

SciChart Documentation is best viewed inside a Navigation Frame.

Click [HERE](#) to load it!

JavaScript Charting Documentation - SciChart JavaScript Charts SDK v3.x



Get Started: Tutorials, Examples > Tutorials (SciChart React) > Tutorial 03 - Modifying Chart Data and Behavior in React

Tutorial 03 - Modifying Chart Data and Behavior in React

In this tutorial we're going to show how to modify the `initChart` callback to pass properties, functions or data back into `onInit` in `scichart-react`. This is so you can connect to other UI in your application such as buttons, controls to manipulate the chart, update data or modify chart state.



The previous tutorial [Tutorial 01 - Understanding the scichart-react boilerplate](#) should serve as a reference for the initial project setup. Go ahead and copy the boilerplate to a new folder or project. You can get the code from here: [Boilerplates/scichart-react](#)

To begin with, copy the code from [Boilerplates/scichart-react](#) into a new folder. Open this folder in webstorm or vscode and perform `npm install`.

Accessing & Controlling the Chart via Nested Components

We'd like to modify `App.jsx` as follows.

1. Delete the `chartConfig` from the `scichart-react` boilerplate. We're going to use `initChart` functions instead.
2. Modify the `initChart` function to initialize a chart, but we also want to return extra functions to control the chart. These are going to be passed through to `initResult` which can be accessed in `onInit` or via `SciChartSurfaceContext`. See more below:

APP.JSX INITCHART

```
import {
  SciChartJsNavyTheme,
  SciChartSurface,
  NumericAxis,
  SplineMountainRenderableSeries,
  CursorModifier,
  XyDataSeries,
} from "scichart";
import React, { useContext } from "react";
import { SciChartReact, SciChartSurfaceContext } from "scichart-react";

const initChart = async (rootElement) => {
  const { sciChartSurface, wasmContext } = await SciChartSurface.create(
    rootElement,
    {
      theme: new SciChartJsNavyTheme(),
    }
  );

  console.log(`scichartsurface is ${sciChartSurface}`);

  sciChartSurface.xAxes.add(new NumericAxis(wasmContext));
  sciChartSurface.yAxes.add(new NumericAxis(wasmContext));

  const xValues = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
  const yValues = [1, 4, 7, 3, 7, 6, 7, 4, 2, 5];

  const mountainSeries = new SplineMountainRenderableSeries(wasmContext, {
    dataSeries: new XyDataSeries(wasmContext, { xValues, yValues }),
    fill: "SteelBlue",
    stroke: "White",
    strokeThickness: 4,
    opacity: 0.4,
  });

  sciChartSurface.renderableSeries.add(mountainSeries);

  const cursor = new CursorModifier({
    showTooltip: true,
    showYLine: true,
    showXLine: true,
    showAxisLabels: true,
    crosshairStroke: "White",
    crosshairStrokeDashArray: [5, 5],
  });
  cursor.isEnabled = false;
  sciChartSurface.chartModifiers.add(cursor);

  const addData = () => {
```

```

    console.log(`Adding data`);
    const x = xValues.length;
    const y = Math.random() * 10;
    xValues.push(x);
    yValues.push(y);
    mountainSeries.dataSeries.append(x, y);

    sciChartSurface.zoomExtents(500);
  };

  const enableTooltip = (enable) => {
    console.log(`cursorEnabled: ${enable}`);
    cursor.isEnabled = enable;
  };

  const getTooltipEnabled = () => {
    return cursor.isEnabled;
  };

  return { sciChartSurface, addData, enableTooltip, getTooltipEnabled };
};

```

3. Modify the `function App()` React Component to return a `SciChartReact` element as well as some extra buttons. These are added as nested elements inside the `SciChartReact` element, meaning they can gain access to `const initResult = useContext(SciChartSurfaceContext)` which contains the `sciChartSurface` as well as additional functions and props returned from `initChart` like the `addData`, `enableTooltip` and `getTooltipEnabled` functions.

APP.JSX REACT COMPONENT

```

const AddDataButton = () => {
  const initResult = useContext(SciChartSurfaceContext);
  const handleClick = () => {
    initResult.addData();
  };
  return <input type="button" onClick={handleClick} value="Add Data"></input>;
};

const EnableTooltipButton = () => {
  const initResult = useContext(SciChartSurfaceContext);
  const handleClick = () => {
    const tooltipEnabled = initResult.getTooltipEnabled();
    initResult.enableTooltip(!tooltipEnabled);
  };
  return (
    <input type="button" onClick={handleClick} value="Toggle Tooltip"></input>
  );
};

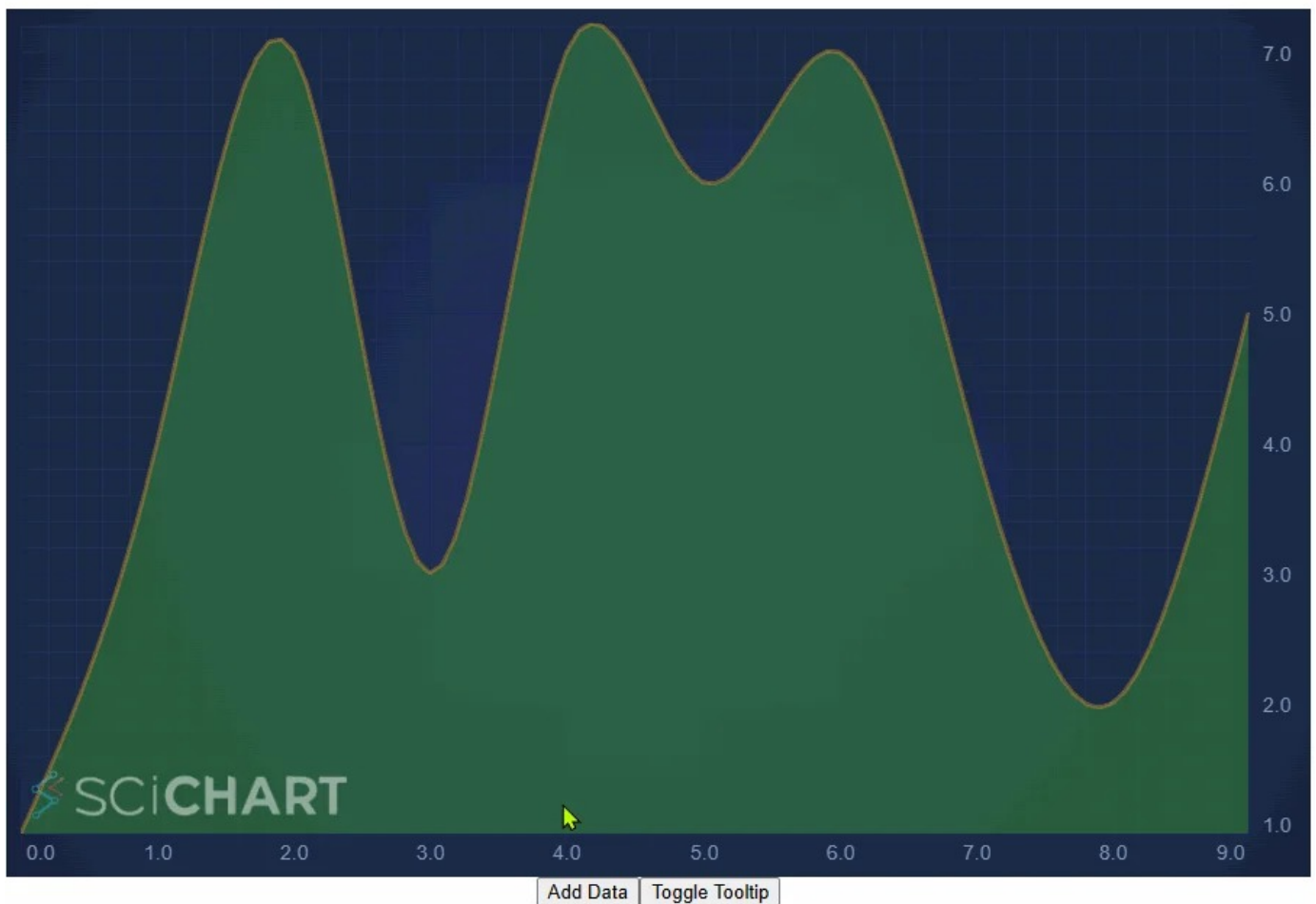
```

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>&lt;SciChartReact/&gt; with custom chart controls</h1>
      </header>
      <SciChartReact
        initChart={initChart}
        style={{ maxWidth: 900, height: 600 }}
      >
        <div style={{ display: "flex", justifyContent: "center" }}>
          <AddDataButton />
          <EnableTooltipButton />
        </div>
      </SciChartReact>
    </div>
  );
}

export default App;
```

Now run the app. You should see this behaviour

<SciChartReact/> with custom chart controls





You can get the full source code for this tutorial over at scichart.js.examples on Github under the folder [Tutorials/React/Tutorial_03_Controlling_Chart_Behaviour_In_React](#)

Accessing the Chart via Non-Nested Components

In the previous example, we accessed the `initResult` and functions to add data and control the chart state via buttons added as nested components to `<SciChartReact>`

What if you wanted to access the chart via non-nested components, e.g. buttons or a toolbar elsewhere in the application DOM? In this case you would need to store the `initResult` in a state somewhere to access the chart. Other than that, the principle is the same. `initResult` returns the `sciChartSurface` as well as any other objects or functions that you decide to return, so you can modify the chart data, state or appearance at any time.

Let's build on the previous tutorial by creating a simple toolbar to turn on/off zooming, panning and tooltip behaviours and show how to access `initResult` and modify chart state elsewhere in your app, outside the `<SciChartReact/>` component.

First start by modifying your `initChart` function. We're going to put this into a separate file, called **initChart.js**.

INITCHART.JS

```
import {
  SciChartJsNavyTheme,
  SciChartSurface,
  NumericAxis,
  SplineMountainRenderableSeries,
  RubberBandXyZoomModifier,
  ZoomPanModifier,
  RolloverModifier,
  XyDataSeries,
  EllipsePointMarker,
  ZoomExtentsModifier,
  MouseWheelZoomModifier,
} from "scichart";

// Initialize a SciChartSurface
export const initChart = async (rootElement) => {
  const { sciChartSurface, wasmContext } = await SciChartSurface.create(
    rootElement,
    {
      theme: new SciChartJsNavyTheme(),
    }
  );
};
```

```

// Add axis
sciChartSurface.xAxes.add(
    new NumericAxis(wasmContext, { axisTitle: "X Axis" })
);
sciChartSurface.yAxes.add(
    new NumericAxis(wasmContext, { axisTitle: "Y Axis" })
);

// Add a series with some data
const xValues = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const yValues = [1, 4, 7, 3, 7, 6, 7, 4, 2, 5];

const mountainSeries = new SplineMountainRenderableSeries(wasmContext, {
    dataSeries: new XyDataSeries(wasmContext, {
        dataSeriesName: "Mountain Series",
        xValues,
        yValues,
    }),
    fill: "SteelBlue",
    stroke: "White",
    strokeThickness: 4,
    opacity: 0.4,
    pointMarker: new EllipsePointMarker(wasmContext, {
        width: 7,
        height: 7,
        fill: "White",
        strokeThickness: 0,
    }),
});

// Setup series rollovermodifier properties
mountainSeries.rolloverModifierProps.tooltipTextColor = "#fff";
mountainSeries.rolloverModifierProps.tooltipColor = "SteelBlue";
mountainSeries.tooltipLabelX = "X";
mountainSeries.tooltipLabelY = "Y";
sciChartSurface.renderableSeries.add(mountainSeries);

// Add some modifiers to the chart
const rolloverModifier = new RolloverModifier({
    rolloverLineStroke: "LightSteelBlue",
    snapToDataPoint: true,
});
const rubberBandZoomModifier = new RubberBandXyZoomModifier({
    stroke: "#FFFFFF77",
    fill: "#FFFFFF33",
    strokeThickness: 1,
});
const zoomPanModifier = new ZoomPanModifier();
const zoomExtentsModifier = new ZoomExtentsModifier();
const mouseWheelZoomModifier = new MouseWheelZoomModifier();

// Set the initial state of zoom, pan and tooltip modifiers
rolloverModifier.isEnabled = false;
zoomPanModifier.isEnabled = false;

```

```

rubberBandZoomModifier.isEnabled = true;

sciChartSurface.chartModifiers.add(
  rolloverModifier,
  rubberBandZoomModifier,
  zoomPanModifier,
  zoomExtentsModifier,
  mouseWheelZoomModifier
);

// Return the SciChartSurface and modifiers. This will be passed to initResult in the onIn
// by SciChartReact
return {
  sciChartSurface,
  rolloverModifier,
  zoomPanModifier,
  rubberBandZoomModifier,
};
};

```

This code has been updated as follows. We've deleted the `addData`, `enableTooltip`, `getTooltipEnabled` functions returned from `initChart`. Instead, we've added a `RolloverModifier`, `RubberBandXyZoomModifier`, `ZoomPanModifier` and `ZoomExtentsModifier` to the chart. These are returned from `initChart` and will be stored in the `initResult` which we can access later.



In this example, the `isEnabled` properties of `rollover`, `zoompan` and `rubberbandzoom` have been set to `[false, false, true]`. By default (unless you change `modifier.executeOn`, which allows you assign a different mouse-button to each modifier), each of these chart modifiers will trigger on the mouse-left button. One way of handling this is to selectively enable & disable zooming or panning behaviours, so in the second part of this tutorial

It's not possible for all three modifiers to be enabled at the same time, as each requires the mouse left button down event, so we're going to create a toolbar that updates the chart state to enable zoom, pan or tooltip independently.

Next, modify your `App.jsx`. We are going to create some toggle buttons to provide the toolbar behaviour & UI, and pass chart state between them.

APP.JSX

```

import React, { useState } from "react";
import { SciChartReact } from "scichart-react";
import { ToggleButton } from "../ToggleButton";
import { ChartContext } from "../ChartContext";

```

```

import { initChart } from "../initChart";
import "../styles.css";

function App() {
  const [chartState, setChartState] = useState(null);


  return (
    <ChartContext.Provider value={{ chartState, setChartState }}>
      <div className="App">
        <header className="App-header">
          <h1>&lt;SciChartReact&gt; with custom chart controls</h1>
        </header>
        <div
          style={{
            display: "flex",
            justifyContent: "left",
            backgroundColor: "lightgrey",
            padding: "10px",
          }}
        >
          <ToggleButton label="Zoom" modifierKey="rubberBandZoomModifier" />
          <ToggleButton label="Pan" modifierKey="zoomPanModifier" />
          <ToggleButton label="Tooltip" modifierKey="rolloverModifier" />
          <button
            onClick={() => chartState?.sciChartSurface?.zoomExtents(500)}
            className={`normal-button`}
          >
            Zoom to Fit
          </button>
        </div>
        <SciChartReact
          initChart={initChart}
          onInit={(initResult) => setChartState(initResult)}
          style={{ maxWidth: 900, height: 600 }}
        ></SciChartReact>
      </div>
    </ChartContext.Provider>
  );
}

export default App;

```

In the code above, `<SciChartReact/>` calls `initChart` as before, but now we have an `onInit` callback. Here we store the `initResult` in a state object which you can see declared at the top of `function App()`.

App()

 Copy Code

```

function App() {
  const [chartState, setChartState] = useState(null); // useState to store initResult once we
  return (
    <ChartContext.Provider value={{ chartState, setChartState }}> // ChartContext allows shar

```



```
<div className="App">
```

We've wrapped the whole React component in a `<ChartContext/>`, part of the [React Context API](#), which allows you to share state across multiple nested components. At any point within this `<ChartContext/>` you can access `chartState` and `setChartState` which contains the `intResult`.

Next, we add three `ToggleButton` components and a `<button>` in order to provide the toolbar behaviour.

TOGGLEBUTTON.JSX

```
import React, { useContext } from "react";
import { ChartContext } from "../ChartContext";
import "../styles.css";

export const ToggleButton = ({ label, modifierKey }) => {
  const { chartState, setChartState } = useContext(ChartContext);

  const handleClick = () => {
    if (!chartState) return;

    const { sciChartSurface, ...modifiers } = chartState;

    // Disable all modifiers
    Object.values(modifiers).forEach((modifier) => {
      modifier.isEnabled = false;
    });

    // Enable only the clicked modifier
    modifiers[modifierKey].isEnabled = true;

    // Update state
    setChartState({ ...chartState });

    console.log(`${modifierKey} is now enabled`);
  };

  return (
    <button
      onClick={handleClick}
      className={`toggle-button ${
        chartState?.[modifierKey]?.isEnabled ? "active" : ""
      }`}
    >
      {label}
    </button>
  );
};
```

CHARTCONTEXT.JSX

```
import React, { createContext } from "react";

export const ChartContext = createContext(null);
```


How the ToggleButton React Component works:

- It can access the `chartState` from the `ChartContext` that we created.
- In the props passed in are a label and `modifierKey`. It can use `modifierKey` to access the correct `chartModifier` from `chartState`.
- It returns a `<button>` with a CSS class which is selected based on the `modifier.isEnabled` property
- Last but not least, `handleClick` updates the `modifier.isEnabled` property. This also calls `setChartState()` to update the context and trigger other modifier toggle buttons to show as enabled/disabled.

One last thing, we extracted the CSS for the buttons into a stylesheet, `styles.css`.


To load this we had to install `style-loader` and `css-loader` packages and update `webpack.config.js` with a new rule and a test for `*.css` files. You can find the required changes below.

npm

 Copy Code

```
npm install --save-dev style-loader css-loader
```

styles.css

 Copy Code

```
/* Toggle Button (Windows/macOS-style) */
.toggle-button {
  background-color: #e0e0e0; /* Default OFF color */
  color: #333;
  border: none;
  padding: 8px 16px;
  border-radius: 2px;
  cursor: pointer;
  font-size: 14px;
  font-weight: bold;
  transition: background 0.2s, color 0.2s, box-shadow 0.2s;
  box-shadow: 0 1px 3px rgba(0, 0, 0, 0.2);
  outline: none;
}

.toggle-button.active {
```

```


background-color: #007aff; /* ON color */
color: white;
box-shadow: 0 2px 5px rgba(0, 122, 255, 0.5);
}

/* Normal Button (Zoom to Fit) */
.normal-button {
background-color: #e0e0e0; /* Default OFF color */
color: #333;
border: none;
padding: 8px 16px;
border-radius: 2px;
cursor: pointer;
font-size: 14px;
font-weight: bold;
transition: background 0.2s, color 0.2s, box-shadow 0.2s;
box-shadow: 0 2px 5px rgba(0, 0, 0, 0.2);
outline: none;
}

.normal-button:hover {
background-color: #55aaff;
}

```

Example Title

 Copy Code

```

const path = require("path");
const CopyPlugin = require("copy-webpack-plugin");
const webpack = require("webpack");

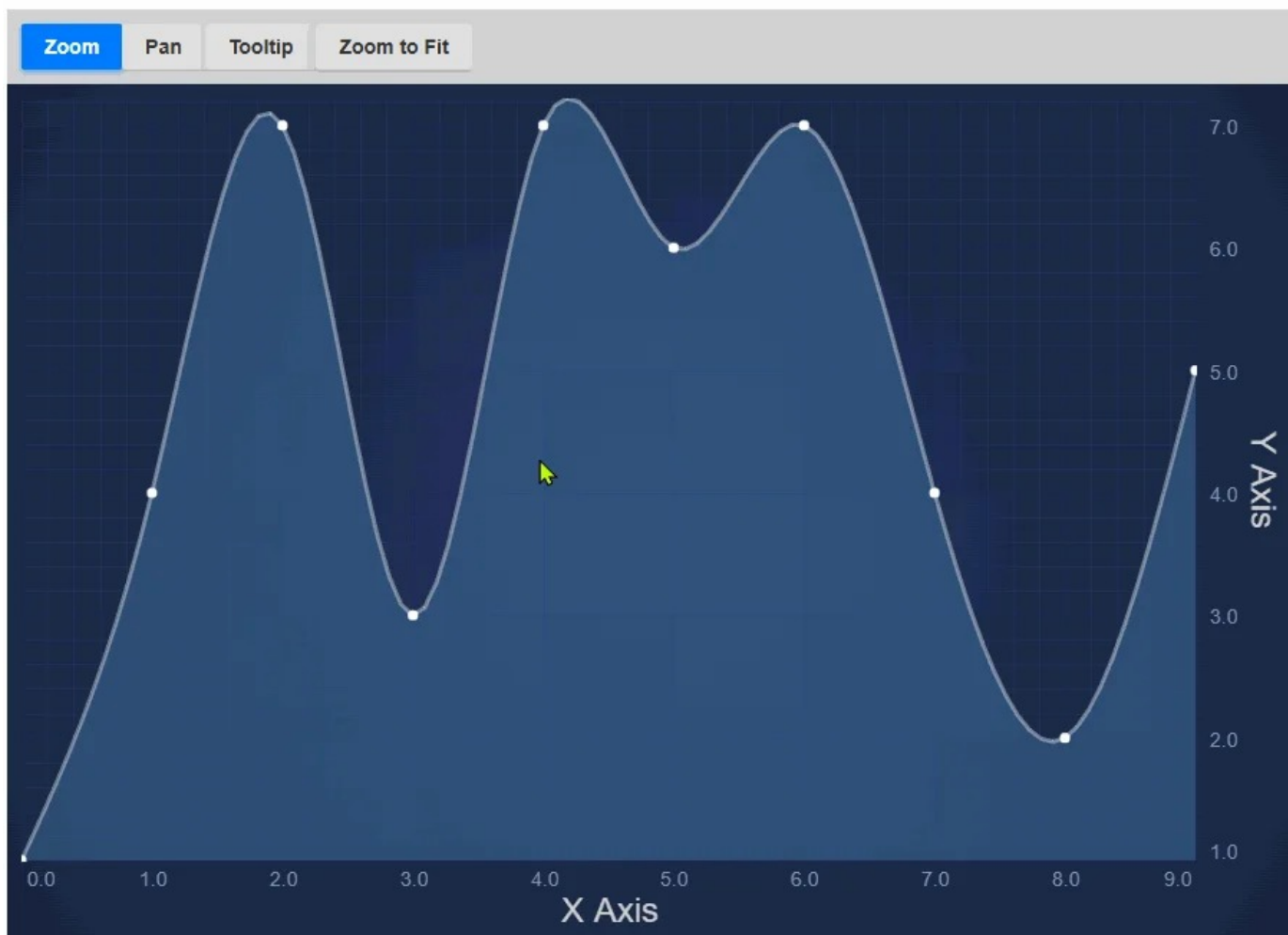
module.exports = {
  mode: "production",
  entry: "./src/index.jsx",
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader",
        },
      },
      {
        test: /\.css$/, // Add CSS Loader Rule
        use: ["style-loader", "css-loader"],
      },
    ],
  },
  // ...

```

Here's the completed tutorial output, with the three modifiers enabled/disabled via toolbar

buttons.

<SciChartReact/> with custom chart controls



You can get the full source code for this tutorial over at scichart.js.examples on Github under the folder [Tutorials/React/Tutorial_03b_Controlling_Chart_Behaviour_With_Toolbar](#)

SCICHART® is a Registered Trademark in the UK, US and EU. Copyright SciChart Ltd 2011-2024.

[Sitemap](#) | [Send Feedback](#) 