EXPERIMENT 7

Aim: To write meta data of your Ecommerce PWA in a Web app manifest file

Theory:

Regular Web App :

- A regular web app is a website that is designed to be accessible on all mobile devices such that the content gets fit as per the device screen.
- It is designed using a web technology stack (HTML, CSS, JavaScript, Ruby, etc.) and operates via a browser.
- They have various native-device features and functionalities. However, it entirely depends on the browser the user is using. In other words, it might be possible that you can access a native-device feature on Chrome but not on Safari or Mozilla Firefox because the browsers are incompatible with that feature.

Progressive Web App:

- Progressive Web App (PWA) is a regular web app, but some extras enable it to deliver an excellent user experience.
- It is a perfect blend of desktop and mobile application experience to give both platforms to the end users.
- Difference between PWAs vs. Regular Web Apps: A Progressive Web is different and better than a Regular Web app with features like:

  1. Native Experience Though a PWA runs on web technologies (HTML, CSS, JavaScript) like a Regular web app, it gives user experience like a native mobile application. It can use most native device features, including push notifications, without relying on the browser or any other entity. It offers a seamless and integrated user experience that it is quite tough for one to differentiate between a PWA and a Native application by considering its look and feel.

  2. Ease of Access Unlike other mobile apps, PWAs do not demand longer download time and make memory space available for installing the applications. The PWAs can be shared and installed by a link, which cuts down the number of steps to install and use. These applications can easily keep an app icon on the user's home screen, making the app easily accessible to the users and helps the brands remain in the users' minds, and improving the chances of interaction.

3. Faster Services PWAs can cache the data and serve the user with text stylesheets, images, and other web content even before the page loads completely. This lowers the waiting time for the end-users and helps the brands improve the user engagement and retention rate, which eventually adds value to their business.

4. Engaging Approach As already shared, the PWAs can employ push notifications and other native device features more efficiently. Their interaction does not depend on the browser user uses. This eventually improves the chances of notifying the user regarding your services, and other options related to your brand and keeping them hooked to your brand. In simpler words, PWAs let you maintain the user engagement and retention rate.

5. Updated Real-Time Data Access Another plus point of PWAs is that these apps get updated on their own. They do not demand the end users to go to the App Store or other such platforms to download the update and wait until installed. In this app type, the web app developers can push the live update from the server, which reaches the apps residing on the user's devices automatically. Therefore, it is easier for the mobile app developer to provide the best of the updated functionalities and services to the end-users without forcing them to update their app.

6. Discoverable PWAs reside in web browsers. This implies higher chances of optimizing them as per the Search Engine Optimization (SEO) criteria and improving the Google rankings like that in websites and other web apps.

7. Lower Development Cost Progressive web apps can be installed on the user device like a native device, but it does not demand submission on an App Store. This makes it far more cost-efficient than native mobile applications while offering the same set of functionalities.

- The main features are:

    1. Progressive — They work for every user, regardless of the browser chosen because they are built at the base with progressive improvement principles. Responsive — They adapt to the various screen sizes: desktop, mobile, tablet, or dimensions that can later become available.
    2. App-like — They behave with the user as if they were native apps, in terms of interaction and navigation.

3. Updated — Information is always up-to-date thanks to the data update process by service workers. Secure — Exposed over HTTPS protocol to prevent the connection from displaying information or altering the contents.
4. Searchable — They are identified as “applications” and are indexed by search engines.
5. Reactivable — Make it easy to reactivate the application thanks to capabilities such as web notifications.
6. Installable — They allow the user to “save” the apps that he considers most useful with the corresponding icon on the screen of his mobile terminal (home screen) without having to face all the steps and problems related to the use of the app store.
7. Linkable — Easily shared via URL without complex installations.

Code:

.Step 1: Create an HTML page that would be the starting point of the application. This HTML will contain a link to the file named manifest.json. This is an important file that would be created in the next step

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Flower Shop</title>
    <link rel="stylesheet" href="styles.css" />
    <link rel="manifest" href="manifest.json" />
  </head>

  <body>
    <nav class="navbar">
      <div class="container">
        <a href="#" class="logo">Flower Shop</a>
        <ul class="nav-links">
          <li><a href="#">Home</a></li>
          <li><a href="#">About</a></li>
          <li><a href="#">Support</a></li>
          <li><a href="#">Social Media</a></li>
          <li><a href="#">Products</a></li>
```

```
        <li><a href="#">View Cart</a></li>
      </ul>
    </div>
  </nav>

  <div class="container">
    <h1>Our Products</h1>
    <div class="products">
      <!-- Product 1 -->
      <div class="product">
        <img src="assets/rose.png" alt="Flower 1" />

        <h3>Rose Bouquet</h3>
        <p class="price">RS. 200</p>
        <button>Add to Cart</button>
      </div>

      <div class="product">
        <img src="assets/sunflower.png" alt="Flower 1" />

        <h3>Sunflower Bouquet</h3>
        <p class="price">RS. 200</p>
        <button>Add to Cart</button>
      </div>

      <!-- Product 2 -->
      <div class="product">
        <img src="assets/tulip.png" alt="Flower 3" />

        <h3>Tulip Arrangement</h3>
        <p class="price">RS. 200</p>
        <button>Add to Cart</button>
      </div>
      <!-- Add more products as needed -->
    </div>
  </div>

  <footer class="bottom-navbar">
    <div class="container">
      <ul class="bottom-nav-links">
```

```
          <li><a href="#">Terms of Service</a></li>
          <li><a href="#">Privacy Policy</a></li>
          <li><a href="#">Contact Us</a></li>
        </ul>
      </div>
    </footer>
  </body>
</html>
```

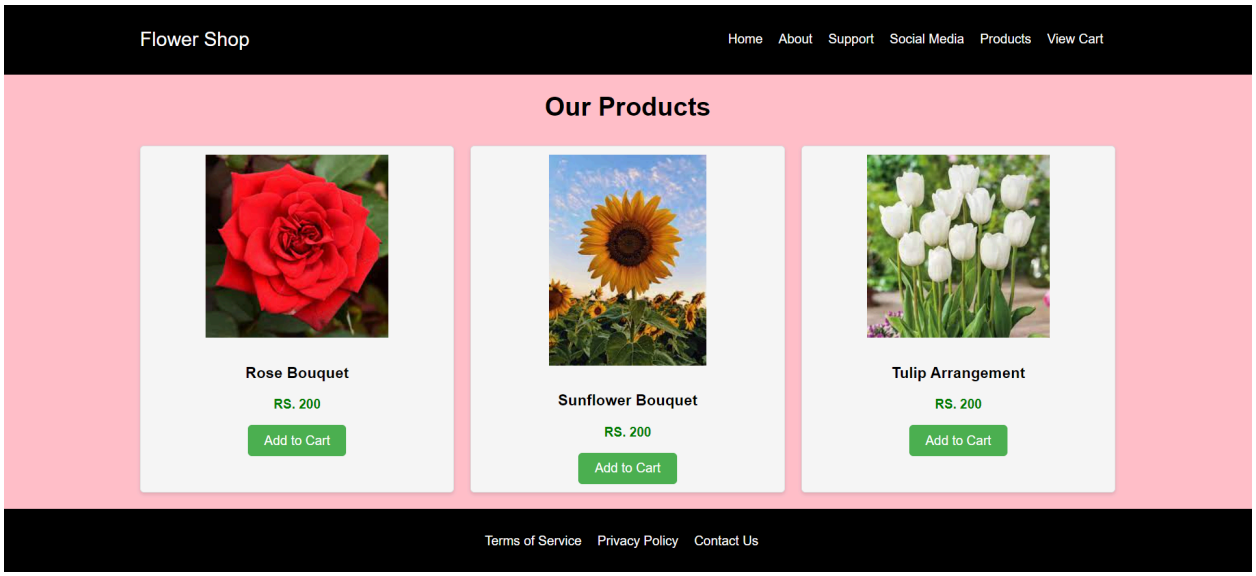
Manifest.json file :

Step 2: Create a manifest.json file in the same directory. This file basically contains information about the web application. Some basic information includes the application name, starting URL, theme color, and icons. All the information required is specified in the JSON format. The source and size of the icons are also defined in this file.

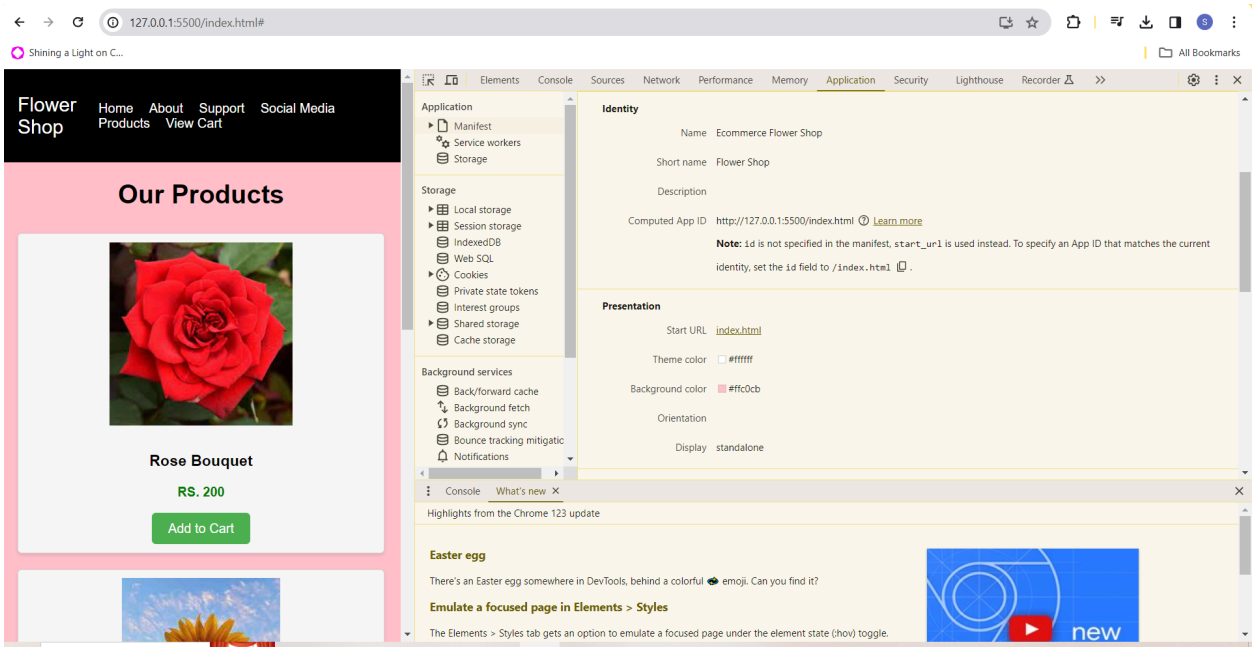
```
{
  "name": "Ecommerce Flower Shop",
  "short_name": "Flower Shop",
  "start_url": "index.html",
  "display": "standalone",
  "background_color": "#ffc0cb",
  "theme_color": "#ffffff",
  "icons": [
    {
      "src": "assets/rose.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "assets/sunflower.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "assets/tulip.png",
      "sizes": "512x512",
      "type": "image/png"
    }
```

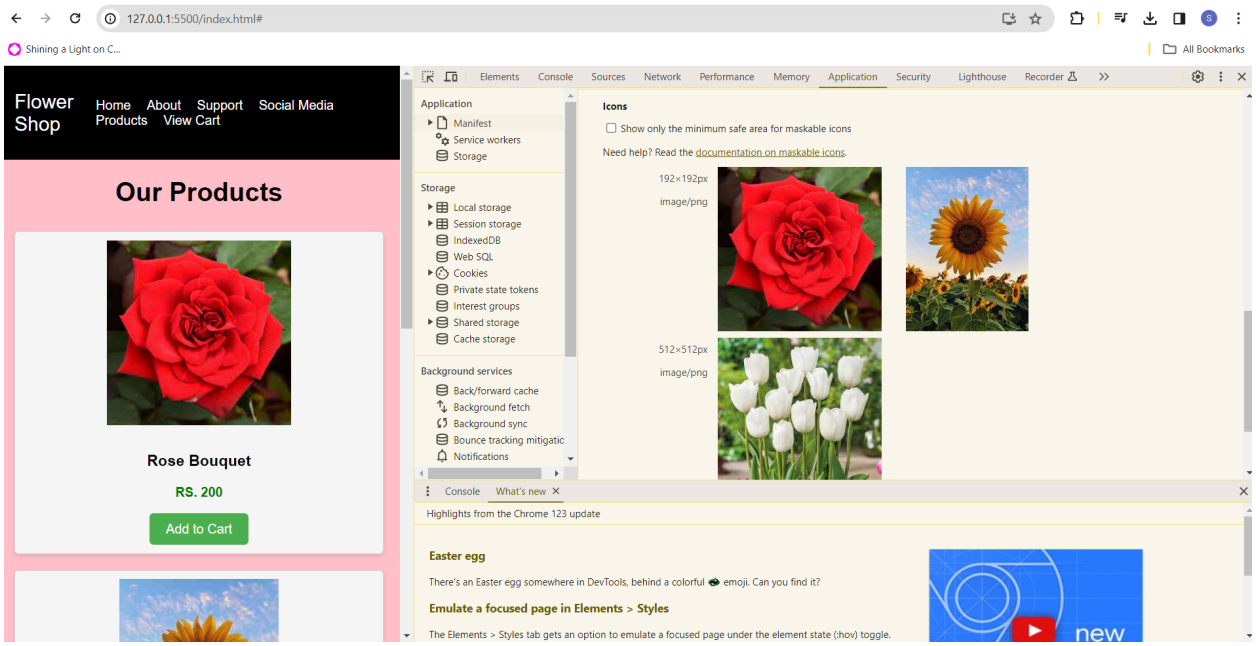
```
  ]
}
```

Step 3: Serve the directory using a live server so that all files are accessible



Step 4: Open the index.html file in Chrome navigate to the Application Section in the Chrome Developer Tools. Open the manifest column from the list





Step 6: Under the installability tab, it would show that no service worker is detected. We will need to create another file for the PWA, that is, serviceworker.js in the same directory. This file handles the configuration of a service worker that will manage the working of the application

```
// Service Worker Installation
self.addEventListener('install', event => {
    console.log('Service Worker installed');
});

// Service Worker Activation
self.addEventListener('activate', event => {
    console.log('Service Worker activated');
});

// Fetch Event - Cache and Network Strategy
self.addEventListener('fetch', event => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                // Cache hit - return response
                if (response) {
                    return response;
                }
```

```
                // Clone the request
                let fetchRequest = event.request.clone();

                // Make a network request
                return fetch(fetchRequest)
                    .then(networkResponse => {
                        // Check if we received a valid response
                        if (!networkResponse || networkResponse.status !==
200 || networkResponse.type !== 'basic') {
                            return networkResponse;
                        }

                        // Clone the network response
                        let responseToCache = networkResponse.clone();

                        // Cache the response
                        caches.open('your-cache-name')
                            .then(cache => {
                                cache.put(event.request, responseToCache);
                            });

                        return networkResponse;
                    });
            })
    );
});
```

Step 6 : The last step is to link the service worker file to index.html. This is done by adding a short JavaScript script to the index.html created in the above steps. Add the below code inside the script tag in index.html.

```
<script>
     if ("serviceWorker" in navigator) {
       window.addEventListener("load", () => {
         navigator.serviceWorker
           .register("/serviceworker.js")
           .then((registration) => {
             console.log(
               "Service Worker registered with scope:",
```

```
            registration.scope
          );
        })
        .catch((error) => {
          console.log("Service Worker registration failed:", error);
        });
    });
  }
</script>
```

Installing the application: Navigating to the Service Worker tab, we see that the service worker is registered successfully and now an install option will be displayed that will allow us to install our app. Click on the install button to install the application. The application would then be installed, and it would be visible on the desktop. For installing the application on a mobile device, the Add to Home screen option in the mobile browser can be used. This will install the application on the device.

Conclusion: Thus writing metadata for the PWA, especially for an eCommerce application, is crucial for enabling features like the "add to homescreen" functionality. By crafting a well-structured manifest.json file with accurate metadata properties such as name, description, icons, and colors, developers can enhance the accessibility and user experience of their PW