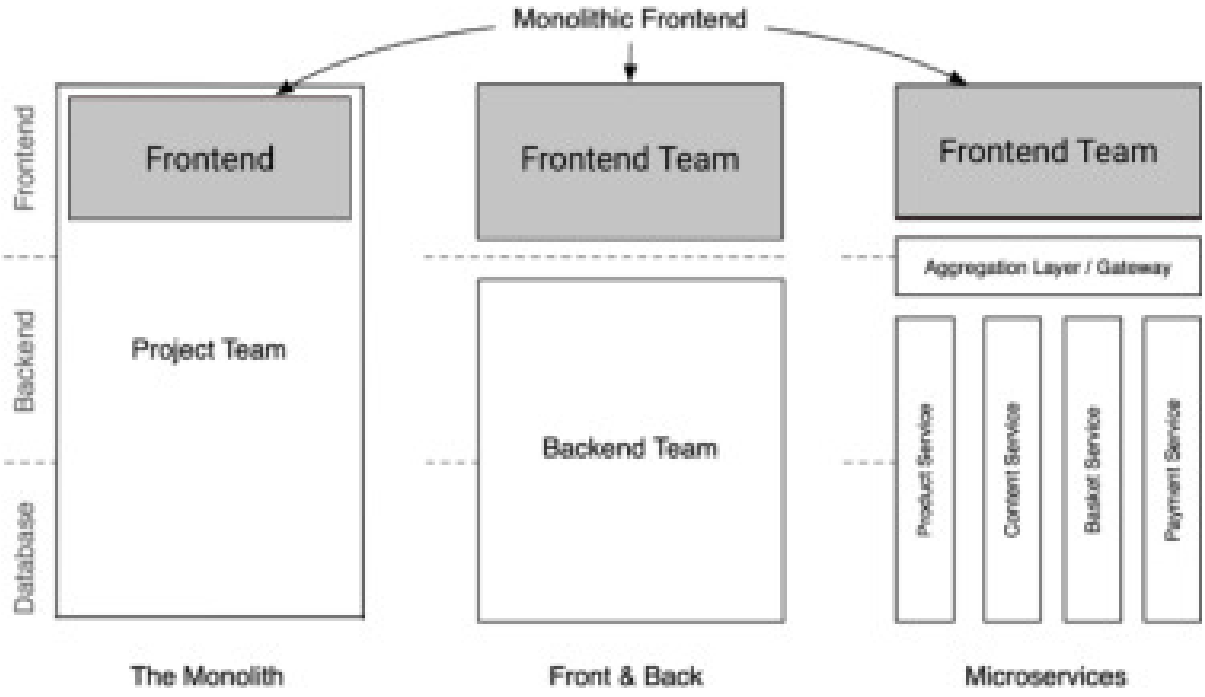
Module Federation

Index

Monolithic Frontend Architecture

There will be a single repository for the whole app (in our case we can consider ARC) and the every single module will be contained inside one big app. For example : Payroll, Accounting, Parts, etc would not exist as separate react (just an example) apps but as sub modules to the single react app. Only this app will be build and will be served to browser. Monoliths were the industry standard and the way websites have been built for many years. Monolith is a viable option for smaller apps, but as the app grows bigger over time problems associated with this architecture will start to become apparent such as

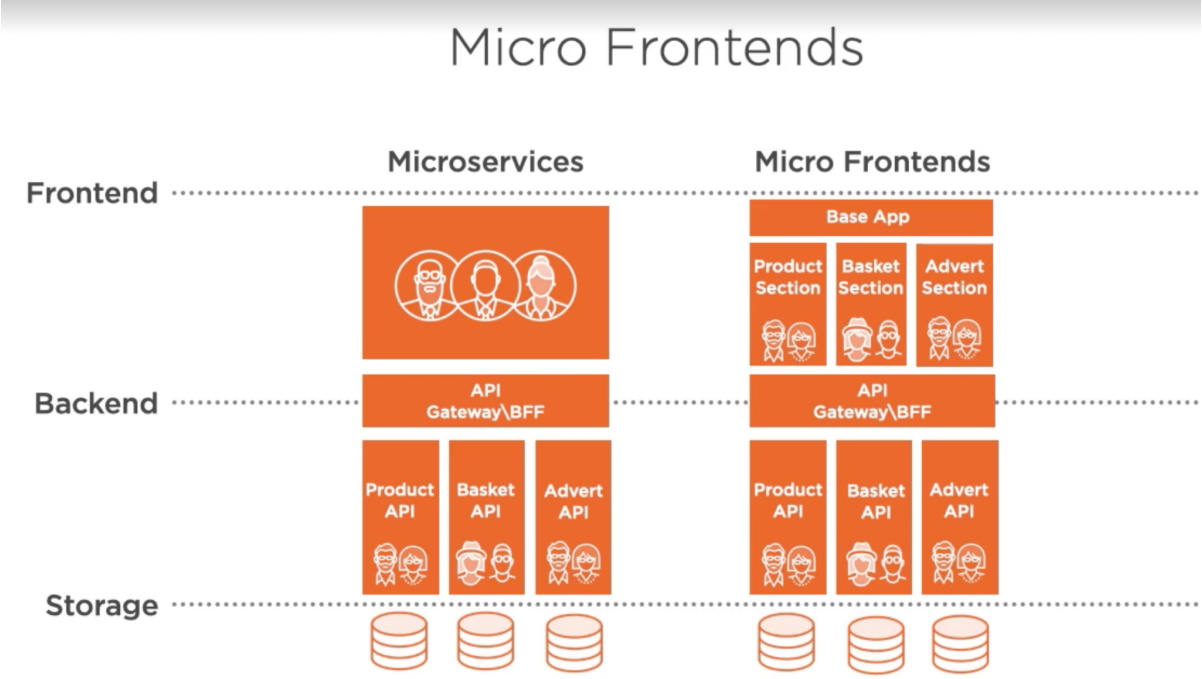
- High build time
- Big chunk size
- Harder to maintain as everybody is working on the same app, so a change in one module can affect other module as well.



Micro-Frontends

The Micro Frontend Philosophy

A micro-frontend is a frontend app which can be developed, tested and deployed separately. This is analogous to micro-services that we have in the backend. So, a micro-frontend can be a module or a fragment of module, basically any component or group of components which can be build and developed separately but can be used by other apps or teams somewhere in the module or page they are developing.



Current Architecture Of ARC and What Are The Challenges We Face

The repository being used for ARC is `tekion-web` . Its a monorepo containing multiple apps such as `tekion-payroll`, `tekion-accounting`, etc. Every app has its own node servers and is served based on the path provided in url. This is

being controlled by Edge and ELB where we provide routing for each apps. So our earlier implementation has routing based micro frontends. This architecture has its pros and cons :

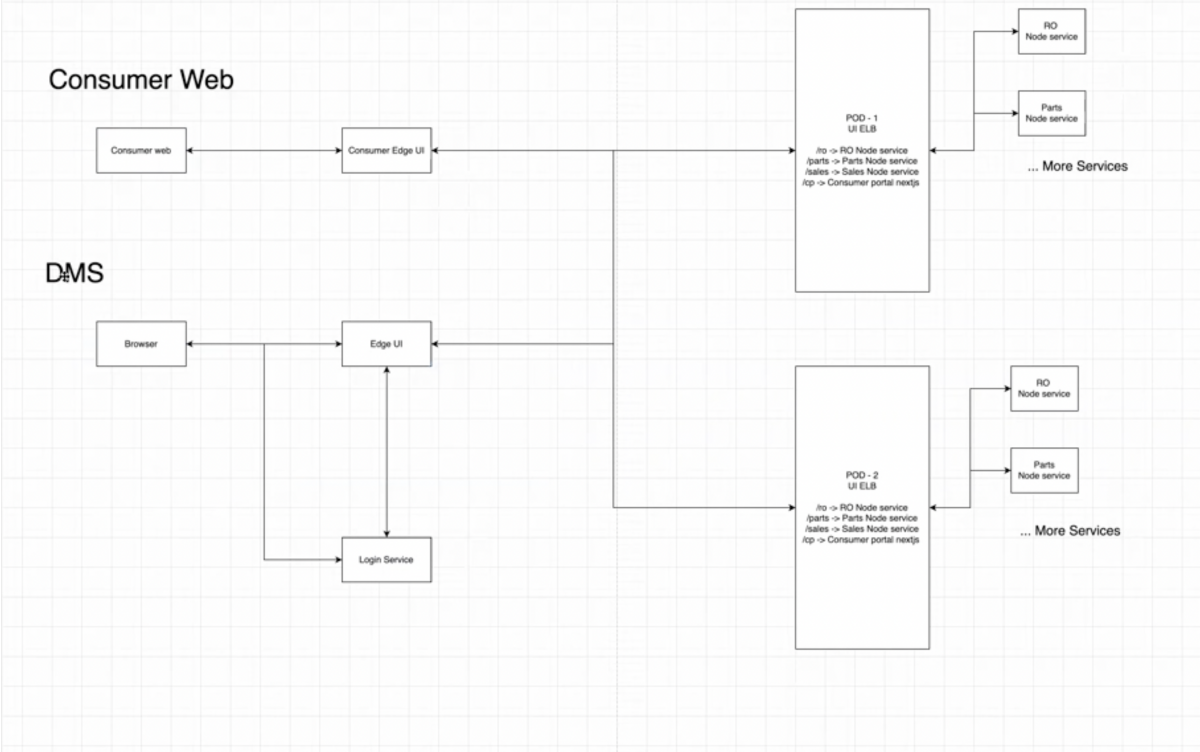
- **Pros**
  - Every module is independent from other modules, hence change in one module would not affect or break other modules.
  - Easier to maintain each module
  - Release cycle can be flexible

- **Cons**
  - Separate servers required for every module
  - Too much independence of one module can cause isolation from other modules which results in difficulty to follow common standards over the whole app.
  - As individual modules grow bigger they start to become a kind of monolith themselves, which brings us back to the problem of big chunk size.

Every app mounts its own app skeleton and other components. Whenever we switch from one app to other app, the browser will fetch the resources for that app and it will replace the earlier app. Which results in the app skeleton being mounted again and all the common apis (for eg: permissions) will be called again. Due to this our app seems very slow while switching. The transition from one app to other app almost feels like a hard refresh, which is not a good experience for users.

Each app is becoming a monolith in itself which results in more time to fetch and load its resources on the browser. The problem is with the code being fetched for the unwanted screens. For example, if the user visits `payroll/payroll-setup` , the resources that will be fetched will have code for

payroll-processing, tax forms, and pay check register as well, which are not wanted for the current scenario.



Module Federation By Webpack

Module Federation was introduced in Webpack 5 in 2020. Its a plugin which can provide support for developers to adapt micro-frontend architecture for their app. The definition provided by webpack is as follows :

> *Multiple separate builds should form a single application. These separate builds act like containers and can expose and consume code between builds, creating a single, unified application.*

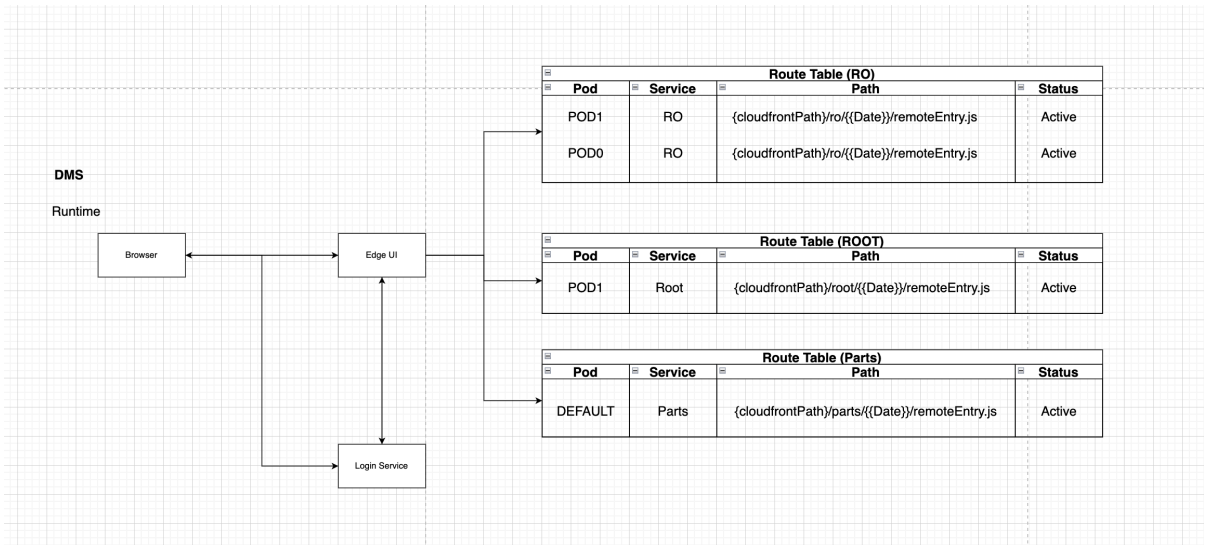
> *This is often known as Micro-Frontends, but is not limited to that.*

Module Federation allows us to dynamically import code at runtime from another app which is build separately. We can expose the components of an app which we are allowing to be imported by other apps. The app importing those components will have remotes which specifies the components it needs to import.

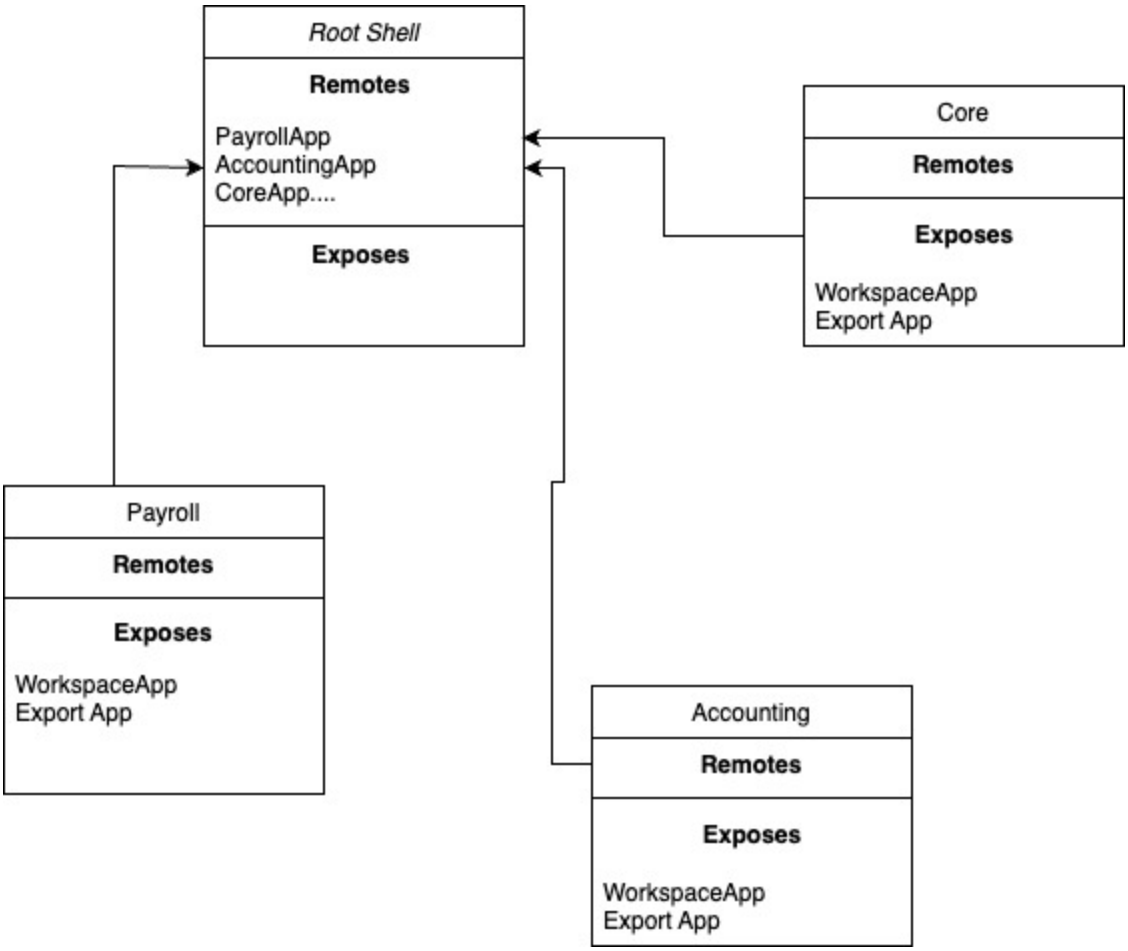
It boils down to two roles i) Host and ii) Remote. Host is a webpack build which is initialised first during the page load. Remote is a webpack build that is being consumed by the host.

Module Federation in ARC

As explained above, there will be a Host and Remote. Now for `tekion-web`, we have a root-shell app which will be loaded initially and will be hosting other remotes. In root-shell app there will be the app skeleton as the main component and multiple other HOCs and contexts. It will also be responsible for fetching some common apis and shared resources. After its initialised, it will fetch the remotes based on the path.



Every federated apps will expose their workspace app, root app and export app. This remotes will be imported by our root shell app based on the route. For let’s say Payroll app, whenever the url has payroll appended, the remoteEntry file will be fetched. After that the actual payroll app required will be fetched and loaded.



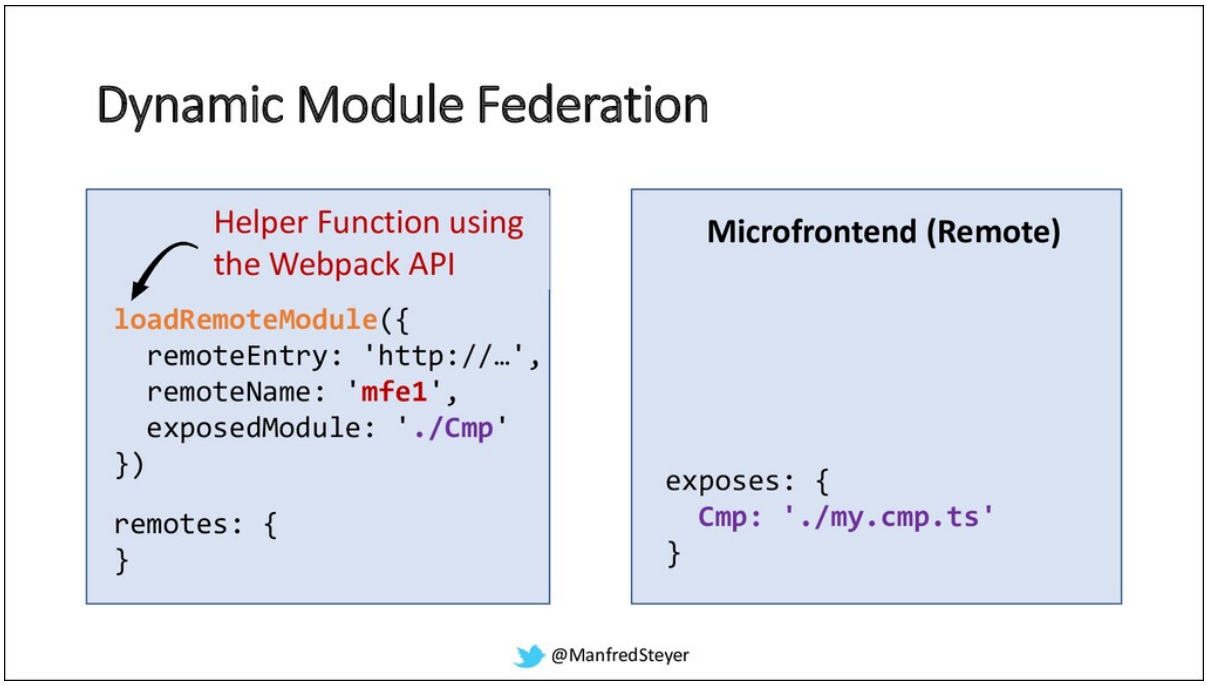
Shell (Host)

```
import('mfe1/Cmp')

// Maps Urls in
// webpack config
remotes: {
  mfe1: "mfe1"
}
```

Microfrontend (Remote)

```
// Expose files in
// webpack config
exposes: {
  Cmp: './my.cmp.ts'
}
```



Payroll Implementation

What Are Required Changes In Frontend And What Payroll Needs To Change ?

Now we have a root shell app present in `tekion-web`, with the name being `tekion-root-shell`. This app contains our app skeleton and other common resources. Payroll needs to expose the remotes which are going to be imported by the root shell dynamically. The exposed components will not contain app skeleton, some of the common wrappers and containers. This means now onwards if a common wrapper is required across all the apps, we can simply add them inside root shell rather than adding it inside all the apps.

Payroll also needs to change the routing as well as the way reducers were exposed. Earlier, history used to be initiated in payroll with base path being /payroll, so all our routers only had to take care of the path being appended to payroll. Now we don't have that privilege, as the root shell is being loaded first it will initiate the history with no base path, and the same history instance is going to be used by every remote. Hence, we have add /payroll at the start of every path if payroll is being used as remote.

Payroll used to provide the redux store to its sub components. Reducers earlier were being directly passed to the redux store. Now, the store will also be provided by the root shell. Root shell will not be aware about the reducers of the remotes. Hence, we cant directly pass the payroll specific reducers inside the store. Payroll needs to inject the reducers asynchronously inside the store. Here is a catch, earlier when payroll used to provide the store the reducers being added to the store will only contain the ones which are being used by Payroll. So, there was no problem in giving the reducers general names like appReducer, metadataReducer, etc. If we switch from Payroll to Core app, the store initiated by payroll will be removed completely along with its reducers. Hence, Core can create a new store and can have the same names as payroll for some reducers. After module federation if we switch from one app to other, then the store will remain as it is with its reducers being intact. Hence, we cannot expose our reducers globally to the store without specifying it to be linked with Payroll. So, now onwards all payroll reducers will be combined together and then will be passed to store.

What are the Infra changes required ?

Earlier whenever the url has payroll in it, edge used to direct the request to the specific node server for payroll based on the pod. Now we will only require one server per pod which is going to provide the root shell. So, we would need to route to the payroll server. Hence, we can remove that routing from the lambda. Hence,

this helps reducing the number of servers significantly if all the apps are federated.

Problems We Faced And Why ?

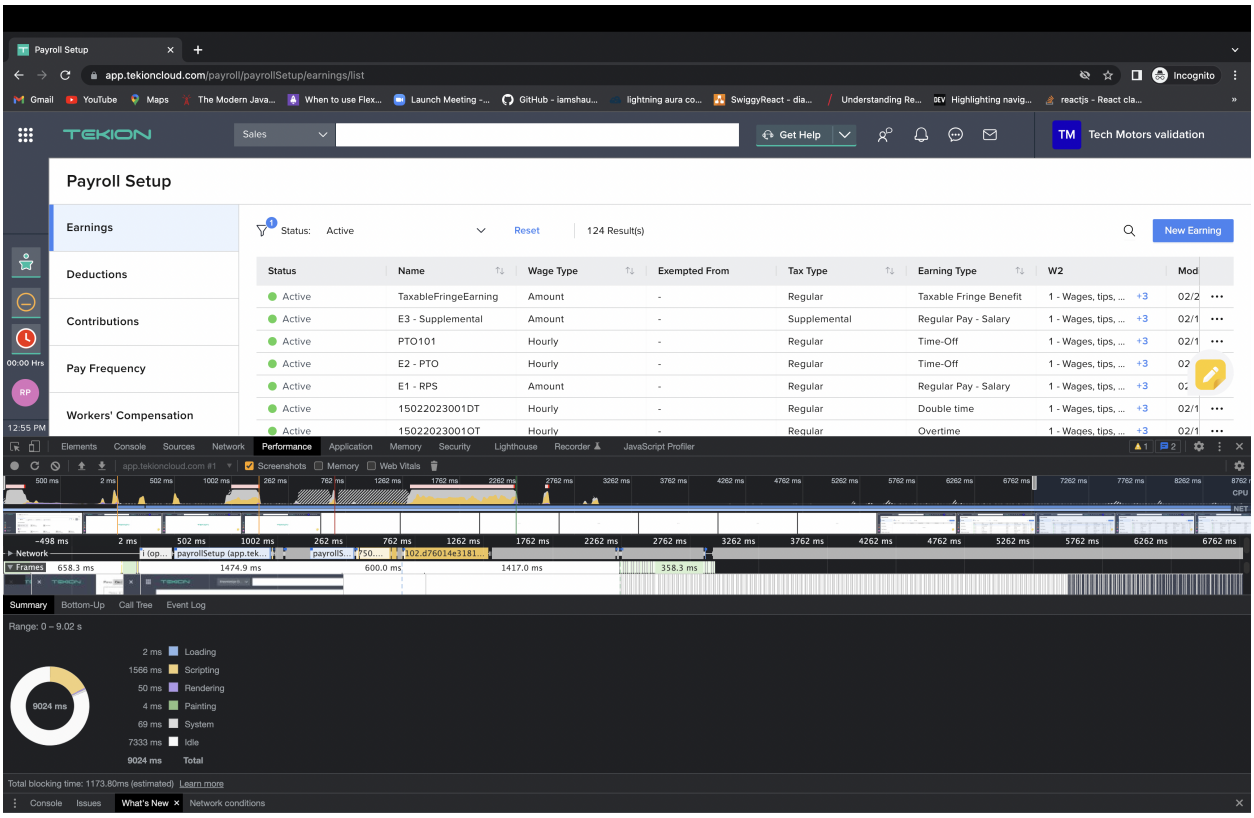
On the first attempt to implement module federation for Payroll, there were multiple problems due to which contexts, css, export service, etc. was breaking. For the most of the problems, the solution was already implemented before our 2nd attempt or in the 2nd attempt. Still the CSS problem was persisting. This issue was due to multiple factors. Our global css utilities were being overrode by the css of the common components (Forms, etc) whenever they collided (flex row vs flex column).

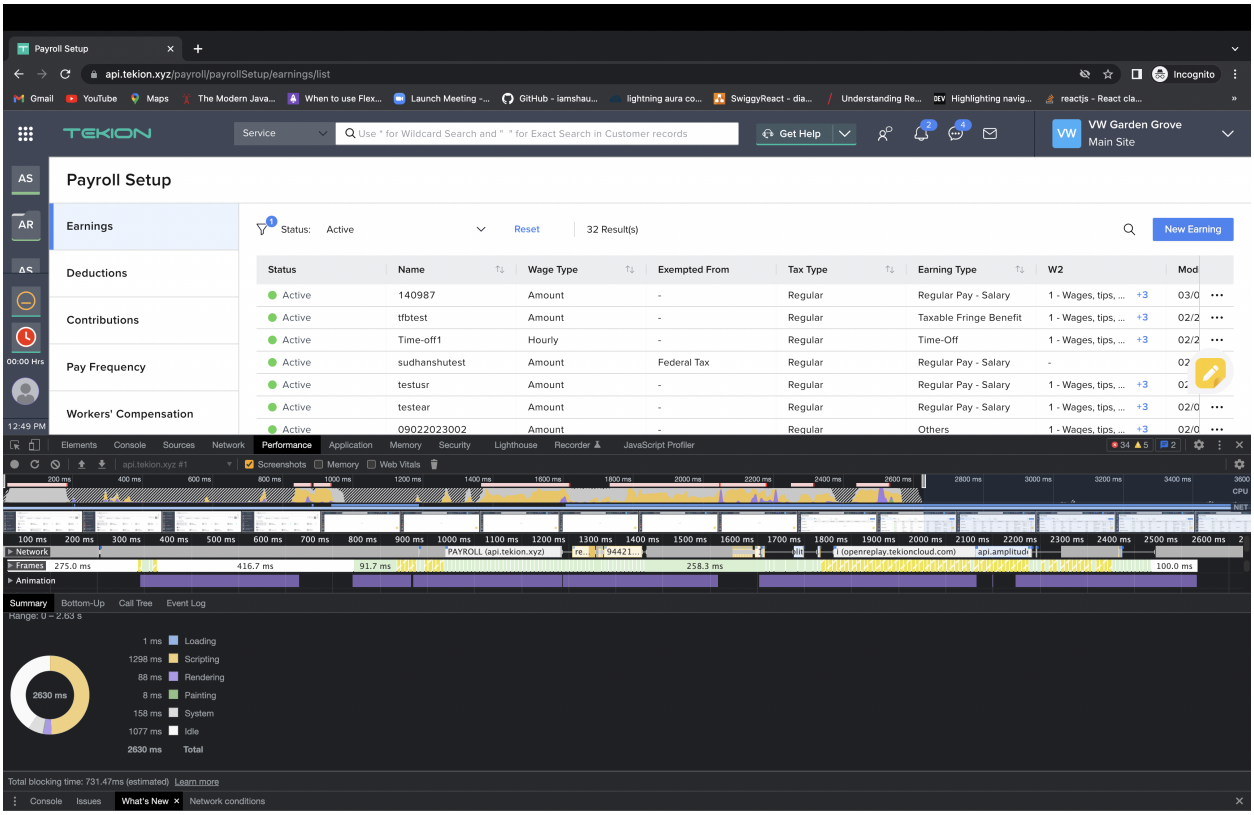
When module federation was not present, we had passed this global css utility classnames to override the default css offered by the component. It was working well and good for payroll. Now, after module federation the hierarchy was reversed. The global utils were not able to override the component's default css. Most apparent collision was between flex-row and flex-column. Almost all of our forms that use global classes were breaking. Primary reason was due the fact that global utility classes were not marked `!important` in our codebase. Due to this the specificity of the global classes and the component classes were same. So, now it all depends on which css class is loaded first. Whichever is loaded later will be applied. As we are loading the payroll app on the later stage, CSS required by payroll components would be loaded later. Hence, the reversal of the hierarchy. We had to fix wherever there was case of collision with custom utilities or by using bootstrap utility classes with `!important`.

Advantages and Metrics of Improvement

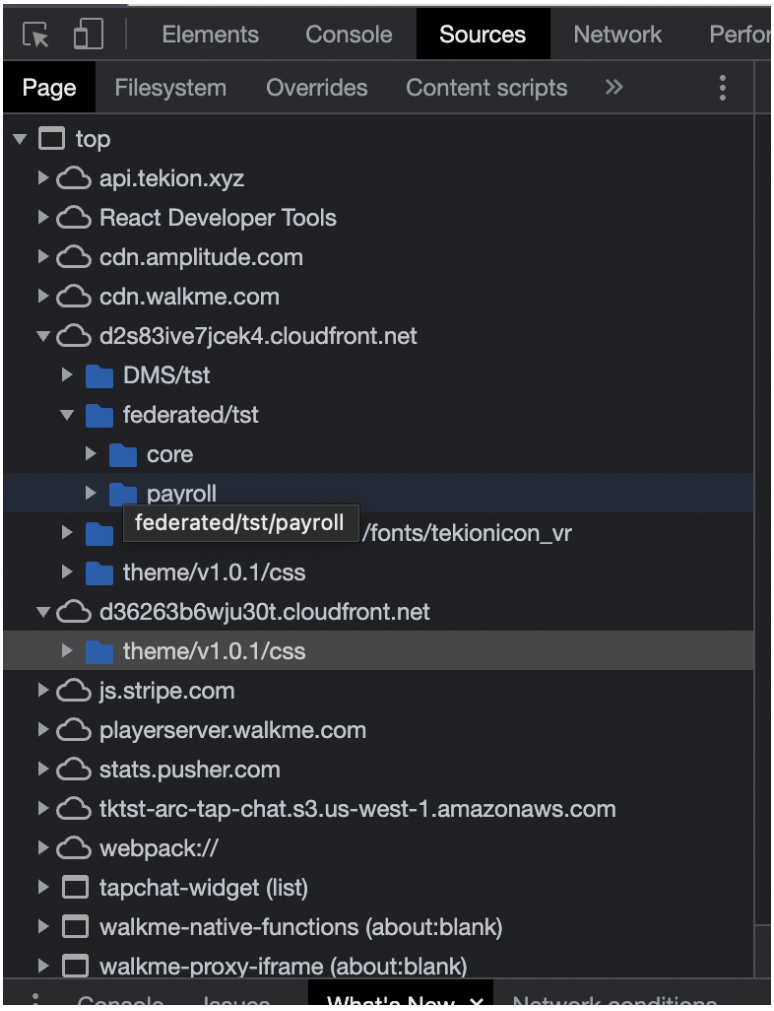
The advantages by shifting to module federation are very noticeable by any user. Some of the salient are mentioned below :

- The most apparent improvement can be observed while switching between different apps (payroll → core or vice versa). It takes significantly less amount of time to load the new app. The screenshots attached below describes the performance while switching from /home to /payroll. In Prod environment we have original implementation and in Tst we have module federation implemented.





- There will be no need to cherry pick changes for appSkeleton in every app. There will be only one source to load appSkeleton.
- We can eventually shut the servers dedicated to the remote apps.
- We have support to share node_modules as well. This way we will only need to fetch any library once and then other app can reuse that. Also the remote entries and chunks are cached so we don't have to fetch the app specific chunks again as well.



Road Ahead

Module Federation has given us advantages by importing the necessary remotes in runtime. Currently the remotes have all the code for that app/module. Let's say for Payroll, if its being fetched it will contain the code of the whole app. Now the user only needs to visit payroll setup, then why are we fetching the code of the other screens ? We can optimise on this further. Webpack already has provided the feature to dynamically import the components/modules in runtime, termed as **Code Splitting**. The definition as per Webpack goes :

> *Code Splitting allows you to split the code into various bundles which can then be loaded on demand or in parallel. It can be used to achieve smaller bundles and control resource load prioritization which, if used correctly, can have a major impact on load time.*

This way we can fetch the required whenever we actually need them. Thus making the initial time to load faster.

Check if you had no employees and paid no wages this quarter