创心

第十四届

D2前端技术论坛

2019-12-14 杭州和达希尔顿逸林酒店

二号会场翻译群 - Translation...

7人

扫一扫群二维码，立刻加入该群。

A journey into small-to-micro frontends

Over the past several years...

- UIs have become more elaborate and their development more involved
- Teams got bigger, and so did code-bases, but not all the tools scaled
- Making use of new technology has become increasingly more expensive

How can we build UIs fast, smart and happy without sacrificing quality?

Five years ago...

Our system had a lot of entangled state, which made it hard to grow the product and the team without sacrificing quality...

- Developers had to keep a lot of context in mind to avoid introducing regressions
- Debugging was difficult
- Unit tests were very hard to write and so they weren't

Decided to do a rewrite...

- Had to be progressive; new features will be added to new system
- Estimated 3 months, took 24
- Maintaining two systems in-tandem took a lot of effort

Post-rewrite and two years later, application has become huge...

- Adopting new tech required a lot of investment
- Refactors were both risky and costly
- Tooling became a bottle-neck: building is slow, testing and deployment too

A second application was developed...

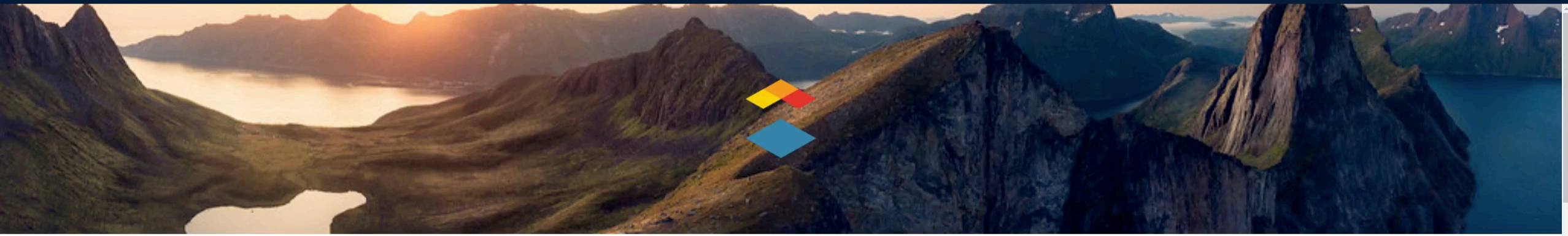
- The UX suffered: users would jump between two domains and do full-page reloads upon navigation
- Developers (re)implemented their own version of authentication, navigation and other generic things because reusability wasn't always straightforward

My Learning - Bridge

instructure.bridgeapp.com/learner/courses

MY CAREER

1ON1 GOALS ASSESSMENTS CAREER PLAN LEARNING TOOLS



MY LEARNING LEARNING LIBRARY TRAINING CALENDAR

SEARCH

List Filter by

REQUIRED

Employee Assistance Program (EAP)

Due 292 days ago / About 3 minutes remaining

PI Training for Bridge

Due 102 days ago / About 17 minutes remaining

Workday: Employee Training

Next step Workday: Employee: Intro to Workday / No due date / Takes about 4 minutes

A third application was developed...

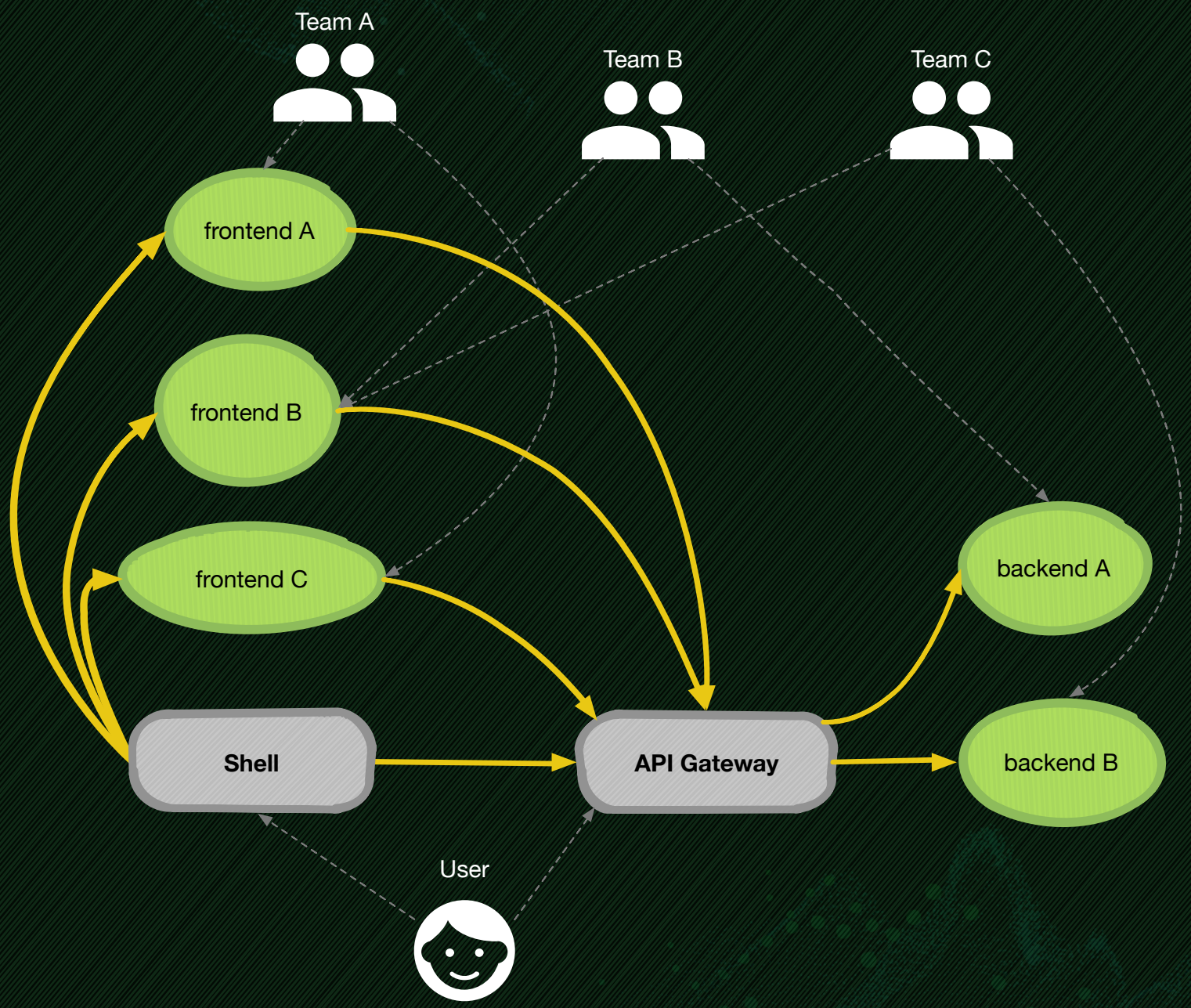
- The UX now abysmal
- Lots of redundancy in data and in logic
- Identical bugs appeared in multiple applications, and each had to be fixed separately

Built a library to make navigation re-usable...

- Newer applications started using the library: good
- First application couldn’t use it without great investment, so it kept its variant
- Coordinating upgrades to the navigation was very difficult

Taking a step back, we looked for a solution that addresses our main problems:

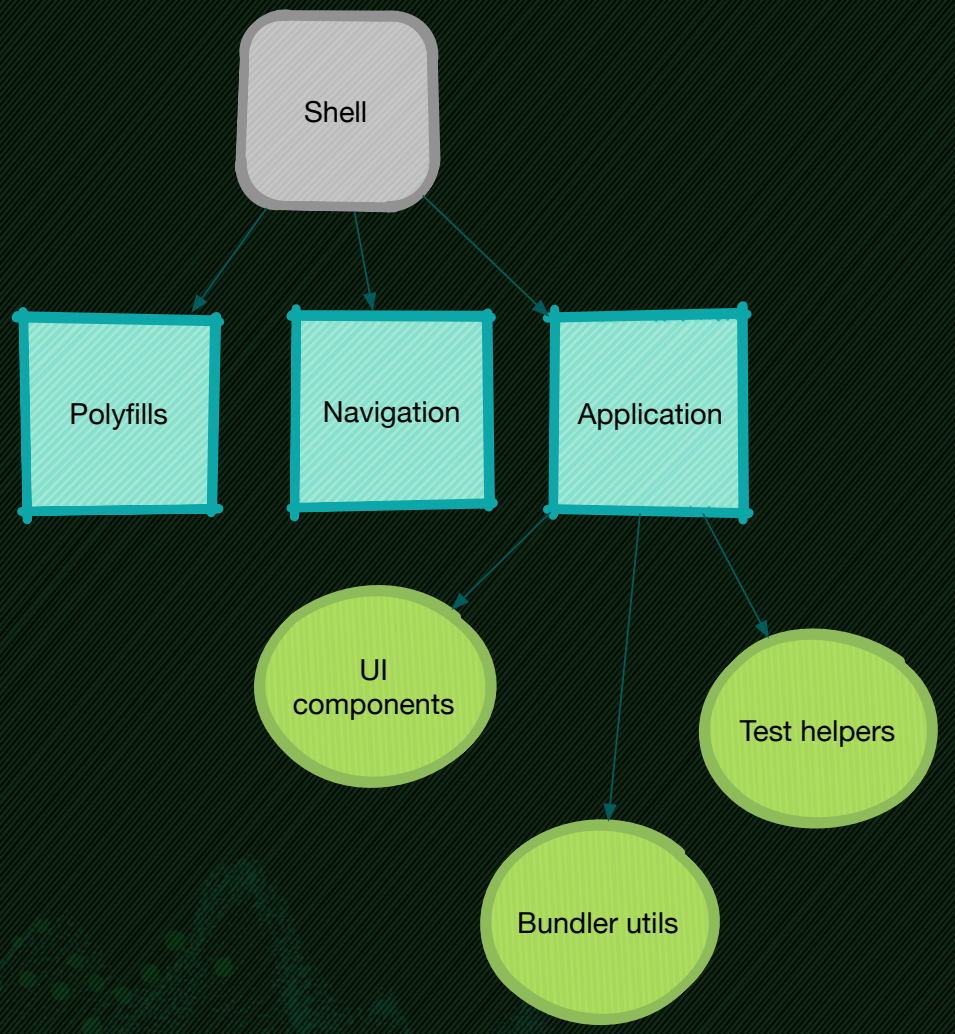
- Business logic is repeated
- Change is expensive
- Extending the application often causes regressions
- On-boarding new developers takes a long time


Team A
Team B
Team C
frontend A
frontend B
frontend C
backend A
backend B
Shell
API Gateway
User

- DRY in a different way

- Applications can become an alternative to libraries in terms of reusability

- Examples of such “reusable” applications include: navigation, polyfills, error reporting, analytics tracking



- Applications can remain small without prohibiting product growth

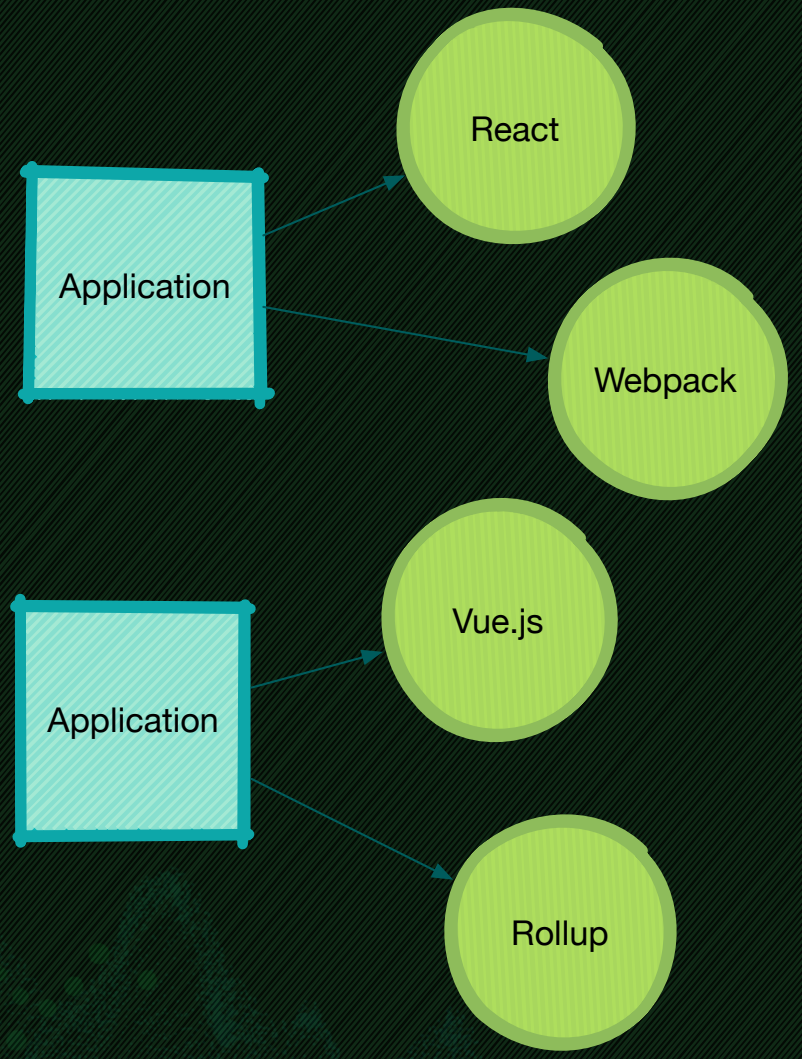
- Fewer things for a developer to know about outside the scope of their own applications

- The complexity didn’t go away, however; it only moved to a different layer

- Teams can experiment with new tech and methods with far less risk

- Cost of failed experiments is much lower since they're confined to smaller subsets of the whole product

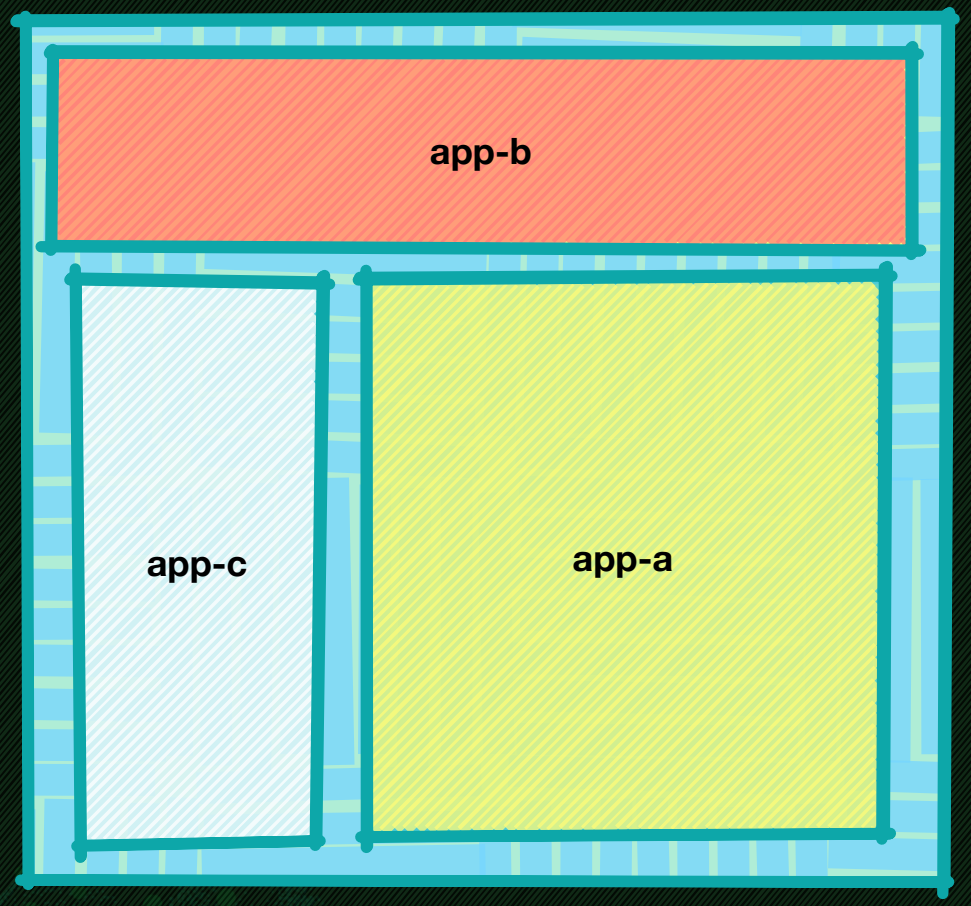
- Can potentially *reduce* productivity if the stacks vary too wildly



- Can compose a screen of components from entirely different applications; and the user wouldn't know the difference

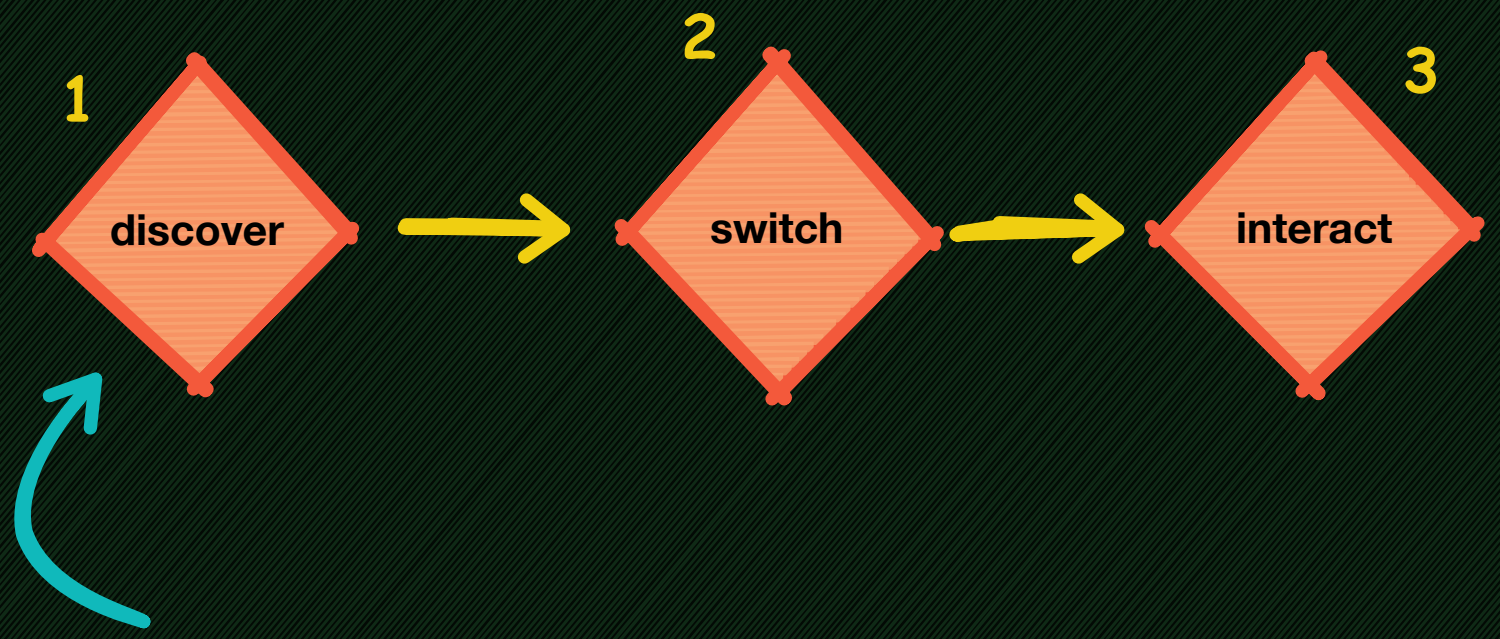
- Practical applications include: a member landing page, dashboards, and admin sections

- Implementation could become complicated (e.g. for animated transitions between disparate components)



For the system to function at a minimum, we need three things:

1. A way to discover applications (e.g. Consul or Kubernetes)
2. A way to switch applications in and out (e.g. SingleSPA)
3. A way to interact with the applications (e.g. supply information, manage events)


1
discover
2
switch
3
interact

We need to know which applications are available, and how they can be loaded...

- An application specifies its entry JavaScript file(s) and additional resources (e.g. CSS assets) in a dedicated manifest file

- Manifest would also contain the rules for activating the application (e.g. a route)

```yaml
---
# file: d2-demo/shell.yml
- application:
    id: d2-demo
    entry:
      - '/d2-demo/main.js'
    styles:
      - '/d2-demo/main.css'
    active_when:
      - location.pathname.startsWith('/d2/')

```

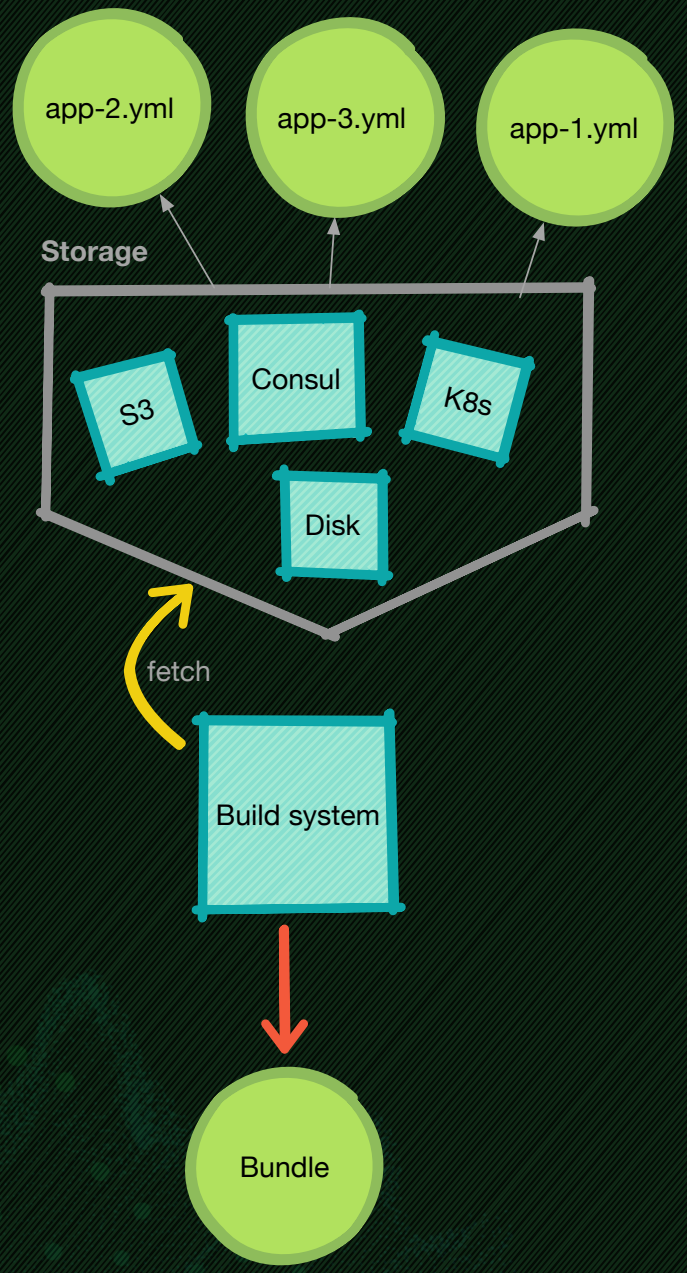
The manifests of all applications are an input to the shell, but **how** do we load them? We could:

- Bake them into the shell's bundle at build-time, or
- Fetch them from an HTTP endpoint at run-time

A static discovery system in which the manifests are fetched at build-time and “baked” into the shell’s bundle...

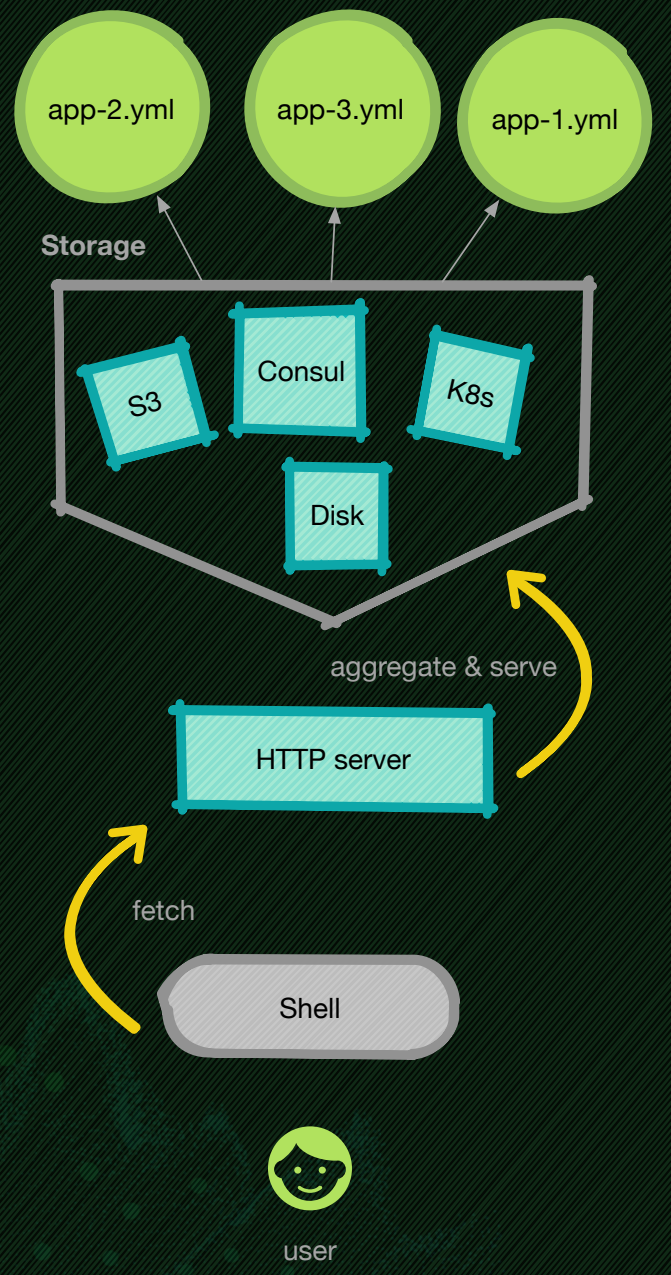
- Simple to implement and has no additional HTTP overhead at run-time

- Requires a re-deploy of the shell anytime the manifests change

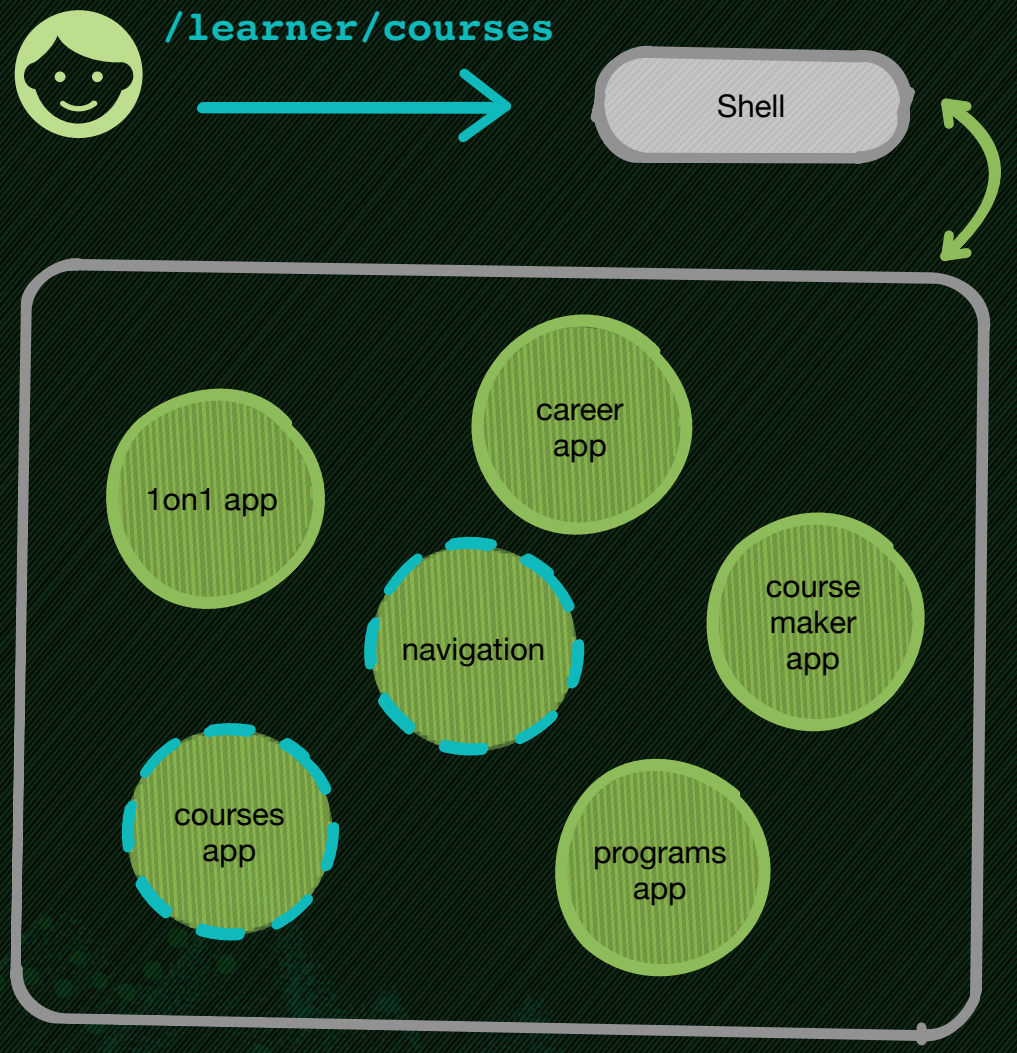


A dynamic discovery system in which the manifests are **aggregated and fetched** at run-time over HTTP...

- Decouples the deployment of the shell from the deployment of (application) manifests
- Has some HTTP overhead
- Requires more end-to-end testing to be resilient
- Enables a better DX; developer doesn't need to rebuild the shell when working on their manifest



Manifests tell us about the applications, how they can be loaded, and when they should be activated... but *how* do we activate them?



- A library that makes it easy to run multiple applications at the same time

- The activation logic can be based on the URL, but can also incorporate business logic (e.g. product tier)

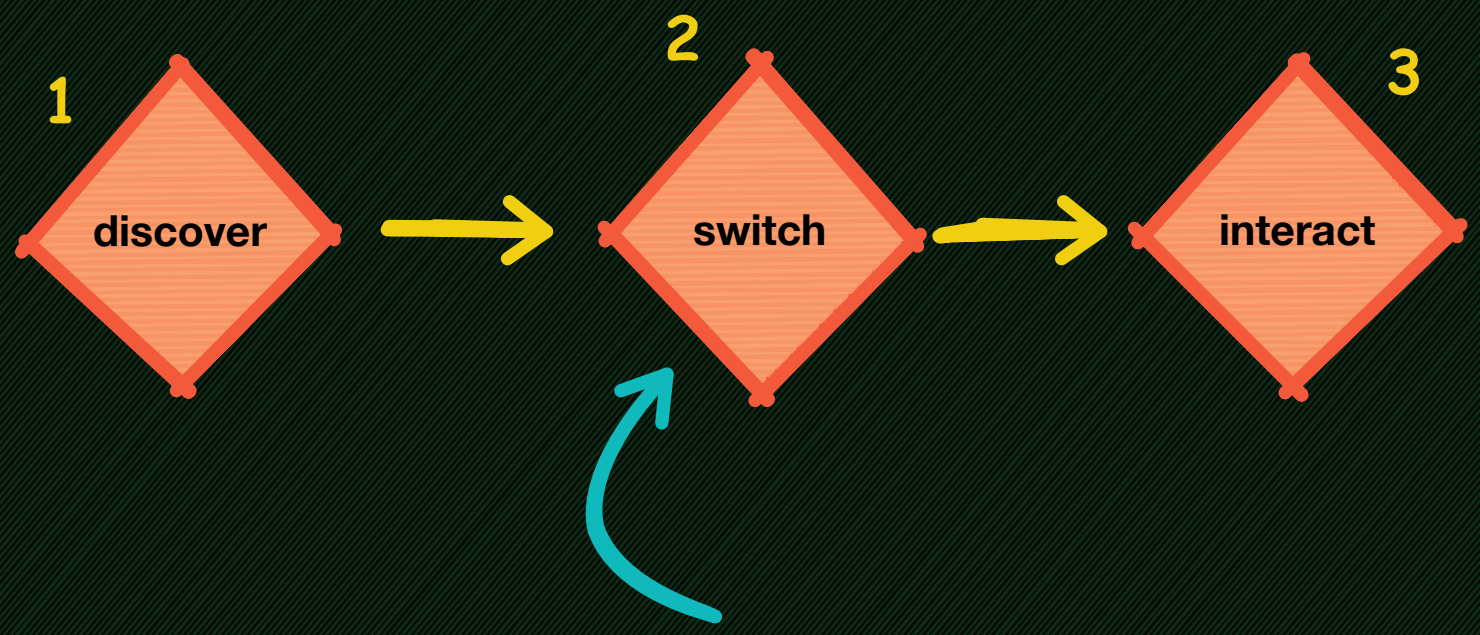
- You still need to **load** the application assets yourself

- For an application to be activated, its JavaScript (and potentially CSS) assets must be loaded first

- A library such as load.js can do this for you; the URLs are available in the manifests the shell has access to

- This can get complicated due to fingerprinting (e.g. webpack chunk hashes)

```
1 import load from 'loadjs'
2
3 const loadApplication = manifest => load([
4   ...manifest.entry,
5   ...manifest.styles
6  ], { returnPromise: true })
```


1
discover
2
switch
3
interact

- Shell needs a way to tell an application to start and to stop -- *switching*

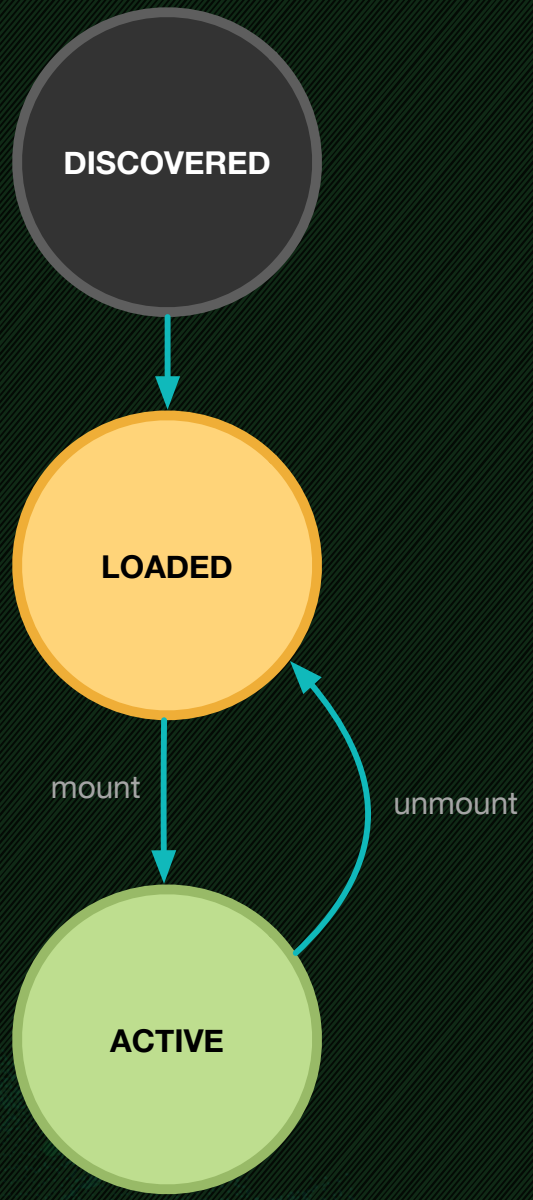
- With an application's source code loaded, the shell needs a reference to it so that it can interact with it

- Applications are expected to export such references *somewhere* the shell knows about (e.g. `{window.SHELL_APPLICATIONS}`)

```
// define ourselves if Shell is running:
if (typeof window.SHELL_APPLICATIONS !== 'undefined') {
  window.SHELL_APPLICATIONS.push({
    id: 'd2-demo-app',
    mount: () => {},
    unmount: () => {}
  })
}
```

An application is no longer “always on” as was traditionally the case...

- May require re-structuring of existing applications
- When asked to shut down, an application must clean up its resources
- Incidentally, this makes an application easier to test and promotes better coding practices



```
1  export function mount({ domNode, location }) {
2    return Promise.all([
3      startRouter({ location }),
4      ReactDOM.render(<MyApp />, domNode)
5    ])
6  }
7
8  export function unmount({ domNode }, [ router ]) {
9    return Promise.all([
10     router.stop(),
11     ReactDOM.unmountComponentAtNode(domNode)
12   ])
13 }
```

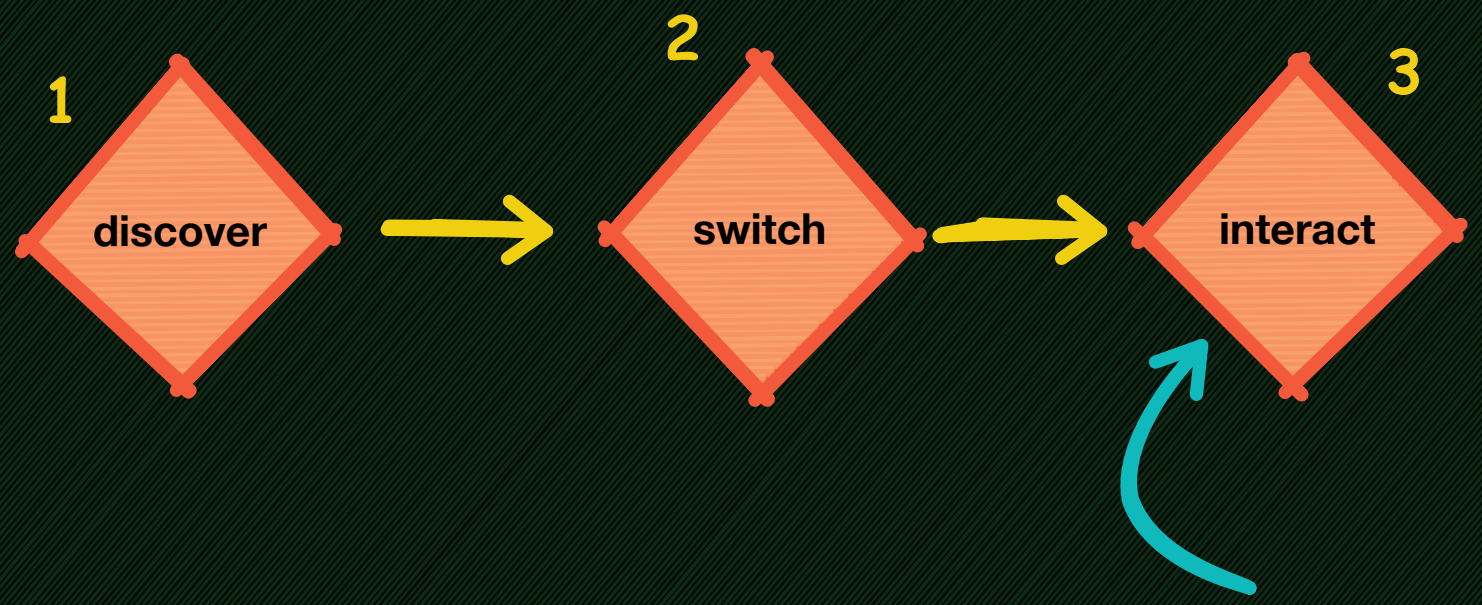
- A library (like an SDK) can be used to abstract the details of registering an application's definition with the Shell

- This library would also be a good place to provide backward-compatibility

- A definition must be accompanied by some version tag (doesn't need to be semver)

```
import Shell from 'shell-sdk'

Shell.define('d2-demo-app', {
  version: 1,
  mount: () => {},
  unmount: () => {}
})
```


1
discover
2
switch
3
interact

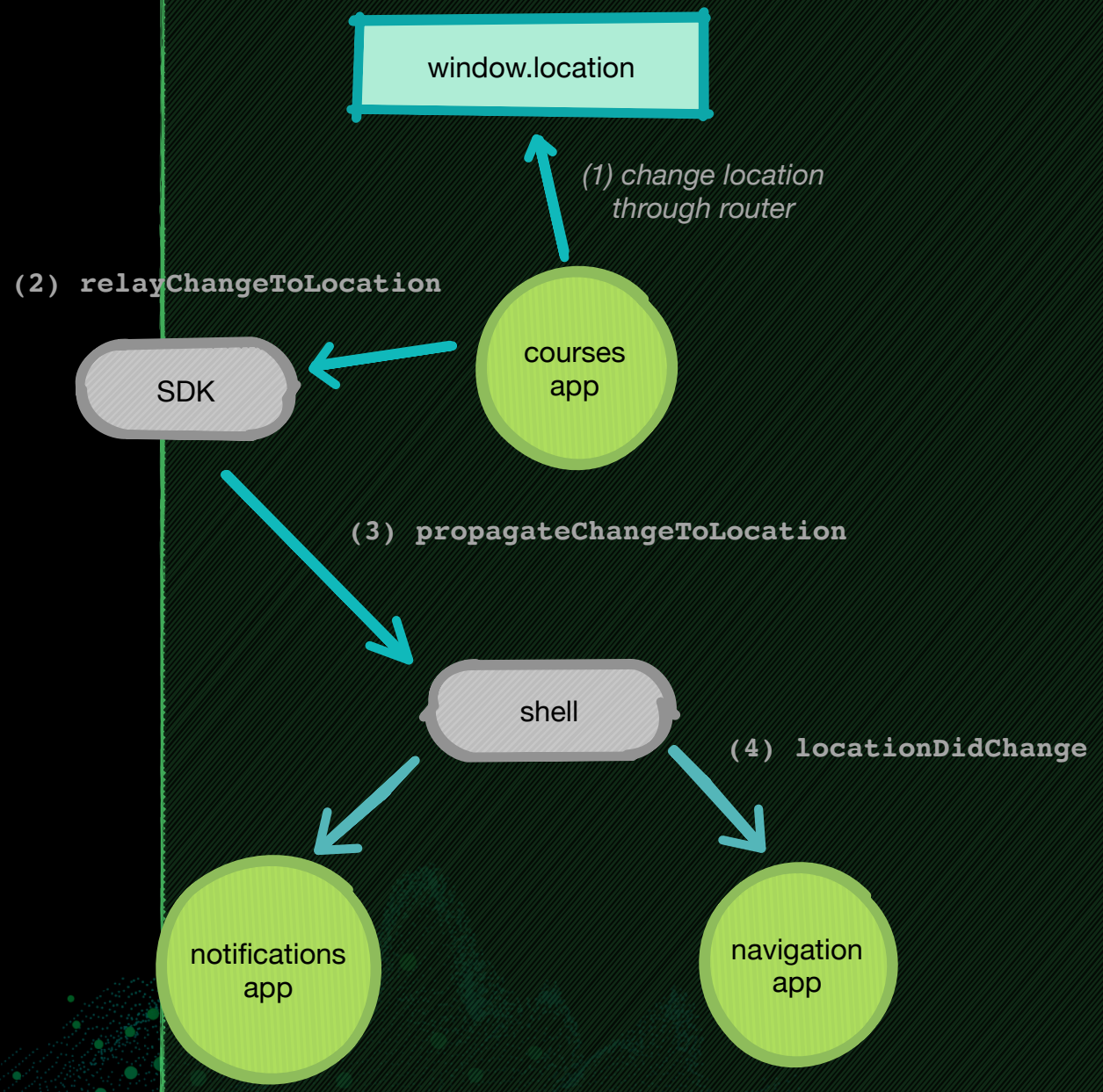
- Shell has to organize the different applications in the DOM tree
- Application can request a specific placement in its manifest
- Shell provides each application with a blank DOM node when it’s about to be mounted
- Application is then free to populate that node

```
1  const singleSpa = // ...
2  const shellDOMNode = document.createElement('div')
3  const applicationDOMNodes = {
4    'd2-demo-app-1': shellDOMNode.appendChild(document.createElement('div')),
5    'd2-demo-app-2': shellDOMNode.appendChild(document.createElement('div')),
6  }
7
8  singleSpa.addEventListener('change', activeApps => {
9    // ... reorder DOM nodes if needed here ...
10   for (const appId of activeApps) {
11     definitions[appId].start({
12       domNode:  applicationDOMNodes[appId]
13     })
14   }
15 })
```

- Global resources like {location} and {localStorage} are no longer “owned” by an application

- Access and changes to these resources must be communicated to the shell

- Shell should reconcile its view of these resources and broadcast changes to other applications so that they can do the same



```
import Shell from 'shell-sdk'

Shell.define('d2-demo-trigger-app', {
  mount: ({ domNode }) => {
    const router = // start a router...

    domNode.addEventListener('click', () => {
      router.transitionTo('/d2/demo').then(() => {
        Shell.relayLocationChange({
          from: 'd2-demo-trigger-app'
        })
      })
    })
  },
  // ...
})
```

```
import Shell from 'shell-sdk'

Shell.define('d2-demo-listener-app', {
  // ...
  locationDidChnage: (appState, { location }) => {
    console.log('location has been changed to:', location.pathname)
  }
})
```

- An application must free any resources it acquired during its life-cycle;
  - Timers (setTimeout / setInterval)
  - Animations (requestAnimationFrame)
  - Event listeners
- An application must stop any system that's listening to changes, like a router
- An application must remove itself from the DOM if it rendered anything

- Information that may be required by multiple applications can be resolved once in the shell and passed down to the applications that require it:
  - Reduces network overhead
  - Removes redundancy in code
- Examples include: user, account, and session data
- As with the shared JavaScript resources (like `window.location`), applications must take care to relay any change they make to this shared information (e.g. changing user's preferred locale)

- Fewer resources are available to any one application in this shared execution environment
- Memory footprint can potentially become a concern
- Duplicate code (e.g. large libraries) is worth “hoisting” to the shell layer
- Mechanism for hoisting will be heavily influenced by the bundler(s) in use
- Webpack externals are effective for this task

- Applications may need to “integrate” with each other in some aspect:
  - *Navigation application renders links to pages in other applications*
  - *Toast application renders a notification in response to an event handled by another application*
  - *Error application renders some UI when another application signals a 404/403/401*
- It’s important to not introduce tight coupling between applications
- It’s also just as important to not couple the shell to an application
- The {window} can be used as an event bus for dynamic interactions
- Manifests can be used for interactions that can be expressed statically

My Career 1on1

instructure.suite.bridgeapp.com/talent/career/1on1

MY CAREER

1ON1 GOALS ASSESSMENTS CAREER PLAN LEARNING TOOLS

+ SHARE AN AGENDA

1on1's with my Manager & Peers

Performance Conversation

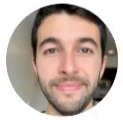
Derek DeVries

Dir, Engineering

Prepared: Derek Me

CREATE AGENDA

Check-in



Andres Soto

TL, Software Engineer

LAST MET 250 DAYS AGO

VIEW AGENDA

Invite someone to share a 1on1 agenda with you.

+ SHARE AN AGENDA

第十四届
D2前端技术论坛

- Instructure - Bridge
- SingleSPA
- Micro-frontends.org
- Micro frontends—a microservice approach to front-end web development
- Including Front-End Web Components Into Microservices
- Front-end Microservices at HelloFresh
- Microservices to Micro-Frontends
- Introduction to Micro Frontends
- Modernizing Upwork with Micro Frontends
- Micro Front-Ends: Available Solutions
- Custom Elements Framework Support

github: amireh
email: ahmad@amireh.net