Oracle® JavaScript Extension Toolkit (Oracle JET)

Developing Oracle JET Apps Using Virtual DOM Architecture

19.0.0
G33835-02
August 2025

ORACLE

Oracle JavaScript Extension Toolkit (Oracle JET) Developing Oracle JET Apps Using Virtual DOM Architecture, 19.0.0

G33835-02

Primary Author: Oracle Corporation

Contents

4 Work with Oracle JET VComponent-based Web Components

5 Use Oracle JET Components and Data Providers

6 Add Third-Party Tools or Libraries to Your Oracle JET App

7 Validate and Convert Input

8 Internationalize and Localize Oracle JET Apps

Preface

Developing Oracle JET Apps Using Virtual DOM Architecture describes how to build responsive web apps and web components using the Oracle JET virtual DOM architecture.

Topics:

Audience

Developing Oracle JET Apps Using Virtual DOM Architecture is intended for intermediate to advanced front-end developers who want to create client-side responsive web apps and web components based on the virtual DOM architecture supported by Oracle JET.

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:

- Oracle JET Web Site
- Oracle JET Function-Based VComponent Tutorial

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 Virtual DOM Architecture in Oracle JET

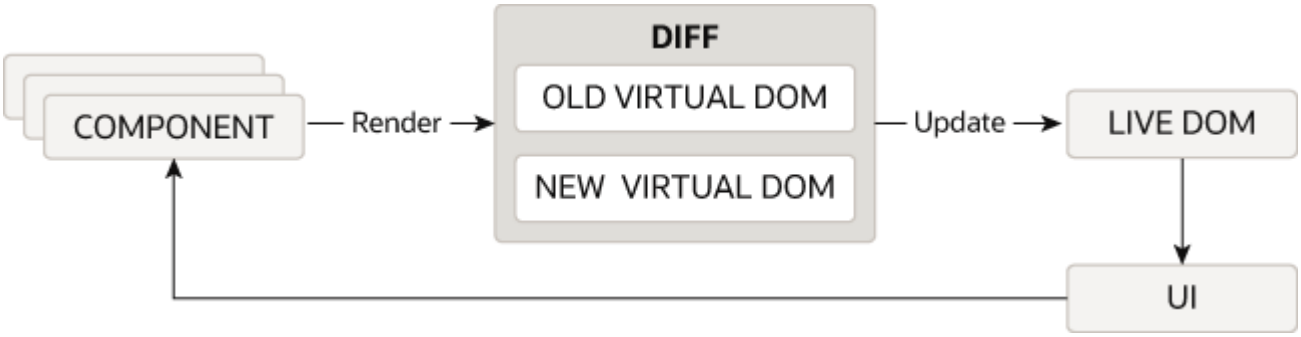
Oracle JET provides the capability to build apps and web components using a virtual DOM architecture that renders components using a virtual DOM engine.

About Oracle JET Virtual DOM Architecture

Virtual DOM architecture is a different way of building apps and web components from the Model-View-ViewModel (MVVM) architecture that Oracle JET has offered to date. Both architectures are supported. You choose the architecture that you want to use.

Virtual DOM architecture is a programming pattern where a virtual representation of the DOM is kept in memory and synchronized with the live DOM by a JavaScript library. As a pattern, it has gained popularity for its ability to efficiently update the browser's DOM (the live DOM). Preact is the JavaScript library that Oracle JET uses to synchronize changes in the virtual DOM to the live DOM.

The following diagram illustrates how virtual DOM architecture apps use the virtual DOM to compute the difference when a state change occurs so that only one action, a re-render of the DOM node(s) affected, occurs in the live DOM.



React, as one of the JavaScript libraries, to popularize the use of the virtual DOM architecture provides extensive documentation that describes many of the main concepts. To learn more about these concepts, consider reading the React documentation. Preact is the JavaScript library that Oracle JET uses to manage the virtual DOM and update the live DOM of apps and Web Components that you develop using Oracle JET's virtual DOM architecture. For more information, see https://preactjs.com/.

What Can You Do with the Oracle JET Virtual DOM Architecture?

You can build web apps and you can also build web components that you publish for inclusion in other Oracle JET web apps.

To accomplish either of these tasks, you must first install the Oracle JET tooling, as described in Install Oracle JET Tooling. Once you have installed the Oracle JET Tooling, you create a virtual DOM app using the following command:

```
ojet create your-First-JET-VDOM-app --template=basic --vdom
```

Once Oracle JET tooling creates the virtual DOM app for you, you can start to develop the functionality of the app. To do this, you write TypeScript and JavaScript XML (JSX). Use of JavaScript is not supported by the virtual DOM architecture.

If you are using the virtual DOM app that you created as a location to develop VComponent-based web components that you later publish to a component exchange, you'll be familiar with the steps if you previously developed Composite Component Architecture-based web components using the Oracle JET tooling. That is, you do the following:

1. Use the Oracle JET tooling to create the component by running the following command:
   `ojet create component oj-hello-world --vcomponent`

   Use of the `--vcomponent` parameter is optional, unless you want to create a class-based VComponent, in which case you specify the `class` option with the `--vcomponent` parameter:

   ```
   ojet create component oj-hello-world --vcomponent=class
   ```

   By default, Oracle JET tooling creates function-based VComponents.

2. If creating multiple components, use JET Pack. The following commands illustrate how you create a JET Pack, *yourJETPack*, and add three components to it (two function components and one class component).

   ```
   ojet create pack yourJETPack
   ojet create component oj-class-component-1 --pack=yourJETPack
   ojet create component oj-function-component --vcomponent=function --
   pack=yourJETPack
   ojet create component oj-class-component-2 --vcomponent=class --
   pack=yourJETPack
   ```

We'll go into more detail for these tasks later in this guide. This brief introduction is for those of you who are already familiar with Oracle JET web app and web component development.

Prerequisite Knowledge

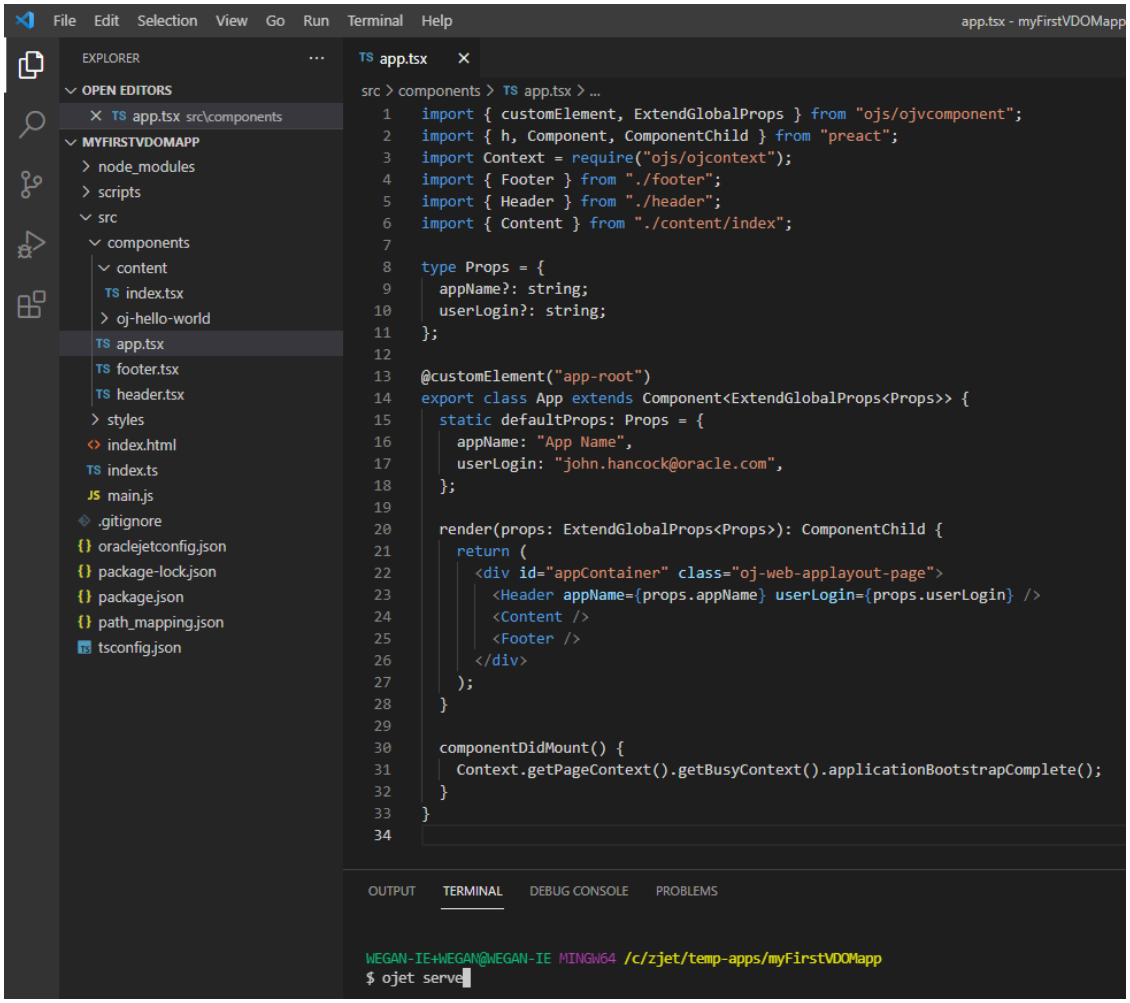
To develop virtual DOM apps and components, you need to understand the following topics:

- Preact: Understand how Preact and, by extension, React works. To get started, read the Main Concepts section of the React documentation. If you are familiar with React concepts, read the Differences to React section of the Preact documentation.
- TypeScript and JSX: You write all virtual DOM apps and web components using TypeScript and JSX. If you previously developed Oracle JET apps using the MVVM architecture, you need to know that the virtual DOM architecture uses JSX to implement binding and conditional behavior that the MVVM architecture implemented in the View (HTML) layer. To learn more about TypeScript, see https://www.typescriptlang.org/ and to learn more about JSX, see JSX In Depth in React's documentation.

Create a Development Environment for Virtual DOM Apps

You can develop virtual DOM apps in any integrated development environment (IDE) that supports TypeScript, HTML, and CSS. However, an IDE is not required, and you can use any text editor to develop your app.

You can use an IDE in conjunction with the Oracle JET Tooling, where you create a virtual DOM app by using the provided app template. You proceed to develop the app in the IDE of your choice by opening the project that was created using the Oracle JET Tooling, in that IDE. After saving changes to your app files in the IDE, you use the Oracle JET Tooling to build and run the app, as demonstrated in the following example from Microsoft Visual Studio Code.



Install Oracle JET Tooling

You must install Node.js to use Oracle JET tooling to develop Oracle JET apps. You'll also need the Oracle JET CLI, `ojet-cli`. You can use the `ojet-cli` through the Node.js package runner (`npx`), or you can install it on your development platform.

If you already have Oracle JET tooling installed on your development platform, check that you are using the minimum versions supported by Oracle JET and upgrade as needed. For the list of minimum supported versions, see Oracle JET Support.

Install Node.js

Install Node.js on your development machine.

From a web browser, download and install one of the installers appropriate for your OS from the Node.js download page. Oracle JET recommends that you install the latest LTS version. Node.js is pre-installed on macOS, but is likely an old version, so upgrade to the latest LTS version if necessary.

After you complete installation and setup, you can enter `npm` commands from a command prompt to verify that your installation succeeded. For example, enter `npm config list` to show config settings for Node.js.

If your computer is connected to a network, such as your company's, that requires you to use a proxy server, run the following commands so that your npm installation can work successfully. This task is only required if your network requires you to use a proxy server. If, for example, you connect to the internet from your home, you may not need to perform this task.

```
npm config set proxy http-proxy-server-URL:proxy-port
npm config set https-proxy https-proxy-server-URL:proxy-port
```

Include the complete URL in the command. For example:

```
npm config set proxy http://my.proxyserver.com:80
npm config set https-proxy http://my.proxyserver.com:80
```

Use the npx NPM CLI Command

We recommend that you use the `npx` NPM CLI command to create and manage Oracle JET apps. With the `npx` NPM CLI command, you won't need to uninstall and reinstall the NPM packages that deliver the ojet-cli if you frequently change releases of the `ojet-cli`.

To use `npx`, you must install Node.js and you must uninstall any globally installed instances of the Oracle JET CLI from your computer. To list globally-installed packages, run the `npm list --depth=0 -g` command in a terminal window. To uninstall a globally installed instance of the Oracle JET CLI, run the `npm -g un @oracle/ojet-cli` command.

Once you have installed Node.js, you can use `npx` and the version of the Oracle JET CLI NPM package that you want to use, plus the appropriate command. The following examples demonstrate how you create Oracle JET apps using different releases of the CLI and then serve them on your local development computer.

```
// Create and serve an Oracle JET 15.0.0 app
$ npx @oracle/ojet-cli@15.0.0 create myJET15app --template=basic --vdom
$ cd myJET15app
$ npx @oracle/ojet-cli@15.0.0 serve

// Create and serve an Oracle JET 19.0.0 app
$ npx @oracle/ojet-cli@19.0.0 create myJETapp --template=basic --vdom
$ cd myJETapp
$ npx @oracle/ojet-cli@19.0.0 serve
```

You can use all the Oracle JET CLI commands (`create`, `build`, `serve`, `strip`, `restore`, and so on) by following the syntax shown in the previous examples (`npx` *`package command`*).

The `npx` package runner fetches the necessary package (for example, `@oracle/ojet-cli@19.0.0`) from the NPM registry and runs it. The package is installed in a temporary cache directory, as in the following example for a Windows computer:

```
C:\Users\JDOE\AppData\Roaming\npm-cache\_npx
```

No NPM packages for the releases of the Oracle JET CLI shown in the previous examples are installed on your computer, as you will see if you run the command to list globally installed NPM packages:

```
$ npm list --depth=0 -g
C:\Users\JDOE\AppData\Roaming\npm
+-- json-server@0.16.3
+-- node-gyp@9.0.0
+-- typescript@4.2.3
`-- yarn@1.22.18
```

Use the following command to clear the temporary cache directory:

```
npx clear-npx-cache
```

To learn more about the `npx` NPM CLI command, see the documentation for the npm CLI. For more information about the commands that the Oracle JET CLI provides, see Understand the Web App Workflow.

Install the Oracle JET Command-Line Interface

Use npm to install the Oracle JET command-line interface (`ojet-cli`).

- At the command prompt of your development machine, enter the following command as Administrator on Windows or use `sudo` on Macintosh and Linux machines:

  ```
  [sudo] npm install -g @oracle/ojet-cli
  ```

  It may not be obvious that the installation succeeded. Enter `ojet help` to verify that the installation succeeded. If you do not see the available Oracle JET commands, scroll through the install command output to locate the source of the failure.

  - 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`.

  You can also verify the Oracle JET version with `ojet --version` to display the current version of the Oracle JET command-line interface. If the current version is not displayed, please reinstall by using the `npm install` command for your platform.

Yarn Package Manager

Oracle JET CLI supports usage of the Yarn package manager.

You must install Node.js as Oracle JET uses the npm package manager by default. However, if you install Yarn, you can use it instead of the default npm package manager by specifying the `--installer=yarn` parameter option when you invoke an Oracle JET command.

The `--installer=yarn` parameter can be used with the following Oracle JET commands:

- `ojet create --installer=yarn`
- `ojet build --installer=yarn`
- `ojet serve --installer=yarn`
- `ojet strip --installer=yarn`

Enter `ojet help` at a terminal prompt to get additional help with the Oracle JET CLI.

As an alternative to specifying the `--installer=yarn` parameter option for each command, add `"installer": "yarn"` to your Oracle JET app's `oraclejetconfig.json` file, as follows:

```
{
. . .
  "generatorVersion": "19.0.0",
  "installer": "yarn"
}
```

The Oracle JET CLI then uses Yarn as the default package manager for the Oracle JET app.

For more information about the Yarn package manager, including how to install it, see Yarn's website.

2

Understand the Web App Workflow

Developing virtual DOM apps with Oracle JET is designed to be simple and efficient using the development environment of your choice and a starter template to ease the development process.

Oracle JET supports creating web apps from a command-line interface:

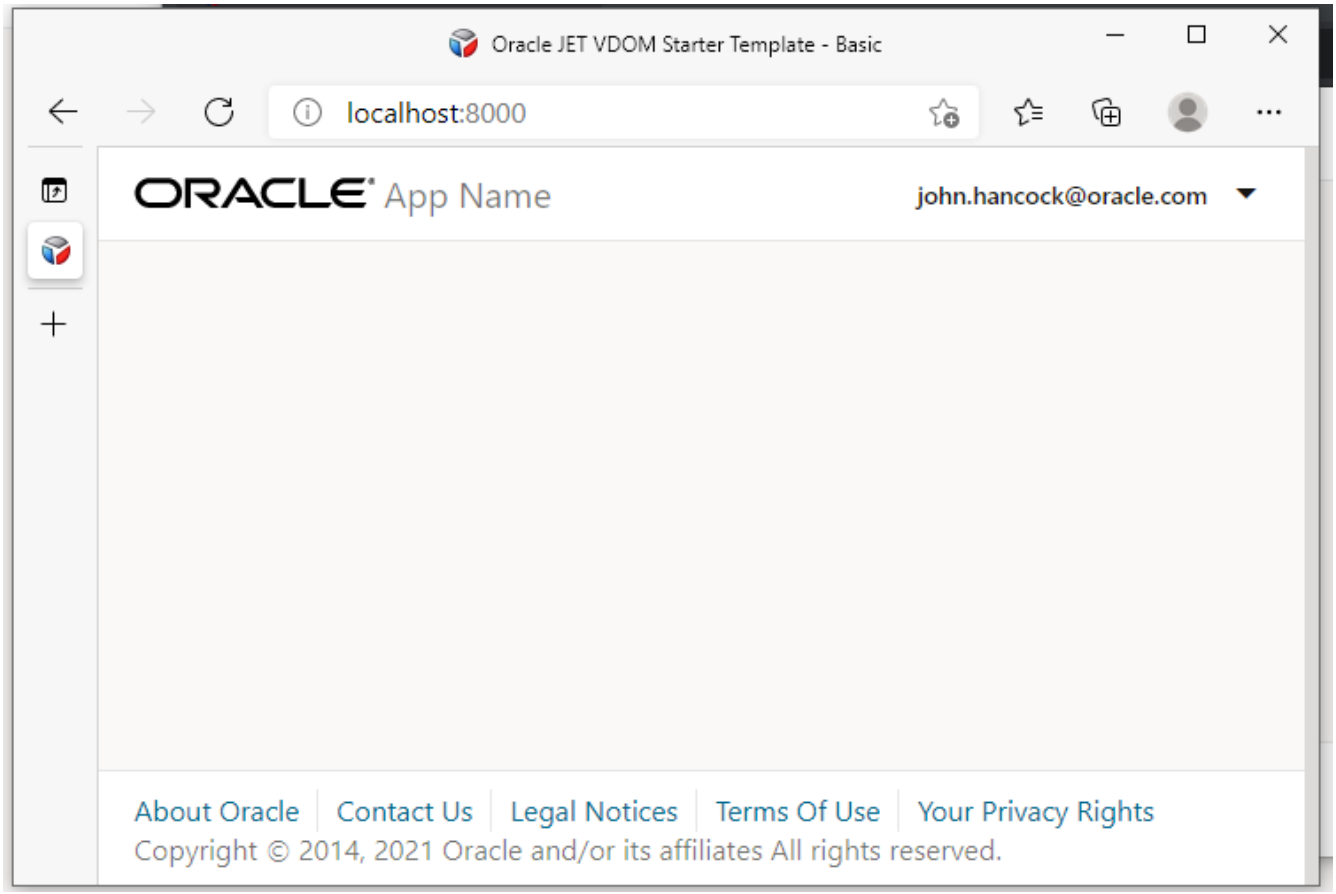
- Before you can create your first Oracle JET web app using the CLI, you must install the prerequisite packages if you haven't already done so. For details, see Install Oracle JET Tooling.
- Then, use the Oracle JET command-line interface package (`ojet-cli`) to scaffold a web app using a basic starter template that creates a pre-configured app that you can modify as needed.
- After you have scaffolded the app, use the `ojet-cli` to build the app, serve it in a local web browser, and create a package ready for deployment.

You must not use more than one version of Oracle JET to add components to the same HTML document of your web app. Oracle JET does not support running multiple versions of Oracle JET components in the same web page.

Scaffold a Web App

Scaffolding is the process you use in the Oracle JET command-line interface (CLI) to create an app that is pre-configured with a content area, a header and a footer layout. After scaffolding, you can modify the app as needed.

The following image shows an app generated by the starter template. It includes responsive styling that adjusts the display when the screen size changes.



Before you can create your first Oracle JET web app using the CLI, you must also install the prerequisite packages if you haven't already done so. For details, see Install Oracle JET Tooling.

To scaffold an Oracle JET web app:

1. At a command prompt, enter `ojet create` with optional arguments to create the Oracle JET app and scaffolding.

```
ojet create [directory]
            [--template={template-name:[basic]|template-url|template-
file}]
            [--vdom]
            [--use-global-tooling]
            [--help]
```

> **Tip**
>
> You can enter `ojet help` at a terminal prompt to get additional help with the Oracle JET CLI.

For example, the following command creates a web app in the `my-First-JET-VDOM-app` directory using the `basic` template:

```
ojet create my-First-JET-VDOM-app --template=basic --vdom
```

To scaffold the web app that will use the globally-installed `@oracle/oraclejet-tooling` rather than install it locally in the app directory, enter the following command:

```
ojet create my-web-app --use-global-tooling --vdom
```

2. Wait for confirmation.

   The scaffolding will take some time to complete. When successful, the console displays:

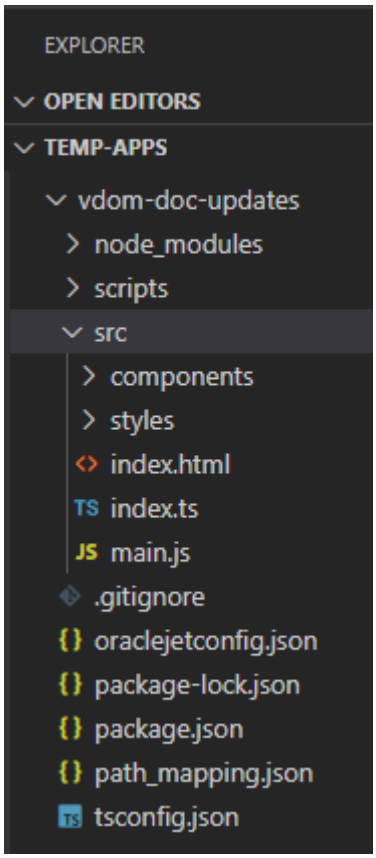
   ```
   Oracle JET: Your app is ready! Change to your new app directory my-First-
   JET-VDOM-app and try ojet build and serve...
   ```

3. In your development environment, update the code for your app.

About the Virtual DOM App Layout

Oracle JET Tooling creates an app for you with the necessary source files and directory structure.

The new application will have a directory structure similar to the one shown in the following image.



The application folders contain the application and configuration files that you will modify as needed for your own application.

Directory or File	Description
`node_modules`	Contains the Node.js modules used by the tooling.
`scripts`	Contains template hook scripts that you can modify to define new build and serve steps for your application. See Customize the Web App Tooling Workflow
`src`	Site root for your application. Contains the application files that you can modify as needed for your own application and should be committed to source control. The starter template includes an `app` component that imports `footer`, `header`, and `content` components to help you get started.
`.gitignore`	Defines rules for application folders to ignore when using a GIT repository to check in application source. Users who do not use a GIT repository can use `ojet strip` to avoid checking in content that Oracle JET always regenerates. Note this file must not be deleted since the `ojet strip` command depends on it.
`oraclejetconfig.json`	Contains the default source and staging file paths that you can modify if you need to change your application's file structure.
`package.json`	Defines npm dependencies and project metadata.

Whether you are developing a virtual DOM app or a VComponent-based web component, you'll write your code in the files under the `src` directory. The Oracle JET Tooling creates a corresponding `web` and `dist` directory when you build and/or serve the app, or publish a JET Pack with web components.

```
||| Contents of virtual DOM app directory prior to build or serve
|   oraclejetconfig.json
|   package-lock.json
|   package.json
|   path_mapping.json
|   tsconfig.json
+---scripts
|   +---config
|   \---hooks
\---src
    |   index.html
    |   index.ts
    |   main.js
    +---components
    |   |   app.tsx
    |   |   footer.tsx
    |   |   header.tsx
    |   \---content
    |           index.tsx
    \---styles
        |   app.css
        +---fonts
        \---images
                ...
                oracle_logo.svg
```

Oracle JET virtual DOM architecture uses modules to organize VComponents and the Oracle JET Tooling manages the modules in a project.

Custom element-based VComponents must reside in a directory with a name that matches the name of the component. That is, the `oj-hello-world` custom component must reside in a directory named `oj-hello-world`. By default, the name of the directory where you store all components in your virtual DOM app is `components`, but you can change it to a different value by editing the value of `components` in the `oraclejetconfig.json` file in your virtual DOM app's root directory.

```
{
  "paths": {
    "source": {
      ...,
      "components": "changeToYourPreferredValue",
      ...
    },
    ...
```

Each custom element-based VComponent requires an accompanying `loader.ts` module file, as illustrated by the following example for a custom element-based VComponent named `oj-hello-world`. Oracle JET Tooling includes this file as Oracle Visual Builder, the Oracle Component Exchange, and the Oracle JET Tooling itself require it.

```
...
+---src
|   index.html
...
+---components
|   |   app.tsx
|   |   footer.tsx
|   |   header.tsx
|   |
|   +---content
|   |       index.tsx
|   |
|   \---oj-hello-world
|       |   loader.ts
|       |   oj-hello-world-styles.css
|       |   oj-hello-world.tsx
|       |   README.md
|       |
|       +---resources
|       |   \---nls
|       |           oj-hello-world-strings.ts
|       |
...
```

The Oracle JET Tooling can manage components individually but we recommend that you manage components in a pack (also known as a JET Pack) if you intend to create more than one component.

About the Binding Provider for Virtual DOM Apps

Oracle JET's default binding provider is Knockout. The binding provider that you use affects how Oracle JET custom elements render.

If the default binding provider (`data-oj-binding-provider="knockout"`) is used, all Oracle JET custom elements, such as `oj-list-view`, wait on the apply bindings traversal to complete before rendering. When Oracle JET custom elements are used in a Preact environment (virtual DOM app), you must use the preact binding provider (`data-oj-binding-provider="preact"`).

VComponent-based web components implicitly set the binding provider to `preact` on behalf of their child components. As a result, Oracle JET custom elements inside of a VComponent use the `preact` binding provider. However, for virtual DOM apps, you, the app developer, must configure the binding provider manually (`data-oj-binding-provider="preact"`) somewhere in the DOM above where you add Oracle JET custom elements. The following code snippet shows three examples of how you can set the binding provider to `preact`.

```
// Update the entry in the appRootDir/src/index.html file of your virtual DOM
app
<body class="oj-web-applayout-body" data-oj-binding-provider="preact">

// Set the data-oj-binding-provider="preact" on a parent element to a
collection
// of JET custom elements
<body>
    <div id="preact-content-goes-here" data-oj-binding-provider="preact">
     <oj-c-button . . .>
         <oj-list-view . . .>
    </div>

// Set the data-oj-binding-provider="preact" on an individual JET custom
element
<oj-list-view data-oj-binding-provider="preact"...>
```

> **Note**
>
> The `index.html` file that the Oracle JET Tooling generates when you create a virtual DOM app is a regular HTML document with no active binding provider. For this reason, its use of `<body ... data-oj-binding-provider="none">` is appropriate. Note too that the `<app-root>` element that the Oracle JET Tooling also generates in the `index.html` file is a VComponent-based web component and it, like other VComponent-based web components, sets the binding provider to `preact` on behalf of any child components that it references.

For more information about the binding provider, see the Binding Providers section in the *Oracle® JavaScript Extension Toolkit (JET) API Reference for Oracle JET*.

Add Progressive Web App Support to Web Apps

Add Progressive Web App (PWA) support to your JET web app if you want to give users a native-like mobile app experience on the device where they access your JET web app.

Using the `ojet add pwa` command, you add both a service worker script and a web manifest to your JET web app. You can customize these artifacts to determine how your JET web app behaves when accessed as a PWA.

Using the `ojet add pwa` command, you add both a service worker script, an `assets` folder with a series of image files for app launcher icons and splash screens, plus a web manifest file to your JET web app. You can customize these artifacts to determine how your JET web app behaves when accessed as a PWA.

To add PWA support to your web app:

- At a terminal prompt, in your app's top level directory, enter the following command to add PWA support to your JET web app:

  ```
  ojet add pwa
  ```

When you run the command, Oracle JET tooling makes the following changes to your JET web app:

- Adds the following two files to the app's `src` folder:
  - `assets` folder

    The `assets` folder contains an additional two sub-folders, `icons` and `splashscreens`, that contain image files of various dimensions to use as app launcher icons and splash screens on the devices where you install the JET web app.
  - `manifest.json`
    This file tells the browser about the PWA support in your JET web app, and how it should behave when installed on a user's desktop or mobile device. Use this file to specify the app name to appear on a user's device, plus device-specific icons. For information about the properties that you can specify in this file, see Add a web app manifest.
  - `swinit.js`

    This is the initialization file for the service worker script (`sw.js`).
  - `sw.js`

    This is the service worker script that the browser runs in the background. Use this file to specify any additional resources from your JET app that you want to cache on a user's device if the PWA service worker is installed. By default, JET specifies the following resources to cache:

    ```
    const resourcesToCache = [
          'index.js',
          'index.html',
          'bundle.js',
          'manifest.json',
          'components/',
          'libs/',
          'styles/',
          'assets/'
        ];
    ```

- Registers the splash screen files, the manifest file, and the service worker script in the JET web app's `./src/index.html` file:

```
<html lang="en-us">
  <head>
  . . .
   <meta name="apple-mobile-web-app-title" content="Oracle JET" />
    <meta name="theme-color" content="#000000">
    <!-- Splash screens -->
    <link rel="apple-touch-startup-image" href="assets/splashscreens/
splash-640x1136.jpg" . . .">
  . . .
  <link rel="manifest" href="manifest.json">
</head>
 . . .
<script src="swinit.js"></script>
</body>
```

With these changes, a user on a mobile device, such as an Android phone, can initially access your JET web app through its URL using the Chrome browser, add it to the Home screen of the device, and subsequently launch it like any other app on the phone. Note that browser and platform support for PWA is not uniform. To ensure an optimal experience, test your PWA-enabled JET web app on your users' target platforms (Android, iOS, and so on) and the browsers (Chrome, Safari, and so on).

PWA-enabled JET web apps and service workers require HTTPS. The production environment where you deploy your PWA-enabled JET web app will serve the app over HTTPS. If, during development, you want to serve your JET web app to a HTTPS-enabled server, see Serve a Web App to a HTTPS Server Using a Self-signed Certificate.

Build a Web App

Use the Oracle JET command-line interface (CLI) to build a development version of your web app before serving it to a browser. This step is optional.

Change to the app's root directory and use the `ojet build` command to build your app.

```
ojet build [--cssvars=enabled|disabled
            --theme=themename
            --themes=theme1,theme2,...]
```

> **Tip**
>
> You can enter `ojet help` at a terminal prompt to get help for specific Oracle JET CLI options.

The command will take some time to complete. If it's successful, you'll see the following message:

```
Done.
```

The command will also create a web folder in your app's root to contain the built content.

> **Note**
>
> You can also use the `ojet build` command with the `--release` option to build a release-ready version of your app. For information, see Package and Deploy Apps.

Serve a Web App

Use `ojet serve` to run your web app in a local web server for testing and debugging. By default, the Oracle JET live reload option is enabled which lets you make changes to your app code that are immediately reflected in the browser.

To run your web app from a terminal prompt:

1. At a terminal prompt, change to the app's root directory and use the `ojet serve` command with optional arguments to launch the app.

   ```
   ojet serve [--server-port=server-port-number --livereload-port=live-reload-
   port-number
               --livereload
               --watch-files
               --build
               --theme=themename --themes=theme1,theme2,...
               --server-only
               --server-url=server-url
              ]
   ```

   For example, the following command launches your app in the default web browser with live reload enabled.

   ```
   ojet serve
   ```

2. Make any desired code change in the `src` folder, and the browser will update automatically to reflect the change unless you set the `--no-livereload` flag.

   While the app is running, the terminal window remains open, and the watch task waits for any changes to the app. For example, if you change the content in `src/components/oj-hello-world/oj-hello-world-styles.css` , the watch task will reflect the change in the terminal as shown below on a Windows desktop.

   ```
   . . .
   Watcher: themes is ready.
   Changed: C:\jetVDOMapp\src\components\content\index.tsx
   Running before_watch hook.
   Compile application typescript
   Running before_app_typescript hook.
   Compile application typescript finished
   Running after_watch hook.
   Page reloaded resume watching.
   ```

3. To terminate the process, close the app and press Ctrl+C at the terminal prompt. You may need to enter Ctrl+C a few times before the process terminates.

To get additional help for the supported `ojet serve` options, enter `ojet serve --help` at a terminal prompt.

About ojet serve Command Options and Express Middleware Functions

Use `ojet serve` to run your web app in a local web server for testing and debugging.

The following table describes the available `ojet serve` options and provides examples for their use.

Oracle JET tooling uses Express, a Node.js web app framework, to set up and host the web app when you run `ojet serve`. If the ready-to-use `ojet serve` options do not meet your requirements, you can add Express configuration options or write Express middleware functions in Oracle JET's `before_serve.js` hook point. For an example that demonstrates how to add Express configuration options, see Serve a Web App to a HTTPS Server Using a Self-signed Certificate.

The `before_serve` hook point provides options to determine whether to replace the existing middleware or instead prepend and append a middleware function to the existing middleware. Typically, you'll prepend a middleware function (`preMiddleware`) that you write if you want live reload to continue to work after you serve your web app. Live reload is the first middleware that Oracle JET tooling uses. You must use the `next` function as an argument to any middleware function that you write if you want subsequent middleware functions, such as live reload, to be invoked by the Express instance. In summary, use one of the following arguments to determine when your Express middleware function executes:

- `preMiddleware`: Execute before the default Oracle JET tooling middleware. The default Oracle JET tooling middleware consists of `connect-livereload`, `serve-static`, and `serve-index`, and executes in that order.
- `postMiddleware`: Execute after the default Oracle JET tooling middleware.
- `middleware`: Replaces the default Oracle JET tooling middleware. Use if you need strict control of the order in which middleware runs. Note that you will need to redefine all the default middleware that was previously added by Oracle JET tooling.

Option	Description
`server-port`	Server port number. If not specified, defaults to 8000.
`livereload-port`	Live reload port number. If not specified, defaults to 35729.
`watch-files`	Enable the watch files feature. Watch files is enabled by default (`--watch-files=true`). Use `--watch-files=false` or `--no-watch-files` to disable the watch files feature. Disabling watch files also disables the live reload feature. Configure the interval at which the watch files feature polls the Oracle JET project for updates by configuring a value for the `watchInterval` property in the `oraclejetconfig.json` file. The default value is `1000` milliseconds.
`livereload`	Enable the live reload feature. Live reload is enabled by default (`--livereload=true`). Use `--livereload=false` or `--no-livereload` to disable the live reload feature. Disabling live reload can be helpful if you're working in an IDE and want to use that IDE's mechanism for loading updated apps. The interval at which the live reload feature polls the Oracle JET project depends on the `watch-files` option.
`build`	Build the app before you serve it. By default, an app is built before you serve it (`--build=true`). Use `--build=false` or `--no-build` to suppress the build if you've already built the app and just want to serve it.
`theme`	Theme to use for the app. The theme defaults to `redwood`.

Option	Description
`themes`	Themes to use for the app, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme. Otherwise Oracle JET will serve the app with the theme specified in `--theme`.
`server-only`	Serves the app, as if to a browser, but does not launch a browser. Use this option in cloud-based development environments so that you can attach your browser to the app served by the development machine.
`server-url`	Specify the server URL to serve from. For example, `ojet serve --server-url=https://www.example.com/jet`. If not specified, defaults to `localhost`.

Serve a Web App to a HTTPS Server Using a Self-signed Certificate

You can customize the JET CLI tooling to serve your web app to a HTTPS server instead of the default HTTP server that the Oracle JET `ojet serve` command uses.

Do this if, for example, you want to approximate your development environment more closely to a production environment where your web app will eventually be deployed. Requests to your web app when it is deployed to a production environment will be served from an SSL-enabled HTTP server (HTTPS).

To implement this behavior, you'll need to install a certificate in your web app directory. You'll also need to configure the `before_serve.js` hook to do the following:

- Create an instance of Express to host the served web app.
- Set up HTTPS on the Express instance that you've created. You specify the HTTPS protocol, identify the location of the self-signed certificate that you placed in the app directory, and specify a password.
- Pass the modified Express instance and the SSL-enabled server to the JET tooling so that `ojet serve` uses your middleware configuration rather than the ready-to-use middleware configuration provided by the Oracle JET tooling.
- To ensure that live reloads works when your web app is served to the HTTPS server, you'll also create an instance of the live reload server and configure it to use SSL.

If you can't use a certificate issued by a certificate authority, you can create your own certificate (a self-signed certificate). Tools such as OpenSSL, Keychain Access on Mac, and the Java Development Kit's `keytool` utility can be used to perform this task for you. For example, using the Git Bash shell that comes with Git for Windows, you can run the following command to create a self-signed certificate with the OpenSSL tool:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes
```

Once you've obtained the self-signed certificate that you want to use, install it in your app's directory. For example, place the two files generated by the previous command in your app's root directory:

```
...
.gitignore
cert.pem
key.pem
node_modules
...
```

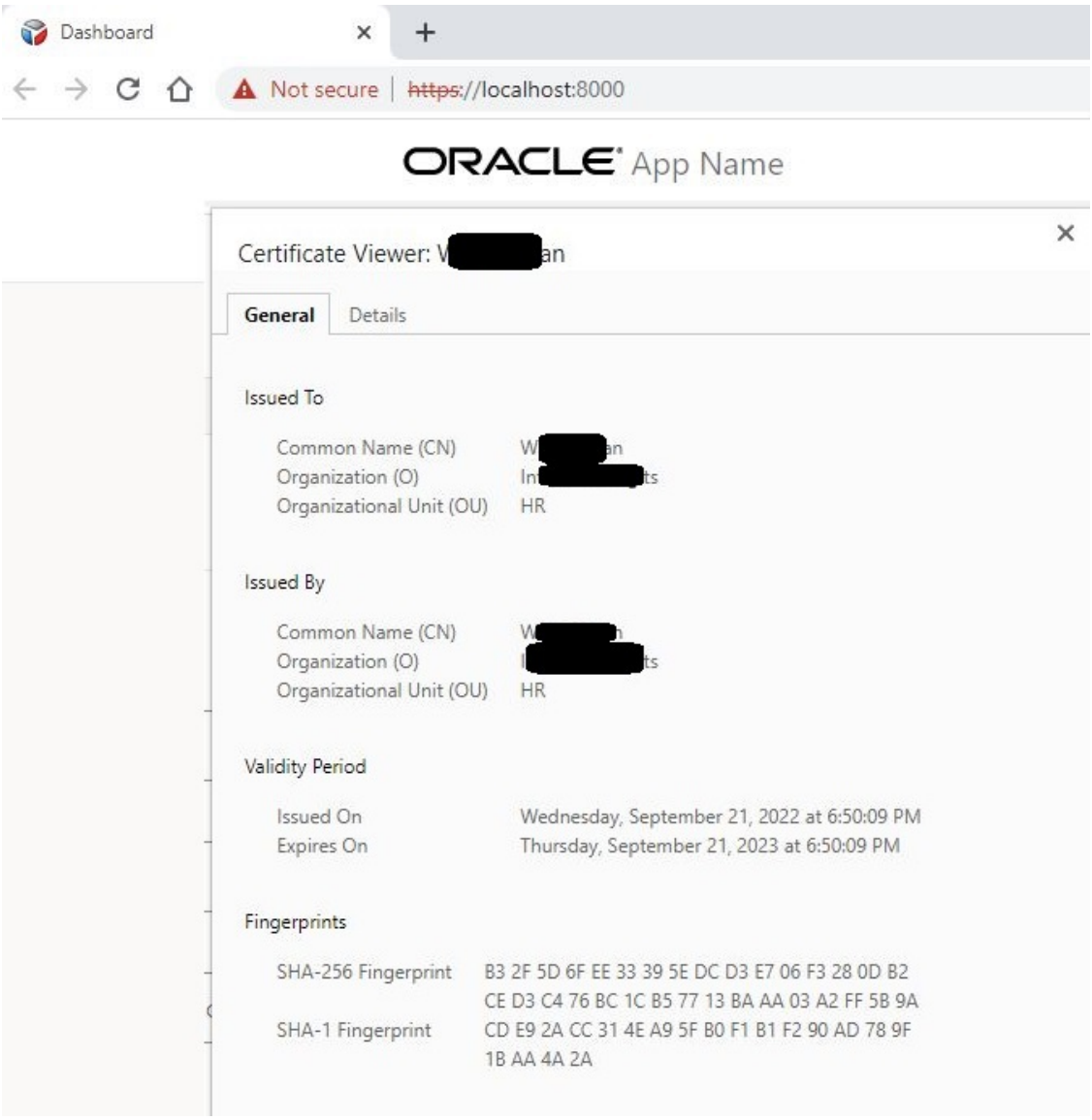
Once you have installed the self-signed certificates in your app, you configure the script for the `before_serve` hook point. To do this, open the `AppRootDir/scripts/hooks/before_serve.js` with your editor and configure it as described by the comments in the following example configuration.

```
'use strict';

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running before_serve hook.");

    // Create an instance of Express, the Node.js web app framework that
Oracle
    // JET tooling uses to host the web apps that you serve using ojet serve
    const express = require("express");

    // Set up HTTPS
    const fs = require("fs");
    const https = require("https");

    // Specify the self-signed certificates. In our example, these files
exist
    // in the root directory of our project.
    const key = fs.readFileSync("./key.pem");
    const cert = fs.readFileSync("./cert.pem");
    // If the self-signed certificate that you created or use requires a
    // password, specify it here:
    const passphrase = "1234";

    const app = express();

    // Pass the modified Express instance and the SSL-enabled server to the
Oracle JET tooling
    configObj['express'] = app;
    configObj['urlPrefix'] = 'https';
    configObj['server'] = https.createServer({
      key: key,
      cert: cert,
      passphrase: passphrase
    }, app);

    // Enable the Oracle JET live reload option using its default port number
so that
    // any changes you make to app code are immediately reflected in the
browser after you
    // serve it
    const tinylr = require("tiny-lr");
    const lrPort = "35729";

    // Configure the live reload server to also use SSL
    configObj["liveReloadServer"] = tinylr({ lrPort, key, cert, passphrase });

    resolve(configObj);
```

```
  });
};
```

Once you have completed these configuration steps, run the series of commands (`ojet build` and `ojet serve`, for example) that you typically run to build and serve your web app. As the certificate that you are using is a self-signed certificate rather than a certificate issued by a certificate authority, the browser that you use to access the web app displays a warning the first time that you access the web app. Acknowledge the warning and click the options that allow you to access your web app. On Google Chrome, for example, you click **Advanced** and **Proceed to localhost (unsafe)** if your web app is being served to `https://localhost:8000/`.

Once your web app opens in the browser, you'll see that the HTTPS protocol is used and an indicator that the connection is not secure, because you are not using a certificate from a certificate authority. You can also view the certificate information to confirm that it is the self-signed certificate that you created. In Google Chrome, click **Not secure** and **Certificate** to view the certificate information.



The `before_serve` hook point is one of a series of script hook points that you can use to customize the tooling workflow for Oracle JET apps. See Customize the Web App Tooling Workflow.

Customize the Web App Tooling Workflow

Hook points that Oracle JET tooling defines let you customize the behavior of the JET build and serve processes when you want to define new steps to execute during the tooling workflow using script code that you write.

When you create an app, Oracle JET tooling generates script templates in the `/scripts/hooks` app subfolder. To customize the Oracle JET tooling workflow, you can edit the generated templates with the script code that you want the tooling to execute for specific hook points during the build and serve processes. If you do not create a custom hook point script, Oracle JET tooling ignores the script templates and follows the default workflow for the build and serve processes.

To customize the workflow for the build or serve processes, you edit the generated script template file named for a specific hook point. For example, to trigger a script at the start of the tooling's build process, you would edit the `before_build.js` script named for the hook point triggered before the build begins. That hook point is named `before_build`.

Therefore, customization of the build and serve processes that you enforce on Oracle JET tooling workflow requires that you know the following details before you can write a customization script.

- The Oracle JET build or serve mode that you want to customize:
  - Debug — The default mode when you build or serve your app, which produces the source code in the built app.
  - Release — The mode when you build the app with the `--release` option, which produces minified and bundled code in a release-ready app.
- The appropriate hook point to trigger the customization.
- The location of the default hook script template to customize.

About the Script Hook Points for Web Apps

The Oracle JET hooks system, also called hook points, allow you to customize various worksflows across command-line interface processes.

Customization relies on script files and the script code that you want to trigger for a particular hook point.

The following table identifies the hook points and the workflow customizations they support in the Oracle JET tooling create, build, serve, and restore processes. Unless noted, hook points for the build and serve processes support both debug and release mode.

Hook Point	Supported Tooling Process	Description
`after_app_create`	create	This hook point triggers the script with the default name `after_app_create.js` immediately after the tooling concludes the create app process.
`after_app_restore`	restore	This hook point triggers the script with the default name `after_app_restore.js` immediately after the tooling concludes the restore app process.

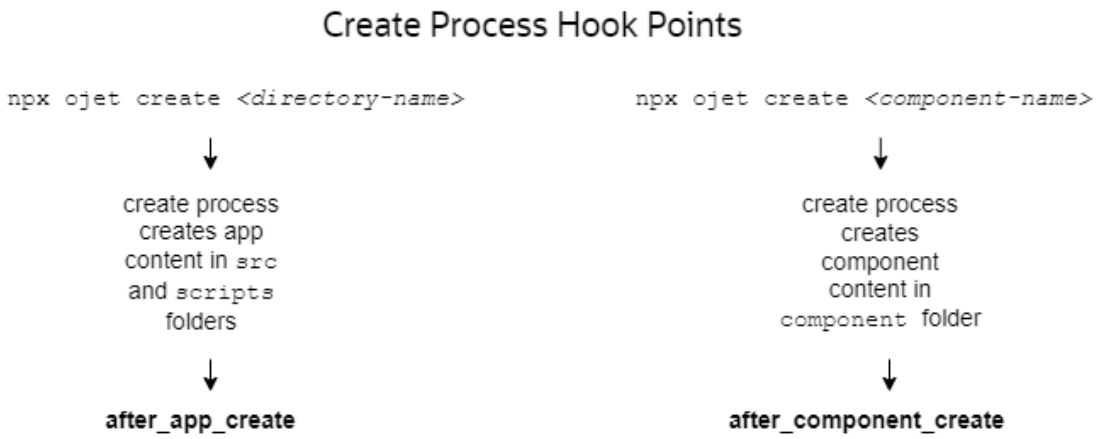
Hook Point	Supported Tooling Process	Description
`before_build`	build	This hook point triggers the script with the default name `before_build.js` immediately before the tooling initiates the build process.
`before_release_build`	build (release mode only)	This hook point triggers the script with the default name `before_release_build.js` before the minification step and the requirejs bundling step occur.
`before_app_typescript`	build / serve	This hook point triggers the script with the default name `before_app_typescript.js` after the build process or serve process steps occur and before the app is transpiled. Use the hook to update, add or remove TypeScript compiler options defined by your app's `tsconfig.json` compiler configuration file. The hook system passes your reference to the modified `tsconfig` object to the TypeScript compiler. A script for this hook point can only be used with a TypeScript app.
`after_app_typescript`	build / serve	This hook point triggers the script with the default name `after_app_typescript.js` after the build process or serve process steps occur and immediately after the `before_app_typescript` hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated `.jsx` output using babel. A script for this hook point can only be used with a TypeScript app.
`before_component_typescript`	build / serve	This hook point triggers the script with the default name `before_component_typescript.js` after the build process or serve process steps occur and before the component is transpiled. Use the hook to update, add or remove TypeScript compiler options defined by your app's `tsconfig.json` compiler configuration file. The hook system passes your reference to the modified `tsconfig` object to the TypeScript compiler. A script for this hook point can only be used with a TypeScript app.
`after_component_typescript`	build / serve	This hook point triggers the script with the default name `after_component_typescript.js` after the build process or serve process steps occur and immediately after the `before_component_typescript` hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated `.jsx` output using babel. A script for this hook point can only be used with a TypeScript app.
`before_injection`	build / serve	This hook point triggers the script with the default name `before_injection.js` after the tooling concludes the before build process and before Oracle JET injects the correct path mappings into your application and performs the tasks to insert the CSS theme into the app. In other words, this hook point controls which files and which markers are patched by Oracle JET itself.
`before_optimize`	build / serve (release mode only)	This hook point triggers the script with the default name `before_optimize.js` before the release mode build/serve process minifies the content.
`before_component_optimize`	build / serve	This hook point triggers the script with the default name `before_component_optimize.js` before the build/serve process minifies the content. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.

Hook Point	Supported Tooling Process	Description
`after_build`	build	This hook point triggers the script with the default name `after_build.js` immediately after the tooling concludes the build process.
`after_component_create`	build	This hook point triggers the script with the default name `after_component_create.js` immediately after the tooling concludes the create Web Component process. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.
`after_component_build`	build (debug mode only)	This hook point triggers the script with the default name `after_component_build.js` immediately after the tooling concludes the Web Component build process. A script for this hook point can be used to modify the build process specifically for a project that defines a Web Component.
`before_serve`	serve	This hook point triggers the script with the default name `before_serve.js` before the web serve process connects to and watches the app.
`after_serve`	serve	This hook point triggers the script with the default name `after_serve.js` after all build process steps complete and the tooling serves the app.
`before_watch`	serve	This hook point triggers the script with the default name `before_watch.js` after the tooling serves the app and before the tooling starts watching for app changes.
`after_watch`	serve	This hook point triggers the script with the default name `after_watch.js` after the tooling starts the watch and after the tooling detects a change to the app.
`after_component_package`	package	This hook point triggers the script with the default name `after_component_package.js` immediately after the tooling concludes the component package process.
`before_component_package`	package	This hook point triggers the script with the default name `before_component_package.js` immediately before the tooling initiates the component package process.

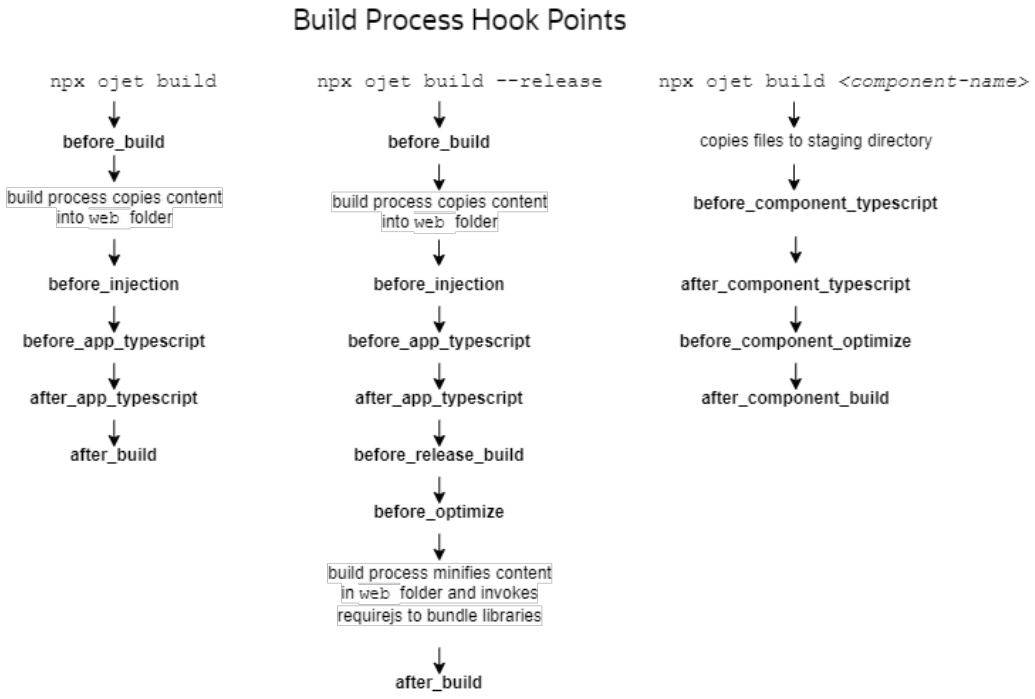
About the Process Flow of Script Hook Points

The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the create, build, serve, and restore workflow across the various command-line interface processes.

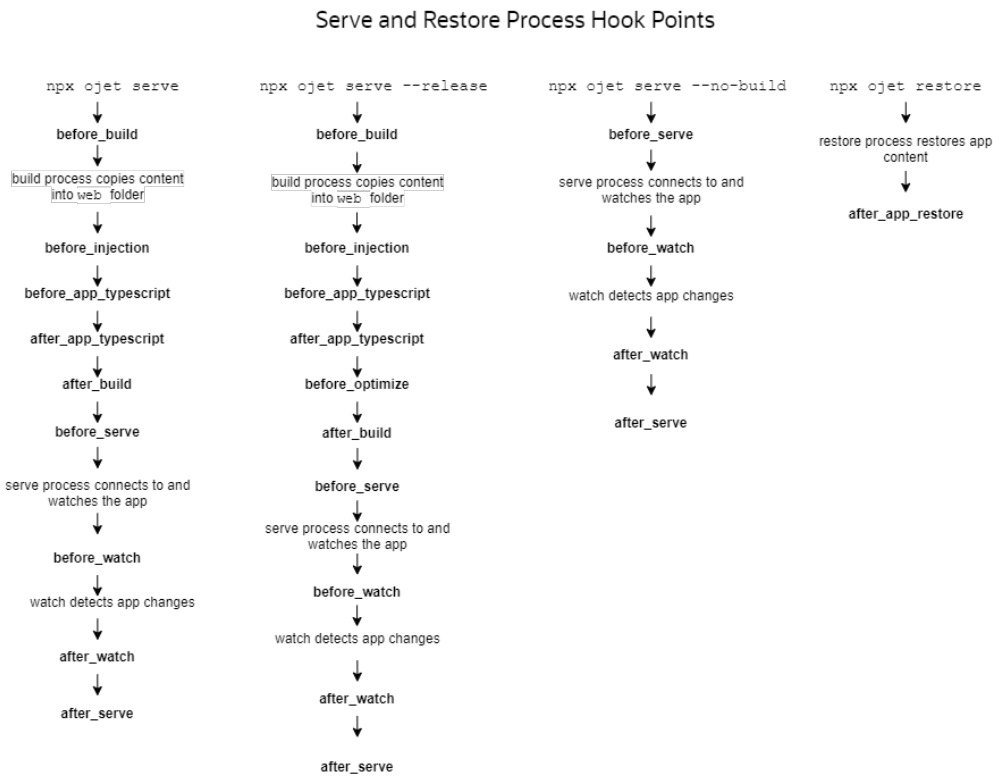
The following diagram shows the script hook point flow for the create process.



The following diagram shows the script hook point flow for the build process.



The following diagram shows the script hook point flow for the serve and restore processes.



Change the Hooks Subfolder Location

When you create an app, Oracle JET tooling generates script templates in the `/scripts/hooks` app subfolder. Your development effort may require you to relocate hook scripts to a common location, for example to support team development.

By default, the hooks system locates the scripts in the `hooks` subfolder using a generated JSON file (`hooks.json`) that specifies the script paths. When the tooling reaches the hook point, it executes the corresponding script which it locates using the `hooks.json` file. If you relocate hook script(s) to a common location, you must edit the `hooks.json` file to specify the new location for the hook scripts that you relocated, as illustrated by the following example.

```
{
    "description": "OJET-CLI hooks configuration file",
    "hooks": {
        "after_app_create": "scripts/hooks/after_app_create.js",
    ...
        "after_serve": "http://example.com/cdn/common/scripts/hooks/
after_serve.js "
    }
}
```

Create a Hook Script for Web Apps

You can create a hook point script to define a new command-line interface process step for your web app. To create a hook script, you edit the hook script template associated with a specific hook point in the tooling build and serve workflow.

The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the build and serve workflow across the various build and serve modes. Customization relies on script files and the script code that you want to trigger for a particular hook point. Note that the generated script templates that you modify with your script code are named for their corresponding hook point.

To customize the workflow for the build or serve processes, you edit the generated script template file named for a specific hook point. For example, to trigger a script at the start of the tooling's build process, you would edit the `before_build.js` script named for the hook point triggered before the build begins. That hook point is named `before_build`.

A basic example illustrates a simple customization using the `before_optimize` hook, which allows you to control the RequireJS properties shown in bold to modify the app's bundling configuration.

```
requirejs.config(
{
  baseUrl: "web/js",
  name: "main-temp",
  paths: {
  // injector:mainReleasePaths
    "knockout":"libs/knockout/knockout-3.x.x.debug",
    "jquery":"libs/jquery/jquery-3.x.x",
    "jqueryui-amd":"libs/jquery/jqueryui-amd-1.x.x",
     ...
  }
  // endinjector
  out: "web/main.js"
}
...
```

A script for this hook point might add one line to the `before_optimize` script template, as shown below. When you build the app with this customization script file in the default location, the tooling triggers the script before calling `requirejs.out()` and changes the `out` property setting to a custom directory path. The result is that the app-generated `main.js` is created in the named directory instead of the default `web/js/main.js` location.

```
module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running before_optimize hook.");
    configObj.requirejs.out = 'myweb/js/main.js';
    resolve(configObj);
  });
};
```

You can retrieve more information about the definition of `configObj` that is passed into many script hook points as a parameter by making the following modification in one of the build-related hook points and then running `ojet build`. For example, the `before_build.js` hook point can be modified as follows:

```
module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
      console.log("Running before_build hook.", configObj);
      resolve(configObj);
```

```
   });
};
```

The console from where you run the `ojet build` command then displays the available options that you can customize in `configObj`.

```
Cleaning staging path.
Running before_build hook {
  buildType: 'dev',
  opts: {
    stagingPath: 'web',
    injectPaths: {
      startTag: '// injector:mainReleasePaths',
. . .
```

Elsewhere, read examples that illustrate how to use the `configObj` to customize a hook point to, for example, add Express configuration options or write Express middleware functions in the `before_serve.js` hook point if the ready-to-use `ojet serve` options do not meet your requirements. See Serve a Web App to a HTTPS Server Using a Self-signed Certificate.

> **Tip**
>
> If you want to change app path mappings, it is recommended to always edit the `path_mappings.json` file. An exception might be when you want app runtime path mappings to be different from the mappings used by the bundling process, then you might use a `before_optimize` hook script to change the `requirejs.config` `paths` property.

The following example illustrates a more complex build customization using the `after_build` hook. This hook script adds a customize task after the build finishes.

```
'use strict';

const fs = require('fs');
const archiver = require('archiver');

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
   console.log("Running after_build hook.");

    //Set up the archive
    const output = fs.createWriteStream('my-archive.war');
    const archive = archiver('zip');

    //Callbacks for the archiver
    output.on('close', () => {
      console.log('Files were successfully archived.');
      resolve();
    });

    archive.on('warning', (error) => {
      console.warn(error);
    });
```

```
    archive.on('error', (error) => {
      reject(error);
    });

    //Archive the web folder and close the file
    archive.pipe(output);
    archive.directory('web', false);
    archive.finalize();
  });
};
```

In this example, assume the script templates reside in the default folder generated when you created the app. The goal is to package the app into a ZIP file. Because packaging occurs after the app build process completes, this script is triggered for the `after_build` hook point. For this hook point, the modified script template `after_build.js` will contain the script code to ZIP the app, and because the `.js` file resides in the default location, no hooks system configuration changes are required.

> **Tip**
>
> Oracle JET tooling reports when hook points are executed in the message log for the build and serve process. You can examine the log in the console to understand the tooling workflow and determine exactly when the tooling triggers a hook point script.

Pass Arguments to a Hook Script for Web Apps

You can pass extra values to a hook script from the command-line interface when you build or serve the web app. The hook script that you create can use these values to perform some workflow action, such as creating an archive file from the contents of the `web` folder.

You can add the `--user-options` flag to the command-line interface for Oracle JET to define user input for the hook system when you build or serve the web app. The `--user-options` flag can be appended to the build or serve commands and takes as arguments one or more space-separated, string values:

```
ojet build --user-options="some string1" "some string2" "some stringx"
```

For example, you might write a hook script that archives a copy of the build output after the build finishes. The developer might pass the user-defined parameter `archive-file` set to the archive file name by using the `--user-options` flag on the Oracle JET command line.

```
ojet build web --user-options="archive-file=deploy.zip"
```

If the flag is appended and the appropriate input is passed, the hook script code may write a ZIP file to the `/deploy` directory in the root of the project. The following example illustrates this build customization using the `after_build` hook. The script code parses the user input for the value of the user defined `archive-file` flag with a promise to archive the app after the build finishes by calling the NodeJS function `fs.createWriteStream()`. This hook script is an

example of taking user input from the command-line interface and processing it to achieve a build workflow customization.

```
'use strict';
const fs = require('fs');
const archiver = require('archiver');
const path = require('path');

module.exports = function (configObj) {
  return new Promise((resolve, reject) => {
    console.log("Running after_build hook.");

    //Check to see if the user set the flag
    //In this case we're only expecting one possible user defined
    //argument so the parsing can be simple
    const options = configObj.userOptions;
    if (options){
      const userArgs = options.split('=');
      if (userArgs.length > 1 && userArgs[0] === 'archive-file'){
        const deployRoot = 'deploy';
        const outputArchive = path.join(deployRoot,userArgs[1]);

        //Ensure the output folder exists
        if (!fs.existsSync(deployRoot)) {
          fs.mkdirSync(deployRoot);
        }

        //Set up the archive
        const output = fs.createWriteStream(outputArchive);
        const archive = archiver('zip');

        //callbacks for the archiver
        output.on('close', () => {
          console.log(`Archive file ${outputArchive} successfully created.`);
          resolve();
        });

        archive.on('error', (error) => {
          console.error(`Error creating archive ${outputArchive}`);
          reject(error);
        });

        //Archive the web folder and close the file
        archive.pipe(output);
        archive.directory('web', false);
        archive.finalize();
      }
      else {
        //Unexpected input - fail with information message
        reject(`Unexpected flags in user-options: ${options}`);
      }
    }
    else {
      //nothing to do
      resolve();
    }
```

```
  });
};
```

Use Webpack in Oracle JET App Development

You can use Webpack to manage your Oracle JET app, as well as the build and serve tasks.

If you decide to use Webpack, Oracle JET passes responsibility to Webpack to build and serve the source files of your Oracle JET project. Before you decide to use Webpack, note that it is not possible to use Webpack with Oracle JET apps that need to build, package or publish web components, to test apps using Oracle JET's Component WebElements UI automation library (TestAdapters), or to use the newer Core Pack components through their component class names. That is, the following syntax won't work:

```
import { SelectMultiple } from "oj-c/select-multiple";

export function Content() {
  return (
    <SelectMultiple labelHint="Select Multiple" itemText="foo"/>
  );
};
```

Though you can use Core Pack components using their custom element names, as in the following example:

```
import "oj-c/select-multiple";

export function Content() {
  return (
    <oj-c-select-multiple labelHint="Select Multiple" />
  );
};
```

If you decide to use Webpack in your Oracle JET app, you can specify it as a command-line argument when you scaffold the project, as demonstrated by the following example command:

```
ojet create <app-name> --template=basic --vdom --webpack
```

To build and serve the app with Webpack, simply run `ojet build` and `ojet serve` respectively. You cannot use the `ojet serve --release` command. To run a release build from your local development environment, use the `ojet build --release` command, and then use a static server of your choice (for example, http-server) from the `/web` folder.

To add Webpack to an existing Oracle JET app, run the following command from the root directory of your Oracle JET project:

```
ojet add webpack
```

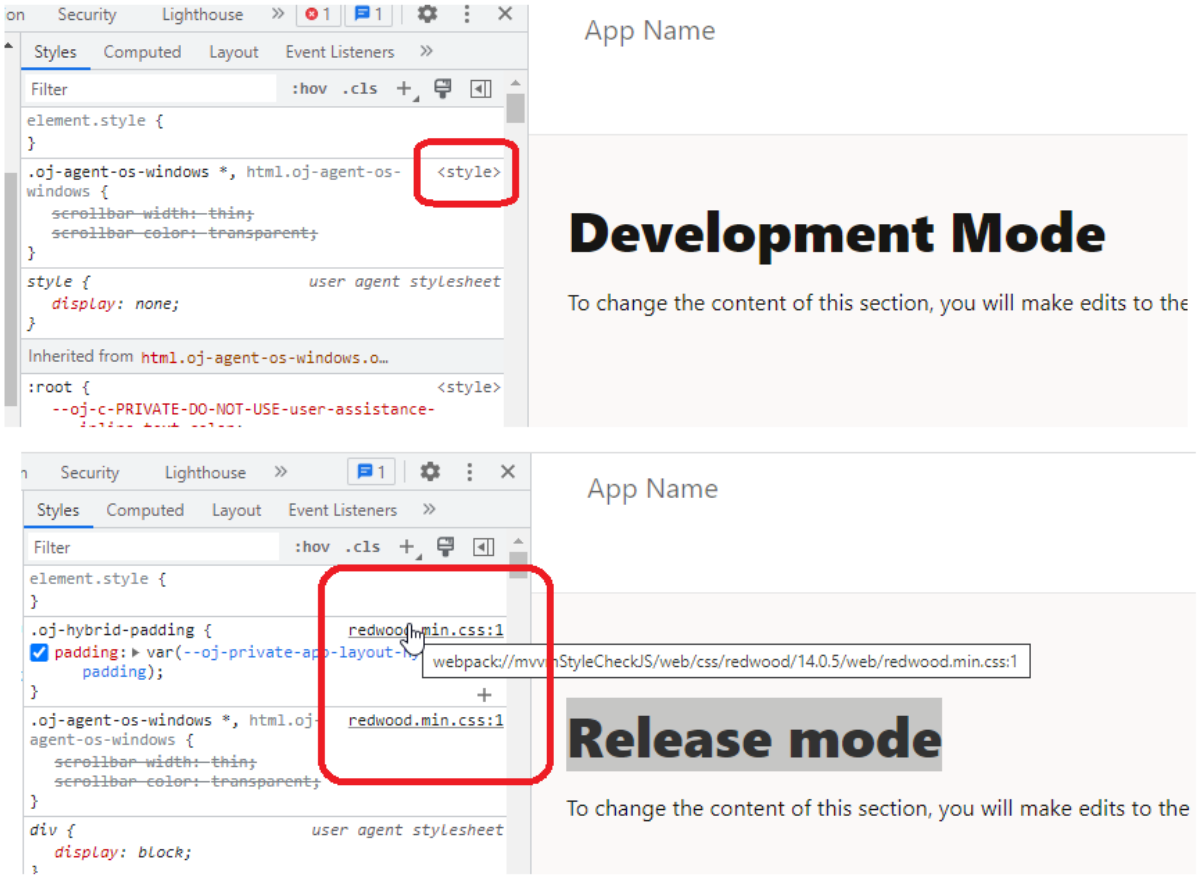
The files and directories in an Oracle JET project that uses Webpack differ to a project an app generated without specifying the `--webpack` argument. The following table describes the differences that result from use of the `--webpack` argument.

Option	Description
`/types`	The `types` directory contains a `/types/components/index.d.ts` file with a stub to add custom element tag names to the list of known JSX intrinsic elements. This is important if you plan to use custom elements within JSX that do not have `preact.JSX.IntrinsicElements` type definitions. Note that `/types` should only include type definitions which do not emit a JavaScript file after compilation. If you wish to rename the folder, make sure to also update the reference to it in the `tsconfig.json` file under the `typeRoots` option.
`ojet.config.js`	This file manages Oracle JET's default webpack configuration. For more detail about this file and how to configure it, see Configure Oracle JET's Default Webpack Configuration.
`tsconfig.json`	In constrast to the `tsconfig.json` generated for apps without Webpack, the `esModuleInterop` and `resolveJsonModule` flags are set to `true`. The `esModuleInterop` flag allows apps to standardize on default imports for all module types. Calls such as `import * as <importName> from "path/to/import"` should be written as `import <importName> from "path/to/import"`. The `resolveJsonModule` flag allows apps to directly import JSON files. Paired with Webpack's automatic support for resolving JSON file imports, you do not have to use the RequireJs `text!` plugin followed by `JSON.parse` to consume JSON files in the Oracle JET app.

As mentioned at the start of this topic, it is not possible to use Webpack in Oracle JET projects that build, package or publish web components, or projects that need to use Oracle JET theming. In other words, this means that you cannot use the following commands from the Oracle JET CLI:

- `ojet build (component|pack)`
- `ojet package (component|pack)`
- `ojet publish (component|pack)`

One other thing to note is that when you serve your Oracle JET app in development mode (the default), the Oracle JET app loads styles from memory and styles appear in the `<styles>` tag in the HTML of your browser. In contrast, when you serve an Oracle JET app that you have built in release mode (`ojet build --release`), styles come from the CSS file link that is included in the HTML file. In the following image, with an Oracle JET app instance that runs in development mode and release mode instance, you can see the different entries using the browser's developer tools.



Configure Oracle JET's Default Webpack Configuration

You can configure the default Webpack configuration generated by the Oracle JET CLI through the `webpack` function in the `ojet.config.js` file.

The `webpack` function receives an object with the Oracle JET build context (`context`) and Webpack configuration (`config`). Note the `buildType` property which indicates whether Webpack executes in development or release mode. As for `config`, the default Webpack configuration generated by Oracle JET, you can customize it to fit your needs.

To view the default options in the `ojet.config.js` file, add a console log statement to the `ojet.config.js` file, as demonstrated by the following examples:

```
. . .
webpack: ({ context, config }) => {
  if (context.buildType === "release") {
    // update config with release / production options
  } else {
    // Print out the default webpack configuration options
    // as a JSON string
    console.log(JSON.stringify(config));
    // Or let your terminal console determine how to
```

```
    // present the configuration
    console.log(config);
  }
. . .
```

Then build your Oracle JET project using the following command to render the default configuration in the terminal console:

```
ojet build
```

> **Tip**
>
> Create a JSON-formatted file in Visual Studio Code to view the default configuration in a more readable form to that returned by the terminal.

The default Webpack configuration for an Oracle JET app built in development mode includes the following top-level nodes:

```
{
  "entry": { },
  "output": { },
  "module": {},
  "resolve": {},
  "resolveLoader": {    },
  "plugins": [],
  "mode": "development",
  "devServer": { }
}
```

Once you have identified the configuration setting in Oracle JET's default Webpack configuration that you want to change, you add the alternative value in the `ojet.config.js` file. The following example illustrates how to change the port number when you serve your Oracle JET app in development mode using Webpack.

```
module.exports = {

  webpack: ({ context, config }) => {
    if (context.buildType === 'release') {
      // update config with release / production options
    } else {

      // update config with development options. In the following example, we
specify
      // a different server port number to the default of 8000
      config.devServer.port = 3000;

     // Print out the default webpack configuration options as a JSON string
     console.log(JSON.stringify(config));
     // Or let your terminal console determine how to present the
configuration
     console.log(config);
    }
    return config;
```

```
    }
};
```

3

Understand VComponent-based Web Components

Oracle JET provides you with a web component API, `VComponent`, to create web components that use virtual DOM rendering.

The web components that you create using VComponent use virtual DOM rendering. For those of you who previously used the Composite Component Architecture (CCA) to develop web components, you'll see many differences. Those of you who are familiar with the Preact library that underpins the Oracle JET virtual DOM architecture will see some familiar concepts. This chapter attempts to introduce you to the concepts that you'll need to know to develop VComponent-based web components.

One difference to note is that unlike CCA-based web components, VComponent-based web components do not use Knockout or its built-in expression evaluator to evaluate expressions. Instead, VComponent-based web components use JET's `CspExpressionEvaluator` to ensure that expressions you use comply with Content Security Policy. `CspExpressionEvaluator` supports a limited set of expressions to ensure compliance with Content Security Policy. Familiarize yourself with the syntax that JET's `CspExpressionEvaluator` supports when using expressions in your VComponent-based web component. See the CspExpressionEvaluator API documentation.

The JET tooling assists you with creating, packaging, and publishing web components. Usage of the JET tooling remains the same as for CCA-based web component development, but the output differs. We'll go through the creation of a standalone VComponent-based web component and a series of web components to include in a JET Pack in the next chapter.

For now, let's look at usage of `VComponent` to create web components, assuming that you have already acquired the Prerequisite Knowledge that we described in the introductory chapter of this guide.

> **Note**
>
> You can complement your reading of this chapter by also reading the VComponent entry in the API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET) and the Oracle JET VComponent Tutorial.

Hello VComponent, an Introduction

You write a VComponent-based web component as a TypeScript module in a file with the `.tsx` file extension.

The example that follows shows a VComponent named `HelloWorld` with a custom element name of `hello-world` in a file named `hello-world.tsx`.

```
import { ExtendGlobalProps, registerCustomElement } from 'ojs/ojvcomponent';
import { ComponentProps, ComponentType } from 'preact';
import componentStrings = require('ojL10n!./resources/nls/hello-world-
```

```
strings');
import 'css!./hello-world-styles.css';

type Props = Readonly<{
  message?: string;
}>;

/**
 *
 * @ojmetadata version "1.0.0"
 * @ojmetadata displayName "A user friendly, translatable name of the
component"
 * @ojmetadata description "A translatable high-level description for the
component"
 *
 */
function HelloWorldImpl({ message = 'Hello from  hello-world' }: Props) {
  return <p>{message}</p>;
}

export const HelloWorld: ComponentType<
  ExtendGlobalProps<ComponentProps<typeof HelloWorldImpl>>
> = registerCustomElement('hello-world', HelloWorldImpl);
```

The Oracle JET tooling helps you create VComponent web components by generating a template `.tsx` file plus additional files and folders with resources to support the component. The example just shown with the custom element name of `hello-world` was created by the following command:

```
ojet create component hello-world
```

If you want to create a VComponent-based web component in an app that does not use the virtual DOM architecture, you need to include `--vcomponent` in the command to create the component (`ojet create component hello-world --vcomponent`). The Oracle JET tooling also supports the creation of class-based web components if you append the `class` option to the `--vcomponent` parameter (`ojet create component hello-world --vcomponent=class`). The default behavior is to create function-based VComponents.

Irrespective of the type of VComponent that you create (class or function), the tooling generates these files in the directory referenced by the `components` property in the `appRootDir/oraclejetconfig.json` file. By default, the value of the `components` property is also `components`.

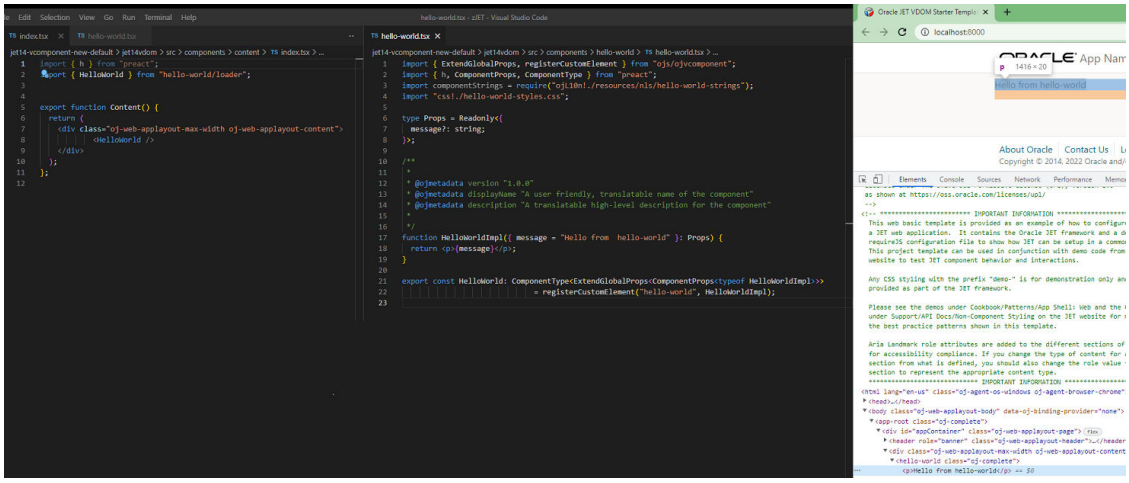
```
appRootDir/components/hello-world/
|   loader.ts
|   hello-world-styles.css
|   hello-world.tsx
|   README.md
+---resources
+---themes
```

Readers who previously developed CCA-based web components will recognize the `loader.ts` file that the Oracle JET tooling includes so that the component can be used by the Component Exchange, Oracle Visual Builder, and the Oracle JET tooling itself. For a VComponent-based web component, the `loader.ts` file includes an entry to export the VComponent module, as in the following example:

```
export { HelloWorld } from "./hello-world";
```

Once you build a VComponent web component, you can import it into the app where it is to be used. The following example demonstrates how you import our example component into the `content` component of an app that was scaffolded using the virtual DOM architecture starter template:

```
import { h } from "preact";
import { HelloWorld } from "hello-world/loader";

export function Content() {
  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <HelloWorld />
    </div>
  );
}
```



Metadata for VComponents

JET metadata expresses information that may be useful to both tools and consumers of the VComponent-based web components that you create.

You write metadata in the VComponent's module class. You'll have seen examples of this metadata in the HelloWorld VComponent that we introduced earlier. Specifically, the HelloWorld VComponent included a TypeScript decorator, `@customElement("hello-world")`, to add custom element behavior to the VComponent at runtime, and it is also used at build time as a source of the component's "`name`" metadata. The other example is the use of the `@ojmetadata` doc annotation where a series of entries provide version, display name, and description information, as in the following example:

```
* @ojmetadata version "1.0.0"
* @ojmetadata displayName "A user friendly, translatable name of the
component"
* @ojmetadata description "A translatable high-level description for the
component"
```

You'll notice that each `@ojmetadata` annotation specifies a single name/value pair. The values must be valid JSON values. As shown above, string values should be double-quoted. Object, array, and primitive values can be specified directly within the annotation (without quotes). You can also extend the metadata to append extra information in an extension field, as shown by the following example.

```
* @ojmetadata extension {
*    vbdt: {
*      someVisualBuilderDesignTimeField: true
*    }
* }
```

For reference information about JET Metadata, see JET Metadata.

Nest VComponents

Custom element-based VComponents can be embedded directly into HTML.

This allows you to integrate VComponents into existing Oracle JET content, including into composite components, oj-module content, or pages authored in Oracle Visual Builder. In addition to being hosted within HTML, VComponents can be nested inside of other VComponents. A parent VComponent can reference a child VComponent using the component class name. In the following example, a VComponent class, `HelloParent`, nests a child VComponent, `Hello`.

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild } from "preact";
import { Hello } from "oj-greet/hello/loader";

type Props = {
  message?: string;
};

/**
 * @ojmetadata pack "oj-greet"
 * ...
 */
@customElement("oj-greet-hello-parent")
export class HelloParent extends Component<ExtendGlobalProps<Props>> {
  static defaultProps: Partial<Props> = {
    message: "Hello from oj-greet-hello-parent!",
  };

  render(props: Props): ComponentChild {
    return (
      <div>
        <p>{props.message}</p>
        <p>The HelloParent VComponent nests the Hello VComponent class in the
          next line:</p>
        <Hello />
      </div>
    );
  }
}
```

The resulting content in the HTML is:

```
<div class="oj-web-applayout-max-width oj-web-applayout-content">
 <oj-greet-hello-parent class="oj-complete">
    <div>
      <p>Hello from oj-greet-hello-parent!</p>
        <p>The HelloParent VComponent nests the Hello VComponent class in the
next line:</p>
          <oj-greet-hello class="oj-complete"><p>Hello from oj-greet-hello!
</p></oj-greet-hello>
    </div>
 </oj-greet-hello-parent>
</div>
```

An `<oj-greet-hello>` custom element ends up in the live DOM.

VComponent Properties

Properties are read-only arguments of a VComponent class that you pass into an instance of the VComponent.

Properties that you declare may also be passed to web components as HTML attributes. Essentially, the properties of a VComponent API component module are like function arguments in JSX and attributes in HTML usages.

Declare VComponent Properties

You declare a VComponent property through a type alias that is, by convention, named `Props`.

Each field in the type represents a single public component property. A field specifies the property's name, type, and whether the value for the field is optional or required. Default values are specified in the `static defaultProps` field on the component class.

In the following example, we declare a single property (`preferredGreeting`) of type string. TypeScript's optional indicator (?) identifies it as an optional property, and the default value of `Hello` is specified in the `static defaultProps` field.

One subtle requirement that may be easy to miss: to associate the properties class with the VComponent implementation, you need to specify the class as the value of the VComponent's first type parameter (`export class WithProps extends Component<ExtendGlobalProps<Props>>`).

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild } from "preact";

type Props = {
  preferredGreeting?: string;
};

/**
 * @ojmetadata pack "oj-greet"
 * ...
 */
@customElement("oj-greet-with-props")
export class WithProps extends Component<ExtendGlobalProps<Props>> {
```

```
  static defaultProps: Partial<Props> = {
    preferredGreeting: "Hello",
  };

  render(props: Props): ComponentChild {
    return <p>{props.preferredGreeting}, World!</p>;
  }
}
```

Reference Properties in JSX

To work with the properties, VComponent requires that you first associate the property class with the VComponent instance implementation.

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild } from "preact";

type Props = {
  preferredGreeting?: string;
};

/**
 * @ojmetadata pack "oj-greet"
 */
@customElement("oj-greet-with-props")
export class GreetWithProps extends Component<ExtendGlobalProps<Props>> {
  static defaultProps: Partial<Props> = {
    preferredGreeting: "Hello from oj-greet-with-props!"
  };

  render(props: Readonly<Props>): ComponentChild {
    return <p>{props.preferredGreeting}</p>;
  }
}
```

Access Properties

You can access declared properties in the VComponent API component implementation through a special object: `this.props`. An example of this usage can be found at line 12, where the `this.props` field extracts the value of the `preferredGreeting` property into the variable `greeting`. This variable subsequently influences the state of the rendered virtual DOM tree, where the value of the `preferredGreeting` property gets embedded into the virtual DOM at line 16.

One point to keep in mind is that the property values in `this.props` are always defined by the consumer of the VComponent API component. In the HTML case, `this.props` is populated based on attribute/property values specified on the custom element by the application. In the case where the VComponent API component is used within a parent VComponent API component, the property values are provided by the parent component. A VComponent API component implementation can read these property values, but must never mutate the `this.props` object.

Reference Properties of a Child Component in JSX

VComponent API custom elements can also be embedded inside of other parent VComponent API custom elements.

As was described in Nest VComponents, a VComponent API component parent can refer to a child using the VComponent API component's implementation class name directly. Inside of JSX, always specify component properties using their camelCase property names.

Here is an example of a VComponent, `GreetWithPropsParent`, that demonstrates this:

```
import { h, Component } from "preact";
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { GreetWithProps } from "oj-greet/with-props/loader";

/**
 * @ojmetadata pack "oj-greet"
 * ...
 */
@customElement("oj-greet-with-props-parent")
export class GreetWithPropsParent extends Component<ExtendGlobalProps<Props>>
{
  render() {
    return (
      <div>
        <GreetWithProps preferredGreeting="Hola" />
      </div>
    );
  }
}
```

Note how the sample uses the `preferredGreeting` property name, not the `preferred-greeting` attribute name.

Type-Checking Support

Although the use of different naming conventions between HTML and JSX markup appears confusing at first, it is important to use only property names within JSX to maintain type checking.

When specifying a JSX element like this:

```
<oj-greet-with-props preferredGreeting="Hey there"/>
```

Or this:

```
<GreetWithProps preferredGreeting="Hola"/>
```

The properties on each JSX element populate a `props` object that eventually ends up populating the child component's `this.props` field. The type of this `props` object is based on the child VComponent API component instance's `props` type parameter. So using the property names as declared by the VComponent API component instance's property type, ensures type checking (and catching errors) happens in the parent component's JSX.

Global HTML Attributes

The naming convention of camelCase supports referencing component properties from within JSX. Ideally, this same convention can work for global HTML attributes, such as `id` or `tabIndex`. However, not all global HTML attributes are exposed as properties. For example, `aria-` and `data-` attributes do not have property equivalents.

This leads to the following rules for working with global HTML properties/attributes:

- If the global HTML attribute is available as a property, use the property name.
- If the global HTML attribute is not available as a property, use the attribute name.

In many cases, global HTML attribute names will be identical to the property name (such as `id`, `title`, and `style`). However, there are some cases where the attribute and property name differ, or where the property name requires a specific case-folding. For example, since attributes are case insensitive, HTML allows any capitalization of the `tabindex` attribute. However, JSX requires that you use the actual property name `tabIndex`:

```
protected render() {
  // While "tabindex" is a valid way to specify the tab index
  // in an HTML document, in JSX, the property name "tabIndex"
  // must be used.
  return <div tabIndex="0" />
 }
```

There is one exception to the rule that governs property name references. Although the property name for specifying style classes is `className`, this name is not commonly known. `VComponent` allows use of the more familiar attribute name `class`:

```
protected render() {
  // Use "class" instead of "className"
  return <div class="awesome-class" />
}
```

Children and Slot Content

In addition to exposing properties, components can also allow children to be passed in. With VComponent API custom elements, children are specified in one of two ways:

1. As direct children, with no `slot` attribute. This is also known as the *default slot*.
2. As a named slot, with the name set through the `slot` attribute.

Components can leverage both of these approaches. For example, the following `oj-c-collapsible` element is configured both with default slot content, as well as content in the `header` named slot:

```
<oj-c-collapsible>
  <h3 slot='header'>This is named slot content</h3>
     <span>This is default slot content</span>
    </oj-c-collapsible>
```

The VComponent API supports authoring of custom elements that expose default slots, named slots, or both.

One thing to note is that conditional rendering of slots can have unexpected consequences. When possible, conditionally render content inside of the slot rather than conditionally render the slot itself. That is, rather than:

```
condition && <div slot="actionButtons">
```

Write:

```
<div slot="actionButtons"> {
   condition && <> buttons go here </>
  }
</div>
```

We also provider a `Remounter` component, described in Refresh Custom Elements with Dynamic Children and Slot Content, that you can use to ensure that a custom element re-renders correctly when its children or slot content changes dynamically.

Default Slots

VComponent API favors the use of code constructs over external metadata for defining a component's public API. There is no need to declare a VComponent API custom element children/slot contract through JSON metadata; instead default slots are added by writing code.

The children/slot contract for the VComponent API custom element is defined by adding fields to a `Props` class. In particular, you indicate that a component can accept default slot content by declaring a children property of type `ComponentChildren`:

```
import { h, Component, ComponentChildren } from 'preact';
     type Props = {
      preferredGreeting?: string;
      children?: ComponentChildren;
    }
```

And, you can then associate the `Props` class with the VComponent through the `Props` type parameter:

```
@customElement('oj-greet-with-children')
   export class GreetWithChildren extends Component<ExtendGlobal<Props>> {
  }
```

Once this is done, any default slot children will be made available to the VComponent API component implementation through `props.children`. This is true regardless of whether the component implementation is used as a custom element within an HTML document, a custom element within JSX, or through the VComponent component implementation class within JSX.

The VComponent component implementation is free to place the default slot children anywhere within the component's virtual DOM tree. For example, a VComponent API button likely would place these children inside of an HTML `<button>` element:

```
protected render() {
      return <button> { props.children } </button>;
 }
```

Named Slots

Like the default slot, named slots are also declared as fields on the `props` class.

Named slot declarations must adhere to two conventions:

- The named slot field must use the `Slot` type.
- The name of the field must match the slot name.

The declaration for a slot named `startIcon` looks like this:

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild } from "preact";
import "oj-c/avatar";
import { GreetWithChildren } from 'oj-greet/with-children/loader'

type Props = {
    startIcon?: Slot;
}

/**
 * @ojmetadata pack "oj-greet"
 * @ojmetadata dependencies {
 *   "oj-greet-with-children": "^1.0.0"
 * }
 */
@customElement('oj-greet-with-children-parent')
export class GreetWithChildrenParent extends
Component<ExtendGlobalProps<Props>> {
  render() {
    return (
      <div>
        <p>This child is rendered as a VComponent class:</p>
        <GreetWithChildren startIcon={<oj-c-avatar initials="HW"
size="xs" />}>
          World
          </GreetWithChildren>
      </div>
    );
  }
```

When the VComponent is referenced through its class, named slot content is provided by specifying virtual DOM nodes directly as values for slot properties, as demonstrated in the following parent VComponent.

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild } from "preact";
```

```
import "oj-c/avatar";
import { GreetWithChildren } from "oj-greet/with-children/loader";

/**
 * @ojmetadata pack "oj-greet"
 * @ojmetadata dependencies {
 *   "oj-greet-with-children": "^1.0.0"
 * }
 */
@customElement("oj-greet-with-children-parent")
export class GreetWithChildrenParent extends
Component<ExtendGlobalProps<Props>> {
  render() {
    return (
      <div>
        <p>This child is rendered as a VComponent class:</p>
        <GreetWithChildren startIcon={<oj-c-avatar initials="HW"
size="xs" />}>
          World
        </GreetWithChildren>
      </div>
    );
  }
}
```

Refresh Custom Elements with Dynamic Children and Slot Content

Virtual DOM architecture provides a Remounter component to make sure that a custom element re-renders correctly when its children or slot content changes dynamically.

A concrete example demonstrates when to use the Remounter component. In the following example, the `oj-c-button` element's display of a start icon depends on the value of the `showStartIcon` property. We want to ensure that the `oj-c-button` renders the start icon in the correct location when the `showStartIcon` property changes.

```
type Props = {
  showStartIcon?: boolean
}

function ButtonStartIconSometimes(props: Props) {
  return (
    <oj-c-button label="Click Me!">
      { props.showStartIcon && <span slot="startIcon" class="some-icon-
class" /> }
    </oj-c-button>
  )
}
```

A more appropriate approach for the virtual DOM architecture is to assign a unique key that reflects the `oj-c-button showStartIcon` property's different states. In the above example, assigning a unique key is straightforward, as there are only two possible states. However, in some cases producing a unique key for all possible children states is more challenging. For these more challenging cases, use the Remounter component to wrap a single custom element child and generate a unique key based on the current set of children. The following

revised example demonstrates how you use the Remounter component to ensure that the `oj-c-button` re-renders with the start icon in the correct location:

```
import { h } from 'preact';
import { Remounter } from 'ojs/ojvcomponent-remounter';
import "oj-c/button";

type Props = {
  showStartIcon?: boolean
}

function ButtonStartIconSometimes(props: Props) {
  return (
    <Remounter>
      <oj-c-button label="Click Me!">
        { props.showStartIcon && <span slot="startIcon">Start Icon</span> }
      </oj-c-button>
    </Remounter>
  )
}

export { ButtonStartIconSometimes };
```

Note that you only need to use the Remounter component when configuring custom elements where the number of types of children change across renders.

Template Slots

In addition to simple, non-contextual slots, JET components also support slots that can receive context. These are called template slots.

Template slots are typically found in collection components that iterate over a data set, stamping out content for each item or row. For example, `<oj-list-view>` exposes an `itemTemplate` slot that controls how the content of each list item renders. Within HTML, a `template` element with a `slot` attribute specifies a template slot, as in the following example:

```
<oj-list-view  data="[[ items ]]">
  <template slot="itemTemplate" data-oj-as="item">
    <div>
      <oj-bind-text value="[[item.data.value]]"></oj-bind-text>
    </div>
  </template>
</oj-list-view>
```

VComponent API custom elements can also expose template slots. To understand how this works, consider a greeting component that takes an array of names to greet and renders a greeting for each name. The property declaration might look like this:

```
class Props {
  names: Array<string>
}
```

It is possible to just iterate over the names and render the content for each item:

```
protected render() {
  return (
    <div>
      { this.props.names.map(name => <div>Hello, {name}!</div>) }
    </div>
  );
}
```

With the above approach, the decision about how to render each greeting would be hardcoded into the component implementation. Instead, this can be made more flexible by exposing a template slot that allows the app to customize how each greeting is rendered.

Similar to simple, non-contextual slots, template slots are declared as properties with a well known type: `TemplateSlot`. Let's take a look at this type alias:

```
export type TemplateSlot<Data> = (data: Data) => Slot;
```

The `TemplateSlot` is a generic function type that accepts a single argument: the data to use when rendering a specific instance of the template. The type of this data is defined through the `Data` type parameter, which must be specified when the `TemplateSlot` property is declared.

Here is a new version of the greeting component that delegates rendering to an optional `greetingTemplate` slot:

```
oj-greet/hello-many.tsx:
1    import { h, Component } from 'preact';
2    import { customElement, ExtendGlobalProps, TemplateSlot } from 'ojs/
ojvcomponent';
3
4    export type GreetingContext = {
5      name: string;
6    }
7
8    type Props = {
9      names: Array<string>;
10      greetingTemplate?: TemplateSlot<GreetingContext>;
11    }
12
13    /**
14     * @ojmetadata pack "oj-greet"
15     */
16    @customElement('oj-greet-hello-many')
17    export class GreetHelloMany extends Component<ExtendGlobalProps<Props>>
{
18      render() {
19        return (
20          <div>
21            {
22              this.props.names.map((name) => {
23                return this.props.greetingTemplate?.({ name }) ||
24                         <div>Hello, { name }!</div>
25              })
26            }
```

```
27          </div>
28        );
29      }
30    }
```

This sample declares the `greetingTemplate` slot at line 10. Note that the `Data` type parameter must be an object type. The sample uses the `GreetingContext` type as declared at line 4.

The template slot (if non-null) is invoked for each item in the names array at line 23. The sample passes in an object of type `GreetingContext` with each invocation. Alternatively, if no slot is provided, it returns the default content at line 24.

Provide Template Slot Content within HTML

After you expose the template slot in the VComponent implementation, then within HTML, you provide slot content the same as any JET custom element: by specifying a `<template>` element with a `slot` attribute. Within the `template` element, you use JET binding expressions and elements to render the desired greeting:

```
<oj-greet-hello-many names="[[ ['Joel', 'Mike', 'Jonah' ] ]]">
  <template slot="greetingTemplate" data-oj-as="greeting">
    <div>
      Hi, <oj-bind-text value="[[ greeting.name ]]"></oj-bind-text>!
    </div>
  </template>
</oj-greet-hello-many>
```

Template Slots in JSX

When rendering a component in JSX through its VComponent API component class (such as `<GreetHelloMany>`), template slots are passed in as functions that adhere to the TemplateSlot contract. This means you must implement template slots as functions that take some data, and return either a single virtual DOM node or an array of nodes.

This might look something like:

```
<GreetHelloMany names={names}
    greetingTemplate={ (data) => <div>Hello, { data.name}!</div> }
```

Of course, you can also reference the `GreetHelloMany` component by using its custom element tag name.

As the previous HTML sample shows, custom element template slots are specified using JET binding expressions (such as `value="[[ greeting.name ]]"`) and elements (such as `oj-bind-text`) inside a `<template>` element. While this approach fits in nicely within an HTML document alongside other content that is configured using JET bindings, it doesn't fit well inside of a JSX render function. Within JSX, rather than configuring template slot content using JET binding syntax, JSX syntax is preferred.

To allow template slot content to be specified using JSX-based render functions, VComponent API introduces a special, VComponent-specific property on the `<template>` element: the `render` property. The type of this property is `TemplateSlot`.

This allows us to configure template slots on custom elements using JSX-based render functions, for example:

```
<oj-greet-hello-many names={names}>
    <template slot="greetingTemplate"
       render={ (data) => <div>Hello, { data.name}!</div> }/>
  </oj-greet-hello-many>
```

Note that you still need to specify a `<template>` element with a `slot` attribute. Rather than configuring the template slot with JET's binding syntax, instead specify a `TemplateSlot` function that returns virtual DOM.

A more complete parent component shows this:

oj-greet/hello-many-parent.tsx:

```
1     import { h, Component } from 'preact';
2     import { customElement, GlobalProps } from 'ojs/ojvcomponent';
3     import "oj-c/avatar";
4     import { GreetHelloMany, GreetingContext } from 'oj-greet/hello-many/
loader';
5
6     /**
7      * @ojmetadata pack "oj-greet"
8      */
9     @customElement('oj-greet-hello-many-parent')
10    export class GreetHelloManyParent extends
Component<ExtendGlobalProps<Props>> {
11      render() {
12
13        const names = [ 'Joel', 'Mike', 'Jonah' ];
14
15        return (
16          <div>
17            <p>This child is rendered as a custom element:</p>
18            <oj-greet-hello-many names={names}>
19              <template slot="greetingTemplate"
render={ this.renderGreeting }/>
20           </oj-greet-hello-many>
21           <br />
22            <p>This child is rendered as a VComponent class:</p>
23            <GreetHelloMany names={names}
greetingTemplate={ this.renderGreeting }/>
24          </div>
25        );
26      }
27
28      private renderGreeting(data: GreetingContext) {
29        const name = data.name;
30        const firstInitial = name.charAt(0);
31        const greeting = name.length < 5 ? 'Hey' : 'Hi';
32
33        return (
34          <p class="centerAlignVertical">
35            <oj-c-avatar size="xxs" initials={ firstInitial } />
```

```
36            {greeting}, { name }!
37          </p>
38        );
39      }
40    }
```

In the above example, the `render` property provides JSX-based content for the `<oj-greet-hello-many>` custom element, which happens to be implemented with VComponent API. However, this property can also be used when configuring template slot content for any JET component. For example, you can configure the `<oj-list-view> itemTemplate` slot as follows, even though this custom element is not implemented with the VComponent API:

```
protected render() {
  <oj-list-view  data={ this.props.items }>
    <template slot="itemTemplate"
     render={ ( item ) => { return <div>{ item.data.value }</div> } } />
  </oj-list-view>
```

Understand Events and Actions

Two terms seem interchangeable at first, but in VComponent API, `event` and `action` have two distinct meanings:

- `event` specifically refers to DOM Events that are dispatched by calling to dispatchEvent.
- `action` is a higher-level abstraction for event-like APIs, which may or may not actually involve dispatching an Event at the DOM level.

This distinction arises due to the fact that VComponent API component instances can be used in two ways:

- As a custom element, using the string tag name.
- As a VComponent API component, using the component implementation class.

When a VComponent API component is used as a custom element, invoking an action results in the dispatch of a DOM event.

However, when referencing a VComponent API component through its implementation class, no DOM event is created or dispatched. Instead, the action callback provided by the parent component is invoked directly.

To support these different usage models, a higher level abstraction than DOM events is required. VComponent API actions provide that abstraction.

For simplicity, usage of the term **action** refers to the general behavior by which VComponent API component instances notify the outside world of activity. Whereas usage of the term **event** is reserved specifically for DOM events that are dispatched by custom (or plain old HTML) elements.

Listeners

Event listeners are functions that take a DOM `event` and have no return value. VComponents can listen for and respond to standard HTML events plus custom events on custom elements.

The naming convention that you use for the event listener differs depending on whether you listen for a standard HTML event or custom event.

For standard HTML events, such as `click`, `change`, `mouseover`, add a property name that uses the naming convention: `on<UpperCaseStandardEventName>`. The following example shows you how to register an event listener for a `click` event.

```
render() {
  return <div onClick={this._handleClick}>Click Me!</div>
}
```

For custom events such as `<oj-c-button>`'s `ojAction` event, use the `on<customEventName>` naming convention. The following example shows you how to register an event listener for an `ojAction` event.

```
protected render() {
   return <oj-c-button label="Click Me!" onojAction={this._handleAction}>
               <oj-c-button>
  }
```

Note how the first character of the custom event name is not capitalized compared to the standard event (`onojAction` versus `onClick`).

There are a number of ways to enable event listener access to a VComponent instance. You can, for example, explicitly call `bind(this)` on the event listener function or, alternatively, use one of the following approaches:

- Define and use an arrow function inline in the `render()` method.
- A class method can be bound and saved away in the constructor.
- An arrow function can be declared and stored in a class field.

The last two options avoid creating a new function on each call to the `render()` method and, by using the same function instance across all `render()` methods, avoid virtual DOM diffs that cause DOM `addEventListener` and `removeEventListener` calls on each call to the `render()` method. The class field approach (`private _handleEvent`), demonstrated in the following example, is slightly more concise.

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { Component } from "preact";
import "oj-c/button";

type Props = Readonly<{
  message?: string;
}>;

@customElement("oj-greet-with-listeners")
export class GreetWithListeners extends Component<ExtendGlobalProps<Props>> {

  render() {
    return (
      <div>
        <div onClick={this._handleEvent}>
          <p>Hello, World!</p>
        </div>
        <oj-c-button label="Click Me!" onojAction={this._handleEvent}></oj-c-
button>
      </div>
```

```
    );
  }

  private _handleEvent = (event: Event) => {
    console.log(`Received ${event.type} event`);
  };
}
```

Actions

We need to make a distinction between `event` and `action` because of the different behavior that occurs when you use a VComponent as a custom element or a component class.

In a VComponent, an `event` refers to DOM Events that are dispatched through a call to dispatchEvent while an `action` is a higher-level abstraction for event-like APIs, which may or may not actually involve dispatching an event at the DOM level. When you use a VComponent as a custom element, invoking an action results in the dispatch of a DOM event while no DOM event is created or dispatched when you use the component class. Instead, for the latter case, the action callback provided by the parent VComponet is invoked directly. VComponent action provides a higher-level abstraction than DOM events that is needed to support these two usage models.

Declare Actions

You need to be able to define actions to dispatch in response to VComponents-defined events.

The following example demonstrates how you add a `responseDetected` event action to a VComponent that is dispatched in response to a user action, such as a click. You add a field to the `Props` class. The field that you add must follow the standard event listener property naming convention (`on<UpperCaseEventName>`) and it must use the Action type defined by the `ojs/ojvcomponent` module.

```
import { customElement, ExtendGlobalProps, Action } from 'ojs/ojvcomponent';
    type Props = {
      preferredGreeting?: string;
      // This is an action declaration:
      onResponseDetected?: Action;
    }
```

Dispatch Actions

VComponent's Action type is a callback function.

```
export type Action<Detail extends object = {}> = (detail?: Detail) => void;
```

To dispatch an action, the VComponent invokes the Action-typed property as a function. Actions are typically dispatched in response to some underlying event. In the following

example, the VComponent instance dispatches the `responseDetected` action in response to a click:

```
private _handleClick = (event: MouseEvent) => {
   this.props.onResponseDetected?.();
 }
```

Note that the consumer of the component is not required to provide a value for `onResponseDetected`. As a result, we need to guard against a null value for `this.props.onResponseDetected`. To do this, we can take advantage of TypeScript's support for the optional chaining operator (`?.`). This allows us to invoke the action callback if it is provided but short-circuit if not, without the need for a more verbose null check.

Respond to Actions

The response to an invoked action depends on usage.

If the VComponent is used as a custom element (either within an HTML document or within JSX in a parent VComponent), the VComponent framework creates a DOM CustomEvent that it dispatches through the custom element. The event type is derived from the name of the Action property by removing the on prefix and lower casing the first letter. For example, the `onResponseDetected` action results in the dispatch of a `responseDetected` DOM event type.

If the VComponent is being used by a parent VComponent and is referenced by its class name rather than the custom element name, no DOM event is created. If the parent VComponent provides a value for the Action property, this is invoked directly.

In JSX, action callbacks are always specified using the Action property name. This is true regardless of whether the parent references the child VComponent by its custom element name of class:

```
<p>This child is rendered as a custom element:</p>
<oj-greet-with-actions onresponseDetected={this.handleResponse}/>

<p>This child is rendered as a VComponent class:</p>
<GreetWithActions onResponseDetected={this.handleResponse}
```

However, if the custom element lives within an HTML document, event listeners are typically registered with JET's event binding syntax. This might look something like:

```
<oj-greet-with-actions on-response-
detected="[[ expressionPointingToEventHandler ]]">…
            </oj-greet-with-actions>
```

Though it is also possible to call the DOM addEventListener API directly.

Action Payloads

You may have noticed that Action is a generic type with a Detail type parameter. The Detail type parameter is useful when the action needs to deliver additional information beyond just the action type.

For example, our Greeting component may want to include a flag along with the `responseDetected` action to indicate urgency. This would be declared using the Detail type parameter as follows:

```
type Props = {
  preferredGreeting?: string;
  onResponseDetected?: Action<{
  urgent: boolean;
 }>;
}
```

When invoking the action, the detail payload is passed in as an argument to the action callback, as in the following example:

```
private _handleClick = (event: MouseEvent) => {
   // Pass in a detail payload.  Determine urgency based on
   // number of clicks.
   this.props.onResponseDetected?.({
     urgent: event.detail > 1
  });
}
```

The `Detail` type parameter can also be specified using a type alias, as in the following example:

```
export type ResponseDetectedDetail = {
  urgent: boolean;
};

type Props = {
  preferredGreeting?: string;
  onResponseDetected?: Action<ResponseDetectedDetail>;
};
```

On the consuming side, there is one subtlely in how the detail payloads are accessed. In the custom element case, the action callback is registered as a DOM EventListener. That is, when using the following form:

```
<oj-greet-with-actions onresponseDetected={this.handleEventResponse}/>
```

The callback acts as a true DOM event listener, and, as such, receives a single `event` argument of type `CustomEvent<Detail>`. However, when using the VComponent class form:

```
<GreetWithActions onResponseDetected={this.handleActionResponse}/>
```

The callback will again receive a single argument, but of type `Detail` rather than `CustomEvent<Detail>`. The difference between custom element usage and VComponent class usage is admittedly non-obvious. Our recommendation for you is to use the VComponent class form when that is available.

Manage State Properties

Components may track internal state that is not reflected through their properties. VComponent API provides a state mechanism to support this.

A VComponent API custom element can determine what content to render based exclusively on properties that are passed into the component by the parent component. This is useful for VComponent API components that are fully controlled by the parent component. However, some components may benefit from their own internal state properties that are not passed in, but rather exist locally in the component and render content based on state changes. VComponent authors can leverage Preact's local state mechanism for these cases.

Declare State

The process of declaring local state fields is very similar to the way that you define VComponent API properties. In both cases, start by declaring a type.

This example updates our component to display a goodbye message in response to the user clicking a Done button. The sample uses a boolean local state field done to track whether this state is reached.

```
type State = {
   done: boolean;
   // Other state fields go here
};
```

As with the properties type, we associate the state type with the VComponent implementation by leveraging generics. The VComponent API component class exposes two type parameters:

- `Props`: the first type parameter specifies the properties object type
- `State`: the second type parameter specifies the State object type

The new declaration with both type parameters looks like this:

```
export class GreetWithState extends Component<ExtendGlobalProps<Props>,
State> {
    }
```

Each VComponent API component has access to the local state through the `this.state` field. This field must be initialized at construction time. Initialization can be done in one of two ways.

If your VComponent API component instance has a constructor, initialize local state there:

```
export class GreetWithState extends Component<ExtendGlobalProps<Props>,
State> {
      constructor() {
        this.state = {
          done: false
        };

        // Do other construction-time work here
      }
```

Alternatively, TypeScript supports inline initialization of class fields. This slightly more compact form works well if you do not otherwise need a constructor:

```
export class GreetWithState extends Component<ExtendGlobalProps<Props>,
State> {
      state = {
        done: false
      };

      // Component implementation goes here
   }
```

Once local state has been declared and initialized, it can be referenced from within the render function to adjust how the virtual DOM content is rendered. For example, this sample shows our greeting component rendering a different message when the conversation is "done":

```
render(props: Props, state: State) {
        // Derive greeting message off of the "done" state field
        const greeting = state.done ?
          'Goodbye' :
          props.preferredGreeting;

        return (
          <div onClick={this._handleClick}>
            <p>{greeting}, World!</p>
          </div>
        );
      }
```

Update State

Updating local state has an important side effect: it triggers re-rendering of the component.

Note that `this.state` is declared as a `Readonly` type. Other than the initial assignment to `this.state` in the constructor (or class field initialization), neither `this.state` nor fields on `this.state` should be mutated directly.

Instead, state updates are performed by calling the Preact Component's `setState` method. This method has two forms. The first form simply takes an object representing the new state. For example, we can update our `done` state field by calling:

```
this.setState({ done: true });
```

This call queues the state update, which will trigger an (asynchronous) re-render of the component with the new state.

Although our sample only has a single state field, components can have an arbitrary number of fields. The `setState()` method accepts sparsely populated objects; you are not required to provide values for all state fields. Any new values that are provided will be merged on top of the current state.

The following version of our greeting component has been modified to use a numeric, enum-based counter to track how engaged the end user is. After three clicks, the greeting component ends the conversation.

oj-greet/with-state/with-state.tsx:

```
import { h, Component } from "preact";
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";

type Props = {
  preferredGreeting?: string;
};

enum EngagementLevel {
  Interested,
  Bored,
  Impatient,
  Done,
}

type State = {
  engagement: EngagementLevel;
};

/**
 * @ojmetadata pack "oj-greet"
 */
@customElement("oj-greet-with-state")
export class GreetWithState extends Component<ExtendGlobalProps<Props>,
State> {
  state = {
    engagement: EngagementLevel.Interested,
  };

  render() {
    const greeting = this.getGreeting();

    return (
      <div onClick={this._handleClick}>
        <p>{greeting}, World!</p>
      </div>
    );
  }

  private getGreeting() {
    let greeting;

    switch (this.state.engagement) {
      case EngagementLevel.Bored:
        greeting = "Okay";
        break;
      case EngagementLevel.Impatient:
        greeting = "Whatever";
        break;
      case EngagementLevel.Done:
        greeting = "Later";
```

```
        break;
      default:
        greeting = this.props.preferredGreeting;
        break;
    }

    return greeting;
  }

  private _handleClick = (event: MouseEvent) => {
    this.setState((state: Readonly<State>) => {
      // Once we have reached the Done state, we return null
      // to indicate that no state update is needed.
      return state.engagement === EngagementLevel.Done
        ? null
        : { engagement: state.engagement + 1 };
    });
  };

  static defaultProps: Partial<Props> = {
    preferredGreeting: "Hello",
  };
}
```

Understand the State Mechanism

One potential problem with the object-based form of `setState()` arises when the new value for a state field is derived from the previous value. Given the asynchronous nature of this method, simply inspecting `this.state` may not be sufficient to determine what the next value should be. If there is an outstanding call to `setState()` that has not yet been fully processed, `this.state` might not reflect the pending update.

To better support cases where a state field's next value is dependent on the previous value, `setState()` supports a callback form. Rather than passing in an object representing the new state, callers pass in a function that takes two arguments: the current state and properties. This callback function can inspect the state and props and return one of the following values:

- A sparsely populated object representing any state updates to apply
  or:
- `null`, if no state updates are required.

When multiple calls to `setState()` are issued, the state updates are chained. That is, the results of one call (whether object or callback form) are fed into the subsequent callback. This ensures the callback always sees the most up to date values, and that it can use this information to correctly produce the next value.

Reference Child VComponents by Value

You can reference a VComponent child component from within JSX in two ways:

```
// Use intrinsic element name
function Parent() {
   return <some-comp />;
}
```

```
// Use value-based element to reference
// the VComponent class or function value
function Parent() {
   return <SomeComp />
}
```

We recommend that, whenever possible, you use the value-based element because:

1. It is slightly more efficient as we are able to do more rendering in virtual DOM even before the VComponent's DOM element is created.
2. It provides a more React/Preact-centric approach to use certain APIs, such as slots and listeners. (More on this below.)
3. When working within a single project (that is, where both the VComponent and consuming code is in the same project), you have access to the VComponent class type information directly in your project source. You are not dependent on a build to produce a type definition for the class.

Elaborating on point 2, when you reference a VComponent through its intrinsic element name, you are limited to using this syntax for slots, as in the following example:

```
function Parent() {
 <some-comp>
  // This is a plain slot:
  <img src="foo.png" slot="startIcon" />

  // This is a template slot:
  <template slot="itemTemplate render={ renderItem } />
 </some-comp>
}
```

With a value-based element approach, you can achieve the same outcome more concisely, and in a form that is more familiar to app developers with a React/Preact background:

```
function Parent() {
   <SomeComp startIcon={ <img src="foo.png" } itemTemplate={ renderItem }>
}
```

References to event listeners when using the value-based element form also aligns better with what a React/Preact developer would expect. For example, for intrinsic elements, we need to follow Preact's custom event naming conventions. So, for an event name of someCustomEvent, we end up with this listener prop:

```
function Parent() {
 <some-comp onsomeCustomEvent={ handleSomeCustomEvent }>
}
```

When referencing the value-based element, this is:

```
function Parent() {
 <SomeComp onSomeCustomEvent={ handleSomeCustomEvent }>
}
```

To conclude, use the value-based element whenever possible. For cases where you need to interact directly with the DOM element, use the intrinsic element form and then obtain a reference to it, as in this example:

```
function Parent() {
    const someRef = useRef(null);
    return <some-comp ref={ someRef } />;
}
```

4

Work with Oracle JET VComponent-based Web Components

Oracle JET VComponent-based Web Components are reusable pieces of user interface code that you can embed as custom HTML elements. Web Components can contain Oracle JET components, other Web Components, HTML, JavaScript, and CSS. You can create your own Web Component or add one to your page.

Create Web Components

Oracle JET supports a variety of custom Web Component component types. You can create standalone Web Components or you can create sets of Web Components that you intend to be used together and you can then assemble those in a JET Pack, or pack component type. You can enhance JET Packs by creating resource components when you have re-usable libraries of assets that are themselves not specifically UI components. And, if you need to reference third-party code in a standalone component, you can create reference components to define pointers to that code.

Create Standalone Web Components

Use the Oracle JET command-line interface (CLI) to create an Oracle JET Web Component template that you can populate with content. If you're not using the CLI, you can add the Web Component files and folders manually to your Oracle JET application.

The procedure below lists the high-level steps to create a Web Component.

Before you begin:

- Familiarize yourself with the list of reserved names for a Web Component that are not available for use, see valid custom element name
- Familiarize yourself with the list of existing Web Component properties, events, and methods, see HTMLElement properties, event listeners, and methods
- Familiarize yourself with the list of global attributes and events, see Global attributes

To create a Web Component:

1. Determine a name for your Web Component.

   The Web Component specification restricts custom element names as follows:

   - Names must contain a hyphen.
   - Names must start with a lowercase ASCII letter.
   - Names must not contain any uppercase ASCII letters.
   - Names should use a unique prefix to reduce the risk of a naming collision with other components.

     A good pattern is to use your organization's name as the first segment of the component name, for example, *org-component-name*. Names must not start with the prefix `oj-` or `ns-`, which correspond to the root of the reserved `oj` and `ns` namespaces.

- Names must not be any of the reserved names. Oracle JET also reserves the `oj` and the `ns` namespace and prefixes.

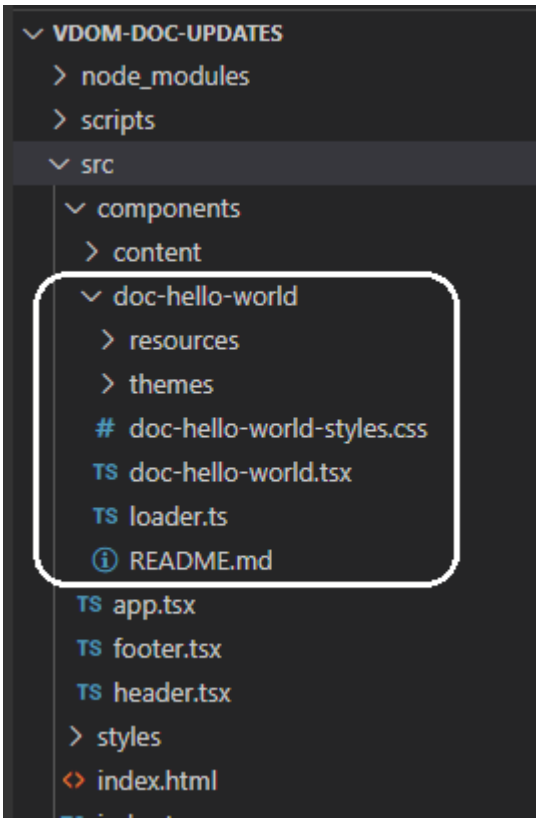
2. Determine where to place your Web Component, using one of the following options.
   - Add the Web Component to an existing Oracle JET application that you created with the Oracle JET CLI.

     If you use this method, you'll use the CLI to create a Web Component template that contains the folders and files you'll need to store the Web Component's content.
   - Manually add the Web Component to an existing Oracle JET application that doesn't use the Oracle JET CLI.

     If you use this method, you'll create the folders and files manually to store the Web Component's content.
3. Depending upon the choice you made in the previous step, perform one of the following tasks to create the Web Component.
   - If you used the Oracle JET CLI to create an application, then in the application's top-level directory, enter the following command at a terminal prompt to generate the Web Component template:

     ```
     ojet create component component-name
     ```

     For example, enter `ojet create component doc-hello-world` to create a Web Component named `doc-hello-world`. The command adds `doc-hello-world` to the application's `src\components\` folder with files containing stub content for the Web Component.



The value of the `components` property in the `oracleconfig.json` file determines the folder location where the Oracle CLI generates the Web Component. The default value is `components`, but you can change it to another value.

- If you're not using the Oracle JET CLI, create a folder in your application's `src` folder, and add folders containing the name of each Web Component you will create.

4. If you're not using the Oracle JET CLI, create a `loader.js` RequireJS module and place it in the Web Component's root folder.

   The `loader.js` module defines the Web Component dependencies and registers the component's class name, `DocHelloWorld` in this example.

   ```
   export { DocHelloWorld } from "./doc-hello-world";
   ```

5. Configure any custom styling that your Web Component will use.

   - If you only have a few styles, add them to *web–component-name*`-styles.css` file in the Web Component's root folder, creating the file if needed.

     For example, the `DocHelloWorld` Web Component defines styles for the component's display, width, and height.

     ```
     /* This prevents the flash of unstyled content before the Web
        Component properties have been setup. */
     doc-hello-world:not(.oj-complete) {
       visibility: hidden;
     }

     doc-hello-world {
        min-height: 50px;
        width: 50px;
     }
     ```

   - If you used the Oracle JET tooling to create your application and want to use Sass to generate your CSS:

     a. If needed, at a terminal prompt in your application's top level directory, type the following command to add node-sass to your application: `ojet add sass`.

     b. Create *web–component-name*`-styles.scss` and place it in the Web Component's top level folder.

     c. Edit *web–component-name*`-styles.scss` with any valid SCSS syntax and save the file.

        In this example, a variable defines the demo card size:

        ```
        $doc-hello-world-size: 200px;

        /* This prevents the flash of unstyled content before the
           Web Component properties have been setup. */
        doc-hello-world:not(.oj-complete) {
          visibility: hidden;
        }

        doc-hello-world {
          width: $doc-hello-world-size;
        ```

```
    min-height: $doc-hello-world-size;
  }
```

6. If you want to add documentation for your Web Component, add content to `README.md` in your Web Component's root folder, creating the file if needed.

   Your `README.md` file should include an overview of your component with well-formatted examples. Include any additional information that you want to provide to your component's consumers. The recommended standard for `README` file format is markdown.

Create JET Packs

Create JET Packs to simplify project management for consumers who might pick up a component that is related to one or more components. You may require specific versions of the referenced components for individual JET Packs.

Fundamentally, the JET Pack is a library of related Web Components that does not directly include those assets, but is as an index to a particular versioned stripe of components.

> **Note**
>
> Note there is one exception to the pack as a reference mechanism for related components. A pack might include one or more RequireJS bundle files which package up optimized forms of the component set into a small number of physical downloads. This, however, is always in addition to the actual components being available as independent entities in Oracle Component Exchange.

The components referenced by the JET Pack are intended to be used together and their usage is restricted by individual component version. Thus, the JET Pack that you create will tie very specific versions of each component into a relationship with very specific, fixed versions of the other components in the same set. Thus, a JET Pack itself has a "version stripe" which determines the specific components that users import into their apps. Since the version number of individual components may vary, the JET Pack guarantees the consumer associates their app with the version of the pack as a whole, and not with the individual components contained by the pack.

1. Create the JET Pack using the JET tooling from the root folder of your app.

   ```
   ojet create pack my-pack
   ```

   Consider the pack name carefully, as the name will determine the prefix to any components within that pack.

   The tooling adds the folder structure with the template files that you will need to modify:

   ```
   /(working folder)
     /src
       /components
         /my-pack
   ```

2. Create the components that you want to bundle with the JET Pack by using the JET tooling from the root folder of your app. The component name that you specify must be unique within the pack.

   ```
   ojet create component my-widget-1 --pack=my-pack
   ```

   The tooling nests the component folder `/my-widget-1` under the `my-pack` root folder and the new component files resemble those created for a standalone Web Component.

   ```
   /(working folder)
     /src
       /components
         /my-pack
             component.json
             /my-widget-1
             |   loader.ts
           |   my-widget-1-styles.css
           |   my-widget-1.tsx
           |   README.md
         +---resources
           |   /---nls
           |     /---root
           |           my-widget-1-strings.ts
         /---themes
   ```

   `loader.ts` includes a one-line entry that exports the VComponent module class of the new component, as in the following example:

   ```
   export { MyWidget1 } from "./my-widget-1";
   ```

3. Optionally, for any Resource components that you created, as described in Create Resource Components for JET Packs, add the component's working folder with its own `component.json` file to the pack file structure.

   The tooling nests the component folder `/my-widget-1` under the `my-pack` root folder and the new component files resemble those created for a standalone Web Component.

   ```
   /(working folder)
     /src
       /components
           /my-pack
             component.json
             /my-widget-1
               /resources
                 /nls
                   /root
                     my-widget-1-strings.js
               component.json
               loader.ts
               README.md
               my-widget-1.tsx
               my-widget-1-styles.css
             /my-resource-component-1
               component.json
               /converters
   ```

```
    file1.js
    ...
  /resources
    /nls
     /root
      string-file1.js
  /validators
    file1.js
    ...
```

4. Optionally, generate any required bundled for desired components of the pack. Refer to RequireJS documentation for details at the https://requirejs.org web site.

   > **Tip**
   >
   > You can use RequireJS to create optimized bundles of the pack components, so that rather than each component being downloaded separately by the consuming app at runtime, instead a single JavaScript file can be downloaded that contains multiple components. It's a good idea to use this facility if you have sets of components that are almost always used together. A pack can have any number of bundles (or none at all) in order to group the available components as required. Be aware that not every component in the pack has to be included in one of the bundles and that each component can only be part of one bundle.

5. Use a text editor to modify the `component.json` file in the pack folder root similar to the following sample, to identify pack dependencies and optional bundles. Added components must be associated by their full name and a specific version.

```
{
    "name": "my-pack",
    "version": "1.0.0",
    "type": "pack",
    "displayName": "My JET Pack",
    "description": "An example JET Pack",
    "dependencies": {
      "my-pack-my-widget-1":"1.0.0",
      ...
    },
    "bundles":{
        "my-pack/my-bundle":[
          "my-pack/my-bundle-file1/loader",
          ...
        ]
      },
    "extension": {
        "catalog": {
            "coverImage": "coverimage.png"
        }
     }
  }
}
```

Your pack component's `component.json` file must contain the following unique definitions:

- **name** is the name of the JET Pack has to be unique, and should be defined with the namespace relevant to your group. This name will be prepended to create the full name of individual components of the pack.
- **version** defines the exact version number of the pack, not a SemVer range.

> **Note**
>
> Changes in version number with a given release of a pack should reflect the most significant change in the pack contents. For example, if the pack contained two components and as part of a release one of these had a Patch level change and the other a Major version change then the pack version number change should also be a Major version change. There is no requirement for the actual version number of the pack to match the version number(s) of any of it's referenced components.

- **type** must be set to `pack`.
- **displayName** is the name of the pack component that you want displayed in in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain how the pack is intended to be used.
- **dependencies** defines the set of components that make up the pack, specified by the component full name (a concatenation of pack name and component name). Note that exact version numbers are used here, not SemVer ranges. It's important that you manage revisions of dependency version numbers to reflect changes to the referenced component's version and also to specify part of the path to reach the components within the pack.

  If you want to include all components in the JET Pack directory, use a token, `"@dependencies@"`, as the value for `dependencies` rather than defining individual entries for all the components in the pack. The following snippet illustrates how you use this token in your `component.json` file:

```
{
  "name": "my-pack",
  "version": "1.0.0",
  "type": "pack",
  "displayName": "My JET Pack",
  "description": "An example JET Pack",
  "dependencies": "@dependencies@"
}
```

- **bundles** defines the available bundles (optional) and the contents of each. Note how both the bundle name and the contents of that bundle are defined with the pack name prefix as this is the RequireJS path that is needed to map those artifacts.
- **catalog** defines the working metadata for Oracle Component Exchange, including a cover image in this case.

6. Optionally, create a readme file in the root of your working folder. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).
7. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. The file name can be the same as the name attribute in the `component.json` file.

8. Use the JET tooling to create a zip archive of the JET Pack working folder when you want to upload the component to Oracle Component Exchange, as described in Package Web Components.
9. Support consuming the JET Pack in Oracle Visual Builder projects by uploading the component to Oracle Component Exchange.

Create Resource Components for JET Packs

Create a resource component when you want to reuse assets across web components that you assemble into JET Packs. The resource component can be reused by multiple JET Packs.

When dealing with complex sets of components you may find that it makes sense to share certain assets between multiple components. In such cases, the components can all be included into a single JET Pack and then a resource component can be added to the pack in order to hold the shared assets. There is no constraint on what can be stored in a pack, typically it may expose shared JavaScript, CSS, and JSON files and images. Note that third party libraries should generally be referenced from a reference component and should not be included into a resource component.

You don't need any tools to create the resource component. You will need to create a folder in a convenient location. This folder will ultimately be zipped to create the distributable resource component. Internally this folder can then hold any content in any structure that you desire.

To create a resource component:

1. If you have not already done so, create a JET Pack using the following command from the root folder of your app to contain the resource component(s):

   ```
   ojet create pack my-resource-pack
   ```

2. Still in the root folder of your app, create the resource component in the JET Pack:

   ```
   ojet create component my-resource-comp --type=resource --pack=my-resource-pack
   ```

   The tooling adds the folder structure with a single template `component.json` file and an index file.

   ```
   /root folder
     /src
       /components
        /my-resource-pack
          /my-resource-comp
            component.json
   ```

3. Populate the created folder (`my-resource-comp`, in our example) with the desired content. You can add content in any structure desired, with the exception of NLS content for translation bundles. In the case of NLS content, preserve the typical JET folder structure; this is important if your resource component is going to include such bundles.

   ```
   /(my-resource-folder)
     /converters
       phoneConverter.js
       phoneConverterFactory.js
     /resources
       /nls
         /root
          oj-ext-strings.js
       /phone
   ```

```
      countryCodes.json
  /validators
    emailValidator.js
    emailValidatorFactory.js
    phoneValidator.js
    phoneValidatorFactory.js
    urlValidator.js
    urlValidatorFactory.js
```

In this sample notice how the `/resources/nls` folder structure for translation bundles is preserved according to the folder structured of the app generated by JET tooling.

4. Use a text editor to update the `component.json` file in the folder root similar to the following sample, which defines the resource `my-resource-comp` for the JET Pack `my-resource-pack`.

```
{
  "name": "my-resource-comp",
  "pack": "my-resource-pack",
  "displayName": "Oracle Jet Extended Utilities",
  "description": "A set of reusable utility classes used by the Oracle JET
extended
                  component set and available for general use.  Includes
various
                  reusable validators",
  "license": "https://opensource.org/licenses/UPL",
  "type": "resource",
  "version": "2.0.2",
  "jetVersion": ">=8.0.0 <10.1.0",
  "publicModules": [
    "validators/emailValidatorFactory",
    "validators/urlValidatorFactory"
  ],
  "extension":{
    "catalog": {
      "category": "Resources",
        "coverImage": "cca-resource-folder.svg"
    }
  }
}
```

Your resource component's `component.json` file must contain the following unique definitions:

- **name** is the name of the resource component has to be unique, and should be defined with the namespace relevant to your group.
- **pack** is the name of the JET Pack containing the resource component.
- **displayName** is the name of the resource component as displayed in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain the available assets provided by the component.
- **type** must be set to `resource`.

- **version** defines the semantic version (SemVer) of the resource component as a whole. It's important that you manage revisions of this version number to inform consumers of the compatibility of a given change.

  > **Note**
  >
  > Changes to the resource component version should roll up all of the changes within the resource component, which might not be restricted to changes only in `.js` files. A change to a CSS selector defined in a shared `.css` file can trigger a major version change when it forces consumers to make changes to their downstream uses of that selector.

- **jetVersion** defines the supported Oracle JET version range using SemVer notation. This is *optional* and depends on the nature of what you include into the resource component. If the component contains JavaScript code and any of that code makes reference to Oracle JET APIs, then you really should include a JET version range in that case.
- **publicModules** lists entry points within the resource component that you consider as being public and intend to be consumed by any component that depends on this component. Any API not listed in the array is considered to be pack-private and therefore can only be used by components within the same pack namespace, but may not be used externally.
- **catalog** defines the working metadata for Oracle Component Exchange, including a cover image in this case.

5. Optionally, create a readme file in the root of your working folder. A readme can be used to document the assets of the resource. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).

   > **Tip**
   >
   > Take care to explain the state of the assets. For example, you might choose to include utility classes in the resource component that are deemed public and can safely be used by external consumers (for example, code outside of the JET Pack that the component belongs to). However, you may want to document other assets as private to the pack itself.

6. Optionally, create a change log file in the root of your working folder. The change log can detail significant changes to the pack over time and is strongly recommended. This should be defined as a text file called `CHANGELOG.txt` (or `CHANGELOG.md` when using markdown format).
7. Optionally, include a License file in the root of your working folder.
8. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. Using the third party logo can be helpful here to identify the usage. The file name can be the same as the name attribute in the `component.json` file.
9. Create a zip archive of the working folder when you want to upload the component to Oracle Component Exchange. Oracle recommends using the format `<fullName>-<version>.zip` for the archive file name. For example, `my-resource-pack-my-resource-comp-2.0.2.zip`.

For information about using the resource component in a JET Pack, see Create JET Packs.

Create Reference Components for Web Components

Create a reference component when you need to obtain a pointer to third-party libraries for use by Web Components.

Sometimes your JET Web Components need to use third party libraries to function and although it is possible to embed such libraries within the component itself, or within a resource component, it generally better to reference a shared copy of the library by defining a reference component.

Create the Reference Component

You don't need any tools to create the reference component. You will need to create a folder in a convenient location where you will define metadata for the reference component in the `component.json` file. This folder will ultimately be zipped to create the distributable reference component.

Reference components are generally standalone, so the `component.json` file you create must not be contained within a JET Pack.

To create a reference component:

1. Create the working folder and use a text editor to create a `component.json` file in the folder root similar to the following sample, which references the moment.js library.

```
{
  "name": "oj-ref-moment",
  "displayName": "Moment library",
  "description": "Supplies reference information for moment.js used to
parse,
                  validate, manipulate, and display dates and times in
JavaScript",
  "license": "https://opensource.org/licenses/MIT",
  "type": "reference",
  "package":"moment",
  "version": "2.24.0",
  "paths": {
    "npm": {
      "debug": "moment",
      "min": "min/moment.min"
    },
    "cdn": {
      "debug": "https://static.oracle.com/cdn/jet/packs/3rdparty/moment/
2.24.0/moment.min",
      "min": "https://static.oracle.com/cdn/jet/packs/3rdparty/moment/
2.24.0/moment.min"
    }
  },
  "extension": {
    "catalog": {
      "category": "Third Party",
      "tags": [
        "momentjs"
      ],
      "coverImage": "coverImage.png"
    }
```

```
  }
}
```

Your reference component's `component.json` file must contain the following unique definitions:

- **name** is the name of the reference component has to be unique, and should be defined with the namespace relevant to your group.
- **displayName** is the name of the resource component as displayed in Oracle Component Exchange. Set this to something readable but not too long.
- **description** is the description that you want displayed in Oracle Component Exchange. For example, use this to explain the function of the third party library.
- **license** comes from the third party library itself and must be specified.
- **type** must be set to `reference`.
- **package** defines the npm package name for the library. This will also be used as the name of the associated RequireJS path that will point to the library and so will be used by components that depend on this reference.
- **version** should reflect the version of the third party library that this reference component defines. If you need to be able to reference multiple versions of a given library then you will need multiple versions of the reference component in order to map each one.
- **paths** defines the CDN locations for this library. See below for more information about getting access to the Oracle CDN.
- **min** points to the optimal version of the library to consume. The debug path can point to a debug version or just the min version as here.
- **catalog** defines the working metadata for Oracle Component Exchange including a cover image in this case.

2. Optionally, create readme file in the root of your working folder. A readme can be used to point at the third party component web site for reference. This should be defined as a plain text file called `README.txt` (or `README.md` when using markdown format).
3. Optionally, create a cover image in the root of your working folder to display the component on Oracle Exchange. Using the third party logo can be helpful here to identify the usage. The file name can be the same as the name attribute in the `component.json` file.
4. Create a zip archive of the working folder when you want to upload the component to Oracle Component Exchange. Oracle recommends using the format `<fullName>-<version>.zip` for the archive file name. For example, `oj-ref-moment-2.24.0.zip`.
5. Support consuming the reference component in Oracle Visual Builder projects by uploading the component to a CDN. See below for more details.

Consume the Reference Component

When your Web Components need access to the third party library defined in one of these reference components, you use the dependency attribute metadata in the `component.json` to point to either an explicit version of the reference component or you can specify a semantic range. Here's a simple example of a component that consumes two such reference components at specific versions:

```
{
    "name":"calendar",
    "pack":"oj-sample",
```

```
"displayName": "JET Calendar",
"description": "FullCalendar wrapper with Accessibility added.",
"version": "1.0.2",
"jetVersion": "^9.0.0",
"dependencies": {
    "oj-ref-moment":"2.24.0",
    "oj-ref-fullcalendar":"3.9.0"
},

...
```

When the above component is added to an Oracle JET or Oracle Visual Builder project this dependency information will be used to create the correct RequireJS paths for the third party libraries pointed to be the reference component.

Alternatively, when you install a Web Component that depends on a reference component and you use Oracle JET CLI, the tooling will automatically do an npm install for you so that the libraries are local. However, with the same component used in Oracle Visual Builder, a CDN location must be used and therefore the reference component must exist on the CDN in order to be used in Visual Builder.

Add Web Components to Your Page

To use a VComponent-based Web Component, you import the loader module that provides the entry point to the component in the TSX page where you will use the component, unless you import the component from a mono-pack JET Pack.

Those of you who previously built Web Components using the Composite Component Architecture will be familiar with the use of the loader module to import the Web Component. VComponent-based Web Components also use this convention of a `loader` module serving as the main entry point to the Web Component. It allows the Oracle JET CLI, Visual Builder, and the Component Exchange to work with VComponent-based Web Components.

If you import the Web Component into a VDOM app, you can choose between importing the Web Component through the use of the component class name or the custom element name for the Web Component. If you import the VComponent-based Web Component into an MVVM JET app, you must use the custom element syntax. If importing a component from a JET Pack, you'll also need to identify the JET Pack in the import statement, but the import statement differs slightly depending on whether you import from a conventional JET Pack or a mono-pack JET Pack. In the case of an import of a component from a mono-pack, you don't need to include the loader module in the import statement as components created within this type of pack are automatically loaderless.

The following commented example illustrates the syntax to use for each option.

```
import { h } from "preact";
//// Import standalone components.
//   Import as a custom element.
import "doc-hello-world/loader"
//   Import as a component class.
import { DocHelloWorld } from "doc-hello-world/loader"
//// Import components from a conventional JET Pack using loader module
//   Import as a custom element.
import "my-component-pack/my-widget-1/loader";
//   Import as a component class.
import { MyWidget2 } from "my-component-pack/my-widget-2/loader";
```

```
// Import components from a Mono-Pack JET Pack (Note: No reference to
loader.ts)
//   Import as a custom element.
import "my-mono-pack/my-widget-3";
//   Import as a component class.
import { MyWidget4 } from "my-mono-pack/my-widget-4";

export function Content() {
  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      {/* Standalone component's custom element */}
      <doc-hello-world></doc-hello-world>
      {/* Standalone component's component class */}
      <DocHelloWorld />
      {/* JET Pack component's custom element */}
      <my-component-pack-my-widget-1></my-component-pack-my-widget-1>
      {/* JET Pack component's component class */}
      <MyWidget2 />
      {/* JET Mono-Pack component's custom element */}
      <my-mono-pack-my-widget-3></my-mono-pack-my-widget-3>
      {/* JET Mono-Pack component's component class */}
      <MyWidget4 />
    </div>
  );
};
```

Generate API Documentation for VComponent-based Web Components

The Oracle JET CLI includes a command (`ojet add docgen`) that you can use to assist with the generation of API documentation for the VComponent-based web components (VComponent) that you develop.

When you run the command from the root of your project, the JSDoc NPM package is installed and an `apidoc_template` directory is added to the `src` directory of your project. The `apidoc_template` directory contains HTML files (footer, header, and main) that you can customize with appropriate titles, subtitles, and footer information, such as copyright information, for the API reference documentation that you'll subsequently generate for your VComponent(s). The command also adds an `"enableDocGen": true` entry to the `oraclejetconfig.json` file of the Oracle JET app. When `true`, API doc is generated. When `false`, no API doc is generated for the VComponents.

You write comments in the source file of your VComponent, as in the following example:

```
import { ExtendGlobalProps, registerCustomElement } from "ojs/ojvcomponent";
. . .

type Props = Readonly<{
  message?: string;
  address?: string;
}>;

/**
 *
```

```
 * @ojmetadata version "1.0.0"
 * @ojmetadata displayName "A user friendly, translatable name of the
component"
 * @ojmetadata description "<p>Write a description here.</p>
                            <p>Use HTML tags to put in new paragraphs</p>
                            <ul>
                                <li>Bullet list item 1</li>
                                <li>Bullet list item 2</li></ul>
        * <p>Everything before the closing quote is rendered</p>
 * "
 *
 */


function StandaloneVcompFuncImpl({ address = "Redwood shores",
                         message = "Hello from  standalone-vcomp-func" }:
Props) {
  return (
    <div>
    . . .
    </div>
  );
}
```

Once you have completed documenting your VComponent's API in the source file, you run the `build` command for your component or the JET Pack, if the component is part of a JET pack (`ojet build component` *`component-name`* or `ojet build component` *`jet-pack-name`*) to generate API reference doc in the `appRootDir/web/components/`*`component-or-pack-name/vcomponent-version`*`/docs` directory.

The following `/docs` directory listing shows the files that the Oracle JET CLI generates for a standalone VComponent. You can't generate the API documentation by building the Oracle JET app that contains the component. You have to build the individual VComponent or the JET Pack that contains VComponents. Note too that you can't generate API doc for CCA-based web components using the Oracle JET CLI `ojet add docgen` command.

```
appRootDir/web/components/standalone-vcomp-func/1.0.0/docs

|   index.html
|   jsDocMd.json
|   standalone-vcomp-func.html
|   standalone.StandaloneVcompFunc.html
|
+---scripts
|   |   deprecated.js
|   |
|   \---prettify
|           Apache-License-2.0.txt
|           lang-css.js
|           prettify.js
|
\---styles
    |   jsdoc-default.css
    |   prettify-jsdoc.css
    |   prettify-tomorrow.css
    |
```

```
\---images
        bookmark.png
        linesarrowup.png
        linesarrowup_blue.png
        linesarrowup_hov.png
        linesarrowup_white.png
        oracle_logo_sm.png
```

One final thing to note is that if you want to include an alternative logo and/or CSS styles to change the appearance of the generated API doc, you update the content in the following directory `appRootDir/node_modules/@oracle/oraclejet/dist/jsdoc/static/styles/`.

Build Web Components

You can build your Oracle JET Web Component to optimize the files and to generate a minified folder of the component that can be shared with the consumers.

When your Web Component is configured and is ready to be used in different apps, you can build the Web Components of the type: standalone Web Component, JET Pack, and Resource component. Building these components using JET tooling generates a minified content with the optimized component files. This minified version of the component can be easily shared with the consumers for use. For example, you would build the component before publishing it to Oracle Component Exchange. To build the Web Component, use the following command from the root folder of the JET app containing the component:

```
ojet build component my-web-component-name
```

For example, if your Web Component name is `hello-world`, use the following command:

```
ojet build component hello-world
```

For a JET Pack, specify the pack name.

```
ojet build component my-pack-name
```

Note that the building individual components within the pack is not supported, and the whole pack must be built at once.

This command creates a `/min` folder in the `web/components/hello-world/x.x.x/` directory of your Oracle JET web app, where `x.x.x` is the version number of the component. The `/min` folder contains the minified (release) version of your Web Component files.

Reference component do not require minification or bundling and therefore do not need to be built.

When you build Web Components:

- If your JET app contains more than one component, you can build the containing JET app to build and optimize all components together. The `build component` command with the component name provides the capability to build a single component.
- You can optionally use the `--release` flag with the build command, but it is not necessary since the `build` command generates both the debug and minified version of the component.

- You can optionally use the `--optimize=none` flags with the `build` command when you want to generate compiled output that is more readable and suitable for debugging. The component's `loader.js` file will contain the minified app source, but content readability is improved, as line breaks and white space will be preserved from the original source.

Package Web Components

You can create a sharable zip file archive of the minified Oracle JET Web Component from the Command-Line Interface.

When you want to share Web Components with other developers, you can create an archive file of the generated output contained in the `components` subfolder of the app's `/web`. After you build a standalone Web Component or a Resource component, you use the JET tooling to run the `package` command and create a zip file that contains the Web Component compiled and minified source.

```
ojet package component my-web-component-name
```

Similarly, in the case of JET packs, you cannot create a zip file directly from the file system. It is necessary to use the JET tooling to package JET packs because the output under the `/jet-composites/<packName>` subfolder contains nested component folders and the tooling ensures that each component has its own zip file.

```
ojet package pack my-JET-Pack-name
```

The `package` command packages the component's minified source from the `/web/components` directory and makes it available as a zip file in a `/dist` folder at the root of the containing app. This zip file will contain both the specified component and a minified version of that component in a `/min` subfolder.

Reference components do not require minification or bundling and therefore do not need to be built. You can archive the Reference component by creating a simple zip archive of the component's folder.

The zip archive of the packaged component is suitable to share, for example, on Oracle Component Exchange, as described in Publish Web Components to Oracle Component Exchange. To help organize components that you want to publish, the JET tooling appends the value of the `version` property from the `component.json` file for the JET pack and the individual components to the generated zip in the `dist` folder. Assume, for example, that you have a component pack, `my-component-pack`, that has a `version` value of `1.0.0` and the indiviudal components (`my-widget-1`, and so on) within the pack also have version values of `1.0.0`, then the zip file names for the generated files will be as follows:

```
appRootDir/dist/
my-web-component-name_1-0-0.zip
my-component-pack_1-0-0.zip
my-component-pack-my-widget-1_1-0-0.zip
my-component-pack-my-widget-2_1-0-0.zip
my-component-pack-my-widget-3_1-0-0.zip
```

You can also generate an archive file when you want to upload the component to a CDN. In the CDN case, additional steps are required before you can share the component, as described in Upload and Consume Web Components on a CDN.

5

Use Oracle JET Components and Data Providers

Review the following recommendations to make effective use of Oracle JET components and associated APIs, such as data providers, so that the app that you develop is performant and provides an optimal user experience.

Access Subproperties of Oracle JET Component Properties

JSX does not support the dot notation that allows you to access a subproperty of a component property. You cannot, for example, use the following syntax to access the `max` or `count-by` subproperties of the Input Text element's `length` property.

```
<oj-input-text
  length.max={3}
  length.count-by="codeUnit"
</oj-input-text>
```

To access these subproperties using JSX, first access the element's top-level property and set values for the subproperties you want to specify. For example, for the `countBy` and `max` subproperties of the Oracle JET `oj-input-text` element's `length` property, import the component props that the Input Text element uses. Then define a `length` object, based on the type `InputTextProps["length"]`, and assign it values for the `countBy` and `max` subproperties. Finally, pass the `length` object to the `oj-input-text` element as the value of its `length` property.

```
import {ComponentProps } from "preact";

type InputTextProps = ComponentProps<"oj-input-text">;

const length: InputTextProps["length"] = {
  countBy: "codeUnit",
  max: 3
};

function Parent() {
  return (
    <oj-input-text length={ length } />
  )
}
```

Mutate Properties on Oracle JET Custom Element Events

Unlike typical Preact components, mutating properties on Oracle JET custom elements invokes property-changed callbacks. As a result, you can end up with unexpected behavior, such as infinite loops, if you:

1. Have a property-changed callback
2. The property-changed callback triggers a state update
3. The state update creates a new property value (for example, copies values into a new array)
4. The new property value is routed back into the same property

Typically, Preact (and React components) invoke callbacks in response to user interaction, and not in response to a new property value being passed in by the parent component. You can simulate the Preact component-type behavior when you use Oracle JET custom elements if you check the value of the `event.detail.updatedFrom` field to determine if the property change is due to user interaction (`internal`) instead of your app programmatically mutating the property value (`external`). For example, the following event callback is only invoked in response to user interaction:

```
. . .
const onSelection = (event) => {
      if (event.detail.updatedFrom === 'internal') {
        const selectedValues = event.detail.value;
        setSelectedOptions([selectedValues]);
      }
    };
. . .
```

Avoid Repeated Data Provider Creation

Avoid creating a new data provider each time a VComponent renders.

For instance, avoid doing this in a VComponent with an `oj-list-view` component, as it recreates the `MutableArrayDataProvider` instance every time the VComponent renders:

```
<oj-list-view
  data={new MutableArrayDataProvider....}
  . . . >
</oj-list-view>
```

Instead, consider using Preact's useMemo hook to ensure the data provider instance is only recreated when the data in the provider changes. The following example shows how to include the useMemo hook to prevent the data provider for an `oj-list-view` component from being recreated, even if the VComponent with this code re-renders.

```
import MutableArrayDataProvider = require("ojs/ojmutablearraydataprovider");
import { Task, renderTask } from "./data/task-data";
import { useMemo } from "preact/hooks";
import "ojs/ojlistview";
import { ojListView } from "ojs/ojlistview";

type Props = {
  tasks: Array<Task>;
}

export function ListViewMemo({ tasks }: Props) {
  const dataProvider = useMemo(() => {
      return new MutableArrayDataProvider(
```

```
        tasks, {
          keyAttributes: "taskId"
        }
      );
    },
    [ tasks ]
  );

  return (
    <oj-list-view data={dataProvider} class="demo-list-view">
      <template slot="itemTemplate" render={renderTask} />
    </oj-list-view>
  )
}
```

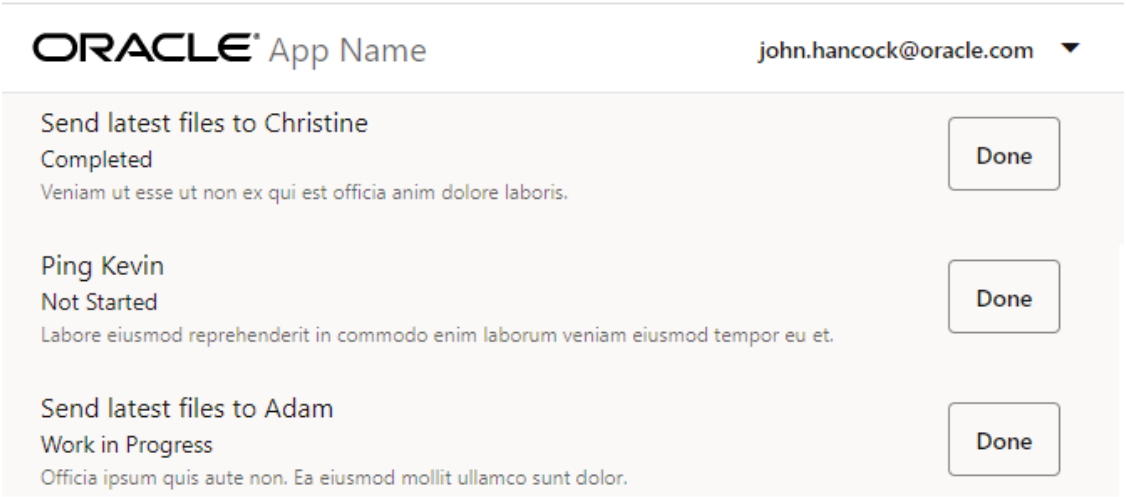
This recommendation is because a VComponent can re-render for various reasons, but as long as the data provider's data remains unchanged, there's no need to recreate the instance, which can impact app performance and usability due to unnecessary rendering and collection component flashes and scroll position loss.

Avoid Data Provider Re-creation When Data Changes

Previously, we described how the useMemo hook avoids re-creating a data provider unless data changes. When data does change, we still end up re-creating the data provider instance, and this can result in collection components, such as list view, re-rendering all data displayed by the component and losing scroll position.

Here, we'll try to illustrate how you can opitimize the user experience by implementing fine-grained updates and maintaining scroll position for collection components, even if the data referenced by the data provider changes. To accomplish this, the data provider uses a `MutableArrayDataProvider` instance. With a `MutableArrayDataProvider` instance, you can mutate an existing instance by setting a new array value into the `MutableArrayDataProvider`'s data field. The collection component is notified of the specific change (create, update, or delete) that occurs, which allows it to make a fine-grained update and maintain scroll position.

In the following example, an app displays a list of tasks in a list view component and a Done button that allows a user to remove a completed task. These components render in the Content component of a virtual DOM app created with the basic template (`appRootDir/src/components/content/index.tsx`).



To implement fine-grained updates and maintain scroll position for the `oj-list-view` component, we store the list view's `MutableArrayDataProvider` in local state using Preact's useState hook and never re-create it, and we also use Preact's useEffect hook to update the data field of the `MutableArrayDataProvider` when a change to the list of tasks is detected. The user experience is that a click on Done for an item in the tasks list removes the item (with removal animation). No refresh of the `oj-list-view` component or scroll position loss occurs.

```
import "ojs/ojlistview";
import { ojListView } from "ojs/ojlistview";
import MutableArrayDataProvider = require("ojs/ojmutablearraydataprovider");
import { Task, renderTask } from "./data/task-data";
import { useState, useEffect } from "preact/hooks";

type Props = {
  tasks: Array<Task>;
}

export function ListViewState({ tasks, onTaskCompleted }: Props) {
  const [ dataProvider ] = useState(() => {
    return new MutableArrayDataProvider(
      tasks, {
        keyAttributes: "taskId"
      }
    );
  });

  useEffect(() => {
    dataProvider.data = tasks;
  }, [ tasks ]);

  return (
      <oj-list-view data={dataProvider} class="demo-list-view">
          <template slot="itemTemplate"
render={renderTaskWithCompletedCallback} />
        </oj-list-view>

  )
}
```

Use Oracle JET Popup and Dialog Components

To use Oracle JET's popup or dialog components (popup content) in a VComponent or a virtual DOM app, create a reference to the popup content. We recommend too that you place the popup content in its own `div` element so that it works when used with Preact's reconciliation logic.

Currently, to launch popup content from within JSX, you must create a reference to the custom element and manually call `open()`, as in the following example for a VComponent class component that uses Preact's `createRef` function:

```
import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
import { h, Component, ComponentChild, createRef } from "preact";
import "ojs/ojdialog";
import "oj-c/button";
. . .
```

```
@customElement("popup-launching-component")
export class PopupLaunchingComponent extends
Component<ExtendGlobalProps<Props>> {

  private dialogRef = createRef();

  showDialog = () => {
    this.dialogRef.current?.open();
    console.log("open dialog");
  }

  render(): ComponentChild {
    return <div>
      <oj-c-button onojAction={this.showDialog} label="Show Dialog"></oj-c-
button>
      <div>
        <oj-dialog ref={this.dialogRef} cancelBehavior="icon"
                   modality="modeless" dialogTitle="Dialog Title">
          <div>Hello, World!</div>
        </oj-dialog>
      </div>
    </div>
  }
}
```

As a side effect of the `open()` call, Oracle JET relocates the popup content DOM to an Oracle JET-managed popup container, outside of the popup-launching component. This works and the user can see and interact with the popup.

If, while the popup is open, the popup-launching component is re-rendered by, for example, a state change, Preact's reconciliation logic detects that the popup content element is no longer in its original location and will reparent the still-open popup content back to its original parent. This interferes with Oracle JET's popup service, and unfortunately leads to non-functional popup content. To avoid this issue, we recommend that you ensure that the popup content is the only child of its parent element. In the following functional component example, we illustrate one way to accomplish this by placing the `oj-dialog` component within its own `div` element.

> **Note**
>
> In the following example we use Preact's `useRef` hook to get the reference to the DOM node inside the functional component. Use Preact's `createRef` function to get the reference to the popup content DOM node in class-based components.

```
import { h } from "preact";
import { useRef } from "preact/hooks";
import "ojs/ojdialog";
import "oj-c/button";
import { CButtonElement } from "oj-c/button";
import { DialogElement } from "ojs/ojdialog";

export function Content() {

  const dialogRef = useRef<DialogElement>(null);
```

```
  const onSubmit = (event: CButtonElement.ojAction) => {
    event.preventDefault();
    dialogRef.current?.open();
    console.log("open dialog");
  };

  const close = () => {
    dialogRef.current?.close();
    console.log("close dialog");
  };

  return <div class="oj-web-applayout-max-width oj-web-applayout-content">

    <oj-c-button onojAction={onSubmit} disabled={false} label="Open dialog">
       </oj-c-button>

    <div>
      <oj-dialog ref={dialogRef} dialogTitle="Dialog Title"
cancelBehavior="icon">
        <div>Hello, World!</div>
        <div slot="footer">
          <oj-c-button id="okButton" onojAction={close} label="Close dialog">
              </oj-c-button>
        </div>
      </oj-dialog>
    </div>
  </div>;
}
```

One further recommendation is to also use a `div` element for any wrapper component that conditionally renders popup content. Take, for example, the following statement:

```
shouldPopupContentDisplay ? <div>renderPopupContent(inputValue)</div> : null
```

Where `renderPopupContent(inputValue)` renders a wrapper component named `PopupContent` if the condition is `true`:

```
renderPopupContent(inputValue. . . )
 . . .

return {
   <PopupContent
     . . . </PopupContent>
}
 . . .
```

To avoid confusing Preact's reconciliation logic and to ensure that the wrapper component and the Oracle JET popup or dialog component render inside its own `div` element, revise the `return` method as follows:

```
renderPopupContent(inputValue. . . )
 . . .

return {
```

```
    <div><PopupContent
      . . . </PopupContent>
    </div>
  }
  . . .
```

6

Add Third-Party Tools or Libraries to Your Oracle JET App

You can add third-party tools or libraries to your Oracle JET app. The steps to do this vary depending on the method you used to create your app.

If you used the Oracle JET command-line tooling to scaffold your app, you install the library and make modifications to `appRootDir/path_mapping.json`. If you created your app using any other method and are using RequireJS, you add the library to your app and update the RequireJS bootstrap file, typically `main.js`.

> **Note**
>
> This process is provided as a convenience for Oracle JET developers. Oracle JET does not support the additional tools or libraries and cannot guarantee that they will work correctly with other Oracle JET components or toolkit features.

To add a third-party tool or library to your Oracle JET app, complete one of the following procedures.

- If you created your app with command-line tooling, perform the following steps:
  1. In your app's root directory, enter the following command in a terminal window to install the library using NPM:

     ```
     npm install library-name --save
     ```

  2. Add the new library to the path mapping configuration file.
     a. Open `appRootDir/path_mapping.json` for editing.

        A portion of the file is shown below.

        ```
        {

          "use": "local",
          "cdns": {
            "jet": "https://static.oracle.com/cdn/jet/19.0.0/default/js",
            "css": "https://static.oracle.com/cdn/jet/19.0.0/default/css",
            "config": "bundles-config.js"
          },
          "3rdparty": "https://static.oracle.com/cdn/jet/19.0.0/3rdparty"
        },
        "libs": {
          "knockout": {
              ...
          },
        ...
        ```

Oracle JET's CLI helps manage third-party libraries for your app by adding entries to its local `path_mapping.json` file.

For each library listed in the `libs` map in the `path_mapping.json` file, the following two actions happen at build time. First, one or more files are copied from somewhere (usually `appRootDir/node_modules`) into the `appRootDir/web` output folder. Second, a `requireJS` path value is created for you to represent the path to the library; it is injected into the built `main.js` file or optimized bundle. If you use TypeScript, then this should be the same path name that you use at design time for imports from the library.

The following example is an existing path mapping entry for the `persist` library, which we analyze below to describe what each attribute does and how it relates to the two build-time actions defined above.

```
1  | "persist": {
2  |   "cdn": "3rdparty",
3  |   "cwd": "node_modules/@oracle/oraclejet/dist/js/libs/persist",
4  |   "debug": {
5  |     "cwd": "debug",
6  |     "src": [
7  |       "**"
8  |     ],
9  |     "path": "libs/persist/debug",
10 |     "cdnPath": "persist/debug"
11 |   },
12 |   "release": {
13 |     "cwd": "min",
14 |     "src": [
15 |       "**"
16 |     ],
17 |     "path": "libs/persist/min",
18 |     "cdnPath": "persist/min"
19 |   }
20 | },
```

- Line 1: This is the name of the `libs` map entry ("`persist`", in this case). This name is used as the name of the `requireJS` path entry that is created; therefore, it is also the import path. The same name is used for the folder that gets created under the `appRootDir/web/js/libs` folder in the built application.

  > **Note**
  >
  > If your app uses TypeScript, then you'll also need a `tsconfig.json` file `paths` entry with the same name.

- Line 2 (Optional): The `cdn` attribute sets the name of the CDN root for the location of this library. In this case, it is set to the value `3rdparty`, which maps to a particular location in the Oracle CDN. Any third-party libraries that you add yourself (that is, those that are not present on the Oracle CDN) should not have a `cdn` or `cdnPath` value at all, unless you maintain your own CDN infrastructure where you can place the library.
- Line 3: The first role of each path mapping entry at build time is to copy files. The `cwd` attribute on this line tells the tooling where to start copying from, and

subsequent paths are relative to this root. This is generally some folder under `appRootDir/node_modules/<library>`.

- Lines 4 through 12: The `debug` and `release` sections here are the same, with the exception that if you do a normal build, then the specifications in the `debug` section are followed, and if you do a `--release` build, then the specifications in the `release` section are followed. For example, the latter case can allow the copying of only optimized assets for release.
- Line 5 (Optional): In the context of the debug or release build, this second `cwd` attribute further refines the copy root for everything that follows. For example, when running a normal (debug) build for the `persist` library, the copy root is `appRootDir/node_modules/@oracle/oraclejet/dist/js/libs/persist/debug`, where the debug `cwd` value is appended to the top-level `cwd` value. Additionally, this folder name is used in the output; the copied files end up in the `appRootDir/web/js/libs/persist/debug` directory. This attribute is optional; if omitted, then all copies are copied from paths relative to the root `cwd` location and placed in the root of the destination folder.
- Line 6: The `src` attribute holds an array of all the files that you want to copy from your root location into the built app (that is, all the files that are actually needed at runtime). The paths you add here can either be specific file names, or you can use globs to match multiple files at once. In the example above, the glob `**` is used, which results in copying everything from the `/debug` folder, including subfolders. Folder structure is preserved in the copied output under the `/web` directory.
- Line 9: The `path` attribute is used for the second role performed by each path mapping entry; it defines the value of the `requireJS` path that gets injected into the `main.js` file. For the above example, the `requireJS` path mapping is `"persist":"libs/persist/min"`, which is relative to the `requireJS` load root, typically `appRootDir/web/js`.
- Line 10 (Optional): The `cdnPath` attribute is optional and should only be used if you have actually put a copy of the library onto a CDN. If that is the case, then if the `path_mapping.json` file has the `use` attribute set to `"cdn"` rather than `"local"`, then the generated `requireJS` path statement uses this `cdnPath` value rather than the `path` value (line 9). The path here is relative to the CDN root defined by the alias allocated to the CDN and used in the `cdn` attribute for that library. If `“use”` is set to `“cdn”` but a library does not include CDN information, then the build falls back to using the local copy of the library and set up the `requireJS` path accordingly.

b. Copy one of the existing library entries in the `"libs"` map and modify as needed for your library.

The code sample below shows modifications for `my-library`, a library that contains both minified and debug versions.

```
...
 "libs": {
    "my-library": {
      "cwd": "node_modules/my-library/dist",
      "debug": {
        "src": "my-library.debug.js",
        "path": "libs/my-library/my-library.debug.js"
      },
      "release": {
```

```
        "src": "my-library.js",
        "path": "libs/my-library/my-library.js"
      }
    },
...
```

In this example, the `cwd` attribute points to the location where NPM installed the library, the `src` attribute points to a path or array of paths containing the files that are copied during a build, and the `path` attribute points to the destination that contains the built version.

When defining your own library entry in the `path_mapping.json` file, you should test it out by running the `ojet build` command and then confirm that all the expected files have been copied into the `appRootDir/web/js/libs` directory and that the `requireJS` path mapping has been injected in the built `appRootDir/web/js/main.js` file.

> **Note**
>
> If the existing library entry that you copy to modify includes `"cdn": "3rdparty",` remove that line from the newly-created entry for your library. This line references the Oracle JET third-party area on the content distribution network (CDN) managed by Oracle. Your library won't be hosted there, and keeping this line causes a release build to fail at runtime by mapping your library's path to a non-existent URL.
>
> If you use a CDN, add the URL to the CDN in the entry for the `cdnPath` attribute.

3. If your project uses TypeScript, then you also need to define a `paths` entry in your `tsconfig.json` file that allows you to import the library using the same path name at design time as you'll use at runtime. If the library in question also provides some types for you to use, then the path should point to these to allow your editor to provide TypeScript support for your usage of that library. Here is an example of an existing `paths` entry in an Oracle JET app created with the command-line tooling.

```
"paths": {
  "ojs/*": [
    "./node_modules/@oracle/oraclejet/dist/types/*"
],
...
```

- If you didn't use command-line tooling to create your app, perform the following steps to add the tool or library.

  1. In the app's `/libs` directory, create a new directory and add the new library and any accompanying files to it.

     For example, for a library named `my-library`, create the `my-library` directory and add the `my-library.js` file and any needed files to it. Be sure to add the minified version if available.

  2. In your RequireJS bootstrap file, typically `main.js`, add a path for the library file in the path mapping section of the `requirejs.config()` definition.

For example, add the highlighted code below to your bootstrap file to use a library named `my-library`.

```
requirejs.config({
  // Path mappings for the logical module names
  paths:
  {
    'knockout': 'libs/knockout/knockout-3.x.x',
    'jquery': 'libs/jquery/jquery-3.x.x.min',
    ...
    'text': 'libs/require/text',
    'my-library': 'libs/my-library/my-library
  },
  require(['knockout', 'my-library'],
  // this callback gets executed when all
  // required modules are loaded
  function(ko)
  {
      // Add any start-up code that you want here
  }
);
```

3. If your project uses TypeScript, then you also need to define a `paths` entry in your `tsconfig.json` file that allows you to import the library using the same path name at design time as you'll use at runtime. If the library in question also provides some types for you to use, then the path should point to these to allow your editor to provide TypeScript support for your usage of that library. Here is an example of an existing `paths` entry in an Oracle JET app created with the command-line tooling.

```
"paths": {
  "ojs/*": [
    "./node_modules/@oracle/oraclejet/dist/types/*"
 ],
...
```

7

Validate and Convert Input

Oracle JET includes validators and converters on a number of Oracle JET editable elements, including `oj-combobox`, `oj-input*`, and `oj-text-area`. You can use them as is or customize them for validating and converting input in your Oracle JET app.
Some editable elements such as `oj-checkboxset`, `oj-radioset`, and `oj-select` have a simple attribute for `required` values that implicitly creates a built-in validator.

> **Note**
>
> The `oj-input*` mentioned above refers to the family of input components such as `oj-input-date-time`, `oj-input-text`, and `oj-input-password`, among others.

About Oracle JET Validators and Converters

Oracle JET provides converter classes that convert user input strings into the data type expected by the app and validator classes that enforce a validation rule on those input strings.

For example, you can use Oracle JET's `IntlDateTimeConverter` to convert a user-entered date to a date-only ISO string and then use `DateTimeRangeValidator` to validate that input against a specified date range. If the converters or validators included in Oracle JET are not sufficient for your app, you can create custom converters or validators.

About Validators

All Oracle JET editable elements support a `value` attribute and provide UI elements that allow the user to enter or choose a value. These elements also support other attributes that you can set to instruct the element how to validate its value.

An editable element may implicitly create a built-in converter and/or built-in validators for its normal functioning when certain attributes are set. For example, editable elements that support a `required` property create the required validator implicitly when the property is set to `true`. Other elements like `oj-input-date`, `oj-input-date-time`, and `oj-input-time` create a datetime converter to implement its basic functionality.

About the Oracle JET Validators

The following table describes the Oracle JET validators and provides links to the API documentation:

Validator	Description	Link to API	Module
`DateTimeRangeValidator`	Validates that the input date is between two dates, between two times, or within two date and time ranges	DateTimeRangeValidator	`ojvalidation-datetimerange`
`DateRestrictionValidator`	Validates that the input date is not a restricted date	DateRestrictionValidator	`ojvalidation-daterestriction`

Validator	Description	Link to API	Module
`LengthValidator`	Validates that an input string is within a specified length	LengthValidator	`ojvalidation-length`
`NumberRangeValidator`	Validates that an input number is within a specified range	NumberRangeValidator	`ojvalidation-numberrange`
`RegExpValidator`	Validates that the regular expression matches a specified pattern	RegExpValidator	`ojvalidation-regexp`
`RequiredValidator`	Validates that a required entry exists	RequiredValidator	`ojvalidation-required`

About Oracle JET Component Validation Attributes

The attributes that a component supports are part of its API, and the following validation specific attributes apply to most editable elements.

Element Attribute	Description
`converter`	When specified, the `converter` instance is used over any internal converter the element might create. On elements such as `oj-input-text`, you may need to specify this attribute if the value must be processed to and from a number or a date value.
`countBy`	When specified on `LengthValidator`, `countBy` enables you to change the validator's default counting behavior. By default, this property is set to `codeUnit`, which uses JavaScript's String length property to count a UTF-16 surrogate pair as length === 2. Set this to `codePoint` to count surrogate pairs as length ===1.
`max`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the element creates an implicit range validator.
`min`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the component creates an implicit range validator.
`required`	When specified on an Oracle JET element, the element creates an implicit required validator.
`validators`	When specified, the element uses these validators along with the implicit validators to validate the UI value. Can be implemented with `Validators` or `AsyncValidators` to validate the user input on the server asynchronously.

Some editable elements do not support specific validation attributes as they might be irrelevant to its intrinsic functioning. For example, `oj-radioset` and `oj-checkboxset` do not support a converter attribute since there is nothing for the converter to convert. For an exact list of attributes and how to use them, refer to the `Attributes` section in the element's API documentation. For Oracle JET API documentation, see API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). Select the component you're interested in viewing from the API list.

About Oracle JET Component Validation Methods

Oracle JET editable elements support the following methods for validation purposes. For details on how to call this method, its parameters and return values, refer to the component's API documentation.

Element Method	Description
`reset()`	Use this method to reset the element by clearing all messages and messages attributes - `messagesCustom` - and update the element's display value using the attribute value. User entered values will be erased when this method is called.
`validate()`	Use this method to validate the component using the current display value.

For details on calling an element's method, parameters, and return values, see the `Methods` section of the element's API documentation in API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET). You can also find detail on how to register a callback for or bind to the event and for information about what triggers the events. Select the component you're interested in viewing from the API list.

About Oracle JET Converters

Oracle JET provides converters to convert date, date-time, number, color, and string.

These converters extend the `Converter` object which defines a basic contract for converter implementations. The converter API is based on the ECMAScript Internationalization API specification (ECMA-402 Edition 1.0) and uses the Unicode Common Locale Data Repository (CLDR) for its locale data. Both converters are initialized through their constructors, which accept options defined by the API specification. For additional information about the ECMA-402 API specification, see https://www.ecma-international.org/publications-and-standards/standards/ecma-402/. For information about the Unicode CLDR, see http://cldr.unicode.org.

The Oracle JET implementation extends the ECMA-402 specification by introducing additional options. You can use the converters with an Oracle JET component or instantiate and use them directly on the page. Each converter has a `ConverterOptions` type definition that specifies the conversion options it supports. The following table describes the available converters and provides a link to the API documentation for each converter, including detailed descriptions of the properties supported by each converter's `ConverterOptions` type definition.

Converter	Description	Link to API
`ColorConverter`	Converts Color object formats	ColorConverter
`IntlDateTimeConverter`	Parses a string into an ISO string format (`yyyy-mm-dd`) or converts an ISO string into a standard string format. For example: • `10/12/2020` ---> `2020-10-12` • `2020-10-12` ---> `10/12/2020`	IntlDateTimeConverter
`NumberConverter`	Converts a string into a number and formats a number into a locale-specific string	NumberConverter
`BigDecimalStringConverter`	Converts a big-decimal string into a locale-specific string. Use for very large numbers (greater than `Number.MAX_SAFE_INTEGER`) and numbers with large scale (more than 17 fractional digits). Can only be used with components that expect a string value, such as `oj-c-input-text`.	BigDecimalStringConverter

Converter	Description	Link to API
`LocalDateConverter`	Converts a date-only ISO string to a formatted string or a string to a date-only ISO string	LocalDateConverter

The Oracle JET converters support lenient number and date parsing when the user input does not exactly match the expected pattern. The parser does the lenient parsing based on the leniency rules for the specific converter. For example, both `NumberConverter` and `BigDecimalStringConverter` remove unexpected characters. The `ConverterOptions` type definition typically includes a `lenientParse` property that you can set to `none` so that user input matches the expected input or an exception is thrown. For specific details, see the API documentation for the converter that you are using.

The resource bundles that hold the locale symbols and data used by the Oracle JET converters are downloaded automatically based on the locale set on the page when using RequireJS and the `ojs/ojvalidation-datetime` or `ojs/ojvalidation-number` module. If your app does not use RequireJS, the locale data will not be downloaded automatically.

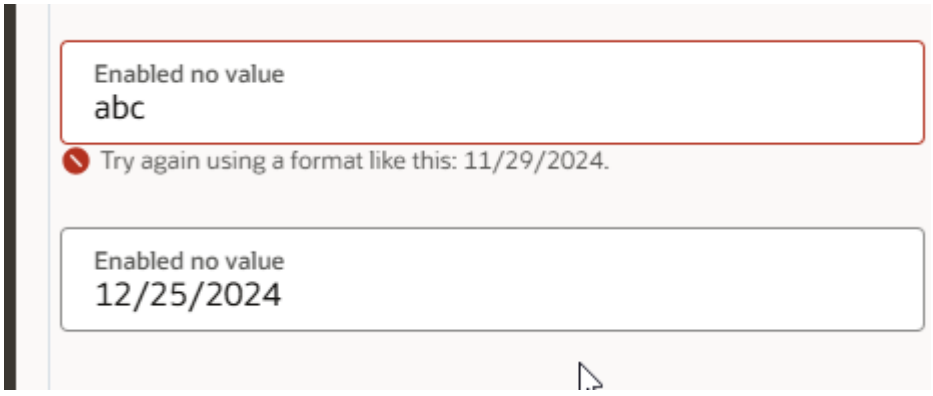
You can use the converters with an Oracle JET component or instantiate and use them directly on the page.

Use Oracle JET Converters with Oracle JET Components

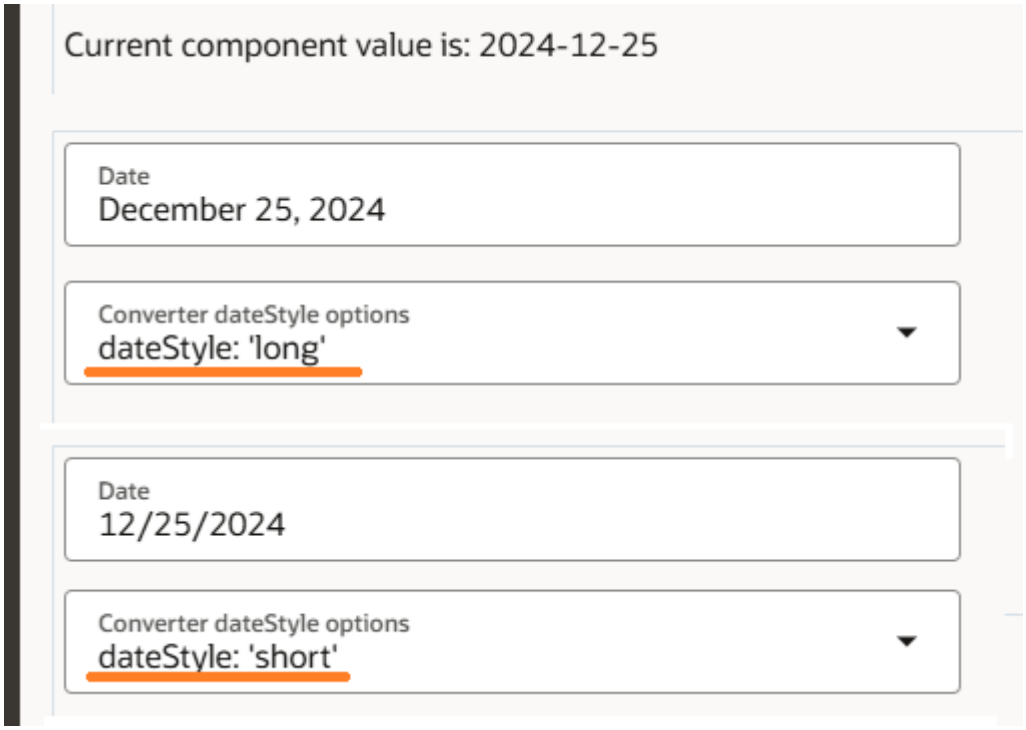
Oracle JET components that accept user input, such as `oj-c-input-date-text`, already include an implicit converter that parses user input. However, you can also specify an explicit converter on the component that will be used instead when converting data.

In the following example, the `oj-c-input-date-text` component displays an error message if you enter "`abc`" while it renders input of "`12/25/24`" as "`12/25/2024`" using its implicit converter.

The error that the converter throws when there are errors during parsing or formatting operations is represented by the `ConverterError` object, and the error message is represented by an object of type `Message`. The messages that Oracle JET converters use are resources that are defined in the translation bundle included with Oracle JET.



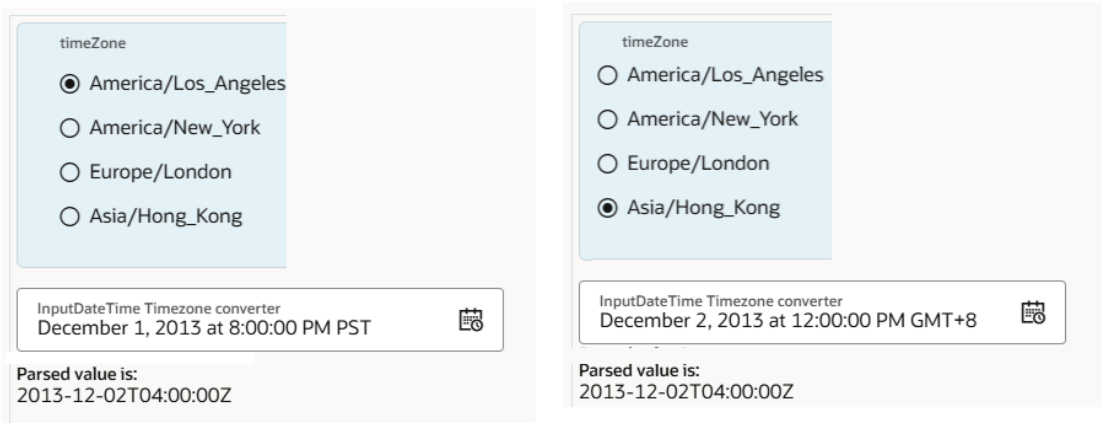
You can also specify the converter directly on the component's converter `attribute`, if it exists. For example, the `oj-c-input-date-text` component uses the `converter` attribute to change the date style that the component renders to one of the converter options supported by `LocalDateConverter`.



Understand Time Zone Support in Oracle JET

Oracle JET input components, such as `oj-input-date-time`, support local time zone input. You can enable time zone support by using Oracle JET's `IntlDateTimeConverter`, which relies on JavaScript's `Intl.DateTimeFormat` API.

In the following image, the Input Date Time component's `converter` attribute references code that renders the input time of `2013-12-02T04:00:00Z` appropriately depending on whether the time zone is `America/Los_Angeles` or `Asia/Hong_Kong`.



For more information about Oracle JET's `IntlDateTimeConverter`, see IntlDateTimeConverter.

About Oracle JET Validators

Oracle JET validators provide properties that allow callers to customize the validator instance. The properties are documented as part of the validators' API. Unlike converters where only one instance of a converter can be set on an element, you can associate one or more validators with an element.

When a user interacts with the element to change its value, the validators associated with the element run in order. When the value violates a validation rule, the `value` attribute is not populated, and the validator highlights the element with an error.

You can use the validators with an Oracle JET element or instantiate and use them directly on the page.

Use Oracle JET Validators with Oracle JET Components

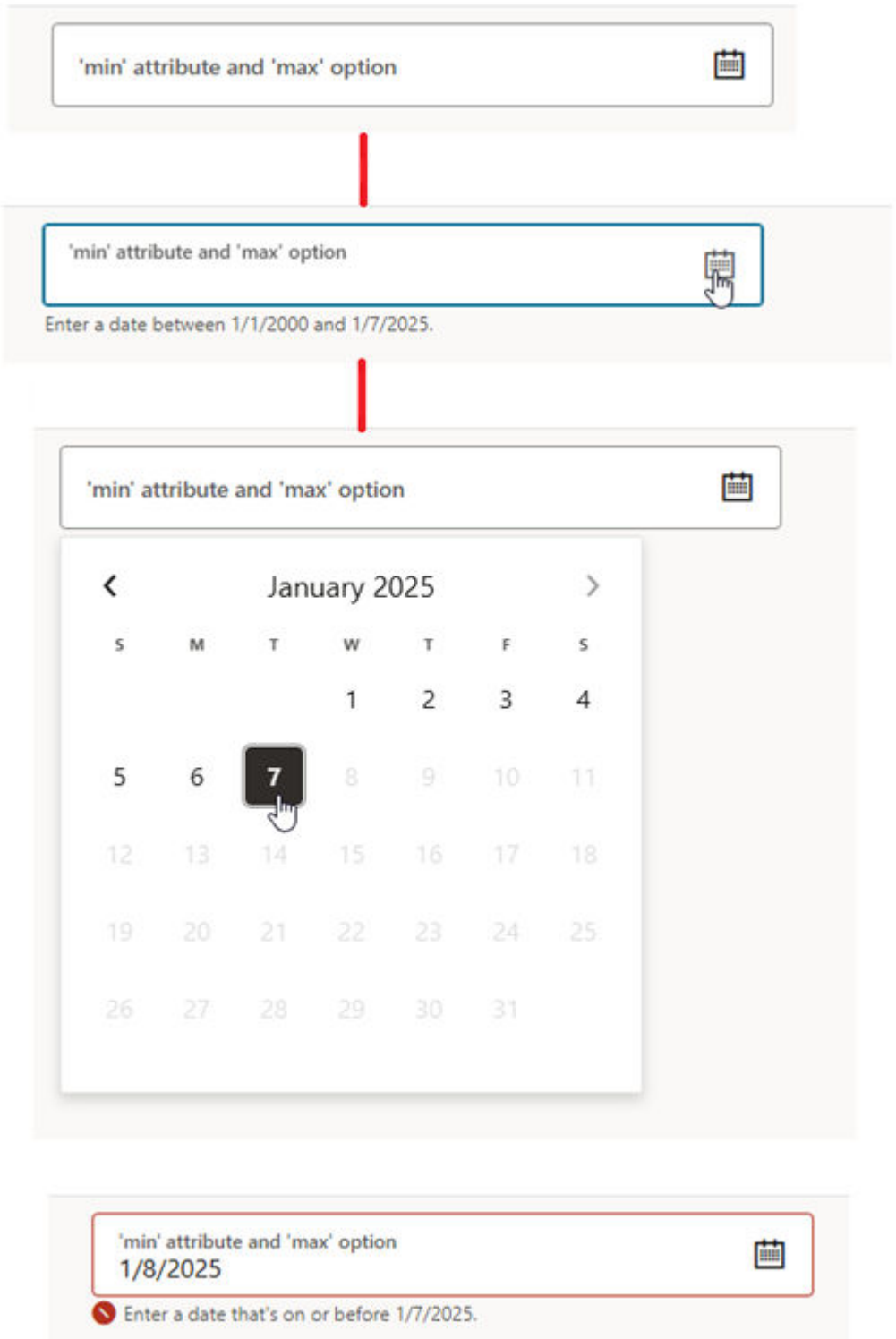
Oracle JET editable elements, such as `oj-input-text` and `oj-input-date`, set up validators both implicitly, based on certain attributes they support such as `required`, `min`, `max`, and so on, and explicitly by providing a means to set up one or more validators using the component's `validators` attribute.

For example, the following code shows an `oj-input-date` element that uses the default validator supplied by the component implicitly. When the `oj-input-date` component reads the `min` and `max` attributes, it creates the implicit `DateTimeRangeValidator`.

```
import { h } from "preact";
import * as ConverterUtilsI18n from "ojs/ojconverterutils-i18n";
import "ojs/ojformlayout";
import "ojs/ojdatetimepicker";

export function Content() {
  let todayIsoDate: string =
ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(
    new Date()
  );

  let milleniumStartIsoDate: string =
ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(
    new Date(2000, 0, 1)
  );

  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <oj-form-layout id="formLayout1" columns={1}>
        <oj-input-date
          id="dateTimeRange1"
          min={milleniumStartIsoDate}
          max={todayIsoDate}
          labelHint="'min' attribute and 'max' option"
        ></oj-input-date>
      </oj-form-layout>
    </div>
```

```
  );
}
```

When the user runs the page, the `oj-input-date` element displays an input field with a calendar icon. If you input data that is not within the expected range, the built-in validator displays an error message with the expected range.



The error thrown by the Oracle JET validator when validation fails is represented by the `ValidatorError` object, and the error message is represented by an object of type `Message`. The messages and hints that Oracle JET validators use when they throw an error are resources that are defined in the translation bundle included with Oracle JET.

You can also specify the validator on the element's `validators` attribute, if it exists. The code sample below shows another `oj-input-date` element that calls a function which specifies the `DateTimeRangeValidator` validator (`dateTimeRange`) in the `validators` attribute.

```
import { h } from "preact";
import { useMemo } from "preact/hooks";
import * as ConverterUtilsI18n from "ojs/ojconverterutils-i18n";
import "ojs/ojformlayout";
import "ojs/ojdatetimepicker";
import AsyncDateTimeRangeValidator = require("ojs/ojasyncvalidator-
datetimerange");
import * as DateTimeConverter from "ojs/ojconverter-datetime";

export function Content() {
  let todayIsoDate: string =
ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(
    new Date()
  );

  let milleniumStartIsoDate: string =
ConverterUtilsI18n.IntlConverterUtils.dateToLocalIso(
    new Date(2000, 0, 1)
  );

  const dateRange = useMemo(() => {
    return [
      new AsyncDateTimeRangeValidator({
        max: todayIsoDate,
        min: milleniumStartIsoDate,
        hint: {
          inRange:
            "Enter a date that falls in the current millennium and "
            + "is not greater than today's date.",
        },
        converter: new DateTimeConverter.IntlDateTimeConverter({
          day: "2-digit",
          month: "2-digit",
          year: "2-digit",
        }),
      }),
    ];
  }, [todayIsoDate, milleniumStartIsoDate]);

  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <oj-form-layout id="formLayout1" columns={1}>
        <oj-input-date
          id="dateTimeRange2"
          labelHint="'dateTimeRange' type in 'validators' option"
          validators={dateRange}
        ></oj-input-date>
```

```
      </oj-form-layout>
    </div>
  );
}
```

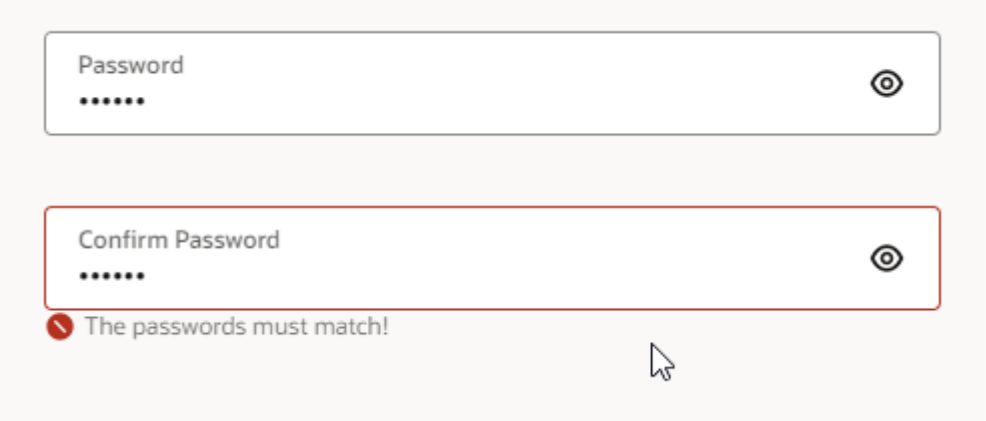
When the user runs the page for the `en-US` locale, the `oj-input-date` element displays an input field that expects the user's input date to be between `01/01/2000` and the current date. When entering a date value into the field, the date converter accepts alternate input as long as it can parse it unambiguously. This offers end users a great deal of leniency when entering date values. For example, typing `1-2-3` converts to a `Date` that falls on the 2nd day of January, 2003. If the `Date` value also happens to fall in the expected `Date` range set in the validator, then the value is accepted. If validation fails, the component displays an error.

Oracle JET elements can also use a `regExp` validator. If the regular expression pattern requires a backslash, while specifying the expression within an Oracle JET element, you need to use double backslashes. The options that each validator accepts are specified in API Reference for Oracle® JavaScript Extension Toolkit (Oracle JET).

Use Custom Validators in Oracle JET

You can create custom validators in Oracle JET that you can reference like built-in validators from the `validators` attribute.

The following image shows a custom validator that displays an error message if the user's password doesn't match.



To create and use a custom validator:

1. Reference the custom validator from the `validators` attribute of your Oracle JET component.

   In the following example, the component that renders the Confirm Password input field references a `confirmPasswordValidator` variable from its `validators` attribute.

   ```
   <oj-c-input-password
      id="cpassword"
      value={passwordRepeat}
      onvalueChanged={handlePasswordRepeat}
      validators={confirmPasswordValidator}
      label-hint="Confirm Password"
   ```

```
   mask-icon="visible">
  </oj-c-input-password>
```

2. Write your custom validator.

   In the following example, the `validateEqualToPassword` function validates that the value entered in the Confirm Password field matches the value previously entered in the Password field. The `confirmPasswordValidator` variable uses a `useMemo` hook to ensure that `confirmPasswordValidator` only recomputes when `validateEqualToPassword` changes.

```
. . .
const validateEqualToPassword = useCallback(
  (value: string) => {
    if (value && value !== password) {
      throw new Error(
        "The passwords must match!"
      );
    }
    setError(null);
    return Promise.resolve();
  },
  [password]
);

const confirmPasswordValidator = useMemo(
  () => [{ validate: validateEqualToPassword }],
  [validateEqualToPassword]
);
. . .
```

About Asynchronous Validators

Oracle JET input components support asynchronous server-side validation via the `validators` attribute. That means you can check input values against server data without the need to submit a form or refresh a page.

Two example scenarios illustrate where you can use asynchronous server-side validation:

- In a form that collects new user data, you validate input in an email field to check if the input value has been registered previously.
- Set number range validators that check against volatile data. For example, on an e-commerce website, you can check the user's cart against the available inventory and inform the user if the goods are unavailable without them submitting the cart for checkout.

The following code shows an `oj-c-input-text` element with the `validators` attribute set to an array of functions (`validators` and `asyncValidator`). The synchronous validator, `validators`, checks if the input value is `500`. If it is, an error is thrown. The asynchronous validator, `asyncValidator`, returns a Promise. A Promise represents a value that may not be available yet but will be resolved at some point in the future. The Promise evaluates to `true` if validation passes and if validation fails, it returns an error. `AsyncNumberRangeValidator` checks if the input value falls within a specified range (100 to 1000). It simulates a server-side delay using

`setTimeout` before performing the validation. The `validators` attribute must be of type `AsyncValidator` to fulfill the API contract required to create the asynchronous validator.

```
import { h } from "preact";
import "ojs/ojformlayout";
import { useState } from "preact/hooks";
import AsyncNumberRangeValidator = require("ojs/ojasyncvalidator-
numberrange");
import { CInputTextElement } from "oj-c/input-text";
import "oj-c/input-text";

export function Content() {
  const [quantityLimit, setQuantityLimit] = useState("");

  const onValueChanged = (event: CInputTextElement.valueChanged<string>) => {
    setQuantityLimit(event.detail.value);
  };

  const minQuantity = 100;
  const maxQuantity = 1000;

  // synchronous validator.
  const validators = {
    validate: (value: string | number) => {
      if (value === 500 || value === "500") {
        throw new Error("500 is invalid");
      } else {
        return;
      }
    },
    getHint: () => {
      return "To see an immediate error from the synchronous validator
(validators), enter 500.";
    },
  };

  // asynchronous validator.
  const asyncValidator = {
    validate: (value: string | number) => {
      const numberRangeValidator = new AsyncNumberRangeValidator({
        min: minQuantity,
        max: maxQuantity,
      });

      return new Promise<void>((resolve, reject) => {
        // Simulate server-side delay
        setTimeout(() => {
          numberRangeValidator.validate(value).then(
            () => {
              resolve();
            },
            (e) => {
              // Handle validation error
              reject(
                new Error(
                  `${value} is not in the accepted range of ${minQuantity} - $
```

```
{maxQuantity}`
                )
              );
            }
          );
        }, 1000);
      });
    },

    hint: new Promise<string>((resolve) => {
      // Simulate server-side delay
      setTimeout(() => {
        resolve(
          `To see an error from the asynchronous validator (asyncValidator)
that appears after 1 second,
           enter a number outside the range of ${minQuantity} - ${maxQuantity}`
        );
      }, 100);
    }),
  };

  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <oj-form-layout columns={1} class="oj-md-margin-4x-horizontal">
        <oj-c-input-text
          id="input-text"
          labelHint="Quantity Limit"
          onvalueChanged={onValueChanged}
          validators={[validators, asyncValidator]}
          value={quantityLimit}
        ></oj-c-input-text>
      </oj-form-layout>
    </div>
  );
}
```

For more information, see the `validators` attribute section of Input Text or see Promise (MDN).

8

Internationalize and Localize Oracle JET Apps

Oracle JET includes support for internationalization (I18N), localization (L10N), and use of Oracle National Language Support (NLS) translation bundles in Oracle JET apps. Configure your Oracle JET app so that the app can be used in a variety of locales and international user environments.

About Internationalizing and Localizing Oracle JET Apps

Internationalization (I18N) is the process of designing software so that it can be adapted to various languages and regions easily, cost effectively, and, in particular, without engineering changes to the software. Localization (L10N) is the use of locale-specific language and constructs at runtime.

Oracle has adopted the industry standards for I18N and L10N, such as World Wide Web Consortium (W3C) recommendations, Unicode technologies, and Internet Engineering Task Force (IETF) specifications to enable support for the various languages, writing systems, and regional conventions of the world. Languages and locales are identified with a standard language tag and processed as defined in BCP 47. Oracle JET includes Oracle National Language Support (NLS) translation support for the languages listed in the following table.

Language	Language Tag
Arabic	`ar`
Brazilian Portuguese	`pt`
Bulgarian	`bg-BG`
Canadian French	`fr-CA`
Chinese (Simplified)	`zh-Hans` (or `zh-CN`)
Chinese (Traditional)	`zh-Hant` (or `zh-TW`)
Croatian	`hr`
Czech	`cs`
Danish	`da`
Dutch	`nl`
Estonian	`et`
Finnish	`fi`
French	`fr`
German	`de`
Greek	`el`
Hebrew	`he`
Hungarian	`hu`
Icelandic	`is`
Italian	`it`
Japanese	`ja`

Language	Language Tag
Korean	`ko`
Latin_Serbian	`sr-Latn`
Latvian	`lv`
Lithuanian	`lt`
Malay	`ms-MY`
Norwegian	`no`
Polish	`pl`
Portuguese	`pt-PT`
Romania	`ro`
Russian	`ru`
Serbian	`sr`
Slovak	`sk`
Slovenian	`sl`
Spanish	`es`
Swedish	`sv`
Thai	`th`
Turkish	`tr`
Ukrainian	`uk-UA`
Vietnamese	`vi`

Oracle JET translations are stored in a resource bundle. You can add your own translations to the bundles. For additional information, see Add RequireJS Translation Bundles to an Oracle JET App.

Oracle JET also includes formatting support for over 196 locales. Oracle JET locale elements are based upon the Unicode Common Locale Data Repository (CLDR) and are stored in locale bundles. For additional information about Unicode CLDR, see http://cldr.unicode.org. You can find the supported locale bundles in the Oracle JET distribution:

```
js/libs/oj/19.0.0/resources/nls
```

It is the app's responsibility to determine the locale used on the page. Typically, the app determines the locale by calculating it on the server side from the browser locale setting or by using the user locale preference stored in an identity store and the supported translation languages of the app.

Once the locale is determined, your app must communicate this locale to Oracle JET for its locale-sensitive operations, such as loading resource bundles and formatting date-time data. Oracle JET determines the locale for locale-sensitive operations in the following order:

1. Locale specification in the ojL10n plugin of the RequireJS configuration.
2. `lang` attribute of the `html` tag.
3. `navigator.language` browser property.

If your app will not provide an option for users to change the locale dynamically, then setting the `lang` attribute on the `html` tag is the recommended practice because, in addition to setting the locale for Oracle JET, it sets the locale for all HTML elements as well. Oracle JET automatically loads the translations bundle for the current language and the locale bundle for

the locale that was set. If you don't set a locale, Oracle JET will default to the browser property. If, however, your app provides an option to change the locale dynamically, we recommend that you set the locale specification in the ojL10n plugin if your app uses RequireJS. Oracle JET loads the locale and resource bundles automatically when your app initializes.

If you use Webpack rather than RequireJS as the module bundler for your Oracle JET app, we recommend that you generate one code bundle for each locale that you want to support and deploy each bundle to a different URL for the locale that you want to support. If, for example, your app URL is https://www.oracle.com/index.html and you want to support the French and Spanish locales, deploy the bundles for these locales to https://www.oracle.com/fr/index.html and https://www.oracle.com/es/index.html respectively.

Finally, Oracle JET includes validators and converters that use the locale bundles. When you change the locale on the page, an Oracle JET component has built-in support for displaying content in the new locale. For additional information about Oracle JET's validators and converters, see Validate and Convert Input.

Internationalize and Localize Oracle JET Apps

Configure your app to use Oracle JET's built-in support for internationalization and localization.

Use Oracle JET's Internationalization and Localization Support

To use Oracle JET's built-in internationalization and localization support, you can specify one of the supported languages or locales on the `lang` attribute of the `html` element on your page. For example, the following setting will set the language to the French (France) locale.

```
<html lang="fr-FR">
```

If you want to specify the French (Canada) locale, you would specify the following instead:

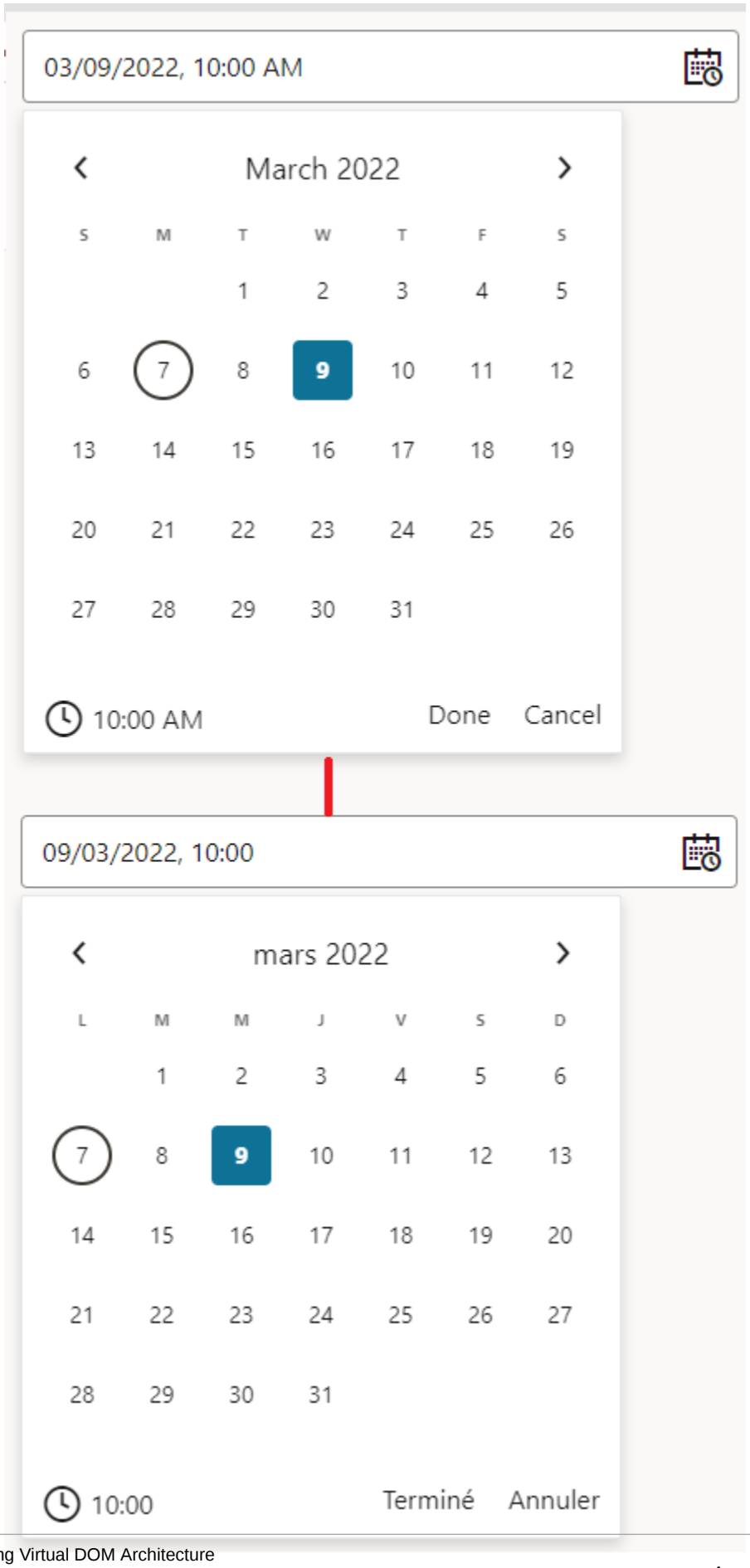
```
<html lang="fr-CA">
```

> **Tip**
>
> The `locale` specification isn't case sensitive. Oracle JET will accept `FR-FR`, `fr-fr`, and so on, and map it to the correct resource bundle directory.

When you specify the locale in this manner, any Oracle JET component on your page will display in the specified language and use locale constructs appropriate for the locale.

If the locale doesn't have an associated resource bundle, Oracle JET will load the lesser significant language bundle. If Oracle JET doesn't have a bundle for the lesser significant language, it will use the default root bundle. For example, if Oracle JET doesn't have a translation bundle for `fr-CA`, it will look for the `fr` resource bundle. If the `fr` bundle doesn't exist, Oracle JET will use the default root bundle and display the strings in English.

In the image below, the page is configured with the `oj-input-date-time` component. The image shows the effect of changing the `lang` attribute to `fr-FR`.



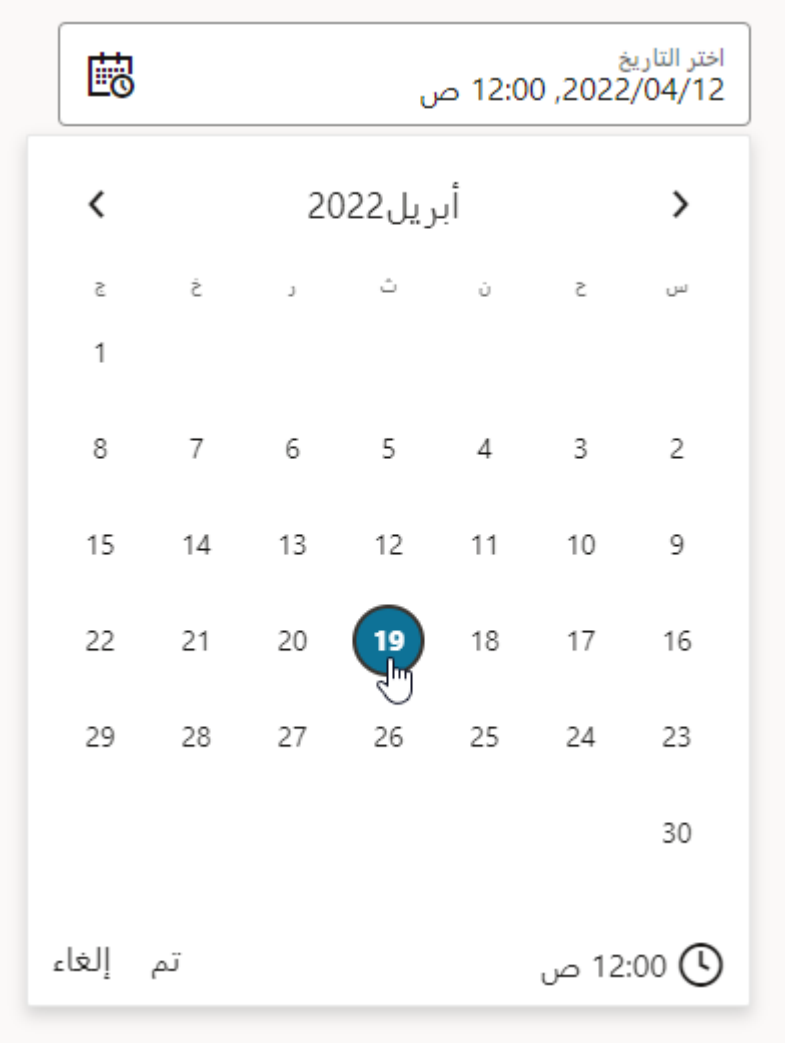
If you type an incorrect value in the `oj-input-date-time` field, the error text displays in the specified language. In this example, the error displays in French.

Enable Bidirectional (BiDi) Support in Oracle JET

If the language you specify uses a right-to-left (RTL) direction instead of the default left-to-right (LTR) direction, such as Arabic and Hebrew, you must specify the `dir` attribute on the `html` tag.

```
<html dir="rtl">
```

The image below shows the `oj-input-date-time` field that displays if you specify the Arabic (Egypt) language code and change the `dir` attribute to `rtl`.



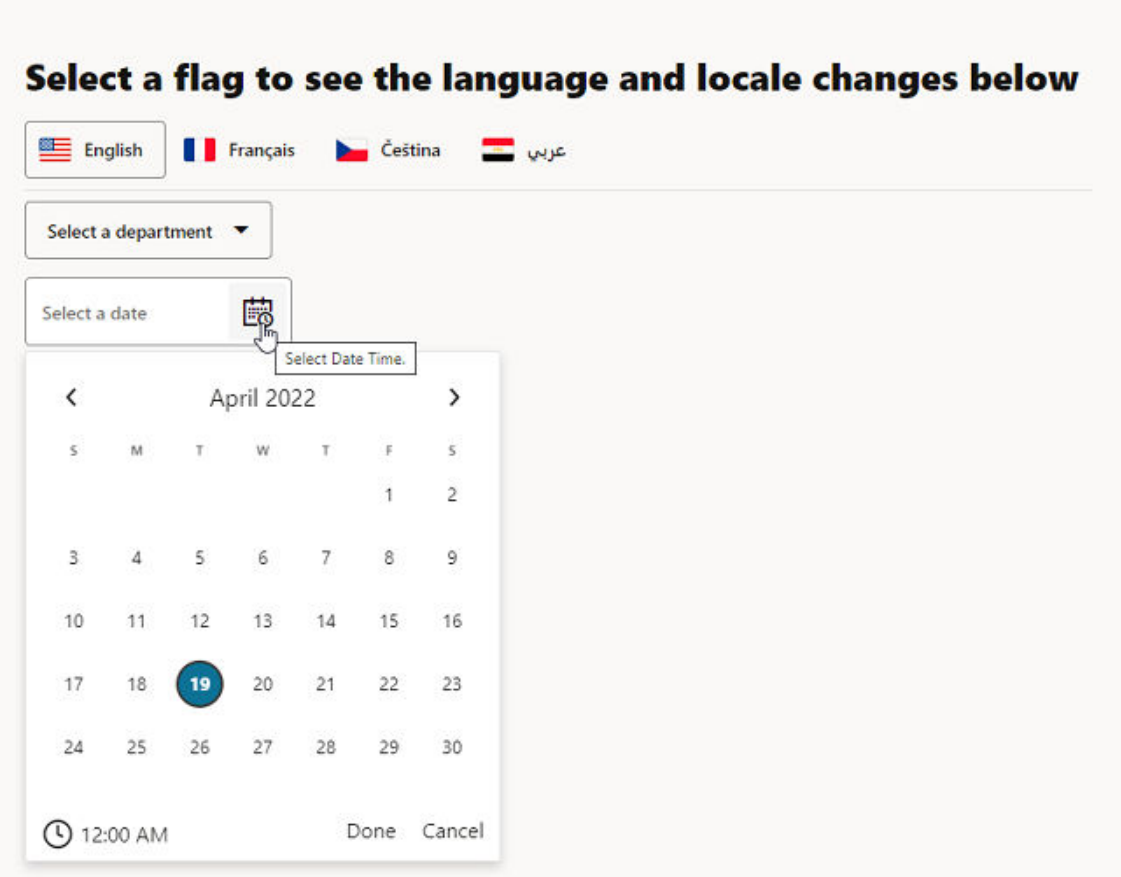
Once you have enabled BiDi support in your Oracle JET app, you must still ensure that your app displays properly in the desired layout and renders strings as expected.

> **Note**
>
> Oracle JET does not support the setting of the `dir` attribute on individual HTML elements which would cause a page to show mixed directions. Also, if you programmatically change the `dir` attribute after the page has been initialized, you must reload the page or refresh each JET component.

Set the Locale and Direction Dynamically

You can configure your app to change its locale and direction dynamically by setting a key-value pair in the app's local storage that your app's RequireJS ojL10n plugin reads when you reload the app URL.

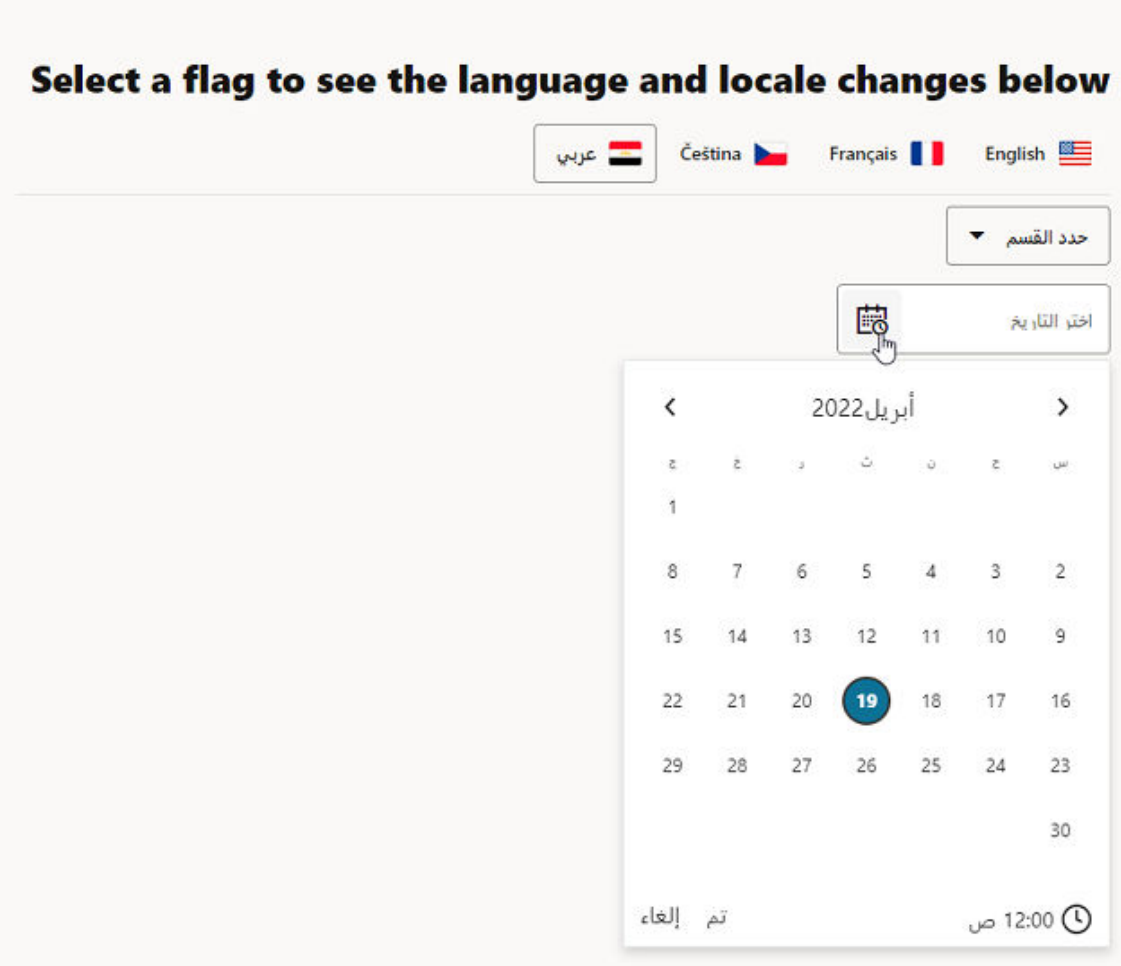
The image below shows an Oracle JET app configured to display a menu that displays a department list when clicked and a date picker. By default, the app is set to the `en-US` locale. Both the menu and date picker display in English.



The app also includes a button set that shows the United States of America, France, Czech Republic, and Egypt flags. When the user clicks one of the flags, the app locale is set to the locale represented by the flag: `en-US`, `fr-FR`, `cs-CZ`, or `ar-EG`.

> **Note**
>
> The flags used in this example are for illustrative use only. Using national flags to select a UI language is strongly discouraged because multiple languages are spoken in one country, and a language may be spoken in multiple countries as well. In a real app, you can use clickable text instead that indicates the preferred language to replace the flag icons.

The image below shows the updated page after the user clicks the Egyptian flag.



Implementing this behavior requires you to make changes in the JSX of your app's component and the `appRootDir/src/main.js` file of your app. In the JSX, the `onvalueChanged` property change listener attribute specifies a `setLang` function that is called when a user changes the selected button.

```
<oj-buttonset-one . . . onvalueChanged={setLang}>
```

In the JSX code of the app, this `setLang` function determines what locale the user selected and sets entries in window.localStorage so that a user's selection persists across browser

sessions. The final step in the function is to reload the current URL using the location.reload() method.

```
setLang = (evt) => {
    let newLocale = "";
    let lang = evt.detail.value;
    switch (lang) {
      case "Čeština":
        newLocale = "cs-CZ";
        break;
      case "Français":
        newLocale = "fr-FR";
        break;
      case "ع":
          newLocale = "ar-EG";
          break;
      default:
        newLocale = "en-US";
    }
    window.localStorage.setItem('mylocale',newLocale);
    window.localStorage.setItem('mylang',lang);
    location.reload();
  };
```

To set the newly-selected locale in the ojL10n plugin of our app's `appRootDir/src/main.js` file, we write the following entries that read the updated locale value from local storage, and set it on the `locale` specification in the ojL10n plugin. We also include a check that sets the direction to `rtl` if the specified locale is Egyptian Arabic (`ar-EG`).

```
(function () {
  ...
const localeOverride = window.localStorage.getItem("mylocale");
  if (localeOverride) {
    // Set dir attribute on <html> element.
    // Note that other Arabic locales and Hebrew also use the rtl direction.
    // Include a check here for other locales that your app must support.
    if(localeOverride === "ar-EG"){
      document.getElementsByTagName('html')[0].setAttribute('dir','rtl');
    } else {
      document.getElementsByTagName('html')[0].setAttribute('dir','ltr');
    }
    requirejs.config({
      config: {
        ojL10n: {
          locale: localeOverride,
        },
      },
    });
  }
})();
...
```

For information about defining your own translation strings and adding them to the Oracle JET translation bundle, see Add RequireJS Translation Bundles to an Oracle JET App or Add ICU

Translation Bundles to an Oracle JET Virtual DOM App, depending on the type of translation bundle that you use.

When you use this approach to internationalize and localize your app, you must consider every component and element on your page and provide translation strings where needed. If your page includes a large number of translation strings, the page can take a performance hit.

Also, if SEO (Search Engine Optimization) is important for your app, be aware that search engines normally do not run JavaScript and access static text only.

> **Tip**
>
> To work around issues with performance or SEO, you can add pages to your app that are already translated in the desired language. When you use pages that are already translated, the Knockout bindings are executed only for truly dynamic pieces.

Work with Currency, Dates, Time, and Numbers

When you use the converters included with Oracle JET, dates, times, numbers, and currency are automatically converted based on the locale settings. You can also provide custom converters if the Oracle JET converters are not sufficient for your app. For additional information about Oracle JET converters, see About Oracle JET Converters.

Work with Oracle JET RequireJS Translation Bundles

Oracle JET includes a RequireJS translation bundle that translates strings generated by Oracle JET components into all supported languages. Add your own RequireJS translation bundle by merging it with the Oracle JET RequireJS translation bundle.

> **Note**
>
> If you are creating a translation bundle for the first time in your Oracle JET app, we recommend that you use the other type of translation bundle (ICU translation bundle) that Oracle JET supports. For more information, see Work with ICU Translation Bundles in an Oracle JET Virtual DOM App.

About Oracle JET Translation Bundles

Oracle JET includes a translation bundle that translates strings generated by Oracle JET components into all supported languages. You can add your own translation bundle following the same format used in Oracle JET.

The Oracle JET translation bundle follows a specified format for the content and directory layout but also allows some leniency regarding case and certain characters.

Translation Bundle Location

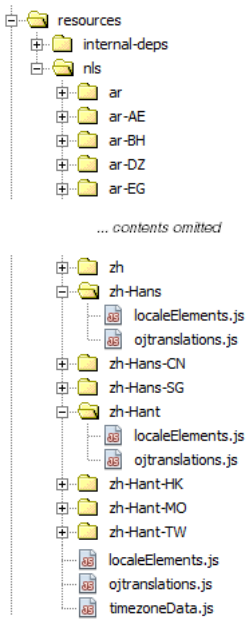
The location of the Oracle JET translation bundle, which is named `ojtranslations.js`, is in the following directory:

```
libs/oj/v19.0.0/resources/nls/ojtranslations
```

Each supported language is contained in a directory under the `nls` directory. The directory names use the following convention:

- lowercase for the language sub-tag (`zh`, `sr`, and so on.)
- title case for the script sub-tag (`Hant`, `Latn`, and so on.)
- uppercase for the region sub-tag (`HK`, `BA`, and so on.)

The language, script, and region sub-tags are separated by hyphens (-). The following image shows a portion of the directory structure.



Top-Level Module

The `ojtranslations.js` file contains the strings that Oracle JET translates and lists the languages that have translations. This is the top-level module or root bundle. In the root bundle, the strings are in English and are the runtime default values when a translation isn't available for the user's preferred language.

Translation Bundle Format

Oracle JET expects the top-level root bundle and translations to follow a specified format. The root bundle contains the Oracle JET strings with default translations and the list of locales that have translations.

```
define({
// root bundle
  root: {
    "oj-message":{
        fatal:"Fatal",
        error:"Error",
        warning:"Warning",
        info:"Info",
        confirmation:"Confirmation",
        "compact-type-summary":"{0}: {1}"
     },
```

```
    // ... contents omitted
  },

// supported locales.
  "fr-CA":1,
   ar:1,
   ro:1,
   "zh-Hant":1,
   nl:1,
   it:1,
   fr:1,
   //  ... contents omitted
   tr:1,fi:1
});
```

The strings are defined in nested JSON objects so that each string is referenced by a name with a prefix: `oj-message.fatal`, `oj-message.error`, and so on.

The language translation resource bundles contain the Oracle JET string definitions with the translated strings. For example, the following code sample shows a portion of the French (Canada) translation resource bundle, contained in `nls/fr-CA/ojtranslations.js`.

```
define({
  "oj-message":{
     fatal:"Fatale",
     error:"Erreur",
     warning:"Avertissement",
     info:"Infos",
     confirmation:"Confirmation",
     "compact-type-summary":"{0}: {1}"
     },
    //  ... contents omitted
});
```

When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. If there are no translations for the user's preferred language, the root language bundle, English, will be displayed.

Named Message Tokens

Some messages may contain values that aren't known until runtime. For example, in the message `"User foo was not found in group bar"`, the `foo` user and `bar` group is determined at runtime. In this case, you can define `{username}` and `{groupname}` as named message tokens as shown in the following code.

```
"MyUserKey":"User {username} was not found in group {groupname}."
```

At runtime, the actual values are replaced into the message at the position of the tokens by calling the `Translations.applyParameters()` method with the key of the message as the first argument and the parameters to be inserted into the translated pattern as the second argument.

```
const parMyUserKey = { username: 'Foo', groupname: 'Test' };
const tmpString = Translations.applyParameters(MenuBundle.MyUserKey,
```

```
parMyUserKey);
const transNamedMessageToken = Translations.getTranslatedString(tmpString);
```

Numeric Message Tokens

Alternatively, you can define numeric tokens instead of named tokens. For example, in the message `"This item will be available in 5 days"`, the number `5` is determined at runtime. In this case, you can define the message with a message token of `{0}` as shown in the following code.

```
"MyKey": "This item will be available in {0} days."
```

A message can have up to 10 numeric tokens. For example, the message `"Sales order {0} has {1} items"` contains two numeric tokens. When translated, the tokens can be reordered so that message token `{1}` appears before message token `{0}` in the translated string, if required by the target language grammar. The code that calls the `applyParameters()` and `getTranslatedString()` methods remains the same no matter how the tokens are reordered in the translated string.

> **Tip**
>
> Use named tokens instead of numeric tokens to improve readability and reuse.

Escape Characters in Resource Bundle Strings

The dollar sign, curly braces and square brackets require escaping if you want them to show up in the output. Add a dollar sign ($) before the characters as shown in the following table.

Escaped Form	Output
$$	$
${	{
$}	}
$[	[
$]	]

For example, if you want your output to display `[Date: {01/02/2020}, Time: {01:02 PM}, Cost: $38.99, Book Name: JET developer's guide]`, enter the following in your resource bundle string:

```
"productDetail": "$[Date: ${01/02/2020$}, Time: ${01:02 PM$}, Cost: $$38.99,
Book Name: {bookName}$]"
```

You then use the `Translations.applyParameters()` method to return the string with the escaped characters and substituted tokens, if any, to display in the UI, as shown in the following example:

```
let parEscapeCharToken = { bookName: "Oracle JET Developer's Guide" };
let EscapeCharToken =
Translations.applyParameters(MenuBundle.EscapeCharToken, parEscapeCharToken)
```

Format Translated Strings

In some situations, you may want to apply formatting to strings in the resource bundle to appear in the UI. Take, for example, a book title to which we may want to apply the `<i>` tag so that the book title renders using italics in the HTML output. In this scenario, you might define the following entry in the resource bundle(s):

```
// root bundle
"FormatTranslatedString": "The <i>{booktitle}</i> describes how to develop
Oracle JET apps"
```

One approach to render this UI string with italics in your app is to make use of the `dangerouslySetInnerHTML` prop, as in the following example:

```
// Format translated strings. Resource bundle entry for a book title includes
italics
const FormatTranslatedStringComponent = () => {
  let parBookTitle = { 'booktitle': 'Oracle JET Developer Guide' };
  const formattedString = Translations.applyParameters
                                    (MenuBundle.FormatTranslatedString,
parBookTitle);
   return (
    <span dangerouslySetInnerHTML={{ __html: formattedString }} />
   );
 };
const formattedString = FormatTranslatedStringComponent();
```

> **⚠ Caution**
>
> The dangerouslySetInnerHTML prop does not validate HTML input provided by an app for integrity or security violations. It is the app's responsibility to sanitize the input to prevent unsafe content from being added to the page.

Add RequireJS Translation Bundles to an Oracle JET App

You can add a translation bundle to your Oracle JET app with the custom strings that your app UI needs and the translations that you want your app to support.

To add translation bundles to Oracle JET:

1. Define the translations.

   For example, the following code defines a translation set for a menu containing a button label and three menu items. The default language is set to English, and the default label and menu items will be displayed in English. The root object in the file is the default resource bundle. The other properties list the supported locales, `fr`, `cs`, and `ar`.

   ```
   define({
     "root": {
       "label": "Select a department",
       "menu1": "Sales",
       "menu2": "Human Resources",
       "menu3": "Transportation"
   ```

```
},
  "fr": true,
  "cs": true,
  "ar": true
});
```

To add a prefix to the resource names (for example, `department.label`, `department.menu1`, and so on), add it to your bundles as shown below.

```
define({
  "root": {
    "department": {
      "label": "Select a department",
      "menu1": "Sales",
      "menu2": "Human Resources",
      "menu3": "Transportation"
    }
  }
},
  "fr": true,
  "cs": true,
  "ar": true
});
```

When the locale is set to a French locale, the French translation bundle is loaded. The code below shows the definition for the label and menu items in French.

```
define({
  "label": "Sélectionnez un département",
  "menu1": "Ventes",
  "menu2": "Ressources humaines",
  "menu3": "Transports"
})
```

You can also provide regional dialects for your base language bundle by just defining what you need for that dialect.

```
define({
    "label": "Canadian French message here"
});
```

When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. In this example, the menu items will be displayed using the French translations. If there are no translations for the user's preferred language, the root language bundle, whatever language it is, will be displayed.

2. Include each definition in a file located in a directory named `nls`.

   For example, the default translation in the previous step is placed in a file named `menu.js` in the `appRootDir/src/resources/nls` directory. The supported translations are located in a file named `menu.js` in child sub-directories that use the name of the locale:

```
appRootDir/src/resources/nls
|   menu.js
```

```
|
+---ar
|     menu.js
|
+---cs
|     menu.js
|
+---fr
      menu.js
```

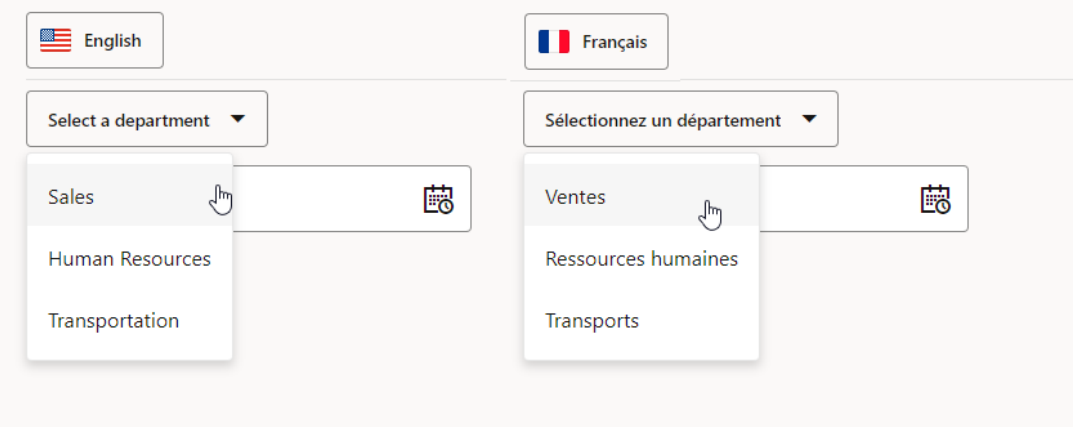
3. You first import the translation bundles where you defined the default and translated strings:

   ```
   import * as MenuBundle from "ojL10n!./resources/nls/menu";
   ```

4. You need to configure changes in your app code to make sure that it retrieves the appropriate translation.

   If, for example, you want to implement the following UI where the menu labels change to English or French depending on the locale, you need to reference `btnLocaleLabel` and `menuNames` variables, as follows:

   ```
   <oj-menu-button id="menuButton1">{btnLocaleLabel}
       <oj-menu id="myMenu1" slot="menu" onojMenuAction={changeLabel}>
         {menuNames.map((menu) => (
           <oj-option value={menu.itemName}>
             {menu.itemName}
           </oj-option>
         ))}
       </oj-menu>
         </oj-menu-button>
   ```



5. You then declare the variables that reference the imported translation bundle (`MenuBundle`) for the appropriate value to use, as demonstrated in the following code:

   ```
   <oj-menu-button id="menuButton1">{btnLocaleLabel}
                   <oj-menu id="myMenu1" slot="menu"
   onojMenuAction={changeLabel}>
                     {menuNames.map((menu) => (
                       <oj-option value={menu.itemName}>
                         {menu.itemName}
                       </oj-option>
   ```

```
                    ))}
                  </oj-menu>
                </oj-menu-button>

// Select a department:  Menu button (Sales, Human Resources, and
Transportation )
  const menuNames = [
    { itemName: MenuBundle.menu1, id: 'menu1' },
    { itemName: MenuBundle.menu2, id: 'menu2' },
    { itemName: MenuBundle.menu3, id: 'menu3' },
  ];
```

6. If the strings from your resource bundle include message tokens or reserved characters ($, {, }, [,]) that need to be escaped, you must use Oracle JET's `Translation.applyParameters` API.

   The following code demonstrates how to render characters that must be escaped and a string that requires a parameter value (also referred to as a token value).

```
/ // App UI is going to render the following strings:
$ { } [ ]
[The Oracle JET Developer's Guide costs $38.99]

// To accomplish this, we enter the following entries in the app's
resource bundle(s):
(appRootDir/src/resources/nls/menu.js)
. . .
        "EscapeChar": "$$ ${ $} $[ $]",
        "EscapeCharToken": "$[The {bookName} costs $$38.99$]"
    },
. . .

// appRootDir/src/components/content/index.tsx
// Import the Translations module that includes the applyParameters() API
import * as Translations from "ojs/ojtranslation";

// Escape characters in resource bundle strings
let parEscapeChar = {};
let EscapeChar = Translations.applyParameters(MenuBundle.EscapeChar,
parEscapeChar)

// Substitute a token and escape a character
let parEscapeCharToken = { bookName: "Oracle JET Developer's Guide" };
let EscapeCharToken =
Translations.applyParameters(MenuBundle.EscapeCharToken,
parEscapeCharToken)

return (
<div class="oj-web-applayout-max-width oj-web-applayout-content">
 <li>
   <p>Escape characters in resource bundle strings</p>
   <p>
     <span>
      {EscapeChar}
    </span>
  </p>
```

```
    </li>
    <li>
      <p>Substitute a token and escape a character</p>
      <p>
        <span>
          {EscapeCharToken}
        </span>
      </p>
    </li>
  </div>
  );
```

Work with ICU Translation Bundles in an Oracle JET Virtual DOM App

Oracle JET supports ICU message format translation bundles that enable Oracle JET apps to provide localized UI strings and support runtime substitution using placeholders.

International Components for Unicode (ICU) is a set of libraries that provides support for the internationalization of text, numbers, dates, times, currencies, and other locale-sensitive data in formatting user-visible strings (messages). To read more about how ICU formats messages, see its documentation.

To use Oracle JET's support for ICU message format translation bundles, run the Oracle JET CLI's `add translation` command in the root directory of your Oracle JET project. This installs the NPM package (`@oracle/oraclejet-icu-l10n`) that you need to generate the runtime ICU translation bundles. It also creates a `resources` directory with ICU translation bundles in your project's `src` directory with the following files to help you get started:

```
appRootDir\src\resources
\---nls
    |   translationBundle.json
    |
    \---de
    translationBundle.json
```

Finally, it updates your project's `oraclejetconfig.json` file with the following properties to facilitate generation of the runtime ICU translation bundles when you run the Oracle JET CLI's `build` or `serve` commands:

```
. . .
"translationIcuLibraries": "@oracle/oraclejet-icu-l10n",
. . .
"buildICUTranslationsBundle": true,
  "translation": {
    "type": "icu",
    "options": {
      "rootDir": "./src/resources/nls",
      . . .
      "supportedLocales": "de"
    }
  }
. . .
```

When you build your Oracle JET project, `@oracle/oraclejet-icu-l10n` parses the ICU translation bundles (for example, `translationBundle.json`) and converts them to TypeScript modules (`translationBundle.ts` using our example). These TypeScript modules are the runtime ICU translation bundles. They return an object with functions to get the localized string for a message key. The function takes parameters to format the message. For plurals, the parameter is always a number. Note that the same message can have multiple parameters (for example, a number for the plural rule and a string for a text placeholder). The parameter types will be described by TypeScript.

As you'll notice from the example ICU translation bundle that the `add translation` command creates, translation bundle files must have a `.JSON` file extension and follow the JSON format. Keys for translation bundle resources should always appear at the top level. Creating sub-objects for translated resources is not supported. Component authors may use some convention-based character like `'_'` in the key name (for example, `"input_messages_error"`) to indicate the intended usage of the resource.

```
"input_message_error": "Error",
"input_message_warning": "Warning"
```

The value is a string that may contain placeholders.

> **Note**
>
> The examples that follow are formatted as multi-line strings for readability. Actual values in an ICU translation bundle must be single-line strings, as required by the JSON specification. Enable word wrapping in your editor to view them conveniently.

Keys starting with the `@` character specify metadata. If a metadata key starts with `@@`, the metadata is considered global. Otherwise, the metadata is associated with the key whose name follows the `@` character:

```
"input_message_error": "Error: {MESSAGE}",
"@input_message_error":
   "placeholders": {
      "MESSAGE":
         "type": "text",
         "description": "translated error message"
      }
   },
   "description": "Error with an embedded message"
}
```

Metadata is optional. Supply it in the following cases:

1. The usage of the text placeholder is going to be unclear to the translator, and supplying an extra description in the metadata is going to help with translating the rest of the message.
2. A particular placeholder is not being used in the message included in the root/development bundle, but the corresponding parameter should be settable for some other locales.

   ICU translation bundle's build process derives parameter type information from the message in the root/development bundle. Type information for any parameters used only in other locales will have to come from metadata.

The reserved special characters within the translated resource are pound # and curly braces ({, }). Use the ASCII apostrophe (', U+0027) around these characters so that they appear in the message. Use double ASCII apostrophe if a single ASCII apostrophe should appear in the message, but only when it is immediately followed by a pound character or a curly brace. We recommend that you use the real apostrophe (single quote) character ’ (U+2019) for human-readable text, and use the ASCII apostrophe ' (U+0027) only as an escape character. This avoids having to use the double ASCII apostrophe.

If a particular part of a message should not be translated, provide it as a parameter for a text placeholder.

Format Placeholders Tokens

Placeholder tokens enable dynamic content generation by substituting values into translated messages.

Text

Use text placeholders to perform simple token substitution. Avoid the use of numeric placeholder names (0, 1, and so on).

```
"input_message_error": "Error: {MESSAGE}"
```

In the previous example, MESSAGE identifies a named parameter to insert into the translated string.

Plural

Plural placeholders define translated messages intended for either certain groups of numbers or for exact numbers. You can think of them as switch statements.

```
"shopping_card_items": "You have {item_count, plural, offset:0
    =0 {no items}
    one {# item}
    other {# items}
} in your cart"
```

To understand the previous example, consider the following:

- item_count is the name of the parameter whose numeric value is used to for matching the message.
- plural is the type of the placeholder.
- =0 will match exactly 0.
- one is one of the language-specific categories that are matched against the category of the parameter's value. For example, 1, 21 and 31 will all be in the category one for Ukrainian. Other categories are zero, two, few, many. Note that the match for an exact number (=1) has a higher precedence than a category-based match.
- offset is the number that will be subtracted from the parameter's value before a category-based rule is applied. The default is 0. Note that offset is not used in exact matches (=1, and so on).
- other is just like a default label in a switch statement. That is, it is a catch all category. Note that other is required in every plural placeholder.

- # will be substituted with the actual numeric value after the offset is subtracted.

Select

Select placeholders specify translated messages that will be chosen based on the value of a string parameter. Just like with the plural selector, you can think of them as a switch statement.

```
"response_message": "{gender, select,
    male {He}
    female {She}
    other {They}
    } will respond shortly."
```

In the example above, `gender` is the name of the parameter whose string value is used to match the translated message. Possible parameter values are `male` and `female`. Similar to the plural placeholder, `other` is a special category acting like a `default` label in a `switch` statement. Note that `other` is required in every select placeholder.

Nesting Select and Plural Placeholders

```
"invite_message": "{gender_of_host, select,
  female {
    {num_guests, plural, offset:1
      =0 {{host} does not give a party.}
      =1 {{host} invites {guest} to her party.}
      =2 {{host} invites {guest} and one other person to her party.}
      other {{host} invites {guest} and # other people to her party.}}}
  male {
    {num_guests, plural, offset:1
      =0 {{host} does not give a party.}
      =1 {{host} invites {guest} to his party.}
      =2 {{host} invites {guest} and one other person to his party.}
      other {{host} invites {guest} and # other people to his party.}}}
  other {
    {num_guests, plural, offset:1
      =0 {{host} does not give a party.}
      =1 {{host} invites {guest} to their party.}
      =2 {{host} invites {guest} and one other person to their party.}
      other {{host} invites {guest} and # other people to their party.}}}}"
```

In the example above, plural placeholders are nested in a select placeholder. The result is that plural rules are applied for each gender-specific group of messages. Note that the value of the offset in plural rules is used only for printing the number of guests, not counting the main guest identified by the `guest` parameter (substituting #).

Set Up an Oracle JET Virtual DOM App to Create ICU Translation Bundles

To set up an Oracle JET virtual DOM app to create and use ICU translation bundles, you run the `add translation` command and edit the properties that this command inserts in your app's `oraclejetconfig.json` file.

To set up your Oracle JET app to support ICU translation bundles:

1. Change to your app's top-level directory, open a terminal window, and enter the following command:

```
npx ojet add translation
```

2. In an editor, open the `oraclejetconfig.json` file to review and edit the properties that the `add translation` command inserted.

   Leave the value of the `type` property in the `translation` section unchanged at `icu`. This is the only supported value at present. Similarly, leave the `buildICUTranslationsBundle` property unchanged at `true`, unless you do not want to generate runtime ICU translation bundles. In that case, set it to `false`.

   The following table describes the child properties of `options`:

Property	Description
`rootDir`	The root directory for ICU translation bundles. The value that the `add translation` command inserts is `"./src/resources/nls"`.
`bundleName`	The bundle name from which to generate runtime ICU translation bundles when you run the `ojet build` or `serve` commands. The value that the `add translation` command inserts is `"translationBundle.json"`.
`locale`	The locale of the ICU translation bundle in the root directory. The value that the `add translation` command inserts is `"en-US"`.
`outDir`	The output directory for runtime ICU translation bundles. The value that the `add translation` command inserts is `"./src/resources/nls"`. Change this to a value of your choice. For example, `"./src/resources-dist"`.
`supportedLocales`	A comma-separated list of additional locales to build. If you specify a locale that does not have a corresponding entry in the `nls` directory under the `rootDir` directory, then the runtime ICU translation bundle is built from the ICU translation bundle in the `rootDir` directory. The value that the `add translation` command inserts is `"de"`.
`componentBundleName`	Specifies the naming pattern for the ICU translation bundle associated with a component. The `add translation` command inserts the value `"${componentName}-strings.json"` where `{componentName}` references the name of the component. If you create a component named `my-component`, then the file name for the ICU translation bundle must be `my-component-strings.json`.

Add ICU Translation Bundles to an Oracle JET Virtual DOM App

Add ICU translation bundles to your Oracle JET virtual DOM app with the custom strings that your app UI needs and the translations that you want your app to support.

To add ICU translation bundles to your Oracle JET app:

1. Define the UI strings and translations in `.JSON` files with a flat key-value structure.

   Each key must be at the top level. Do not nest objects for grouping.

   For example, the following entry, generated by default when you run the `add translation` command, defines a greeting message.

```
{
  "greeting": "Hello! How are you doing?"
}
```

Keys prefixed with `@` provide optional metadata, such as descriptions and parameter types. This metadata helps when placeholders are not self-explanatory or when required only for certain locales.

2. Include each definition in a `.JSON` file located in a directory named `nls`.

   For example, the translation in the previous step is placed in a file named `translationBundle.json` in the `appRootDir/src/resources/nls` directory. The supported translations are in files named `translationBundle.json` in child sub-directories that use the name of the locale:

```
appRootDir/src/resources/nls
|   translationBundle.json
|
/---de
    translationBundle.json
```

Generate Runtime ICU Translation Bundles

Once you have created the `.JSON` files with the UI strings for the locales that your Oracle JET virtual DOM app supports, you need to build the runtime ICU translation bundles to a location from where your app retrieves the UI strings at runtime.

Change to your app's top-level directory, open a terminal window, and enter the following command:

```
npx ojet build
```

This creates the runtime ICU translation bundles (`.TS` files) and the `supportedLocales.ts` file in the directory specified by the `outDir` property in the `oraclejetconfig.json` file. By default, it is `./src/resources/nls`.

```
appRootDir/src/resources
\---nls
    |   supportedLocales.ts
    |   translationBundle.json
    |   translationBundle.ts
    |
    +---de
    |       translationBundle.json
    |       translationBundle.ts
```

Now that you have generated the runtime ICU translation bundles, you can use the UI strings that they contain in your Oracle JET virtual DOM app and in your VComponents.

Use an ICU Translation Bundle in an Oracle JET Virtual DOM App Component

Once you generate the runtime ICU translation bundles, you use the UI strings from the generated bundles.

A runtime ICU translation bundle defines an object with function(s) to get the UI string value for the specified message key. The function can take parameters to format the message. For plurals, the parameter is always a number.

> **Note**
>
> The same message can have multiple parameters. For example, a number for the plural rule and a string for a text placeholder. TypeScript describes the parameter types.

Your app dynamically loads the translation bundles like any other Typescript module. You first determine what locale your app uses. Once you know the desired locale, you find the best match out of the supported locales that the `./appRootDir/src/resources-dist/supportedLocales.ts` file lists. For example, the `add translation` command adds support for `de` by default:

```
export default ["de"];
```

The following code sample demonstrates how to load the UI string for the `label-hint` attribute value of the `oj-input-text` element in the `appRootDir/src/components/content/index.tsx` file of a virtual DOM app. It first imports the runtime ICU translation bundle and the supported locales. A Preact `useEffect` hook loads the imported ICU translation bundles and if one is found to match the preferred locale, it is used. If not, the ICU translation bundle for the German locale is used (`de`).

```
import { h } from "preact";
import { BundleType as ICUBundleType } from "../../resources/nls/
translationBundle";
import supportedLocales from "../../resources/nls/supportedLocales";
import { useState, useEffect } from "preact/hooks";
import "ojs/ojlabel";
import "ojs/ojinputtext";
import "ojs/ojformlayout";

export function Content() {
  const [ICUBundle, setICUBundle] = useState<ICUBundleType>();
  // Load runtime ICU translation bundle in Preact's useEffect hook.
  useEffect(() => {
    _loadTranslationBundle().then((bundle) => setICUBundle(bundle));
  }, []);

  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <div id="icu-app">
      <p>
      Render UI string from the <code>./src/resources/nls/de</code> runtime
      ICU translation bundle.{" "}
      </p>
        {ICUBundle && (
          <oj-form-layout id="ofl1" max-columns="1" direction="column">
            <oj-input-text
              id="input"
              value=" "
              labelHint={ICUBundle.greeting()}
              labelEdge="inside"
            ></oj-input-text>
          </oj-form-layout>
        )}
```

```
      </div>
    </div>
  );
}

async function _loadTranslationBundle(): Promise<ICUBundleType> {
  const preferredLocale = navigator.languages[0];
  const localeToLoad = _matchTranslationBundle(
    preferredLocale,
    supportedLocales
  );

 // The ICU translation bundle name (translationBundle in this example)
derives
 // its name from the value that you specify for the bundleName property
 // in the oraclejetconfig.json file.
  const module = await import(
    `../../resources/nls/${localeToLoad}/translationBundle`
  );
  return module.default;
}

function _matchTranslationBundle(
  preferredLocale: string,
  supportedLocales: string[]
) {
  let match = null;
  const supportedLocaleSet = new Set(supportedLocales);

  if (supportedLocaleSet.has(preferredLocale)) {
    match = preferredLocale;
  } else {
    match = _findPartialMatch(preferredLocale, supportedLocaleSet);
  }
  // Normally, the fallback locale would be 'en-US', but for
  // this example, we're using 'de'.
  return match || 'de';
}

function _findPartialMatch(locale: string, supportedLocales: Set<string>) {
  let match = null;
  const sep = "-";
  const parts = locale.split(sep);

  while (match === null && parts.length > 1) {
    parts.pop();
    const partial = parts.join(sep);
    if (supportedLocales.has(partial)) {
      match = partial;
    }
  }
  return match;
}
```

Use an ICU Translation Bundle in an Oracle JET VComponent

The steps to create and use ICU translation bundles in a VComponent are the same as those for a virtual DOM app. That is, you create ICU translation bundle source files (`.JSON`) in a directory location that matches the pattern specified by the `rootDir` property in the `oraclejetconfig.json` file. By default, this is `"./src/resources/nls"`. When you build the Oracle JET project that contains the VComponent, the runtime ICU translation bundles are generated to the location specified by the `outDir` property. Again, the default is `"./src/resources/nls"`. One additional property in the `oraclejetconfig.json` file to be aware of is `componentBundleName`. This specifies the naming pattern for the VComponent's ICU translation bundle so that Oracle JET can generate the runtime ICU translation bundle. Its default value is `"${componentName}-strings.json"`.

So, assume for example, that you create a VComponent named `translation-icu` in your Oracle JET virtual DOM app:

```
npx ojet create component translation-icu --vcomponent
```

This leads to the creation of a `translation-icu` directory within your virtual DOM app's project file and this directory includes `resources` and `nls` sub-directories as follows:

```
appRootDir/src/components/translation-icu/resources/nls
```

Within the `nls` sub-directory, you create the ICU translation bundles as needed for the locales that you are going to support:

```
appRootDir/src/components/translation-icu/resources/nls
|   translation-icu-strings.json
+---de
|       translation-icu-strings.json
```

Once you have generated the runtime ICU translation bundles by building the Oracle JET virtual DOM app that contains the VComponent, you can use UI strings from the runtime ICU translation bundle in your VComponent. The following code snippets demonstrate how you import and reference a UI string from the ICU translation bundle. For brevity, the full VComponent code and helper functions that are unchanged from the code sample in the previous section for the virtual DOM app have been omitted.

```
. . .
import "css!./translation-icu-styles.css";
import { useState, useEffect } from "preact/hooks";
import "ojs/ojinputtext";
import "ojs/ojformlayout";
import { BundleType as ICUBundleType } from "./resources/nls/translation-icu-
strings";
import supportedLocales from "./resources/nls/supportedLocales";

. . .

function TranslationIcuImpl() {

  const [ICUBundle, setICUBundle] = useState<ICUBundleType>();

  // Load ICU translation bundle in Preact's useEffect hook.
```

```
  useEffect(() => {
    _loadTranslationBundle().then((bundle) => setICUBundle(bundle));
  }, []);

  return (
    <div>
      <p>Render the string from the ICU translation bundle</p>
      {ICUBundle ? (
        <oj-form-layout id="ofl1" max-columns="1" direction="column">
          <oj-input-text
            id="input"
            value=" "
            labelHint={ICUBundle.greeting()}
            labelEdge="inside"
          ></oj-input-text>
        </oj-form-layout>
      ) : (
        <p>ICU translation bundle did not load</p>
      )}
    </div>
  );
}

// Helper functions for ICU translation bundle
async function _loadTranslationBundle(): Promise<ICUBundleType> {
  const preferredLocale = navigator.languages[0];
  const localeToLoad = _matchTranslationBundle(
    preferredLocale,
    supportedLocales
  );

// The component ICU translation bundle name
// (translation-icu-strings in this example) derives
// its name from the VComponent name and other values
// that you specify for the componentBundleName property
// in the oraclejetconfig.json file.
  const module = await import(
    `./resources/nls/${localeToLoad}/translation-icu-strings`
  );
  return module.default;
}

function _matchTranslationBundle(
. . .
// Omitted for brevity. See previous section that includes full code
// for this function

function _findPartialMatch(locale: string, supportedLocales: Set<string>) {
. . .
// Omitted for brevity. See previous section that includes full code
// for this function

export const TranslationIcu: ComponentType<ExtendGlobalProps<
  ComponentProps<typeof TranslationIcuImpl>
  >> = registerCustomElement("translation-icu", TranslationIcuImpl);
```

9

Test and Debug Oracle JET Apps

Test and debug Oracle JET web apps using a recommended set of testing and debugging tools for client-side apps.

Test Oracle JET Apps

Tests help you build complex Oracle JET apps quickly and reliably by preventing regressions and encouraging you to create apps that are composed of testable functions, modules, classes, and components.

We recommend that you write tests as early as possible in your app's development cycle. The longer that you delay testing, the more dependencies the app is likely to have, and the more difficult it will be to begin testing.

Testing Types

There are three main testing types that you should consider when testing Oracle JET apps.

1. Unit Testing
   - Unit testing checks that all inputs to a given function or class produce the expected output or response.
   - These tests typically apply to self-contained business logic, classes, modules, or functions that do not involve UI rendering, network requests, or other environmental concerns.

     Note that REST service APIs should be tested independently.
   - Unit tests are aware of the implementation details and dependencies of a given function or class and focus on isolating the tested function or class.
2. Component Testing
   - Component testing checks that individual components can be interacted with and behave as expected. These tests import more code than unit tests, are more complex, and require more time to execute.
   - Component tests should catch issues related to your component's properties, events, the slots that it provides, styles, classes, lifecycle hooks, and more.
   - These tests are unaware of the implementation details of a component; they mock up as little as possible in order to test the integration of your component and the entire system.

     You should not mock up child components in component tests but instead check the interactions between your component and its children with a test that interacts with the components as a user would (for example, by clicking on an element).
3. End-to-End Testing
   - End-to-end testing, which often involves setting up a database or other backend service, checks features that span multiple pages and make real network requests against a production-built JET app.

End-to-end testing is meant to test the functionality of an entire app, not just its individual components. Therefore, use component tests when testing specific components of your Oracle JET apps.

Unit Testing

Unit testing should be the first and most comprehensive form of testing that you perform.

The purpose of unit testing is to ensure that each unit of software code is coded correctly, works as expected, and returns the expected outputs for all relevant inputs. A unit can be a function, method, module, object, or other entity in an app's source code.

Unit tests are small, efficient tests created to execute and verify the lowest-level of code and to test those individual entities in isolation. By isolating functionality, we remove external dependencies that aren't relevant to the unit being tested and increase the visibility into the source of failures.

Unit tests that you create should adhere to the following principles:

- **Easy to write:** Unit testing should be your main testing focus; therefore, tests should typically be easy to write because many will be written. The standard testing technology stack combined with recommended development environments ensures that the tests are easily and quickly written.
- **Readable:** The intent of each test should be clearly documented, not just in comments, but the code should also allow for easy interpretation of what its purpose is. Keeping tests readable is important should someone need to debug when a failure occurs.
- **Reliable:** Tests should consistently pass when no bugs are introduced into the component code and only fail when there are true bugs or new, unimplemented behaviors. The tests should also execute reliably regardless of the order in which they're run.
- **Fast:** Tests should be able to execute quickly and report issues immediately to the developer. If a test runs slowly, it could be a sign that it is dependent upon an external system or interacting with an external system.
- **Discrete:** Tests should exercise the smallest unit of work possible, not only to ensure that all units are properly verified but also to aid in the detection of bugs when failures occur. In each unit test, individual test cases should independently target a single attribute of the code to be verified.
- **Independent:** Above all else, unit tests should be independent of one another, free of external dependencies, and be able to run consistently irrespective of the environment in which they're executed.

To shield unit tests from external changes that may affect their outcomes, unit tests focus solely on verifying code that is wholly owned by the component and avoid verifying the behaviors of anything external to that component. When external dependencies are needed, consider using mocks to stand in their place.

Component Testing

The purpose of component testing is to establish that an individual component behaves and can be interacted with according to its specifications. In addition to verifying that your component accepts the correct inputs and produces the right outputs, component tests also include checking for issues related to your component's properties, events, slots, styles, classes, lifecycle hooks, and so on.

A component is made up of many units of code, therefore component testing is more complex and takes longer to conduct than unit testing. However, it is still very necessary; the individual

units within your component may work on their own, but issues can occur when you use them together.

Component testing is a form of closed-box testing, meaning that the test evaluates the behavior of the program without considering the details of the underlying code. You should begin testing a component in its entirety immediately after development, though the tested component may in part depend on other components that have not yet been developed. Depending on the development lifecycle model, component testing can be done in isolation from other components in the system, in order to prevent external influences.

If the components that your component depends on have not yet been developed, then use dummy objects instead of the real components. These dummy objects are the stub (called function) and the controller (called function).

Depending on the depth of the test level, there are two types of component tests: small component tests and large component tests.

When component testing is done in isolation from other components, it is called "small component testing." Small component tests do not consider the component's integration with other components.

When component testing is performed without isolating the component from other components, it is called "large component testing", or "component testing" in general. These tests are done when there is a dependency on the flow of functionality of the components, and therefore we cannot isolate them.

End-to-End Testing

End-to-end testing is a method of evaluating a software product by examining its behavior from start to finish. This approach verifies that the app operates as intended and confirms that all integrated components function correctly in relation to one another. Additionally, end-to-end testing defines the system dependencies of the product to ensure optimal performance.

The primary goal of end-to-end testing is to replicate the end-user experience by simulating real-world scenarios and evaluating the system and its components for proper integration and data consistency. This approach allows for the validation of the system's performance from the perspective of the user.

End-to-end testing is a widely adopted and reliable technique that provides the following advantages.

- Comprehensive test coverage
- Assurance of app's accuracy
- Faster time to market
- Reduced costs
- Identification of bugs

Modern software systems are increasingly interconnected, with various subsystems that can cause adverse effects throughout the entire system if they fail. End-to-end testing can help prevent these risks by:

- Verifying the system's flow
- Increasing the coverage of testing areas
- Identifying issues related to subsystems

End-to-end testing is beneficial for a variety of stakeholders:

- Developers appreciate end-to-end testing as it allows them to offload testing responsibilities.
- Testers find it useful as it enables them to write tests that simulate real-world scenarios and avoid potential problems.
- Managers benefit from end-to-end testing as it allows them to understand the impact of a failing test on the end-user.

The end-to-end testing process comprises four stages:

1. **Test Planning:** Outlining key tasks, schedules, and resources required
2. **Test Design:** Creating test specifications, identifying test cases, assessing risks, analyzing usage, and scheduling tests
3. **Test Execution:** Carrying out the test cases and documenting the results
4. **Results Analysis:** Reviewing the test results, evaluating the testing process, and conducting further testing as required

There are two approaches to end-to-end testing:

- **Horizontal Testing:** This method involves testing across multiple apps and is often used in a single ERP (Enterprise Resource Planning) system.
- **Vertical Testing:** This approach involves testing in layers, where tests are conducted in a sequential, hierarchical order. This method is used to test critical components of a complex computing system and does not typically involve users or interfaces.

End-to-end testing is typically performed on finished products and systems, with each review serving as a test of the completed system. If the system does not produce the expected output or if a problem is detected, a second test will be conducted. In this case, the team will need to record and analyze the data to determine the source of the issue, fix it, and retest.

While testing your app end-to-end, consider the following metrics:

- **Test Case Preparation Status:** This metric is used to track the progress of test cases that are currently being prepared in comparison to the planned test cases.
- **Test Progress Tracking:** Regular monitoring of test progress on a weekly basis to provide updates on test completion percentage and the status of passed/failed, executed/unexecuted, and valid/invalid test cases.
- **Defects Status and Details:** Provides a weekly percentage of open and closed defects and a breakdown of defects by severity and priority.
- **Environment Availability:** Information on the number of operational hours and hours scheduled for testing each day.

About the Oracle JET Testing Technology Stack

The recommended stack for testing Oracle JET apps includes Jest and the Preact Testing Library.

Jest is a popular testing framework for JavaScript/Typescript that comes with its own test runner and assertion functions. It supports code coverage and snapshot testing, is simple to create mocks with, and runs tests in parallel, which ensures that they remain isolated.

Jest runs in NodeJS using `jsdom` as a simulated browser environment. These tests run very quickly because the environment doesn't need to render anything; it is a lightweight, in-memory implementation of the DOM that runs headless. Jest tests are suitable for verifying almost every aspect of your component class, except for things that require CSS for styling. `jsdom` does not process CSS, so avoid using these tests to validate any CSS. It is also not suitable

for testing the custom element rendering of a component in your app, as the browser environment from `jsdom` is a simulation rather than a real browser.

The Preact Testing Library provides a set of utility functions that make it easy to write tests that assert the behavior of Preact component classes without relying on their implementation details. It promotes a UI-centric approach to testing: component classes are allowed to go through their full rendering lifecycles, and the library provides query functions to locate elements within the DOM and a user-event simulation library to interact with them.

The functions in the Preact Testing Library work with the actual DOM elements that are rendered by Preact, rather than with the virtual DOM, so tests will resemble how a user interacts with the app and finds elements on the page.

For UI automation testing, we recommend using Selenium WebDriver in conjunction with the Oracle® JavaScript Extension Toolkit (Oracle JET) WebDriver.

Configure Oracle JET Apps for Testing

Use the Oracle JET CLI's `add testing` command to add and configure the dependencies and libraries, including Jest and the Preact Testing Library, to test Oracle JET virtual DOM apps and VComponent components.

After you run this command in your app, you can create and run tests for your app using Jest and the Preact Testing Library. The `add testing` command adds the `test-config` directory to your app's root directory. This directory contains two files that are required to set up testing: `jest.config.js` and `testSetup.ts`. The `testSetup.ts` file imports runtime support for compiled and transpiled `async` functions, while `jest.config.js` is Jest's configuration file.

The extension for files containing tests, known as "spec files," should be `.spec.tsx` so that Oracle JET tooling and your Jest testing configuration recognize them. If spec files are missing from a component, the `add testing` command creates them. Spec files are located within a `__tests__` directory inside the `components` directory, such as `./src/components/<component-name>/__tests__`. This directory holds the test files you write for your component and, by default, the `add testing` command creates a spec file with the following file name pattern: `<component-name>.spec.tsx`.

> **Note**
>
> If you create a new component or JET pack from the command line after running the `add testing` command on your project, then the `__tests__` directory and are spec file are created default when you create the component.

In the following steps, we demonstrate how to set up an Oracle JET app for testing, and how to run a unit test and a component test.

1. Open a terminal window in a directory of your choice to create a new Oracle JET app using the `basic` template:

   ```
   npx @oracle/ojet-cli create vdomTestApp --template=basic --vdom
   ```

2. Navigate to the newly created app's root directory and create a new VComponent component:

   ```
   cd vdomTestApp
   npx ojet create component hello-world
   ```

3. Open the `./vdomTestApp/src/components/content/index.tsx` file to import and display the newly-created `HelloWorld` component:

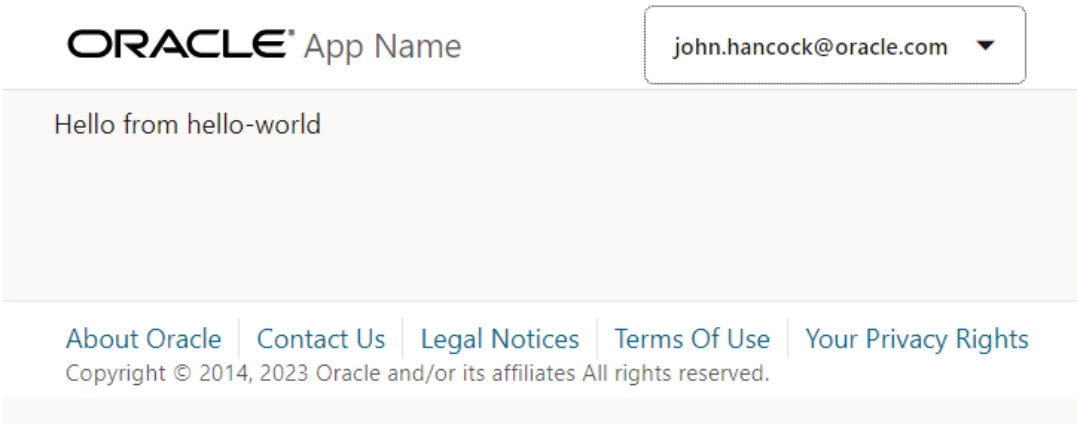
```
import { HelloWorld } from "hello-world/loader"

export function Content() {
  return (
    <div class="oj-web-applayout-max-width oj-web-applayout-content">
      <HelloWorld />
    </div>
  );
};
```

> **Note**
>
> We render the `HelloWorld` component using the Preact component class syntax (`import {HelloWorld} . . . and <HelloWorld />`) because Jest cannot test the custom element alternative (`import "hello-world/loader" . . . and <hello-world></hello-world>`).

4. To run the app in your browser and confirm that the app renders the newly-created `HelloWorld` component, enter the following command in the terminal window for the app's root directory:

```
npx ojet serve
```

This is also a required step for testing. Before you can test components, you must build the component using the Oracle JET CLI.



> **Note**
>
> The `HelloWorld` component passes "Hello from hello world" to the app through its `message` property. This confirms that the app and component function as expected.

5. In the terminal window of your app's root directory, add the testing libraries:

```
npx ojet add testing
```

This adds the `test-config` directory to the root directory of your app and the `__tests__` directory to the `component` directory (`./src/components/hello-world`). The `__tests__` directory contains a spec file (`hello-world.spec.tsx`) with a component test for your `HelloWorld` component that verifies that it renders. Jest provides the test case and assertions (`describe()`, `test()`, and `expect(true).not.toBeUndefined;`), whereas the `render()` function from the Preact Testing Library tests that the component renders in the app.

```
./vdomTest/test-config
jest.config.js
testSetup.ts
```

The `add testing` command also updates the `package.json` file with the testing dependencies, such as the Jest preset that allows Oracle JET Web Elements to be used in Jest tests, and two convenience scripts, `test` and `test:debug`.

6. To provide a function for a unit test to test, open the `./VDOMTestApp/src/components/hello-world/hello-world.tsx` file for the `HelloWorld` component that we created previously and add the `sum` function at the end of the file:

```
. . .
>> = registerCustomElement("hello-world", HelloWorldImpl);

// function for doc example
export const sum = (a: number, b: number) => {
  return a + b;
};
```

7. To write a unit test for this function, open the `./src/components/hello-world/__tests__/hello-world.spec.tsx` file and add the following entries:

```
import { render } from "@testing-library/preact";
import { HelloWorld } from "hello-world/hello-world";
import { sum } from "hello-world/hello-world";

describe("Test description", () => {
  test("Your test title", async () => {
    const content = render(
      <div data-oj-binding-provider="preact">
        <HelloWorld />
      </div>
    );
    expect(true).not.toBeUndefined;
  });

  it("The sum is 10", () => {
    expect(sum(6, 4)).toBe(10);
  });
});
```

8. In the terminal window of your app's root directory, enter the following commands to build the Oracle JET app and execute the tests:

```
npx ojet build
npm run test
```

The terminal window displays the test results:

```
vdomTest
$ npm run test

> vdomTestJET15@1.0.0 test
> jest -c test-config/jest.config.js
vdomTest
$ npm run test

> vdomTestJET15@1.0.0 test
> jest -c test-config/jest.config.js

PASS  src/components/hello-world/__tests__/hello-world.spec.tsx
  Test description
    √ Your test title (5
ms)

    √ The sum is 10 (1
ms)



Test Suites: 1 passed, 1
total

Tests:       2 passed, 2
total

Snapshots:   0 total
Time:        1.966 s, estimated 3 s
Ran all test suites.


PASS  src/components/hello-world/__tests__/hello-world.spec.tsx
  Test description
    √ Your test title (5
ms)

    √ The sum is 10 (1
ms)



Test Suites: 1 passed, 1
total
```

```
Tests:       2 passed, 2
total

Snapshots:   0 total
Time:        1.966 s, estimated 3 s
Ran all test suites.
```

Debug Oracle JET Apps

Since Oracle JET web apps are client-side HTML5 apps written in JavaScript or Typescript, you can use your favorite browser's debugging facilities.

Debug Web Apps

Use your source code editor and browser's developer tools to debug your Oracle JET app.

Developer tools for widely used browsers like Chrome, Edge, and Firefox provide a range of features that assist you in inspecting and debugging your Oracle JET app as it runs in the browser. Read more about the usage of these developer tools in the documentation for your browser.

By default, the `ojet build` and `ojet serve` commands use debug versions of the Oracle JET libraries. If you build or serve your Oracle JET app in release mode (by appending the `--release` parameter to the `ojet build` or `ojet serve` command), your app uses minified versions of the Oracle JET libraries. If you choose to debug an Oracle JET app that you built in release mode, you can use the `--optimize=none` parameter to make the minified output more readable by preserving line breaks and white space:

```
ojet build --release --optimize=none
ojet serve --release --optimize=none
```

Note that browser developer tools offer the option to "pretty print" minified source files to make them more readable, if you choose not to use the `--optimize=none` parameter.

One other way to improve the debugging experience is to set the `generateSourceMaps` in the `oraclejetconfig.json` file to `true` from its default value of `false`. When `true`, the Oracle JET CLI configures Terser and RequireJS packages to generate source map files when you build your Oracle JET app.

You may also be able to install browser extensions that further assist you in debugging your app.

Finally, if you use a source code editor, such as Visual Studio Code, familiarize yourself with the debugging tools that it provides to assist you as develop and debug your Oracle JET app.

Use Preact Developer Tools

You can install a Preact browser extension to provide additional debugging tools in your browser’s developer tools when you debug your virtual DOM app.

Preact provides download links for the various browser extensions at https://preactjs.github.io/preact-devtools/.

Once you have installed the extension for your browser, you need to include an import statement for `preact/debug` as the first line in your app's `appRootDir/src/index.ts` file:

```
import 'preact/debug';
import './components/app';
```

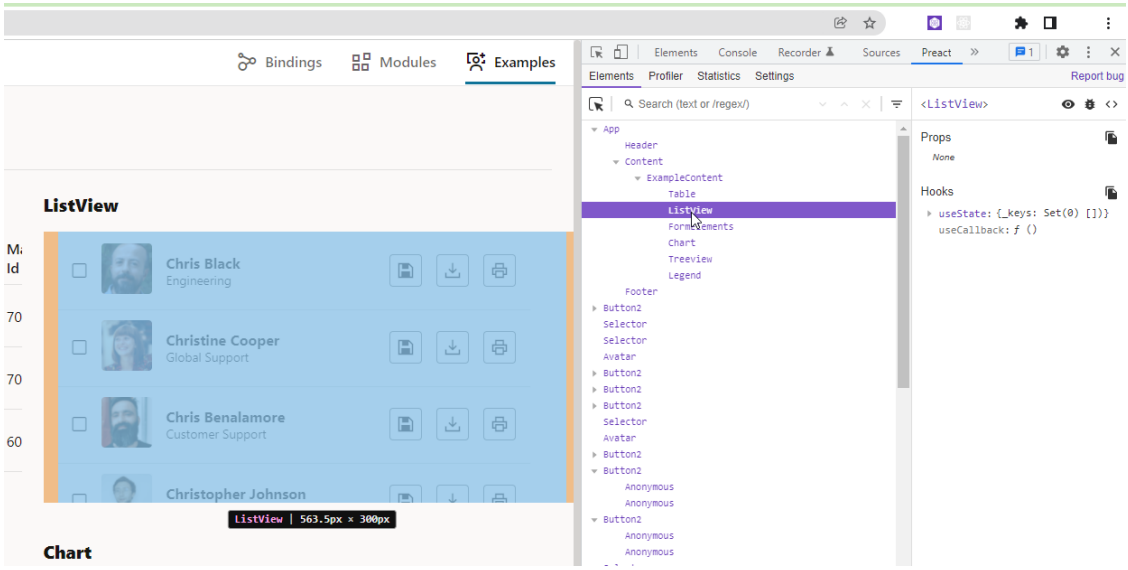
Oracle JET takes care of including this import when your virtual DOM app is built or served in debug mode (the default option for `ojet build` and `ojet serve`) by including the following injector token when you create the virtual DOM app:

```
// injector:preactDebugImport
// endinjector
import './components/app';
```

As a result, you do not need to include the `import 'preact/debug'` statement when you build or serve your app in debug mode or remove it when you build or serve for release, as Oracle JET's injector token ensures that the import statement is only included in debug mode.

When you serve your virtual DOM app in debug mode (the default option for `ojet serve`), you'll see an extra tab, Preact, in your browser's developer tools. In the following image, you see the Preact tab in the Chrome browser's DevTools.

You can view the hierarchy of the components, select and inspect components, and perform other actions that assist you in debugging issues with your virtual DOM app.



When you build or serve the virtual DOM app in release mode, using the `--release` argument, Oracle JET does not import the Preact DevTools. Remove the token if you do not want to use the Preact DevTools in debug mode.

One other thing to note is that Oracle JET includes the following entries in your app's `appRootDir/src/path_mapping.json` file when it creates your app. You need these entries to be able to use the Preact extension discussed here.

```
. . .
"preact/debug": {
      "cdn": "3rdparty",
      "cwd": "node_modules/preact/debug/dist",
      "debug": {
```

```
      "src": [
        "debug.umd.js",
        "debug.umd.js.map"
      ],
      "path": "libs/preact/debug/dist/debug.umd.js",
      "cdnPath": "preact/debug/dist/debug.umd"
    },
    "release": {
      "src": [
        "debug.umd.js",
        "debug.umd.js.map"
      ],
      "path": "libs/preact/debug/dist/debug.umd.js",
      "cdnPath": "preact/debug/dist/debug.umd"
    }
  },
  "preact/devtools": {
    "cdn": "3rdparty",
    "cwd": "node_modules/preact/devtools/dist",
    "debug": {
      "src": [
        "devtools.umd.js",
        "devtools.umd.js.map"
      ],
      "path": "libs/preact/devtools/dist/devtools.umd.js",
      "cdnPath": "preact/devtools/dist/devtools.umd"
    },
    "release": {
      "src": [
        "devtools.umd.js",
        "devtools.umd.js.map"
      ],
      "path": "libs/preact/devtools/dist/devtools.umd.js",
      "cdnPath": "preact/devtools/dist/devtools.umd"
    }
  },
```

A

Migrate Oracle JET Legacy Components to Core Pack Components

You can use the Oracle JET Core Pack migrator to migrate Oracle JET legacy components (`oj-*`) to their equivalent Core Pack components (`oj-c-*`) without breaking your app's functionality. It also migrates Selenium WebDriver test files written for legacy components to work with the migrated Core Pack components.
The Core Pack migrator is packaged with the Oracle JET Audit Framework (Oracle JAF). To begin the Core Pack migration workflow, first download Oracle JAF and run audits on your app. Then use the information in the audit report to prepare your app's code for migration; there are specific issues that must be manually addressed in your app because they will either cause the migration to fail or be ignored by the migrator.

Once you've assessed your code and prepared it for migration, you can perform a dry-run migration that generates a `migrator.log` file where you can review the migrator's intended changes to your code.

You also have the option to create a migration configuration file that will tailor the behavior of the Core Pack migrator. You can run the migrator using its default settings, but you can use the `migrationConfig.json` file to modify its operations so that, for example, it migrates only test files or excludes certain components from migration.

When you are satisfied with your pre-migration code and the results of the dry run, you can perform the full migration.

> **Note**
>
> Not all migration can be automated, and some components may need manual correction after migration in order to function. For a list of known migration issues and help resolving them, see Troubleshoot Core Pack Migration Issues.

After you successfully migrate your app's legacy components to Core Pack, you can migrate the components in its Selenium WebDriver test files. See Migrate WebDriver Test Files to Core Pack.

For information on installing and using Oracle JAF, see Install the Oracle JET Audit Framework. You can verify what versions of Oracle JAF and the Core Pack migrator are installed by running the commands `ojaf -v` and `ojaf -mig -v` in a terminal window.

Prepare Your Oracle JET App for Core Pack

Before you use the Oracle JET Core Pack migrator on your app's code or test files, run an Oracle JAF audit to inspect for issues that might impact migration. For example, deprecated APIs are not available in Core Pack components and will cause your migration to fail if they are present in your code.

Oracle JAF's built-in audit rules assist you in identifying and fixing invalid functionality.

Additionally, in the Oracle JET API documentation, there is a section labeled “Migration” for each legacy component that has a Core Pack equivalent to which you can migrate. This section facilitates the migration of a given legacy component to Core Pack by providing specific guidance and information on the differences between the two component versions. See, for example, the oj-button component’s migration section. Only legacy components with an equivalent Core Pack component are eligible for migration.

> **Note**
>
> If you don’t see the API documentation for superseded components on the API page, you can select the “Show Maintenance Mode” checkbox in the sidebar to reveal it.

To prepare your app for migration, install the latest version of Oracle JAF and run an audit on your code:

1. Open a terminal window and enter the following command to install Oracle JAF globally:

   ```
   npm install -g @oracle/oraclejet-audit
   ```

2. After you install Oracle JAF, navigate in your terminal window to your app’s root directory, then initialize a default JAF configuration:

   ```
   ojaf --init
   ```

   Oracle JAF tooling scaffolds a default JAF configuration file named `oraclejafconfig.json` in your app’s root directory.

3. In your terminal window, navigate to your app’s `./src` directory and enter the command to run an audit:

   ```
   ojaf
   ```

The audit generates a report in your terminal window. Using the report, resolve any issues flagged in your code. For example, you should fix issues raised that include the word “deprecated.” Deprecated APIs are not available in Core Pack components, and your migration will fail if your components use a deprecated API.

> **Note**
>
> Oracle JET Core Pack components do not support the Alta theme. If your app uses Alta, then you must migrate to Redwood before you can use these components in your app. See Oracle JET App Migration to Current Release.

After auditing your code and verifying that all issues have been resolved, you can proceed with your migration.

Migrate Legacy Components Using the Core Pack Migrator

After preparing your app for migration, perform a dry-run migration to understand the migrator’s intended changes to your code and find any issues that should be addressed before or after migration. If there are aspects of the migrator’s behavior that you would like to modify, you can create a migration configuration file that you can use to customize its operations.

Follow the steps in the migration procedure below as a guide.

1. Perform a dry-run migration. The dry-run migration does not update or migrate code but generates a detailed log of all code that would get migrated.

   Open a terminal window in your app's `./src` directory and enter the command `ojaf -mig -dr`.

   A message will appear once the dry run finishes, which provides information on how long the migration took and how many files and components were updated. It also points you to the `migrator.log` file, which was created in the app's `./src` directory and contains details on the files and components that would be affected by migration.

2. Review the results of the dry run in the `migrator.log` file and decide whether to use the migrator's default settings or to modify its behavior using a migration configuration file.

   To learn more about the migration configuration file, see Customize the Core Pack Migrator's Behavior.

   If you create or modify a migration configuration file, then you can test out the changes to the migrator in a dry run by executing the following command in a terminal window, replacing the path with the location of your own configuration file: `ojaf -mig -dr -c ./migrationConfig.json`

3. When you're satisfied with your code and migration configuration, run the migration command from a terminal window in your app's `./src` directory.

   - If you have modified the configuration file, run the following command with the configuration option pointing to the location of your configuration file:

     ```
     ojaf -mig -c ./migrationConfig.json
     ```

   - To run the migrator with its default configuration, execute the following command:

     ```
     ojaf -mig
     ```

Customize the Core Pack Migrator's Behavior

You can fine-tune how the Core Pack migrator operates by using a migration configuration file. The `migrationConfig.json` file supports a range of properties that allow you to configure how the Core Pack migrator behaves. For example, you can specify that the migration only affects certain components or subsets of components, rather than your entire app.

The migration configuration file contains properties that you can use to specify:

- If your migration is for only WebDriver test files and what extension they use.
- What folders, files, and components should be skipped during migration or, conversely, should be the only targets of migration.
- Whether to force the migration of specific legacy components to Core Pack.
- Whether to exclude from migration any components with tags that contain attributes with an expression as a value.
- The base folder where the migration will execute.
- The name and location of the migrator log file.

Any name can be provided to the `migrationConfig.json` file. Therefore, a user can have different sets of migration rules defined by multiple configuration files.

To begin, create the `migrationConfig.json` file in your chosen directory, typically in the directory from which you will run the migrator, such as your app's `./src` directory.

The following sample `migrationConfig.json` file displays the available configuration options that you can include.

```
{
  "testFiles": "",
  "testFilesExtension": "",
  "excludeFolders": [],
  "excludeFiles": [],
  "excludeComponents": ["legacyTag1", "legacyTag2"],
  "includeFiles": [],
  "includeFolders": [],
  "includeComponents": ["legacyTag1", "legacyTag2"],
  "forceMigration": ["legacyTag1", "legacyTag2"],
  "excludeAttributeExpressions": {
    "legacyTag1": ["attributes"],
    "legacyTag2": ["attributes"]
  },
  "baseMigrationDir": "",
  "logFileName": "",
  "logFilePath": ""
}
```

Note that all configuration settings are optional; you can choose to include only those needed for your migration.

The following table describes the available migration configuration options and how to use them.

Table A-1 Properties in the migrationConfig.json File

Property Name	Value Type	Description
`testFiles`	Boolean	Set this value to `true` if you intend to migrate only WebDriver test files and the directory you will migrate contains only test files.
`testFilesExtension`	String	Tells the migrator the specific extension it can use to locate test files.
`excludeFolders`	Array	Contains folder names that the migrator will skip. Provide the full path for each folder name.
`excludeFiles`	Array	Contains file names that the migrator will skip. Provide the full path for each file name.
`excludeComponents`	Array	Contains legacy component tags that the migrator will skip.
`includeFiles`	Array	Contains the names of the only files that the migrator will process for migration. Provide the full path for each file name.
`includeFolders`	Array	Contains the names of the only folders that the migrator will process for migration. Provide the full path for each folder name.
`includeComponents`	Array	Contains the only legacy component tags that the migrator will process for migration.
`forceMigration`	Array	Contains legacy component tags for which the `IgnoreRuleSet` will be ignored and the tags will be migrated to Core Pack.

Table A-1 (Cont.) Properties in the migrationConfig.json File

Property Name	Value Type	Description
`excludeAttributeExpressions`	Object	Contains component-specific attributes that should not be migrated if they contain an expression as a value.
`baseMigrationDir`	String	Determines the root directory that the migrator will run from. By default, the migrator runs from the calling directory if not specified. You can use a relative or absolute path.
`logFileName`	String	Determines the name of the log file generated during migration. If not specified, the name `migrator.log` will be used by default.
`logFilePath`	String	Determines the output location for the migration log file. If not specified, the calling directory will be used by default.

Migrate WebDriver Test Files to Core Pack

The Oracle JET Core Pack migrator helps you migrate your WebDriver test files written for apps with legacy components so that they work with Core Pack components.

To migrate your test files, first follow the pre-migration steps found in Prepare Your Oracle JET App for Core Pack, which include initializing an Oracle JAF configuration in your app, running audits on your test files, and resolving any issues you find.

> **Note**
>
> Before running audits, make sure that your `oraclejafconfig.json` file is configured so that Oracle JAF can locate your test files. Include any paths to your test files within the `files` property as string values, such as `"./src/**/*.spec.ts"`.
>
> ```
> "files": [
>   "./src/**/*.html",
>   "./src/**/*.js",
>   "./src/**/*.ts",
>   "./src/**/component.json",
>   "./src/css/**/*.css",
>   "./src/**/*.spec.ts"
> ],
> ```

Once you have audited your test files and prepared them for migration, open or create your `migrationConfig.json` in your chosen directory, typically your app's `./src` directory, and configure it as follows.

1. Set the `testFiles` property in the `migrationConfig.json` file to `true`.
2. If your test files have a unique extension, such as `.spec.ts` or `.test.ts`, then add it to the migration configuration file as a string value to the `testFilesExtension` property.

   If you do not add an extension for test files to the `testFilesExtension` property, then all files in a directory that will be migrated must be test files.

3. Review the rest of your configuration to ensure that it meets your requirements for migration. For instance, verify that the `baseMigrationDir` property, which affects the directory that the migrator runs from, is correct.

   At a minimum, your `migrationConfig.json` file should resemble the following example.

   ```
   {
     "testFiles": "true",
     "testFilesExtension": ".spec.ts"
   }
   ```

4. Open a terminal window in the same directory as your migration configuration file. Execute the dry-run migration command while pointing to the configuration file:

   ```
   ojaf -mig -dr -c ./migrationConfig.json
   ```

5. Review the results of the dry run in the `migrator.log` file. Once you have made any necessary changes and are satisfied with your code and configuration, run the actual migration:

   ```
   ojaf -mig -c ./migrationConfig.json
   ```

When the migration is complete, try running your test files against the app that uses the Core Pack components they are meant to test. You may need to make manual corrections to get your tests to pass.

For instance, a property that is called using a function in a test file may not have been properly migrated, or perhaps your test files were properly migrated and a legacy component in your app's code contains an attribute that is not supported in Core Pack.

Use the migration report in the `migrator.log` file to troubleshoot issues. To assist you, the `migrator.log` file has links to migration sections in the API documentation for each legacy component that has a Core Pack equivalent.

Troubleshoot Core Pack Migration Issues

Some components may need manual correction after migration. For guidance on migrating specific legacy components to their Core Pack equivalents, read their migration sections in the Oracle JET API docs, which are accessible at the end of the `migrator.log` file. You can also refer to the Core Pack section of the JET API docs for more information.

Here is a list of instances that require manual migration:

- Formatting changes

  Users will see formatting changes in files as a result of migrating. Run formatters post-migration in order to ensure files remain in the correct formats.

- Deprecated APIs

  Deprecated APIs are not available in Core Pack components and must be removed before migration. If a deprecated API is present in your code, then your migration will fail.

- Components that switched to a data-driven approach

  There are some legacy components that took children that are instead data-driven in their Core Pack equivalents. These will be ignored by the migrator and must be migrated manually. See the section on data-driven components in the Core Pack API documentation, as well as the component-specific migration section.

- CSS

CSS will no longer work as expected with some components. See the section on CSS in the Core Pack API documentation, as well as the component-specific migration section.

- Components where a property type has changed

  If a component's property is set, but the property type has changed, then the legacy instance can't be migrated.

  For example, on form controls, such as `oj-input-text` and `oj-input-number`, the Core Pack `converter` type and the legacy `converter` type don't match. Therefore, if the `converter` property is set, then the instance won't be migrated. When this is the case, the `migrator.log` file will have a variation of the following message: `Rule to ignore: "oj-input-text" filtered out due to attribute "converter".`

- Class attributes with an expression as a value

  Class migration will only happen if the value is a string, otherwise migration will not modify anything in the expression.

- Properties with an expression

  If a property has an expression as a value, then the instance can't be migrated.

- References to strongly typed properties

  In TypeScript code, there may be cases where you want to type variables or properties based on the existing types declared by the JET components you are consuming.

  An example of this is when you define a custom component and want to define a chroming property on that component that can be passed directly down to a legacy `oj-button` component that you use as part of that component implementation. This type of dependency must be migrated as well if you are migrating the legacy component to its Core Pack equivalent

  When using the JET legacy components, you would typically use the `IntrinsicProps` of the component that is exported:

  ```
  import { ButtonIntrinsicProps} from 'ojs/ojbutton';
  ...
  const localChroming:ButtonIntrinsicProps['chroming']
  ```

  In Core Pack components, the `props` type is not re-exported from the Core Pack class because there is a more generic way to access the same information using the Preact `ComponentProps` convenience type. Instead, the code would be migrated as follows:

  ```
  import { ComponentProps } from 'preact';
  import 'oj-c/button';
  ...
  type ButtonProps = ComponentProps<'oj-c-button'>
  ...
  const localChroming:ButtonProps['chroming'] = ...
  ```

- References to strongly typed event payloads

  When coding in TypeScript, you can strongly type events emitted from JET legacy components by referencing the event map for the component:

  ```
  import 'ojs/ojbutton';
  import { ojButtonEventMap } from 'ojs/ojbutton';
  ```

```
...
private async onChatActionClick(event: ojButtonEventMap['ojAction']){...}
```

With Core Pack components, rather than import the `eventMap`, you can directly reference the type from the component interface:

```
import 'oj-c/button';
import { CButtonElement } from 'oj-c/button';
...
private async onChatActionClick(event: CButtonElement.ojAction){...}
```

- References to template context

  In TypeScript code, when you define a render function for a template slot for JET legacy components, you can use the relevant `Context` type:

  ```
  import 'ojs/ojtagcloud';
  import { ojTagCloud } from 'ojs/ojtagcloud';
  ...
  const itemRenderer = useCallback((context:
  ojTagCloud.ItemTemplateContext<string>)) => {...}
  ```

  With Core Pack components, these context definitions are not provided. You should instead use the type of the template `slot` property:

  ```
  import { ComponentProps } from 'preact';
  import 'oj-c/tag-cloud';
  ...
  type tagCloudProps = ComponentProps<'oj-c-tag-cloud'>
  ...
  const itemRenderer = useCallback((context: tagCloudProps['itemRenderer'])
  => {...}
  ```

B

Properties in the oraclejetconfig.json File

The `oraclejetconfig.json` file supports a range of properties that you can configure to determine the behavior of your Oracle JET project.

> **Note**
>
> Where `Property` is `<prop>.<subprop>` it indicates that `<subprop>` is a subproperty of `<prop>`. For example, `paths.components` means `"paths": { "components": "value" }`.

Table B-1 Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
architecture	String	`mvvm` or `vdom`	mvvm	Type of app architecture.
components	Object	component name/version value pairs		Component name/version value pairs for components to be restored from the component exchange upon `ojet restore`. Similar format to a `package.json`. For example: `"components": { "oj-doceg-double-picker": "^2.0.0" }`
bundleName	String	simple file name with a .JS extension	bundle.js	Allows an override of the default name used for an optimized app.
bundler	String	`webpack` | `<any>`		In release 11.0.0, JET introduced bundler-only support for Webpack. If `webpack` was specified as the value for the `bundler` property, the `before_webpack` hook was used to bundle the app. Otherwise, the `before_optimize` hook managed the RequireJS-based app bundling. Webpack-based bundler applied only to the app bundling. Custom component optimization continued to use RequireJS-based bundling, and could be configured with the `before_component_optimize` hook. In release 12.0.0, JET introduced end-to-end Webpack support. With the `--webpack` argument in an `ojet create` command, Oracle JET creates an `ojet.config.js` file where you configure Webpack usage. No `bundler` property is configured in the `oraclejetconfig.json` file.
defaultBrowser	String	browser name	chrome	Sent to Apache Cordova when serving hybrid mobile apps as `--target` when the destination is `browser`.
defaultTheme	String	redwood, redwood-notag, stable	redwood	Name of theme to use as the default in the app.

Table B-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes

dependencies	Object	component name/object or version number pairs		Names of potential component or pack dependencies used to check whether certain pre-minified components should be excluded from the `ojet build --release` bundling process. For example:
				`"dependencies": { "oj-pack-comp": { "version": "2.0.0"} }`
				Or
				`"oj-comp": "2.0.0"`

enableDocGen	Boolean	true/false		When `true`, Oracle JET generates API doc for VComponent-based web components in the `.appRootDir/web/components/`*`component-name`*`/docs` directory after you build the component using the `ojet build component` command if you have previously run the `ojet add docgen` command that installs the packages and templates to generate API doc. If you have not installed the packages and templates for API doc, the `ojet build component` command fails. When `false`, Oracle JET does not generate API doc for VComponent-based web components.
enableLegacyPeerDeps	Boolean	True/False	False	When `true`, the Oracle JET CLI modifies any invoked existing NPM install commands, such as the Oracle JET CLI's `add docgen` command, to include NPM's `--legacy-peer-deps` flag. You need to add the `enableLegacyPeerDeps` property to the `oraclejetconfig.json` file if you want to use this optional capability. The default value of `false` means that NPM installations proceed without the `--legacy-peer-deps` flag.
exchange-url	String	URL		Component exchange instance for publishing components. For example: `https://exchange.url.com/api/0.2.0` **Note** This setting can also be inherited (if not present) from a global value defined through `ojet configure --exchange-url=<addr> --global` which will be stored centrally (for example, `.ojet/exchange-url.json`)

Table B-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
fontUrl	String	URL		Link so that an Oracle JET CLI `build` command can insert icon fonts into Oracle JET templates. This property is associated with the `injector:font` token in the `appRootDir/src/index.html` file of your app.
generateSourceMaps	Boolean	true/false	False	When `true`, the Oracle JET CLI configures Terser and RequireJS packages to generate source map files when you build the Oracle JET app using the `ojet build` or `ojet serve` commands.
generatorVersion	String	Oracle JET CLI version		Deprecated. Historical information about the version of JET that was first used to create the project. Not used by the CLI.
installer	String	`yarn` or `npm`	npm	If specified, an alternate installer to run instead of the default npm for `npm install` type commands.
localComponentsSupport	Boolean	true/false		Indicates whether the component exchange backend supports the `local components` extension. The value will be recorded by the CLI in `oraclejetconfig.json`. If a user wants to opt out of the local components support, they can set this value to `false` deliberately.
paths.components	String	path	jet-composites	Path where locally-created components are stored relative to a root that is dependent on the scaffolded project type: 1. Project created with `--vdom` or `--template=basic-vdom` template. Root will be `src/` . 2. Project created with `--typescript`. Root will be `src/ts/` . 3. Default project. Root will be `src/js`. **Note** In a a project created with the vdom `basic` template or `--vdom` option, this value will be pre-set to just `components` rather than the default `jet-composites`.
paths.exchangeComponents	String	path	exchange_components	Folder where components added from the exchange are stored for new virtual DOM apps. Non-virtual DOM apps (MVVM) use the older `jet_components` when this is not set. This path must be a simple folder name which will be created in the root of the project as a peer of the `src/` folder.
paths.source.common	String	path	src	Simple folder name relative to the project root. A folder hierarchy cannot be used here. Other settings such as `paths.components` will be relative to this location.

Table B-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
paths.source.hybrid	String	path	src-hybrid	Simple folder name relative to the project root. A folder hierarchy cannot be used here.
paths.source.javascript	String	path	js	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here. Other settings such as `paths.components` may be relative to this location in the relevant project type.
paths.source.styles	String	path	css	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.source.themes	String	path	themes	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.source.tsconfig	String	path		If specified, this subproperty enables the relocation of the `tsconfig.json` file from its default location at the app root. Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.source.typescript	String	path	ts	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here. Other settings such as `paths.components` may be relative to this location in the relevant project type.
paths.source.web	String	path	src-web	Simple folder name relative to the defined `src` folder. A folder hierarchy cannot be used here.
paths.staging.hybrid	String	path	hybrid	Path where the hybrid build products are generated.
paths.staging.themes	String	path	staged-themes	Path where themes are staged.
paths.staging.web	String	path	web	Path where the web build products are generated.
sassVer	String	semver-style version number	1.80.5	Dart Sass (`sass`) NPM package version that will be installed if sass is added
stripList	Array of strings	path strings		List of `.gitignore`-style paths to strip when `ojet strip` is executed. This bypasses the list in the `.gitignore` file.

Table B-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes

unversioned	Boolean	true/false	false	When `true`, Oracle JET generates components in a directory path without the component version number when you run `ojet build` or `ojet serve`. For example:
				`appRootDir/web/components/my-component-pack/my-widget-1/`
				Without the `unversioned` property or with `"unversioned": false` (the default), the output path is:
				`appRootDir/web/components/my-component-pack/1.0.0/my-widget-1/`
				The `unversioned` entry in the `oraclejetconfig.json` file takes precedence over the Oracle JET command-line argument (`--omit-component-version`). That is, if the `oraclejetconfig.json` file includes `"unversioned": false` (include component version number in the directory path) and you build your Oracle JET app using the following command, Oracle JET includes the component version in the generated directory path:
				`ojet build --omit-component-version`

watchInterval	String	Number of milliseconds	1000	Configure the interval at which the live reload feature polls the Oracle JET project for updates by configuring a value for this property. The default value is `1000` milliseconds.
The remaining entries in this table describe the properties that you use to manage ICU translation bundles, as described in Work with ICU Translation Bundles in an Oracle JET Virtual DOM App.				
translationIcuLibraries	String	@oracle/oraclejet-icu-l10n	@oracle/oraclejet-icu-l10n	The NPM package that the Oracle JET CLI's add translation command installs. You need this NPM package to generate ICU translation bundles.
buildICUTranslationsBundle	Boolean	true/false	false	When `true`, the Oracle JET CLI `build` command generates runtime ICU translations bundles from the ICU translation bundles in your project.
translation.type	String	icu	icu	The only supported value at present for this property is `icu`.

Table B-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
translation.options.rootDir	String	Root directory for the runtime ICU translation bundles.	./src/resources/nls	The root directory for ICU translation bundles. The value that the `add translation` command inserts is `"./src/resources/nls"`.
translation.options.bundleName	String	Name of the ICU translation bundle	translationBundle.json	The bundle name from which to generate runtime ICU translation bundles when you run the `ojet build` or `serve` commands. The value that the `add translation` command inserts is `"translationBundle.json"`.
translation.options.locale	String	Language tag (For example, "en-US")	en-US	The locale of the ICU translation bundle in the root directory. The value that the `add translation` command inserts is `"en-US"`.
translation.options.outDir	String	Path to output directory	./src/resources/nls	The output directory for runtime ICU translation bundles. The value that the `add translation` command inserts is `"./src/resources/nls"`. Change this to a value of your choice. For example, `"./src/resources-dist"`.
translation.options.supportedLocales	String	Array of supported locales		A comma-separated list of additional locales to build. If you specify a locale that does not have a corresponding entry in the `nls` directory under the `rootDir` directory, then the runtime ICU translation bundle is built from the ICU translation bundle in the `rootDir` directory. The value that the `add translation` command inserts is `"de"`.
translation.options.componentBundleName	String	Name of component-specific ICU translation bundle	${componentName}-strings.json	Specifies the naming pattern for the ICU translation bundle associated with a component. The `add translation` command inserts the value `"${componentName}-strings.json"` where `{componentName}` references the name of the component. If you create a component named `my-component`, then the file name for the ICU translation bundle must be `my-component-strings.json`.