

Oracle® JavaScript Extension Toolkit (Oracle JET)

Developing Oracle JET Apps Using Virtual DOM Architecture



15.1.0
F84105-01
October 2023

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Oracle JavaScript Extension Toolkit (Oracle JET) Developing Oracle JET Apps Using Virtual DOM Architecture, 15.1.0

F84105-01

Copyright © 2021, 2023, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

1 Get Started with Virtual DOM Architecture in Oracle JET

About Oracle JET Virtual DOM Architecture	1-1
What Can You Do with the Oracle JET Virtual DOM Architecture?	1-1
Prerequisite Knowledge	1-2
Development Environment for Virtual DOM Apps	1-3
Install Oracle JET Tooling	1-4
Install Node.js	1-4
Install the Oracle JET Command-Line Interface	1-4
Use the npx Node.js Package Runner	1-5
Yarn Package Manager	1-6

2 Understand the Web App Workflow

Scaffold a Web App	2-1
About the Virtual DOM App Layout	2-3
About the Binding Provider for Virtual DOM Apps	2-6
Add Progressive Web App Support to Web Apps	2-7
Build a Web App	2-8
Serve a Web App	2-9
About ojet serve Command Options and Express Middleware Functions	2-10
Serve a Web App to a HTTPS Server Using a Self-signed Certificate	2-11
Customize the Web App Tooling Workflow	2-14
About the Script Hook Points for Web Apps	2-15
About the Process Flow of Script Hook Points	2-17
Change the Hooks Subfolder Location	2-19
Create a Hook Script for Web Apps	2-19
Pass Arguments to a Hook Script for Web Apps	2-22
Use Webpack in Oracle JET App Development	2-23
Configure Oracle JET's Default Webpack Configuration	2-25

3 Understand VComponent-based Web Components

Hello VComponent, an Introduction	3-1
-----------------------------------	-----

Metadata for VComponents	3-4
Nest VComponents	3-4
VComponent Properties	3-5
Declare VComponent Properties	3-5
Reference Properties in JSX	3-6
Access Properties	3-7
Reference Properties of a Child Component in JSX	3-7
Type-Checking Support	3-8
Global HTML Attributes	3-8
Children and Slot Content	3-9
Default Slots	3-9
Named Slots	3-10
Refresh Custom Elements with Dynamic Children and Slot Content	3-11
Template Slots	3-12
Provide Template Slot Content within HTML	3-14
Template Slots in JSX	3-15
Understand Events and Actions	3-16
Listeners	3-17
Actions	3-18
Declare Actions	3-18
Dispatch Actions	3-19
Respond to Actions	3-19
Action Payloads	3-20
Manage State Properties	3-21
Declare State	3-21
Update State	3-23
Understand the State Mechanism	3-25
Reference Child VComponents by Value	3-25

4 Work with Oracle JET VComponent-based Web Components

Create Web Components	4-1
Create Standalone Web Components	4-1
Create JET Packs	4-4
Create Resource Components for JET Packs	4-9
Create Reference Components for Web Components	4-12
Add Web Components to Your Page	4-14
Generate API Documentation for VComponent-based Web Components	4-16
Build Web Components	4-18
Package Web Components	4-19

5 Use Oracle JET Components and Data Providers

Access Subproperties of Oracle JET Component Properties	5-1
Mutate Properties on Oracle JET Custom Element Events	5-2
Avoid Repeated Data Provider Creation	5-2
Avoid Data Provider Re-creation When Data Changes	5-3
Use Oracle JET Popup and Dialog Components	5-5

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

7 Test and Debug Oracle JET Apps

Test Oracle JET Apps	7-1
Testing Types	7-1
About the Oracle JET Testing Technology Stack	7-5
Configure Oracle JET Apps for Testing	7-5
Debug Oracle JET Apps	7-8
Debug Web Apps	7-8
Use Preact Developer Tools	7-9

8 Package and Deploy Apps

Package Web Apps	8-1
Deploy Web Apps	8-1
Remove and Restore Non-Source Files from Your JET App	8-2

A Properties in the oraclejetconfig.json File

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](#)
- [Documentation Accessibility](#)
- [Related Resources](#)
- [Conventions](#)

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 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.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
<code>monospace</code>	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

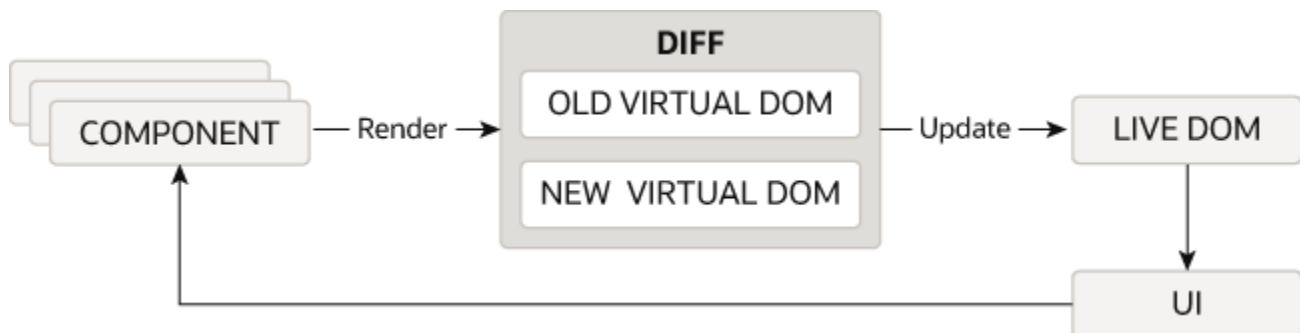
Oracle JET provides the capability to build apps and web components using a virtual DOM architecture that renders components using a virtual DOM engine.

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:

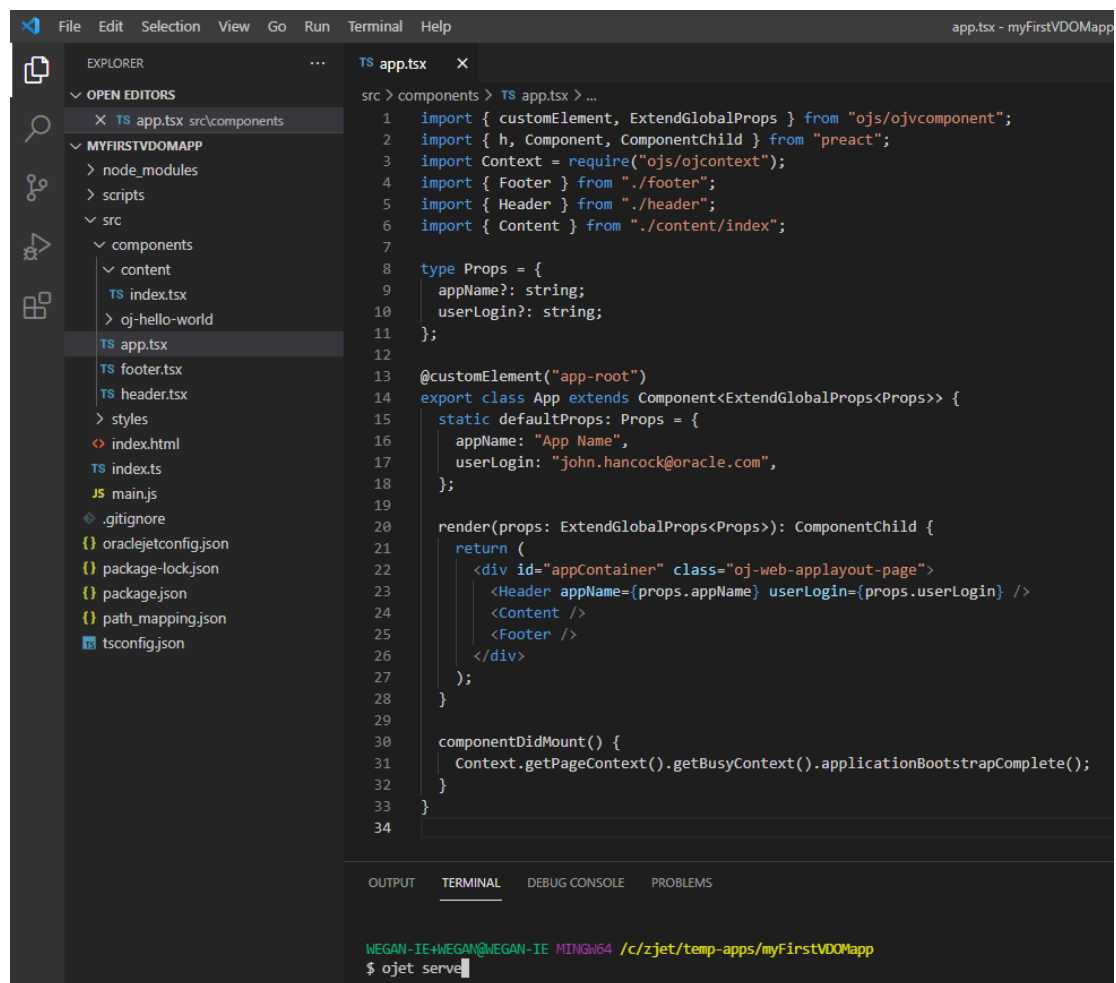
- **Preact:** Understand how Preact and, by extension, React works. To get started, read the [Main Concepts](#) section of the React documentation. If you are familiar with React concepts, read the [Differences to React](#) section of the Preact documentation.
- **TypeScript and JSX:** You write all 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.

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.



```
File Edit Selection View Go Run Terminal Help
app.tsx - myFirstVDOMApp

EXPLORER
  OPEN EDITORS
    TS app.tsx src/components
  MYFIRSTVDOMAPP
    node_modules
    scripts
    src
      components
        content
          TS index.tsx
          > oj-hello-world
            TS app.tsx
            TS footer.tsx
            TS header.tsx
            > styles
            > index.html
            TS index.ts
            JS main.js
            .gitignore
            {} oraclejetconfig.json
            {} package-lock.json
            {} package.json
            {} path_mapping.json
            tsconfig.json

src > components > TS app.tsx > ...
1  import { customElement, ExtendGlobalProps } from "ojs/ojvcomponent";
2  import { h, Component, ComponentChild } from "preact";
3  import Context = require("ojs/ojcontext");
4  import { Footer } from "../footer";
5  import { Header } from "../header";
6  import { Content } from "../content/index";
7
8  type Props = {
9    appName?: string;
10   userLogin?: string;
11 };
12
13 @customElement("app-root")
14 export class App extends Component<ExtendGlobalProps<Props>> {
15   static defaultProps: Props = {
16     appName: "App Name",
17     userLogin: "john.hancock@oracle.com",
18   };
19
20   render(props: ExtendGlobalProps<Props>): ComponentChild {
21     return (
22       <div id="appContainer" class="oj-web-applayout-page">
23         <Header appName={props.appName} userLogin={props.userLogin} />
24         <Content />
25         <Footer />
26       </div>
27     );
28   }
29
30   componentDidMount() {
31     Context.getPageContext().getBusyContext().applicationBootstrapComplete();
32   }
33 }
34

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS
MEGAN-IE+MEGAN@MEGAN-IE MINGW64 /c/zjet/temp-apps/myFirstVDOMApp
$ ojet serve
```

Install Oracle JET Tooling

If you plan to use Oracle JET tooling to develop web apps, you must install Node.js and the Oracle JET command-line interface (CLI), `ojet-cli`.

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](#).



Note:

If you do not want to install the Oracle JET CLI, you can use the Node.js package runner (`npm`) to create and manage Oracle JET apps.

To install the prerequisite packages:

1. [Install Node.js](#)
2. [Install the Oracle JET Command-Line Interface](#)

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
```

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.

Use the npx Node.js Package Runner

If you do not want to install the Oracle JET CLI NPM package on your development computer, you can use the `npx` Node.js package runner as an alternative to create and manage Oracle JET apps.

You may also find this alternative useful if you frequently change releases of the Oracle JET CLI, which requires you to uninstall and reinstall the NPM packages that deliver the Oracle JET 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 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 12.0.0 app
$ npx @oracle/ojet-cli@12.0.0 create myJET12app --template=basic --vdom
$ cd myJET12app
$ npx @oracle/ojet-cli@12.0.0 serve
```

```
// Create and serve an Oracle JET 15.1.0 app
$ npx @oracle/ojet-cli@15.1.0 create myJETapp --template=basic --vdom
$ cd myJETapp
$ npx @oracle/ojet-cli@15.1.0 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@15.1.0`) 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
```

To learn more about `npx`, see the [Node.js documentation](#). For more information about the commands that the Oracle JET CLI provides, see [#unique_27Understand the Web App Workflow](#).

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": "15.1.0",
  "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:

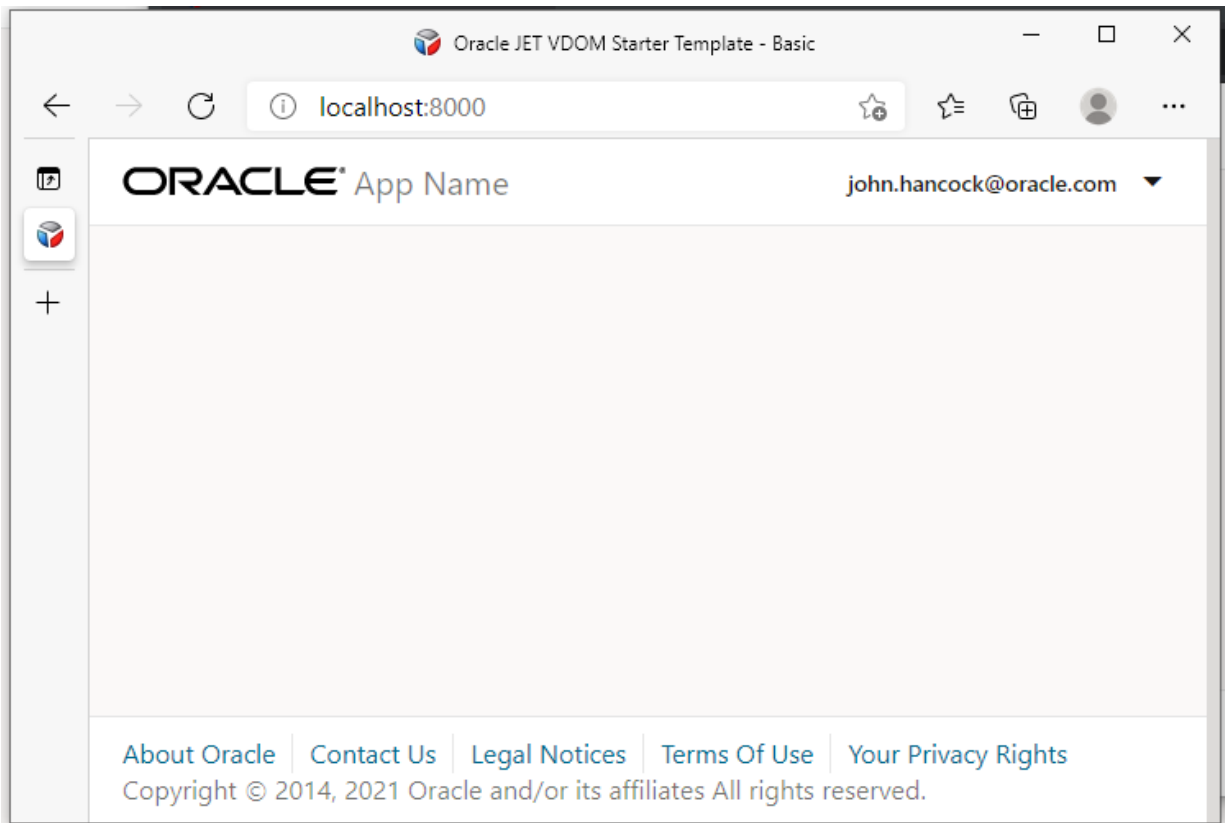
- Before you can create your first Oracle JET web app using the CLI, you must install the prerequisite packages if you haven't already done so. For details, see [Install Oracle JET Tooling](#).
- Then, use the Oracle JET command-line interface package (`ojet-cli`) to scaffold a web app using a basic 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:

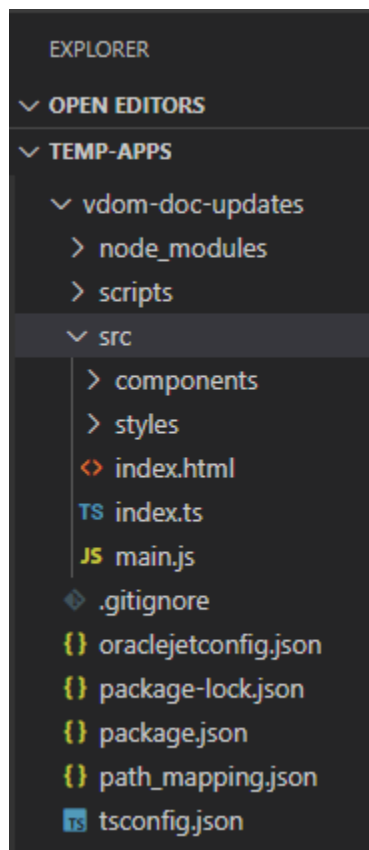
```
Oracle JET: Your app is ready! Change to your new app directory my-First-JET-VDOM-app and try ojet build and serve...
```

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 <code>app</code> component that imports <code>footer</code> , <code>header</code> , and <code>content</code> 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 <code>ojet strip</code> to avoid checking in content that Oracle JET always regenerates. Note this file must not be deleted since the <code>ojet strip</code> 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-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.

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:

- `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](#).

- `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.html',  
  'manifest.json',  
  'js/',  
  'css/'  
];
```

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

```
<html lang="en-us">  
  <head>  
    ...  
    <link rel="manifest" href="manifest.json">  
  </head>  
  ...  
  <script type="text/javascript">  
    if ('serviceWorker' in navigator) {  
      navigator.serviceWorker.register('sw.js').then(function(registration)
```

```
{
    // Registration was successful
    console.log('myPWAapp ServiceWorker registration successful
with scope: ', registration.scope);
}).catch(function(err) {
    // registration failed
    console.log('myPWAapp ServiceWorker registration failed: ',
err);
});
}
</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=livereload-port-number
           --livereload

           --build

           --theme=themename --themes=theme1,theme2,...
           --server-only
]
```

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: sourceFiles is ready.
Watcher: sass is ready.
Watcher: themes is ready.
Changed: C:\jet\vdomea-app\src\components\oj-hello-world\oj-hello-world-styles.css
Running before_watch hook.
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
<code>server-port</code>	Server port number. If not specified, defaults to 8000.
<code>livereload-port</code>	Live reload port number. If not specified, defaults to 35729.

Option	Description
<code>livereload</code>	<p>Enable the live reload feature. Live reload is enabled by default (<code>--livereload=true</code>).</p> <p>Use <code>--livereload=false</code> or <code>--no-livereload</code> to disable the live reload feature.</p> <p>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.</p> <p>You can also configure the interval at which the live reload feature polls the Oracle JET project for updates by configuring a value for the <code>watchInterval</code> property in the <code>oraclejetconfig.json</code> file. The default value is 1000 milliseconds.</p>
<code>build</code>	<p>Build the app before you serve it. By default, an app is built before you serve it (<code>--build=true</code>).</p> <p>Use <code>--build=false</code> or <code>--no-build</code> to suppress the build if you've already built the app and just want to serve it.</p>
<code>theme</code>	<p>Theme to use for the app. The theme defaults to <code>redwood</code>.</p>
<code>themes</code>	<p>Themes to use for the app, separated by commas.</p> <p>If you don't specify the <code>--theme</code> flag as described above, Oracle JET will use the first element that you specify in <code>--themes</code> as the default theme. Otherwise Oracle JET will serve the app with the theme specified in <code>--theme</code>.</p>
<code>server-only</code>	<p>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.</p>

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);

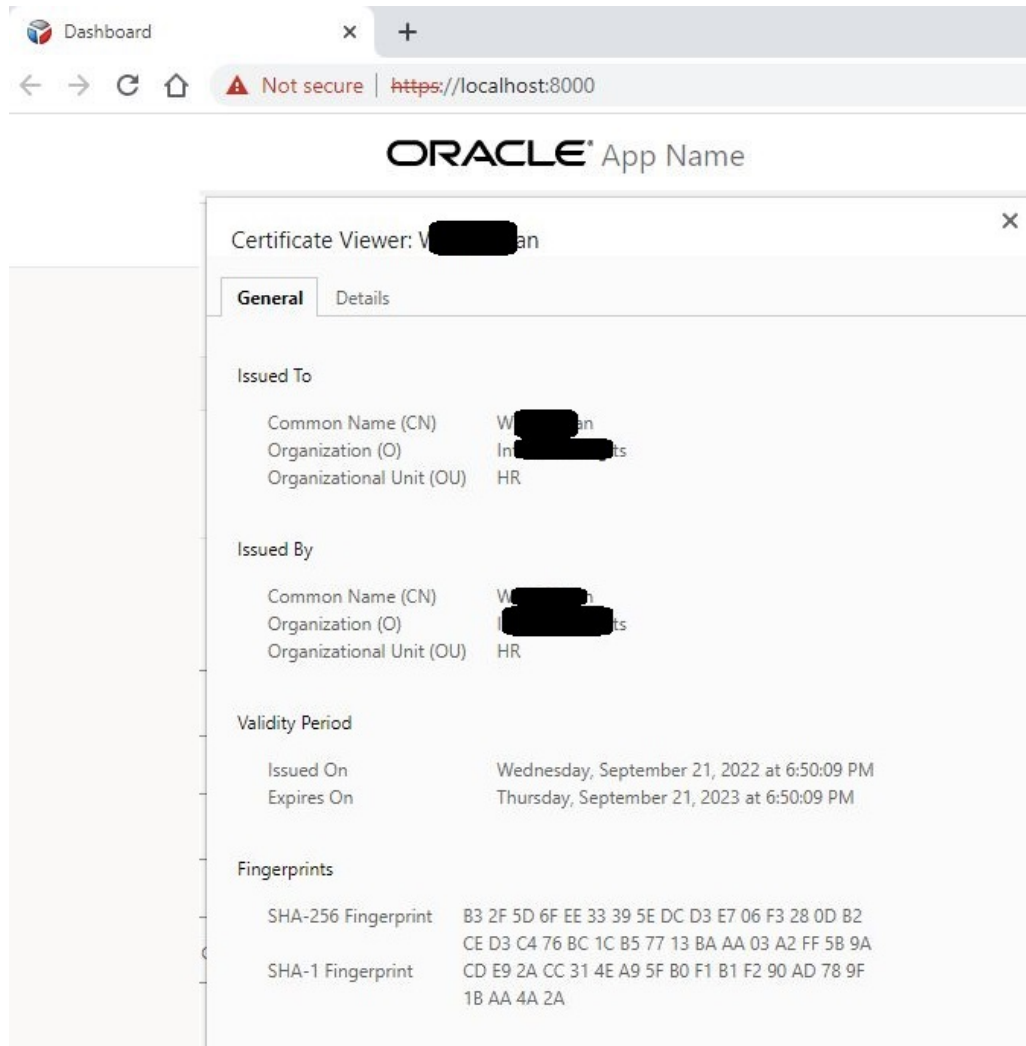
// Enable the Oracle JET live reload option using its default port
number so that
// any changes you make to app code are immediately reflected in the
browser after you
// serve it
const tinylr = require("tiny-lr");
const lrPort = "35729";

// 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 defines various script trigger points, also called hook points, that allow you to customize the create, build, serve, package, and restore workflow across the various 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
<code>after_app_create</code>	create	This hook point triggers the script with the default name <code>after_app_create.js</code> immediately after the tooling concludes the create app process.
<code>after_app_restore</code>	restore	This hook point triggers the script with the default name <code>after_app_restore.js</code> immediately after the tooling concludes the restore app process.
<code>before_build</code>	build	This hook point triggers the script with the default name <code>before_build.js</code> immediately before the tooling initiates the build process.
<code>before_release_build</code>	build (release mode only)	This hook point triggers the script with the default name <code>before_release_build.js</code> before the minification step and the <code>requirejs</code> bundling step occur.
<code>before_app_typescript</code>	build / serve	This hook point triggers the script with the default name <code>before_app_typescript.js</code> 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 <code>tsconfig.json</code> compiler configuration file. The hook system passes your reference to the modified <code>tsconfig</code> object to the TypeScript compiler. A script for this hook point can only be used with a TypeScript app.

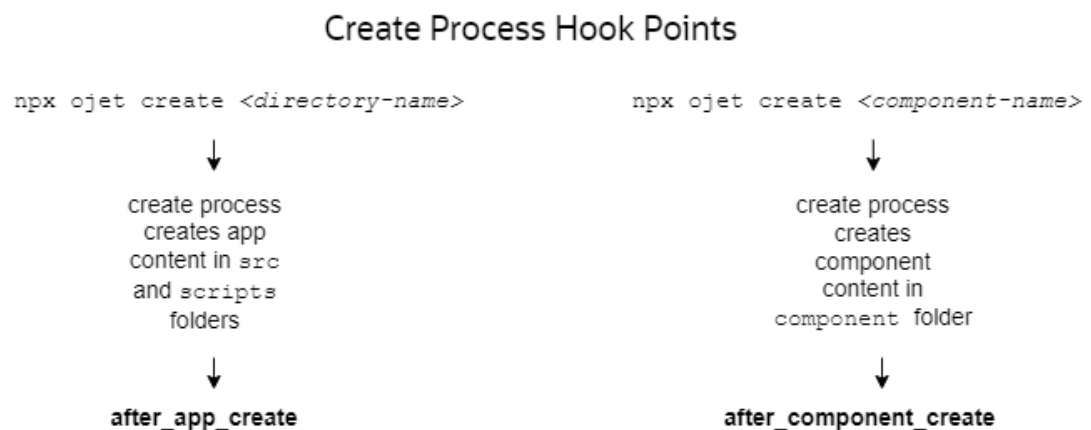
Hook Point	Supported Tooling Process	Description
after_app_typescript	build / serve	This hook point triggers the script with the default name <code>after_app_typescript.js</code> after the build process or serve process steps occur and immediately after the <code>before_app_typescript</code> hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated <code>.jsx</code> 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 <code>before_component_typescript.js</code> 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 <code>tsconfig.json</code> compiler configuration file. The hook system passes your reference to the modified <code>tsconfig</code> 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 <code>after_component_typescript.js</code> after the build process or serve process steps occur and immediately after the <code>before_component_typescript</code> hook point executes. This hook provides an entry point for apps that require further processing, such as compiling generated <code>.jsx</code> 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 <code>before_injection.js</code> after the tooling concludes the before build process and before the tooling performs the tasks to insert the CSS theme into the app. In other words, this hook point provides an entry point to customize the <code>main.js/index.html</code> theme injection process in the build.
before_optimize	build / serve (release mode only)	This hook point triggers the script with the default name <code>before_optimize.js</code> 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 <code>before_component_optimize.js</code> 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.
after_build	build	This hook point triggers the script with the default name <code>after_build.js</code> immediately after the tooling concludes the build process.
after_component_create	build	This hook point triggers the script with the default name <code>after_component_create.js</code> 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.

Hook Point	Supported Tooling Process	Description
after_component_build	build (debug mode only)	This hook point triggers the script with the default name <code>after_component_build.js</code> 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 <code>before_serve.js</code> before the web serve process connects to and watches the app.
after_serve	serve	This hook point triggers the script with the default name <code>after_serve.js</code> 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 <code>before_watch.js</code> 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 <code>after_watch.js</code> 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 <code>after_component_package.js</code> immediately after the tooling concludes the component package process.
before_component_package	package	This hook point triggers the script with the default name <code>before_component_package.js</code> immediately before the tooling initiates the component package process.

About the Process Flow of Script Hook Points

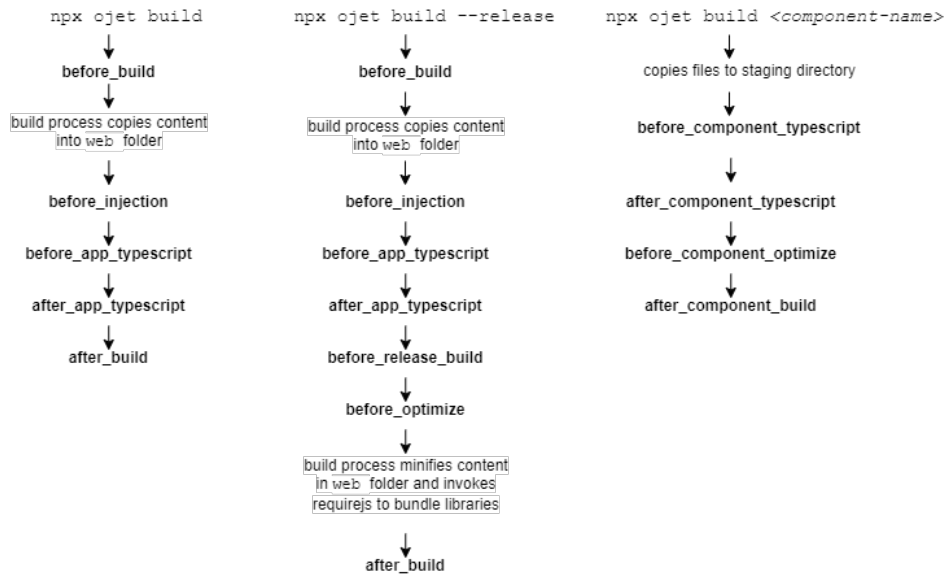
The Oracle JET hooks system defines various script trigger points, also called hook points, that allow you to customize the create, build, serve, and restore workflow across the various command-line interface processes.

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



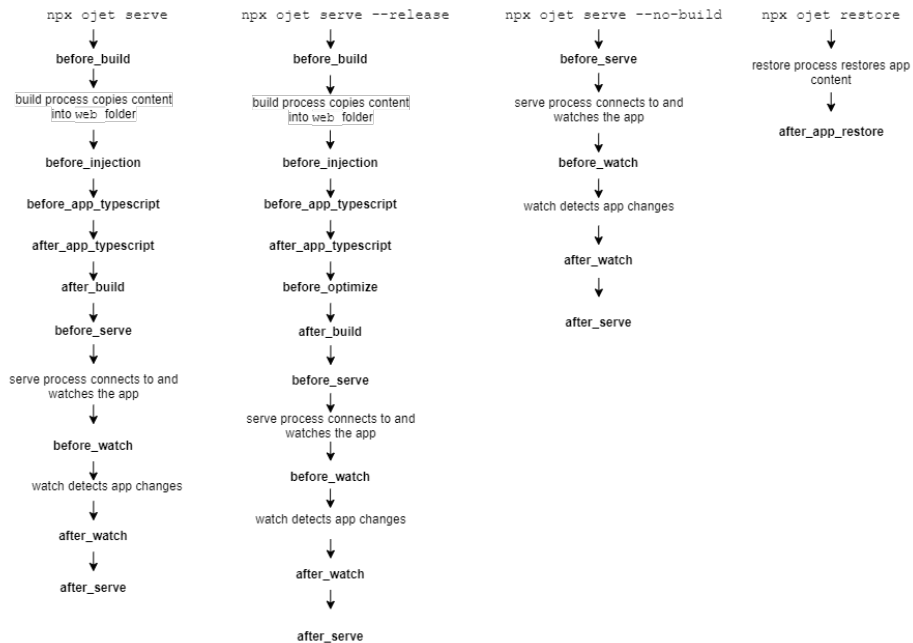
The following diagram shows the script hook point flow for the build process.

Build Process Hook Points



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

Serve and Restore Process Hook Points



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.

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
```

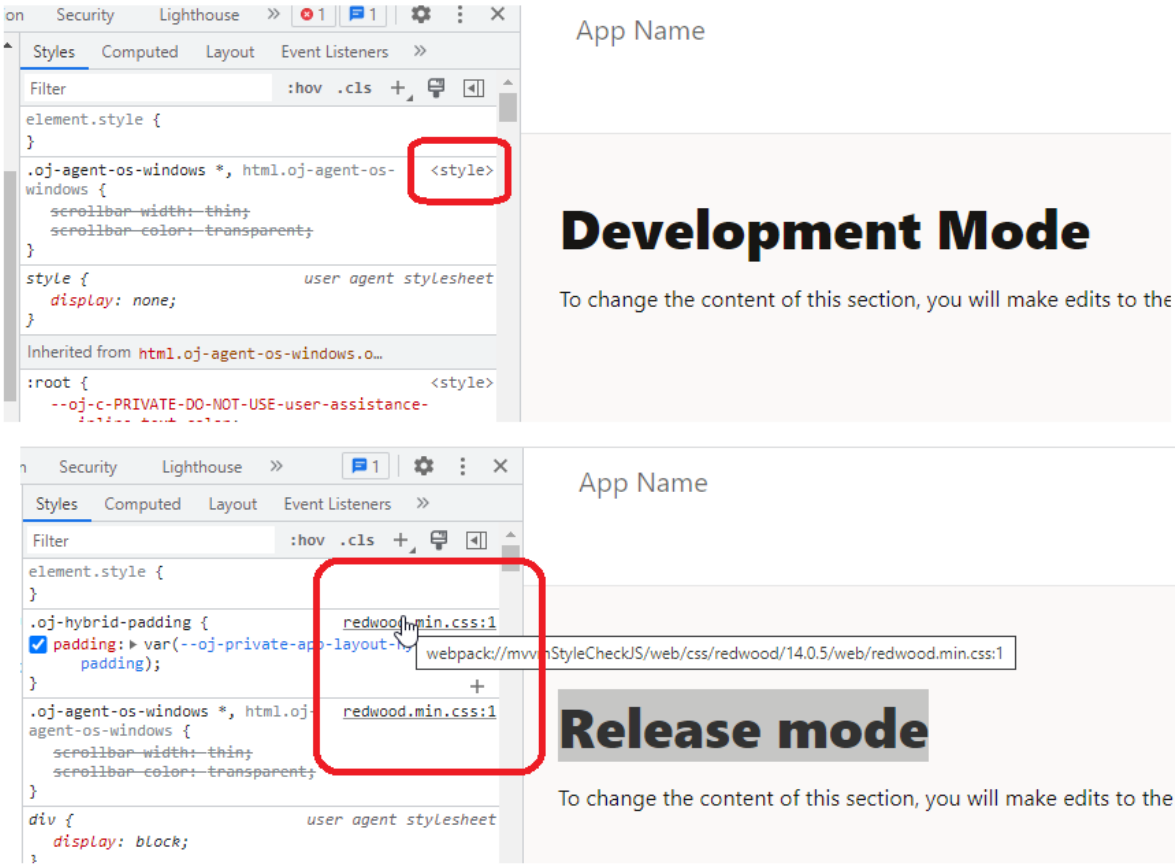
The files and directories in an Oracle JET project that uses Webpack differ to a project an app generated without specifying the `--webpack` argument. The following table describes the differences that result from use of the `--webpack` argument.

Option	Description
<code>/types</code>	The <code>types</code> directory contains a <code>/types/components/index.d.ts</code> 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 <code>preact.JSX.IntrinsicElements</code> type definitions. Note that <code>/types</code> 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 <code>tsconfig.json</code> file under the <code>typeRoots</code> option.
<code>ojet.config.js</code>	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 .
<code>tsconfig.json</code>	In contrast to the <code>tsconfig.json</code> generated for apps without Webpack, the <code>esModuleInterop</code> and <code>resolveJsonModule</code> flags are set to <code>true</code> . The <code>esModuleInterop</code> flag allows apps to standardize on default imports for all module types. Calls such as <code>import * as <importName> from "path/to/import"</code> should be written as <code>import <importName> from "path/to/import"</code> . The <code>resolveJsonModule</code> 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 <code>RequireJs</code> <code>text!</code> plugin followed by <code>JSON.parse</code> 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 use 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` class named `HelloWorld` with a custom element name of `hello-world` in a file named `hello-world.tsx`. Note the following about the entries in the `hello-world.tsx` file:

- JSX elements express the content of the virtual DOM tree (`<p>{props.message}</p>`).

- The `h` function, imported from the `preact` module, turns the JSX elements into virtual DOM elements.

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

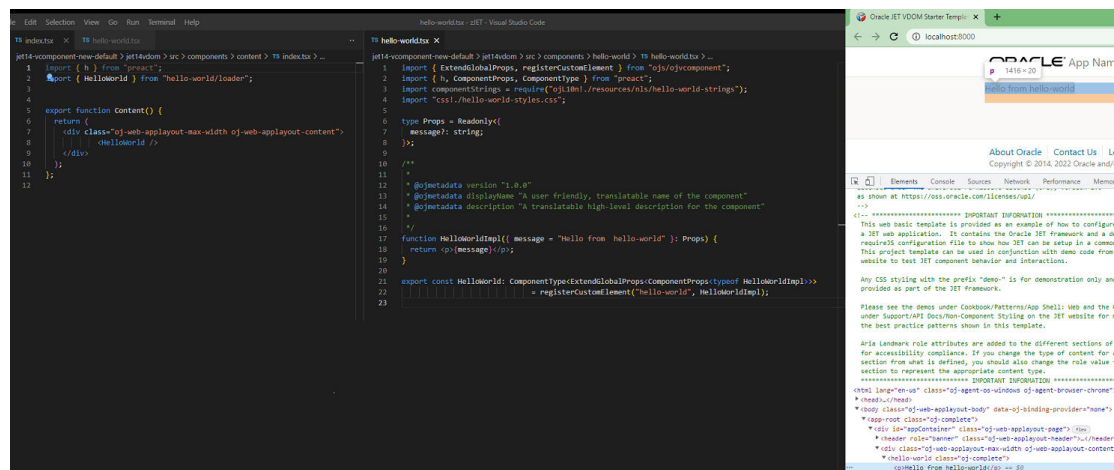
Readers who previously developed CCA-based web components will recognize the loader.ts file that the Oracle JET tooling includes so that the component can be used by the Component Exchange, Oracle Visual Builder, and the Oracle JET tooling itself. For a VComponent-based web component, the loader.ts file 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-button` element is configured both with default slot content, as well as content in the `startIcon` named slot:

```
<oj-button>  
  <span>This is default slot content</span>  
  <span slot="startIcon">This is named slot content</span>  
</oj-button>
```

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

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 "ojs/ojavatar";
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-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 "ojs/ojavatar";
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-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-button` element's display of a start icon depends on the value of the `showStartIcon` property. We want to ensure that the `oj-button` renders the start icon in the correct location when the `showStartIcon` property changes.

In the MVVM architecture, one could use the `oj-button`'s `refresh` method after mutating the child content to ensure the component incorporates the child DOM changes.

```

type Props = {
    showStartIcon?: boolean
}

```

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

A more appropriate approach for the VDOM architecture is to [assign a unique key](#) that reflects the `oj-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-button` re-renders with the start icon in the correct location:

```
import { h } from 'preact';
import { Remounter } from 'ojs/ojvcomponent-remounter';
import 'ojs/ojbutton';

type Props = {
  showStartIcon?: boolean
}

function ButtonStartIconSometimes(props: Props) {
  return (
    <Remounter>
      <oj-button>
        { props.showStartIcon && <span slot="startIcon">Start Icon</
span> }
        Click Me!
      </oj-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-obj-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 "ojs/ojavatar";
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-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-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-button onojAction={this._handleAction}>Click Me!<oj-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 { h, Component } from "preact";
import { customElement, GlobalProps } from "ojs/ojvcomponent";

@customElement("oj-greet-with-listeners")
export class GreetWithListeners extends Component<GlobalProps<Props>> {
  render() {
    return (
      <div>
        <div onClick={this._handleEvent}>
          <p>Hello, World!</p>
        </div>
        <oj-button onojAction={this._handleEvent}>Click Me</oj-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 `VComponent` 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 `VComponent`-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 or 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 subtlety 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:
    

    // 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={ 
}
```

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.

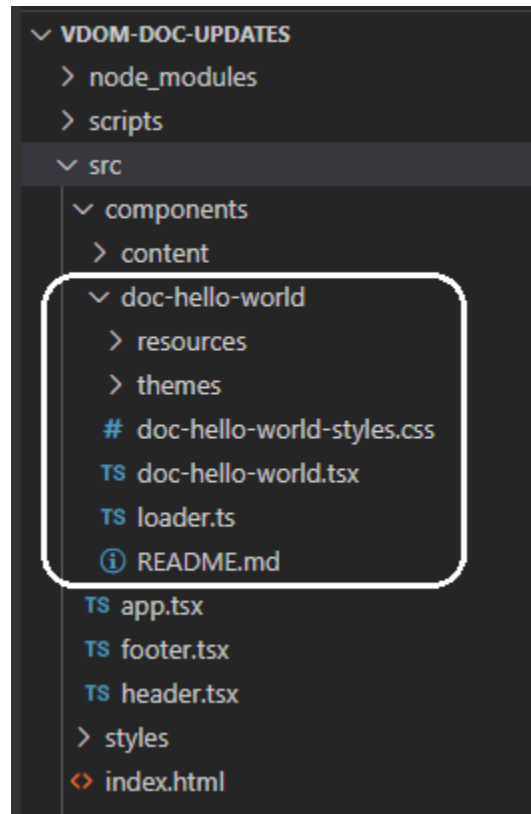
If you use this method, you'll create the folders and files manually to store the Web Component's content.

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 its referenced components.

- **type** must be set to `pack`.
- **displayName** is the name of the pack component that you want 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 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 Web Component in the TSX page where you will use the Web Component.

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. 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 { DocBonjourWorld } from "doc-bonjour-world/loader";

// Import components from a JET Pack
// 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";

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 */}
      <DocBonjourWorld />

      {/* 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 />
    </div>
  );
}
```

The following image shows each example rendering in the content component of the starter template.

ORACLE App Name

Hello from doc-hello-world! (Custom Element)

Bonjour de doc-bonjour-world! (Component Class)

Hello from my-component-pack-my-widget-1! (Custom Element)

Bonjour de my-component-pack-my-widget-2! (Component Class)

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 the following files 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).

```
footer.html
header.html
main.html
```

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 individual 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"
/>
```

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 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 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

Do not re-create a data provider each time that a `VComponent` renders.

For example, do not do the following in a `VComponent` using an `oj-list-view` component, as you re-create the `MutableArrayDataProvider` instance each time the `VComponent` renders:

```

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

```

Instead, consider using Preact's `useMemo` hook to ensure that a data provider instance is re-created only if the data in the data provider actually changes. The following example demonstrates how you include the `useMemo` hook to ensure that the data provider providing data to an `oj-list-view` component is not re-created, even if the `VComponent` that includes this code re-renders.

```

import MutableArrayDataProvider = require("ojs/
ojmutablearraydataprovder");

```

```

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>
  )
}

```

The reason for this recommendation is that a VComponent can re-render for any number of reasons, but as long as the data provided by the data provider does not change, there is no need to incur the cost of re-creating the data provider instance. Re-creating a data provider can affect your app's performance, due to unnecessary rendering, and usability as collection components may flash and lose scroll position as they re-render.

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 optimize 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/
ojmutablearraydatapreview");
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, you need to create a reference to the popup content. We recommend too that you place the popup content on its own within a `div` element so that it continues to work 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, . . . createRef } from "preact";  
. . .  
import "ojs/ojdialog";  
import { DialogElement } from "ojs/ojdialog";  
. . .  
  
@customElement('popup-launching-component')  
export class PopupLaunchingComp extends Component<GlobalProps> {  
  
  private dialogRef = createRef();  
  
  render(props) {  
    return (  
      <div>  
        <oj-button onojAction={ this.showDialog }>Show Dialog</oj-button>  
        <div>  
          <oj-dialog ref={this.dialogRef} cancelBehavior="icon"  
modality="modeless">  
            . . . </oj-dialog>  
          </div>  
        </div>  
      );  
    }  
  
    showDialog = () => {  
      this.dialogRef.current?.open();  
    }  
  }  
}
```

As a side effect of the `open()` call, Oracle JET relocates the popup content DOM to a 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 re-parent the still-open popup content back to its original parent. This interferes with 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 "ojs/ojbutton";
import { ojButton } from "ojs/ojbutton";
import { DialogElement } from "ojs/ojdialog";

export function Content() {

  const dialogRef = useRef<DialogElement>(null);

  const onSubmit = (event: ojButton.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-button onojAction={onSubmit} disabled={false}>
        Open dialog
      </oj-button>
      <div>
        <oj-dialog
          ref={dialogRef}
          dialogTitle="Dialog Title"
          cancelBehavior="icon">
          <div>Hello, World!</div>
          <div slot="footer">
            <oj-button id="okButton" onojAction={close}>
              Close dialog
            </oj-button>
          </div>
        </oj-dialog>
      </div>
    </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 take will vary, depending on the method you used to create your app.

If you used command-line tooling to scaffold your app, you will 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 will 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 will 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, do one of the following:

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

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

For example, enter the following command to install a library named `my-library`:

```
npm install my-library --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/15.1.0/default/js",  
    "css": "https://static.oracle.com/cdn/jet/15.1.0/default/css",  
    "config": "bundles-config.js"  
  },  
  "3rdparty": "https://static.oracle.com/cdn/jet/15.1.0/3rdparty"  
},
```

```

"libs": {

  "knockout": {
    "cdn": "3rdparty",
    "cwd": "node_modules/knockout/build/output",
    "debug": {
      "src": "knockout-latest.debug.js",
      "path": "libs/knockout/knockout-#{version}.debug.js",
      "cdnPath": "knockout/knockout-3.x.x"
    },
    "release": {
      "src": "knockout-latest.js",
      "path": "libs/knockout/knockout-#{version}.js",
      "cdnPath": "knockout/knockout-3.x.x"
    }
  },
  ... contents omitted
}

```

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

The 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, `cwd` points to the location where npm installed the library, `src` points to a path or array of paths containing the files that will be copied during a build, and `path` points to the destination that will contain the built version.

 **Note:**

If the existing entry that you copy to modify includes "cdn": "3rdparty", remove it from the newly-created entry for your library. This line references the Oracle JET third-party area on the content distribution network managed by Oracle. Your library won't be hosted there and keeping this line will cause a release build to fail at runtime by mapping the path for your library to a non-existent URL.

If you use a CDN, add the URL to the CDN in the entry for `cdnPath`.

- If you didn't use the 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 link to the new file in the path mapping section and include the new library in the `require()` 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',
    ... contents omitted
    'text': 'libs/require/text',
    'my-library': 'libs/my-library/my-library'
  },
  require(['knockout', 'my-library'],
  function(ko) // this callback gets executed when all required modules are
loaded
  {
    // add any startup code that you want here
  }
});
```

7

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, class, or component are producing the expected output or response.
- These tests are typically applied to self-contained business logic, components, 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 component and focus on isolating the tested component.

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 unit tests and 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 should interact with the component's public application programming interface (API) and pass the API as many different combinations of test data as necessary to exercise as much of the component's code paths as possible. This includes testing the component's properties, events, methods, and slots.

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 components, except for things that require CSS for styling. `jsdom` does not process CSS, so avoid using these tests to validate any CSS.

The Preact Testing Library provides a set of utility functions that make it easy to write tests that assert the behavior of Preact components without relying on their implementation details. It promotes a UI-centric approach to testing: components 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 `ojet add testing` CLI command to add testing capability to your Oracle JET app by setting up the framework, dependencies, and libraries required for testing for JET components, including Jest and the Preact Testing Library.

The `ojet add testing` Command

Run the command from a terminal window in your app's root directory. After it configures your app's testing environment, you can proceed to create and run tests on your app using Jest and the Preact Testing Library.

The configuration performed by the `ojet add testing` command creates some directories, dependencies, and files within your app that are essential to testing.

For instance, the configuration adds the `test-config` folder to your app's root directory. It contains two files that are required for testing: `jest.config.js` and `testSetup.ts`.

The file `testSetup.ts` imports runtime support for compiled and transpiled `async` functions, while `jest.config.js` is Jest's configuration file.

Additionally, existing components are checked for test files. The extension for files containing tests, known as "spec files," should be `.spec.tsx` so that tooling and your Jest testing configuration recognize them. If spec files are missing from a component, then they are injected. Spec files are located within a `__tests__` folder inside the `components` folder, such

as `src/components/<component-name>/__tests__`. This folder holds the test files you write for your component and, by default, is created with a spec file: `<component-name>.spec.tsx`.

 **Note:**

If you create a new component or pack from the command line after running the `add testing` command on your project, then the `__tests__` folder and its spec file are injected by default.

Testing Demo

Here we will demonstrate how to use the `ojet add testing` command to configure a testing environment for an Oracle JET app. After testing is enabled, we will run a unit test and a component test on the sample app.

1. First, create a new Oracle JET app using the `basic` template. Open a terminal window in your working directory and run the command `npx @oracle/ojet-cli create vdomTestApp --template=basic --vdom`.
2. Enter `cd vdomTestApp` to navigate to the app's root directory, then enter the command `npx ojet create component hello-world` to create a new component.
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>
  );
};
```

4. Enter the `npx ojet serve` command to run the app in your browser and confirm that the app runs and the new component renders. This is also a required step for testing, as before the components can be tested, they must first be built by the Oracle JET CLI.

ORACLE® App Name

john.hancock@oracle.com ▼

Hello from hello-world

[About Oracle](#) | [Contact Us](#) | [Legal Notices](#) | [Terms Of Use](#) | [Your Privacy Rights](#)
Copyright © 2014, 2023 Oracle and/or its affiliates All rights reserved.

 **Note:**

The text "Hello from hello world" visible in the running app is passed into your app's content through the `HelloWorld` component's `message` property, confirming that the component is working.

5. Run the `npx ojet add testing` command from a terminal window in your app's root directory.
In addition to configuring your testing environment, in your app's directory structure, you can see that the `test-config` folder was added to the root directory and the `__tests__` folder was added to the `/src/components/hello-world` directory.
Inside the `__tests__` folder is the spec file `hello-world.spec.tsx`, created by default 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 is used to render the component in the app.
If you look in the `package.json` file, you can see the testing dependencies that were added, 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. In addition to the component test, we will run a unit test. First, we must provide a function to be tested.
Open the `./VDOMTestApp/src/components/hello-world/hello-world.tsx` file in your code editor and, at the bottom of the file, insert the following function that returns the sum of two numbers. Save the file.

```
export const sum = (a: number, b: number) => {  
  return a + b;  
}
```

7. Open the `hello-world.spec.tsx` file and, at the top of the file, add an `import` statement for the `sum` function: `import { sum } from 'hello-world/hello-world'`.

- At the bottom of the `hello-world.spec.tsx` file, add the following unit test for the `sum` function. Save the file.

```
it('The sum is 10', () => {  
  expect(sum(6, 4)).toBe(10)  
})
```

- Build your app's project source by running the following command from the `./VDOMTestApp/` directory in a terminal window

```
npx ojet build
```

- Run the tests. Enter `npm run test` in the command line and observe the results in the terminal window.

```
C:\Users\jowilker\Desktop\Testing2023\vdome_content\apps\vdomeTestApp>npm run test  
  
> vdomTestApp@1.0.0 test  
> jest -c test-config/jest.config.js  
  
PASS src/components/hello-world/__tests__/hello-world.spec.tsx  
  ✓ The sum is 10 (1 ms)  
  Test description  
  ✓ Your test title (9 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      2 passed, 2 total  
Snapshots: 0 total  
Time:       3.785 s, estimated 5 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.

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';
```

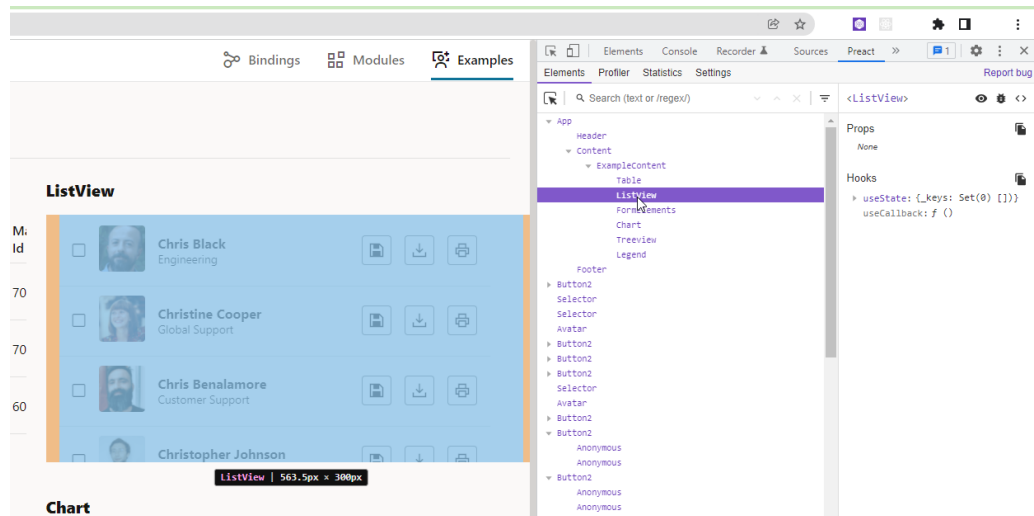
Oracle JET takes care of including this import when your virtual DOM app is built or served in debug mode (the default option for `ojet build` and `ojet serve`) by including the following injector token when you create the virtual DOM app:

```
// 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"  
    }  
  },  
},
```

8

Package and Deploy Apps

If you used Oracle JET tooling to create your Oracle JET app, you can package web apps for deployment to a web or app server.

Package Web Apps

If you created your app using the tooling, use the Oracle JET command-line interface (CLI) to create a release version of your app containing your app scripts and applicable Oracle JET code in minified format.

1. From a terminal prompt in your app's root directory, enter the following command: `ojet build --release`.

The command will take some time to complete. When it's successful, you'll see the following message: `Build finished!`.

The command replaces the development version of the libraries and scripts in `web/js/` with minified versions where available.

2. To verify that the app still works as you expect, run `ojet serve` with the `release` option.

The `ojet serve --release` command takes the same arguments that you used to [serve](#) your web app in development mode.

```
ojet serve --release [--serverPort=server-port-number --serverOnly]
```

Tip:

For a complete list of options, type `ojet help serve` at the terminal prompt.

Deploy Web Apps

Oracle JET is a collection of HTML, JavaScript, and CSS files that you can deploy to any type of web or app server. There are no unique requirements for deploying Oracle JET apps.

Deployment methods are quite varied and depend upon the type of server environment your app is designed to run in. However, you should be able to use the same method for deploying Oracle JET apps that you would for any other client interface in your specific environment.

For example, if you normally deploy apps as zip files, you can zip the `web` directory and use your normal deployment process.

Remove and Restore Non-Source Files from Your JET App

The Oracle JET CLI provides commands (`clean`, `strip`, and `restore`) that manage the source code of your JET app by removing extraneous files, such as the build output for the platforms your JET app supports or npm modules installed into your project.

Consider using these commands when you want to package your source code for distribution to colleagues or others who may work on the source code with you. Use of these commands may not be appropriate in all circumstances. Use of the `clean` and `strip` commands will, for example, remove the content in the `web` directory that is created when you run `ojet build` or `ojet serve`.

ojet clean

Use the `ojet clean` command to clean the build output of your JET app. Specify the `web` parameter with the `ojet clean` command (`ojet clean web`) to remove the contents of your app's root directory's `web` directory.

ojet strip

Use `ojet strip` when you want to remove all non-source files from your JET app. In addition to the build output removed by the `ojet clean` command, `ojet strip` removes additional dependencies, such as npm modules installed into your project. A typical usage scenario for the `ojet strip` command is when you want to distribute the source files of your JET app to a colleague and you want to reduce the number of files to transmit to the minimum.

The `ojet strip` command relies on the presence of the `.gitignore` file in the root directory of your app to determine what to remove. The file lists the directories that are installed by the tooling and can therefore be restored by the tooling. Only those directories and files listed will be removed when you run `ojet clean` on the app folder.

If you do not use Git and you want to run `ojet strip` to make a project easier to transmit, you can create the `.gitignore` file and add it to your app's root folder with a list of the folders and files to remove, like this:

```
#List of web app folders to remove
/node_modules
/bower_components
/themes
/web
```

As an alternative to the `.gitignore` file, you can include a `stripList` property that takes an array of glob pattern values in your app's `oraclejetconfig.json` file. When you specify the `stripList` parameter in the `oraclejetconfig.json` file, Oracle JET ignores the `.gitignore` file and its entries. Specify the list of directories and file types that you want to remove when you run `ojet strip`, as demonstrated by the following example.

```
{
  . . .
  "generatorVersion": "15.1.0",
  "stripList": [
```

```
    "jet_components",
    "node_modules",
    "bower_components",
    "dist",
    "web",
    "staged-themes",
    "themes",
    "myfiles/*.txt"
  ]
}
```

ojet restore

Use the `ojet restore` command to restore the dependencies, plugins, libraries, and Web Components that the `ojet strip` command removes. After the `ojet restore` command completes, use the `ojet build` and/or `ojet serve` commands to build and serve your JET app.

The `ojet restore` command supports a number of additional parameters, such as `ojet restore --ci` that invokes the `npm ci` command instead of the default `npm install` command. This option (`ojet restore --ci`) fetches the dependencies specified in the `package-lock.json` file, and can be useful in CI/CD pipelines.

For additional help with CLI commands, enter `ojet help` at a terminal prompt.

A

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 A-1 Properties in the `oraclejetconfig.json` File

Property	Value Type	Valid Values	Default	Notes
<code>architecture</code>	String	<code>mvvm</code> or <code>vdom</code>	<code>mvvm</code>	Type of app architecture.
<code>components</code>	Object	component name/version value pairs		Component name/version value pairs for components to be restored from the component exchange upon <code>ojet restore</code> . Similar format to a <code>package.json</code> . For example: <pre>"components": { "oj-doceg-double-picker": "^2.0.0" }</pre>
<code>bundleName</code>	String	simple file name with a <code>.JS</code> extension	<code>bundle.js</code>	Allows an override of the default name used for an optimized app.
<code>bundler</code>	String	<code>webpack</code> <code><any></code>		In release 11.0.0, JET introduced <code>bundler-only</code> support for Webpack. If <code>webpack</code> was specified as the value for the <code>bundler</code> property, the <code>before_webpack</code> hook was used to bundle the app. Otherwise, the <code>before_optimize</code> 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 <code>before_component_optimize</code> hook. In release 12.0.0, JET introduced end-to-end Webpack support. With the <code>--webpack</code> argument in an <code>ojet create</code> command, Oracle JET creates an <code>ojet.config.js</code> file where you configure Webpack usage. No <code>bundler</code> property is configured in the <code>oraclejetconfig.json</code> file.

Table A-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
defaultBrowser	String	browser name	chrome	Sent to Apache Cordova when serving hybrid mobile apps as <code>--target</code> when the destination is <code>browser</code> .
defaultTheme	String	redwood, redwood-notag, stable	redwood	Name of theme to use as the default in the app.
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 <code>ojet build --release</code> bundling process. For example: <pre>"dependencies": { "oj-pack-comp": { "version": "2.0.0"} }</pre> Or <pre>"oj-comp": "2.0.0"</pre>

Table A-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
exchange-url	String	URL		Component exchange instance for publishing components. For example: <code>https://exchange.url.com/api/0.2.0</code>
				<div style="border: 1px solid #0070C0; padding: 10px; background-color: #E6F2FF;"> <p> Note:</p> <p>This setting can also be inherited (if not present) from a global value defined through <code>ojet configure --exchange-url=<address> --global</code> which will be stored centrally (for example, <code>.ojet/exchange-url.json</code>)</p> </div>
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.

Table A-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
localComponentsSupport	boolean	true/false		Indicates whether the component exchange backend supports the local components extension. The value will be recorded by the CLI in <code>oraclejetconfig.json</code> . If a user wants to opt out of the local components support, they can set this value to <code>false</code> deliberately.
paths.components	String	path	jet-composites	<p>Path where locally-created components are stored relative to a root that is dependent on the scaffolded project type:</p> <ol style="list-style-type: none"> 1. Project created with <code>--vdom</code> or <code>--template=basic-vdom</code> template. Root will be <code>src/</code>. 2. Project created with <code>--typescript</code>. Root will be <code>src/ts/</code>. 3. Default project. Root will be <code>src/js</code>.

 **Note:**

In a project created with the `basic` template or `--vdom` option, this value will be pre-set to just `components` rather than the default `jet-composites`.

Table A-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
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 <code>jet_components</code> 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 <code>src/</code> 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 <code>paths.components</code> will be relative to this location.
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 <code>src</code> folder. A folder hierarchy cannot be used here. Other settings such as <code>paths.components</code> may be relative to this location in the relevant project type.
paths.source.styles	String	path	css	Simple folder name relative to the defined <code>src</code> folder. A folder hierarchy cannot be used here.
paths.source.themes	String	path	themes	Simple folder name relative to the defined <code>src</code> folder. A folder hierarchy cannot be used here.
paths.source.tsconfig	String	path		If specified, this subproperty enables the relocation of the <code>tsconfig.json</code> file from its default location at the app root. Simple folder name relative to the defined <code>src</code> folder. A folder hierarchy cannot be used here.
paths.source.typescript	String	path	ts	Simple folder name relative to the defined <code>src</code> folder. A folder hierarchy cannot be used here. Other settings such as <code>paths.components</code> 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 <code>src</code> 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.

Table A-1 (Cont.) Properties in the oraclejetconfig.json File

Property	Value Type	Valid Values	Default	Notes
sassVer	String	semver-style version number	5.0.0	node-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 <code>ojet strip</code> is executed. This bypasses the list in the .gitignore file.
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.