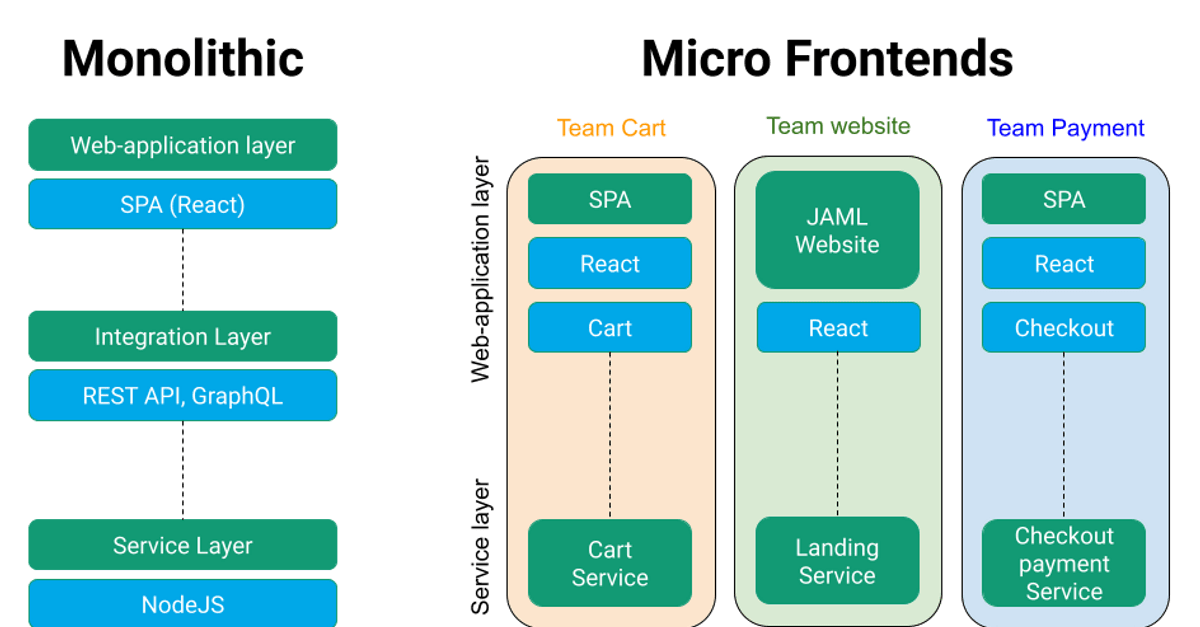
**Introduction to Micro Frontends**

Micro Frontends is a new architectural pattern for building complex, scalable and maintainable web applications. It is an extension of the Microservices architecture pattern, which is widely used in building backend systems.



In Micro Frontends, the user interface (UI) is divided into smaller, more manageable and independent pieces. Each piece of the UI is developed, tested and deployed independently by separate teams, using their own preferred technology stack and programming language.

This allows teams to work on their own pieces of the UI, without having to worry about how other parts of the UI are implemented. It also enables teams to iterate quickly, and deploy changes to their parts of the UI independently, without affecting other parts of the UI.

In this documentation, we will cover the key concepts and best practices of Micro Frontends, and provide guidance on how to implement it in your own web applications.

**Advantages of Micro Frontends:**

Micro frontends have many use cases and advantages that make them an attractive architecture choice for modern web development. Some of the most notable benefits include:

Modular Development: Micro frontends allow for teams to work independently on their own modules, which can be developed, tested, and deployed without impacting other modules. This approach allows for greater flexibility and faster development cycles.

**Scalability**: Micro frontends enable applications to scale horizontally by adding or removing modules as needed. This allows for more granular control over resources and better load balancing.

**Technology Diversity**: With micro frontends, each module can be built with a different technology stack, allowing teams to use the best tools for their specific requirements. This also allows for greater flexibility in updating or replacing modules without affecting the rest of the application.

Improved UX: Micro frontends can improve user experience by allowing for faster page loads, more responsive interfaces, and better handling of errors and edge cases.

**Collaboration**: With micro frontends, teams can collaborate more easily and effectively, as they are able to work independently on their own modules and integrate them seamlessly with the rest of the application.

Overall, micro frontends provide a modern and flexible approach to web development that allows teams to work independently, scale more efficiently, and improve user experience.

**Best Tools to Build Micro Frontends**

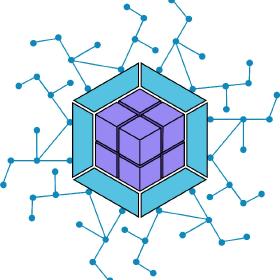
Micro frontends can be built using various tools and frameworks, each with their own set of advantages and disadvantages. Here are some of the most popular tools for building micro frontends:

[**Single SPA**](https://single-spa.js.org/) :

[](https://single-spa.js.org/)

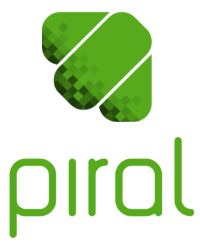
Single SPA is a JavaScript framework for building micro frontends. It allows developers to build multiple independent micro frontends that can be composed into a single page application. Single SPA provides a simple API for loading and unloading micro frontends on the fly, making it easy to add or remove functionality as needed.

[**Module Federation**](https://webpack.js.org/concepts/module-federation/):

[](https://webpack.js.org/concepts/module-federation/)

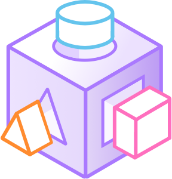
Module Federation is a new feature introduced in Webpack 5 that allows developers to share code between micro frontends at runtime. It enables developers to load code from other micro frontends as if it was part of their own application. Module Federation also enables developers to dynamically load code, which can lead to faster load times and improved performance.

[**Piral**](https://piral.io/):



Piral is a framework for building micro frontends based on the concept of "pilets." Pilets are small, composable pieces of functionality that can be combined together to form a larger application. Piral provides a number of tools for managing pilets, including a pilet API for defining and exporting pilets, a pilet registry for discovering and loading pilets, and a pilet loader for loading pilets at runtime.

[**OpenComponents**](https://opencomponents.github.io/):

[](https://opencomponents.github.io/)

OpenComponents is a framework for building micro frontends based on the concept of "components." Components are small, reusable pieces of functionality that can be composed together to form a larger application. OpenComponents provides a number of tools for managing components, including a component registry for discovering and loading components, and a component API for defining and exporting components.

[**Mosaic**](https://docs.mosaic.js.org/):

[](https://docs.mosaic.js.org/)

Mosaic is a framework for building micro frontends that is based on the concept of "mosaics." Mosaics are small, independent pieces of functionality that can be composed together to form a larger application. Mosaic provides a number of tools for managing mosaics, including a mosaic API for defining and exporting mosaics, a mosaic registry for discovering and loading mosaics, and a mosaic loader for loading mosaics at runtime.

These are just a few of the many tools available for building micro frontends. The choice of tool depends on the specific needs of the project and the preferences of the development team. Regardless of the tool chosen, micro frontends offer a flexible and scalable approach to building complex web applications.

**Proof of Concept**

For this project, we decided to build a basic e-commerce site using Micro Frontends architecture with **Module Federation**. This project was built to demonstrate the capabilities of Module Federation and how it can be used to create scalable, modular, and flexible front-end architectures.

**Why we chose Module Federation:**

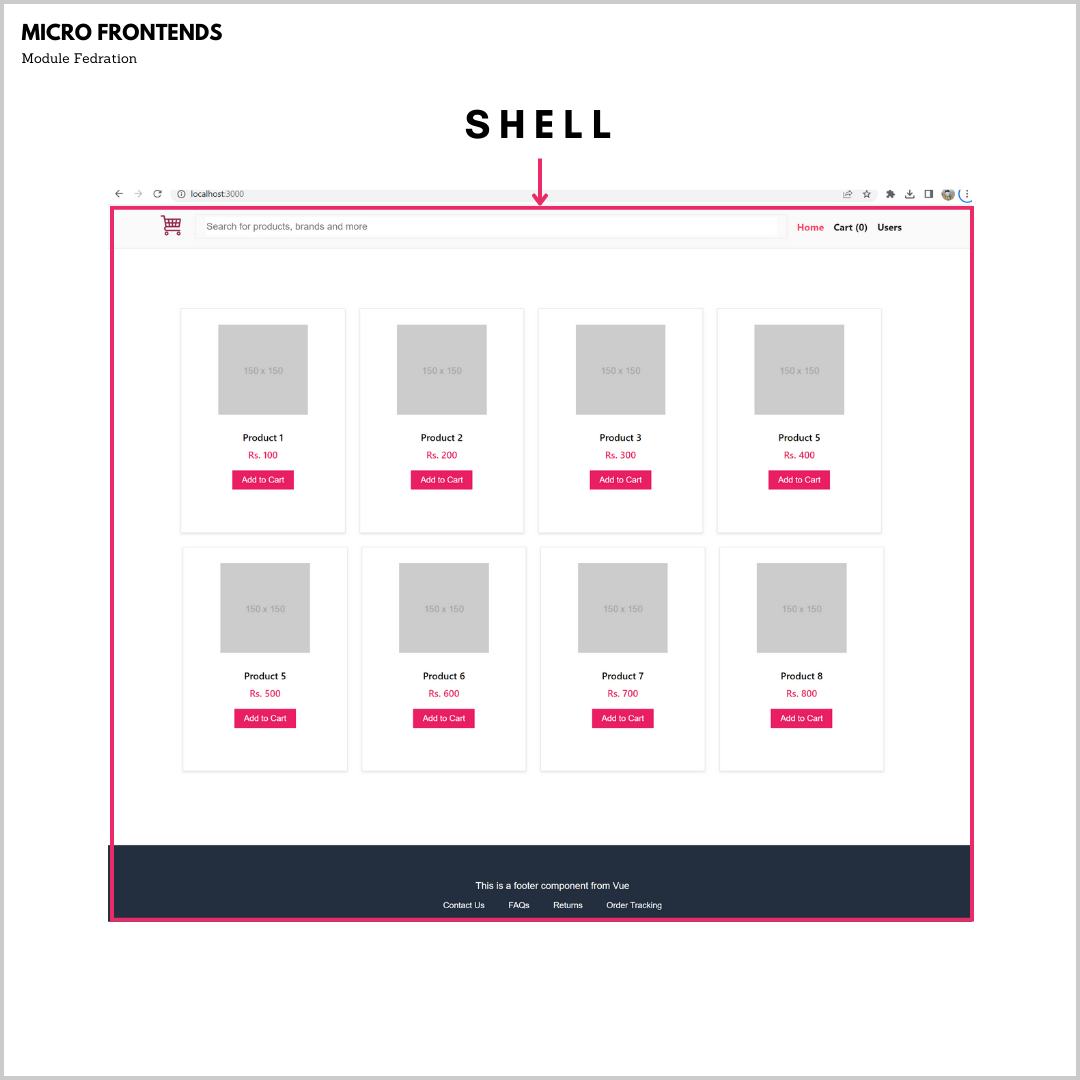
We chose Module Federation because it provides a simple and effective way to share code across different micro frontends. With Module Federation, we can dynamically load remote code from different micro frontends into our main application, allowing us to break down our application into smaller, more manageable parts.

Module Federation also allows us to update our micro frontends independently without affecting other parts of our application. This makes it easier to maintain and update our application in the long run.

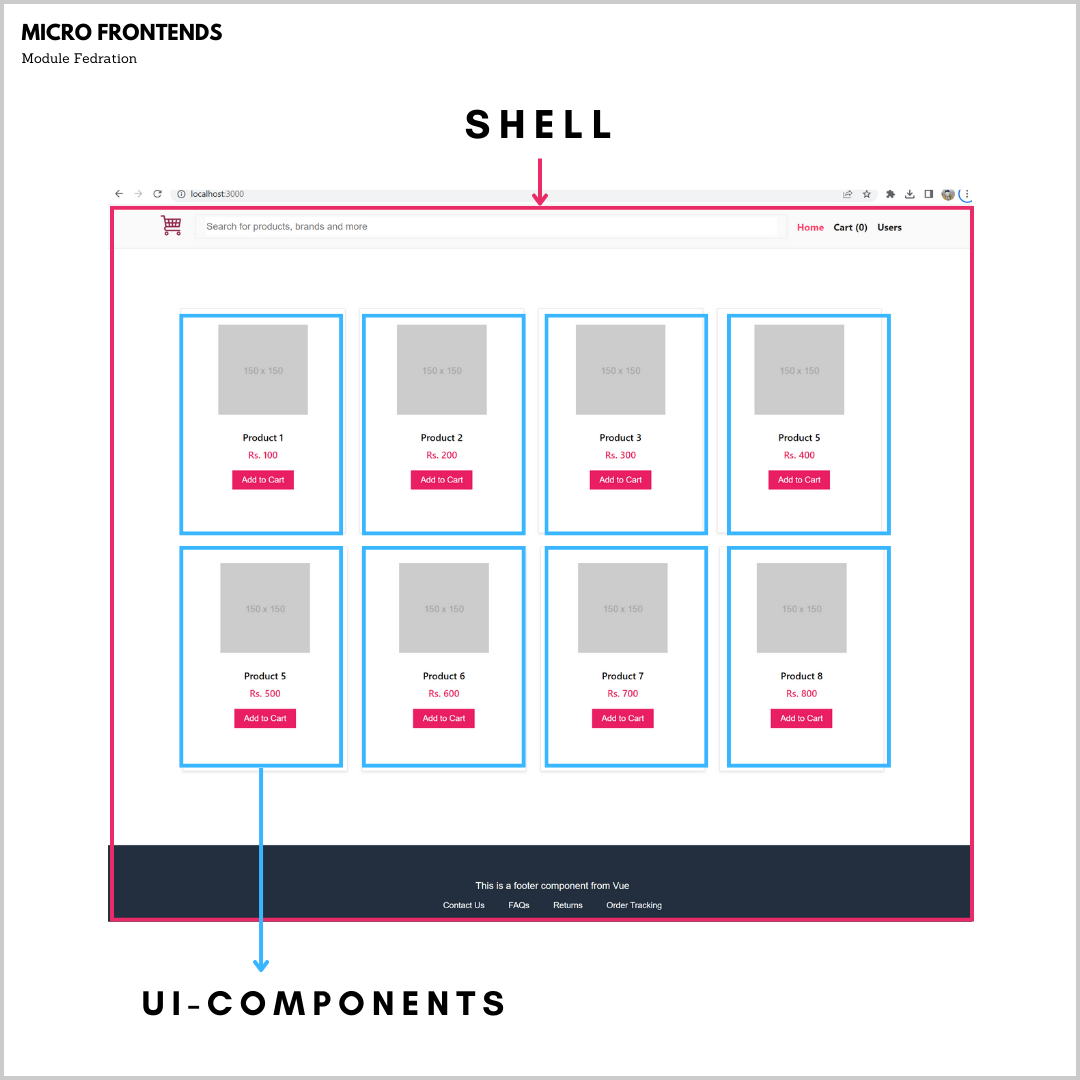
In the next section, we will explain how we built our e-commerce site using Module Federation.

Our e-commerce site is a proof-of-concept project that showcases the power and flexibility of Module Federation in building micro frontends. We have used four micro frontends to build our site, each with a specific purpose and functionality.

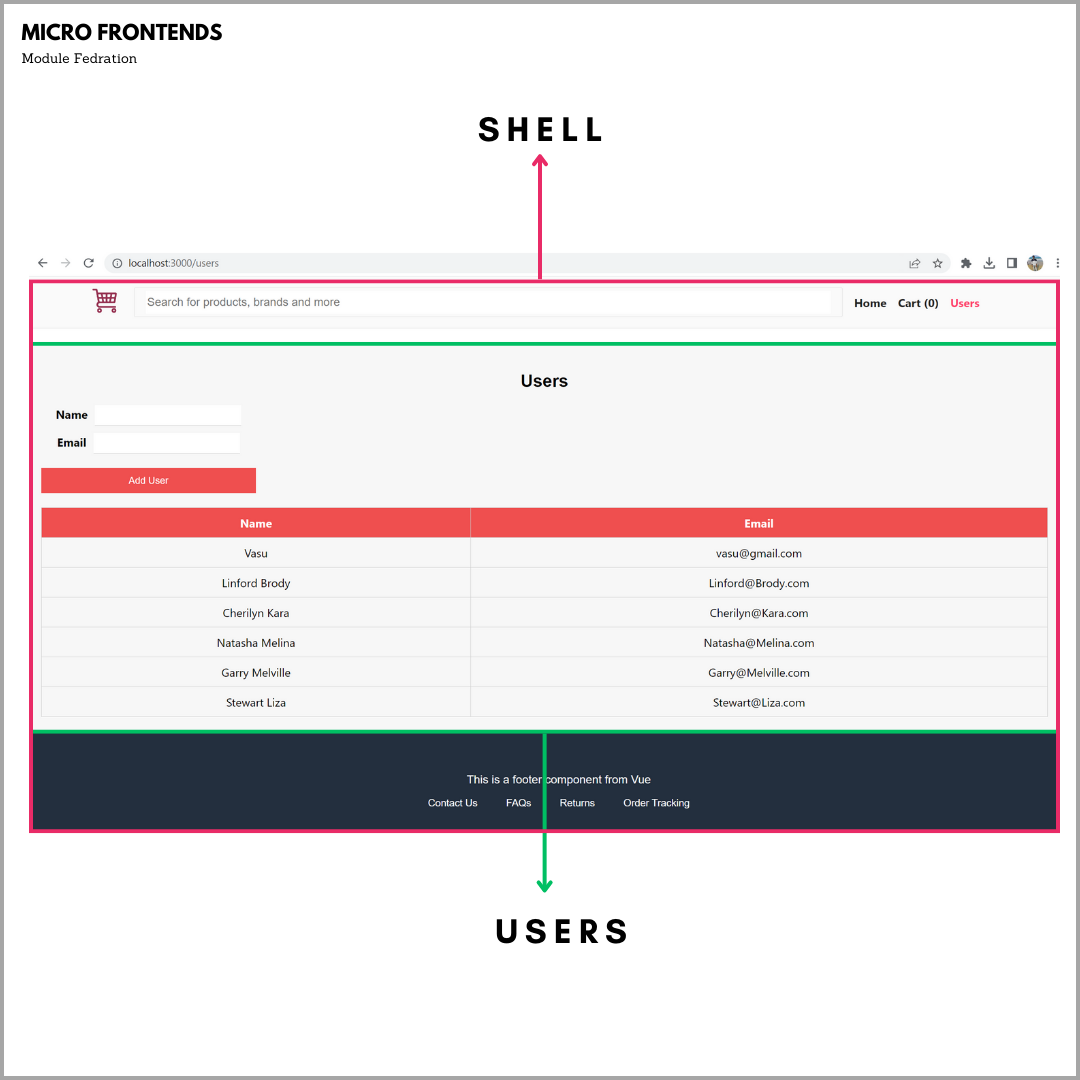
**Shell Root Container**: This is the main container that holds all the micro frontends and provides a unified look and feel to the site. It is built with React and contains the header and navigation menu.



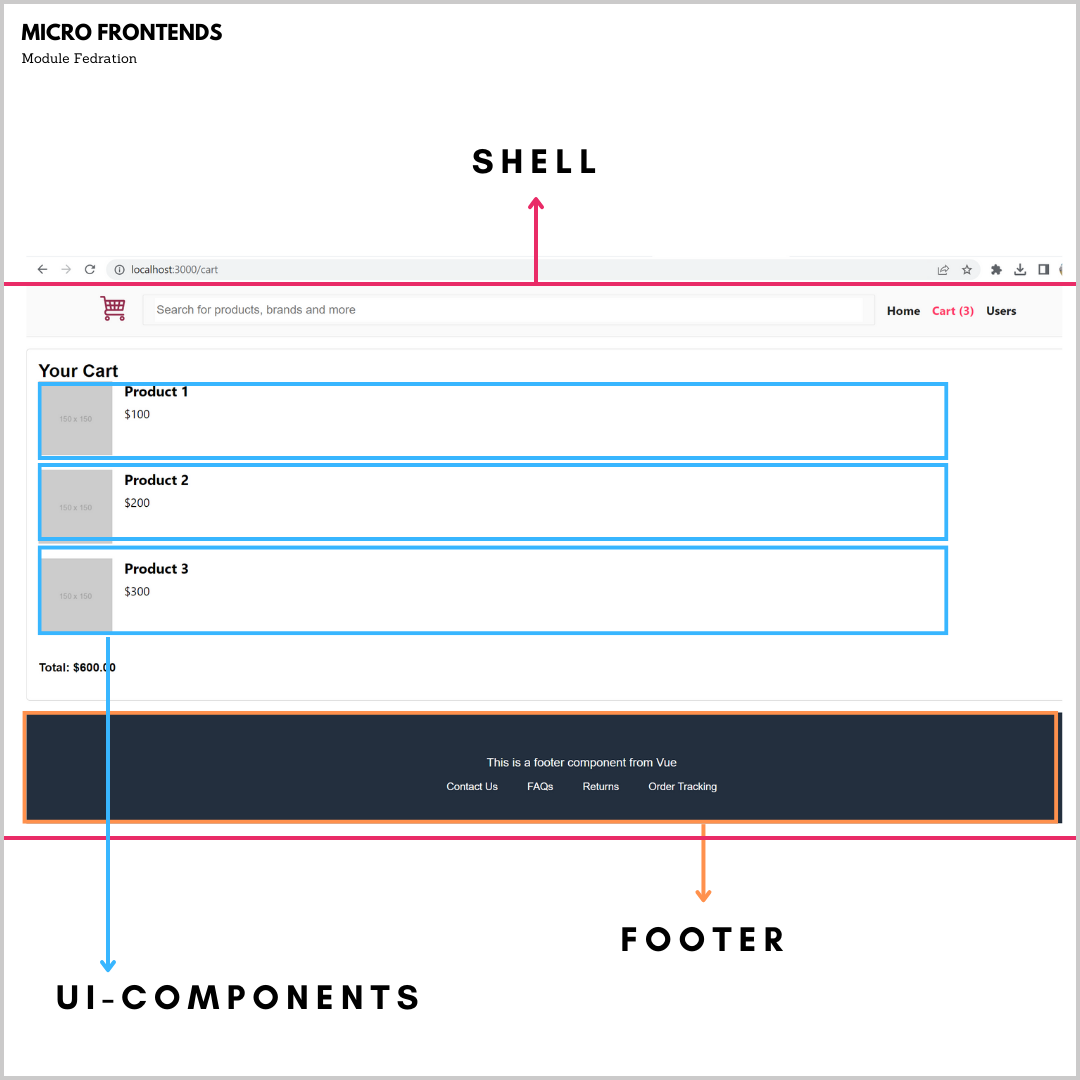
**UI Components:** This micro frontend contains all the reusable UI components that are used across the site. These include buttons, forms, alerts, and other common UI elements.



**Users**: This micro frontend allows the user to add new users to the site. It is built with React and contains a form for adding user details like name, email, and address.



**Footer**: This micro frontend is built with Vue.js and contains the site footer with links to social media profiles and other important pages.



Each micro frontend is built and deployed independently, and they communicate with each other using the Module Federation protocol. This allows us to build and deploy each micro frontend separately, without worrying about the impact on the rest of the site. It also allows us to easily swap out or upgrade individual micro frontends without affecting the rest of the site.

Overall, our e-commerce site built with Module Federation demonstrates the power of micro frontends in building scalable and modular web applications. It provides a flexible and efficient architecture that can be easily extended and maintained over time.

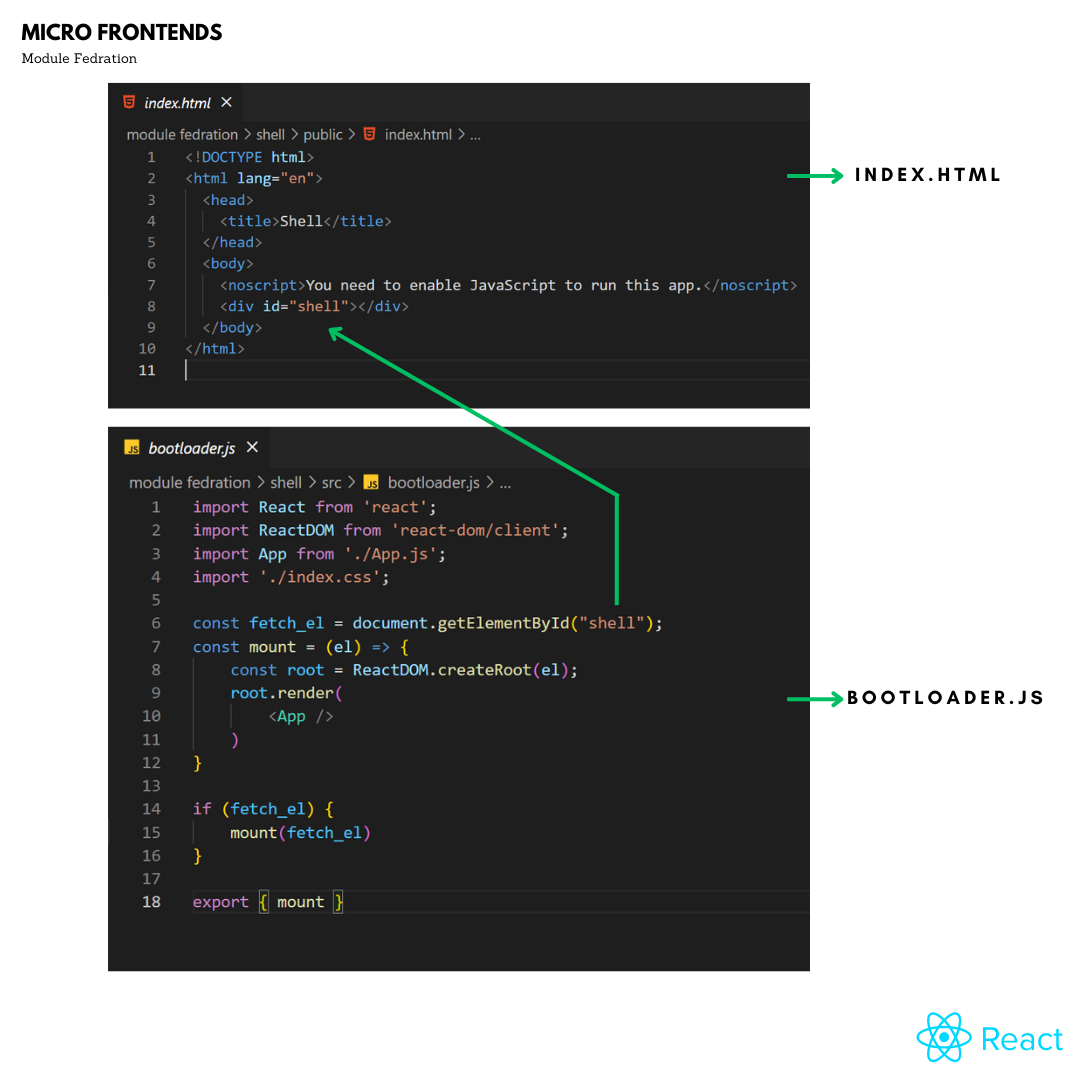
**Developer’s Guide**

**Project Structure – Root Service:**

****

The bootLoader.js file is a key component in the Module Federation architecture. It acts as a bootstrap file that is responsible for loading the required micro frontends dynamically at runtime.

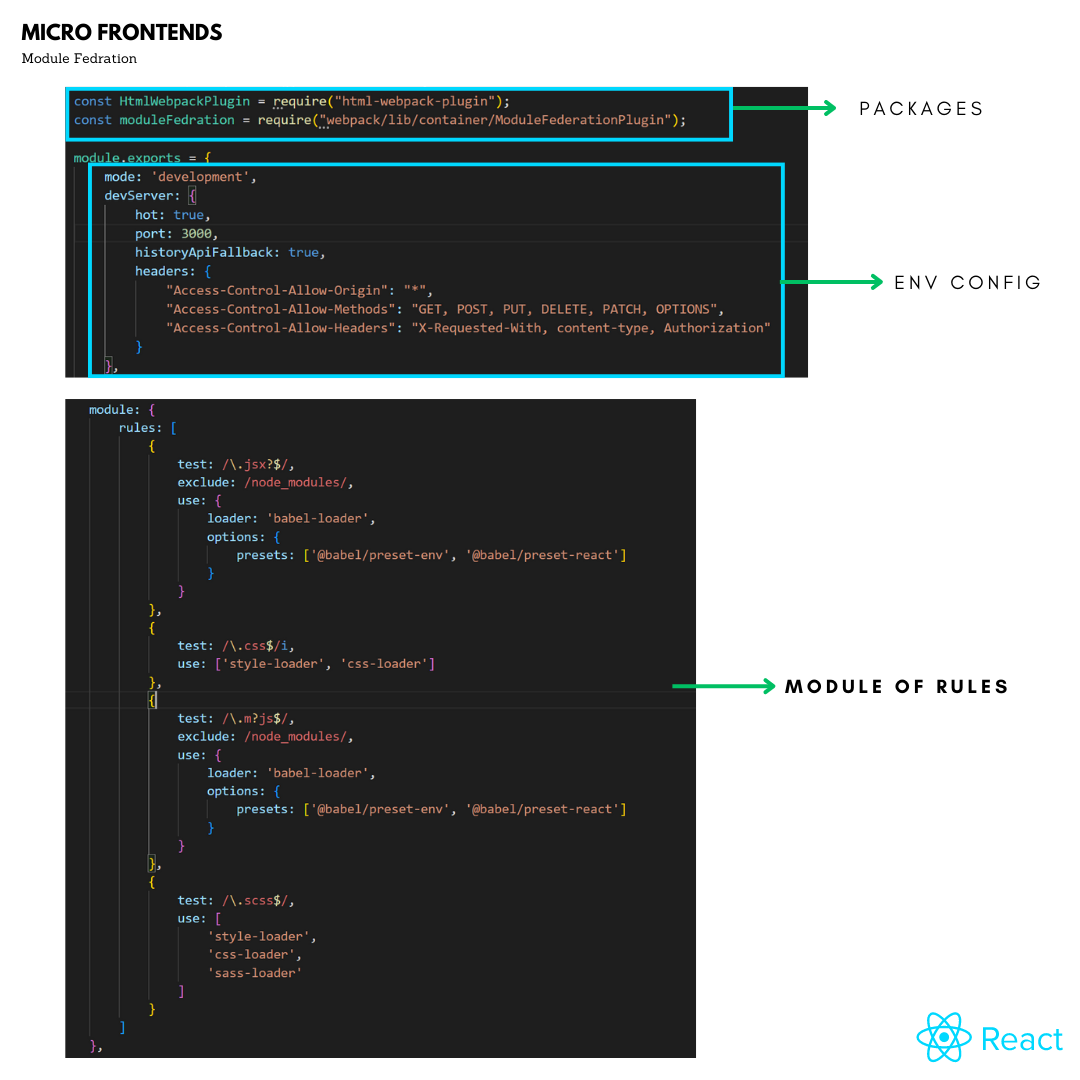
When a user visits a website that is built using Module Federation, the bootLoader.js file is automatically loaded by the browser. This file then uses Webpack to dynamically load the micro frontends as needed.



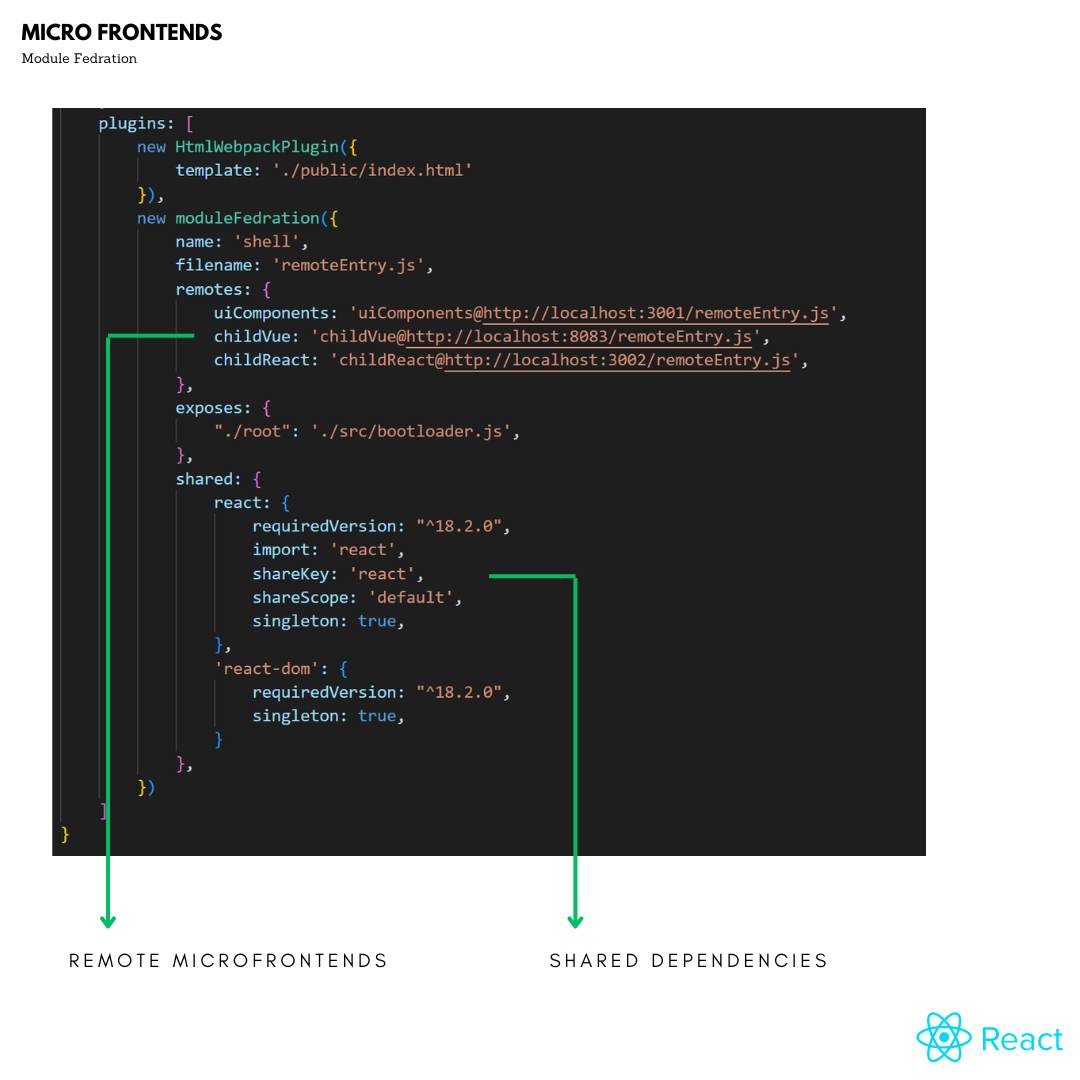
* The **fetch\_el** constant gets the DOM element with the ID "shell". This element is where the React application will be mounted.
* The **mount** function is defined to render the React application into the DOM. It uses ReactDOM.createRoot to create a root element and then uses root.render to render the App component into the root element.
* The **if statement** checks if fetch\_el exists. If it does, it calls the mount function and passes fetch\_el as an argument. This will render the React application into the DOM element with ID "shell".

Finally, the mount function is exported so that it can be used by other and the **Bootloader should be imported in Index.js file which is root of project** files in the project.

**Webpack Config for Shell (root service):**

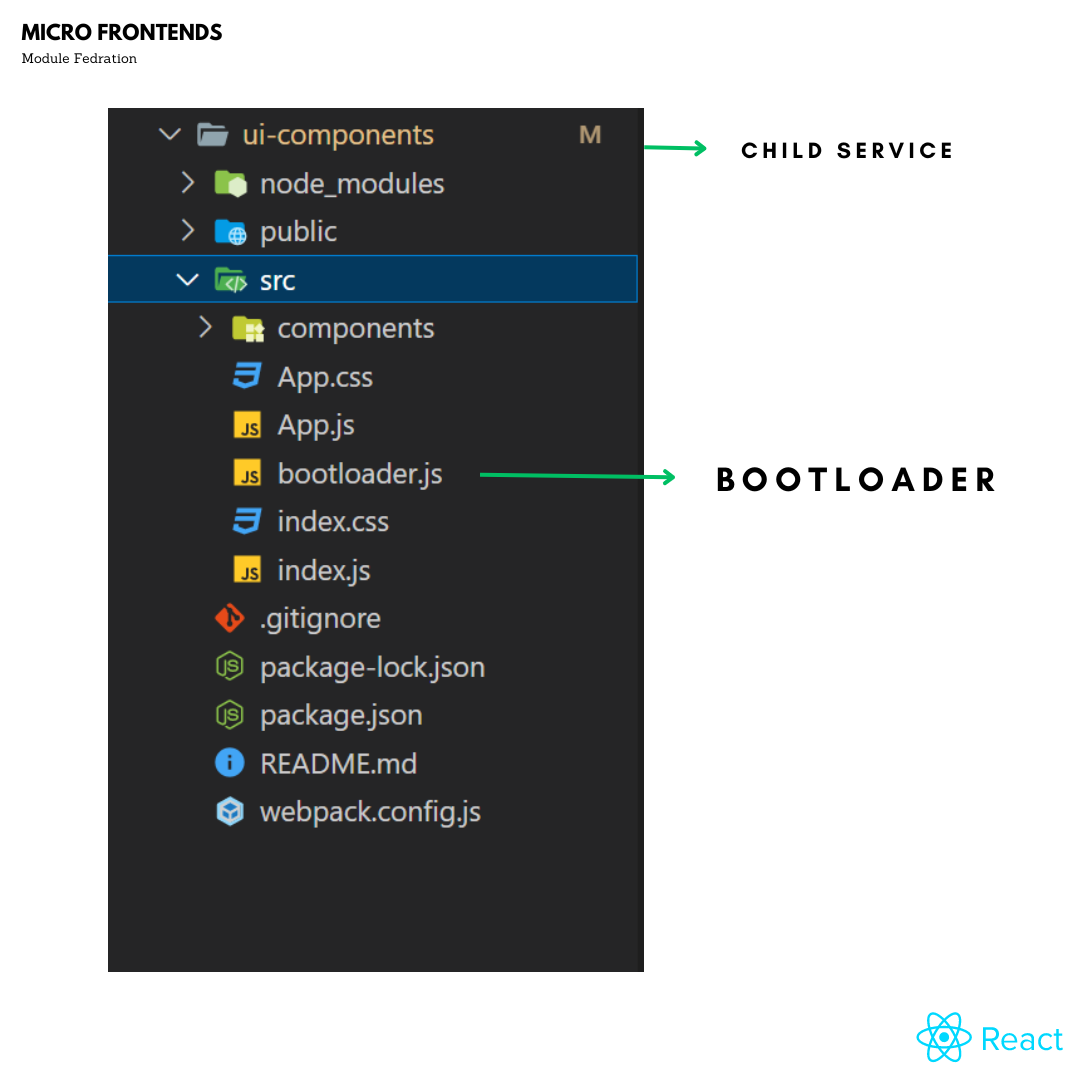
****

* **mode**: It specifies the mode of the application, which can be 'development' or 'production'. In this case, the mode is set to 'development'.
* **devServer**: It is used to configure the development server, which is used to serve the application. Here, we have set the server to run on port 3000 with hot reloading enabled, history API fallback enabled, and headers set for cross-origin resource sharing.
* **module**: It contains an array of rules, which are used to specify how the module should be transformed before being included in the bundle. Here, we have defined four rules:
* The **first rule** matches all .jsx and .js files, except those in the node\_modules folder. The babel-loader is used to transpile the code to a format that can be understood by older browsers, using the @babel/preset-env and @babel/preset-react presets.
* The **second rule** matches all .css files and uses the style-loader and css-loader to convert the CSS to a format that can be used by the browser.
* The **third rule** matches all .mjs files, except those in the node\_modules folder. The babel-loader is used to transpile the code to a format that can be understood by older browsers, using the @babel/preset-env and @babel/preset-react presets.
* The **fourth rule** matches all .scss files and uses the style-loader, css-loader, and sass-loader to convert the SCSS to a format that can be used by the browser.

****

* **plugins**: It contains an array of plugins, which are used to extend the functionality of Webpack. Here, we have defined two plugins:
* The **HtmlWebpackPlugin plugin** generates an HTML file with a script tag that includes the bundled JavaScript file.
* The **ModuleFederationPlugin** plugin is used to enable module federation, which allows the application to dynamically load remote modules at runtime. It defines the name of the module, the name of the file that contains the remote entry points, the remote modules that the application needs to load, the modules that the application exposes, and the shared modules between the remote modules and the application.
* **remotes**: This property is used to specify the remote entry points for the federated modules. It allows the application to use modules from other applications as if they were installed locally. In this configuration, there are three remote entry points specified: uiComponents, childVue, and childReact, each with their own URL.
* **exposes**: This property is used to expose specific parts of a module for other applications to use. In this configuration, there is only one thing being exposed: the ./src/bootloader.js file, which is the entry point for the shell application.
* **shared**: This property is used to specify dependencies that should be shared between the federated modules. This can help reduce the size of the application by not duplicating dependencies. In this configuration, the react and react-dom packages are being shared, with specific versions specified. The react package is also given a shareKey of react, which means that it can be shared between modules that use the same key. The react and react-dom packages are also marked as singletons, which means that there should only be one instance of them across all modules that use them.

**Project Structure – Child Service:**

****

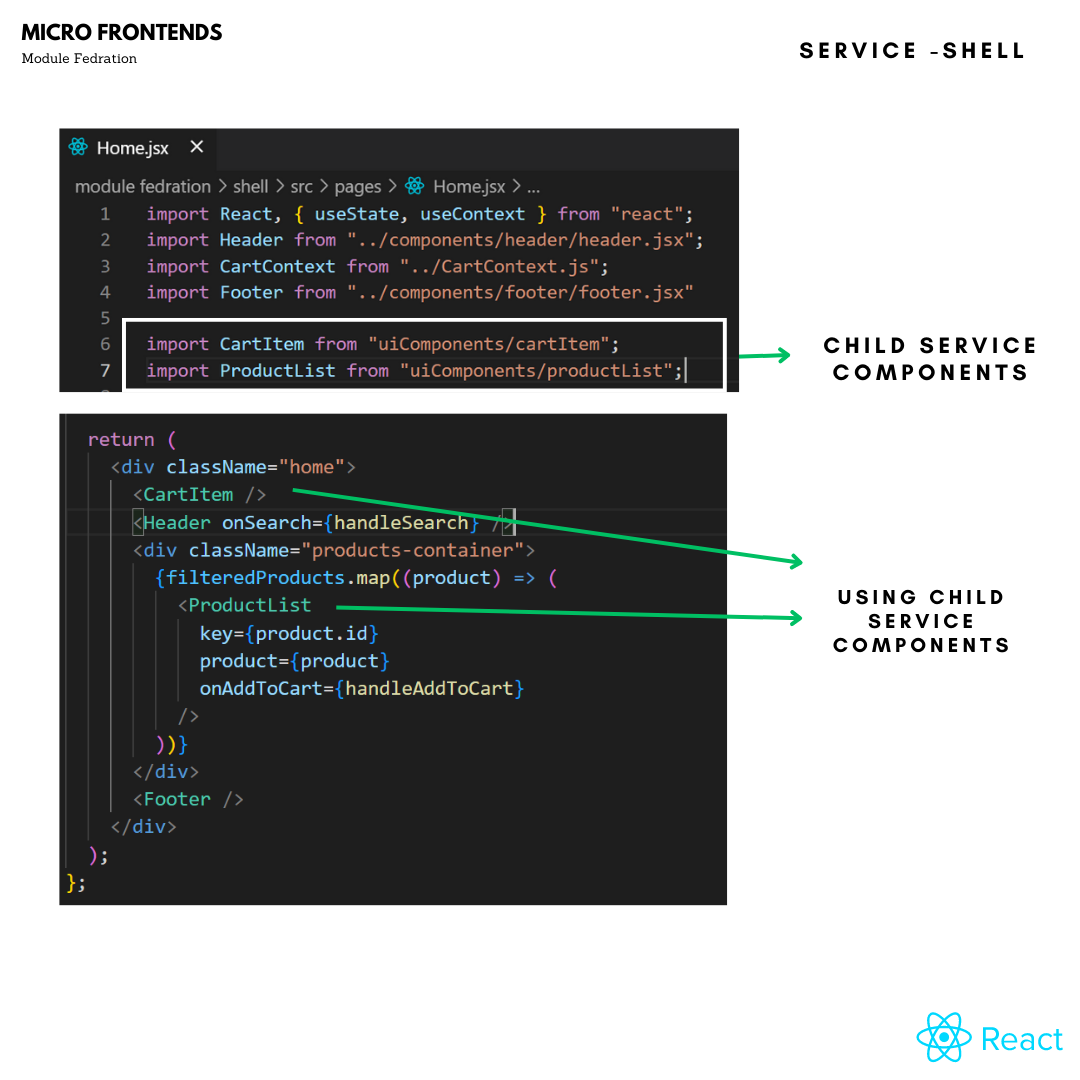
The bootLoader.js file is a key component in the Module Federation architecture. It acts as a bootstrap file that is responsible for loading the required micro frontends dynamically at runtime

It will act same as root boatloader.js only Div Id changes.

**WEBPACK CONFIG – CHILD SERVICE:**

****

* The exposes configuration in webpack is used to indicate which modules should be exposed by a module federation remote. In the code snippet provided, the exposes configuration exposes various components of a module.
* In this case, the components that are being exposed include banner, button, footer, productList, header, and cartItem. Each of these components is mapped to its corresponding file in the ./src/components directory.
* By exposing these components, other modules that consume this module can use them as if they were defined locally in their own codebase. This helps to promote code reuse and modularity, as well as reducing duplication of effort in building similar components across multiple codebases.



* The code is importing the ProductList component from the uiComponents module federation remote. This remote is declared in the webpack config under the remotes property. The ProductList component is then used in the Shell component's render method to display a list of products.
* The code is importing the ProductList component from the uiComponents module federation remote. This remote is declared in the webpack config under the remotes property. The ProductList component is then used in the Shell component's render method to display a list of productsBy importing ProductList from the uiComponents remote, the Shell component is able to use a component that is defined and managed by a separate codebase, without having to maintain its own copy of the component. This allows for better code organization, modularization, and reuse across multiple projects.

**Webpack packages and versions:**

webpack: version 5.75.0

webpack-cli: version 5.0.1

webpack-dev-server: version 4.11.1

html-webpack-plugin: version 5.5.0

babel-loader: version 9.1.2

css-loader: version 6.7.3

style-loader: version 3.3.1

**Technologies Used:**

ReactJS: 18.2.0

VueJS: 3.2.45

I hope that the information provided was helpful in understanding the concepts related to your project. If you notice any incorrect information in this document, please do not hesitate to reach out to me so that I can make the necessary corrections. Thank you!"