Orion Container Theming

Authors: caseyflynn@google.com
Last Updated: 2016-12-02

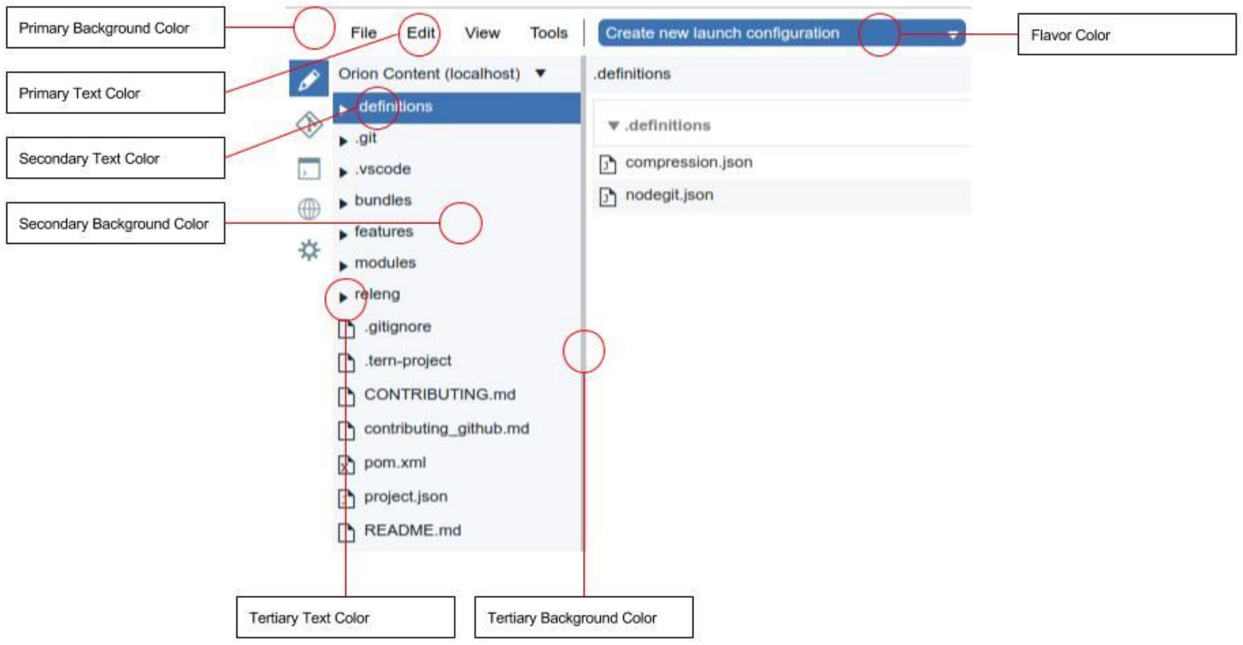
Objective

To add the ability to customize the Orion container theme.

Background

Currently, Orion supports the ability to customize the code editor by allowing users to define the color of text associated with lexemes from various languages. In previous revisions of the product there was a similar mechanism that allowed users to modify various elements of the user interface. This functionality, however, was disabled as a part of a redesign. Since then, the evolution of Orion has rendered the existing code obsolete and thus must be rewritten to regain functionality.

Overview



The color pallette of Orion can be broken down into seven categories:

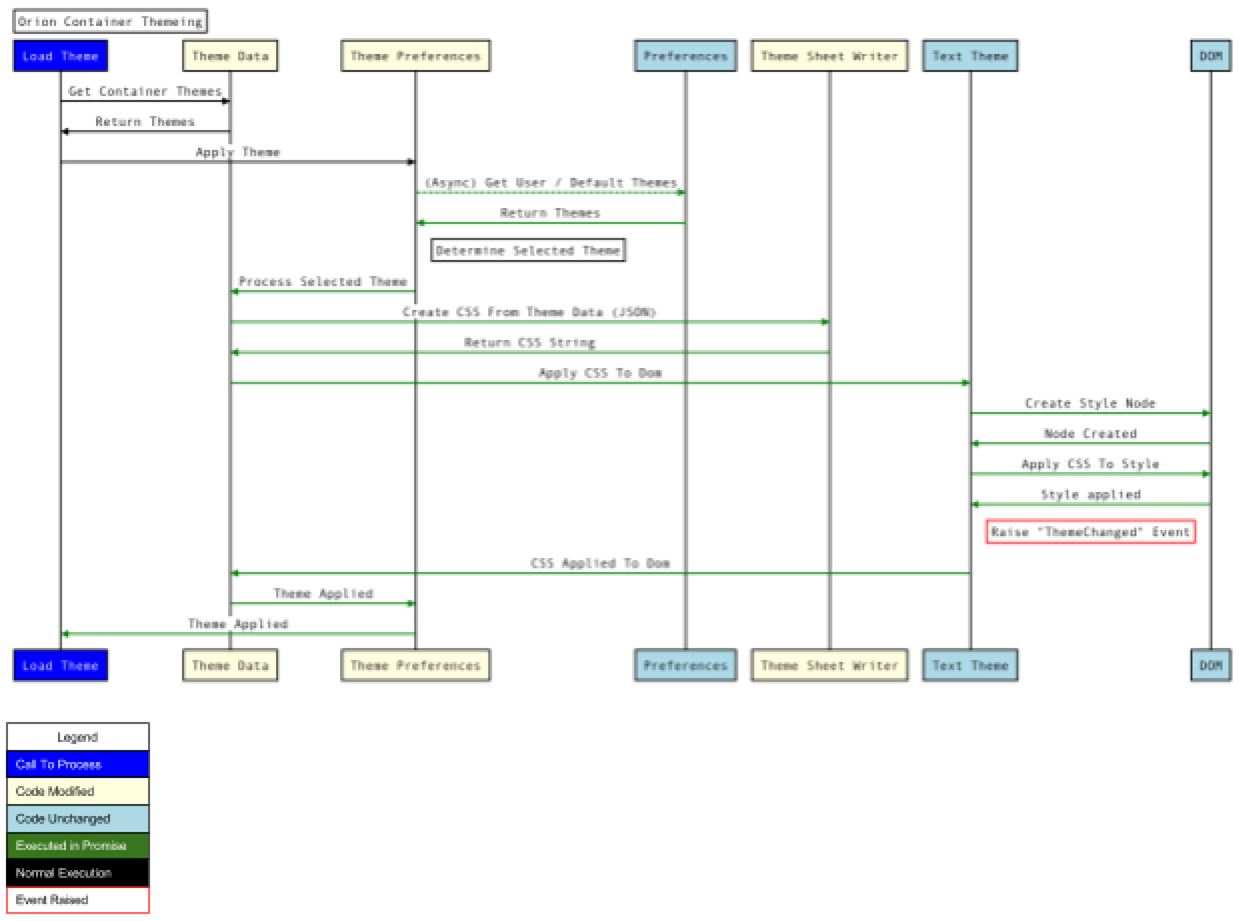
- Primary Text Color
- Primary Background Color
- Secondary Background
- Secondary Text Color
- Tertiary Background Color
- Tertiary Text Color
- Flavor Color

Where each category correlates to one or more items in the DOM.

To enable theming of Orion, a UI will be offered to users that allows the assignment of color values to each of the categories listed above. By default the UI will have two predefined themes (Light Page, Orion) that the user may select as a base to begin creating their own theme. When color values are changed (either by modifying individual color values, or by selecting a predefined, or previously saved theme) the color values will be applied to the current theme. The application will subsequently modify CSS that exists in the DOM to alter the appearance of orion in real time. In addition, the user provided color values will also be stored in user preferences so the theme data will persist across sessions.

Detailed Design

Loading Theme Data Dynamically



The diagram above describes the dynamic loading of container theme data into a style node in the DOM. Currently code to support this feature is not enabled due to style changes within Orion. To enable container theming the following javascript files (light yellow in the diagram) must be modified:

- Theme Data
- Theme Sheet Writer

Theme Data

Theme data will contain the following components:

- An array containing Predefined Theme Data
- Theme Storage Information
- A method that will invoke Theme Sheet Writer to write a given theme to the DOM.

Predefined theme data is housed in JSON objects using following schema:

```
{
  "className" : Base Class Name,
  "name"      : User defined name for the theme,
  "styles"    : JSON object containing CSS style information
}
```

Where the styles object contains one or more JSON objects.

Style Specification Option 1

In this scenario we will define the styles object recursively as:

```
"cssSelector" : {
    ("cssSelector" | "CSS Property Name" : "CSS Property Value")
}
```

Where “cssSelector” is any valid CSS Selector (e.g. .class, #id, element, etc). “CSS Property Name” and “CSS Property Value” are valid CSS property names and values respectively. **This differs from the current editor theming approach.**

For example, if we wanted to define the following CSS style:

```
.someClass { color: black; }
.parent .attribute1.attribute2.attribute3 { color: white; }
#id {  background-color: blue !important; }
.base .selectors:not(.other)>.another:hover { color: red; }
```

We would specify the following Style Object:

```
{
  ".someClass" : {
    "color" : "black"
  },
  ".parent" : {
    ".attribute1.attribute2.attribute3" : {
      "color": "white"
    },
  "#id" : { "background-color" : "blue !important"}
  ".base" : {
    ".selectors:not(.other)>.another:hover" : {
      "color" : "red"
    }
  }
}
```
and use parser option 1 to generate the respective CSS.

Style Specification Option 2

This option (current implementation of the editor theme) uses the following schema for the Styles JSON object :
```
"cssClass" : {
    ("cssClass" | "Property Name camelCase" : "CSS Property Value")
}
```

Where "cssClass" is a valid CSS style class, "Property Name camelCase" is a representation of a CSS property that uses camelCase instead of dash-separated keywords (e.g. backgroundColor instead of background-color), and "CSS Property Value" is any valid CSS Property Value.

This specification only allows for the selection of CSS classes, and the current implementation does not support the specification of descendant selectors.

For example, if we wanted to define the following CSS style:

```
.someClass { color: black; }
.attribute1.attribute2.attribute3  { background-color:  white; }
```

We would specify the following style Object
```
{
  "someClass" : {
    "color" : "black"
  },
  “attribute1” : {
     “attribute2” : {
          “attribute3” : {
               “backgroundColor”: “white”
    }
}
```

Theme Storage Information

Theme storage information provides the preferences service information regarding how to store persistent values along with values themselves. The following JSON schema will be used to specify this information (in accordance with editor theming):

```
{
    “storage:”     : Relative “path” to store the theme information
                     in preferences
    “styleset”     : Key name to retrieve styles stored in
                     preferences
    “defaultTheme” : Name of a theme contained in the styleset object
                     that is used as the default
    “selectedKey”  : Key name of the currently selected editor
    “version”      : Current theme version
}
```

Theme Sheet Writer

The theme sheet writer class will contain a method for converting a Styles JSON object to an array containing CSS strings using a recursive parser.

Parser Option 1

```
var parseStyles = function(object, ancestors, className, result)  {
     if (!ancestors) {
          parseStyles(object, "." + className, className,result);
     }
     var localResult = [];
     var keys = Object.keys(object);
     keys.forEach(function(key) {
          var value = object[key];
          if (typeof(value) === "string") {
               localResult.push("\t" + key + ": " + value + ";");
          } else {
                  parseStyles( value,
                  className === key ? ancestors : ancestors +  " ." +
                  key,
                  className, result);
            }
      });
      if (localResult.length) {
            result.push(ancestors +  " {");
            result.push.apply(result,localResult); result.push( "}");
      }
};
```

Parser Option 2

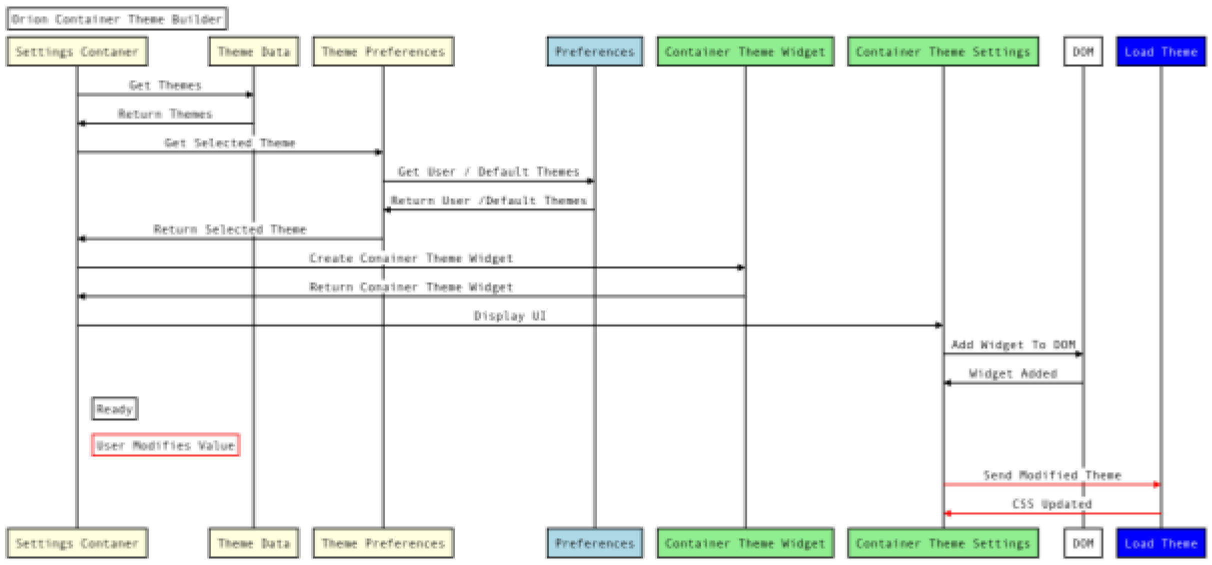
The current implementation utilizes the following parsing algorithm:

```
function(object, ancestors, className, isTopLevel, result)  {
      var localResult = [];
      var keys = Object.keys(object);
      keys.forEach(function(key) {
            var value = object[key];
            if (typeof(value) === "string") {
                  localResult.push("\t" + convertCSSname(key) +  ": " +
            value + ";");
            } else {
                  parseStyles(value,
                  className === key ? ancestors : ancestors +
                  (isTopLevel ? " ." : ".") + key,
                  className,
                  false,
                  result);
            }
      });
      if (localResult.length) {
            result.push(ancestors + (isTopLevel ?  ".textview" : "") +
      " {");
            result.push.apply(result,localResult);
            result.push("}");
      }
}
```

If we select this option, it may be prudent to refactor the parser to place it in a more logical code file.

User Interface For Modifying Theme

In addition to adding support for dynamically loading theme data, we must also add functionality for users to modify color values of Theme Data:



Legend
🟦 Call To Process
🟨 Code Modified
🟦 Code Unchanged
🟩 New Code
⬛ Normal Execution
🟥 User Event

To keep a consistent user interface we will create Container Theme Widget and Container Theme Settings. We will then integrate these components into Settings Container.

Container Theme Widget (Theme Builder)

Container theme widget will house an array of objects, each representing a distinct color of the color palette. The objects will use the following schema:

```
{
  "Display" : string containing data to display on the theme control
  "objPath" : Array of string values indicating *paths to style
              elements in the target Style JSON to modify
  "Id"      : unique identifier of the object
  "Value"   : value to set to the objects in the objPath paths
}
```

On initialization, the widget will create create HTML controls for each object in the schema and bind the controls to a method that will update a selected theme at run time. It will also contain logic for creating a save, revert, and delete buttons and bind them to respective methods.

Update Selected Theme

This method is responsible for validating that a change has occurred by comparing all Style objects specified in objPath against the user provided value, and if required, updating Style object. If a change has occurred, the method will apply the updated style via Theme Sheet Writer.

Saving a Theme

To save a theme, the user will be prompted to enter a theme name. After validation of input (ensure the theme is not a default theme name, does not contain XSS, etc.), the current Styles object will be stored in preferences so that it may be persisted across sessions.

Reverting a Theme

To revert a theme, the container merely needs to replace the current Style object with the last saved version read from user preferences.

Deleting a Theme

After validation of user input, the method will verify the theme proposed for deletion is not a default theme, then it will delete the theme from user preferences. After this is complete, a default theme will be loaded.

Container Theme Settings

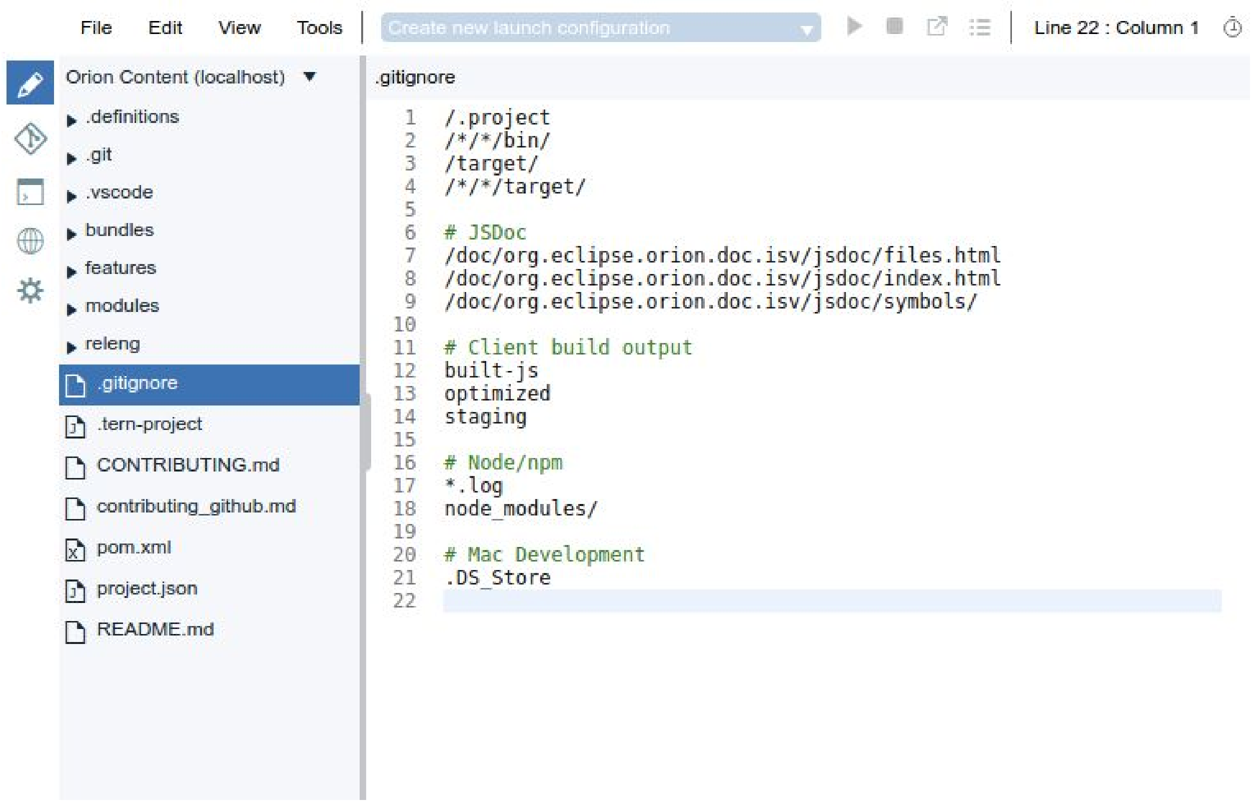
Container theme settings will contain initialization logic to integrate the Container Theme Widget into Settings Container following the pattern established by editor theme settings.

Settings Container

Settings container will be modified to add an additional section to initialize Container Theme Settings which houses the Container Theme widget following similar logic to displaying editor theme settings.

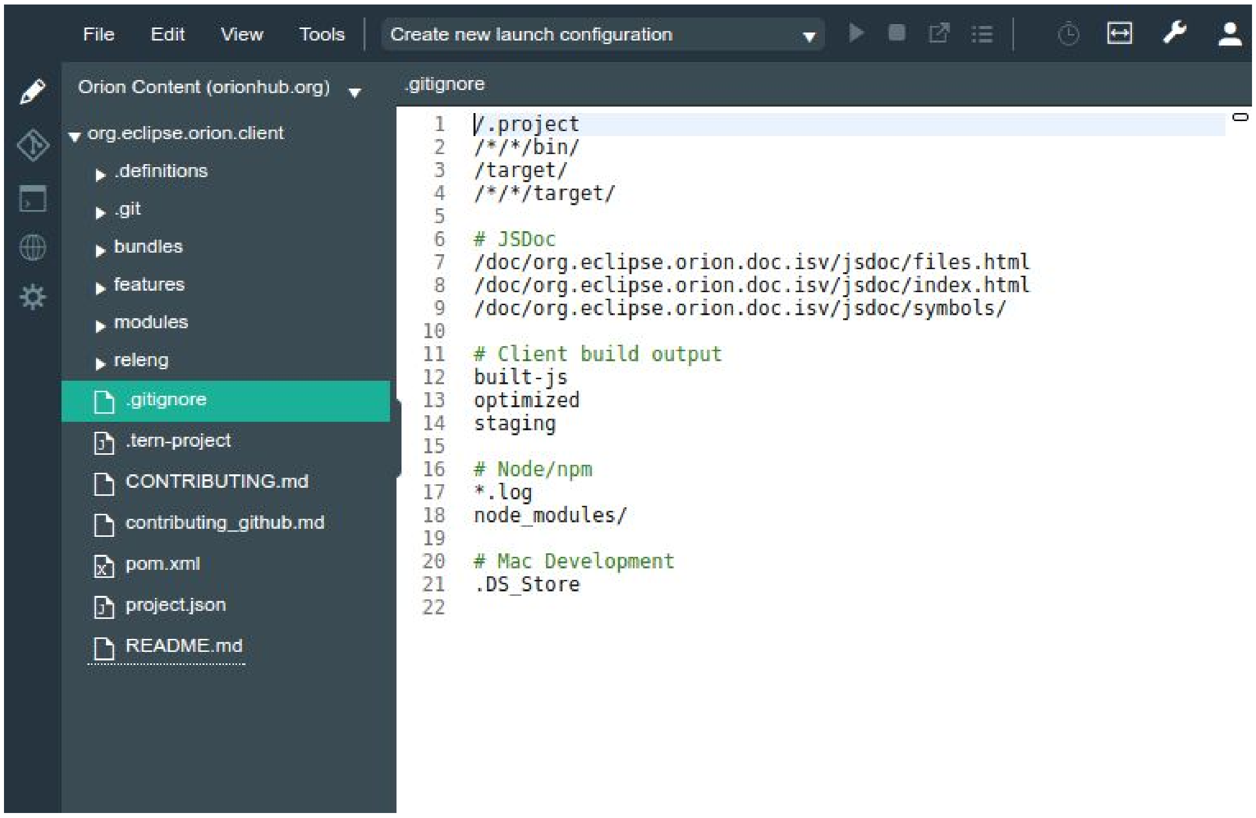
Mocks

Light Page



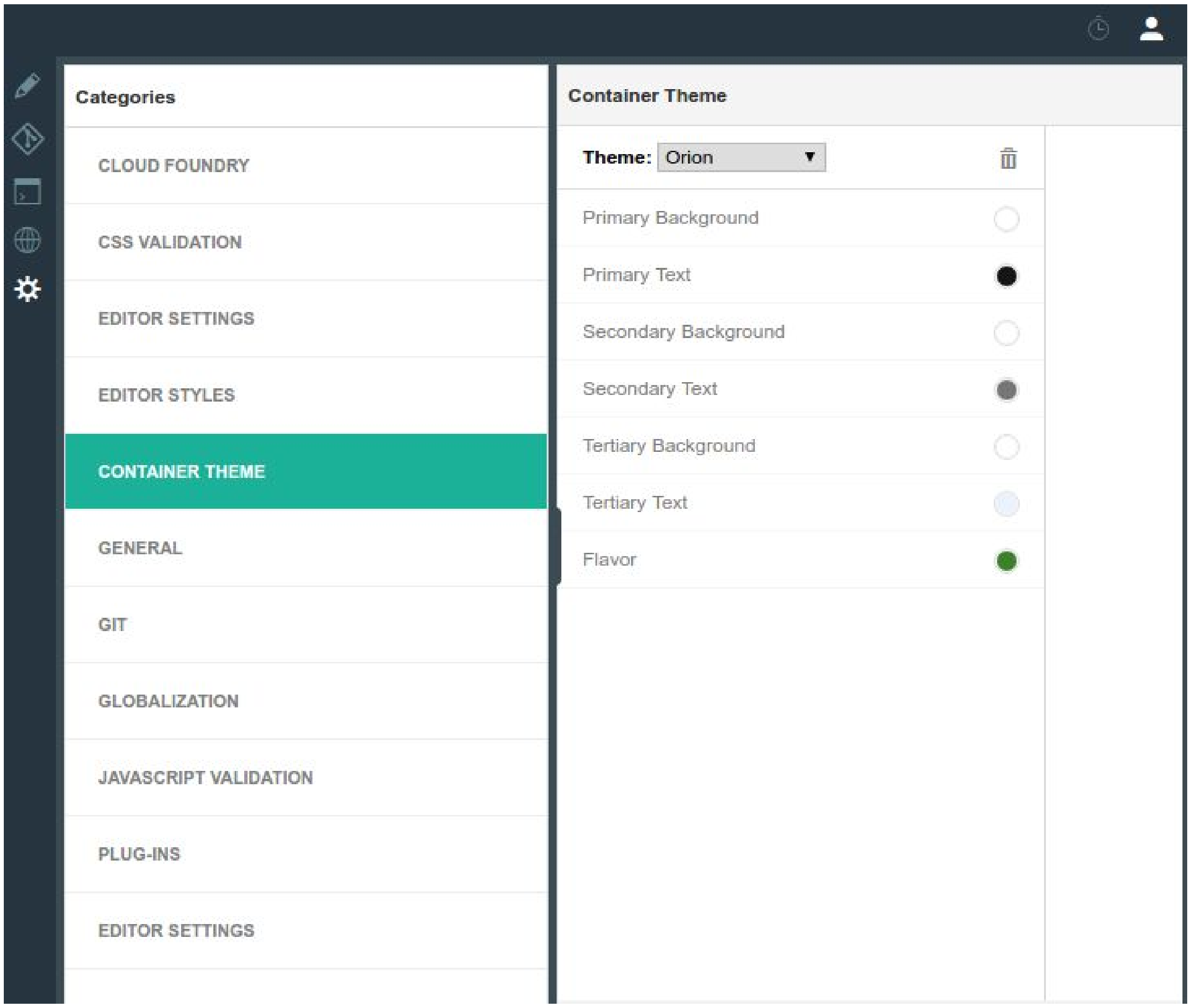
```
/.project
/*/*/bin/
/target/
/*/*/target/

# JSDoc
/doc/org.eclipse.orion.doc.isv/jsdoc/files.html
/doc/org.eclipse.orion.doc.isv/jsdoc/index.html
/doc/org.eclipse.orion.doc.isv/jsdoc/symbols/

# Client build output
built-js
optimized
staging

# Node/npm
*.log
node_modules/

# Mac Development
.DS_Store

```

Orion



```
/.project
/*/*/bin/
/target/
/*/*/target/

# JSDoc
/doc/org.eclipse.orion.doc.isv/jsdoc/files.html
/doc/org.eclipse.orion.doc.isv/jsdoc/index.html
/doc/org.eclipse.orion.doc.isv/jsdoc/symbols/

# Client build output
built-js
optimized
staging

# Node/npm
*.log
node_modules/

# Mac Development
.DS_Store

```

Theme Widget


Categories

- CLOUD FOUNDRY
- CSS VALIDATION
- EDITOR SETTINGS
- EDITOR STYLES
- CONTAINER THEME
- GENERAL
- GIT
- GLOBALIZATION
- JAVASCRIPT VALIDATION
- PLUG-INS
- EDITOR SETTINGS

Container Theme

Theme: Orion

Primary Background	○
Primary Text	●
Secondary Background	○
Secondary Text	●
Tertiary Background	○
Tertiary Text	○
Flavor	●