App, Code & Technology Documentation

Sisense Compose SDK Hackathon

By Nhat K. Nguyen @resolve

Table of Contents

Introduction 2
Use case / Scenario 2
Setting up Sisense for the app 3
Charts 5
measureFactory 5
filterFactory 6
Filter tile components 7
Working with custom data 8

Introduction

This document is made as the requirement for the submission for [Sisense Compose SDK Hackathon](). It is suggested that the documentation should include explicitly which, how and where the **Sisense Compose SDK** features and functionality are being used within the web application that used for submission.

Therefore, the sections below will present the app that I built and submitted to the hackathon, its notable features, and how the **Compose SDK** is utilized in the codebase. The app uses **Vite + React + TypeScript** as the base as well as the required packages such as **@sisense/sdk-ui**, **@sisense/sdk-data**, etc. For installation & deployment, please check out the **README.md** file.

Use case / Scenario

The app that I made is named **CompetiSense** *(get it?)* – it has the feature to show overviews and insights of different workspaces using Sisense Compose SDK charts with some touch to different types of filters and style customization.

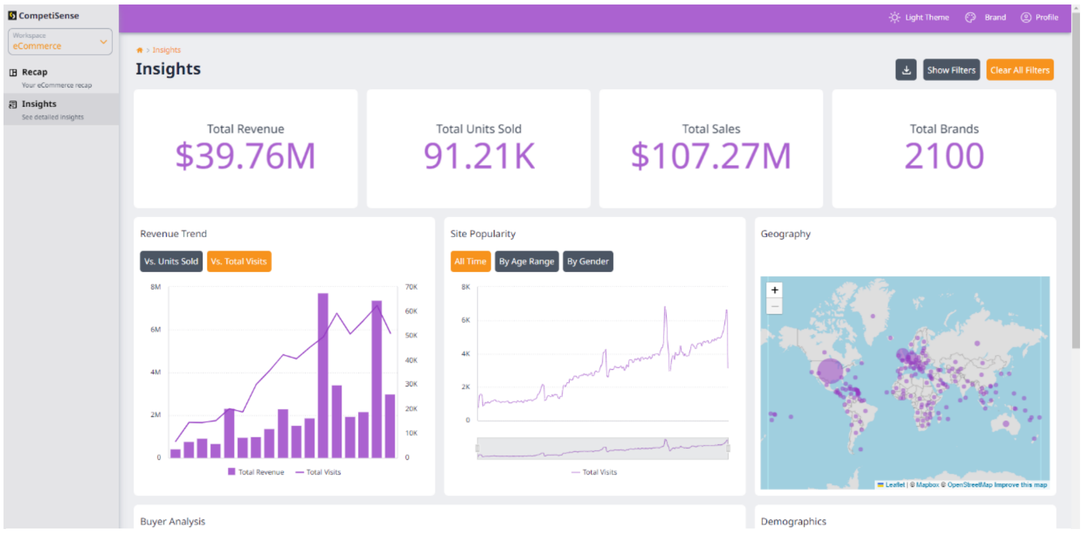


Figure 1: App Screenshot

There are 2 available workspaces that you can toggle in the app: (1) **eCommerce** and (2) **Custom Retail**.

eCommerce

This workspace uses the data set **'Sample Ecommerce'** that stored in the Sisense environment.

Custom Retail

This workspace uses the custom data stored locally in the code which was inspired by Sisense Compose SDK React Demo in order to see if **@sisense/sdk-ui** can be used on external data. Click here to see the sample data from the codebase.

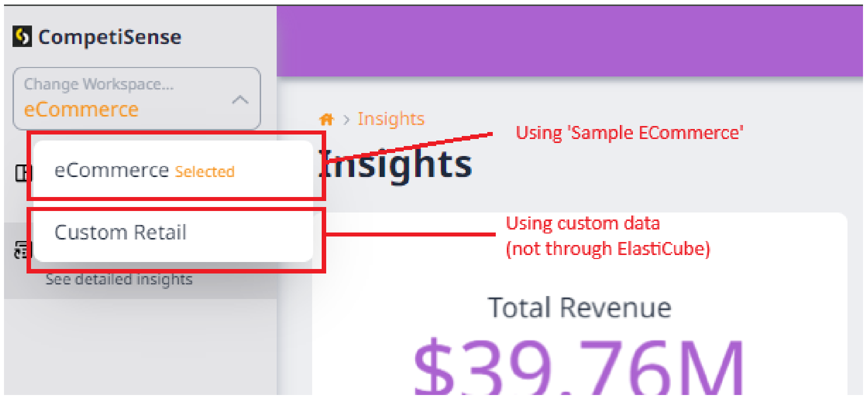
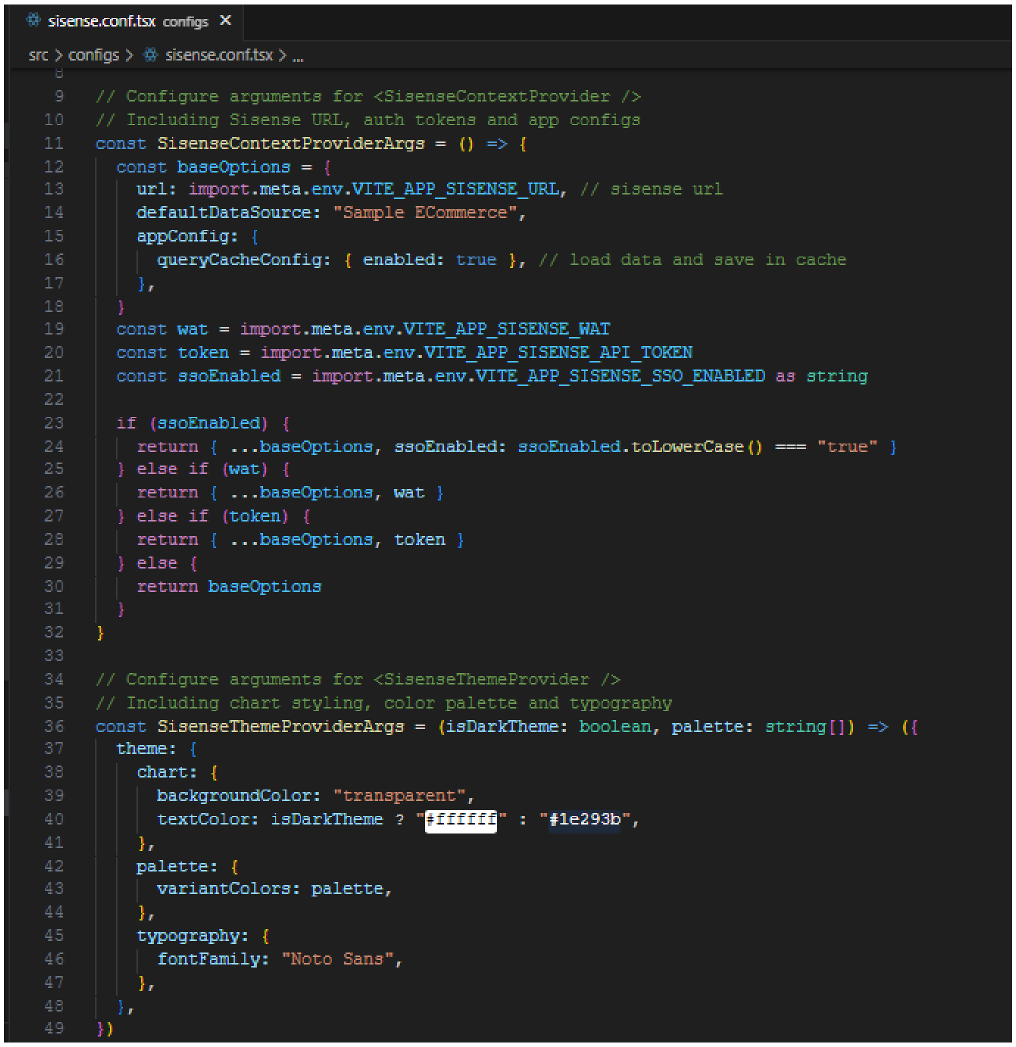


Figure 2: Available Workspaces

Setting up Sisense for the app

In order to set up Sisense on a React app, we need to create a **<SisenseProvider />** component which consists of **<SisenseContextProvider />** and **<SisenseThemeProvider />** from **@sisense/sdk-ui**. See the figures below, or click here to see the config file named **sisense.conf.tsx** on GitHub.

It is also worth noting that I store all the credentials for setting this up in the environment file **.env.local**. The example of how it should look like can be found here in the file **.env.local.example**.

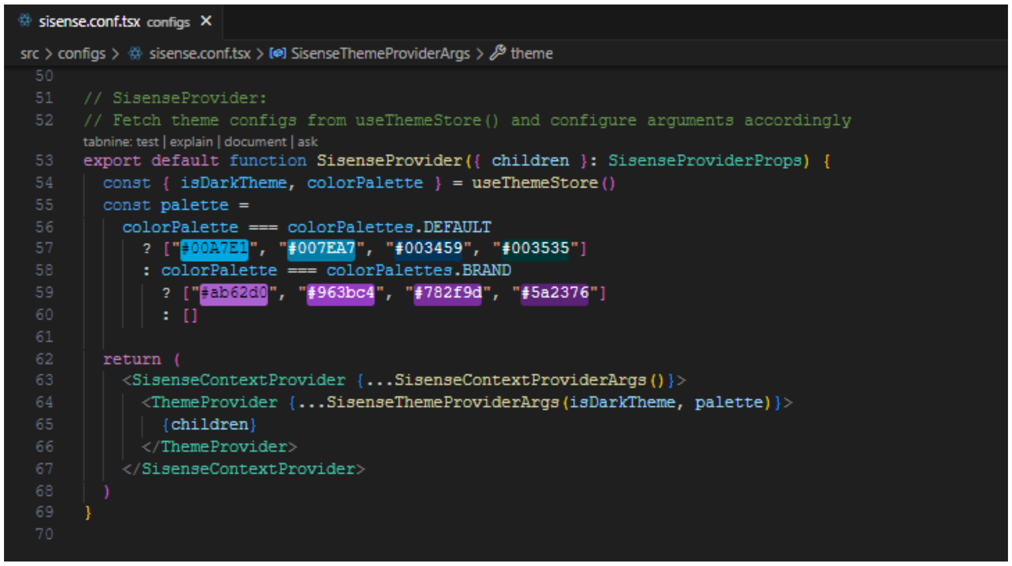


sisense.conf.tsx configs

src > configs > sisense.conf.tsx > ...

```tsx
// Configure arguments for <SisenseContextProvider />
// Including Sisense URL, auth tokens and app configs
const SisenseContextProviderArgs = () => {
  const baseOptions = {
    url: import.meta.env.VITE_APP_SISENSE_URL, // sisense url
    defaultDataSource: "Sample ECommerce",
    appConfig: {
      queryCacheConfig: { enabled: true }, // load data and save in cache
    },
  }
  const wat = import.meta.env.VITE_APP_SISENSE_WAT
  const token = import.meta.env.VITE_APP_SISENSE_API_TOKEN
  const ssoEnabled = import.meta.env.VITE_APP_SISENSE_SSO_ENABLED as string

  if (ssoEnabled) {
    return { ...baseOptions, ssoEnabled: ssoEnabled.toLowerCase() === "true" }
  } else if (wat) {
    return { ...baseOptions, wat }
  } else if (token) {
    return { ...baseOptions, token }
  } else {
    return baseOptions
  }
}

// Configure arguments for <SisenseThemeProvider />
// Including chart styling, color palette and typography
const SisenseThemeProviderArgs = (isDarkTheme: boolean, palette: string[]) => ({
  theme: {
    chart: {
      backgroundColor: "transparent",
      textColor: isDarkTheme ? "#ffffff" : "#1e293b",
    },
    palette: {
      variantColors: palette,
    },
    typography: {
      fontFamily: "Noto Sans",
    },
  },
})
```

Figure 3: Configure arguments for <SisenseProvider /> ***(source: src/configs/sisense.conf.tsx)***



sisense.conf.tsx configs

src > configs > sisense.conf.tsx > SisenseThemeProviderArgs > theme

```tsx

// SisenseProvider:
// Fetch theme configs from useThemeStore() and configure arguments accordingly
tabnine: test | explain | document | ask
export default function SisenseProvider({ children }: SisenseProviderProps) {
  const { isDarkTheme, colorPalette } = useThemeStore()
  const palette =
    colorPalette === colorPalettes.DEFAULT
      ? ["#00A7E1", "#007EA7", "#003459", "#003535"]
      : colorPalette === colorPalettes.BRAND
        ? ["#ab62d0", "#963bc4", "#782f9d", "#5a2376"]
        : []

  return (
    <SisenseContextProvider {...SisenseContextProviderArgs()}>
      <ThemeProvider {...SisenseThemeProviderArgs(isDarkTheme, palette)}>
        {children}
      </ThemeProvider>
    </SisenseContextProvider>
  )
}

```

Figure 4: Setting up <SisenseProvider /> ***(source: src/configs/sisense.conf.tsx)***

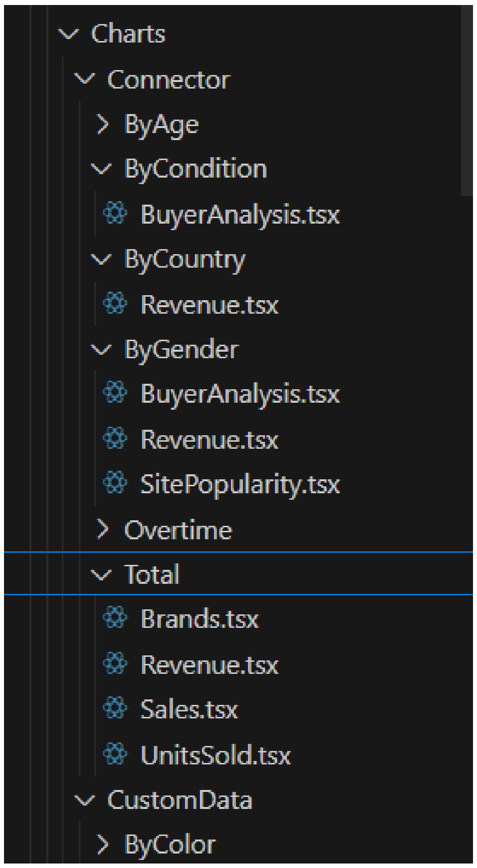
Charts

Charts are the main selling points of **Sisense Compose SDK**. With the exception of the <**ScattermapChart />** component in src/components/Charts/Connector/ByCountry/Revenue.tsx presenting the revenue broken by country and **<Table />** in src/components/Charts/CustomData/ByEmployee/Table.tsx presenting employee sales records, most of the charts used in the app are using the **<Chart />** component.

The figure on the right represents how I arrange the charts within the codebase. While similar charts can be used as one big **<Chart />** component with sophisticated prop values, with this set up you can easily find and reuse the charts on the dashboards based on what data they represent.

Different types and subtypes of **<Chart />** I used throughout the project:

- Line chart (line – line/spine)
- Bar chart (bar – bar/classic)
- Column chart (column – column/classic)
- Pie chart (pie – pie/classic & pie/donut)
- Scatter chart (scatter)
- Indicator chart (indicator – indicator/numeric)



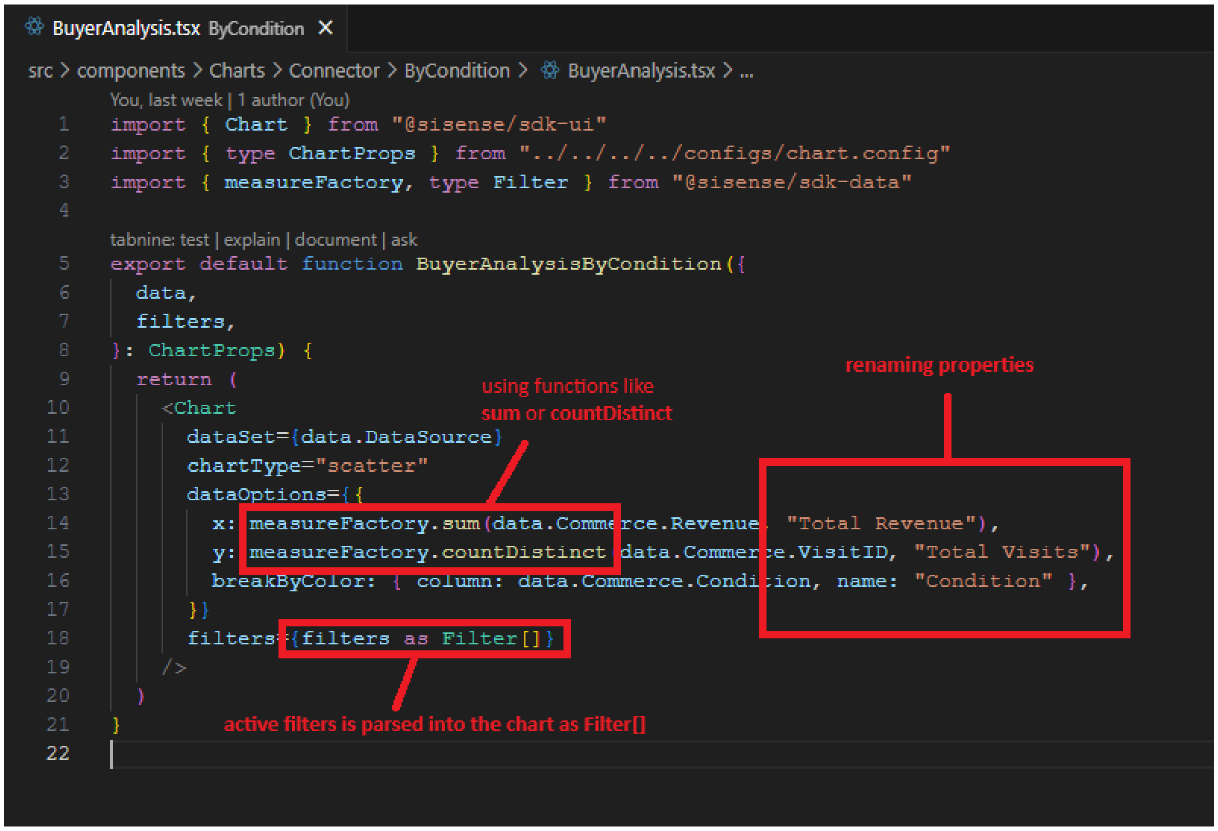
```
Charts
  Connector
    ByAge
    ByCondition
      BuyerAnalysis.tsx
    ByCountry
      Revenue.tsx
    ByGender
      BuyerAnalysis.tsx
      Revenue.tsx
      SitePopularity.tsx
    Overtime
    Total
      Brands.tsx
      Revenue.tsx
      Sales.tsx
      UnitsSold.tsx
  CustomData
    ByColor
```

Figure 5: Folder structure for charts

You can also find a set of options I particularly saved for certain charts' style options in the file src/configs/chart.config.ts from different number format configurations *(e.g. number vs. currency)* to certain style options I want to achieve on charts such as bar charts, indicator charts or pie charts.

measureFactory

measureFactory can only be used in the eCommerce workspace since **'Sample ECommerce'** dataset is stored in Sisense environment while the custom data isn't. For example, this file here is an example of how measureFactory is used in a **<Chart />** component:


BuyerAnalysis.tsx ByCondition
src > components > Charts > Connector > ByCondition > BuyerAnalysis.tsx > ...
You, last week | 1 author (You)
tabnine: test | explain | document | ask
using functions like sum or countDistinct
renaming properties
active filters is parsed into the chart as Filter[]

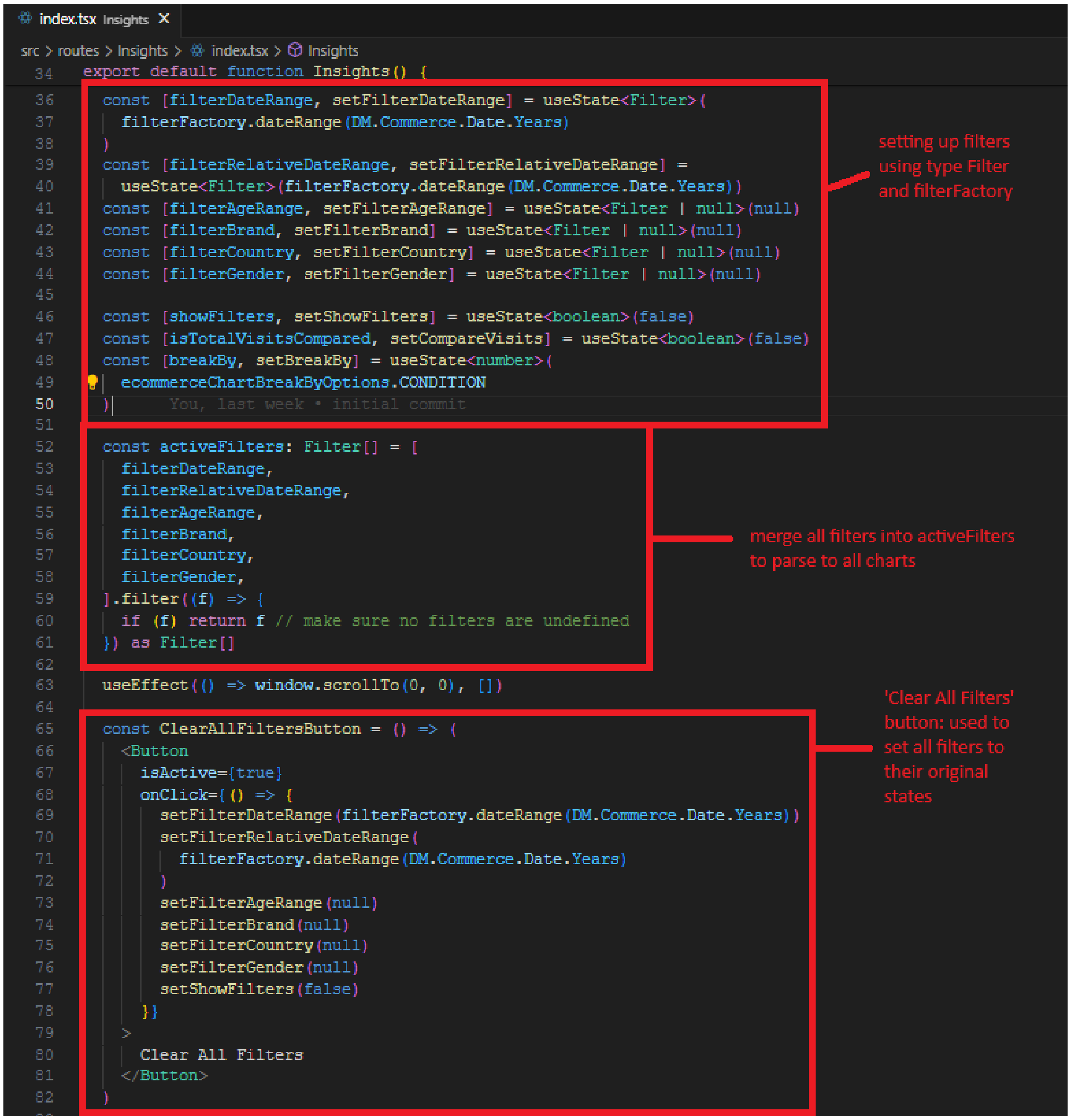
```tsx
import { Chart } from "@sisense/sdk-ui"
import { type ChartProps } from "../../../../../configs/chart.config"
import { measureFactory, type Filter } from "@sisense/sdk-data"

export default function BuyerAnalysisByCondition({
  data,
  filters,
}: ChartProps) {
  return (
    <Chart
      dataSet={data.DataSource}
      chartType="scatter"
      dataOptions={{
        x: measureFactory.sum(data.Commerce.Revenue, "Total Revenue"),
        y: measureFactory.countDistinct(data.Commerce.VisitID, "Total Visits"),
        breakByColor: { column: data.Commerce.Condition, name: "Condition" },
      }}
      filters={filters as Filter[]}
    />
  )
}
```

Figure 6: how measureFactory was used ***(source: src/components/Charts/Connector/ByCondition/BuyerAnalysis.tsx)***

filterFactory

Before the set of filters is parsed to each **<Chart />** *(see figure 6)* to perform dynamic cross-chart filtering, **filterFactory** is used to set up said filters. Much like **measureFactory**, it can only be used in a workspace that integrating the **'Sample Ecommerce'** data set.

The figure below from this file shows how the filter set is formed and used.


index.tsx Insights
src > routes > Insights > index.tsx > Insights
setting up filters using type Filter and filterFactory
merge all filters into activeFilters to parse to all charts
'Clear All Filters' button: used to set all filters to their original states

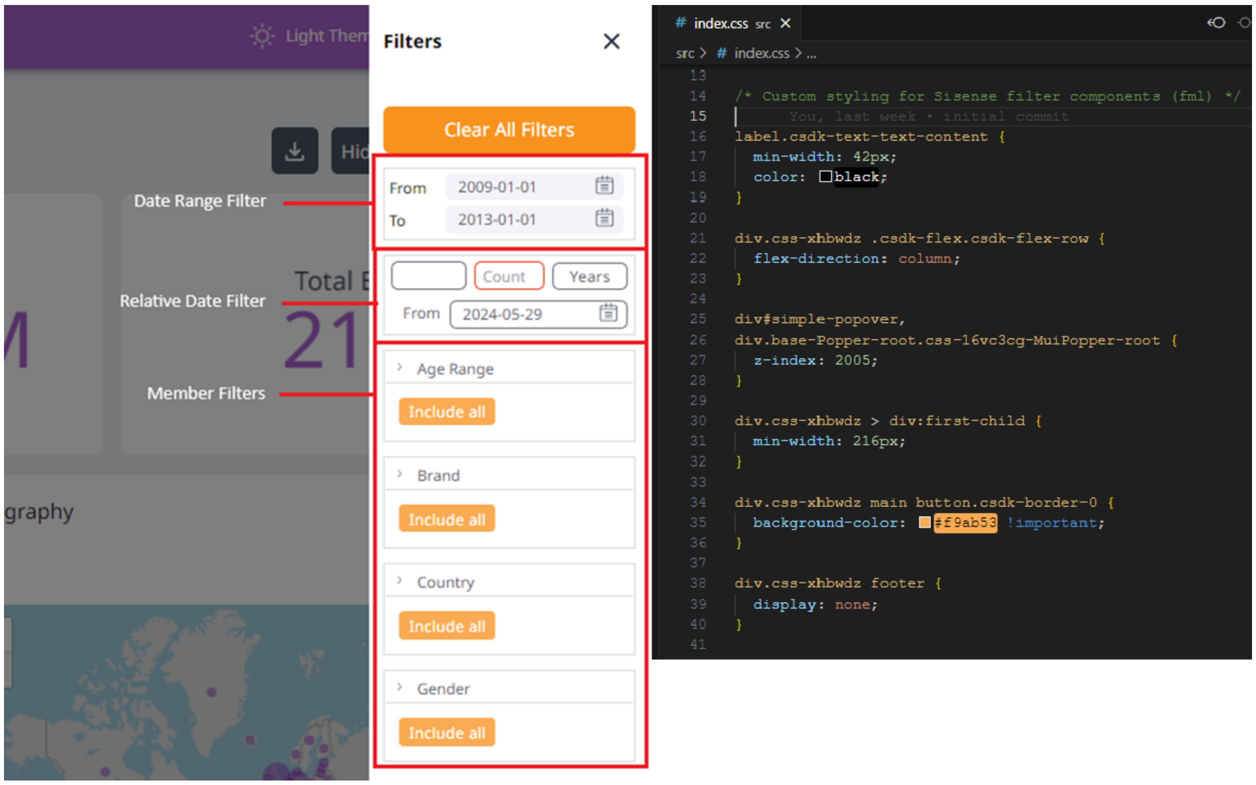
```
export default function Insights() {
  const [filterDateRange, setFilterDateRange] = useState<Filter>(
    filterFactory.dateRange(DM.Commerce.Date.Years)
  )
  const [filterRelativeDateRange, setFilterRelativeDateRange] =
    useState<Filter>(filterFactory.dateRange(DM.Commerce.Date.Years))
  const [filterAgeRange, setFilterAgeRange] = useState<Filter | null>(null)
  const [filterBrand, setFilterBrand] = useState<Filter | null>(null)
  const [filterCountry, setFilterCountry] = useState<Filter | null>(null)
  const [filterGender, setFilterGender] = useState<Filter | null>(null)

  const [showFilters, setShowFilters] = useState<boolean>(false)
  const [isTotalVisitsCompared, setCompareVisits] = useState<boolean>(false)
  const [breakBy, setBreakBy] = useState<number>(
    ecommerceChartBreakByOptions.CONDITION
  )        You, last week • initial commit

  const activeFilters: Filter[] = [
    filterDateRange,
    filterRelativeDateRange,
    filterAgeRange,
    filterBrand,
    filterCountry,
    filterGender,
  ].filter((f) => {
    if (f) return f // make sure no filters are undefined
  }) as Filter[]

  useEffect(() => window.scrollTo(0, 0), [])

  const ClearAllFiltersButton = () => (
    <Button
      isActive={true}
      onClick={() => {
        setFilterDateRange(filterFactory.dateRange(DM.Commerce.Date.Years))
        setFilterRelativeDateRange(
          filterFactory.dateRange(DM.Commerce.Date.Years)
        )
        setFilterAgeRange(null)
        setFilterBrand(null)
        setFilterCountry(null)
        setFilterGender(null)
        setShowFilters(false)
      }}
    >
      Clear All Filters
    </Button>
  )
```

Figure 7: How filters are set ***(source: src/routes/insights/index.tsx)***

filterFactory is also used in some certain charts where the filter triggers whenever the user clicks on a data record via the function **onDataPointClick()**. The file src/components/Charts/Connector/ByAge/Revenue.tsx is one of the charts having this feature.

Filter tile components

Sisense Compose SDK has a built-n series of filter tile components. However, compared to the existing chart components, these tile components aren't too easily customizable. The figure 8 below will show you the CSS settings as well as the result of that alteration to those components.



index.css src

src > index.css > ...

```css

/* Custom styling for Sisense filter components (fml) */
        You, last week • initial commit
label.csdk-text-text-content {
  min-width: 42px;
  color: black;
}

div.css-xhbwdz .csdk-flex.csdk-flex-row {
  flex-direction: column;
}

div#simple-popover,
div.base-Popper-root.css-16vc3cg-MuiPopper-root {
  z-index: 2005;
}

div.css-xhbwdz > div:first-child {
  min-width: 216px;
}

div.css-xhbwdz main button.csdk-border-0 {
  background-color: #f9ab53 !important;
}

div.css-xhbwdz footer {
  display: none;
}

```

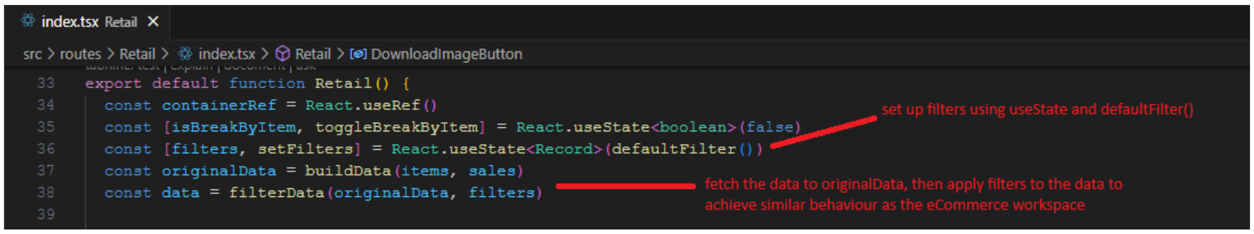
Figure 8: My filter tile components' look & feel vs. CSS customization ***(source: src/index.css)***

While I am fully aware that you can configure the filter tile color palette and related styling from **<SisenseThemeProvider />,** with different filter tile components used, they will just break. Hence, I customize those using this method.

Working with custom data

Originally, I was planning to use the **/custom-data API** to upload, add and edit the custom data records in this file; however, I did not have much time to complete this, so the data is now stored locally in the codebase.

While it cannot use **measureFactory** and **filterFactory**, I managed to make it work in a different way…



index.tsx Retail

src > routes > Retail > index.tsx > Retail > DownloadImageButton

```tsx
export default function Retail() {
  const containerRef = React.useRef()
  const [isBreakByItem, toggleBreakByItem] = React.useState<boolean>(false)
  const [filters, setFilters] = React.useState<Record>(defaultFilter())
  const originalData = buildData(items, sales)
  const data = filterData(originalData, filters)

```

set up filters using useState and defaultFilter()

fetch the data to originalData, then apply filters to the data to achieve similar behaviour as the eCommerce workspace

Figure 9: How custom data is fetched and processed ***(source: src/routes/Retail/index.tsx)***

For more information, you can check out the file above or even the file data.conf.ts which contains functions such as **buildData()**, **filterData()**, and **defaultFilter()**.