**Microfrontends**

Table of Contents

[Microfrontends 3](#_Toc175429987)

[Benefits of Microfrontends 3](#_Toc175429988)

[Monolithic to Microfrontends 4](#_Toc175429989)

[Container 4](#_Toc175429990)

[Integration 5](#_Toc175429991)

[Major Categories of Integration 5](#_Toc175429992)

[Build Time (Compile Time Integration) 6](#_Toc175429993)

[Run Time (Client-Side Integration) 7](#_Toc175429994)

[Project Setup 8](#_Toc175429995)

[Webpack 9](#_Toc175429996)

[Integration Process 11](#_Toc175429997)

[Module Federation Plugin 12](#_Toc175429998)

[Steps to implement Module Federation 12](#_Toc175429999)

[container/webpack.config.js 14](#_Toc175430000)

[container/bootstrap.js 14](#_Toc175430001)

[products/webpack.config.js 14](#_Toc175430002)

[Overall MFE look 15](#_Toc175430003)

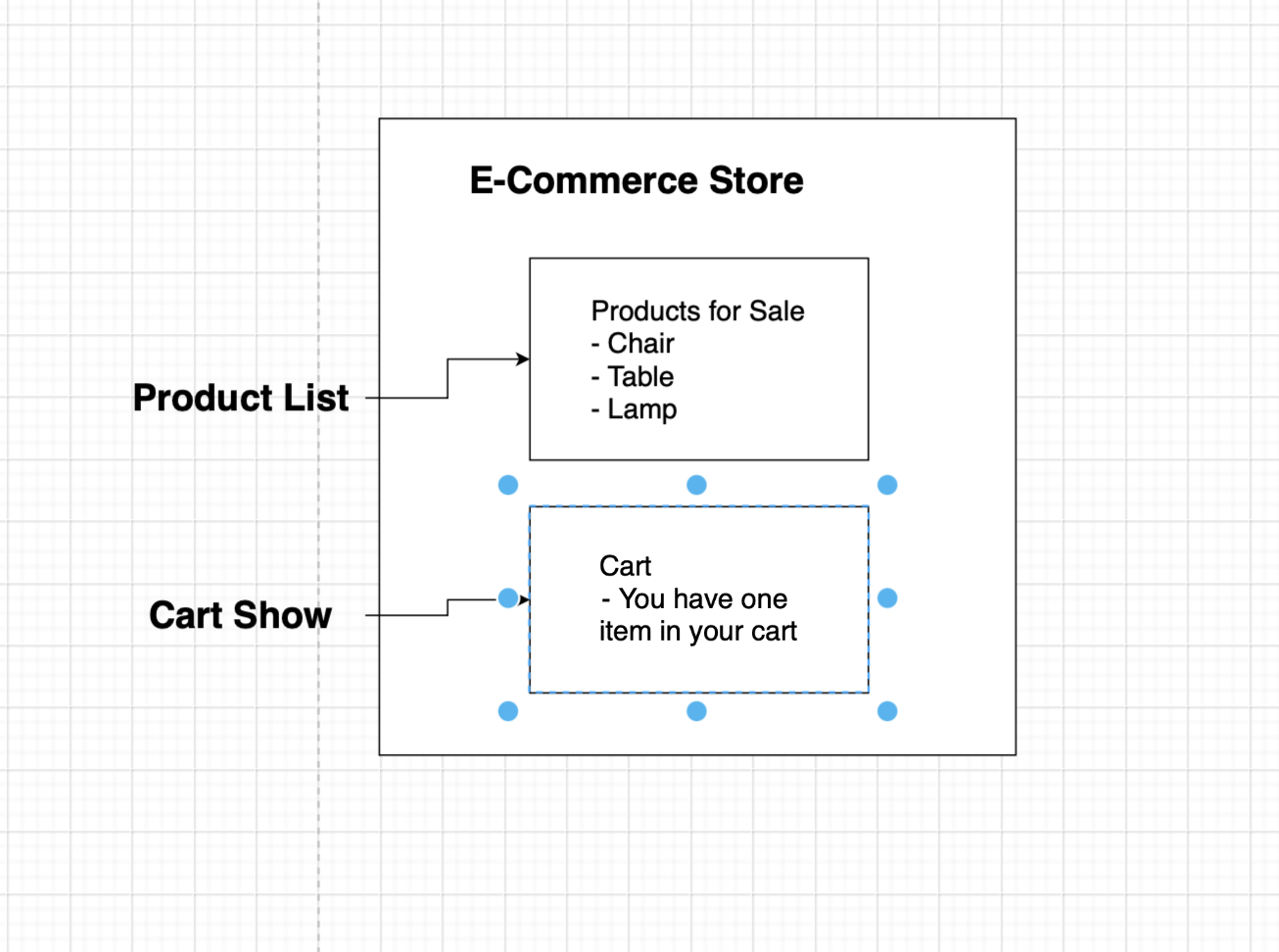
[Shared Modules 15](#_Toc175430004)

# Microfrontends

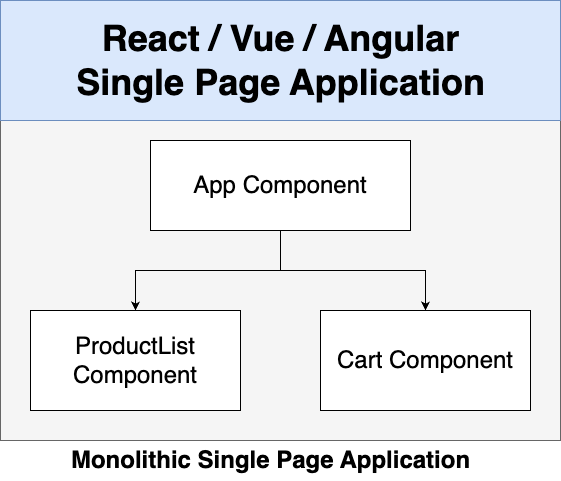
* Where we take a monolithic application, and we divide it into smaller applications.
* Each of these smaller applications are responsible for one distinct major feature of our product.
* As much as possible, we try to prevent these different micro applications from communicating with each other directly.

## Benefits of Microfrontends

1. Separate independent apps
2. no direct communication between them
3. no direct dependency
4. It allows multiple different engineering teams to work on the same overall applications but in total isolation.
5. Each smaller app is easier to understand and make changes to.



## Monolithic to Microfrontends



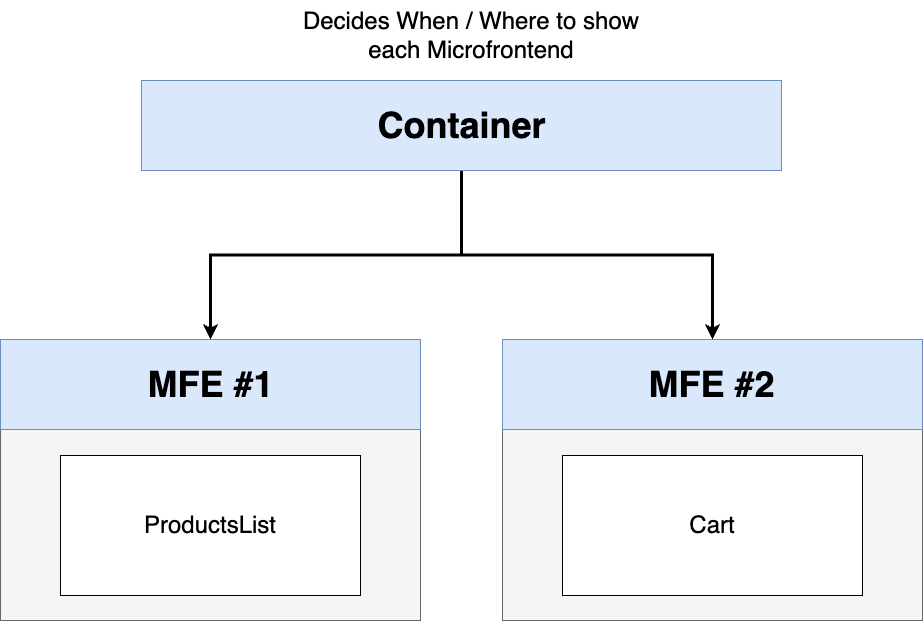
A screenshot of a computer screen

Description automatically generated

## Container

The challenge is to show how to show two MFE in a screen and we create a third micro frontend app that usually refers to as the **container**.

The container decides when and where to show all the different micro frontends that we have.



## Integration

* Assembling or orchestrating our different micro frontends, or different ways of making sure the container gets access to products list and cart.
* There is no single perfect solution to integration
* Many solutions, each have pros and cons.
* Decide on what to use based on your requirements of your application.

### Major Categories of Integration

1. **Build Time (Compile Time Integration)**

**Before** Container gets loaded in the browser, it gets access to ProductsList source code.

1. **Run Time (Client-Side Integration)**

**After** container gets loaded in the browser, it gets access to ProductsList source code.

1. **Server**

While sending down JS to load up Container, a server decides on whether to include ProductsList source code.

### Build Time (Compile Time Integration)

A diagram of a product

Description automatically generated

Pro –

1. Easy to setup and understand

Cons –

1. Container must be re-deployed every time ProductsList is updated.
2. Tempting to tightly couple the Container + ProductsList together.

### Run Time (Client-Side Integration)

A screenshot of a application

Description automatically generated

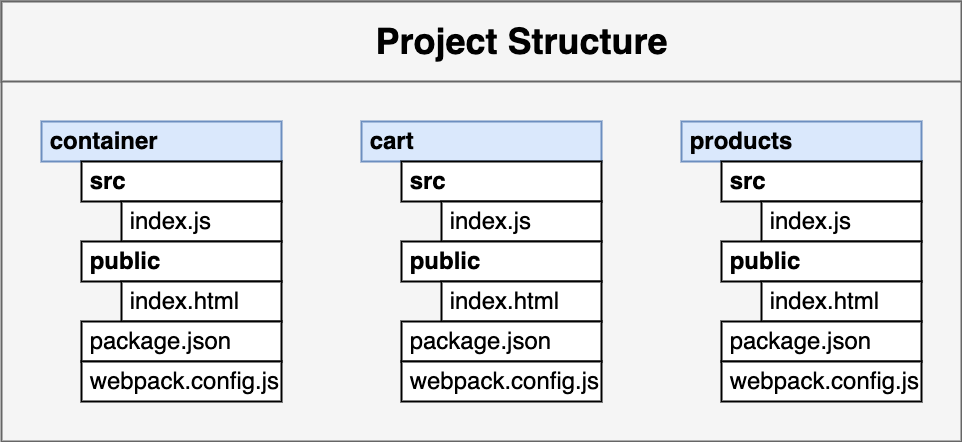
Pros

1. ProductsList can be deployed independently at any time
2. Different versions of ProductsList can be deployed and Container can decide which one to use. (Particularly helps in the A/B Testing)
3. Most flexible and performant solution around right now.

Cons –

1. Tooling + setup is far more complicated.

## Project Setup



1. Creating the Products MFE
   1. No Framework for now, with plain vanilla js
   2. Install the exact same versions of webpack
   3. npm install webpack@5.88.0 webpack-cli@4.10.0 webpack-dev-server@4.7.4 faker@5.1.0 html-webpack-plugin@5.5.0 --save-exact

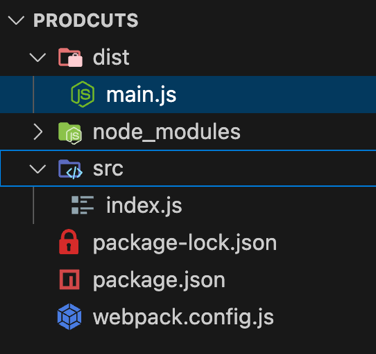
## Webpack

* We throw in multiple code files with their own dependencies to generate a single bundle or main file.

A diagram of a webpage

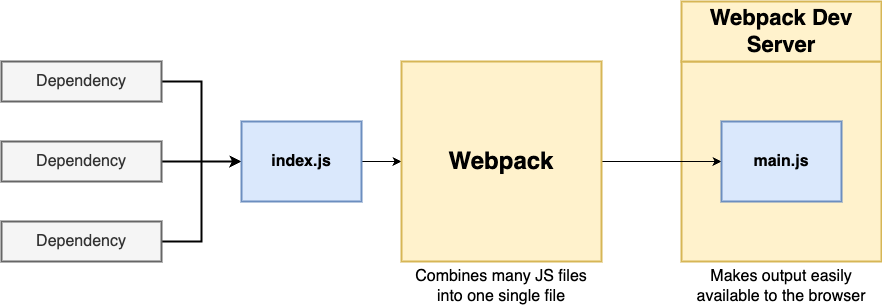
Description automatically generated

* After running webpack command a dist folder is generated with main.js or bundle.js file as the output.



Now we need to run the code in the browser, and we need to setup Webpack Dev server.

* It takes the output from Webpack process and makes it available in the browser.



* Update the port in the webpack.config.js file

devServer: {

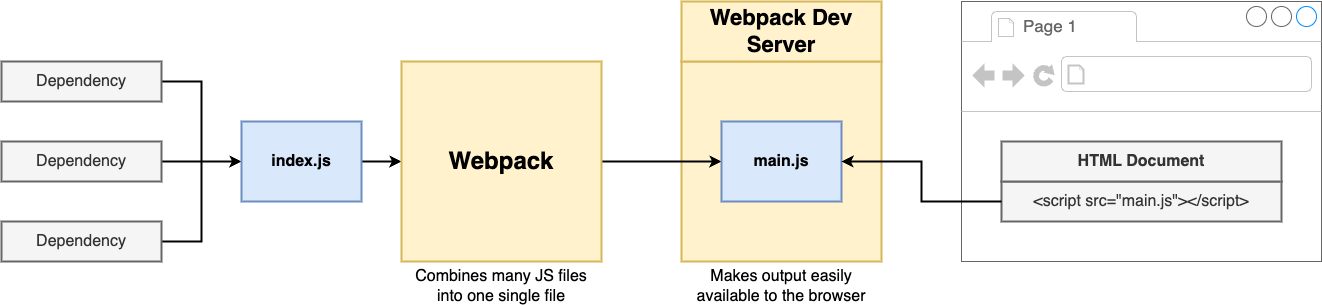
port: 8081,

},

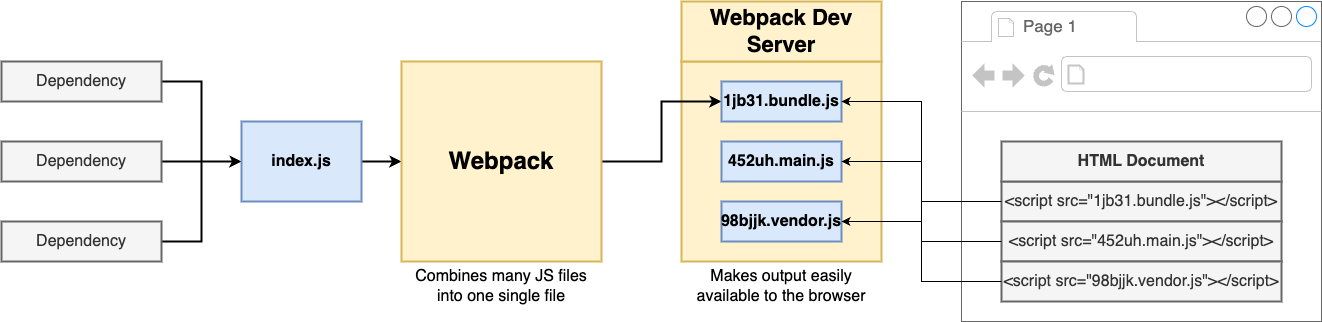
* update the script in package.json to

“start”: “webpack serve”

* To get the content as an html add index.html in the public folder and include the script



Not Always the main.js file will be ideally generated, to avoid file caching at client browser we opt for random name generated files and to guess the file name in index.html we use the plugin HTML webpack plugin.



Code to update webpack.config.js file as

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

mode: "development",

devServer: {

port: 8081,

},

plugins: [

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

],

};

Update the public/index.html as below

<!DOCTYPE html>

<html>

<head></head>

<body>

<div id="dev-products"></div>

</body>

</html>

And src/index.js as

import faker from "faker";

let products = "";

for (let i = 0; i < 5; i++) {

const name = faker.commerce.productName();

products += `<div>${name}</div>`;

}

document.querySelector("#dev-products").innerHTML = products;

Now Setting up the Container MFE

* Create container folder inside ecomm and create the package.json file and install the dependencies inside container folder as
* npm install webpack@5.88.0 webpack-cli@4.10.0 webpack-dev-server@4.7.4 html-webpack-plugin@5.5.0 nodemon --save-exact

Integration Process

* Setting Products into Container
* define Module Federation plugin in webpack.config.js of both Container and Products like below and restart both servers

Container

const HtmlWebpackPlugin = require("html-webpack-plugin");

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

module.exports = {

mode: "development",

devServer: {

port: 8080,

},

plugins: [

new ModuleFederationPlugin({

name: "container",

remotes: {

products: "products@http://localhost:8081/remoteEntry.js",

},

}),

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

],

};

Products

const HtmlWebpackPlugin = require("html-webpack-plugin");

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

module.exports = {

mode: "development",

devServer: {

port: 8081,

},

plugins: [

new ModuleFederationPlugin({

name: "products",

filename: "remoteEntry.js",

exposes: {

"./ProductsIndex": "./src/index",

},

}),

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

],

};

## Module Federation Plugin

### Steps to implement Module Federation

A screenshot of a remote control

Description automatically generated

* In Module Federation, **Hosts** requires code from **Remote.**

Diagram view of the way Module Federation Works inside Products MFE (Remote) .

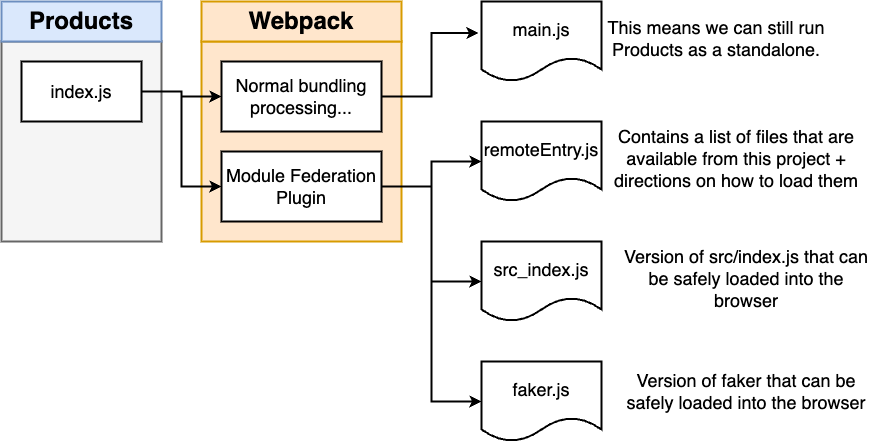


Diagram view of the way Module Federation Works inside Container MFE (Host) .

A diagram of a webpack

Description automatically generated

* The reason to have **import("./bootstrap");** & adding the **index.js** file, we are just giving Webpack an opportunity inside the browser to go and fetch some dependencies from products before actually executing the code of bootstrap.
* From the diagram **index.js** generating **main.js** and then our bootstrap is being output as a separate file.
* We first run **main.js** that’s going to try to fetch some additional code from products and then execute the code inside of **bootstrap.js.’**

Q1. In the container project, why does the **index.js** file have an import statement in. it for **bootstrap.js**? Why didn’t we just run **bootstrap.js** directly?

Ans – The import statement gives webpack the opportunity to load up code from **Products** before running **bootstrap.js**

Q2. What is the **remoteEntry.js** file in the Products project for?

Ans – It lists different modules that can be loaded from the Products project.

Execution Sequence so far -

A screenshot of a computer

Description automatically generated

### container/webpack.config.js

A computer screen with text and images

Description automatically generated with medium confidence

### container/bootstrap.js



* In bootstrap it sees this command and tries to find the source first in node\_modules, if it doesn’t finds it there then it goes in to the webpack.config.js file to look for an Entry with products, which matches to key inside ModuleFederationPlugin code.

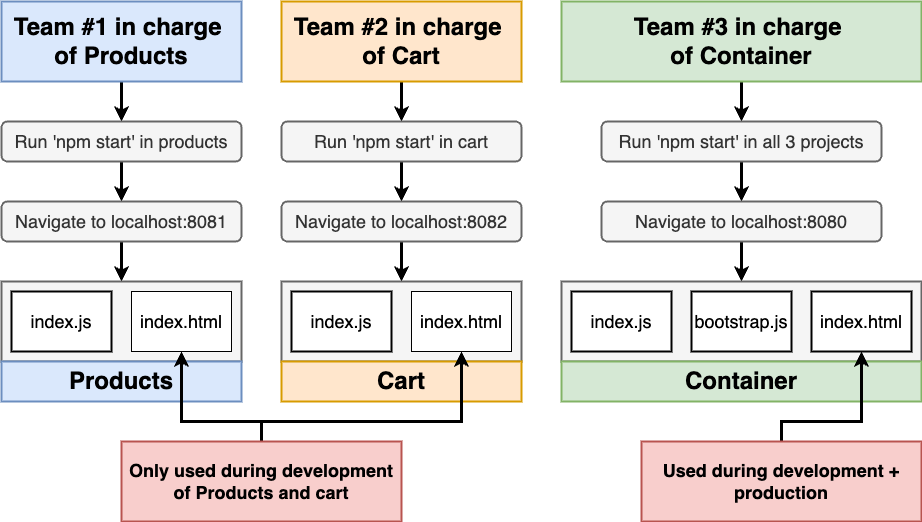
### products/webpack.config.js

A computer screen shot of a program

Description automatically generated

* The name: “**products**” should match to the value [**products**@http://localhost:8081/remoteEntry.js](mailto:products@http://localhost:8081/remoteEntry.js) in the container, else it will not be able to fetch the code.
* ./**ProductsIndex** is an alias name but used in the import statement in the container to import the file exposed.

### Overall MFE look



### Shared Modules

A group of black text on a white background

Description automatically generated

* Module Federation plugin makes all these functionality.
* shared key is used in products and cart webpack, along with the shared module.

shared: ["faker"],

* Unfortunately we broke other applications when we run them in isolation.

Error: Shared module is not available for eager consumption: webpack/sharing/consume/default/faker/faker

* When we load up products/cart in isolation the first file that really gets executed is our source index.js file and which tries to access the module.
* The shared module loads up asynchronously by default and we don’t have the module available, but not an issue when we load products/cart via container.
* As in container we first load up the remoteEntry file for products/carts and remoteEntry file has some codes and configuration gets us access to respective index.js and to run we need the file Faker.
* To solve this we make the changes as

import("./bootstrap");

* It’s going to load up bootstrap file asynchronously, by doing this we give webpack the opportunity to take a look at what files requires the module to run successfully.
* Here the import statement is a function rather a standalone import.
* If the versions in different apps differ based on ^ or ~ then Module Federation takes the responsibility of loading one file same module or multiple files of same module.

Ex – products use ^5.1.0 and cart use ^4.10.0, then two different files are loaded.

If products use ^5.1.0 and cart use ^5.0.0, then one file is loaded.

### Singleton

* Singleton true means that we only want to load up one single copy of the given module no matter what.
* In Some cases we don’t want to load multiple versions of a given module,

ex- different react versions in the same project for different app throws an error message “you’ve got multiple copies of React running”

* to avoid these we update the shared key as

shared: {

faker: {

singleton: true,

},

},

* but we get a warning message says that we are trying to get version 5.1.0 or we have 5.1.0 version but we want 4.1.0.
* So whenever you get these warnings debug to increment or decrement accordingly and making two sub-projects compatible.

### Sub-App Execution Context

* The **products** team doesn’t have control over the index.html of **container** team and vice versa.
* We need to Refactor our bootstrap.js and below are the ways
* // **Context/Situation #1**
  + // We are running this file in development in isolation.
  + // We are using our local index.html file.
  + // Which DEFENITELY has an element with an id of ‘dev-products’.
  + // We want to immediately render our app into that an element.
* // **Context/Situation #2**
  + // We are running this file in development or production through the CONTAINER App.
  + // No GURANTEE that an element with an id of ‘dev-products’ exists
  + // We DO NOT WANT to try to immediately render the app.