





Magento[®]
An Adobe Company

| U

Contents

Unit 1. Introduction	5
1.1 Introduction to PWA	5
Exercise 1.1.1	5
1.3 What Is PWA Studio?	6
Exercise 1.3.1	6
1.4 PWA Storefront vs. Magento Theme	6
Exercise 1.4.1	6
Unit 2. Environment Setup	8
2.2 Generate a Local SSL Certificate	8
Exercise 2.2.1	8
Unit 3. Build & Dev Tools	9
3.2 Build CLI Commands	9
Exercise 3.2.1	9
Unit 4. Storefront Structure	10
4.1 Storefront Structure Introduction	10
Exercise 4.1.1	10
4.2 Architecture of a Storefront	11
Exercise 4.2.1	11
Unit 5. UPWARD Server	12
5.1 UPWARD overview	12
Exercise 5.1.1	12
5.3 UPWARD implementations	13
Exercise 5.3.1	13
5.4 UPWARD Configuration	15
Exercise 5.4.2	15
Unit 6. Working with Components	16
6.1 Child Module Replacement	16
Exercise 6.1.1	16
6.2 Overriding a Component's Style	19
Exercise 6.2.1	19
6.3 Creating a New Storefront Component	21
Exercise 6.3.1	21
Unit 7. Peregrine Hooks & Talons	24
7.2 Peregrine Hooks	24
Exercise 7.2.1	24
7.3 Peregrine Talons	27
Exercise 7.3.1	27
Unit 8. Application State Management	33
8.1 Application State Data	33
Exercise 8.1.1	33
8.1 Update Application State	35
Exercise 8.2.1	35
8.3 Create a New Application State	37
Exercise 8.3.1	37

Unit 9. REST & GraphQL	39
9.1 Introduction to Magento Cloud	39
Exercise 9.1.1	39
9.2 Using Rest	43
Exercise 9.2.1	43
9.3 Using GraphQL	45
Exercise 9.3.1	45

Unit 1. Introduction

1.1 Introduction to PWA

Exercise 1.1.1

Diagram the technology stack using the following components :PHP, JS, React, Redux, Node.JS, Apollo, Webpack, Workbox

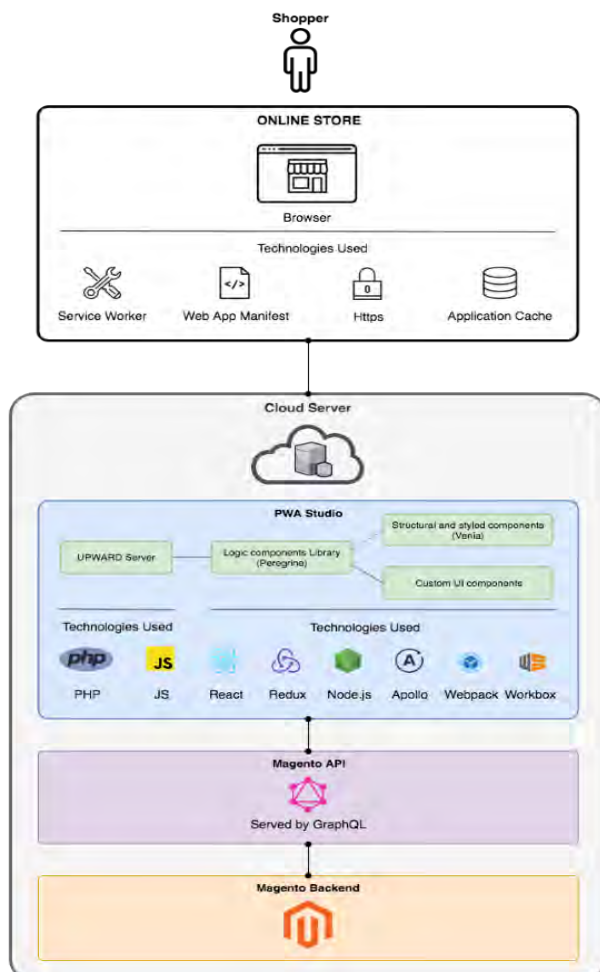
A client initiates a request using a web browser and sends the request through the stack.

Show the high-level workflow from the browser to the Magento backend.

See the documentation for help:

<https://magento.github.io/pwa-studio/technologies/tools-libraries/#javascript>

Solution



1.3 What Is PWA Studio?

Exercise 1.3.1

Magento provides a public URL to review a PWA Studio project.

URL: <https://venia.magento.com/>

Go through this URL and see the PWA Studio live experience by visiting different pages, adding items to a cart and checkout flow, and then compare the performance with a regular Magento instance.

Solution

Step 1

Open URL in the browser.

URL: <https://venia.magento.com/>

Step 2

Click on any category from the menu.

Step 3

Click on any product so it will display the product page.

Step 4

Add an item to the cart.

Step 5

Visit other pages to see the performance and features available in the Venia theme built using PWA Studio.

1.4 PWA Storefront vs. Magento Theme

Exercise 1.4.1

Review and identify the following components used in the ProductFullDetail component and see how the same component can be re-used multiple times.

- getProductDetail GraphQL Query
- Product quantity input field
- Swatches selection
- Price (reusable)

Use the pwa-studio github repository as a reference:

<https://github.com/magento/pwa-studio/tree/develop/packages/venia-ui/lib/RootComponents/Product>

Solution

Step 1

Open the linked github repository and view the Product Root Component directory. Click into the product.js file.

Step 2 – GraphQL

Find the reference to the getProductDetail.graphql file.

```
import GET_PRODUCT_DETAIL from '../..//queries/getProductDetail.graphql';
```

In the github repository, open the getProductDetail.graphql file and identify the attributes loaded.

<https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/queries/getProductDetail.graphql>

Step 3 - Quantity

From back in the Product root component product.js file, find the ProductFullDetail component and navigate to that in github.

<https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/components/ProductFullDetail/productFullDetail.js>

In the productFullDetail.js file the ProductQuantity component reference and implementation:

```
import Quantity from '../ProductQuantity';
...
<Quantity
    initialValue={quantity}
    onChange={handleSetQuantity}
/>
```

Step 4 – Swatches

From back in the productFullDetail.js component find the reference to product options.

```
const Options = React.lazy(() => import('../ProductOptions'));
```

Follow the components to the following files:

<https://github.com/magento/pwa-studio/tree/develop/packages/venia-ui/lib/components/ProductOptions>
<https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/components/ProductOptions/swatchList.js>
<https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/components/ProductOptions/swatch.js>

Step 5 – Price

From back in the productFullDetail.js component find the reference to Price.

Re-usable Price component

<https://github.com/magento/pwa-studio/tree/develop/packages/peregrine/lib/Price>

Used in:

1. <https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/components/ProductFullDetail/productFullDetail.js>
2. <https://github.com/magento/pwa-studio/blob/develop/packages/venia-ui/lib/components/Gallery/item.js>

Unit 2. Environment Setup

2.2 Generate a Local SSL Certificate

Exercise 2.2.1

Create a new PWA Studio storefront project using the scaffolding feature.

Solution

Step 1:

Run this command `npx @magento/create-pwa` in terminal.

The command will need the following inputs:

- **Project root directory (will be created if it does not exist):**
Enter in the project name. Generally, it is the folder name
- **Short name of the project to put in the package.json "name" field:**
Enter the project name that will be added to the package.json file
- **Name of the author to put in the package.json "author" field:**
Your name and e-mail following this pattern: Author Name <author-email@email.com>
- **Magento instance to use as a backend (will be added to `.env`` file):**
Choose a Magento 2 instance with Venia sample data installed. It will use a Magento instance with products and categories
- **Braintree API token to use to communicate with your Braintree instance (added to `.env`` file):**
Use the default value here. To use the default value just press "enter"
- **NPM package management client to use:**
The options are npm or yarn. You can choose either
- **Install package dependencies with yarn after creating project:**
If yes, it runs the yarn/npm install. If not, it finishes the configuration, and you'll need to run it manually.

Step 2

After step 1 you need to move into the project folder
`cd <project_folder>`

Step 3

If you chose not to install the dependencies in step 1, you need to install them now:
`npm install` or `yarn install`

Step 4

Next, generate the SSL certificate with:
`yarn run buildpack create-custom-origin .`

Step 5

To launch the project run the following command:
`yarn run watch`

That's it, the project should now run in the browser with the URL shown in the terminal.

Unit 3. Build & Dev Tools

3.2 Build CLI Commands

Exercise 3.2.1

Configure a new `.env` file setting these variables and values:

```
MAGENTO_BACKEND_URL=https://magento-instance.local  
IMAGE_OPTIMIZING_ORIGIN=backend  
CUSTOM_ORIGIN_EXACT_DOMAIN=pwa-project.local
```

Solution

Run these commands in the terminal:

```
MAGENTO_BACKEND_URL=https://magento-instance.local \  
IMAGE_OPTIMIZING_ORIGIN=backend \  
CUSTOM_ORIGIN_EXACT_DOMAIN=pwa-project.local \  
yarn run buildpack create-env-file .  
  
yarn run create-custom-origin .
```

Unit 4. Storefront Structure

4.1 Storefront Structure Introduction

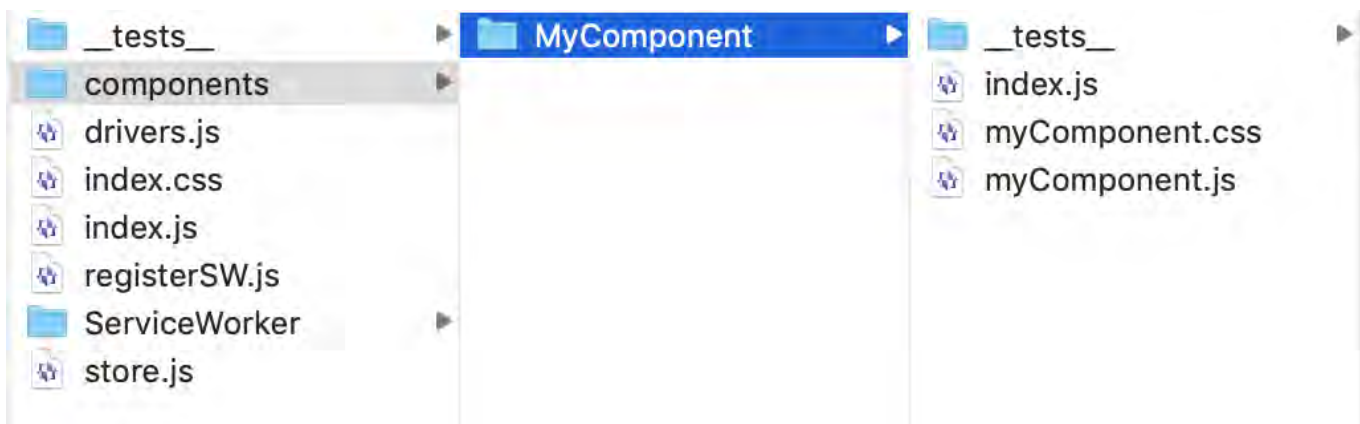
Exercise 4.1.1

Create the directory structure and key files for a custom component in your PWA storefront. You do not need to add content to these files yet.

1. Create the public API for your component
2. Create the component JavaScript file
3. Create stylesheet file
4. Create a unit testing directory

See the documentation for help: <https://github.com/magento/pwa-studio/wiki/Project-coding-standards-and-conventions>.

Solution



Step 1

1. From the root directory of your project, navigate to the src/ folder.
2. Within this folder, create a new folder, if it does not exist, called components.
3. In the components directory, create a new folder with the name of your component, in this case MyComponent.
4. Ensure the correct capitalization is used.

Step 2

In your MyComponent directory, create a new directory for __tests__

This is where any future unit tests will be added.

Step 3

In the MyComponent directory, create a new file called index.js. This is the public API for your new component.

For any other components to include your component, use

```
import { MyComponent } from '/src/components/MyComponent';
```

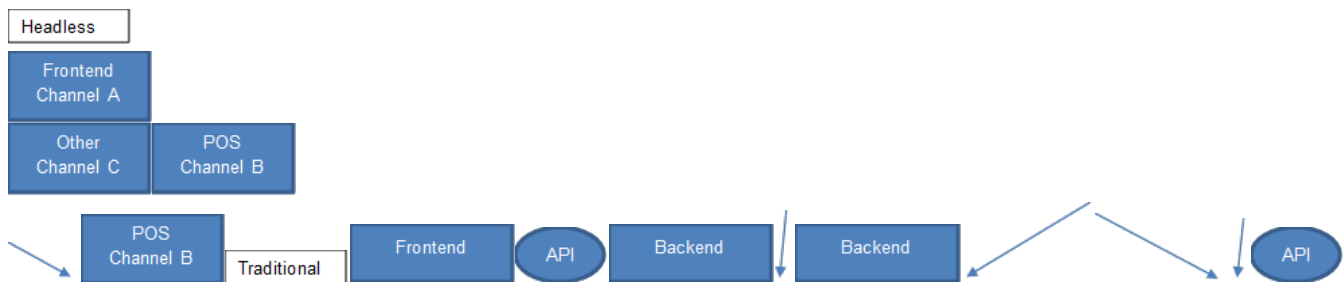
Step 4

Create the myComponent.css and myComponent.js files in the MyComponent directory.

4.2 Architecture of a Storefront**Exercise 4.2.1**

Diagram the difference from a headless solution and a coupled solution.

Show how the interface between the backend differs in both solutions.

Solution**Solution 2**

With a headless solution the interface to the backend is always through an API layer. All interfaces are decoupled and need to be created or attached.

The Traditional solution shows the frontend and the backend built together, and it is not possible to run the frontend independently of the backend. (However, it is possible to turn off the frontend and only use the backend in Magento.)

Unit 5. UPWARD Server

5.1 UPWARD overview

Exercise 5.1.1

Create UPWARD Demo Server

Create Basic Upward Configuration file that has 3 properties

- Status Code: 200
- Headers: content-type : text/html
- Body: Return "Hello World"

Start and Test your UPWARD server

Solution

Step 1. Create Upward Demo server

```
//Create new folder
% mkdir upward-demo
% cd upward-demo

//Run npm init and select default options to generate package.json file. See example below.
% npm init
```

```
manish@manish:~/Projects/upward-demo$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (upward-demo)
version: (1.0.0)
description:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/manish/Projects/upward-demo/package.json:

{
  "name": "upward-demo",
  "version": "1.0.0",
  "main": "upward.js",
  "scripts": {
    "start": "node upward-demo.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@magento/upward-js": "^4.0.0",
    "express": "^4.17.1"
  },
  "devDependencies": {},
  "description": ""
}
```

```
//Install upward js and express server (dependency)
% yarn add @magento/upward-js express
//Create upward-demo.js file
% touch upward-demo.js
//Edit upward-demo.js to add below.

const { createUpwardServer } = require('@magento/upward-js');

const { app } = createUpwardServer({
  logUrl: true,
  bindLocal: true,
  upwardPath: './upward.yml'
})
```

For further details on createUpwardServer configuration options, check <https://github.com/magento/pwa-studio/blob/develop/packages/upward-js/lib/createUpwardServer.js>

Step 2. Create Upward Specification File

```
//Create upward.yml file
% touch upward.yml
//Add basic configuration to the upward file. (see below)
status: 200
headers:
  inline:
    content-type:
      inline: text/html
body:
  inline: 'Hello World'
```

Step 3. Start your UPWARD server

```
//Start Upward Demo server
% node upward-demo.js

//This will start upward-demo server on available port.
//Refer createUpwardServer implementation to add a custom port
```

Step 4. Open UPWARD Server link in your browser and verify the output

Note: When configuring our UPWARD Server, we set configuration option **logUrl** to true. This will log all URLs served by our UPWARD server in the terminal.

5.3 UPWARD implementations

Exercise 5.3.1

1. Install the upward-php module in your Magento project.
2. Configure the upward-php module. Use the same upward.yml file from Exercise 5.1.
3. Check your frontend in a browser and observe the effect of installing the upward-php module.
4. Disable the upward-php module and check your frontend again.

Solution

Step 1. Install upward-php module in your Magento Project

```
//Add repository information to your composer.json file
% composer config repositories.upward-connector vcs https://github.com/magento-research/magento2-upward-connector

//install upward-php module using composer
% composer require magento/module-upward-connector
```

Step 2. Configure upward-php module

Copy upward.yml file you created in exercise 5.1 to your Magento root Folder.

The Magento 2 UPWARD connector is configured in the admin area under:

Stores > Configuration > General > Web > UPWARD PWA Configuration.

Browser Capabilities Detection

UPWARD PWA Configuration

UPWARD Config File <small>[global]</small>	<input type="text" value="PATH_TO_YOUR_MAGENTO_ROOT_FOLDER/upward.yml"/>
	<small>Server path to YAML configuration file for UPWARD</small>
Front Name Whitelist <small>[store view]</small>	<input type="text"/>
	<small>Line break separated list of URL's that will load the default Magento frontend.</small>

Enter the absolute path on the server for the value of this configuration.

Save the configuration and refresh required cache types.

Step 3. Check the storefront of your Magento application

You will notice that your Magento storefront is now replaced by “Hello World” text coming from UPWARD

Step 4. Disable upward-php module

```
//Disable Magento_UpwardConnector
% bin/magento module:disable Magento_UpwardConnector
```

Step 5. Check the frontend of your Magento application

You will notice that your Magento storefront theme is back.

5.4 UPWARD Configuration

Exercise 5.4.2

Use Context Lookup to rewrite/reduce the configuration file provided in the Notes section.

```
body:
  resolver: conditional
  when:
    - matches: request.url.query.shipmentId
      pattern: '\d+'
      use: getShipment
    - matches: request.url.query.shipmentName
      pattern: '\w+'
      use: getShipment
    - matches: request.url.query.shipmentTrackingNumber
      pattern: '[\w\d\.]+'
      use: getShipment
  default:
    inline: 'Please provide a query'

getShipment:
  resolver: template
  engine: mustache
  template:
    resolver: file
    file:
      resolver: inline
      inline: './renderShipment.mst'
  provide:
    shipment:
      resolver: service
      url: env.SHIPMENTS_SVC
      query:
        resolver: file
        file:
          resolver: inline
          inline: './getShipment.graphql'
    variables:
      id: request.url.query.shipmentId
      name: request.url.query.shipmentName
      trackingNumber: request.url.query.trackingNumber
```

Unit 6. Working with Components

6.1 Child Module Replacement

Exercise 6.1.1

In this solution we use Webpack to override components.

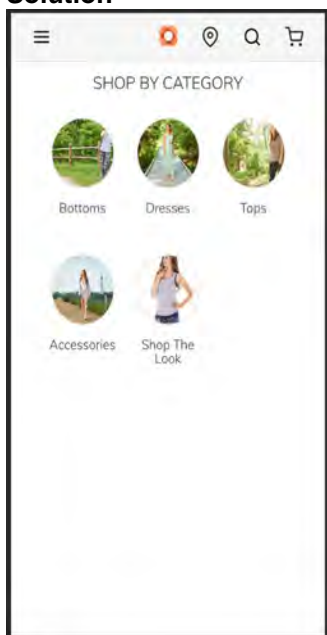
You can find the code in the zipped file [unit-components-webpack-aliases.zip](#).

In [unit-components-tree-override.zip](#) you can find the same solution using the tree replacement.

Add a new element in the Header component.

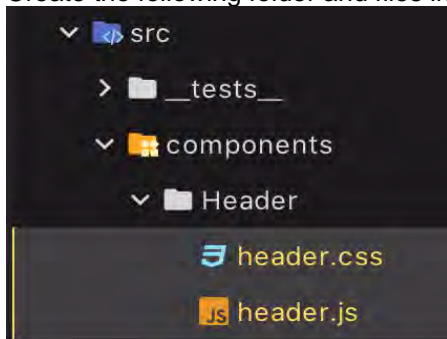
1. Copy and replace the Header component.
2. Add a new element in the custom component.
3. BONUS: The new element is responsive.

Solution



Step 1

Create the following folder and files in the **src** folder.



header.js needs to contain the code from **@magento/venia-ui/lib/components/Header/header.js**

Step 2

Replace Venia UI Header component using a Webpack alias.

In **webpack.config.js**, add the following lines before the end.

```
// Custom code - Start
const moduleReplacement = {
  './header': 'src/components/Header/header.js',
};
// Custom code - Finish

clientConfig.resolve.alias = {...clientConfig.resolve.alias, ...moduleReplacement};

return [clientConfig, serviceWorkerConfig];
```

This will replace the default component with your own.

Note that every time you update this file, you need to restart your watch process.

Step 3

In **src/components/Main/main.js**, update all the imports so they point to the right files.

```
/* From this */
import Logo from '../Logo';
import { Link, resourceUrl, Route } from '@magento/venia-drivers';

import CartTrigger from './cartTrigger';
import NavTrigger from './navTrigger';
import SearchTrigger from './searchTrigger';
import OnlineIndicator from './onlineIndicator';
import { useHeader } from '@magento/peregrine/lib/talons/Header/useHeader';

import { mergeClasses } from '../../classify';
import defaultClasses from './header.css';

const SearchBar = React.lazy(() => import('../SearchBar'));

/* To this */
import Logo from '@magento/venia-ui/lib/components/Logo';
import { Link, resourceUrl, Route } from '@magento/venia-drivers';

import CartTrigger from '@magento/venia-ui/lib/components/Header/cartTrigger';
import NavTrigger from '@magento/venia-ui/lib/components/Header/navTrigger';
import SearchTrigger from '@magento/venia-ui/lib/components/Header/searchTrigger';
import OnlineIndicator from '@magento/venia-ui/lib/components/Header/onlineIndicator';
import { useHeader } from '@magento/peregrine/lib/talons/Header/useHeader';

import { mergeClasses } from '@magento/venia-ui/lib/classify';
import defaultClasses from '@magento/venia-ui/lib/components/Header/header.css';

const SearchBar = React.lazy(() => import('@magento/venia-ui/lib/components/SearchBar'));
```

Step 4

Create your new elements.

Here, we will create a link to a fake page and use an icon from React Feather.

We also import our custom styles.

```
import customClasses from './header.css';
import Icon from "@magento/venia-ui/lib/components/Icon";
import { MapPin } from 'react-feather';
```

We need to add our custom styles to the classes object using mergeClasses().

```
const classes = mergeClasses(defaultClasses, props.classes, customClasses);
```

You can then add your link and icon to the header.

```
<div className={customClasses.secondaryActions}>
  {/* Custom Code - Start */}
  <Link
    to={resourceUrl('/stores')}
    className={customClasses.storesTrigger}
  >
    <Icon src={MapPin} />
  </Link>
  {/* Custom Code - End */}
  <SearchTrigger
    active={searchOpen}
    onClick={handleSearchTriggerClick}
  />
  <CartTrigger />
</div>
```

Step 5

Add your new styles and import them.

In **src/lib/components/Header/header.css**, add the following lines.

```
/* Overrides existing class, imports their styles and updates them */
.secondaryActions {
  composes: secondaryActions from '~@magento/venia-ui/lib/components/Header/header.css';

  grid-template-columns: 1fr 1fr 1fr;
}

/* New class for our new element */
/* The styles from clickable.css is used on multiple other elements */
.storesTrigger {
  composes: root from '~@magento/venia-ui/lib/components/clickable.css';
  height: 3rem;
  min-width: 3rem;
}
```

Step 6

If you haven't done it yet, restart your watch process to see the updates.

6.2 Overriding a Component's Style

Exercise 6.2.1

In this solution we use Webpack to override components.

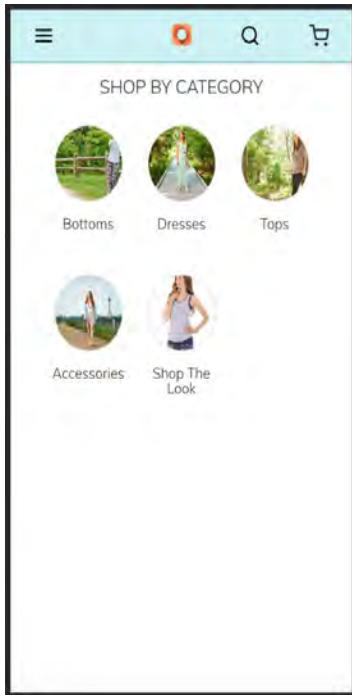
You can find the code in the zipped file [unit-components-webpack-aliases.zip](#).

In [unit-components-tree-override.zip](#) you can find the same solution using the tree replacement.

Update the Header component using custom styles.

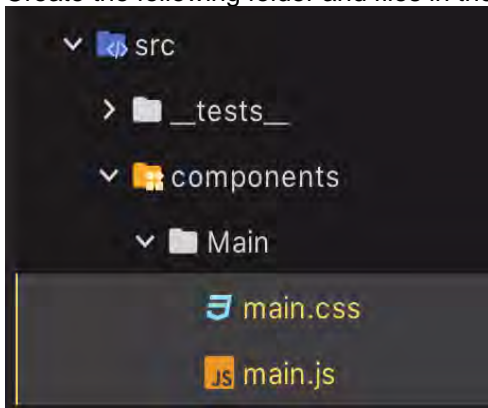
1. Pass the custom styles by using the component's props.
2. Apply the styles to an existing class.

Solution



Step 1

Create the following folder and files in the **src** folder.



main.js needs to contain the code from **@magento/venia-ui/lib/components/Main/main.js**

Step 2

Replace Venia UI Main component using a Webpack alias.

In **webpack.config.js**, add the following lines before the end.

```
// Custom code - Start
const moduleReplacement = {
  './main': 'src/components/Main/main.js',
};
// Custom code - Finish

clientConfig.resolve.alias = {...clientConfig.resolve.alias, ...moduleReplacement};

return [clientConfig, serviceWorkerConfig];
```

This will replace the default component with your own.

Note that every time you update this file, you need to restart your watch process.

Step 3

In **src/components/Main/main.js**, update all the imports so they point to the right files.

```
/* From this */
import { mergeClasses } from '../../classify';
import Footer from '../Footer';
import Header from '../Header';
import defaultClasses from './main.css';

/* To this */
import { mergeClasses } from '@magento/venia-ui/lib/classify';
import Footer from '@magento/venia-ui/lib/components/Footer';
import Header from '@magento/venia-ui/lib/components/Header';
import defaultClasses from '@magento/venia-ui/lib/components/Main/main.css';
```

Step 4

Add your new styles and import them.

In **src/components/Main/main.css**, add the following lines.

```
:root {
  --header-bg-color: 205, 241, 243;
  --header-border-color: 39, 162, 169;
}

.root {
  composes: root from '~@magento/venia-ui/lib/components/Header/header.css';

  background-color: rgb(var(--header-bg-color));
  border-bottom: 1px solid rgb(var(--header-border-color));
}

.open {
  composes: root;
}
```

```
.closed {  
  composes: root;  
}
```

In `src/components/Main/main.js`, import your CSS.

```
import customHeaderClasses from './main.css';
```

Step 5

Connect your custom styles to the Header Component.

In `src/components/Main/main.js`, modify the Header component like this.

```
<Header  
  classes={{  
    open: customHeaderClasses.open,  
    closed: customHeaderClasses.closed  
  }}  
>
```

Step 6

If you haven't done it yet, restart your watch process to see the updates.

6.3 Creating a New Storefront Component

Exercise 6.3.1

In this solution we use Webpack to override components.

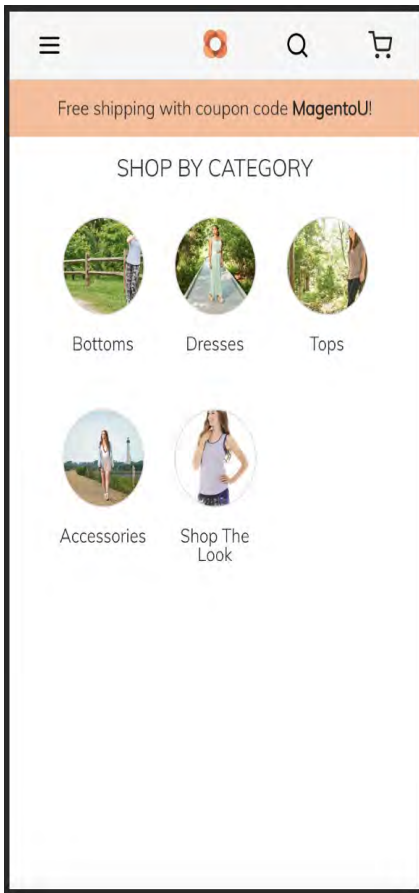
You can find the code in the zipped file [unit-components-webpack-aliases.zip](#).

In [unit-components-tree-override.zip](#) you can find the same solution using the tree replacement.

Create a TopBanner component under the Header component.

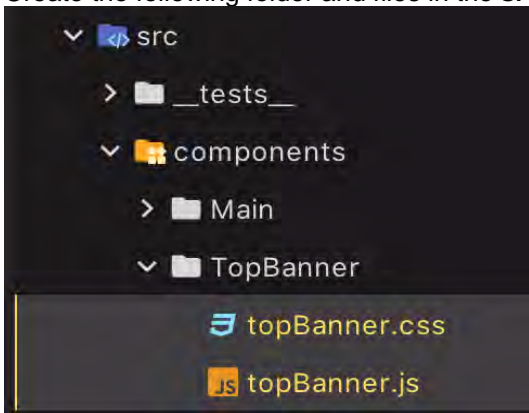
1. Create the new folders and files for the component.
2. List the prop-types of the props the element can receive.
3. Create styles for the component.
4. Import and instantiate the TopBanner component under the Header component.

Solution



Step 1

Create the following folder and files in the **src** folder.



The Main Component is used so we can instantiate the custom component.

You can follow the steps in 6.1 to create the override.

Step 2

In `src/components/TopBanner/topBanner.js`, copy the following code.

```
import React from 'react';

import { shape, string } from 'prop-types';
import { mergeClasses } from '@magento/venia-ui/lib/classify';
import defaultClasses from './topBanner.css';

const TopBanner = props => {
  const classes = mergeClasses(defaultClasses, props.classes);

  return (
    <div className={classes.root}>
      Free shipping with coupon code <strong>MagentoU</strong>!
    </div>
  );
};

TopBanner.propTypes = {
  classes: shape({
    root: string
  })
};

export default TopBanner;
```

Step 3

In `src/components/TopBanner/topBanner.css`, copy the following code.

```
.root {
  position: sticky;
  top: 3.5rem;
  z-index: 1;
  background: #fbbc97;
  text-align: center;
  padding: 1em;
}
```

Step 4

In `src/components/Main/main.js`, import and instantiate your custom component.

```
import TopBanner from "../TopBanner/topBanner";

/* === */

<main className={rootClass}>
  <Header />

  <TopBanner /> { /* Custom Component */}

  <div className={pageClass}>{children}</div>
  <Footer />
</main>
```

Step 5

If you haven't done it yet, restart your watch process to see the updates.

Unit 7. Peregrine Hooks & Talons

7.2 Peregrine Hooks

Exercise 7.2.1

Using the Peregrine library, create a ScrollToggle component on the bottom right of the app that toggles browser scrolling when clicked.

Solution

Part 1 – Project setup

If you already have a storefront project set up, you can skip to **Part 2**.

Step 1 – Run the scaffolding tool

On the command line, run the following to generate a new storefront project:

```
npx @magento/create-pwa
```

Step 2 – Answer the questions about the project

On the interactive questionnaire, use the following values as answers:

```
? Project root directory (will be created if it does not exist) .
? Short name of the project to put in the package.json "name" field my-pwa-storefront
? Name of the author to put in the package.json "author" field Freddy Frontendgineer
? Magento instance to use as a backend (will be added to `.env` file) 2.3.3-venia-cloud
? Braintree API token to use to communicate with your Braintree instance (will be added
to `.env` file) sandbox_8yrzsvtm_s2bg8fs563crhqzk
? NPM package management client to use yarn
? Install package dependencies with yarn after creating project Yes
```

Part 2 – Create the necessary project files

Step 1 – Create the component folder

Using the command line, navigate to your project's root directory and create a ScrollToggle folder under the src/lib/components directories:

```
mkdir -p src/components/ScrollToggle
```


Step 2 – Copy over the provided style definitions

Inside the new ScrollToggle directory, create a file called scrollToggle.css with the following content:

```
.root {
  position: fixed;
  bottom: 5px;
  right: 5px;
  z-index: 1;
  border: solid 1px;
  padding: 5px;
}

.rootActive {
  composes: root;
  background-color: rgb(134, 133, 133);
}

.rootInactive {
  composes: root;
  background-color: #ffffff;
}
```

Step 3 – Create the component

Inside the ScrollToggle directory, create a file called index.js with the following content:

```
import React, { useState } from 'react';

// Import the Peregrine hook for locking the scroll feature
import { useScrollLock } from '@magento/peregrine';

// Import the style sheet as a CSS module
import defaultClasses from './scrollToggle.css';

// ScrollToggle component definition
const ScrollToggle = () => {
  // The Peregrine hook does not maintain the scroll state, so
  // this component will maintain it
  const [locked, setLocked] = useState(false);

  // Set the scroll lock to the value of locked every time the
  // component renders
  useScrollLock(locked);

  // Determine which class to use based on the current locked state
  const classes = locked
    ? defaultClasses.rootActive
    : defaultClasses.rootInactive;

  // Define the handler for click events on this component
  const clickHandler = () => {
    // Toggle the value of the locked state
    // This causes the component to re-render causing it to
```

```
    // call the Peregrine hook with the new state
    setLocked(!locked);
  };

  return (
    <button onClick={clickHandler} className={classes}>
      Scroll lock
    </button>
  );
};

export default ScrollToggle;
```

Part 3 – Add the project files to your storefront app

Step 1 – Import the new component

At the top of your project's `src/index.js` file, import the `ScrollToggle` component:

```
import './index.css';

import ScrollToggle from './components/ScrollToggle'

const { BrowserPersistence } = Util;
```

Step 2 – Add the component to the rendered DOM

Inside the `ReactDOM.render()` function call, add the component to the DOM tree:

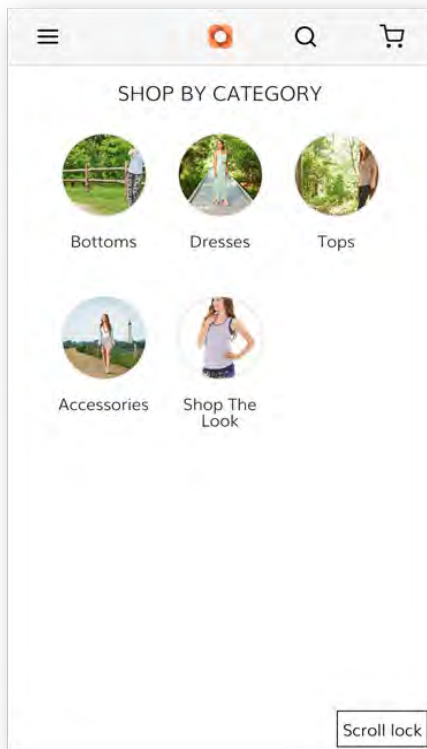
```
ReactDOM.render(
  <Adapter apiBase={apiBase} apollo={{ link: apolloLink }} store={store}>
    <AppContextProvider>
      <App />
      <ScrollToggle/>
    </AppContextProvider>
  </Adapter>,
  document.getElementById('root')
);
```

Step 3 – Run the app

Use the following command to run the dev server:

```
yarn watch
```

Now, when you navigate to the app in your browser, you will see the button. Click on it to lock scrolling.



7.3 Peregrine Talons

Exercise 7.3.1

Using the ScrollToggle component solution you created in exercise 7.2, extract the component logic into an appropriately named talon and import it into the UI component.

Solution

Part 1 – Project setup

If you already have a storefront project set up, you can skip to **Part 2**.

Step 1 – Run the scaffolding tool

On the command line, run the following to generate a new storefront project:

```
npx @magento/create-pwa
```

Step 2 – Answer the questions about the project

On the interactive questionnaire, use the following values as answers:

```
? Project root directory (will be created if it does not exist) .
? Short name of the project to put in the package.json "name" field my-pwa-storefront
? Name of the author to put in the package.json "author" field Freddy Frontendgineer
```

```
? Magento instance to use as a backend (will be added to `.env` file) 2.3.3-venia-cloud  
? Braintree API token to use to communicate with your Braintree instance (will be added  
to `.env` file) sandbox\_8yrzsvtm\_s2bg8fs563crhqzk  
? NPM package management client to use yarn  
? Install package dependencies with yarn after creating project Yes
```

Part 2 – Create the 7.2 exercise solution files

If you already have the 7.2 exercise solution in your project, you can skip to **Part 3**.

Step 1 – Create the component folder

Using the command line, navigate to your project's root directory and create a ScrollToggle folder under the src/lib/components directories:

```
mkdir -p src/components/ScrollToggle
```

Step 2 – Copy over the provided style definitions

Inside the new ScrollToggle directory, create a file called scrollToggle.css with the following content:

```
.root {  
  position: fixed;  
  bottom: 5px;  
  right: 5px;  
  z-index: 1;  
  border: solid 1px;  
  padding: 5px;  
}  
  
.rootActive {  
  composes: root;  
  background-color: rgb(134, 133, 133);  
}  
  
.rootInactive {  
  composes: root;  
  background-color: #ffffff;  
}
```

Step 3 – Create the component

Inside the ScrollToggle directory, create a file called index.js with the following content:

```
import React, { useState } from 'react';  
  
// Import the Peregrine hook for locking the scroll feature  
import { useScrollLock } from '@magento/peregrine';  
  
// Import the style sheet as a CSS module  
import defaultClasses from './scrollToggle.css';  
  
// ScrollToggle component definition  
const ScrollToggle = () => {  
  // The Peregrine hook does not maintain the scroll state, so  
  // this component will maintain it
```

```

const [locked, setLocked] = useState(false);

// Set the scroll lock to the value of locked every time the
// component renders
useScrollLock(locked);

// Determine which class to use based on the current locked state
const classes = locked
  ? defaultClasses.rootActive
  : defaultClasses.rootInactive;

// Define the handler for click events on this component
const clickHandler = () => {
  // Toggle the value of the locked state
  // This causes the component to re-render causing it to
  // call the Peregrine hook with the new state
  setLocked(!locked);
};

return (
  <button onClick={clickHandler} className={classes}>
    Scroll lock
  </button>
);
};

export default ScrollToggle;

```

Step 4 – Import the new component

At the top of your project's `src/index.js` file, import the `ScrollToggle` component:

```

import './index.css';
import ScrollToggle from './components/ScrollToggle'
const { BrowserPersistence } = Util;

```

Step 5 – Add the component to the rendered DOM

Inside the `ReactDOM.render()` function call, add the component to the DOM tree:

```

ReactDOM.render(
  <Adapter apiBase={apiBase} apollo={{ link: apolloLink }} store={store}>
    <AppContextProvider>
      <App />
      <ScrollToggle/>
    </AppContextProvider>
  </Adapter>,
  document.getElementById('root')
);

```

Part 3 – Extract the talon

Step 1 – Remove the logic pieces

Edit the ScrollToggle/index.js file and remove the lines in red and add the lines in green:

```
import React, { useState } from 'react';
import React from 'react';

// Import the Peregrine hook for locking the scroll feature
import { useScrollLock } from '@magento/peregrine';

// Import the style sheet as a CSS module
import defaultClasses from './scrollToggle.css';

// ScrollToggle component definition
const ScrollToggle = () => {
  // The Peregrine hook does not maintain the scroll state, so
  // this component will maintain it
  const [locked, setLocked] = useState(false);

  // Set the scroll lock to the value of locked every time the
  // component renders
  useScrollLock(locked);

  // Determine which class to use based on the current locked state
  const classes = locked
    ? defaultClasses.rootActive
    : defaultClasses.rootInactive;

  // Define the handler for click events on this component
  const clickHandler = () => {
    // Toggle the value of the locked state
    // This causes the component to re-render causing it to
    // call the Peregrine hook with the new state
    setLocked(!locked);
  };

  return (
    <button onClick={clickHandler} className={classes}>
      Scroll lock
    </button>
  );
};

export default ScrollToggle;
```

Step 2 – Create the talon for the component

Create a `useScrollToggle.js` file in your component folder with the following content:

```
import { useState } from 'react';

import { useScrollLock } from '@magento/peregrine';

const useScrollToggle = () => {
  const [locked, setLocked] = useState(false);

  useScrollLock(locked);

  const clickHandler = () => {
    setLocked(!locked);
  };

  return {
    locked,
    clickHandler
  };
};

export default useScrollToggle
```

Step 3 – Import and use the talon

Edit the `ScrollToggle/index.js` file and add the lines in **green**:

```
import React from 'react';

// Import this component's talon
import useScrollToggle from './useScrollToggle';

// Import the style sheet as a CSS module
import defaultClasses from './scrollToggle.css';

// ScrollToggle component definition
const ScrollToggle = () => {
  // Call the talon and destructure the return object
  const { locked, clickHandler } = useScrollToggle();

  // Determine which class to use based on the current locked state
  const classes = locked
    ? defaultClasses.rootActive
    : defaultClasses.rootInactive;

  return (
    <button onClick={clickHandler} className={classes}>
      Scroll lock
    </button>
  );
};

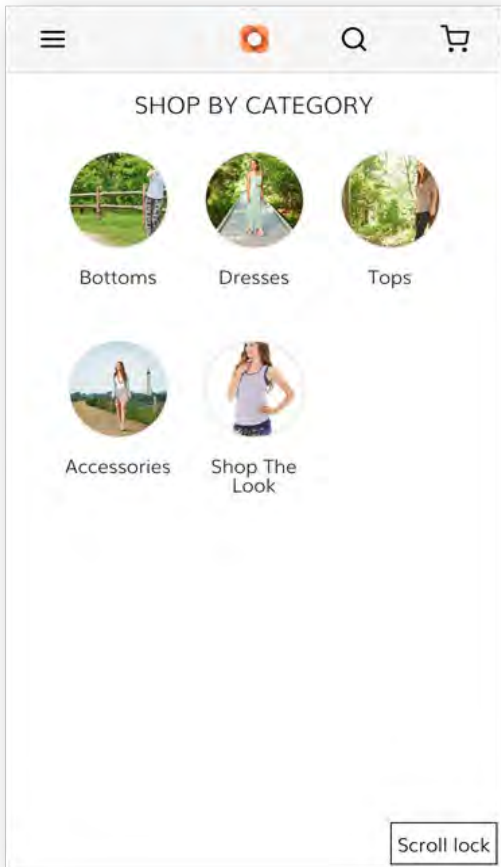
export default ScrollToggle;
```

Step 4 – Run the app

Use the following command to run the dev server:

```
yarn watch
```

Now, when you navigate to the app in your browser, you will see the button. Click on it to lock scrolling.



Unit 8. Application State Management

8.1 Application State Data

Exercise 8.1.1

Create a react component that uses a custom peregrine hook to read user data from the context and renders the following:

1. If the user is logged in, collect first name and last name from the custom context and print the following message on the UI: **Hello FIRSTNAME LASTNAME**
2. If the user is not logged in, show a message that says **Please Sign In**.

Solution

```
import { useUserContext } from '@magento/peregrine/lib/context/user';

function useCustomUserContext() {
  const [{ isSignedIn, currentUser }] = useUserContext();











  return {
    isSignedIn,
    firstName: currentUser.firstname,
    lastName: currentUser.lastname
  };
}

function GreetingComponent() {
  const { isSignedIn, firstName, lastName } = useCustomUserContext();
  const message = isSignedIn
    ? `Hello ${firstName} ${lastName}`
    : 'Please Sign In';

  return <div style={{ textAlign: 'center' }}>{message}</div>;
}
```

First, we are creating a custom user context hook with the name **useCustomUserContext** that picks up information it needs from the **useUserContext** hook. Then we are using this custom hook in the **GreetingComponent**.

To know if the user has signed in, we use the **isSignedIn** flag from the custom hook. If the user is signed in, we create a greeting message using the user's first and last name. If the user is not signed in, set the message as **Sign In**.

Please Sign In	Hello Revanth Annavarapu
<p data-bbox="240 359 576 394">SHOP BY CATEGORY</p> <div data-bbox="154 443 293 583"></div> <p data-bbox="167 604 280 638">Bottoms</p> <div data-bbox="342 443 482 583"></div> <p data-bbox="358 604 464 638">Dresses</p> <div data-bbox="531 443 670 583"></div> <p data-bbox="563 604 630 638">Tops</p> <div data-bbox="154 709 293 850"></div> <p data-bbox="147 871 298 905">Accessories</p> <div data-bbox="342 709 482 850"></div> <p data-bbox="349 871 470 930">Shop The Look</p>	<p data-bbox="868 359 1205 394">SHOP BY CATEGORY</p> <div data-bbox="782 443 922 583"></div> <p data-bbox="792 604 906 638">Bottoms</p> <div data-bbox="971 443 1110 583"></div> <p data-bbox="984 604 1089 638">Dresses</p> <div data-bbox="1159 443 1299 583"></div> <p data-bbox="1188 604 1255 638">Tops</p> <div data-bbox="782 709 922 850"></div> <p data-bbox="773 871 924 905">Accessories</p> <div data-bbox="971 709 1110 850"></div> <p data-bbox="976 871 1097 930">Shop The Look</p>

8.1 Update Application State

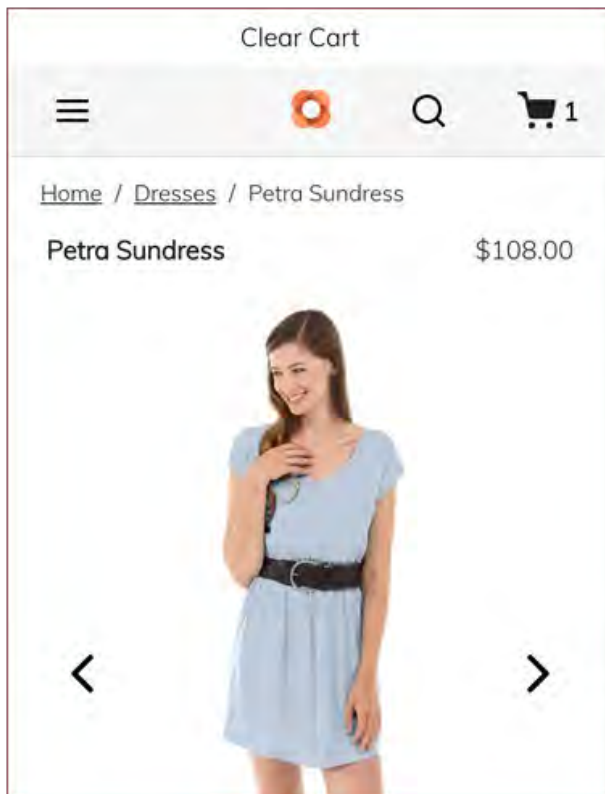
Exercise 8.2.1

Write a custom peregrine context hook which uses cart context's **removeCart** function to clear the cart when a user clicks on a mock button on the home page.

1. Create a custom context hook.
2. Define and return a function to remove the cart when called. Also return if the cart is empty.
3. Create a button component and use the custom context's clear cart function and call it when the user clicks on the button. Disable the button if the cart is empty.

Note: Use the following style for the button.

```
.clearButton {  
  padding: 0.5rem 9rem;  
}
```



Solution

```
import { useCartContext } from '@magento/peregrine/lib/context/cart';

function useCustomCartContext() {
  const [{ isEmpty }, { removeCart }] = useCartContext();

  const clearCart = useCallback(() => {
    removeCart();
    console.log('Cleared cart');
  }, [removeCart]);

  return [
    { isEmpty },
    {
      clearCart
    }
  ];
}

function ClearCartButton() {
  const [{ isEmpty }, { clearCart }] = useCustomCartContext();

  return (
    <button
      style={{ padding: '0.5rem 9rem' }}
      onClick={clearCart}
      disabled={isEmpty}
    >
      Clear Cart
    </button>
  );
}
```

We are creating a custom cart context hook called `useCustomCartContext` which picks up `isEmpty` flag and the `removeCart` cart API function from `useCartContext`.

Next we create a function called `clearCart` which uses the `removeCart` function to clear the cart and prints a message in the console.

We return both the flag and the newly created function which will be used by the `ClearCartButton` component to render a button.

The button's click handler is set to the `clearCart` function from the `useCustomCartContext` hook and it is disabled if the cart is empty.

8.3 Create a New Application State

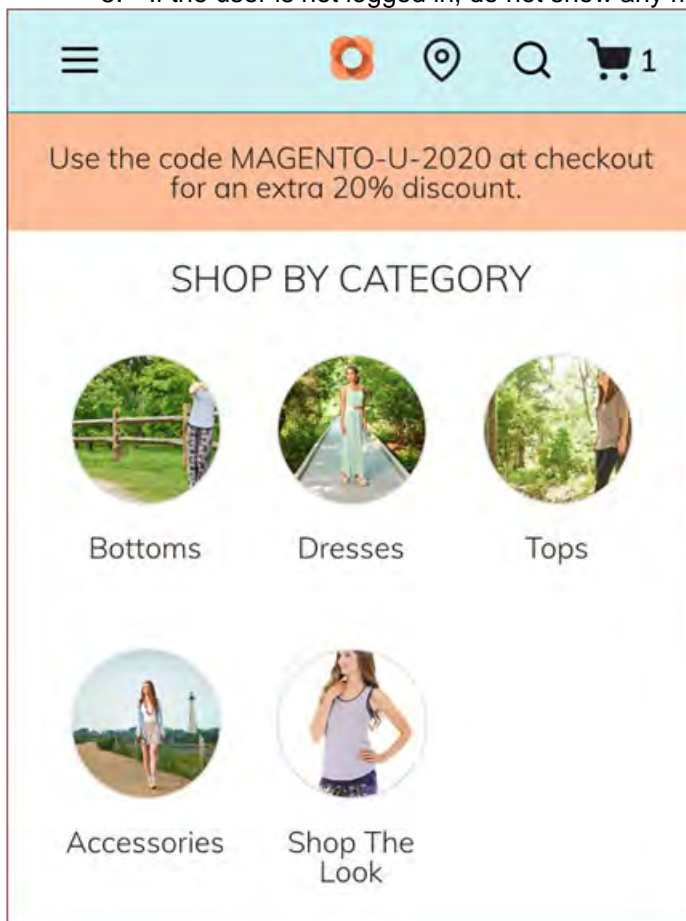
Exercise 8.3.1

This exercise uses the **TopBanner** component implemented in Section 6.3.

1. Create custom context to store a sample coupon code.
2. Create a provider and a hook for the new coupon context.
3. Change the **TopBanner** component to consume the coupon code context and the user's first name and last name from the user context.
4. **TopBanner** component should render the following if the user has logged in:

*Use the code **COUPON_CODE** at checkout for an extra 20% discount.*

5. If the user is not logged in, do not show any message on the top.



Solution

Step 1

Create a file for the context code. Use the react API to create context, its consumer and the provider.

```
import { useContext, createContext } from 'react'

const CouponCodeContext = createContext();

export const CouponCodeContextProvider = CouponCodeContext.Provider;

export const useCouponCodeContext = () => useContext(CouponCodeContext);
```

Step 2

Import the provider into the main app file and wrap the app inside the coupon context.

```
import { CouponCodeContextProvider } from './contexts/couponContext.js';

...

ReactDOM.render(
  <Adapter apiBase={apiBase} apollo={{ link: apolloLink }} store={store}>
    <AppContextProvider>
      <CouponCodeContextProvider value="MAGENTO-U-2020">
        <App />
      </CouponCodeContextProvider>
    </AppContextProvider>
  </Adapter>,
  document.getElementById('root')
);
```

Step 3

Modify the **TopBanner** component to consume the coupon context and user context using their respective context hooks.

```
import { useUserContext } from '@magento/peregrine/lib/context/user';

import { useCouponCodeContext } from '../contexts/couponContext';

const TopBanner = props => {
  const classes = mergeClasses(defaultClasses, props.classes);
  const couponCode = useCouponCodeContext();
  const [{ isSignedIn }] = useUserContext();

  return isSignedIn ? (
    <div className={classes.root}>
      { `Use the code ${couponCode} at checkout for an extra 20% discount.` }
    </div>
  ) : null;
};
```

Unit 9. REST & GraphQL

9.1 Introduction to Magento Cloud

Exercise 9.1.1

Use the same PWA project which you have setup in earlier Unit and do following items

1. Get GraphQL playground URL
2. Open GraphQL playground URL in the browser
3. Use the API query queries/getProductDetailBySku.graphql
4. Get the Swagger URL of Magento 2 setup
5. Open swagger URL in the browser
6. Use the GET API Request for catalogProductsRepositoryV1/Products/{sku}
7. Try to retrieve only Product Name using both API call and share the result

Solution

Step 1

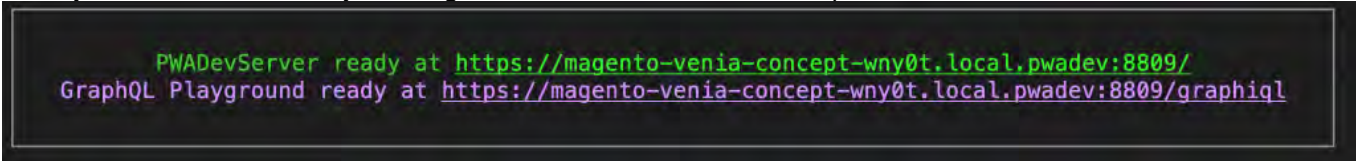
Get GraphQL playground URL.

We can achieve these two ways, either running command or jus directly putting word “graphql” at the end of base URL.

Way 1:

1. Open terminal
2. Run command: *yarn run watch:all*

Once you run this command you can get URLs, as shown in this example



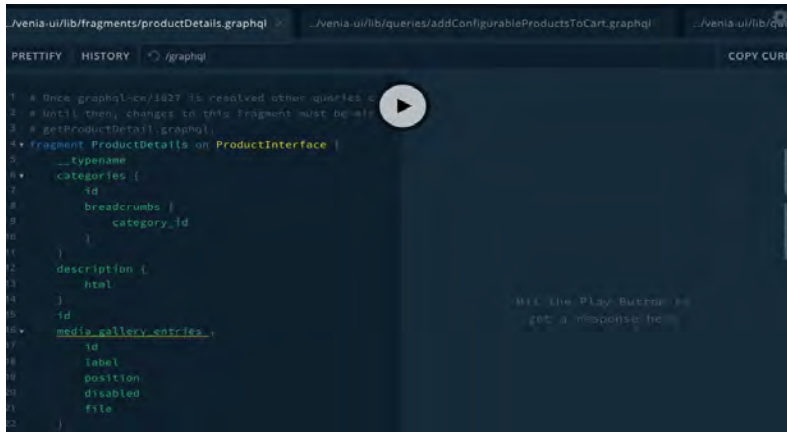
```
PWADevServer ready at https://magento-venia-concept-wny0t.local.pwadev:8809/
GraphQL Playground ready at https://magento-venia-concept-wny0t.local.pwadev:8809/graphql
```

Way 2: Make a URL with format https://PWA_STOREFRONT_URL/graphql

Step 2

URL which we received from step 1, open that URL in your browser.

You will see an example like this



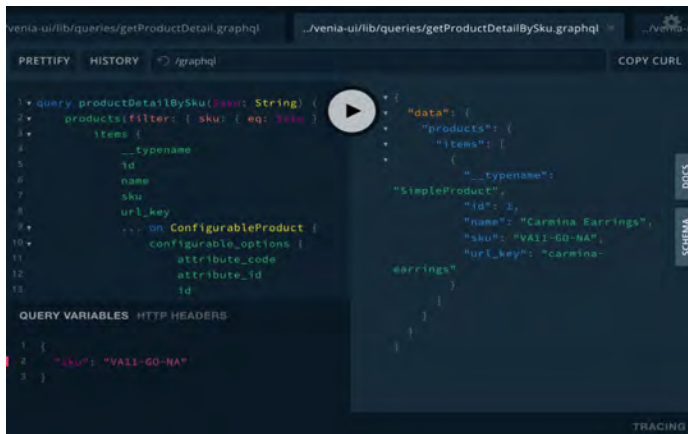
Step 3

On that screen at top of the page, you can find one horizontal tabs with scroller with different GraphQL queries.

Search for **getProductDetailBySKU.graphql** and click on it so this takes you to the specific query which return product information by sku.

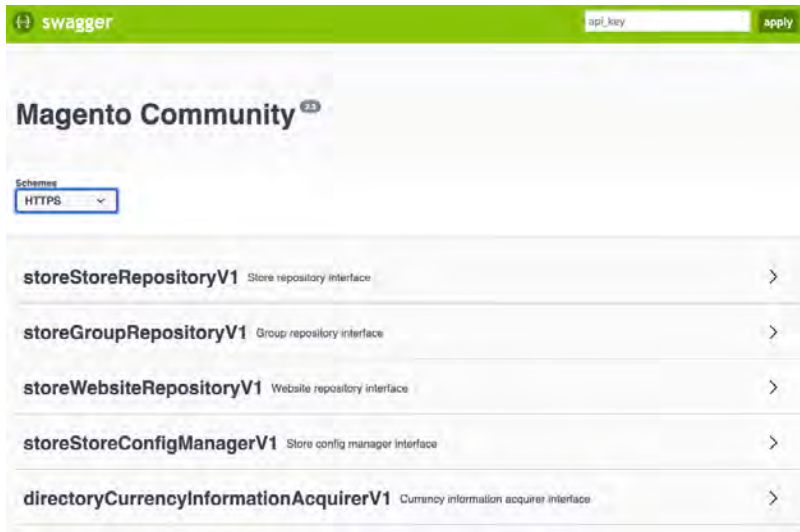
Step 4

At left bottom there is a Variable section. Click on it and add variable SKU as shown. Click Play so you can see the result in the right panel.



Step 5

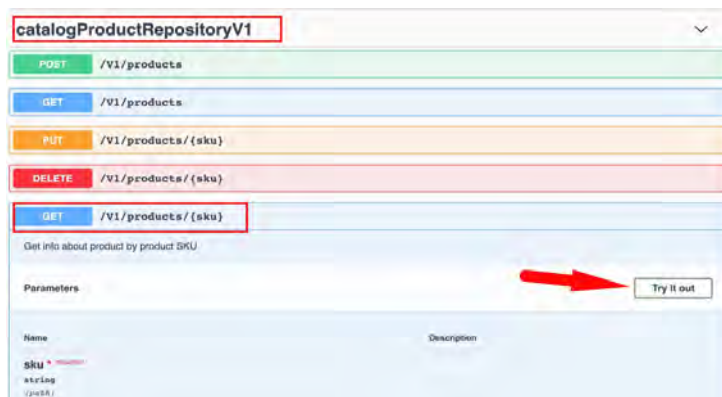
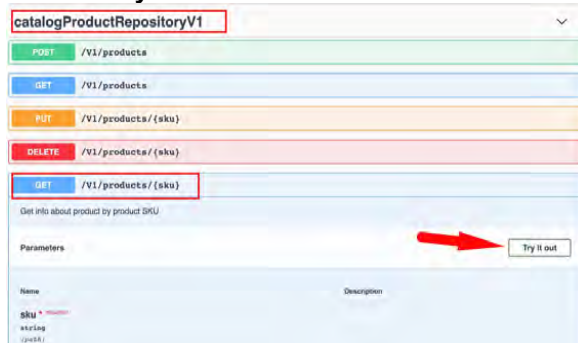
1. Prepare Swagger URL.
Swagger URL format: https://MAGENTO_BASE_URL/swagger
2. Open this swagger URL in browser.



Step 6

Now search for API “catalogProductRepositoryV1” and find the API call called “V1/products/{sku}”

Click on **Try it out** button



Step 7

This will open form where you can add SKU and click on **Execute** button

GET /v1/products/{sku}

Get info about product by product SKU

Parameters Cancel

Name	Description
sku *required string (path)	<input type="text" value="sku"/>
editMode boolean (query)	--
storeId integer (query)	storeId
forceReload boolean (query)	--

Execute

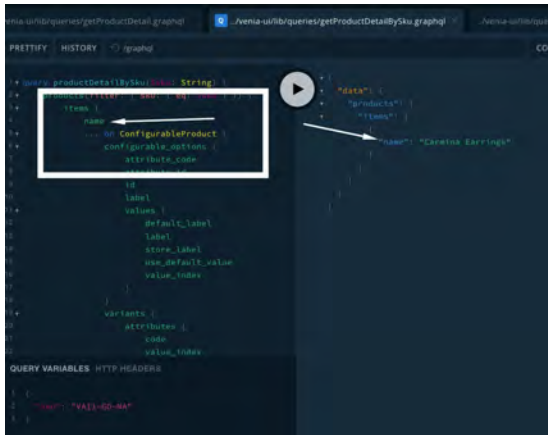
Step 8

It will show the response like below

```
Code Details
200 Response body
{
  "id": 20,
  "sku": "XXXXXXXXXXXXXXXXXXXX",
  "name": "XXXXXXXXXXXXXXXXXXXX",
  "attribute_set_id": 4,
  "price": 20,
  "status": 1,
  "visibility": 4,
  "type_id": "simple",
  "created_at": "2019-10-16 15:54:44",
  "updated_at": "2020-02-07 09:51:20",
  "extension_attributes": {
    "website_id": 1
  }
  "category_links": [
    {
      "position": 0,
      "category_id": "20"
    }
  ]
  "stock_item": {
    "item_id": 20,
    "product_id": 20,
    "stock_id": 1,
    "qty": 100,
    "is_in_stock": true,
    "is_qty_decimal": false,
    "show_default_notification_message": false,
    "use_config_notify_qty": true,
    "notify_qty": 0,
    "use_config_notify_qty": true,
    "notify_qty": 1,
    "use_config_notify_qty": true,
    "notify_qty": 10000,
    "use_config_backorders": true,
    "backorders": 0,
    "use_config_notify_stock_qty": true,
    "notify_stock_qty": 1,
    "use_config_notify_stock_qty": true,
    "notify_stock_qty": 0,
    "use_config_notify_stock_qty": true,
    "notify_stock_qty": 10000,
    "use_config_notify_stock_qty": true,
    "notify_stock_qty": 10000,
    "is_decimal_divided": false,
    "manage_stock": true,
    "low_stock_date": null,
    "is_decimal_divided": false,
    "stock_status_changed_qty": 0
  }
}
```

Step 9

Try to edit query and get only Name field.



When you try to edit the query you get only name field. This works with only GraphQL not with REST API.

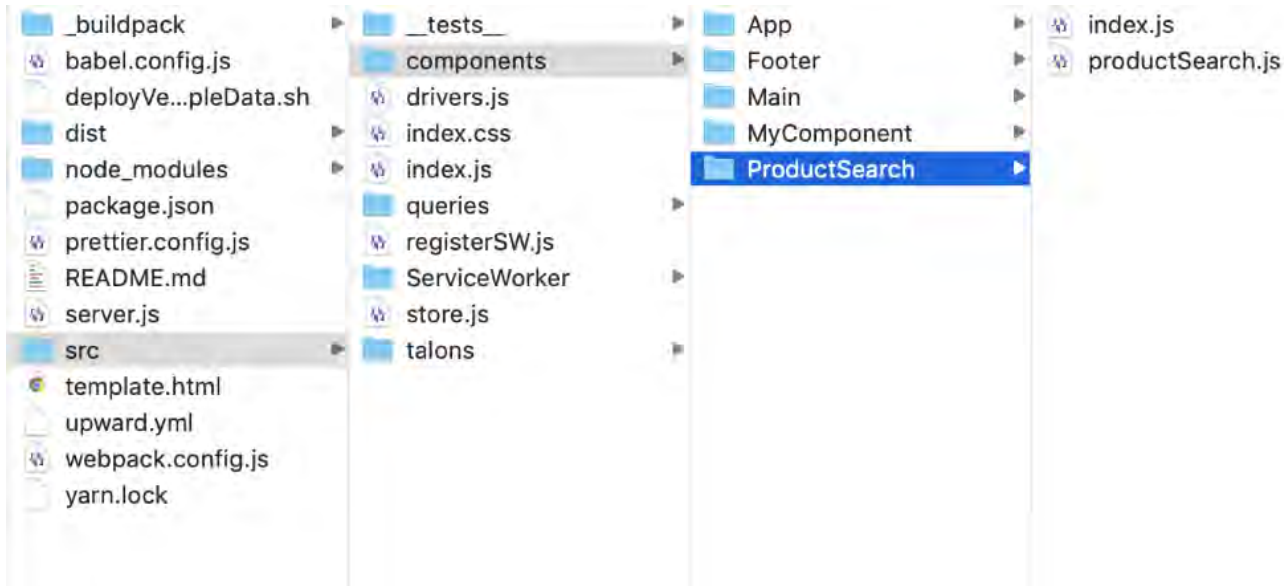
9.2 Using Rest**Exercise 9.2.1**

Create a new component that retrieves data from the Magento 2 Search REST API.

You do not need to display this information anywhere, simply retrieve the data and log it to the console.

You can use this query to retrieve the data:

```
<your_base_store_url>/rest/V1/search?searchCriteria[requestName]=quick_search_container&searchCriteria[filterGroups][0][filters][0][field]=search_term&searchCriteria[filterGroups][0][filters][0][value]=Isadora Skirt
```

Solution**Step 1**

Create a new directory in your src/components directory for your new component. Eg: ProductSearch.

In this directory, create the public API for your component and also a matching component file.
EG: index.js and productSearch.js

Step 2

Within your index.js file add the following to create the public API

```
export { default } from './productSearch';
```

Step 3

In your productSearch.js file, add the code to import the Magento2 REST API class from peregrine.

In this file you should define a request constant as Magento2;

You should then define an endpoint for your REST call, as described in the exercise question. You will then be able to call request() on your endpoint and log the results.

```
import React from 'react';
import { Magento2 } from '@magento/peregrine/lib/RestApi';

const { request } = Magento2;

const ProductSearch = props => {
  const SEARCH_API_URL =
    '/rest/V1/search?searchCriteria[requestName]=quick_search_container&searchCriteria[filterGroups][0][filters][0][field]=search_term&searchCriteria[filterGroups][0][filters][0][value]=Isadora Skirt';

  // request data from server
  const searchResults = request(SEARCH_API_URL)
    .then(res => {
```

```
        console.log(res.items)
      });

      return 'Success';
    }
  }
  export default ProductSearch;
```

Step 4

Include your new component somewhere it will get called, eg: /src/index.js

```
import ProductSearch from './components/ProductSearch';
...
<Adapter apiBase={apiBase} apollo={{ link: apolloLink }} store={store}>
  <AppContextProvider>
    <App />
    <MyComponent />
    <ProductSearch />
  </AppContextProvider>
</Adapter> ,
```

9.3 Using GraphQL

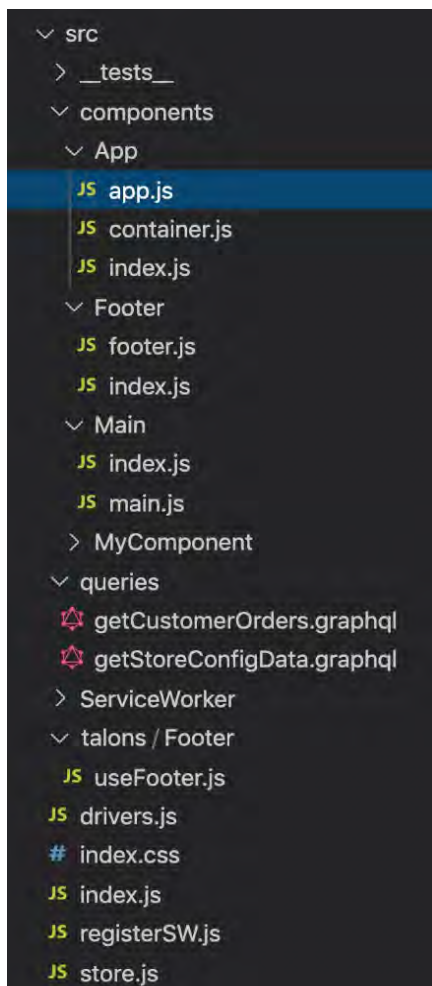
Exercise 9.3.1

Your client wishes to display the currency the PWA Storefront is operating in the footer of the site.

Using GraphQL, add the store `base_currency_code` value to the footer of your PWA Storefront

You should refer to the App, Main and Footer classes in addition to the `getStoreConfig` GraphQL definition to achieve this.

Solution



Step 1

Copy the necessary Venia UI Components and Peregrine footer talon to your local src/components directory

Copy these files:

```
node_modules@magento/venia-ui/lib/components/App/app.js -> /src/components/App/app.js
node_modules@magento/venia-ui/lib/components/App/container.js -> /src/components/App/container.js
node_modules@magento/venia-ui/lib/components/App/index.js -> /src/components/App/index.js
node_modules@magento/venia-ui/lib/components/Footer/footer.js -> /src/components/Footer/footer.js
node_modules@magento/venia-ui/lib/components/Footer/index.js -> /src/components/Footer/index.js
node_modules@magento/venia-ui/lib/components/Main/index.js -> /src/components/Main/index.js
node_modules@magento/venia-ui/lib/components/Main/main.js -> /src/components/Main/main.js
node_modules@magento/venia-ui/lib/queries/getStoreConfigData.graphql ->
/src/queries/getStoreConfigData.graphql
node_modules@magento/peregrine/lib/talons/Footer/index.js -> /src/talons/Footer/useFooter.js
```

Step 2

Update references in these files to local or library references.

In your `src/index.js` file, modify the reference from:

```
import App, { AppContextProvider } from '@magento/venia-ui/lib/components/App';
```

to:

```
import App, { AppContextProvider } from './components/App';
```

In the `src/components/App/index.js` file update the second line to this:

```
export { default as AppContextProvider } from '@magento/venia-ui/lib/components/App/contextProvider';
```

In the `src/components/App/container.js` file, update the 5th line to this:

```
import { useErrorBoundary } from '@magento/venia-ui/lib/components/App/useErrorBoundary';
```

In the `src/components/App/app.js` file update the imports to @magento references EXCEPT for Main

```
import { useApp } from '@magento/peregrine/lib/talons/App/useApp';
import { HeadProvider, Title } from '@magento/venia-ui/lib/components/Head';
import Main from './Main';
import Mask from '@magento/venia-ui/lib/components/Mask';
import MiniCart from '@magento/venia-ui/lib/components/MiniCart';
import Navigation from '@magento/venia-ui/lib/components/Navigation';
import Routes from '@magento/venia-ui/lib/components/Routes';
import { registerMessageHandler } from '@magento/venia-ui/lib/util/swUtils';
import { HTML_UPDATE_AVAILABLE } from '@magento/venia-ui/lib/constants/swMessageTypes';
import ToastContainer from '@magento/venia-ui/lib/components/ToastContainer';
import Icon from '@magento/venia-ui/lib/components/Icon';
```

In `src/components/Main/main.js` update the references to @magento references EXCEPT Footer

```
import { mergeClasses } from '@magento/venia-ui/lib/classify';
import Footer from './Footer';
import Header from '@magento/venia-ui/lib/components/Header';
import defaultClasses from '@magento/venia-ui/lib/components/Main/main.css';
```

In `src/components/Footer/footer.js` update the references to @magento references EXCEPT for your useFooter talon and local GraphQL query

```
import { useFooter } from '../../talons/Footer/useFooter';
import { mergeClasses } from '@magento/venia-ui/lib/classify';
import defaultClasses from '@magento/venia-ui/lib/components/Footer/footer.css';
import GET_STORE_CONFIG_DATA from '../../queries/getStoreConfigData.graphql';
```

Step 3 – Retrieve the value

In your local `getStoreConfigData.graphql` file, add a new Field for `base_currency_code`

```
query storeConfigData {
  storeConfig {
    id
    copyright
    base_currency_code
  }
}
```

In the useFooter talon, modify the response to include the base_currency_code attribute in the response

```
return {
  copyrightText: data && data.storeConfig && data.storeConfig.copyright,
  currencyCode: data
  && data.storeConfig && data.storeConfig.base_currency_code
};
```

Step 4 Display the Value

In your footer.js component set the response from the useFooter talon to set the currency to a local variable. Adjust line 15 to this:

```
const { copyrightText, currencyCode } = talonProps;
```

After line 20 set that currencyCode to a variable like this:

```
let currency = null;
if (currencyCode) {
  currency = <span>{currencyCode}</span>
}
```

Adjust the footer output to include the new value

```
<div className={classes.tile}>
  <h2 className={classes.tileTitle}>
    <span>Help Center</span>
  </h2>
  <p className={classes.tileBody}>
    <span>Get answers from our community online.</span>
  </p>
</div>
<div className={classes.tile}>
  <h2 className={classes.tileTitle}>
    <span>Store Currency</span>
  </h2>
  <p className={classes.tileBody}>
    <span>{currency}</span>
  </p>
</div>
<small className={classes.copyright}>{copyright}</small>
```