if not remapped.
unmap(*string* **sev)**	Array.<*string*>	Unmaps a severity. If the severity does not represent a mapped severity value, the supplied severity is returned. If there more than one severity has been unmapped, it returns all unmapped severity strings in an arrary. For example, given the following: `"sevMap" :{` `  "sevSet" : {` `  "blocker" :  "mustfix",` `  "critical" : "mustfix",` `  "major" :  "mustfix",` `  "minor" :  "warning",` `  "info" :  "warning"` `  }` `}` Then `context.utils.sevLib.unmap('mustfix')` returns `["blocker","critical","major"]`.
isMapped(*string* **sev)**	boolean	Test if a severity is a remapped severity.
getList()	Array.<*string*>	Returns an array of JAF default severities. For example, **blocker**, **critical**, and so on in ascending order of priority.
getMap()	Object	Returns an object whose properties are the JAF default properties, and the corresponding values are the user mapped severities. If severities have not been remapped, `null` is returned.
getInvertedMap()	Object	Returns an inverted form of the object returned by `getMap()`, whose properties are the remapped properties, and the corresponding values are the JAF default properties. If severities have not been remapped, `null` is returned.
isSev(*string* **sev)**	boolean	Returns `true` if the supplied severity is a valid value. May be a core severity or a remapped value.
getMsgSev(*string* **msgId)**	string | null	Returns the severity for the `msgId` (for example, JET-2000), or `null` if the `msgId` is not defined.
matchSeverityLevel(*string* **sev)**	boolean	Tests if the supplied severity is matched by the JAF configuration property **severity** value/expression.
isBlocker(*string* **sev)**	boolean	Returns `true` if the severity (mapped or unmapped) represents **blocker**.

Name	Returns	Description
isCritical(*string* **sev)**	boolean	Returns `true` if the severity (mapped or unmapped) represents **critical**.
isMajor(*string* **sev)**	boolean	Returns `true` if the severity (mapped or unmapped) represents **major**.
isMinor(*string* **sev)**	boolean	Returns `true` if the severity (mapped or unmapped) represents **minor**.
isInfo(*string* **sev)**	boolean	Returns `true` if the severity (mapped or unmapped) represents **info**.

TsxUtils: TSX Utility Functions

TsxUtils is a utility library available through the rule and registration contexts as **context.utils.tsxUtils**.

Method	Returns	Description
extractTsxProperties(*Object* **ruleCtx**, *Object* **TsxRenderComponent**, *Function* **callback**)	nothing	Given a **TsxRenderComponent** object, the *callback* function is called for each tag attribute found in the **TsxRenderComponent** object. `( Object ruleCtx, Object TsxComponent, Object Prop )` where *TsxComponent* is a TsxComponent object, and *Prop* is a property object (a member of TsxComponent)

Note:

1. When the **extractTsxProperties()** method has processed all properties (or iteration is terminated, see 2) below), it makes a final call to the callback method with a null value for argument Prop.
2. For normal processing, the callback function does not need to return any value. If, however, it wishes to terminate the iteration before all properties have been processed, it can return one of two string codes:
   - Return value of **end** that terminates all futher iteration over the TsxRenderComponent.
   - Return value of **next-comp** that terminates all futher iteration over the TsxRenderComponent.

Method	Returns	Description
getExpressionObject(*TsxProperty* **prop**)	Object | null	Returns an object derived from a Preact object expression. For example, given the following snippet: `<xxx myProp={{prop1: 42, prop2: "foo"}}` Then using the TsxProperty object for **myProp** `getExpressionObject(TsxProperty)` returns the following object: `{` `prop1: 42,` `prop2: "foo"` `}` The method returns *null* if the outer-braces of the property value do not contain an object expression.
isPropertyObjectExpr(*TsxProperty* | *tsxComponent* **propOrComp**, [*string* **propName**])	boolean	Returns *true* if the property value of the **TsxProperty**, or the value of the **TsxComponent** with the named property, is an Object expression. For example: `someProp={ { . . . } }` where argument **propOrComp** is a **TsxProperty** or a **TsxComponent** object. If the first argument is a **TsxComponent**, the second argument, **propName**, is required. It is not used for a **TsxProperty**.
isPropertyPreactExpr(*TsxProperty* | *TsxComponent* **propOrComp**, [*string* **propName**])	boolean	Returns true if the property value of the **TsxProperty**, or the value of the **TsxComponent** with the named property, is a Preact expression. For example: `someProp=(. . .)` where argument **propOrComp** is a **TsxProperty** or a **TsxComponent** object. If the first argument is a **TsxComponent**, the second argument, **propName**, is required. It is not used for a **TsxProperty**.
isPropertyString(*TsxProperty* | *TsxComponent* **propOrComp**, [*string* **propName**])	boolean	Returns *true* if the value of **TsxProperty**, or the value of a **TsxComponent** with the named property, is a string value Where argument **propOrComp** is a **TsxProperty** or a **TsxComponent** object. If the first argument is a **TsxComponent**, the second argument, **propName**, is required. It is not used for a **TsxProperty**.

Method	Returns	Description
setIssuePosition(*Object* **issue**, *Object* **compOrProp**)	nothing	Updates the supplied **Issue** instance, with the positional information in the supplied **TsxComponent** or **TsxProperty** object. **issue** is the instantiated ***Issue*** object, and **compOrProp** is a ***TsxComponent*** or ***TsxProperty*** object.
setIssuePropValuePosition(*Object* **issue**, *Object* **tsxProp**)	nothing	Updates the supplied **Issue** instance, with the property value positional information in the supplied **TsxProperty** object. **issue** is the instantiated ***Issue*** object, and **tsxProp** is a ***TsxProperty*** object.
getChildren(*Object* **tsxComp**)	Object|null	Returns an array of child ***TsxComponent*** objects. **tsxComp** is the ***TsxComponent*** whose children are required.
getAncestor(*Object* **tsxComp** [*string* **compName**])	Object|null	Returns the first found named ancestor component of the supplied component. where: 1. **tsxComp** is the *TsxComponent* object whose ancestors are to be searched 2. optional **compName** is the name of the ancestor component to be found. If omitted, the name of the **tsxComp** component is used. Returns the found *TsxComponent*, or *null* if not found.
getDescendant(*Object* **tsxComp** [*string* **compName**])	Object|null	Returns the first found named descendant component of the supplied component. where: 1. **tsxComp** is the *TsxComponent* object whose descendants are to be searched 2. optional **compName** is the name of the descendant component to be found. If omitted, the name of the **tsxComp** component is used. Returns the found *TsxComponent*, or *null* if not found.
getProperty(*TsxProperty* | *TsxComponent* **propOrComp**, *string* **propName**)	Object| null	Returns the **TsxProperty** object for the named property from a **TsxComponent**. If a **TsxProperty** object is supplied, the argument is returned (and the property name is ignored if supplied).
getPropertyValue(*Object* **tsxComp**, *string* **propName**)	Object	Returns the *value* node for the named property from a **TsxComponent**.
getPropertyRawValue(*Object* **tsxComp**, *string* **propName**, *boolean* **stripDelims**)	string	Returns the raw value of the named property from a **TsxComponent**. Note: if the property value is a string, this will include the delimiting quotes unless the optional ***stripDelims*** argument is set to *true*.

Method	Returns	Description
getPropertyStringValue(*TsxProperty* | *TsxComponent* **propOrComp**, [*string* **propName**]	string | null | undefined	Returns the string value of the property if the property type is a string. If the property is empty and has no value (e.g. the HTML 'disabled' attribute) the returned value is undefined. If the value type is not a string or the property is not found, null is returned. Argument **propOrComp** may be a **TsxProperty** or a **TsxComponent** object. If **propOrComp** is a **TsxComponent**, the second argument, **propName**, is required. It is not used for a **TsxProperty**.
getRawText(*Object* **ruleCtx**, *Object* **tsxRC**)	string	Returns the raw text used to create the **TsxRenderComponent**.
getSlotParent(*Object* **ruleCtx**, *TsxProperty* **tsxProp**, *string* **slotName**)	TsxComponent | null	Returns the parent **TsxComponent** that resolves the specified slot name.

The following additional methods (normally exported by **DomUtils** for HTML rules) can also be found in **TsxUtils**, since **DomUtils** is not available in the rule/registration context for TSX rules.

Method	Returns	Description
isAriaAttr(*string* **attrName)**	boolean	Returns `true` if the specified attribute is a known ARIA attribute.
isCommonAttr(*string* **attrName)**	boolean	Returns `true` if the specified attribute is a standard defined xxx. (Note: if **attrName** is prefixed with '`:`', the prefix is ignored).
isCommonElem(*string* **elemName)**	boolean	Returns `true` if the specified element is a standard defined xxx.
isCommonEventAttr(*string* **attrName)**	boolean	Returns `true` if the specified attribute is a standard xxx defined xxx. (Note: if **attrName** is prefixed with '`:`', the prefix is ignored).
isHtml5ObsoleteElem(*string* **elemName)**	boolean	Returns `true` if the specified element is obsolete in HTML5
isHtml5ObsoleteAttr(*string* **elemName,** *string* **attrName)**	boolean	Returns `true` if the attribute is obsolete in HTML5 for the specified element. Note: the return value is unconditional. There may be additional exception conditions that need to be evaluated (for example, the `border` attribute on `<img>`) - refer to xxx.
isNamespacePrefix(*string* **tagPrefix)**	boolean	Given a Web Component name prefix starting with `oj-` (for example `oj-ext`), this returns `true` , if the prefix is defined in the xxx .
isNamespaceTag(*string* **tagName)**	boolean	Given a Web Component element name starting with `oj-` (e.g. `oj-ext-foo`), this returns `true` , if the prefix (`oj-ext` in this example) is defined in the xxx.
isSelfClosingTag(*string* **tagName**)	boolean	Returns true if the tag is a self-closing tag (such as or <hr>, and so on).
isSvgElem(*string* **elemName**)	boolean	Returns *true* if the element name is an SVG element.
isSvgPath(*string* **svg**)	boolean	Returns *true* if the string appears to be a valid SVG path. Note: This method does not validate the path for being syntactically correct SVG.

Use TSX Functions

The following example uses the **extractTsxProperties()** utility method to inspect all tag attributes in the **TsxRenderComponent** structure. It provides a callback mechanism and iterates over the entire TsxRenderComponent.

```
function register() {
    // listen for .tsx renderable content sets
    return { "TsxRenderComponent": _onTsxRC };
};


/**
  * Listener for Web Component tags. Iterate over the tag's properties.
  * @param {Object} ruleCtx    the rule context
  * @param {Object} tsxRC      the TsxRenderComponent tag
  */
function _onTsxRC(ruleCtx, tsxRC) {
    // Extract the attributes and invoke callback _onProperty for each
attribute
    ruleCtx.utils.tsxUtils.extractTsxProperties(ruleCtx, tsxRC, _onProperty);
}

/**
  * Check an attribute and emit an Issue if necessary.
  * @param {Object}  ruleCtx      the rule context
  * @param {Object}  tsxComponent the TsxComponent object
  * @param {*Object} tsxProp      the TsxProperty object
  */
function _onProperty(ruleCtx, tsxComponent, tsxProp) {
    // if tsxProp is null, the iteration is complete, and
    //  this is the final callback.
    if (!tsxProp) { return; }

   // Inspect tsxProp object
   . . .
}
```

As another example, the TsxUtils method **getExpressionObject()** is useful when you need to examine sub-properties declared in a TSX expression. Consider the two sub-properties **one** and **two** of the following property:

```
<xxx someProp={{ one: "left", two: "top" }}
```

The sub-properties can be acquired from the TsxProperty object `someProp` without inspecting the AST nodes by using **getExpressionObject()**:

```
let subprops = ruleCtx.utils.tsxUtils.getExpressionObject(someProp);
```

Where the returned value in subprops will be:

```
{
  one: "left",
  two: "top"
}
```