- Philosophy: Micro-frontends vs. traditional SPAs
- Build
- Deploy
- Versioning

Feature comparison

Feature / Workflow	MONO REPO / SEPARATE APPS	MULTI REPO / MICRO FRONTENDS	Effort
Granular Updates how do you do granular updates for a specific application. So lets say we have the pricing ui in place and we start doing auto ordering ui and we want to update the auto-ordering app in an automatic way which means creating a pull request for a feature and the CI creates an automatic version • if you deploy everything (all the apps) the code is changed so you have to do a smoke test for all the apps	Using lernajs (https://lerna.js.org/) tool it is possible to create separate versions and deploy based on those versions for each of the apps. Demo: https://youtu.be/7Lr8xYPKG5w?t=1387 https://www.npmjs.com/package/@lerna/version Basically lerna allows users to create a separate package.json for each project and increment the version independently for each project. another youtube video on lerna and versioning: https://www.youtube.com/watch?v=ZSdCNf1ncOE&t=435s I don’t know how exactly it is actually done but there are people on the interwebs, including in the talk above, who say lerna supports this feature.	Using multiple repositories/ applications (per product) allows independent / automatic versioning and deployment using the existing DevOps infrastructure. Having a single deploy/ version/release model provides clear understanding of the application status. When using GitFlow for code review, merging PR with features / fixes from different applications can affect the speed of release, and can cause	

		inconsistency updates on the same repository. Testing effort decreases with independent release per application (versions) since integration and smoke tests can be performed independently on code changes.
State across applications since we have multiple apps: • lets take pricing and auto-ordering ◦ if you have access to both you wont be able to keep the state of one application ◦ doing some confing on ao and switch to pricing app, with the current solution you would lose the state of ao because you are reloading the applications	"doing some config on ao and switch to pricing app" - persisting unsaved changes in one app while changing to another app is a **feature** that would need to handle on the frontend regardless of architecture - this would be handled by either saving constantly to the backend via API calls or by saving changes to localStorage Actually this raises more questions: - What are the kind of changes that are supposed to be done in one app, abandoned, and then picked up in a different app? - What is an actual use case?	The end user can use multiple products from the same interface, benefiting from seamless integration of functionality between them. Active application can interact with background applications or main application using events, hence increasing the user experience and separating. (Ex: an application can handle it’s own notifications, dispatching events to the main app).

Semantic versioning for applications

regarding the code base since having few products is much easier for a team to have a separate git repo, now its not the case but this might change in the feature and speed up the development process

Using lernajs (https://lerna.js.org/) it is possible to create separate versions and deploy based on those versions for each of the apps. Demo: https://youtu.be/7Lr8xYPKG5w?t=1387

In the docs: https://lerna.js.org/#command-publish

another youtube video on lerna and versioning: https://www.youtube.com/watch?v=ZSdCNf1ncOE&t=435s

Semantic versioning can be applied independently and can reuse the existing versioning strategy. The automatic versioning reflects the status of the release.

Research independent versioning using L

Versioning

it's easier to handle versioning with separate repos

- how would you handle the mono repo approach when deploying different projects?
  - mono-repo would produce artifacts in each of the applications

Using lernajs (https://lerna.js.org/) tool it is possible to create separate versions and deploy based on those versions for each of the apps. Demo: https://youtu.be/7Lr8xYPKG5w?t=1387

In the docs: https://lerna.js.org/#command-publish

another youtube video on lerna and versioning: https://www.youtube.com/watch?v=ZSdCNf1ncOE&t=435s

Having separate versions for applications allows multiple distributions options and feature enablement using different recipes. (static/dynamic JSON files)

Ex: Pricing New Version (2.0.0) + Auto-Ordering (1.0.0)

Distributions with different configuration can be roll-out to specific customers and enable AB testing without code updates.

Example of application recipes:

```
{
    "name" : "ECOMM UI",
    "url" : "https://app.cy
    "version" : "1.0.0",
    "config" : "https://con

    "apps" : [
      {
        "name" : "Pricing",
        "version" : "0.2.0"
        "config" : "https:/
      },

      {
        "name" : "Auto-Orde
        "version" : "0.5.0"
        "config" : "https:/
      }
    ]
}
```

Adding and building a new app version is

Each application is

Dynamic application registration how can we do dynamic loading of applications? • when adding a new app you have to update the main application which will redirect the user to v2	covered in the Versioning part of the document, down below. Dynamic urls can be generated and stored in users configuration in the API. The frontend would generate the top level menu in each of the apps using the config returned by the API.	independent and relies on the main app functionality. The main application can load different applications using an recipe and provide new functionality distribution without code changes.	Example of mixed application: • main-app • user settings • dashboards • pricing • sourcing • pse-services • data-distribution
Libraries and artifacts maintenance sharing the same modules through out all apps • we finish pricing and moving on to ao, and want to use new versions of the libraries ◦ how do you handle different library versions for different applications?	Since we have the same technology stack across all apps, upgrading a library would benefit all apps because we'd have to do it in only one place. For example, having pricing and ao in different repos would mean upgrading common libraries separately for each one. Lernajs has provisions for this use case. I don't know how exactly it is actually done but there are people on the interwebs, including in the talk above, who say lerna supports this feature.	The technology stack will be upgraded in time. This operation requires uniform upgrades over the whole set of application, increasing delivery time. Using multiple applications written with different frameworks or versions allows independent maintenance of each application. This lead to faster upgrades / features delivery. For common components a library can be defined and shared across applications.	

		With a shared component library, you could eliminate many of the user experience inconsistency problems.
Performance		If we are disciplined in not bloating our micro frontends with unnecessary dependencies, or if we know that users generally stick to just one or two pages within the application, we may well achieve a net gain in performance terms, even with duplicated dependencies.

Next steps:

- discuss the contents of this document and create tickets for research

Frequently Asked Questions:

- can we handle separate deploys?
    - the storybook project works the same way, by building a standalone app. since it is delivered with scripts handling the build separately from our projects, it must be possible to build each of our projects separately
- multi repo maintenance overhead?
    - should not be a problem, at this stage at least. Since the frontend team is small, chances of conflicts are minimal for now

- will a monorepo and deploying all apps at the same time have a bad effect on:
    - software delivery speed?
    - software maintenance time?

Micro-frontends vs. traditional SPAs

A **Micro-frontends** architecture means that multiple separate web apps coexist on the same page.

A **Traditional SPA** means that we have only one app per page.

At this point, wrapping up what we have with single-spa would not be preferred at this stage since:

- we are starting a project from scratch (instead of a re-write legacy code)
- each application has it's own page
- our technology stack is consistent across all apps (react + react-router)
- the design of the frontend is not granular (at least for now)
- single-spa adds an unnecessary layer of complexity

File structure

`src` folder houses all the source files which will be used by webpack to create the resulting html and javascript files.

Webpack will take each entry point in each of the `app1`, `app2` and `app3` folders and create an html file and **separate** javascript bundles for the code written by the developer and the packages imported from node_modules.

Right now this is done all in one, meaning that each project is built every time a change is made in only one project. This can be changed if needed in order to build each project on it's own just like we are doing now with the Storybook app.

When using the browser to navigate to

- `https://www.cst-apps.com/app1` or
- `https://www.cst-apps.com/app2` or
- `https://www.cst-apps.com/app3`

the browser will automatically load the `index.html` file, which, in turn, will automatically load the javascript files in that folder.

The internal routing of each of the apps would be handled by react-router:

- `https://www.cst-apps.com/app1/feature-1`
- `https://www.cst-apps.com/app1/feature-2`
- `https://www.cst-apps.com/app1/feature-2/sub-feature-1`
- `https://www.cst-apps.com/app1/feature-2/sub-feature-2`
- `https://www.cst-apps.com/app2/feature-1`
- `https://www.cst-apps.com/app2/feature-2`
- `https://www.cst-apps.com/app2/feature-2/sub-feature-1`
- `https://www.cst-apps.com/app2/feature-2/sub-feature-2`
- `https://www.cst-apps.com/app3/feature-1`
- `https://www.cst-apps.com/app3/feature-2`
- `https://www.cst-apps.com/app3/feature-2/sub-feature-1`
- `https://www.cst-apps.com/app3/feature-2/sub-feature-2`

Authentication and authorization

Authentication is **done in one place** while authorization is **done in each of the apps** by using an external provider like Amazon Amplify.

In order to create users and authenticate them we add a new application to the stack.

The same webpack file will handle the build of the frontend `authentication` which will result in an app looking the same way as the other apps:

This means that the `authentication` app will take care of all the routes necessary for authentication:

- `https://www.cst-apps.com/authentication/register`
- `https://www.cst-apps.com/authentication/login`
- `https://www.cst-apps.com/authentication/forgot-password`
- `https://www.cst-apps.com/authentication/recover-password`

The authorization part will be done in each of the apps separately, thus giving the developer the tools to restrict a users access to certain apps.

Let’s say that a user has access to `/app1` but not to `/app2`. Trying to access the url `https://cst-apps.com/app2` would result in a redirect to the login page:

Please note that all we do in a child app is `AUTHORIZATION` by using an external library:

Downsides:

- will not work with different subdomains for each app:
    - for example, using `app1.cst-apps.com` and `app2.cst-apps.com` will force the user to login each time he/she decides to switch between the 2 apps. The reason for this is that Amplify uses `` `localStorage `` (https://github.com/aws-amplify/amplify-js/blob/a047ce73/packages/auth/src/Auth.ts#L162) to keep all the necessary information about the user's session, information that is domain specific.

This would not be a problem if the aforementioned architecture would be used since there is no subdomain switching involved when navigating from one app to the other:

- `https://www.cst-apps.com/app1`
- `https://www.cst-apps.com/app2`
- `https://www.cst-apps.com/app3`

Versioning

This is not a discussion about code versioning which will follow the `semantic versioning` approach.

So let's say we want to slowly retire `app1` and replace it with a new version.

All we have to do is add a new app folder to the structure we already have in place and re-direct users towards the appropriate app:

Check the user groups:

And see if the user has the appropriate authorization to access the app:

Sharing resources between the old and the new versions of the app would be done through a `shared-resources` folder which would not be built into a stand-alone app.

Custom apps

Creating a custom app that has extra features or less features would be done in the same way, by creating a new app, importing what is needed and restricting access to it from the cognito user pool.

Mono-repo vs multi-repo

Marcel Cutts talks about the advantages of a mono-repo at **ReactNext 2018**: https://www.youtube.com/watch?v=rdeBtjBNcDI

Mattias Petter Johansson talks about monorepos at **Spotify**:
https://youtu.be/0_qhdOeMuhk?t=174

> At Spotify, when I joined, the company was very micro-service oriented, the backend had hundreds of micro-services all in their sepparate repos divided over teams.
>
> The front-end was also inspired by that. It had micro-frontends and everything was in its own repo. Spotify had a shared NPM before anybody had it.
>
> I was a big beliver in that, in having a thousand of small repos.
>
> But after a while I started to see the downside of that.
>
> Because there's a lot of contact area you get when you are spreading your code out like that.
>
> When we moved to a single repo for the desktop client and moved to one build system and sharing all of that infrastructure and also being able to make changes in three different modules and the main code using the modules at the same time in one pull request that was just so awesome.
> In improved productivity so much.
>
> And I am nowadays very skeptical before pushing things out in a sepparate repo, because I have understood that the cost of pushing things out in a separate repositories and separate modules completely autonomous, the cost of that is pretty high.

Aimee Lucido talks about how **Uber** went from a mono-repo to a multi-repo and then back again to a mono-repo. She talks about mono-repos in an Android development environment but it's still pretty interesting: https://www.youtube.com/watch?v=lV8-1S28ycM.

Frontend architecture for the CST suite

- Marketing website (cyber-solutions.com)
- Authentication app
- Standalone apps (Main Dashboard, Pricing, etc.)

Marketing app technology stack

Some of the criteria this project should meet:

- SEO friendly
- internationalisation (i18n)
- Analytics

- performance
    - lighthouse in the build phase

Tech Stack

One of the solutions that works great with SEO is serving static HTML files. In order to achieve this and maintain a decent project structure is to use a static site generator:

- HUGO
    - language: go
    - url: https://gohugo.io
- gatsby
    - language: javascript
    - url: https://www.gatsbyjs.org
- jekyll:
    - language: ruby on rails
    - url: https://jekyllrb.com

Questions:

- CMS based content?

Authentication app

The authentication part could be

- abstracted into a standalone app.
- be part of the dashboard app.

Tech stack

- small react app

Standalone apps

The Pricing app is part of several standalone apps, including the main Dashboard, which could share the same technology stacks.

Tech stack

- front-end framework
  - react
- state management
  - redux + re-ducks
- type checking
  - flow
- styling solution
  - styled-components
- testing (unit and component)
  - jest
  - lighthouse
  - linting
  - saucelabs
- performance
  - time to first interaction
  - automated performance tests
- translations
  - i18n npm package
- living style guide
  - eg storybook, react cosmos

Each of these projects would be built to produce one or more javascript files (or chunks) which can be served via own server or CDN.

Cross project discipline

How it all comes together

A server could take care of each of the respective routes:

- /dashboard
- /dashboard/pricing

- /dashboard/forcasting
- /dashboard/logistics

The technology used for routing the main pages is irrelevant (could be done in either server side languages: python, node, php, rails etc.). After one of the main routes is served by the server, the SPA (single page application) takes care of the rest of the navigation:

- main route:
  - /dashboard/pricing
- inner routes:
  - /dashboard/pricing/page-1
  - /dashboard/pricing/page-2
  - /dashboard/pricing/nested/route/to/hell/and/back