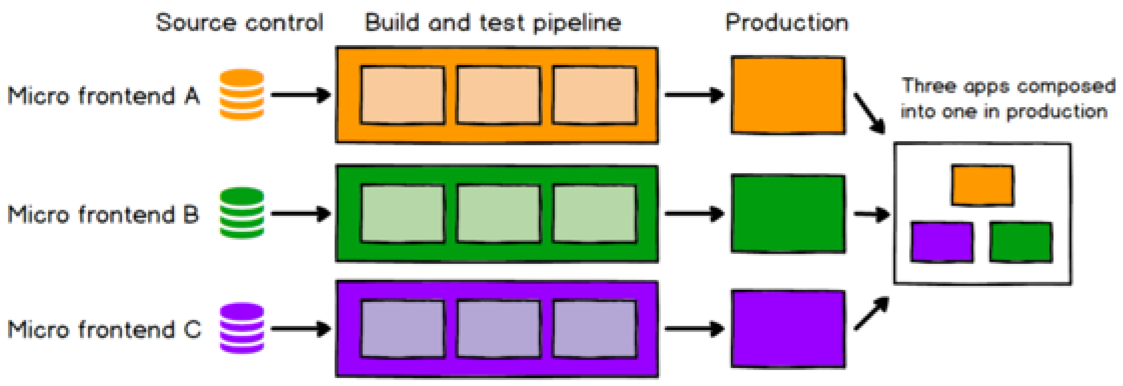
import { Meta } from "@storybook/addon-docs";

Architecture

The **commure Infinity OS Platform** employs a micro frontend architecture.

Micro frontend architecture is a design pattern in which a frontend, or **host application** is decomposed into *individual*, *semi-independent* **applications**, also known as **parcels**, working *loosely together* through an eventing mechanism.



Benefits of **applications**, or **parcels**, include:

- they can be much simpler and easier to reason about, implement, manage and maintain,
- allowing independent development teams to collaborate on an **application**, much more easily
- providing a means for migrating from an “old” app by having a “new” **application** running *side-by-side* with the old application
- they run in complete isolation in their own process meaning bringing down one one of them doesn't bring down the whole **application**
- allowing many teams to work simultaneously on a large and complex product (or "**application**").
- they can have and manage their own dependencies, whilst receiving core dependencies and components from the **application**.

Frontend codebases continue to get more complex. We MUST have a more scalable architecture that provides a way of drawing clear boundaries that establish the right levels of *coupling* and *cohesion* between technical and domain entities.

This architecture SHOULD allow us to

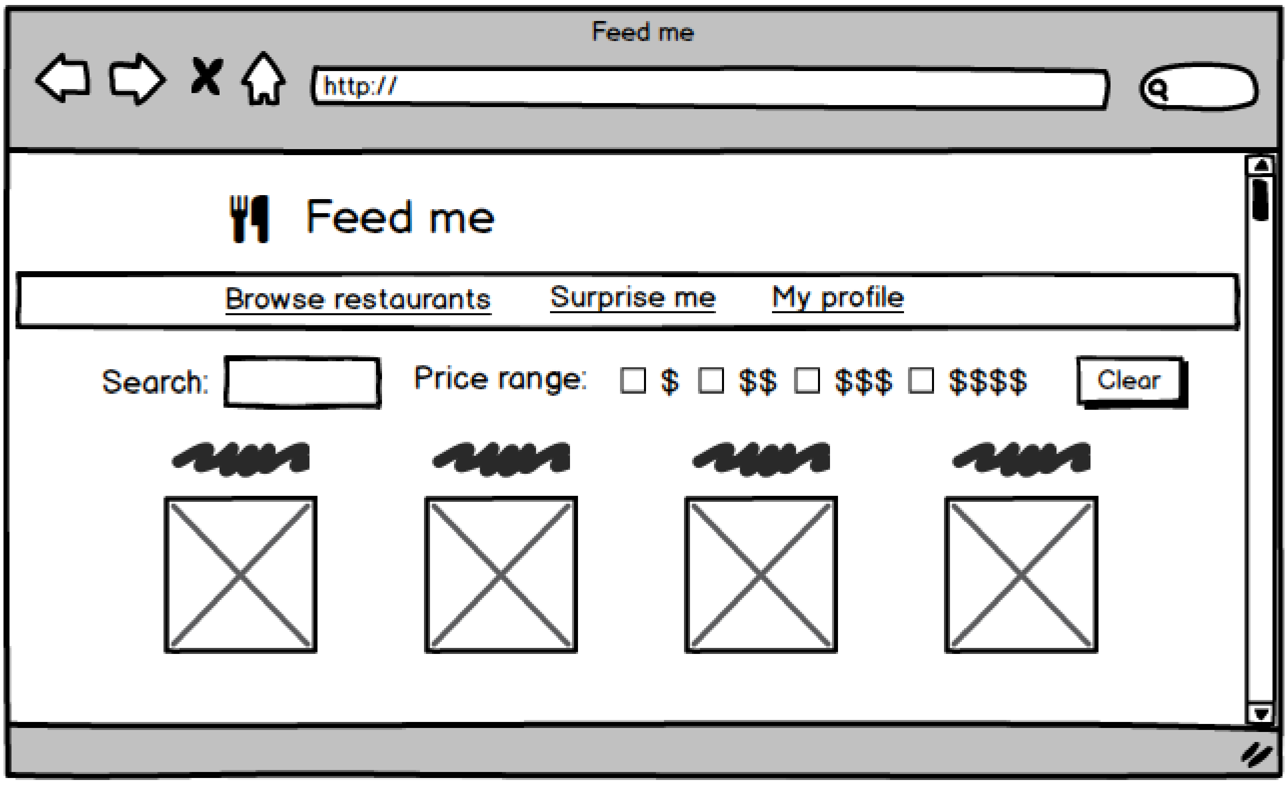
- scale software delivery across *independent, autonomous teams*
- support a variety of technologies and applications

Implications

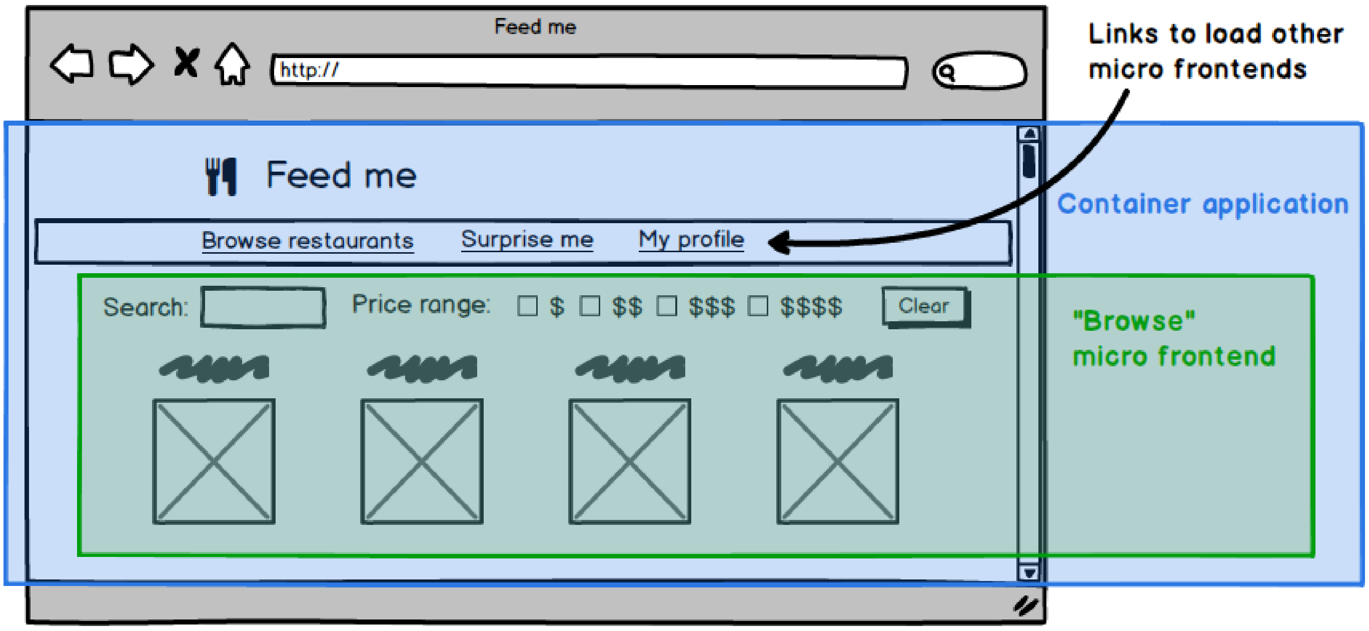
Application Design

From a design and decomposition perspective, the frontend, or **application**, MUST be "sliced" into **applications**, or **parcels**.

For instance, take the following wireframe:



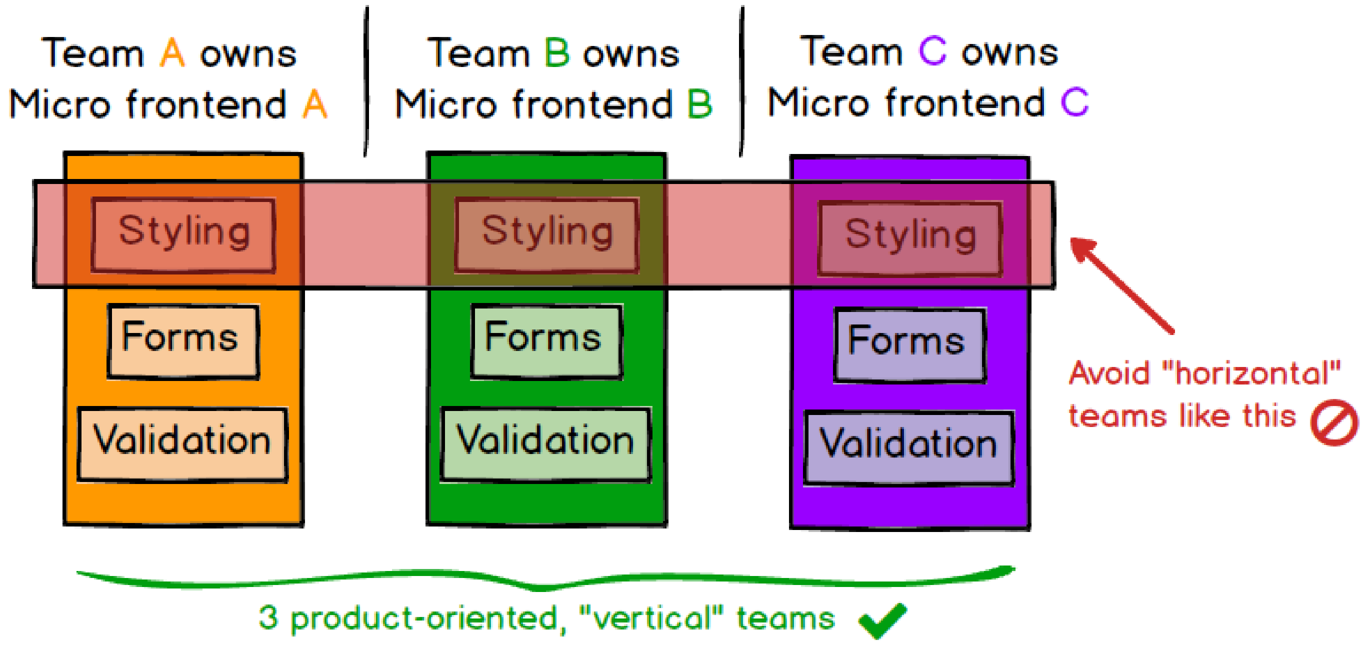
We can slice it like so:



Actually, in practice, we also "slice" out the navigation as its own **application** as well. Think of each **application** as serving its own purpose, or functionality, and having its own state.

Team Structure and Organization

Micro frontends imply, from a team organization perspective, that teams be assigned their *own* **application**, or **parcel**, fully owning everything needed to deliver that **application**. We MUST avoid cross-cutting, or "horizontal" teams.


Team A owns Micro frontend A
Team B owns Micro frontend B
Team C owns Micro frontend C
Styling
Forms
Validation
Styling
Forms
Validation
Styling
Forms
Validation
Avoid "horizontal" teams like this
3 product-oriented, "vertical" teams

UI/UX Inconsistencies

Given that different teams will create different **applications**,

- if a styleguide is NOT adhered to, or
- a single design system is NOT used,

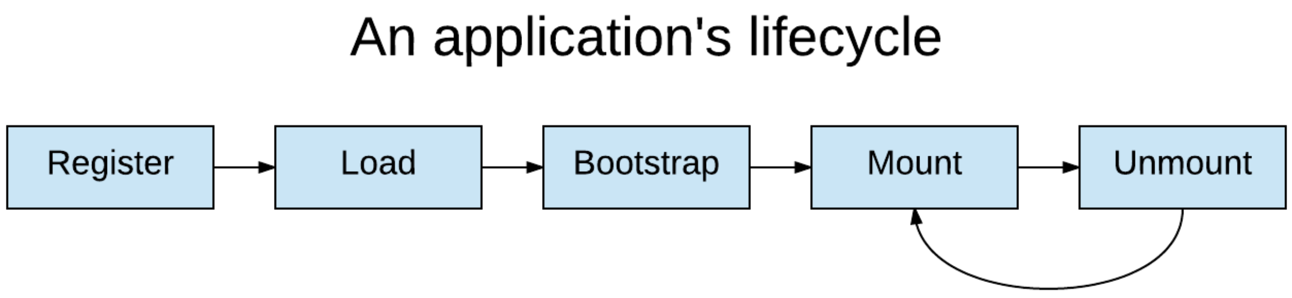
then UI/UX COULD become inconsistent.

Technologies Used

Orchestration

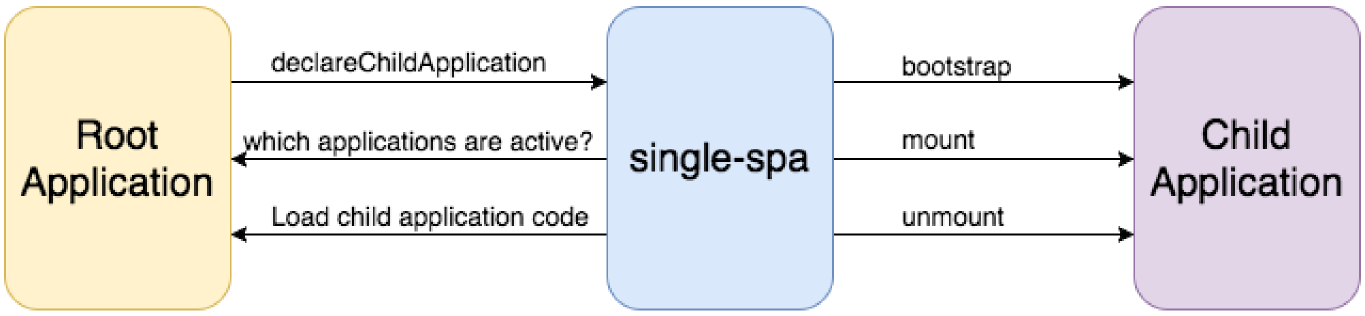
single-spa 5.x is used for registering and orchestrating which applications display for a particular URL route.

The lifecycles for a particular micro frontend/application are:



Orchestration first involves loading the modules for the micro frontend (also called an **application** or **parcel**) and `bootstrap`pping it. Once it is `bootstrap`ped, the orchestrator manages when micro frontends are `mount`ed and `unmount`ed, based on the result of a pure function that takes the `window`'s current `location` as its only argument---called an **activity function**.

SystemJS is used to load the micro frontends. After that, `single-spa` manages `bootstrap`, `mount` and `unmount` of each child application, or micro frontend.



Data Acquisition

Commure Data Platform via the Relay GraphQL specification.

Communication

Commure Event Bus is used for transporting events, keeping the **application**, its **pages** and their **components** loosely coupled.

UI Frameworks

provides core UI framework, and since the platform is technology agnostic, we have experimented with several: React, svelte, and Vanilla JavaScript

Design Systems

provides low-level UI components, and since the platform is Design System agnostic, we have experimented with several: For `React`: MUI, chakra For `svelte`: MDBSvelte

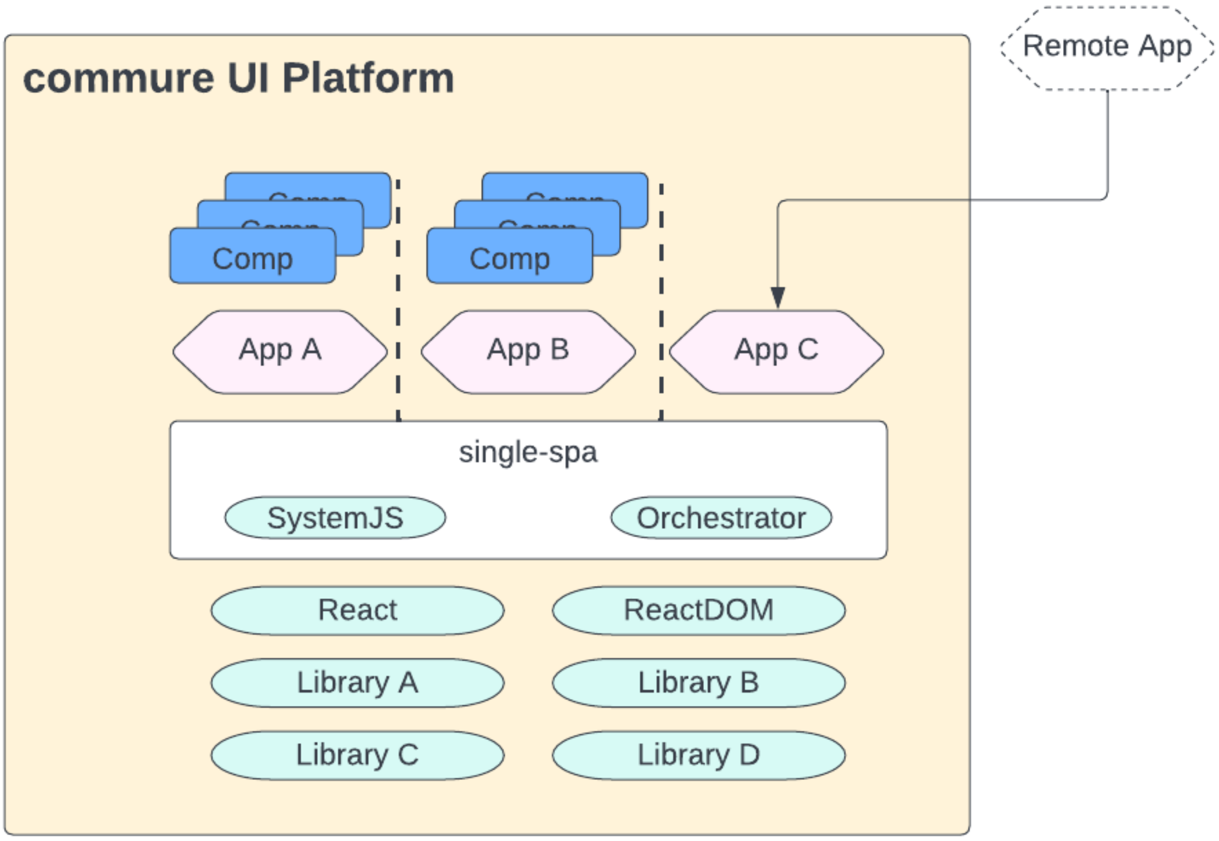
Module Build/Bundling

builds and packages application parcels into smaller chunks for the browser to handle. For `React`: webpack - v5 For `svelte`: rollup.js

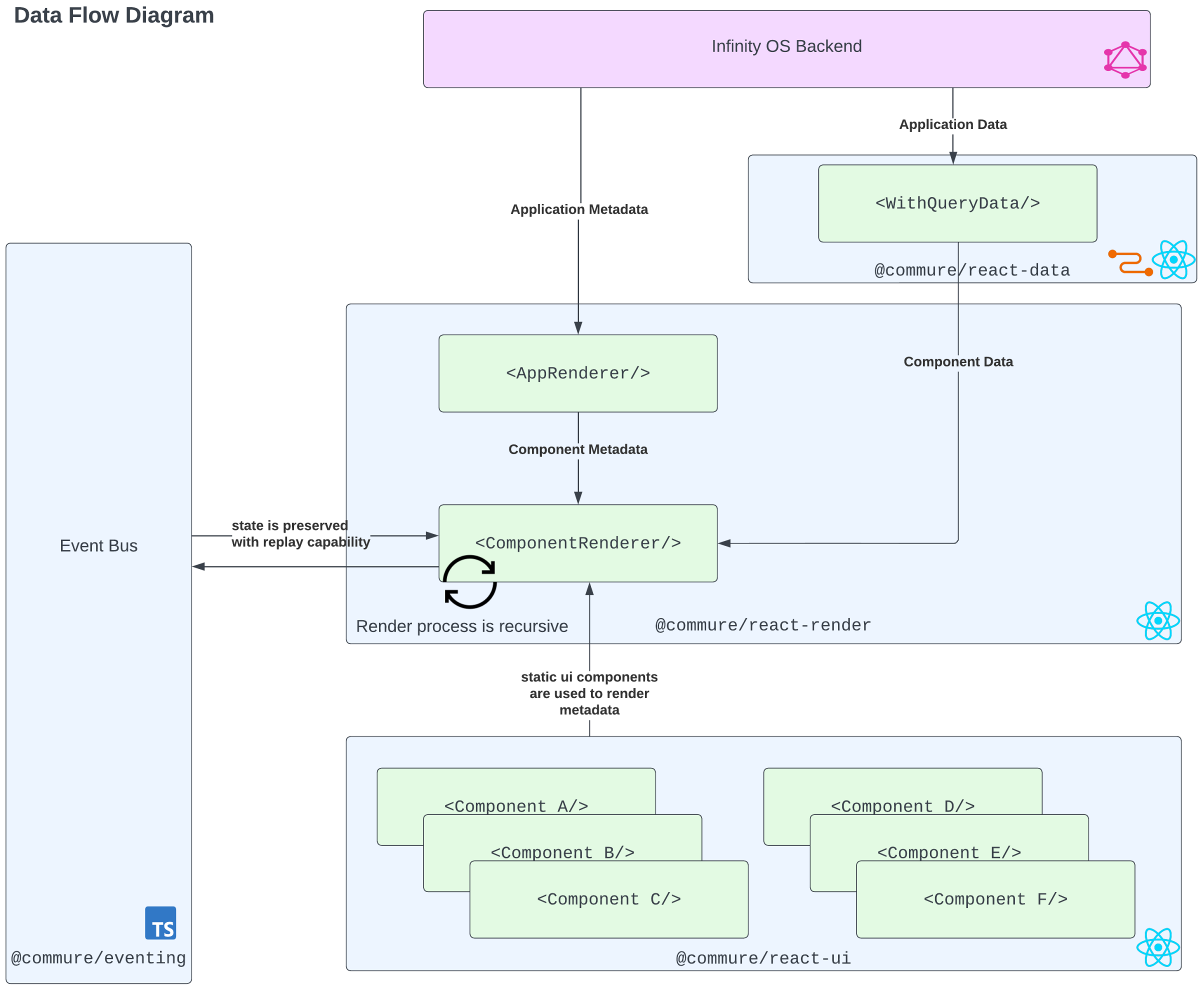
Unit Testing

used for unit testing applications and components. Jest

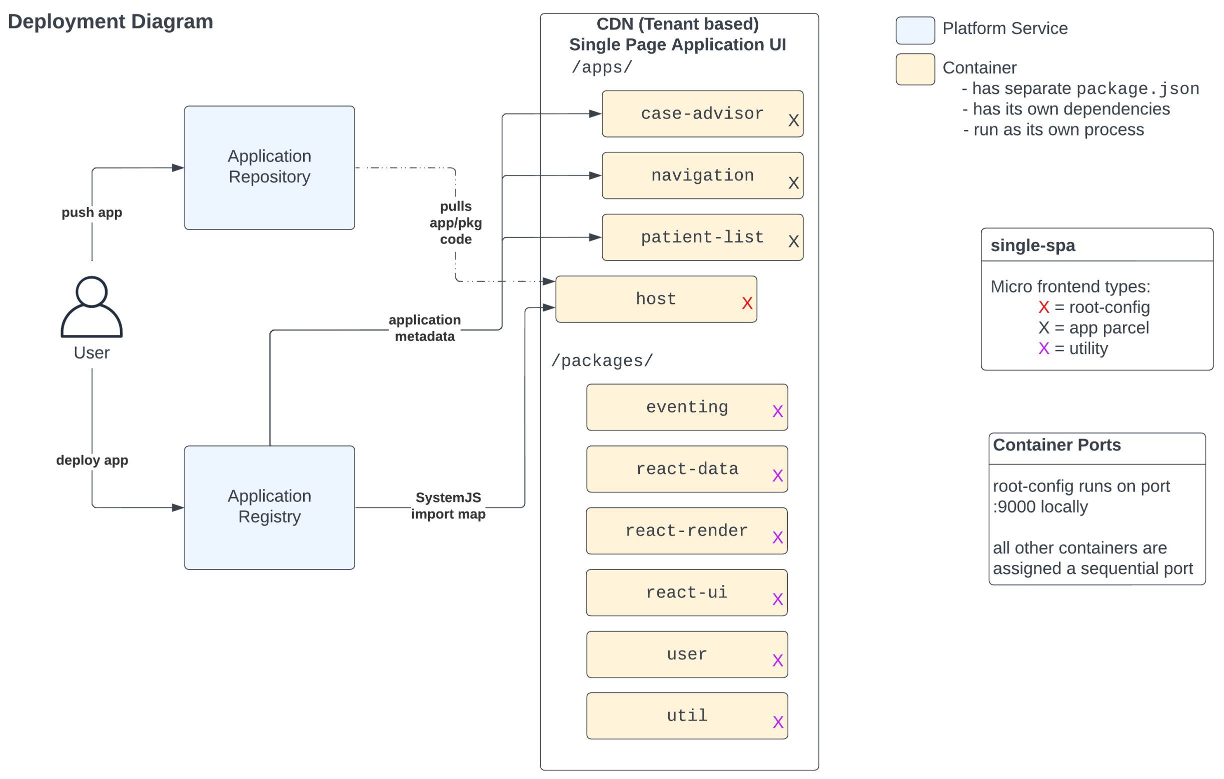
Block Diagram



Data Flow Diagram



Deployment Overview



Global Libraries

Global libraries are libraries that are provided by the **application** itself, typically via a CDN, when there SHOULD be ONLY a single version and instance of the library.

Examples include libraries like React, dayjs, and Commure Event Bus.

Platform Components

These micro frontends compose the platform.

Platform Component	Micro frontend type *	Purpose	Documentation	Package Source
`root-config`	`application`	The configuration for the main platform application.	README.md	@commure/host

Platform Component	Micro frontend type *	Purpose	Documentation	Package Source
Commure Event Bus	`utility`	Provides a technology agnostic communication mechanism for the platform.	README.md	@commure/eventing
Commure Utilities	`utility`	Provides utility methods for use with the platform.	README.md	@commure/util