Mastering TanStack Table: A Comprehensive Guide

This book provides an in-depth exploration of TanStack Table, covering installation, core concepts, features, and APIs. It's an essential resource for developers looking to leverage the full potential of TanStack Table in their projects.

Table of Contents

Introduction and Getting Started

Core Guides

Feature Guides

Core APIs

Feature APIs

- Column Filtering core
- Column Faceting core
- Column Ordering core
- Column Pinning core
- Column Sizing core
- Column Visibility core
- Global Faceting core
- Global Filtering core
- Sorting core
- Grouping core
- Expanding core
- Pagination core
- Row Pinning core
- Row Selection core

Enterprise

- AG Grid core

Examples

- Basic react
- Header Groups react
- Column Filters react
- Column Filters (Faceted) react
- Fuzzy Search Filters react
- Column Ordering react
- Column Ordering (DnD) react
- Column Pinning react
- Sticky Column Pinning react
- Column Sizing react
- Performant Column Resizing react
- Column Visibility react
- Editable Data react
- Expanding react
- Sub Components react
- Fully Controlled react
- Grouping react
- Pagination react
- Pagination Controlled react
- Row DnD react
- Row Pinning react
- Row Selection react
- Sorting react
- Virtualized Columns react
- Virtualized Columns (Experimental) react
- Virtualized Rows react
- Virtualized Rows (Experimental) react
- Virtualized Infinite Scrolling react
- Kitchen Sink react
- React Bootstrap react
- Material UI Pagination react
- React Full Width react
- React Full Width Resizable react
- Custom Features react

- Query Router Search Params react

Introduction

TanStack Table Docs

Table v8

On this page

Introduction

TanStack Table is a **Headless UI** library for building powerful tables & datagrids for TS/JS, React, Vue, Solid, Qwik, and Svelte.

What is "headless" UI?

Headless UI is a term for libraries and utilities that provide the logic, state, processing and API for UI elements and interactions, but **do not provide markup, styles, or pre-built implementations.** Scratching your head yet? 😉 Headless UI has a few main goals:

The hardest parts of building complex UIs usually revolve around state, events, side-effects, data computation/management. By removing these concerns from the markup, styles and implementation details, our logic and components can be more modular and reusable.

Building UI is a very branded and custom experience, even if that means choosing a design system or adhering to a design spec. To support this custom experience, component-based UI libraries need to support a massive (and seemingly endless) API surface around markup and style customization. Headless UI libraries decouple your logic from your UI.

When you use a headless UI library, the complex task of **data-processing, state-management, and business logic** are handled for you, leaving you to worry about higher-cardinality decisions that differ across implementations and use cases.

> Want to dive deeper? Read more about Headless UI.

Component-based libraries vs Headless libraries

In the ecosystem of table/datagrid libraries, there are two main categories:

- Component-based table libraries
- Headless table libraries

Which kind of table library should I use?

Each approach has subtle tradeoffs. Understanding these subtleties will help you make the right decision for your application and team.

Component-based Table Libraries

Component-based table libraries will typically supply you with a feature-rich drop-in solution and ready-to-use components/markup complete with styles and theming. **AG Grid** is a great example of this type of table library.

Pros:

- Ship with ready-to-use markup/styles
- Little setup required
- Turn-key experience

Cons:

- Less control over markup
- Custom styles are typically theme-based
- Larger bundle sizes
- Highly coupled to framework adapters and platforms

  If you want a ready-to-use table and design/bundle-size are not hard requirements, then you should consider using a component-based table library.

There are a lot of component-based table libraries out there, but we believe **AG Grid** is the gold standard and is by far our favorite grid-sibling (don't tell the others 🤫).

Headless Table Libraries

Headless table libraries will typically supply you with functions, state, utilities, and event listeners to build your own table markup or attach to existing table markups.

Pros:

- Full control over markup and styles
- Supports all styling patterns (CSS, CSS-in-JS, UI libraries, etc)
- Smaller bundle sizes
- Portable. Run anywhere JS runs!

Cons:

- More setup required
- No markup, styles, or themes provided

  If you want a lighter-weight table or full control over the design, then you should consider using a headless table library.

There are very few headless table libraries out there and obviously, **TanStack Table** is our favorite!

Edit on GitHub

On this page

Additional Links

- Discord
- Overview

Our Partners

- **TanStack Start**: Full-document SSR, Streaming, Server Functions, bundling and more, powered by TanStack Router and Vite - Ready to deploy to your favorite hosting provider.
- **TanStack Store**: The immutable-reactive data store that powers the core of TanStack libraries and their framework adapters.

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Subscribe

On this page

Overview

TanStack Table Docs

On this page

Introduction

TanStack Table's core is **framework agnostic**, which means its API is the same regardless of the framework you're using. Adapters are provided to make working with the table core easier depending on your framework. See the Adapters menu for available adapters.

Typescript

While TanStack Table is written in TypeScript, using TypeScript in your application is optional (but recommended as it comes with outstanding benefits to both you and your codebase).

Headless

As extensively mentioned in the Intro section, TanStack Table is **headless**. This means it doesn't render any DOM elements, and instead relies on you, the UI/UX developer, to provide the table's markup and styles. This approach allows building a table compatible with any UI framework, including React, Vue, Solid, Svelte, Qwik, and even JS-to-native platforms like React Native!

Core Objects and Types

The table core uses the following abstractions, commonly exposed by adapters:

- **Column Defs**
  Objects used to configure a column and its data model, display templates, and more.
- **Table**
  The core table object containing both state and API.
- **Table Data**
  The core data array you provide to the table.
- **Columns**
  Each column mirrors its respective column def and provides column-specific APIs.
- **Rows**
  Each row mirrors its respective row data and provides row-specific APIs.
- **Header Groups**
  Computed slices of nested header levels, each containing a group of headers.
- **Headers**
  Each header is either directly associated with or derived from its column def and provides header-specific APIs.
- **Cells**
  Each cell mirrors its respective row-column intersection and provides cell-specific APIs.

There are additional structures related to features like filtering, sorting, and grouping, which you can explore in the features section.

Documentation (from GitHub link)

Edit on GitHub

On this page (repeated navigation)

Additional sections

Our Partners

- React Data Grid
  (opens in new tab)

Additional Resources

TanStack Router

TanStack Router
A powerful React router for client-side and full-stack React applications. Fully type-safe APIs, first-class search parameters for managing state in the URL, and seamless integration with the React ecosystem.
Learn More

TanStack Ranger

TanStack Ranger
Headless, lightweight, and extensible primitives for building range and multi-range sliders.
Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.
No spam. Unsubscribe at any time.

Installation

On this page

Introduction

Before we dig in to the API, let's get you set up!

Install your table adapter as a dependency using your favorite npm package manager.

Only install ONE of the following packages:

React Table

```
npm install @tanstack/react-table
```

The `@tanstack/react-table` package works with React 16.8, React 17, React 18, and React 19.

> **NOTE:** Even though the react adapter works with React 19, it may not work with the new React Compiler that's coming out alongside React 19. This may be fixed in future TanStack Table updates.

Vue Table

```
npm install @tanstack/vue-table
```

The `@tanstack/vue-table` package works with Vue 3.

Solid Table

```
npm install @tanstack/solid-table
```

The `@tanstack/solid-table` package works with Solid-JS 1.

Svelte Table

```
npm install @tanstack/svelte-table
```

The `@tanstack/svelte-table` package works with Svelte 3 and Svelte 4.

> **NOTE:** There is not a built-in Svelte 5 adapter yet, but you can still use TanStack Table with Svelte 5 by installing the `@tanstack/table-core` package and using a custom adapter from the community. See this PR for inspiration.

Qwik Table

```
npm install @tanstack/qwik-table
```

The `@tanstack/qwik-table` package works with Qwik 1.

> **NOTE:** There will be a "breaking change" release in the near future to support Qwik 2. This will be released as a minor version bump, but will be documented. Qwik 2 itself will have no breaking changes, but its name on the npm registry will change, and require different peer dependencies.

> **NOTE:** The current Qwik adapter only works with CSR. More improvements may not be available until a future table version.

Angular Table

```
npm install @tanstack/angular-table
```

The `@tanstack/angular-table` package works with Angular 17. The Angular adapter uses a new Angular Signal implementation.

Lit Table

```
npm install @tanstack/lit-table
```

The `@tanstack/lit-table` package works with Lit 3.

Table Core (no framework)

```
npm install @tanstack/table-core
```

Don't see your favorite framework (or favorite version of your framework) listed? You can always just use the `@tanstack/table-core` package and build your own adapter in your own codebase. Usually, only a thin wrapper is needed to manage state and rendering for your specific framework. Browse the source code of all of the other adapters to see how they work.

Edit on GitHub

On this page

Migrating to V8 Guide

On this page

Migrating to V8

TanStack Table V8 was a major rewrite of React Table v7 from the ground up in TypeScript. The overall structure/organization of your markup and CSS will largely remain the same, but many of the APIs have been renamed or replaced.

Notable Changes

- Full rewrite to TypeScript with types included in the base package
- Removal of plugin system to favor more inversion of control
- Vastly larger and improved API (and new features like pinning)
- Better controlled state management
- Better support for server-side operations
- Complete (but optional) data pipeline control
- Agnostic core with framework adapters for React, Solid, Svelte, Vue, and potentially more in the future
- New Dev Tools

Install the new Version

The new version of TanStack Table is published under the `@tanstack` scope. Install the new package using your favorite package manager:

```
npm uninstall react-table @types/react-table
npm install @tanstack/react-table
```

Example usage

```
- import { useTable } from 'react-table' // [!code --]
+ import { useReactTable } from '@tanstack/react-table' // [!code ++]

- import { useTable } from 'react-table' // [!code --]
+ import { useReactTable } from '@tanstack/react-table' // [!code ++]
```

Types are now included in the base package, so you can remove the `@types/react-table` package.

> **Note:** If you want, you can keep the old `react-table` packages installed so that you can gradually migrate your code. You should be able to use both packages side-by-side for separate tables without any issues.

Update Table Options

- Rename `useTable` to `useReactTable`
- The old hook and plugin systems have been removed, but they have been replaced with tree-shakable row model imports for each feature.

```
- import { useTable, usePagination, useSortBy } from 'react-table'; // [!code --]
+ import { // [!code ++]
+   useReactTable, // [!code ++]
+   getCoreRowModel, // [!code ++]
+   getPaginationRowModel, // [!code ++]
+   getSortedRowModel // [!code ++]
+ } from '@tanstack/react-table'; // [!code ++]

-   const tableInstance = useTable( // [!code --]
-     { columns,  data }, // [!code --]
-     useSortBy, // [!code --]
-     usePagination, //order of hooks used to matter // [!code --]
-     // etc. // [!code --]
-   ); // [!code --]
+   const tableInstance = useReactTable({ // [!code ++]
+     columns, // [!code ++]
+     data, // [!code ++]
+     getCoreRowModel: getCoreRowModel(), // [!code ++]
+     getPaginationRowModel: getPaginationRowModel(), // [!code ++]
+     getSortedRowModel: getSortedRowModel(), //order doesn't matter anymore! // [!cod
+     // etc. // [!code ++]
+   }); // [!code ++]
```

Repeat the above pattern for other instances.

- All `disable*` table options were renamed to `enable*` table options. (e.g., `disableSortBy` is now `enableSorting`, `disableGroupBy` is now `enableGrouping`, etc.)

Update column definitions

- `accessor` was renamed to either `accessorKey` or `accessorFn` (depending on whether you are using a string or function)
- `width`, `minWidth`, `maxWidth` were renamed to `size`, `minSize`, `maxSize`
- Optionally, you can use the new `createColumnHelper` function around each column definition for better TypeScript hints. (You can still just use an array of column definitions if you prefer.)

The first parameter is the accessor function or string. The second parameter is an object of column options.

Example:

```
const columns = [
-  { // [!code --]
-    accessor: 'firstName', // [!code --]
-    Header: 'First Name', // [!code --]
-  }, // [!code --]
-  { // [!code --]
-    accessor: row => row.lastName, // [!code --]
-    Header: () => <span>Last Name</span>, // [!code --]
-  }, // [!code --]
```

```
+  columnHelper.accessor('firstName', { // accessorKey // [!code ++]
+    header: 'First Name', // [!code ++]
+  }), // [!code ++]
+  columnHelper.accessor(row => row.lastName, { // accessorFn // [!code ++]
+    header: () => <span>Last Name</span>, // [!code ++]
+  }), // [!code ++]
]
```

Alternatively:

```
const columns = [
-  { // [!code --]
-    accessor: 'firstName', // [!code --]
-    Header: 'First Name', // [!code --]
-  }, // [!code --]
-  { // [!code --]
-    accessor: row => row.lastName, // [!code --]
-    Header: () => <span>Last Name</span>, // [!code --]
-  }, // [!code --]
+  {
+    accessorKey: 'firstName', // [!code ++]
+    header: 'First Name', // [!code ++]
+  },
+  {
+    accessorFn: row => row.lastName, // [!code ++]
+    header: () => <span>Last Name</span>, // [!code ++]
+  },
]
```

> **Note:** If defining columns inside a component, store the column definitions in either a `useMemo` or `useState` hook to give them a stable identity. This helps with performance and prevents unnecessary re-renders.

Column Option Name Changes

- `Header` was renamed to `header`
- `Cell` was renamed to `cell` (the cell render function has also changed)
- `Footer` was renamed to `footer`
- All `disable*` column options were renamed to `enable*`
- `sortType` to `sortingFn`

Changes to custom cell renderers

- `value` was renamed to `getValue`. Instead of providing the value directly, a function `getValue` is exposed for evaluating it.
- `cell: { isGrouped, isPlaceholder, isAggregated }` → now `cell: { getIsGrouped, getIsPlaceholder, getIsAggregated }`
- Column's base props are now RT-specific; user-added properties move into `columnDef`.
- Props passed into the `useTable` hook now appear under `options`.

Migrate Table Markup

- Use `flexRender()` instead of `cell.render('Cell')` or `column.render('Header')`.

- Methods like `getHeaderProps`, `getFooterProps`, `getCellProps`, `getRowProps` are deprecated.

  Important: TanStack Table no longer automatically provides default styles or accessibility attributes. Define `onClick` handlers manually using new `get*Handler` helpers. Define `key` props yourself, and `colSpan` props if needed.

Example:

```
- <th {...header.getHeaderProps()}>{cell.render('Header')}</th> // [!code --]
+ <th colSpan={header.colSpan} key={column.id}> // [!code ++]
+   {flexRender( // [!code ++]
+     header.column.columnDef.header, // [!code ++]
+     header.getContext() // [!code ++]
+   )} // [!code ++]
+ </th> // [!code ++]

- <td {...cell.getCellProps()}>{cell.render('Cell')}</td> // [!code --]
+ <td key={cell.id}> // [!code ++]
+   {flexRender( // [!code ++]
+     cell.column.columnDef.cell, // [!code ++]
+     cell.getContext() // [!code ++]
+   )} // [!code ++]
+ </td> // [!code ++]
```

In column definitions:

```
- Header: ({ getToggleAllRowsSelectedProps }) => ( // [!code --]
-   <input type="checkbox" {...getToggleAllRowsSelectedProps()} /> // [!code --]
- ), // [!code --]
- Cell: ({ row }) => ( // [!code --]
-   <input type="checkbox" {...row.getToggleRowSelectedProps()} /> // [!code --]
- ), // [!code --]
+ header: ({ table }) => ( // [!code ++]
+   <Checkbox // [!code ++]
+     checked={table.getIsAllRowsSelected()} // [!code ++]
+     indeterminate={table.getIsSomeRowsSelected()} // [!code ++]
+     onChange={table.getToggleAllRowsSelectedHandler()} // [!code ++]
+   /> // [!code ++]
+ ), // [!code ++]
+ cell: ({ row }) => ( // [!code ++]
+   <Checkbox // [!code ++]
+     checked={row.getIsSelected()} // [!code ++]
+     disabled={!row.getCanSelect()} // [!code ++]
+     indeterminate={row.getIsSomeSelected()} // [!code ++]
+     onChange={row.getToggleSelectedHandler()} // [!code ++]
+   /> // [!code ++]
+ ), // [!code ++]
```

Other Changes

- Custom `filterTypes` (now called `filterFns`) have a new function signature; they return a boolean indicating if the row should be included.

```
- (rows: Row[], id: string, filterValue: any) => Row[] // [!code --]
+ (row: Row, id: string, filterValue: any) => boolean // [!code ++]
```

> **Note:** This guide is a work in progress. Please consider contributing if you have time!

Edit on GitHub

FAQ

How do I stop infinite rendering loops?

If you are using React, there is a very common pitfall that can cause infinite rendering. If you fail to give your `columns`, `data`, or `state` a stable reference, React will enter an infinite loop of re-rendering upon any change to the table state.

Why does this happen? Is this a bug in TanStack Table? No, it is not. *This is fundamentally how React works*, and properly managing your columns, data, and state will prevent this from happening.

TanStack Table triggers a re-render whenever the `data` or `columns` that are passed into it change, or when any of the table's state changes.

> **Failing to give `columns` or `data` stable references can cause an infinite loop of re-renders.**

Pitfall 1: Creating new columns or data on every render

```
export default function MyComponent() {
  // 😵 BAD: This will cause an infinite loop because `columns` is redefined each rend
  const columns = [
    // ...
  ];

  // 😵 BAD: This will cause an infinite loop because `data` is redefined each render
  const data = [
    // ...
  ];

  const table = useReactTable({
    columns,
    data,
  });

  return <table>...</table>;
}
```

Solution 1: Stable references with `useMemo` or `useState`

In React, you can give stable references by defining variables outside or above the component, or by using `useMemo` or `useState`, or third-party state management.

```
// ✅ OK: Define columns outside of the component
const columns = [
  // ...
];

// ✅ OK: Define data outside of the component
const data = [
  // ...
];

export default function MyComponent() {
```

```
  // ✅ GOOD: Use `useMemo` for stability
  const columns = useMemo(() => [
    // ...
  ], []);

  // ✅ GOOD: Use `useState` for stability
  const [data, setData] = useState(() => [
    // ...
  ]);

  const table = useReactTable({
    columns,
    data,
  });

  return <table>...</table>;
}
```

Pitfall 2: Mutating columns or data in place

Even if you initialize `columns` and `data` with stable references, mutating them in place can cause re-render loops, e.g., inline `data.filter()`.

```
export default function MyComponent() {
  // ✅ GOOD
  const columns = useMemo(() => [
    // ...
  ], []);

  // ✅ GOOD (React Query provides stable `data`)
  const { data, isLoading } = useQuery({ /* ... */ });

  const table = useReactTable({
    columns,
    // ❌ BAD: `data` is filtered inline, mutating its reference
    data: data?.filter(d => d.isActive) ?? [],
  });

  return <table>...</table>;
}
```

Solution 2: Memoize your data transformations

Memoize such transformations with `useMemo`:

```
export default function MyComponent() {
  const columns = useMemo(() => [
    // ...
  ], []);

  const { data } = useQuery({ /* ... */ });

  const filteredData = useMemo(() => data?.filter(d => d.isActive) ?? [], [data]);

  const table = useReactTable({
```

```
    columns,
    data: filteredData,
  });

  return <table>...</table>;
}
```

React Forget

When React Forget is released, these issues might be obsolete, or consider using Solid.js for such reactivity.

How do I stop my table state from automatically resetting when my data changes?

Most plugins reset state on data updates, but you can prevent this to support external filtering or edits.

Set relevant plugins' options to `false` to disable auto-reset:

```
const [data, setData] = React.useState([]);
const skipPageResetRef = React.useRef();

const updateData = newData => {
  skipPageResetRef.current = true; // disable resetting
  setData(newData);
};

React.useEffect(() => {
  skipPageResetRef.current = false; // re-enable
}, []);

useReactTable({
  // ...
  autoResetPageIndex: !skipPageResetRef.current,
  autoResetExpanded: !skipPageResetRef.current,
});
```

Now, updates won't auto-reset the table's state.

Edit on GitHub

Edit this page on GitHub

On this page

React Table | TanStack Table React Docs

On this page

React Table

The `@tanstack/react-table` adapter is a wrapper around the core table logic. Most of its job is related to managing state in the *"react"* way, providing types and the rendering implementation of cell/header/footer templates.

useReactTable

Takes an `options` object and returns a table.

```
import { useReactTable } from '@tanstack/react-table'

function App() {
  const table = useReactTable(options)

  // ...render your table
}
```


```
import { useReactTable } from '@tanstack/react-table'

function App() {
  const table = useReactTable(options)

  // ...render your table
}
```


Edit on GitHub

Additional Links

- FAQ
- Data Guide

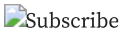
Our Partners

- **TanStack Query**
  Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state".

- **TanStack DB**
  TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.



No spam. Unsubscribe at any time.

Data Guide

Data Guide

Tables start with your data. Your column definitions and rows will depend on the shape of your data. TanStack Table has some TypeScript features that will help you create the rest of your table code with a great type-safe experience. If you set up your data and types correctly, TanStack Table will be able to infer the shape of your data and enforce that your column definitions are made correctly.

TypeScript

TypeScript is **NOT** required to use the TanStack Table packages... *but* TanStack Table is written and organized in such a way that makes the awesome TypeScript experience that you get feel like it is one of the main selling points of the library. If you are not using TypeScript, you will be missing out on a lot of great autocompletion and type-checking features that will both speed up your development time and reduce the number of bugs in your code.

TypeScript Generics

Having a basic understanding of what TypeScript Generics are and how they work will help you understand this guide better, but it should be easy enough to pick up as you go. The official TypeScript Generics Docs may be helpful for those not yet familiar with TypeScript.

Defining Data Types

`data` is an array of objects that will be turned into the rows of your table. Each object in the array represents a row of data (under normal circumstances). If you are using TypeScript, we usually define a type for the shape of our data. This type is used as a generic type for all of the other table, column, row, and cell instances. This generic is usually referred to as `TData` throughout the rest of the TanStack Table types and APIs.

For example, if we have a table that displays a list of users in an array like this:

```
[
  {
    "firstName": "Tanner",
    "lastName": "Linsley",
    "age": 33,
    "visits": 100,
    "progress": 50,
    "status": "Married"
  },
  {
    "firstName": "Kevin",
    "lastName": "Vandy",
    "age": 27,
    "visits": 200,
    "progress": 100,
    "status": "Single"
  }
]
```

We can define a `User` type like this:

```
// TData
type User = {
  firstName: string
  lastName: string
  age: number
  visits: number
  progress: number
  status: string
}
```

Then we can define our `data` array with this type, and TanStack Table will be able to infer lots of types for us later on in our columns, rows, cells, etc. This is because the `data` type is literally defined as the `TData` generic type. Whatever you pass to the `data` table option will become the `TData` type for the rest of the table instance. Just make sure your column definitions use the same `TData` type as the `data` type when you define them later.

```
// note: data needs a "stable" reference in order to prevent infinite re-renders
const data: User[] = []
// or
const [data, setData] = React.useState<User[]>([])
// or
const data = ref<User[]>([]); // Vue
// etc...
```

Deep Keyed Data

If your data is not a nice flat array of objects, that's okay! Once you get around to defining your columns, there are strategies for accessing deeply nested data in your accessors.

If your `data` looks something like this:

```
[
  {
    "name": {
      "first": "Tanner",
      "last": "Linsley"
    },
    "info": {
      "age": 33,
      "visits": 100
    }
  },
  {
    "name": {
      "first": "Kevin",
      "last": "Vandy"
    },
    "info": {
      "age": 27,
      "visits": 200
    }
  }
]
```

You can define a type like this:

```
type User = {
  name: {
    first: string
    last: string
  }
  info: {
    age: number
    visits: number
  }
}
```

And you will be able to access the data in your column definitions with either dot notation in an `accessorKey` or simply by using an `accessorFn`.

```
const columns = [
  {
    header: 'First Name',
    accessorKey: 'name.first',
  },
  {
    header: 'Last Name',
    accessorKey: 'name.last',
  },
  {
    header: 'Age',
    accessorFn: row => row.info.age,
  },
  //...
]
```

This is discussed in more detail in the Column Def Guide.

> **NOTE:** The "keys" in your JSON data can usually be anything, but any periods in the keys will be interpreted as a deep key and will cause errors.

Nested Sub-Row Data

If you are using expanding features, it can be common to have nested sub-rows in your data. This results in a recursive type that is a bit different.

So if your data looks like this:

```
[
  {
    "firstName": "Tanner",
    "lastName": "Linsley",
    "subRows": [
      {
        "firstName": "Kevin",
        "lastName": "Vandy"
      },
      {
        "firstName": "John",
        "lastName": "Doe",
        "subRows": [
          //...
        ]
```

```
      }
    ]
  },
  {
    "firstName": "Jane",
    "lastName": "Doe"
  }
]
```

You can define a type like this:

```
type User = {
  firstName: string
  lastName: string
  subRows?: User[] // does not have to be called "subRows", can be called anything
}
```

Where `subRows` is an optional array of `User` objects. This is discussed in more detail in the Expanding Guide.

Give Data a "Stable" Reference

The `data` array that you pass to the table instance **must** have a "stable" reference in order to prevent bugs that cause infinite re-renders (especially in React).

This will depend on which framework adapter you are using, but in React, you should often use `React.useState`, `React.useMemo`, or similar to ensure that both the `data` and `columns` table options have stable references.

```
const fallbackData = [];

export default function MyComponent() {
  // ✅ GOOD: This will not cause an infinite loop of re-renders because `columns` is
  const columns = useMemo(() => [
    // ...
  ], []);

  // ✅ GOOD: This will not cause an infinite loop of re-renders because `data` is a s
  const [data, setData] = useState(() => [
    // ...
  ]);

  // Columns and data are defined in a stable reference, will not cause infinite loop!
  const table = useReactTable({
    columns,
    data ?? fallbackData, // also good to use a fallback array that is defined outside
  });

  return <table>...{/* table rendering */}...</table>;
}

const fallbackData = [];

export default function MyComponent() {
  // ✅ GOOD: This will not cause an infinite loop of re-renders because `columns` is
  const columns = useMemo(() => [
```

```
    // ...
  ], []);

  // ✅ GOOD: This will not cause an infinite loop of re-renders because `data` is a s
  const [data, setData] = useState(() => [
    // ...
  ]);

  // Columns and data are defined in a stable reference, will not cause infinite loop!
  const table = useReactTable({
    columns,
    data ?? fallbackData,
  });

  return <table>...{/* table rendering */}...</table>;
}
```

`React.useState` and `React.useMemo` are not the only ways to give your data a stable reference. You can also define your data outside of the component or use a third-party state management library like Redux, Zustand, or TanStack Query.

The main thing to avoid is defining the `data` array inside the same scope as the `useReactTable` call. That will cause the `data` array to be redefined on every render, which leads to an infinite loop of re-renders.

```
export default function MyComponent() {
  // 😵 BAD: This will cause an infinite loop of re-renders because `columns` is redef
  const columns = [
    // ...
  ];

  // 😵 BAD: This will cause an infinite loop of re-renders because `data` is redefine
  const data = [
    // ...
  ];

  // ❌ Columns and data are defined in the same scope as `useReactTable` without a st
  const table = useReactTable({
    columns,
    data ?? [], // ❌ Also bad because the fallback array is re-created on every rende
  });

  return <table>...{/* table rendering */}...</table>;
}

export default function MyComponent() {
  // 😵 BAD: This will cause an infinite loop of re-renders because `columns` is redef
  const columns = [
    // ...
  ];

  // 😵 BAD: This will cause an infinite loop of re-renders because `data` is redefine
  const data = [
    // ...
  ];

  // ❌ Columns and data are defined in the same scope as `useReactTable` without a st
  const table = useReactTable({
```

```
    columns,
    data ?? [], // ❌ Also bad because the fallback array is re-created on every rende
  });

  return <table>...{/* table rendering */}...</table>;
}
```

How TanStack Table Transforms Data

Later, in other parts of these docs, you will see how TanStack Table processes the `data` that you pass to the table and generates the row and cell objects that are used to create the table. The `data` that you pass to the table is never mutated by TanStack Table, but the actual values in the rows and cells may be transformed by the accessors in your column definitions, or by other features performed by row models like grouping or aggregation.

How Much Data Can TanStack Table Handle?

Believe it or not, TanStack Table was actually built to scale up to handle potentially hundreds of thousands of rows of data in the client. This is obviously not always possible, depending on the size of each column's data and the number of columns. However, the sorting, filtering, pagination, and grouping features are all built with performance in mind for large datasets.

The default mindset of a developer building a data grid is to implement server-side pagination, sorting, and filtering for large datasets. This is still usually a good idea, but many developers underestimate how much data can actually be handled in the client with modern browsers and the right optimizations. If your table will never have more than a few thousand rows, you can probably leverage the client-side features in TanStack Table instead of implementing them on the server. Of course, you should test with your actual data to ensure performance.

This is discussed in more detail in the Pagination Guide.

On this page

Columns Guide

API

Table API

Column Definitions Guide

Column Definitions Guide

Column defs are the single most important part of building a table. They are responsible for:

- Building the underlying data model that will be used for everything including sorting, filtering, grouping, etc.
- Formatting the data model into what will be displayed in the table
- Creating header groups, headers and footers
- Creating columns for display-only purposes, eg. action buttons, checkboxes, expanders, sparklines, etc.

Column Def Types

The following "types" of column defs aren't actually TypeScript types, but more so a way to talk about and describe overall categories of column defs:

- **Accessor Columns**
  - Have an underlying data model which means they can be sorted, filtered, grouped, etc.
- **Display Columns**
  - Do *not* have a data model which means they cannot be sorted, filtered, etc, but they can display arbitrary content in the table, e.g., a row actions button, checkbox, expander, etc.
- **Grouping Columns**
  - Do *not* have a data model, so they too cannot be sorted or filtered, and are used to group other columns together. It's common to define a header or footer for a column group.

Column Helpers

While column defs are just plain objects, a **createColumnHelper** function is exposed from the table core which, when called with a row type, returns a utility for creating different column definition types with the highest type-safety possible.

Here's an example of creating and using a column helper:

```
// Define your row shape
type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
```

```
}

const columnHelper = createColumnHelper<Person>()

// Make some columns!
const defaultColumns = [
  // Display Column
  columnHelper.display({
    id: 'actions',
    cell: props => <RowActions row={props.row} />,
  }),
  // Grouping Column
  columnHelper.group({
    header: 'Name',
    footer: props => props.column.id,
    columns: [
      // Accessor Column
      columnHelper.accessor('firstName', {
        cell: info => info.getValue(),
        footer: props => props.column.id,
      }),
      // Accessor Column
      columnHelper.accessor(row => row.lastName, {
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      }),
    ],
  }),
  // Grouping Column
  columnHelper.group({
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      // Accessor Column
      columnHelper.accessor('age', {
        header: () => 'Age',
        footer: props => props.column.id,
      }),
      // Grouping Column
      columnHelper.group({
        header: 'More Info',
        columns: [
          // Accessor Column
          columnHelper.accessor('visits', {
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          }),
          // Accessor Column
          columnHelper.accessor('status', {
            header: 'Status',
            footer: props => props.column.id,
          }),
          // Accessor Column
```

```
          columnHelper.accessor('progress', {
            header: 'Profile Progress',
            footer: props => props.column.id,
          }),
        ],
      }),
    ],
  }),
]
```

(The duplicated code block is the same as above.)

Creating Accessor Columns

Data columns must be configured to extract primitive values for each item in your `data` array. There are 3 ways:

- If your items are **objects**, use a key that corresponds to the value you want to extract.
- If items are nested **arrays**, use an array index.
- Use an accessor function that returns the value.

Object Keys

If each item is an object:

```
columnHelper.accessor('firstName')

// OR

{
  accessorKey: 'firstName',
}
```

Array Indices

If each item is an array:

```
columnHelper.accessor(1)

// OR

{
  accessorKey: 1,
}
```

Accessor Functions

If each item is an object and you want to compute the value:

```
columnHelper.accessor(row => `${row.firstName} ${row.lastName}`, {
  id: 'fullName',
})

// OR
```

```
{
  id: 'fullName',
  accessorFn: row => `${row.firstName} ${row.lastName}`,
}
```

> 🧠 *Remember, the accessed value is used for sorting, filtering, etc. Ensure your accessor returns a primitive value for meaningful operations. Returning non-primitives like objects or arrays requires custom filter/sort functions or your own logic!*

Unique Column IDs

Columns are identified uniquely via:

- For object keys or array indices: the same key/index is used; periods (`.`) in object keys are replaced with underscores (`_`).
- For accessor functions:
  - The column's `id` property is used OR
  - If a string header is provided, it's used to identify the column.

> 🧠 *Remember: if you define an accessor with a function, provide either a string header or a unique* `id` *.*

Column Formatting & Rendering

Default is displaying the data model value as a string. You can override this with custom renderers that receive cell/header/footer info and return framework-specific rendering (JSX/components/strings). The formatter type (cell, aggregatedCell, header, footer) determines where it applies.

Cell Formatting

Custom cells:

```
columnHelper.accessor('firstName', {
  cell: props => <span>{props.getValue().toUpperCase()}</span>,
})
```

You can also access row and table objects:

```
columnHelper.accessor('firstName', {
  cell: props => (
    <span>{`${props.row.original.id} - ${props.getValue()}`}</span>
  ),
})
```

Aggregated Cell Formatting

For more info, see grouping.

Header & Footer Formatting

Headers and footers do not have row data but can still be customized similarly.

Edit on GitHub

On this page

- API
- Column Definitions Guide
- Column Def Types
- Column Helpers
- Creating Accessor Columns
- Object Keys
- Array Indices
- Accessor Functions
- Unique Column IDs
- Column Formatting & Rendering
- Cell Formatting
- Aggregated Cell Formatting
- Header & Footer Formatting

Table Instance Guide | TanStack Table Docs

Our Partners

Row Models Guide

Overview

If you take a look at the most basic example of TanStack Table, you'll see a code snippet like this:

```
import { getCoreRowModel, useReactTable } from '@tanstack/react-table'

function Component() {
  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(), // row model
  })
}
```

What is this `getCoreRowModel` function? And why do you have to import it from TanStack Table only to just pass it back to itself?

Well, the answer is that TanStack Table is a modular library. Not all code for every single feature is included in the createTable functions/hooks by default. You only need to import and include the code that you will need to correctly generate rows based on the features you want to use.

What are Row Models?

Row models run under the hood of TanStack Table to transform your original data in useful ways that are needed for data grid features like filtering, sorting, grouping, expanding, and pagination. The rows that get generated and render on screen won't necessarily be a 1:1 mapping of the original data that you passed to the table. They may be sorted, filtered, paginated, etc.

Import Row Models

You should only import the row models that you need. Here are all of the row models that are available:

```
// only import the row models you need
import {
  getCoreRowModel,
  getExpandedRowModel,
  getFacetedMinMaxValues,
  getFacetedRowModel,
  getFacetedUniqueValues,
  getFilteredRowModel,
  getGroupedRowModel,
  getPaginationRowModel,
  getSortedRowModel,
} // ...
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
  getFacetedMinMaxValues: getFacetedMinMaxValues(),
```

```
  getFacetedRowModel: getFacetedRowModel(),
  getFacetedUniqueValues: getFacetedUniqueValues(),
  getFilteredRowModel: getFilteredRowModel(),
  getGroupedRowModel: getGroupedRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  getSortedRowModel: getSortedRowModel(),
})
```

Customize/Fork Row Models

You don't have to use the exact row models that are provided by TanStack Table. If you need some advanced customization for certain row models, feel free to copy the source code for the row model you want to customize and modify it to your needs.

Using Row Models

Once your table instance has been created, you can access all of the row models that you may need directly from the table instance. There are even more derived row models available apart from the ones that you may have imported.

For normal rendering use cases, you will probably only need to use the `table.getRowModel()` method, as this row model will use all/any of the other row models depending on which features you have enabled or disabled. All of the other row models are available for you to "dig into" some of the underlying data transformations that are happening in the table.

Available Row Models on Table Instance

- `getRowModel` - This is the main row model that you should use for rendering your table rows markup. It will use all of the other row models to generate the final row model that you will use to render your table rows.
- `getCoreRowModel` - returns a basic row model that is just a 1:1 mapping of the original data passed to the table.
- `getFilteredRowModel` - returns a row model that accounts for column filtering and global filtering.
- `getPreFilteredRowModel` - returns a row model before column filtering and global filtering are applied.
- `getGroupedRowModel` - returns a row model that applies grouping and aggregation to the data and creates sub-rows.
- `getPreGroupedRowModel` - returns a row model before grouping and aggregation are applied.
- `getSortedRowModel` - returns a row model that has had sorting applied to it.
- `getPreSortedRowModel` - returns a row model before sorting is applied (rows are in original order).
- `getExpandedRowModel` - returns a row model that accounts for expanded/hidden sub-rows.
- `getPreExpandedRowModel` - returns a row model that only includes root level rows with no expanded sub-rows included. Still includes sorting.
- `getPaginationRowModel` - returns a row model that only includes the rows that should be displayed on the current page based on the pagination state.
- `getPrePaginationRowModel` - returns a row model without pagination applied (includes all rows).
- `getSelectedRowModel` - returns a row model of all selected rows (but only based on the data that was passed to the table). Runs after getCoreRowModel.
- `getPreSelectedRowModel` - returns a row model before row selection is applied (Just returns getCoreRowModel).
- `getGroupedSelectedRowModel` - returns a row model of selected rows after grouping. Runs after getSortedRowModel, which runs after getGroupedRowModel, which runs after getFilteredRowModel.
- `getFilteredSelectedRowModel` - returns a row model of selected rows after column filtering and global filtering. Runs after getFilteredRowModel.

The Order of Row Model Execution

Knowing how TanStack Table processes rows internally can help you gain a better understanding of what is happening under the hood, and help you debug any issues you may encounter.

Internally, this is the order in which each of the row models are applied to the data, if their respective features are enabled:

`getCoreRowModel` → `getFilteredRowModel` → `getGroupedRowModel` → `getSortedRowModel` → `getExpandedRowModel` → `getPaginationRowModel` → `getRowModel`

If in any case the respective feature is disabled or turned off with a `"manual*"` table option, the `getPre*RowModel` will be used instead in that step of the process.

As you can see above, first the data is filtered, then grouped, then sorted, then expanded, and then finally paginated as the final step.

Row Model Data Structure

Each row model will provide you the rows in 3 different useful formats:

1. **`rows`** - An array of rows.
2. **`flatRows`** - An array of rows, but all sub-rows are flattened into the top level.
3. **`rowsById`** - An object of rows, where each row is keyed by its `id`. This is useful for quickly looking up rows by their `id` with better performance.

```
console.log(table.getRowModel().rows) // array of rows
console.log(table.getRowModel().flatRows) // array of rows, but all sub-rows are flatt
console.log(table.getRowModel().rowsById['row-id']) // object of rows, where each row
```

Edit on GitHub

Edit on GitHub

On this page

Partners & Subscription

Our Partners

- React Data Grid - AG Grid

TanStack Router

- TanStack Router: A powerful React router for client-side and full-stack React applications. Fully type-safe APIs, first-class search-params for managing state in the URL, and seamless integration with the existing React ecosystem.

TanStack Ranger

- TanStack Ranger: Headless, lightweight, and extensible primitives for building range and multi-range sliders.

Subscribe to Bytes

Rows Guide

This quick guide will discuss the different ways you can retrieve and interact with row objects in TanStack Table.

API

Row API

Rows Guide

Where to Get Rows From

There are multiple `table` instance APIs you can use to retrieve rows from the table instance.

table.getRow

If you need to access a specific row by its `id`, you can use the `table.getRow` table instance API.

```
const row = table.getRow(rowId)
```

Row Models

The `table` instance generates `row` objects and stores them in useful arrays called Row Models. This is discussed in much more detail in the Row Models Guide, but here are the most common ways you may access the row models.

Render Rows

```
<tbody>
  {table.getRowModel().rows.map(row => (
    <tr key={row.id}>
     {/* ... */}
    </tr>
  ))}
</tbody>
```

Get Selected Rows

```
const selectedRows = table.getSelectedRowModel().rows
```

Row Objects

Every row object contains row data and many APIs to either interact with the table state or extract cells from the row based on the state of the table.

Row IDs

Every row object has an `id` property that makes it unique within the table instance. By default, `row.id` is the same as `row.index` created in the row model, but you can override it using the `getRowId` table option.

```
const table = useReactTable({
  columns,
  data,
  getRowId: originalRow => originalRow.uuid, // override the row.id with a unique iden
})
```

> **Note:** In features like grouping and expanding, the `row.id` will have additional string appended to it.

Access Row Values

The recommended way to access data values from a row is using `row.getValue` or `row.renderValue`. `row.renderValue` will return either the value or `renderFallbackValue` if undefined; `row.getValue` returns the value or `undefined`.

```
// Access data from columns
const firstName = row.getValue('firstName') // read value
const renderedLastName = row.renderValue('lastName') // render value
```

> **Note:** `cell.getValue` and `cell.renderValue` are shortcuts for `row.getValue` and `row.renderValue`.

Access Original Row Data

Access the original data passed to the table via `row.original`.

```
const firstName = row.original.firstName
```

Sub Rows

If using grouping or expanding, rows may contain sub-rows or parent references.

- `row.subRows`: array of sub-rows
- `row.depth`: nesting depth (0 for root)
- `row.parentId`: ID of parent row
- `row.getParentRow()`: returns parent row

More Row APIs

Depending on features, there are many more APIs for interacting with rows. Refer to specific feature docs.

Edit on GitHub

On this page

- Get Selected Rows
- Row Objects
- Row IDs
- Access Row Values
- Access Original Row Data
- Sub Rows
- More Row APIs

Cells Guide

API

Cell API

Cells Guide

This quick guide will discuss the different ways you can retrieve and interact with *cell* objects in TanStack Table.

Where to Get Cells From

Cells come from Rows. Enough said, right?

There are multiple *row* instance APIs you can use to retrieve the appropriate cells from a row depending on which features you are using. Most commonly, you will use the `row.getAllCells` or `row.getVisibleCells` APIs (if you are using column visibility features), but there are a handful of other similar APIs that you can use.

Cell Objects

Every cell object can be associated with a `<td>` or similar cell element in your UI. There are a few properties and methods on *cell* objects that you can use to interact with the table state and extract cell values from the table based on the state of the table.

Cell IDs

Every cell object has an `id` property that makes it unique within the table instance. Each `cell.id` is constructed simply as a union of its parent row and column IDs separated by an underscore.

```
{ id: `${row.id}_${column.id}` }
```

During grouping or aggregation features, the `cell.id` will have additional string appended to it.

Cell Parent Objects

Every cell stores a reference to its parent row and column objects.

Access Cell Values

The recommended way to access data values from a cell is to use either the `cell.getValue` or `cell.renderValue` APIs. Using either of these APIs will cache the results of the accessor functions and keep rendering efficient. The only difference between the two is that `cell.renderValue` will return either the value or the `renderFallbackValue` if the value is undefined, whereas `cell.getValue` will return the value or `undefined` if the value is undefined.

> **Note:** The `cell.getValue` and `cell.renderValue` APIs are shortcuts to `row.getValue` and `row.renderValue`, respectively.

```
// Access data from any of the columns
const firstName = cell.getValue('firstName') // read the cell value from the firstName
const renderedLastName = cell.renderValue('lastName') // render the value from the las
```

Access Other Row Data from Any Cell

Since every cell object is associated with its parent row, you can access any data from the original row that you are using in your table using `cell.row.original`.

```
// Even if we are in the scope of a different cell, we can still access the original r
const firstName = cell.row.original.firstName // { firstName: 'John', lastName: 'Doe'
```

More Cell APIs

Depending on the features that you are using for your table, there are dozens more useful APIs for interacting with cells. See each feature's respective API docs or guide for more information.

Cell Rendering

You can just use the `cell.renderValue` or `cell.getValue` APIs to render the cells of your table. However, these APIs will only output the raw cell values (from accessor functions). If you are using the `cell: () => JSX` column definition options, you will want to use the `flexRender` API utility from your adapter.

Using the `flexRender` API will allow the cell to be rendered correctly with any extra markup or JSX and it will call the callback function with the correct parameters.

```
import { flexRender } from '@tanstack/react-table'

const columns = [
  {
    accessorKey: 'fullName',
    cell: ({ cell, row }) => {
      return <div><strong>{row.original.firstName}</strong> {row.original.lastName}</d
    }
    //...
  }
]
//...
<tr>
  {row.getVisibleCells().map(cell => {
    return <td key={cell.id}>{flexRender(cell.column.columnDef.cell, cell.getContext()
  })}
</tr>
```

On this page

Additional references

- Rows
- Header Groups

Our Partners

- React Data Grid

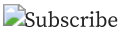
TanStack Query and TanStack DB

- TanStack Query Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state" *Learn More*

- TanStack DB TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.



No spam. Unsubscribe at *any* time.

Header Groups Guide

API

Header Group API

Header Groups Guide

This quick guide will discuss the different ways you can retrieve and interact with header group objects in TanStack Table.

What are Header Groups?

Header Groups are simply "rows" of headers. Don't let the name confuse you, it's just that simple. The large majority of tables will only have one row of headers (a single header group), but if you define your column structure with nested columns as with the Column Groups example, you can have multiple rows of headers (multiple header groups).

Where to Get Header Groups From

There are multiple **table** instance APIs you can use to retrieve header groups from the table instance. `table.getHeaderGroups()` is the most common API to use, but depending on the features that you are using, you may need to use other APIs, such as `table.get[Left/Center/Right]HeaderGroups()` if you are using column pinning features.

Header Group Objects

Header Group objects are similar to Row objects, though simpler since there is not as much going on in header rows as in body rows.

By default, header groups only have three properties:

- `id`: The unique identifier for the header group that is generated from its depth (index). Useful as a key for React components.
- `depth`: The depth of the header group, zero-indexed. Think of this as the row index amongst all header rows.
- `headers`: An array of Header cell objects that belong to this header group (row).

Access Header Cells

To render the header cells in a header group, you just map over the `headers` array from the header group object.

```
<thead>
  {table.getHeaderGroups().map(headerGroup => {
    return (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map(header => (
          <th key={header.id} colSpan={header.colSpan}>
            {/* Header content */}
          </th>
```

```
        ))}
      </tr>
    )
  })}
</thead>
```

Edit on GitHub

On this page

Headers Guide | TanStack Table Docs

API

Header API

Headers Guide

This quick guide will discuss the different ways you can retrieve and interact with *header* objects in TanStack Table.

Headers are the equivalent of cells, but meant for the `<thead>` section of the table instead of the `<tbody>` section.

Where to Get Headers From

Headers come from Header Groups, which are the equivalent of rows, but meant for the `<thead>` section of the table instead of the `<tbody>`.

HeaderGroup Headers

If you are in a header group, the headers are stored as an array in the `headerGroup.headers` property. Usually, you will just map over this array to render your headers.

```
<thead>
  {table.getHeaderGroups().map(headerGroup => {
    return (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map(header => ( // map over the headerGroup headers array
          <th key={header.id} colSpan={header.colSpan}>
            {/* */}
          </th>
        ))}
      </tr>
    )
  })}
</thead>
```

Header Table Instance APIs

There are multiple *table* instance APIs that you can use to retrieve a list of headers depending on the features you are using. The most common API you might use is `table.getFlatHeaders`, which will return a flat list of all headers in the table, but there are at least a dozen other headers useful with features like column visibility and pinning. APIs like `table.getLeftLeafHeaders` or `table.getRightFlatHeaders` could be useful depending on your use case.

Header Objects

Header objects are similar to Cell objects, but meant for the `<thead>` section of the table instead of the `<tbody>`. Every header object can be associated with a `<th>` or similar UI element. They contain properties and methods to interact with table state and extract cell values.

Header IDs

Each header object has a unique `id` property within the table. This `id` is used as a key in React or for column resizing.

- For simple headers, `header.id` is the same as the parent `column.id`.
- For grouped headers or placeholders, `id` is constructed from the header family, depth, column id, and header group id.

Nested Grouped Headers Properties

Properties useful for nested headers include:

- `colspan`: span across multiple columns.
- `rowSpan`: span across multiple rows (not yet implemented in default).
- `depth`: the row index of the header group.
- `isPlaceholder`: true if it's a placeholder header.
- `placeholderId`: unique id for placeholder headers.
- `subHeaders`: array of child headers; empty if leaf.

  Note: `header.index` indicates its position within the header row (from left), while `header.depth` indicates the row index in the header structure.

Header Parent Objects

Each header references its parent `column` and header group.

More Header APIs

Headers provide additional APIs for table state interaction, mainly related to sizing and resizing, as documented in the Column Sizing Guide.

Header Rendering

Use the `flexRender` utility to render headers, handling string, JSX, or function definitions:

```
{headerGroup.headers.map(header => (
  <th key={header.id} colSpan={header.colSpan}>
    {flexRender(header.column.columnDef.header, header.getContext())}
  </th>
))}
```

Edit on GitHub

On this page

- Header Parent Objects
- More Header APIs
- Header Rendering

Columns Guide

API

Columns Guide

> **Note:** This guide is about the actual *column* objects that are generated within the table instance and NOT about setting up the column definitions for your table.

This quick guide will discuss the different ways you can retrieve and interact with *column* objects in TanStack Table.

Where to Get Columns From

You can find the *column* objects in many places. They are often attached

Header and Cell Objects

Before you reach for one of the *table* instance APIs, consider if you actually need to retrieve either headers or cells instead of *columns*. If you are rendering out the markup for your table, you will most likely want to reach for the APIs that return headers or cells instead of columns. The column objects themselves are not really meant to render out the headers or cells, but the *header* and *cell* objects will contain references to these *column* objects from which they can derive the necessary information to render their UI.

```
const column = cell.column; // get column from cell
const column = header.column; // get column from header
```

Column Table Instance APIs

There are dozens of *table* instance APIs you can use to retrieve columns from the table instance. Which APIs you will use will depend entirely on which features you are using in your table and your use-case.

Get Column

If you need to just get a single column by its ID, you can use the `table.getColumn` API.

```
const column = table.getColumn('firstName');
```

Get Columns

The simplest column API is `table.getAllColumns`, which will return a list of all columns in the table. There are dozens of other column APIs that are affected by other features and the state of the table that come alongside this API, such as `table.getAllFlatColumns`, `table.getAllLeafColumns`, `getCenterLeafColumns`, `table.getLeftVisibleLeafColumns`, which are just some examples of other column APIs you might use in tandem with the column visibility or pinning features.

Column Objects

Column objects are not actually meant to be used to render out the table UI directly, so they are not associated 1-to-1 with any `<th>` or `<td>` elements in your table, but they contain a lot of useful properties and methods that you can use to interact with the table state.

Column IDs

Every column must have a unique *id* defined in their associated Column Definition. Usually, you define this *id* yourself, or it is derived from the `accessorKey` or `header` properties in the column definition.

ColumnDef

A reference to the original `columnDef` object that was used to create the column is always available on the column object.

Nested Grouped Columns Properties

There are a few properties on *column* objects that are only useful if the column is part of a nested or grouped column structure. These include:

- `columns`: An array of child columns that belong to a group column.
- `depth`: The header group "row index" that the column group belongs to.
- `parent`: The parent column of the column. If the column is a top-level column, this will be `undefined`.

More Column APIs

There are dozens of Column APIs you can use to interact with the table state and extract cell values based on the state of the table. See each feature's column API documentation for more information.

Column Rendering

Don't necessarily use *column* objects to render headers or cells directly. Instead, use the header and cell objects, as discussed above.

But if you're just rendering a list of columns elsewhere in your UI for a column visibility menu or similar, you can just map over a columns array and render the UI as usual.

Edit on GitHub

On this page

Headers

Link to Headers

Table State

Link to Table State

Our Partners

- React Data Grid
- TanStack Query
- TanStack DB

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

Table State (React) Guide | TanStack Table React Docs

On this page

Examples

Want to skip to the implementation? Check out these examples:

- kitchen sink
- fully controlled

Table State (React) Guide

TanStack Table has a simple underlying internal state management system to store and manage the state of the table. It also lets you selectively pull out any state that you need to manage in your own state management. This guide will walk you through the different ways in which you can interact with and manage the state of the table.

Accessing Table State

You do not need to set up anything special for the table state to work. If you pass nothing into either `state`, `initialState`, or any of the `on[State]Change` table options, the table will manage its own state internally. You can access any part of this internal state by using the `table.getState()` table instance API.

```
const table = useReactTable({
  columns,
  data,
  //...
})

console.log(table.getState()) // access the entire internal state
console.log(table.getState().rowSelection) // access just the row selection state
```

Custom Initial State

If all you need to do for certain states is customize their initial default values, you still do not need to manage any of the state yourself. You can simply set values in the `initialState` option of the table instance.

```
const table = useReactTable({
  columns,
  data,
  initialState: {
    columnOrder: ['age', 'firstName', 'lastName'], // customize the initial column ord
    columnVisibility: {
      id: false // hide the id column by default
    },
    expanded: true, // expand all rows by default
    sorting: [
      {
        id: 'age',
        desc: true // sort by age in descending order by default
      }
    ]
  },
  //...
})
```

> **Note:** Only specify each particular state in either `initialState` or `state`, but not both. If you pass in a particular state value to both `initialState` and `state`, the state in `state` will overwrite the value in `initialState`.

Controlled State

If you need easy access to the table state in other areas of your application, TanStack Table makes it easy to control and manage any or all of the table state in your own state management system. You can do this by passing in your own state and update functions to the `state` and `on[State]Change` options.

Individual Controlled State

You can control just the state you need easy access to. You do NOT have to control all table states if you don't want to. It's recommended to only control the state you need on a case-by-case basis.

To do this, pass in the specific `state` value and the corresponding `on[State]Change` function:

```
const [columnFilters, setColumnFilters] = React.useState([]) // no default filters
const [sorting, setSorting] = React.useState([
  { id: 'age', desc: true }
])
const [pagination, setPagination] = React.useState({ pageIndex: 0, pageSize: 15 })

const table = useReactTable({
  columns,
  data,
  //...
  state: {
    columnFilters, // pass controlled state
    sorting,
    pagination
  },
  onColumnFiltersChange: setColumnFilters,
  onSortingChange: setSorting,
```

```
  onPaginationChange: setPagination,
})
```

Fully Controlled State

Alternatively, you can control the entire table state with the `onStateChange` table option. This pulls out the entire state into your own state management system. Be cautious: raising frequently changing state values higher in your React component tree might cause performance issues.

To do this, initialize the state in your component, and use `table.setOptions()` to merge your state:

```
// create table instance without passing state
const table = useReactTable({
  columns,
  data,
  //... (no `state` here)
})

// create your controlled state
const [state, setState] = React.useState({
  ...table.initialState,
  pagination: {
    pageIndex: 0,
    pageSize: 15
  }
})

// Merge controlled state into table
table.setOptions(prev => ({
  ...prev,
  state,
  onStateChange: setState
}))
```

On State Change Callbacks

These options work to "hoist" table state changes into your own state management. However, there are important considerations:

1. State Change Callbacks MUST have their corresponding state value in the `state` option

Specifying an `on[State]Change` callback indicates that part of the state is controlled. If you do not provide the matching `state` value, that state will remain "frozen" at its initial value.

```
const [sorting, setSorting] = React.useState([])

const table = useReactTable({
  columns,
  data,
  state: {
    sorting, // required for controlled sorting
  },
```

```
  onSortingChange: setSorting,
})
```

2. Updaters can either be raw values or callback functions

The `on[State]Change` and `onStateChange` behave like React's `setState()`. The updater can be a new value or a callback that receives the previous state.

```
const [sorting, setSorting] = React.useState([])
const [pagination, setPagination] = React.useState({ pageIndex: 0, pageSize: 10 })

const table = useReactTable({
  columns,
  data,
  //...
  state: { pagination, sorting },
  // syntax 1
  onPaginationChange: (updater) => {
    setPagination(old => {
      const newValue = updater instanceof Function ? updater(old) : updater
      // do something with newValue
      return newValue
    })
  },
  // syntax 2
  onSortingChange: (updater) => {
    const newValue = updater instanceof Function ? updater(sorting) : updater
    // do something
    setSorting(newValue)
  }
})
```

State Types

All complex states in TanStack Table have corresponding TypeScript types, which you can import for type safety.

```
import { useReactTable, type SortingState } from '@tanstack/react-table'

// example
const [sorting, setSorting] = React.useState<SortingState[]>([
  {
    id: 'age',
    desc: true,
  }
])
```

Edit on GitHub

Column Ordering Guide

Examples

Want to skip to the implementation? Check out these examples:

- column-ordering
- column-dnd

API

Column Ordering API

Column Ordering Guide

By default, columns are ordered in the order they are defined in the **columns** array. However, you can manually specify the column order using the **columnOrder** state. Other features like column pinning and grouping can also affect the column order.

What Affects Column Order

There are 3 table features that can reorder columns, which happen in the following order:

1. Column Pinning - If pinning, columns are split into left, center (unpinned), and right pinned columns.
2. **Manual *Column Ordering*** — A manually specified column order is applied.
3. Grouping - If grouping is enabled, a grouping state is active, and **tableOptions.groupedColumnMode** is set to `'reorder'` or `'remove'`, then the grouped columns are reordered to the start of the column flow.

   Note: *columnOrder* state will only affect unpinned columns if used in conjunction with column pinning.

Column Order State

If you don't provide a **columnOrder** state, TanStack Table will just use the order of the columns in the **columns** array. However, you can provide an array of string column ids to the **columnOrder** state to specify the order of the columns.

Default Column Order

If all you need to do is specify the initial column order, you can just specify the **columnOrder** state in the **initialState** table option.

```
const table = useReactTable({
  //...
  initialState: {
    columnOrder: ['columnId1', 'columnId2', 'columnId3'],
  }
  //...
});
```

Note: If you are using the **state** table option to also specify the **columnOrder** state, the **initialState** will have no effect. Only specify particular states in either **initialState** or **state**, not both.

Managing Column Order State

If you need to dynamically change the column order, or set the column order after the table has been initialized, you can manage the **columnOrder** state just like any other table state.

```
const [columnOrder, setColumnOrder] = useState<string[]>(['columnId1', 'columnId2', 'c

//...

const table = useReactTable({
  //...
  state: {
    columnOrder,
  },
  onColumnOrderChange: setColumnOrder,
  //...
});
```

Reordering Columns

If the table has UI that allows the user to reorder columns, you can set up the logic like this:

```
const [columnOrder, setColumnOrder] = useState<string[]>(columns.map(c => c.id));

// depending on your DnD solution of choice, you may or may not need this state
const [movingColumnId, setMovingColumnId] = useState<string | null>(null);
const [targetColumnId, setTargetColumnId] = useState<string | null>(null);

// util function to splice and reorder the columnOrder array
const reorderColumn = <TData extends RowData>(
  movingColumnId: Column<TData>,
  targetColumnId: Column<TData>,
): string[] => {
  const newColumnOrder = [...columnOrder];
  newColumnOrder.splice(
    newColumnOrder.indexOf(targetColumnId),
    0,
    newColumnOrder.splice(newColumnOrder.indexOf(movingColumnId), 1)[0],
  );
  setColumnOrder(newColumnOrder);
  return newColumnOrder;
};

const handleDragEnd = (e: DragEvent) => {
  if (!movingColumnId || !targetColumnId) return;
  setColumnOrder(reorderColumn(movingColumnId, targetColumnId));
};

// use your DnD solution of choice
```

Drag and Drop Column Reordering Suggestions (React)

There are many ways to implement drag and drop features alongside TanStack Table. Here are some suggestions to make it easier:

1. **Do NOT** try to use react-dnd *if you are using React 18 or newer*. React DnD was important historically but now doesn't get updated often and has incompatibilities with React 18, especially in Strict Mode. It might interfere with other DnD solutions.

2. Use @dnd-kit/core. DnD Kit is a modern, modular, and lightweight library that works well with React and `<table>` markup. Both TanStack DnD examples and others largely now use DnD Kit.

3. Consider other libraries like react-beautiful-dnd, but be aware of their bundle sizes and maintenance status.

4. Use native browser drag-and-drop events for lightweight solutions. This approach may require extra work for touch devices. Material React Table V2 is an example that implements TanStack Table with browser native drag-and-drop events (`onDragStart` , `onDragEnd` , `onDragEnter`) without dependencies.

Edit on GitHub

On this page

Column Pinning Guide

Examples

Want to skip to the implementation? Check out these examples:

- column-pinning
- sticky-column-pinning

Other Examples

- Svelte column-pinning
- Vue column-pinning

API

Column Pinning API

Column Pinning Guide

TanStack Table offers state and APIs helpful for implementing column pinning features in your table UI. You can implement column pinning in multiple ways. You can either split pinned columns into their own separate tables, or you can keep all columns in the same table, but use the pinning state to order the columns correctly and use sticky CSS to pin the columns to the left or right.

How Column Pinning Affects Column Order

There are 3 table features that can reorder columns, which happen in the following order:

1. **Column Pinning** — If pinning, columns are split into left, center (unpinned), and right pinned columns.
2. Manual Column Ordering — A manually specified column order is applied.
3. Grouping — If grouping is enabled, a grouping state is active, and `tableOptions.groupedColumnMode` is set to `'reorder' | 'remove'`, then the grouped columns are reordered to the start of the column flow.

The only way to change the order of the pinned columns is in the `columnPinning.left` and `columnPinning.right` state itself. `columnOrder` state will only affect the order of the unpinned ("center") columns.

Column Pinning State

Managing the `columnPinning` state is optional, and usually not necessary unless you are adding persistent state features. TanStack Table will already keep track of the column pinning state for you. Manage the `columnPinning` state just like any other table state if you need to.

```
const [columnPinning, setColumnPinning] = useState<ColumnPinningState>({
  left: [],
  right: [],
});
//...
const table = useReactTable({
```

```
  //...
  state: {
    columnPinning,
    //...
  },
  onColumnPinningChange: setColumnPinning,
  //...
});
```

Pin Columns by Default

A very common use case is to pin some columns by default. You can do this by either initializing the `columnPinning` state with the pinned columnIds, or by using the `initialState` table option.

```
const table = useReactTable({
  //...
  initialState: {
    columnPinning: {
      left: ['expand-column'],
      right: ['actions-column'],
    },
    //...
  },
  //...
});
```

Useful Column Pinning APIs

Note: Some of these APIs are new in v8.12.0

There are a handful of useful Column API methods to help you implement column pinning features:

- `column.getCanPin`: Use to determine if a column can be pinned.
- `column.pin`: Use to pin a column to the left or right. Or use to unpin a column.
- `column.getIsPinned`: Use to determine where a column is pinned.
- `column.getStart`: Use to provide the correct `left` CSS value for a pinned column.
- `column.getAfter`: Use to provide the correct `right` CSS value for a pinned column.
- `column.getIsLastColumn`: Use to determine if a column is the last in its pinned group. Useful for adding a box-shadow.
- `column.getIsFirstColumn`: Use to determine if a column is the first in its pinned group. Useful for adding a box-shadow.

Split Table Column Pinning

If you are just using sticky CSS to pin columns, you can for the most part, just render the table as you normally would with the `table.getHeaderGroups` and `row.getVisibleCells` methods.

However, if you are splitting up pinned columns into their own separate tables, you can make use of the `table.getLeftHeaderGroups`, `table.getCenterHeaderGroups`, `table.getRightHeaderGroups`, `row.getLeftVisibleCells`, `row.getCenterVisibleCells`, and `row.getRightVisibleCells` methods to only render the columns that are relevant to the current table.

Edit on GitHub

On this page

Expanding Guide

On this page

Examples

Want to skip to the implementation? Check out these examples:

- Expanding
- Grouping
- Sub-components

API

Expanding API

Expanding Feature Guide

Expanding is a feature that allows you to show and hide additional rows of data related to a specific row. This can be useful in cases where you have hierarchical data and want to allow users to drill down into the data from a higher level. Or it can be useful for showing additional information related to a row.

Different use cases for Expanding Features

There are multiple use cases for expanding features in TanStack Table that will be discussed below.

1. Expanding sub-rows (child rows, aggregate rows, etc.)
2. Expanding custom UI (detail panels, sub-tables, etc.)

Enable Client-Side Expanding

To use the client-side expanding features, you need to define the `getExpandedRowModel` function in your table options. This function is responsible for returning the expanded row model.

```
const table = useReactTable({
  // other options...
  getExpandedRowModel: getExpandedRowModel(),
})
```

Expanded data can either contain table rows or any other data you want to display. We will discuss how to handle both cases in this guide.

Table rows as expanded data

Expanded rows are essentially child rows that inherit the same column structure as their parent rows. If your data object already includes these expanded rows data, you can utilize the `getSubRows` function to specify these child rows. If your data object does not contain the expanded rows data, they can be treated as custom expanded data, discussed in the next section.

For example, if you have a data object like this:

```
type Person = {
  id: number
  name: string
  age: number
  children?: Person[] | undefined
}

const data: Person[] = [
  {
    id: 1,
    name: 'John',
    age: 30,
    children: [
      { id: 2, name: 'Jane', age: 5 },
      { id: 5, name: 'Jim', age: 10 }
    ]
  },
  {
    id: 3,
    name: 'Doe',
    age: 40,
    children: [
      { id: 4, name: 'Alice', age: 10 }
    ]
  }
]
```

Then you can use the `getSubRows` function:

```
const table = useReactTable({
  // other options...
  getSubRows: (row) => row.children, // return the children array as sub-rows
  getCoreRowModel: getCoreRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
})
```

Note:
You can have a complicated `getSubRows` function, but keep in mind that it will run for every row and every sub-row. This can be expensive if not optimized. Async functions are not supported.

Custom Expanding UI

In some cases, you may want to show extra details or information, which may or may not be part of your table data object, such as expanded data for rows. This UI has gone by many names over the years including "expandable rows", "detail panels", "sub-components", etc.

By default, the `row.getCanExpand()` row instance API will return false unless it finds `subRows` on a row. This can be overridden by implementing your own `getRowCanExpand` function:

```
const table = useReactTable({
  // other options...
  getRowCanExpand: (row) => true, // All rows can expand
  getCoreRowModel: getCoreRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
})
```

And in rendering:

```
<tbody>
  {table.getRowModel().rows.map((row) => (
    <React.Fragment key={row.id}>
      {/* Normal row UI */}
      <tr>
        {row.getVisibleCells().map((cell) => (
          <td key={cell.id}>
            <FlexRender
              render={cell.column.columnDef.cell}
              props={cell.getContext()}
            />
          </td>
        ))}
      </tr>
      {/* Expanded row UI */}
      {row.getIsExpanded() && (
        <tr>
          <td colSpan={row.getAllCells().length}>
            {/* Your custom expanded UI goes here */}
          </td>
        </tr>
      )}
    </React.Fragment>
  ))}
</tbody>
```

Expanded rows state

If you need to control the expanded state of the rows, you can use the `expanded` state and `onExpandedChange` options:

```
const [expanded, setExpanded] = useState<ExpandedState>({})

const table = useReactTable({
  // other options...
  state: {
    expanded: expanded,
  },
```

```
  onExpandedChange: setExpanded,
})
```

The `ExpandedState` type:

```
type ExpandedState = true | Record<string, boolean>
```

- If `true`, all rows are expanded.
- If record, only rows with IDs as keys and true as value are expanded.

E.g., `{ row1: true, row2: false }` means `row1` is expanded, `row2` is not.

UI toggling handler for expanded rows

TanStack Table does not automatically add toggling UI; you should add controls within your row UI:

```
const columns = [
  {
    accessorKey: 'name',
    header: 'Name',
  },
  {
    accessorKey: 'age',
    header: 'Age',
  },
  {
    header: 'Children',
    cell: ({ row }) => {
      return row.getCanExpand() ? (
        <button
          onClick={row.getToggleExpandedHandler()}
          style={{ cursor: 'pointer' }}
        >
          {row.getIsExpanded() ? '👇' : '👉'}
        </button>
      ) : null;
    },
  },
]
```

Filtering Expanded Rows

Filtering starts from parent rows downward by default. If a parent is excluded, its children are excluded too.

To change this, enable `filterFromLeafRows`:

```
const table = useReactTable({
  // other options...
  getSubRows: (row) => row.subRows,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
  filterFromLeafRows: true,
  maxLeafRowFilterDepth: 1,
})
```

Paginating Expanded Rows

By default, expanded rows are paginated along with the table. To keep expanded rows always on their parent page, disable pagination:

```
const table = useReactTable({
  // other options...
  paginateExpandedRows: false,
})
```

Pinning Expanded Rows

Pin expanded rows to top or bottom, similar to regular rows. See Pinning Guide.

Sorting Expanded Rows

By default, expanded rows are sorted with the table.

Manual Expanding (server-side)

For server-side expansion, enable `manualExpanding`:

```
const table = useReactTable({
  // other options...
  manualExpanding: true,
})
```

This prevents `getExpandedRowModel` from running and expects manual handling of expansion.

Edit on GitHub

Edit this page on GitHub

Pagination Guide | TanStack Table Docs

Our Partners

On this page

Examples

Want to skip to the implementation? Check out these examples:

- Pagination
- Pagination Controlled (React Query)

- Editable Data
- Expanding
- Filters
- Fully Controlled
- Row Selection

API

Pagination API

Pagination Guide

TanStack Table has great support for both client-side and server-side pagination. This guide will walk you through the different ways to implement pagination in your table.

Client-Side Pagination

Using client-side pagination means that the *data* you fetch will contain **ALL** of the rows for the table, and the table instance will handle pagination logic in the front-end.

Should You Use Client-Side Pagination?

Client-side pagination is usually the simplest way to implement pagination when using TanStack Table, but it might not be practical for very large datasets.

However, a lot of people underestimate just how much data can be handled client-side. If your table will only ever have a few thousand rows or less, client-side pagination can still be viable. TanStack Table is designed to scale up to 10s of thousands of rows with decent performance for pagination, filtering, sorting, and grouping. The official pagination example loads 100,000 rows and still performs well, albeit with only a handful of columns.

Every use-case is different and will depend on the complexity of the table, how many columns you have, how large each piece of data is, etc. The main bottlenecks to consider are:

1. Can your server query all of the data in a reasonable time (and cost)?
2. What is the total size of the fetch? (This might not scale badly if you don't have many columns.)
3. Is the client's browser using too much memory if all data is loaded at once?

If unsure, you can start with client-side pagination and switch to server-side as your data grows.

Should You Use Virtualization Instead?

Alternatively, instead of paginating the data, render all rows of a large dataset on the same page, but only render rows visible in the viewport. This is called "virtualization" or "windowing". TanStack offers a virtualization library called TanStack Virtual that works well with TanStack Table. Both strategies have trade-offs; choose what best fits your use-case.

Pagination Row Model

To utilize built-in client-side pagination, pass the pagination row model:

```
import { useReactTable, getCoreRowModel, getPaginationRowModel } from '@tanstack/react

const table = useReactTable({
```

```
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(), // load client-side pagination
});
```

Manual Server-Side Pagination

For server-side pagination, you don't need a pagination row model, but if your table shares components with others that do, disable client-side pagination:

```
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  manualPagination: true, // turn off client-side pagination
  rowCount: dataQuery.data?.rowCount, // total row count
});
```

Note: When using `manualPagination`, pass `rowCount` or `pageCount` so the table knows total pages. If total pages are unknown, set `pageCount` to `-1`.

```
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  manualPagination: true,
  rowCount: dataQuery.data?.rowCount,
});
```

> **Important:** Setting `manualPagination` to `true` tells the table the *data* is already paginated.

Pagination State

Manage pagination state like any other state:

```
import { useState } from 'react';
import { useReactTable, getCoreRowModel, getPaginationRowModel } from '@tanstack/react

const [pagination, setPagination] = useState({
  pageIndex: 0,
  pageSize: 10,
});

const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  onPaginationChange: setPagination,
  state: {
    pagination,
  },
});
```

Alternatively, set initial state without controlling:

```
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  initialState: {
    pagination: {
      pageIndex: 2,
      pageSize: 25,
    },
  },
});
```

> **Note:** Do NOT pass `pagination` both to `state` and `initialState`; `state` takes precedence.

Pagination Options

Besides `manualPagination`, `pageCount`, and `rowCount`, consider `autoResetPageIndex`.

Auto Reset Page Index

By default, `pageIndex` resets to `0` on data updates, filtering, grouping, etc. This can be disabled via:

```
const table = useReactTable({
  column,
  data,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  autoResetPageIndex: false,
});
```

You may need to handle resetting pageIndex manually if turning off this behavior.

Pagination APIs

Useful APIs for pagination controls:

- `getCanPreviousPage()`
- `getCanNextPage()`
- `previousPage()`
- `nextPage()`
- `firstPage()`
- `lastPage()`
- `setPageIndex()`
- `resetPageIndex()`
- `setPageSize()`
- `resetPageSize()`
- `setPagination()`
- `resetPagination()`

Note: Some APIs are new in `v8.13.0`.

```
<Button onClick={() => table.firstPage()} disabled={!table.getCanPreviousPage()}>
  {'<<'}
</Button>
<Button onClick={() => table.previousPage()} disabled={!table.getCanPreviousPage()}>
```

```
  {'<'}
</Button>
<Button onClick={() => table.nextPage()} disabled={!table.getCanNextPage()}>
  {'>'}
</Button>
<Button onClick={() => table.lastPage()} disabled={!table.getCanNextPage()}>
  {'>>'}
</Button>
<select
  value={table.getState().pagination.pageSize}
  onChange={(e) => {
    table.setPageSize(Number(e.target.value))
  }}
>
  {[10, 20, 30, 40, 50].map((pageSize) => (
    <option key={pageSize} value={pageSize}>{pageSize}</option>
  ))}
</select>
```

Pagination Info APIs

- `getPageCount()` : total pages.
- `getRowCount()` : total rows.

Row Pinning Guide | TanStack Table Docs

Navigation

On this page

Examples

Want to skip to the implementation? Check out these examples:

- row-pinning

API

Row Pinning API

Row Pinning Guide

There are 2 table features that can reorder rows, which happen in the following order:

1. **Row Pinning** - If pinning, rows are split into top, center (unpinned), and bottom pinned rows.
2. Sorting

Additional Content

Edit on GitHub

Partner and Subscription Information

Our Partners

Resources

- TanStack Query
  Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state"
  Learn More

- TanStack DB
  TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥
  Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

-  subscribe figure
- **Subscribe**
- No spam. Unsubscribe at *any* time.

Row Selection Guide

Examples

Want to skip to the implementation? Check out these examples:

- React row-selection
- Vue row-selection
- React expanding

API

Row Selection API

Row Selection Guide

The row selection feature keeps track of which rows are selected and allows you to toggle the selection of rows in a myriad of ways. Let's take a look at some common use cases.

Access Row Selection State

The table instance already manages the row selection state for you (though as seen down below, it may be more convenient to manage the row selection state in your own scope). You can access the internal row selection state or the selected rows from a few APIs.

- `getState().rowSelection` - returns the internal row selection state
- `getSelectedRowModel()` - returns selected rows
- `getFilteredSelectedRowModel()` - returns selected rows after filtering
- `getGroupedSelectedRowModel()` - returns selected rows after grouping and sorting

```
console.log(table.getState().rowSelection); // get the row selection state - e.g., { 1
console.log(table.getSelectedRowModel().rows); // get full client-side selected rows
console.log(table.getFilteredSelectedRowModel().rows); // get filtered client-side sel
console.log(table.getGroupedSelectedRowModel().rows); // get grouped client-side selec
```

> **Note:** If you are using **manualPagination**, the `getSelectedRowModel()` API will only return selected rows on the current page because table row models can only generate rows based on the `data` that is passed in. Row selection state, however, can contain row ids that are not present in the `data` array just fine.

Manage Row Selection State

Even though the table instance will already manage the row selection state for you, it is usually more convenient to manage the state yourself in order to have easy access to the selected row ids that you can use to make API calls or other actions.

Use the `onRowSelectionChange` table option to hoist up the row selection state to your own scope. Then pass the row selection state back to the table instance using the `state` table option.

```
const [rowSelection, setRowSelection] = useState({}); // manage your own row selection

const table = useReactTable({
```

```
  //...
  onRowSelectionChange: setRowSelection, //hoist up the row selection state
  state: {
    rowSelection, // pass the row selection state back to the table
  },
});
```

Useful Row Ids

By default, the row id for each row is simply `row.index`. If you are using row selection features, you most likely want to use a more useful row identifier, since the row selection state is keyed by row id. You can use the `getRowId` table option to specify a function that returns a unique row id for each row.

```
const table = useReactTable({
  //...
  getRowId: row => row.uuid, // use the row's uuid from your database as the row id
});
```

Now as rows are selected, the row selection state will look something like this:

```
{
  "13e79140-62a8-4f9c-b087-5da737903b76": true,
  "f3e2a5c0-5b7a-4d8a-9a5c-9c9b8a8e5f7e": false
  //...
}
```

instead of this:

```
{
  "0": true,
  "1": false
  //...
}
```

Enable Row Selection Conditionally

Row selection is enabled by default for all rows. To either enable row selection conditionally for certain rows or disable row selection for all rows, you can use the `enableRowSelection` table option, which accepts either a boolean or a function for more granular control.

```
const table = useReactTable({
  //...
  enableRowSelection: row => row.original.age > 18, // only enable for adults
});
```

To enforce whether a row is selectable in your UI, you can use the `row.getCanSelect()` API for your checkboxes or other selection UI.

Single Row Selection

By default, the table allows multiple rows to be selected at once. If you want to only allow a single row to be selected, set `enableMultiRowSelection` to `false`.

```
const table = useReactTable({
  //...
  enableMultiRowSelection: false, // only allow a single row to be selected
});
```

This is useful for making tables with radio buttons instead of checkboxes.

Sub-Row Selection

Selecting a parent row will select all of its sub-rows by default. To disable auto sub-row selection, set `enableSubRowSelection` to `false`, or pass a function to disable it conditionally.

```
const table = useReactTable({
  //...
  enableSubRowSelection: false, // disable sub-row selection
});
```

Render Row Selection UI

TanStack table does not dictate how you should render your row selection UI. You can use checkboxes, radio buttons, or simply hook up click events to the row itself.

Connect Row Selection APIs to Checkbox Inputs

TanStack Table provides handler functions to connect directly to your checkbox inputs for toggling row selection:

- `row.getToggleSelectedHandler()` - toggle a row's selection
- `table.getToggleAllRowsSelectedHandler()` / `table.getToggleAllPageRowsSelectedHandler()` - toggle all rows

```
const columns = [
  {
    id: 'select-col',
    header: ({ table }) => (
      <Checkbox
        checked={table.getIsAllRowsSelected()}
        indeterminate={table.getIsSomeRowsSelected()}
        onChange={table.getToggleAllRowsSelectedHandler()}
      />
    ),
    cell: ({ row }) => (
      <Checkbox
        checked={row.getIsSelected()}
        disabled={!row.getCanSelect()}
        onChange={row.getToggleSelectedHandler()}
      />
    ),
  },
  // more columns...
];
```

Connect Row Selection APIs to UI

For a simpler UI, hook click events directly onto the row:

```
<tbody>
  {table.getRowModel().rows.map(row => (
    <tr
      key={row.id}
      className={row.getIsSelected() ? 'selected' : null}
      onClick={row.getToggleSelectedHandler()}
```

```
    >
      {row.getVisibleCells().map(cell => (
        <td key={cell.id}>{/* cell content */}</td>
      ))}
    </tr>
  ))}
</tbody>
```

Edit on GitHub

On this page

Sorting Guide | TanStack Table Docs

On this page

Examples

Want to skip to the implementation? Check out these examples:

- sorting
- filters

API

Sorting API

Sorting Guide

TanStack Table provides solutions for just about any sorting use-case you might have. This guide will walk you through the various options that you can use to customize the built-in client-side sorting functionality, as well as how to opt out of client-side sorting in favor of manual server-side sorting.

Sorting State

The sorting state is defined as an array of objects with the following shape:

```
type ColumnSort = {
  id: string
  desc: boolean
}
type SortingState = ColumnSort[]
```

Since the sorting state is an array, it is possible to sort by multiple columns at once. Read more about the multi-sorting customizations down below.

Accessing Sorting State

You can access the sorting state directly from the table instance just like any other state using the `table.getState()` API.

```
const table = useReactTable({
  columns,
  data,
  //...
})

console.log(table.getState().sorting) // access the sorting state from the table insta
```

However, if you need to access the sorting state before the table is initialized, you can "control" the sorting state like below.

Controlled Sorting State

If you need easy access to the sorting state, you can control/manage the sorting state in your own state management with the `state.sorting` and `onSortingChange` table options.

```
const [sorting, setSorting] = useState<SortingState>([]); // can set initial sorting s
//...
// use sorting state to fetch data from your server or something...
const table = useReactTable({
  columns,
  data,
  //...
  state: {
    sorting,
  },
  onSortingChange: setSorting,
})
```

Initial Sorting State

If you do not need to control the sorting state in your own state management or scope, but you still want to set an initial sorting state, you can use the `initialState` table option instead of `state`.

```
const table = useReactTable({
  columns,
  data,
  //...
  initialState: {
    sorting: [
      {
        id: 'name',
```

```
        desc: true, // sort by name in descending order by default
      },
    ],
  },
})
```

> **NOTE:** Do not use both `initialState.sorting` and `state.sorting` at the same time, as the initialized state in `state.sorting` will override the `initialState.sorting`.

Client-Side vs Server-Side Sorting

Whether or not you should use client-side or server-side sorting depends entirely on whether you are also using client-side or server-side pagination or filtering. Be consistent, because using client-side sorting with server-side pagination or filtering will only sort the data that is currently loaded, and not the entire dataset.

Manual Server-Side Sorting

If you plan to just use your own server-side sorting in your back-end logic, you do not need to provide a sorted row model. But if you have provided a sorting row model, and want to disable it, you can use the `manualSorting` table option.

```
const [sorting, setSorting] = useState<SortingState>([])
//...
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  //getSortedRowModel: getSortedRowModel(), //not needed for manual sorting
  manualSorting: true, //use pre-sorted row model instead of sorted row model
  state: {
    sorting,
  },
  onSortingChange: setSorting,
})
```

> **NOTE:** When `manualSorting` is set to `true`, the table will assume that the data that you provide is already sorted, and will not apply any sorting to it.

Client-Side Sorting

To implement client-side sorting, you first need to provide a sorting row model to the table. Import the `getSortedRowModel` function from TanStack Table, and it will be used to transform your rows into sorted rows.

```
import { useReactTable } from '@tanstack/react-table'
//...
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(), //provide a sorting row model
})
```

Sorting Fns

The default sorting function for all columns is inferred from the data type of the column. However, it can be useful to define the exact sorting function that you want to use for a specific column, especially if any of your data is nullable or not a standard data type.

You can determine a custom sorting function on a per-column basis using the `sortingFn` column option.

By default, there are 6 built-in sorting functions to choose from:

- **alphanumeric** - Sorts by mixed alphanumeric values without case-sensitivity. Slower, but more accurate if your strings contain numbers that need to be naturally sorted.
- **alphanumericCaseSensitive** - Sorts by mixed alphanumeric values with case-sensitivity. Slower, but more accurate if your strings contain numbers that need to be naturally sorted.
- **text** - Sorts by text/string values without case-sensitivity. Faster, but less accurate if your strings contain numbers that need to be naturally sorted.
- **textCaseSensitive** - Sorts by text/string values with case-sensitivity. Faster, but less accurate if your strings contain numbers that need to be naturally sorted.
- **datetime** - Sorts by time, use this if your values are `Date` objects.
- **basic** - Sorts using a basic/standard `a > b ? 1 : a < b ? -1 : 0` comparison. This is the fastest sorting function, but may not be the most accurate.

You can also define your own custom sorting functions either as the `sortingFn` column option, or as a global sorting function using the `sortingFns` table option.

Custom Sorting Functions

When defining a custom sorting function in either the `sortingFns` table option or as a `sortingFn` column option, it should have the following signature:

```
// optionally use the SortingFn to infer the parameter types
const myCustomSortingFn: SortingFn<TData> = (rowA: Row<TData>, rowB: Row<TData>, colum
  return // -1, 0, or 1 - access any row data using rowA.original and rowB.original
}
```

> **Note:** The comparison function does not need to consider whether the sorting is descending or ascending; the row models will handle that. `sortingFn` functions only need to provide a consistent comparison.

Every sorting function receives 2 rows and a column ID and is expected to compare the two rows using the column ID to return **-1**, **0**, or **1** in ascending order. Here's a cheat sheet:

Return	Ascending Order
-1	`a < b`
0	`a === b`
1	`a > b`

```
const columns = [
  {
    header: () => 'Name',
    accessorKey: 'name',
    sortingFn: 'alphanumeric', // use built-in sorting function by name
  },
  {
    header: () => 'Age',
    accessorKey: 'age',
    sortingFn: 'myCustomSortingFn', // use custom global sorting function
  },
  {
```

```
    header: () => 'Birthday',
    accessorKey: 'birthday',
    sortingFn: 'datetime', // recommended for date columns
  },
  {
    header: () => 'Profile',
    accessorKey: 'profile',
    // use custom sorting function directly
    sortingFn: (rowA, rowB, columnId) => {
      return rowA.original.someProperty - rowB.original.someProperty
    },
  }
]
//...
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  sortingFns: { // add a custom sorting function
    myCustomSortingFn: (rowA, rowB, columnId) => {
      return rowA.original[columnId] > rowB.original[columnId] ? 1 : rowA.original[col
    },
  },
})
```

Customize Sorting

There are many table and column options that you can use to further customize the sorting UX and behavior.

Disable Sorting

You can disable sorting for a specific column or the entire table using the `enableSorting` column option or table option.

```
const columns = [
  {
    header: () => 'ID',
    accessorKey: 'id',
    enableSorting: false, // disable sorting for this column
  },
  {
    header: () => 'Name',
    accessorKey: 'name',
  },
  //...
]
//...
const table = useReactTable({
  columns,
  data,
  enableSorting: false, // disable sorting for the entire table
})
```

Sorting Direction

By default, the first sorting direction when cycling through the sorting for a column with the `toggleSorting` API is ascending for string columns and descending for number columns. You can change this using the `sortDescFirst` column option or table option.

```
const columns = [
  {
    header: () => 'Name',
    accessorKey: 'name',
    sortDescFirst: true, // sort by name in descending order first (default is ascendi
  },
  {
    header: () => 'Age',
    accessorKey: 'age',
    sortDescFirst: false, // sort by age in ascending order first (default is descendi
  },
  //...
]

//...
const table = useReactTable({
  columns,
  data,
  sortDescFirst: true, // apply to all columns
})
```

> **NOTE:** You may want to explicitly set the `sortDescFirst` option on columns that have nullable values. The table may not correctly determine if a column is a number or string if it contains nullable entries.

Invert Sorting

Inverting sorting is not the same as changing the default sorting direction. If the `invertSorting` column option is `true` for a column, the "desc/asc" states still cycle normally, but the actual order of rows will be inverted. This is useful for values with an inverted scale, like rankings or certain scores.

```
const columns = [
  {
    header: () => 'Rank',
    accessorKey: 'rank',
    invertSorting: true, // invert the sorting for this column
  },
  //...
]
```

Sort Undefined Values

Any undefined values will be sorted to the beginning or end of the list based on the `sortUndefined` column or table option. You can customize this for your needs.

Default `sortUndefined` values:

- `'first'` - Undefined values are pushed to the beginning
- `'last'` - Undefined values are pushed to the end
- `false` - Undefined values are considered tied and will be sorted by the next column or index
- `-1` - Undefined values sorted with higher priority (appear first if ascending)
- `1` - Undefined values sorted with lower priority (appear last if ascending)

```
const columns = [
  {
    header: () => 'Rank',
    accessorKey: 'rank',
    sortUndefined: -1, // 'first' | 'last' | 1 | -1 | false
  },
]
```

> **NOTE:** `'first'` and `'last'` options are new in v8.16.0.

Sorting Removal

By default, the ability to remove sorting when cycling through states is enabled. You can disable this behavior using `enableSortingRemoval`.

```
const table = useReactTable({
  columns,
  data,
  enableSortingRemoval: false, // always cycle through 'none' -> 'desc' -> 'asc'
})
```

Once sorting is enabled, toggling cycles like `'none'` → `'desc'` → `'asc'` → `'none'`...

If `enableSortingRemoval` is `false`, toggling will cycle like `'none'` → `'desc'` → `'asc'` and stay on `'asc'` until changed explicitly.

> Set **`enableSortingRemoval`** to **`false`** if you want at least one column always sorted.

Multi-Sorting

Sorting by multiple columns is enabled by default when using the `column.getToggleSortingHandler` API. Holding `Shift` while clicking on a header adds that column to the sort.

Use the `column.toggleSorting(desc, multi)` API to control whether multi-sorting is used explicitly.

Disable Multi-Sorting

Use the `enableMultiSort` column or table option.

```
const columns = [
  {
    header: () => 'Created At',
    accessorKey: 'createdAt',
    enableMultiSort: false, // only sort by this column
  },
  //...
]

const table = useReactTable({
  columns,
  data,
  enableMultiSort: false, // disable multi-sorting globally
})
```

Customize Multi-Sorting Trigger

By default, `Shift` triggers multi-sorting. You can change this with `isMultiSortEvent`:

```
const table = useReactTable({
  columns,
```

```
  data,
  isMultiSortEvent: (e) => true, // all clicks trigger multi-sorting
  // or
  isMultiSortEvent: (e) => e.ctrlKey || e.shiftKey, // use `Ctrl` or `Shift`
})
```

Multi-Sorting Limit

Limit the number of simultaneously sorted columns:

```
const table = useReactTable({
  columns,
  data,
  maxMultiSortColCount: 3, // only allow 3 columns sorted at once
})
```

Multi-Sorting Removal

Disable removing multi-sorts:

```
const table = useReactTable({
  columns,
  data,
  enableMultiRemove: false, // cannot remove multi-sorts once set
})
```

Sorting APIs

APIs for sorting control and interaction:

- `table.setSorting` - set sorting state directly.
- `table.resetSorting` - reset or clear sorting.
- `column.getCanSort` - enable/disable sorting UI.
- `column.getIsSorted` - show sorting indicator.
- `column.getToggleSortingHandler` - hook up to UI components.
- `column.toggleSorting` - toggle sorting manually.
- `column.clearSorting` - clear sorting for the column.
- `column.getNextSortingOrder` - preview next sort direction.
- `column.getFirstSortDir` - initial sort direction.
- `column.getAutoSortDir` - auto-determined first direction.
- `column.getAutoSortingFn` - internal default sorting function.
- `column.getSortingFn` - current sorting function.
- `column.getCanMultiSort` - enable multi-sort UI.
- `column.getSortIndex` - position in multi-sort order.

Edit on GitHub

Virtualization Guide | TanStack Table Docs

Examples

Want to skip to the implementation? Check out these examples:

- virtualized-columns
- virtualized-rows (dynamic row height)
- virtualized-rows (fixed row height)
- virtualized-infinite-scrolling

Virtualization Guide

The TanStack Table packages do not come with any virtualization APIs or features built-in, but TanStack Table can easily work with other virtualization libraries like react-window or TanStack's own TanStack Virtual. This guide will show some strategies for using TanStack Table with TanStack Virtual.

Edit on GitHub

On this page

Additional Sections

Our Partners

Subscribe to Bytes

Custom Features Guide

Examples

Want to skip to the implementation? Check out these examples:

- custom-features

Custom Features Guide

In this guide, we'll cover how to extend TanStack Table with custom features, and along the way, we'll learn more about how the TanStack Table v8 codebase is structured and how it works.

TanStack Table Strives to be Lean

TanStack Table has a core set of features that are built into the library such as sorting, filtering, pagination, etc. We've received a lot of requests and sometimes even some well thought out PRs to add even more features to the library. While we are always open to improving the library, we also want to make sure that TanStack Table remains a lean library that does not include too much bloat and code that is unlikely to be used in most use cases. Not every PR can, or should, be accepted into the core library, even if it does solve a real problem. This can be frustrating to developers where TanStack Table solves 90% of their use case, but they need a little bit more control.

TanStack Table has always been built in a way that allows it to be highly extensible (at least since v7). The *table* instance that is returned from whichever framework adapter that you are using (`useReactTable` , `useVueTable` , etc) is a plain JavaScript object that can have extra properties or APIs added to it. It has always been possible to use composition to add custom logic, state, and APIs to the table instance. Libraries like Material React Table have simply created custom wrapper hooks around the `useReactTable` hook to extend the table instance with custom functionality.

However, starting in version 8.14.0, TanStack Table has exposed a new `_features` table option that allows you to more tightly and cleanly integrate custom code into the table instance in exactly the same way that the built-in table features are already integrated.

> **TanStack Table v8.14.0** introduced a new `_features` option that allows you to add custom features to the table instance.

With this new tighter integration, you can easily add more complex custom features to your tables, and possibly even package them up and share them with the community. We'll see how this evolves over time. In a future v9 release, we may even lower the bundle size of TanStack Table by making all features opt-in, but that is still being explored.

How TanStack Table Features Work

TanStack Table's source code is arguably somewhat simple (at least we think so). All code for each feature is split up into its own object/file with instantiation methods to create initial state, default table and column options, and API methods that can be added to the `table` , `header` , `column` , `row` , and `cell` instances.

All of the functionality of a feature object can be described with the `TableFeature` type that is exported from TanStack Table. This type is a TypeScript interface that describes the shape of a feature object needed to create a feature.

```
export interface TableFeature<TData extends RowData = any> {
  createCell?: (
    cell: Cell<TData, unknown>,
    column: Column<TData>,
    row: Row<TData>,
    table: Table<TData>
  ) => void
  createColumn?: (column: Column<TData, unknown>, table: Table<TData>) => void
  createHeader?: (header: Header<TData, unknown>, table: Table<TData>) => void
  createRow?: (row: Row<TData>, table: Table<TData>) => void
  createTable?: (table: Table<TData>) => void
  getDefaultColumnDef?: () => Partial<ColumnDef<TData, unknown>>
  getDefaultOptions?: (
    table: Table<TData>
  ) => Partial<TableOptionsResolved<TData>>
  getInitialState?: (initialState?: InitialTableState) => Partial<TableState>
}
```

This might be a bit confusing, so let's break down what each of these methods does:

Default Options and Initial State

getDefaultOptions

The `getDefaultOptions` method in a table feature is responsible for setting the default table options for that feature. For example, in the Column Sizing feature, the `getDefaultOptions` method sets the default `columnResizeMode` option with a default value of `"onEnd"`.

getDefaultColumnDef

The `getDefaultColumnDef` method in a table feature is responsible for setting the default column options for that feature. For example, in the Sorting feature, the `getDefaultColumnDef` method sets the default `sortUndefined` column option with a default value of `1`.

getInitialState

The `getInitialState` method in a table feature is responsible for setting the default state for that feature. For example, in the Pagination feature, the `getInitialState` method sets the default `pageSize` state with a value of `10` and the default `pageIndex` state with a value of `0`.

API Creators

createTable

The `createTable` method in a table feature is responsible for adding methods to the `table` instance. For example, in the Row Selection feature, the `createTable` method adds many table instance API methods such as `toggleAllRowsSelected`, `getIsAllRowsSelected`, `getIsSomeRowsSelected`, etc. So then, when you call `table.toggleAllRowsSelected()`, you are calling a method that was added to the table instance by the RowSelection feature.

createHeader

The `createHeader` method is responsible for adding methods to the `header` instance. For example, in the Column Sizing feature, the `createHeader` method adds many header instance API methods such as `getStart`, and many others. So then, when you call `header.getStart()`, you are calling a method added by the ColumnSizing feature.

createColumn

The `createColumn` method adds methods to the `column` instance. For example, in the Row Sorting feature, it adds methods like `getNextSortingOrder`, `toggleSorting`, etc. When you call `column.toggleSorting()`, that method is added by this feature.

createRow

The `createRow` method adds methods to the `row` instance. For example, in the Row Selection feature, it adds `toggleSelected`, `getIsSelected`, etc. Calling `row.toggleSelected()` uses this.

createCell

The `createCell` method adds methods to the `cell` instance. For example, in the Column Grouping feature, it adds methods like `getIsGrouped`, `getIsAggregated`, etc. Calling `cell.getIsGrouped()` uses this.

Adding a Custom Feature

Let's walk through creating a custom table feature for a hypothetical use case. Suppose we want to add a feature that allows users to change the **density** (padding) of cells.

Check out the full custom-features example for the complete implementation. Here's an in-depth view of the steps:

Step 1: Set Up TypeScript Types

Assuming full type safety, define types for the new feature's state, options, and APIs.

```
// define types for our new feature's custom state
export type DensityState = 'sm' | 'md' | 'lg'
export interface DensityTableState {
  density: DensityState
}

// define types for our new feature's options
export interface DensityOptions {
  enableDensity?: boolean
  onDensityChange?: OnChangeFn<DensityState>
}

// define types for our new feature's table APIs
export interface DensityInstance {
  setDensity: (updater: Updater<DensityState>) => void
  toggleDensity: (value?: DensityState) => void
}
```

Step 2: Use Declaration Merging

Modify the module's types to include our new feature:

```
declare module '@tanstack/react-table' {
  interface TableState extends DensityTableState {}
  interface TableOptionsResolved<TData extends RowData> extends DensityOptions {}
  interface Table<TData extends RowData> extends DensityInstance {}
  // You can also extend Cell, Row, Column types if needed
}
```

Step 3: Create the Feature Object

Use the `TableFeature` type to ensure correctness:

```
export const DensityFeature: TableFeature<any> = {
  getInitialState: (state): DensityTableState => {
    return {
      density: 'md',
      ...state,
    }
  },

  getDefaultOptions: <TData extends RowData>(
    table: Table<TData>
  ): DensityOptions => {
    return {
      enableDensity: true,
      onDensityChange: makeStateUpdater('density', table),
    } as DensityOptions
  },

  createTable: <TData extends RowData>(table: Table<TData>): void => {
    table.setDensity = (updater) => {
      const safeUpdater: Updater<DensityState> = (old) => {
        const newState = functionalUpdate(updater, old)
        return newState
      }
      return table.options.onDensityChange?.(safeUpdater)
    }
    table.toggleDensity = (value) => {
      table.setDensity((old) => {
        if (value) return value
        // cycle through options
        return old === 'lg' ? 'md' : old === 'md' ? 'sm' : 'lg'
      })
    }
  },
}
```

Step 4: Add the Feature to the Table

Include the new feature when creating the table:

```
const table = useReactTable({
  _features: [DensityFeature],
  columns,
  data,
  // other options
})
```

Step 5: Use the Feature in Your Application

Now, you can utilize the new APIs and state:

```
const table = useReactTable({
  _features: [DensityFeature],
  columns,
  data,
  state: {
    density,
  },
```

```
  onDensityChange: setDensity,
})

// In rendering
const { density } = table.getState()
return (
  <td
    style={{
      padding:
        density === 'sm'
          ? '4px'
          : density === 'md'
          ? '8px'
          : '16px',
      transition: 'padding 0.2s',
    }}
  >
    {flexRender(cell.column.columnDef.cell, cell.getContext())}
  </td>
)
```

Do We Have to Do It This Way?

This method offers a clean, scalable way to extend TanStack Table with custom features. Alternatively, you could manage the feature's state separately (e.g., React `useState`) and just use the table's API in parallel. Building features side-by-side without deep integration is also perfectly valid, depending on your architecture.

Edit on GitHub

Edit on GitHub

On this page

Column Sizing Guide | TanStack Table Docs

On this page

Examples

Want to skip to the implementation? Check out these examples:

- column-sizing
- column-resizing-performant

API

Column Sizing API

Column Sizing Guide

The column sizing feature allows you to optionally specify the width of each column, including min and max widths. It also enables dynamic resizing, such as by dragging column headers.

Column Widths

Columns by default have the following measurement options:

```
export const defaultColumnSizing = {
  size: 150,
  minSize: 20,
  maxSize: Number.MAX_SAFE_INTEGER,
}
```

These defaults can be overridden by both `tableOptions.defaultColumn` and individual column definitions, in that order.

```
const columns = [
  {
```

```
    accessorKey: 'col1',
    size: 270, // set column size for this column
  },
  //...
]

const table = useReactTable({
  // override default column sizing
  defaultColumn: {
    size: 200, // starting column size
    minSize: 50, // enforced during resize
    maxSize: 500, // enforced during resize
  },
})
```

Column sizes are stored in table state as numbers, usually interpreted as pixel values, but can be linked to CSS styles as needed.

Applying Column Widths

You can use various approaches to apply these sizes:

- Semantic `<table>` elements or table-like CSS elements
- `<div>` or `<span>` elements in non-table layouts:
  - Block elements with fixed widths
  - Absolutely positioned elements
  - Flexbox with flexible widths
  - CSS Grid with flexible widths
- Any layout mechanism that can interpolate cell widths into a table

Performance may vary based on the method chosen; choose according to your UI design and limitations.

Column Resizing

TanStack Table offers built-in state and APIs for column resizing, enabling easy integration.

Enable Column Resizing

By default, `column.getCanResize()` returns `true`. To disable resizing globally or per column:

```
const columns = [
  {
    accessorKey: 'id',
    enableResizing: false, // disable for this column
    size: 200,
  },
  //...
]
```

Or set the table option:

```
useReactTable({
  enableColumnResizing: false, // disables for all columns
})
```

Column Resize Mode

Default is `"onEnd"`: size updates when the user finishes dragging. To support smoother resizing, set mode to `"onChange"`:

```
const table = useReactTable({
  columnResizeMode: 'onChange',
})
```

Column Resize Direction

Default is left-to-right. For RTL layouts:

```
const table = useReactTable({
  columnResizeDirection: 'rtl',
})
```

Connecting APIs to UI

Use the following APIs to hook drag interactions:

Column Size APIs

To get column width and assign to styles:

```
header.getSize()
column.getSize()
cell.column.getSize()
```

Apply via inline styles:

```
<th
  key={header.id}
  colSpan={header.colSpan}
  style={{ width: `${header.getSize()}px` }}
>
```

Column Resize APIs

Use `header.getResizeHandler()`:

```
<ColumnResizeHandle
  onMouseDown={header.getResizeHandler()} // desktop
  onTouchStart={header.getResizeHandler()} // mobile
/>
```

Resize Indicator & Performance

Use `columnSizingInfo` to render resize indicators:

```
<ColumnResizeIndicator
  style={{
    transform: header.column.getIsResizing()
      ? `translateX(${table.getState().columnSizingInfo.deltaOffset}px)`
      : '',
  }}
/>
```

Performance Tips

For large or complex tables, optimize resize performance by:

- Memoizing column widths calculations
- Memoizing table body during resize
- Using CSS variables for column widths

This helps achieve 60 fps during resize, especially in React.

For non-React frameworks, similar principles apply.

Edit on GitHub

Edit on GitHub

Column Visibility Guide

Examples

Want to skip to the implementation? Check out these examples:

- column-visibility
- column-ordering
- sticky-column-pinning

Other Examples

- SolidJS column-visibility
- Svelte column-visibility

API

Column Visibility API

Column Visibility Guide

The column visibility feature allows table columns to be hidden or shown dynamically. In previous versions of react-table, this feature was a static property on a column, but in v8, there is a dedicated *columnVisibility* state and APIs for managing column visibility dynamically.

Column Visibility State

The `columnVisibility` state is a map of column IDs to boolean values. A column will be hidden if its ID is present in the map and the value is *false*. If the column ID is not present in the map, or the value is *true*, the column will be shown.

```
const [columnVisibility, setColumnVisibility] = useState({
  columnId1: true,
  columnId2: false, // hide this column by default
  columnId3: true,
});

const table = useReactTable({
  //...
  state: {
    columnVisibility,
    //...
  },
  onColumnVisibilityChange: setColumnVisibility,
});
```

Alternatively, if you don't need to manage the column visibility state outside of the table, you can still set the initial default column visibility using the **initialState** option.

> **Note:** If `columnVisibility` is provided to both `initialState` and `state`, the `state` initialization will take precedence, and `initialState` will be ignored. Only provide `columnVisibility` in one place.

```
const table = useReactTable({
  //...
  initialState: {
    columnVisibility: {
      columnId1: true,
      columnId2: false, // hide this column by default
      columnId3: true,
    },
  },
});
```

Disable Hiding Columns

By default, all columns can be hidden or shown. To prevent certain columns from being hidden, set the `enableHiding` column option to `false` for those columns.

```
const columns = [
  {
    header: 'ID',
    accessorKey: 'id',
    enableHiding: false, // disable hiding for this column
  },
  {
    header: 'Name',
    accessor: 'name', // can be hidden
  },
];
```

Column Visibility Toggle APIs

There are several column API methods useful for rendering visibility toggles in the UI:

- `column.getCanHide` – Disables the toggle if set to `false`.
- `column.getIsVisible` – Checks if the column is visible.
- `column.toggleVisibility` – Toggles the visibility.
- `column.getToggleVisibilityHandler` – Shortcut to hook toggle to UI.

```
{table.getAllColumns().map((column) => (
  <label key={column.id}>
    <input
      checked={column.getIsVisible()}
      disabled={!column.getCanHide()}
      onChange={column.getToggleVisibilityHandler()}
      type="checkbox"
    />
    {column.columnDef.header}
  </label>
))}
```

Column Visibility Aware Table APIs

When rendering header, body, and footer cells, use the *visible* variants that respect column visibility:

- `table.getVisibleLeafColumns()`
- `row.getVisibleCells()`

```
<table>
  <thead>
    <tr>
      {table.getVisibleLeafColumns().map((column) => (
        // takes column visibility into account
      ))}
    </tr>
  </thead>
  <tbody>
    {table.getRowModel().rows.map((row) => (
      <tr key={row.id}>
        {row.getVisibleCells().map((cell) => (
          // takes column visibility into account
        ))}
      </tr>
    ))}
  </tbody>
</table>
```

Similarly for header groups, they will already consider column visibility.

Edit on GitHub

On this page

Column Filtering Guide | TanStack Table Docs

TanStack Table v8

Auto

Framework

React

Version: Latest

Search...

+ K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Introduction (core)
- Overview (core)
- Installation (core)
- Migrating to V8 (core)
- FAQ (core)
- React Table Adapter (react)

Core Guides

- Data (core)
- Column Defs (core)
- Table Instance (core)
- Row Models (core)
- Rows (core)
- Cells (core)
- Header Groups (core)

- Headers (core)
- Columns (core)
- Table State (react)

Feature Guides

- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Column Filtering (core)
- Global Filtering (core)
- Fuzzy Filtering (core)
- Column Faceting (core)
- Global Faceting (core)
- Grouping (core)
- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)
- Sorting (core)
- Virtualization (core)
- Custom Features (core)

Core APIs

- Column Def (core)
- Table (core)
- Column (core)
- Header Group (core)
- Header (core)
- Row (core)
- Cell (core)

Feature APIs

- Column Filtering (core)
- Column Faceting (core)
- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Global Faceting (core)
- Global Filtering (core)
- Sorting (core)
- Grouping (core)
- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)

Enterprise

- AG Grid

Examples

- Basic (react)
- Header Groups
- Column Filters
- Column Filters (Faceted)
- Fuzzy Search Filters
- Column Ordering
- Column Ordering (DnD)
- Column Pinning
- Sticky Column Pinning
- Column Sizing
- Performant Column Resizing
- Column Visibility
- Editable Data
- Expanding
- Sub Components
- Fully Controlled
- Grouping
- Pagination
- Pagination Controlled
- Row DnD
- Row Pinning
- Row Selection
- Sorting
- Virtualized Columns
- Virtualized Columns (Experimental)
- Virtualized Rows
- Virtualized Rows (Experimental)
- Virtualized Infinite Scrolling
- Kitchen Sink
- React Bootstrap
- Material UI Pagination
- React Full Width
- React Full Width Resizable
- Custom Features
- Query Router Search Params

On this page

- FilterFns
- Custom Filter Functions
- Customize Filter Function Behavior
- Customize Column Filtering
- Disable Column Filtering
- Filtering Sub-Rows (Expanding)
- Filter From Leaf Rows
- Max Leaf Row Filter Depth
- Column Filter APIs

Global Filtering Guide | TanStack Table Docs

Navigation

Global Filtering Guide

Want to skip to the implementation? Check out these examples:

- Global Filters

API

Global Filtering API

Global Filtering Guide

Filtering comes in 2 flavors: Column Filtering and Global Filtering.

This guide will focus on **global filtering**, which is a filter that is applied across all columns.

Client-Side vs Server-Side Filtering

If you have a large dataset, you may not want to load all of that data into the client's browser in order to filter it. In this case, you will most likely want to implement **server-side filtering**, **sorting**, **pagination**, etc.

However, as also discussed in the Pagination Guide, many developers underestimate how many rows can be loaded client-side without performance issues. The TanStack table examples are often tested to handle up to 100,000 rows or more with decent performance for client-side filtering, sorting, pagination, and grouping. This doesn't necessarily mean your app will handle that many, but if your table is only a few thousand rows at most, you might leverage client-side capabilities.

> **TanStack Table** can handle thousands of rows on the client with good performance. Don't rule out client-side filtering, pagination, sorting, etc., prematurely.

Every use-case is different, depending on factors like table complexity, columns, data size, etc. The main concerns are:

1. Can your server query all data in a reasonable time (and cost)?
2. What is the total fetch size? (It might not scale badly if few columns)
3. Is the browser using too much memory with all data loaded?

Start with client-side filtering and switch to server strategies as needed.

Manual Server-Side Global Filtering

If you opt for server-side filtering, here's how to implement it.

No `getFilteredRowModel` table option is needed. The data passed should already be filtered. If you passed `getFilteredRowModel`, you can disable it by setting `manualFiltering` to `true`.

```
const table = useReactTable({
  data,
  columns,
  // getFilteredRowModel: getFilteredRowModel(), // not needed for manual server filte
  manualFiltering: true,
})
```

When `manualFiltering` is true, the table assumes data is pre-filtered, and skips internal filtering logic.

Client-Side Global Filtering

If using built-in client-side filtering, supply `getFilteredRowModel`:

```
import { useReactTable, getFilteredRowModel } from '@tanstack/react-table'

const table = useReactTable({
  // other options...
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
})
```

Global Filter Function

`globalFilterFn` lets you specify the filter function for global filtering. It can be:

- A string referencing a built-in or custom filter function
- A custom function directly

```
const table = useReactTable({
  data,
  columns,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  globalFilterFn: 'text' // built-in function
})
```

By default, there are **10** built-in filter functions:

- `includesString` - case-insensitive string inclusion
- `includesStringSensitive` - case-sensitive string inclusion
- `equalsString` - case-insensitive string equality
- `equalsStringSensitive` - case-sensitive string equality
- `arrIncludes` - item in array
- `arrIncludesAll` - all items in array
- `arrIncludesSome` - some items in array
- `equals` - object equality
- `weakEquals` - weak object equality
- `inNumberRange` - number range

Custom functions can also be defined and assigned as `globalFilterFn`.

Global Filter State

The global filter state is stored inside the React table state under `table.getState().globalFilter`.

You can persist or synchronize it outside via `onGlobalFilterChange` callback:

```
const [globalFilter, setGlobalFilter] = useState([])

const table = useReactTable({
  //...
  state: {
    globalFilter,
  },
  onGlobalFilterChange: setGlobalFilter
})
```

The shape of global filter state:

```
interface GlobalFilter {
  globalFilter: any
}
```

Adding global filter input to UI

TanStack Table does not add input UI automatically; you must include it manually:

```
return (
  <div>
    <input
      value=""
      onChange={(e) => table.setGlobalFilter(String(e.target.value))}
      placeholder="Search..."
    />
  </div>
)
```

Custom Global Filter Function

To use a custom function:

```
const customFilterFn = (rows, columnId, filterValue) => {
  // custom logic
}

const table = useReactTable({
  // other options...
  globalFilterFn: customFilterFn
})
```

> **Note:** Fuzzy filtering functions are popular for global filtering (see Guide).

Initial Global Filter State

Set initial state on table init:

```
const [globalFilter, setGlobalFilter] = useState("search term") // initialize

const table = useReactTable({
  initialState: {
    globalFilter: 'search term'
  },
  state: {
    globalFilter
  }
})
```

> **Note:** Do not use `initialState.globalFilter` and `state.globalFilter` simultaneously; the `state` value overrides.

Disable Global Filtering

You can disable global filtering per column or for all columns:

```
const columns = [
  {
    header: () => 'Id',
    accessorKey: 'id',
    enableGlobalFilter: false, // disable for this column
  },
  // ...
]

const table = useReactTable({
  // other options...
  columns,
  enableGlobalFilter: false, // disable for all columns
})
```

Disabling global filtering causes `column.getCanGlobalFilter` to return `false`.

Edit on GitHub

https://github.com/tanstack/table/edit/main/docs/guide/global-filtering.md

Fuzzy Filtering Guide

Examples

Want to skip to the implementation? Check out these examples:

- filters-fuzzy

API

Filters API

Fuzzy Filtering Guide

Fuzzy filtering is a technique that allows you to filter data based on approximate matches. This can be useful when you want to search for data that is similar to a given value, rather than an exact match.

You can implement a client-side fuzzy filtering by defining a custom filter function. This function should take in the row, columnId, and filter value, and return a boolean indicating whether the row should be included in the filtered data.

Fuzzy filtering is mostly used with global filtering, but you can also apply it to individual columns. We will discuss how to implement fuzzy filtering for both cases.

> **Note:** You will need to install the `@tanstack/match-sorter-utils` library to use fuzzy filtering.
> TanStack Match Sorter Utils is a fork of match-sorter by Kent C. Dodds. It was forked to work better with TanStack Table's row-by-row filtering approach.

Using the match-sorter libraries is optional, but the TanStack Match Sorter Utils library provides a great way to both fuzzy filter and sort by the rank information it returns, so that rows can be sorted by their closest matches to the search query.

Defining a Custom Fuzzy Filter Function

Here's an example of a custom fuzzy filter function:

```
import { rankItem } from '@tanstack/match-sorter-utils';
import { FilterFn } from '@tanstack/table';

const fuzzyFilter: FilterFn<any> = (row, columnId, value, addMeta) => {
  // Rank the item
  const itemRank = rankItem(row.getValue(columnId), value);

  // Store the itemRank info
  addMeta({ itemRank });

  // Return if the item should be filtered in/out
  return itemRank.passed;
}
```

In this function, we're using the `rankItem` function from the `@tanstack/match-sorter-utils` library to rank the item. We then store the ranking information in the meta data of the row, and return whether the item passed the ranking criteria.

Using Fuzzy Filtering with Global Filtering

To use fuzzy filtering with global filtering, you can specify the fuzzy filter function in the `globalFilterFn` option of the table instance:

```
const table = useReactTable({
  columns,
  data,
  filterFns: {
    fuzzy: fuzzyFilter, // define as a filter function that can be used in column defi
  },
  globalFilterFn: 'fuzzy', // apply fuzzy filter to the global filter (most common use
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(), // client-side filtering
  getSortedRowModel: getSortedRowModel(), // client-side sorting needed if you want to
});
```

Using Fuzzy Filtering with Column Filtering

To use fuzzy filtering with column filtering, first define the fuzzy filter function in the `filterFns` option of the table instance. Then, specify the fuzzy filter function in the `filterFn` option of the column definition:

```
const columns = [
  {
    accessorFn: row => `${row.firstName} ${row.lastName}`,
    id: 'fullName',
    header: 'Full Name',
    cell: info => info.getValue(),
    filterFn: 'fuzzy', // using our custom fuzzy filter function
  },
  // other columns...
];
```

In this example, we're applying the fuzzy filter to a column that combines the `firstName` and `lastName` fields of the data.

Sorting with Fuzzy Filtering

When using fuzzy filtering with column filtering, you might also want to sort the data based on the ranking information. You can do this by defining a custom sorting function:

```
import { compareItems } from '@tanstack/match-sorter-utils';
import { sortingFns } from '@tanstack/table';

const fuzzySort: SortingFn<any> = (rowA, rowB, columnId) => {
  let dir = 0;

  // Only sort by rank if the column has ranking information
  if (rowA.columnFiltersMeta[columnId]) {
    dir = compareItems(
```

```
      rowA.columnFiltersMeta[columnId]?.itemRank!,
      rowB.columnFiltersMeta[columnId]?.itemRank!
    );
  }

  // Provide an alphanumeric fallback for when the item ranks are equal
  return dir === 0 ? sortingFns.alphanumeric(rowA, rowB, columnId) : dir;
};
```

In this function, we're comparing the ranking information of the two rows. If the ranks are equal, we fall back to alphanumeric sorting.

You can then specify this sorting function in the `sortFn` option of the column definition:

```
const columns = [
  {
    accessorFn: row => `${row.firstName} ${row.lastName}`,
    id: 'fullName',
    header: 'Full Name',
    cell: info => info.getValue(),
    filterFn: 'fuzzy', // using our custom fuzzy filter function
    sortFn: 'fuzzySort', // using our custom fuzzy sort function
  },
  // other columns...
];
```

Edit on GitHub

On this page

Column Faceting Guide

Examples

Want to skip to the implementation? Check out these examples:

- filters-faceted

API

Column Faceting API

Column Faceting Guide

Column Faceting is a feature that allows you to generate lists of values for a given column from that column's data. For example, a list of *unique values* in a column can be generated from all rows in that column to be used as search suggestions in an autocomplete filter component. Or, a tuple of *minimum and maximum values* can be generated from a column of numbers to be used as a range for a range slider filter component.

Column Faceting Row Models

In order to use any of the column faceting features, you must include the appropriate row models in your table options.

```
// only import the row models you need
import {
  getCoreRowModel,
  getFacetedRowModel,
  getFacetedMinMaxValues, // depends on getFacetedRowModel
  getFacetedUniqueValues, // depends on getFacetedRowModel
}
// ...
const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getFacetedRowModel: getFacetedRowModel(), // if you need a list of values for a colu
  getFacetedMinMaxValues: getFacetedMinMaxValues(), // if you need min/max values
  getFacetedUniqueValues: getFacetedUniqueValues(), // if you need a list of unique va
  // ...
})
```

```
// only import the row models you need
import {
  getCoreRowModel,
  getFacetedRowModel,
  getFacetedMinMaxValues, // depends on getFacetedRowModel
  getFacetedUniqueValues, // depends on getFacetedRowModel
}
// ...
const table = useReactTable({
```

```
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getFacetedRowModel: getFacetedRowModel(), // if you need a list of values for a colu
  getFacetedMinMaxValues: getFacetedMinMaxValues(), // if you need min/max values
  getFacetedUniqueValues: getFacetedUniqueValues(), // if you need a list of unique va
  // ...
})
```

First, you must include the `getFacetedRowModel`. This row model will generate a list of values for a given column. If you need a list of *unique values*, include the `getFacetedUniqueValues`. If you need a tuple of *minimum and maximum values*, include the `getFacetedMinMaxValues`.

Use Faceted Row Models

Once you have included the appropriate row models in your table options, you will be able to use the faceting column instance APIs to access the lists of values generated by the faceted row models.

```
// list of unique values for autocomplete filter
const autoCompleteSuggestions =
  Array.from(column.getFacetedUniqueValues().keys())
    .sort()
    .slice(0, 5000);

// tuple of min and max values for range filter
const [min, max] = column.getFacetedMinMaxValues() ?? [0, 1];
```

Custom (Server-Side) Faceting

If instead of using the built-in client-side faceting features, you can implement your own faceting logic on the server-side and pass the faceted values to the client-side. You can use the `getFacetedUniqueValues` and `getFacetedMinMaxValues` table options to resolve the faceted values from the server-side.

```
const facetingQuery = useQuery(
  // ...
)

const table = useReactTable({
  columns,
  data,
  getCoreRowModel: getCoreRowModel(),
  getFacetedRowModel: getFacetedRowModel(),
  getFacetedUniqueValues: (table, columnId) => {
    const uniqueValueMap = new Map<string, number>();
    // ...
    return uniqueValueMap;
  },
  getFacetedMinMaxValues: (table, columnId) => {
    // ...
    return [min, max];
  },
  // ...
})
```

Alternatively, you don't have to put any of your faceting logic through the TanStack Table APIs at all. Just fetch your lists and pass them to your filter components directly.

Edit on GitHub

On this page

Global Faceting Guide

Examples

Want to skip to the implementation? Check out these examples:

- filters-faceted

API

Global Faceting API

Global Faceting Guide

Global Faceting allows you to generate lists of values for all columns from the table's data. For example, a list of unique values in a table can be generated from all rows in all columns to be used as search suggestions in an autocomplete filter component. Or, a tuple of minimum and maximum values can be generated from a table of numbers to be used as a range for a range slider filter component.

Global Faceting Row Models

In order to use any of the global faceting features, you must include the appropriate row models in your table options.

```
// only import the row models you need
import {
  getCoreRowModel,
  getFacetedRowModel,
  getFacetedMinMaxValues, // depends on getFacetedRowModel
  getFacetedUniqueValues, // depends on getFacetedRowModel
} from '@tanstack/react-table'

//...
const table = useReactTable({
  // other options...
  getCoreRowModel: getCoreRowModel(),
  getFacetedRowModel: getFacetedRowModel(), // Faceting model for client-side faceting
  getFacetedMinMaxValues: getFacetedMinMaxValues(), // if you need min/max values
  getFacetedUniqueValues: getFacetedUniqueValues(), // if you need a list of unique va
  //...
})
```

```
// only import the row models you need
import {
  getCoreRowModel,
  getFacetedRowModel,
  getFacetedMinMaxValues, // depends on getFacetedRowModel
  getFacetedUniqueValues, // depends on getFacetedRowModel
} from '@tanstack/react-table'

//...
```

```
const table = useReactTable({
  // other options...
  getCoreRowModel: getCoreRowModel(),
  getFacetedRowModel: getFacetedRowModel(), // Faceting model for client-side faceting
  getFacetedMinMaxValues: getFacetedMinMaxValues(), // if you need min/max values
  getFacetedUniqueValues: getFacetedUniqueValues(), // if you need a list of unique va
  //...
})
```

Use Global Faceted Row Models

Once you have included the appropriate row models in your table options, you will be able to use the faceting table instance APIs to access the lists of values generated by the faceted row models.

```
// list of unique values for autocomplete filter
const autoCompleteSuggestions =
  Array.from(table.getGlobalFacetedUniqueValues().keys())
    .sort()
    .slice(0, 5000);
```


```
// list of unique values for autocomplete filter
const autoCompleteSuggestions =
  Array.from(table.getGlobalFacetedUniqueValues().keys())
    .sort()
    .slice(0, 5000);
```


```
// tuple of min and max values for range filter
const [min, max] = table.getGlobalFacetedMinMaxValues() ?? [0, 1];
```


```
// tuple of min and max values for range filter
const [min, max] = table.getGlobalFacetedMinMaxValues() ?? [0, 1];
```


Custom Global (Server-Side) Faceting

If instead of using the built-in client-side faceting features, you can implement your own faceting logic on the server-side and pass the faceted values to the client-side. You can use the `getGlobalFacetedUniqueValues` and `getGlobalFacetedMinMaxValues` table options to resolve the faceted values from the server-side.

```
const facetingQuery = useQuery(
  'faceting',
  async () => {
    const response = await fetch('/api/faceting');
    return response.json();
  },
  {
    onSuccess: (data) => {
      table.getGlobalFacetedUniqueValues = () => data.uniqueValues;
      table.getGlobalFacetedMinMaxValues = () => data.minMaxValues;
    },
  }
);
```

In this example, we use the `useQuery` hook from `react-query` to fetch faceting data from the server. Once the data is fetched, we set the `getGlobalFacetedUniqueValues` and `getGlobalFacetedMinMaxValues` table options to return the faceted values from the server response.

This will allow the table to use the server-side faceting data for generating autocomplete suggestions and range filters.

Edit on GitHub

On this page

Grouping Guide | TanStack Table Docs

Overview

Navigation

Examples

Want to skip to the implementation? Check out these examples:

- GitHub - TanStack table examples

API

Grouping API

Grouping Guide

There are three table features that can reorder columns, which happen in the following order:

1. **Column Pinning**
   Pins columns into left, center, or right positions.
2. **Manual Column Ordering**
   Applies a specified column order.
3. **Grouping**
   When enabled and `tableOptions.groupedColumnMode` is `'reorder'` or `'remove'`, grouped columns are moved to the start.

What is grouping?

Grouping allows you to categorize and organize table rows based on specific columns, especially useful with large datasets.

Usage

Use the `getGroupedRowModel` to enable grouping:

```
import { getGroupedRowModel } from '@tanstack/react-table'

const table = useReactTable({
  // other options...
  getGroupedRowModel: getGroupedRowModel(),
})
```

When active, rows are matched and grouped under a parent row, with subRows representing the grouped data. Expand/collapse feature can be enabled with `getExpandedRowModel`:

```
import { getGroupedRowModel, getExpandedRowModel } from '@tanstack/react-table'

const table = useReactTable({
  // other options...
  getGroupedRowModel: getGroupedRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
})
```

Grouping State

The grouping state is an array of column IDs, which determines the order of grouping:

```
table.setGrouping(['column1', 'column2']);
```

Reset with:

```
table.resetGrouping();
```

Behavior control:

```
const table = useReactTable({
  // other options...
  groupedColumnMode: 'reorder', // or 'remove', or false
})
```

Aggregations

Aggregate grouped data using `aggregationFn`:

```
const column = columnHelper.accessor('key', {
  aggregationFn: 'sum',
});
```

Built-in options:

- `sum`, `count`, `min`, `max`, `extent`, `mean`, `median`, `unique`, `uniqueCount`

Custom Aggregations

Define custom aggregation functions:

```
const table = useReactTable({
  // other options...
  aggregationFns: {
    myCustomAggregation: (columnId, leafRows, childRows) => {
```

```
      // return aggregated value
    },
  },
});
```

Reference in column:

```
const column = columnHelper.accessor('key', {
  aggregationFn: 'myCustomAggregation',
});
```

Manual Grouping

For server-side grouping:

```
const table = useReactTable({
  // other options...
  manualGrouping: true,
})
```

> **Note:** Server-side grouping requires custom cell rendering; it's not straightforward.

Grouping Change Handler

Manage grouping state externally:

```
const [grouping, setGrouping] = useState([]);

const table = useReactTable({
  // other options...
  state: { grouping },
  onGroupingChange: setGrouping,
});
```

Edit on GitHub

Additional links

- Global Faceting
- Expanding

Partners & Subscription

- Primitives for range sliders.

Subscribe to Bytes:
Your weekly dose of JavaScript news. Delivered every Monday.

No spam. Unsubscribe at *any* time.

ColumnDef APIs

Column definitions are plain objects with the following options:

Options

id

```
id: string
```

The unique identifier for the column.

> 🧠 A column ID is optional when:
>
> - An accessor column is created with an object key accessor
> - The column header is defined as a string

accessorKey

```
accessorKey?: string & typeof TData
```

The key of the row object to use when extracting the value for the column.

accessorFn

```
accessorFn?: (originalRow: TData, index: number) => any
```

The accessor function to use when extracting the value for the column from each row.

columns

```
columns?: ColumnDef<TData>[]
```

The child column defs to include in a group column.

header

```
header?:
  | string
  | ((props: {
      table: Table<TData>;
      header: Header<TData>;
      column: Column<TData>;
    }) => unknown)
```

The header to display for the column. If a string is passed, it can be used as a default for the column ID. If a function is passed, it will be passed a props object for the header and should return the rendered header value (the exact type depends on the adapter being used).

footer

```
footer?:
  | string
```

```
  | ((props: {
      table: Table<TData>;
      header: Header<TData>;
      column: Column<TData>;
    }) => unknown)
```

The footer to display for the column. If a function is passed, it will be passed a props object for the footer and should return the rendered footer value (the exact type depends on the adapter being used).

cell

```
cell?:
  | string
  | ((props: {
      table: Table<TData>;
      row: Row<TData>;
      column: Column<TData>;
      cell: Cell<TData>;
      getValue: () => any;
      renderValue: () => any;
    }) => unknown)
```

The cell to display each row for the column. If a function is passed, it will be passed a props object for the cell and should return the rendered cell value (the exact type depends on the adapter being used).

meta

```
meta?: ColumnMeta // This interface is extensible via declaration merging. See below!
```

The meta data to be associated with the column. We can access it anywhere when the column is available via `column.columnDef.meta`. This type is global to all tables and can be extended like so:

```
import '@tanstack/react-table' //or vue, svelte, solid, qwik, etc.

declare module '@tanstack/react-table' {
  interface ColumnMeta<TData extends RowData, TValue> {
    foo: string
  }
}
```

Edit on GitHub

On this page

Table APIs | TanStack Table Docs

Overview

Heavy-duty table utilities for TS/JS frameworks

These functions are used to create a table. Which one you use depends on which framework adapter you are using.

Options

These are **core** options and API properties for the table. More options and API properties are available for other table features.

data

`tsx`

```
data: TData[]
```

The data for the table to display. This array should match the type you provided to `table.setRowType<...>`, but in theory could be an array of anything. It's common for each item in the array to be an object of key/values but this is not required. Columns can access this data via string/index or a functional accessor to return anything they want.

When the `data` option changes reference (compared via `Object.is`), the table will reprocess the data. Any other data processing that relies on the core data model (such as grouping, sorting, filtering, etc) will also be reprocessed.

> 🧠 Make sure your `data` option is only changing when you want the table to reprocess. Providing an inline `[]` or constructing the data array as a new object every time you want to render the table will result in a *lot* of unnecessary re-processing. This can easily go unnoticed in smaller tables, but you will likely notice it in larger tables.

columns

```
type columns = ColumnDef<TData>[]
```

The array of column defs to use for the table. See the Column Def Guide for more information on creating column definitions.

defaultColumn

```
defaultColumn?: Partial<ColumnDef<TData>>
```

Default column options to use for all column defs supplied to the table. This is useful for providing default cell/header/footer renderers, sorting/filtering/grouping options, etc. All column definitions passed to `options.columns` are merged with this default column definition to produce the final column definitions.

initialState

```tsx
initialState?: Partial<
  VisibilityTableState & ColumnOrderTableState & ColumnPinningTableState & FiltersTabl
>
```

Use this option to optionally pass initial state to the table. This state will be used when resetting various table states either automatically by the table (e.g., `options.autoResetPageIndex`) or via functions like `table.resetRowSelection()`. Most reset functions allow you to pass a flag to reset to a blank/default state instead of the initial state.

> 🧠 Table state will not be reset when this object changes, which also means that the initial state object does not need to be stable.

autoResetAll

```tsx
autoResetAll?: boolean
```

Set this option to override any of the `autoReset...` feature options.

meta

```tsx
meta?: TableMeta // This interface is extensible via declaration merging. See below!
```

You can pass any object to `options.meta` and access it anywhere the `table` is available via `table.options.meta`. This type is global to all tables and can be extended like so:

```
declare module '@tanstack/table-core' {
  interface TableMeta<TData extends RowData> {
    foo: string
  }
}
```

> 🧠 Think of this option as an arbitrary "context" for your table. This allows passing arbitrary data or functions to your table without passing them everywhere the table touches. For example, passing a locale object for date/number formatting or functions for editable data (see editable-data example).

state

```tsx
state?: Partial<
  VisibilityTableState & ColumnOrderTableState & ColumnPinningTableState & FiltersTabl
>
```

This option can be used to optionally *control* part or all of the table state. The state you pass here will merge with and overwrite the internal automatically-managed state. You can also listen to state changes via `onStateChange`.

onStateChange

```tsx
onStateChange: (updater: Updater<TableState>) => void
```

Use this to listen to state changes within the table. If provided, you'll be responsible for controlling and updating the table state yourself, and you can pass the new state back via the `state` prop.

Debugging Options

> ⚠️ Debugging is only available in development mode.

debugAll

```tsx
debugAll?: boolean
```

Set to `true` to output all debugging info to the console.

debugTable

```tsx
debugTable?: boolean
```

Set to `true` to output table debugging info.

debugHeaders

```tsx
debugHeaders?: boolean
```

Set to `true` for header debugging output.

debugColumns

```tsx
debugColumns?: boolean
```

Set to `true` for column debugging output.

debugRows

```tsx
debugRows?: boolean
```

Set to `true` for row debugging output.

_features

```tsx
_features?: TableFeature[]
```

An array of extra features to add to the table instance.

render

```tsx
type render = <TProps>(template: Renderable<TProps>, props: TProps) => any
```

Provides a renderer implementation for the table, converting templates into framework-supported output.

> ⚠️ This is only necessary if you're implementing a table adapter.

mergeOptions

```tsx
type mergeOptions = <T>(defaultOptions: T, options: Partial<T>) => T
```

Handles merging of table options, especially relevant for reactive frameworks.

⚠️ Only necessary if you are implementing a custom table adapter.

getCoreRowModel

```tsx
getCoreRowModel: (table: Table<TData>) => () => RowModel<TData>
```

Factory for computing the core row model before processing. Called once per table.

getSubRows

```tsx
getSubRows?: (originalRow: TData, index: number) => undefined | TData[]
```

Access sub-rows for nested data.

getRowId

```tsx
getRowId?: (originalRow: TData, index: number, parent?: Row<TData>) => string
```

Derives a unique ID for a row.

Table API

initialState

```tsx
initialState: VisibilityTableState & ColumnOrderTableState & ColumnPinningTableState &
```

The resolved initial state of the table.

reset

```tsx
reset: () => void
```

Reset table state to initial.

getState

```tsx
getState: () => TableState
```

Get current table state (recommended for manual management).

🧠 Returns the shallow-merged state of internal and externally controlled state.

setState

```tsx
setState: (updater: Updater<TableState>) => void
```

Update table state, recommended via function `(prevState) => newState`.

🧠 If `options.onStateChange` is provided, it triggers with the new state.

options

```tsx
options: TableOptions<TData>
```

Current table options (read-only).

⚠️ Typically used internally or by adapters; updated via `setOptions`.

setOptions

```tsx
setOptions: (newOptions: Updater<TableOptions<TData>>) => void
```

Update table options (usually through adapters).

getCoreRowModel

```tsx
getCoreRowModel: () => {
  rows: Row<TData>[],
  flatRows: Row<TData>[],
  rowsById: Record<string, Row<TData>>,
}
```

Returns the core row model before processing.

getRowModel

```tsx
getRowModel: () => {
  rows: Row<TData>[],
  flatRows: Row<TData>[],
  rowsById: Record<string, Row<TData>>,
}
```

Returns the processed row model.

getAllColumns

```tsx
type getAllColumns = () => Column<TData>[]
```

Returns all columns.

getAllFlatColumns

```tsx
type getAllFlatColumns = () => Column<TData>[]
```

Returns flattened column list.

getAllLeafColumns

```tsx
type getAllLeafColumns = () => Column<TData>[]
```

Returns only leaf columns.

getColumn

```tsx
type getColumn = (id: string) => Column<TData> | undefined
```

Retrieve a column by ID.

getHeaderGroups

```tsx
type getHeaderGroups = () => HeaderGroup<TData>[]
```

Header groups.

getFooterGroups

```tsx
type getFooterGroups = () => HeaderGroup<TData>[]
```

Footer groups.

getFlatHeaders

```tsx
type getFlatHeaders = () => Header<TData>[]
```

Flattened headers.

getLeafHeaders

```tsx
type getLeafHeaders = () => Header<TData>[]
```

Flattened leaf headers.

About this page

On this page

- debugTable
- debugHeaders
- debugColumns
- debugRows
- _features
- render
- mergeOptions
- getCoreRowModel
- getSubRows
- getRowId
- Table API
- initialState (Table API)
- reset
- getState
- setState
- options
- setOptions
- getCoreRowModel (Table API)
- getRowModel
- getAllColumns
- getAllFlatColumns
- getAllLeafColumns
- getColumn
- getHeaderGroups
- getFooterGroups
- getFlatHeaders
- getLeafHeaders

Additional Info

Our Partners

- https://ag-grid.com/react-data-grid/?utm_source=reacttable&utm_campaign=githubreacttable (opens in new tab)

Additional Resources

- TanStack Query: Powerful asynchronous state management, server-state utilities, and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications, all without touching trivial "global state".
  Learn More

- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations that keep your UI reactive, consistent, and blazing fast 🔥.
  Learn More

Column APIs | TanStack Table Docs

Getting Started

Core Guides

Feature Guides

Core APIs

- Column (core)
- Header Group (core)
- Header (core)
- Row (core)
- Cell (core)

Feature APIs

- Column Filtering (core)
- Column Faceting (core)
- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Global Faceting (core)
- Global Filtering (core)
- Sorting (core)
- Grouping (core)
- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)

Enterprise

- AG Grid (core)

Examples

- Basic (react)
- Header Groups (react)
- Column Filters (react)
- Column Filters (Faceted) (react)
- Fuzzy Search Filters (react)
- Column Ordering (react)
- Column Ordering (DnD) (react)
- Column Pinning (react)
- Sticky Column Pinning (react)
- Column Sizing (react)
- Performant Column Resizing (react)
- Column Visibility (react)
- Editable Data (react)
- Expanding (react)
- Sub Components (react)
- Fully Controlled (react)
- Grouping (react)
- Pagination (react)
- Pagination Controlled (react)
- Row DnD (react)
- Row Pinning (react)
- Row Selection (react)
- Sorting (react)
- Virtualized Columns (react)
- Virtualized Columns (Experimental) (react)
- Virtualized Rows (react)

- Virtualized Rows (Experimental) (react)
- Virtualized Infinite Scrolling (react)
- Kitchen Sink (react)
- React Bootstrap (react)
- Material UI Pagination (react)
- React Full Width (react)
- React Full Width Resizable (react)
- Custom Features (react)
- Query Router Search Params (react)

On this page

Column API

These are **core** options and API properties for all columns. More options and API properties are available for other table features.

Column API

All column objects have the following properties:

`id`

```
id: string
```

The resolved unique identifier for the column, prioritized as:

- A manual `id` property from the column def
- The accessor key from the column def
- The header string from the column def

`depth`

```
depth: number
```

The depth of the column (if grouped) relative to the root column def array.

`accessorFn`

```
accessorFn?: AccessorFn<TData>
```

The resolved accessor function to use when extracting the value for the column from each row. Defined if the column def has a valid accessor key or function.

`columnDef`

```
columnDef: ColumnDef<TData>
```

The original column def used to create the column.

`columns`

```
type columns = ColumnDef<TData>[]
```

The child columns (if the column is a group). Empty array if not a group column.

`parent`

```
parent?: Column<TData>
```

The parent column. `undefined` if this is a root column.

`getFlatColumns`

```
type getFlatColumns = () => Column<TData>[]
```

Returns a flattened array including this column and all descendants.

`getLeafColumns`

```
type getLeafColumns = () => Column<TData>[]
```

Returns an array of all leaf columns. If no children, itself is the only leaf.

Edit on GitHub

On this page

Additional Content (Omitted for brevity)

HeaderGroup APIs

These are **core** options and API properties for all header groups. More options and API properties may be available for other table features.

Header Group API

All header group objects have the following properties:

`id`

tsx

```
id: string
```

Description: The unique identifier for the header group.

`depth`

tsx

```
depth: number
```

Description: The depth of the header group, zero-indexed based.

`headers`

tsx

```
type headers = Header<TData>[]
```

Description: An array of Header objects that belong to this header group.

Edit on GitHub

On this page

Header APIs | TanStack Table Docs

On this page

Header API

All header objects have the following properties:

id

```
id: string
```

The unique identifier for the header.

index

```
index: number
```

The index for the header within the header group.

depth

```
depth: number
```

The depth of the header, zero-indexed.

column

```
column: Column<TData>
```

The header's associated Column object.

headerGroup

```
headerGroup: HeaderGroup<TData>
```

The header's associated HeaderGroup object.

subHeaders

```
type subHeaders = Header<TData>[]
```

The hierarchical sub/child headers; empty if the column is a leaf-column.

colSpan

```
colSpan: number
```

The column span value for the header.

rowSpan

```
rowSpan: number
```

The row span value for the header.

getLeafHeaders

```
type getLeafHeaders = () => Header<TData>[]
```

Returns the hierarchically nested leaf headers under this header.

isPlaceholder

```
isPlaceholder: boolean
```

Indicates if the header is a placeholder header.

placeholderId

```
placeholderId?: string
```

If a placeholder, a unique header ID that does not conflict with others.

getContext

```
getContext: () => {
  table: Table<TData>
  header: Header<TData, TValue>
  column: Column<TData, TValue>
}
```

Provides the context (props) for rendering headers, footers, and filters; used with `flexRender`.

```
flexRender(header.column.columnDef.header, header.getContext())
```

Table API

getHeaderGroups

```
type getHeaderGroups = () => HeaderGroup<TData>[]
```

Returns all header groups in the table.

getLeftHeaderGroups

```
type getLeftHeaderGroups = () => HeaderGroup<TData>[]
```

Returns header groups for the pinned (left) columns.

getCenterHeaderGroups

```
type getCenterHeaderGroups = () => HeaderGroup<TData>[]
```

Returns header groups for columns not pinned.

getRightHeaderGroups

```
type getRightHeaderGroups = () => HeaderGroup<TData>[]
```

Returns header groups for right pinned columns.

getFooterGroups

```
type getFooterGroups = () => HeaderGroup<TData>[]
```

Returns footer groups.

getLeftFooterGroups

```
type getLeftFooterGroups = () => HeaderGroup<TData>[]
```

Footer groups for left pinned columns.

getCenterFooterGroups

```
type getCenterFooterGroups = () => HeaderGroup<TData>[]
```

Footer groups for non-pinned columns.

getRightFooterGroups

```
type getRightFooterGroups = () => HeaderGroup<TData>[]
```

Footer groups for right pinned columns.

getFlatHeaders

```
type getFlatHeaders = () => Header<TData, unknown>[]
```

Headers of all columns, including parent headers.

getLeftFlatHeaders

```
type getLeftFlatHeaders = () => Header<TData, unknown>[]
```

Headers of all left pinned columns, including parent headers.

getCenterFlatHeaders

```
type getCenterFlatHeaders = () => Header<TData, unknown>[]
```

Headers of columns not pinned, including parents.

getRightFlatHeaders

```
type getRightFlatHeaders = () => Header<TData, unknown>[]
```

Headers of right pinned columns, including parents.

getLeafHeaders

```
type getLeafHeaders = () => Header<TData, unknown>[]
```

Headers for all leaf columns.

getLeftLeafHeaders

```
type getLeftLeafHeaders = () => Header<TData, unknown>[]
```

Left pinned leaf headers.

getCenterLeafHeaders

```
type getCenterLeafHeaders = () => Header<TData, unknown>[]
```

Non-pinned leaf headers.

getRightLeafHeaders

```
type getRightLeafHeaders = () => Header<TData, unknown>[]
```

Right pinned leaf headers.

Edit on GitHub

Row APIs

On this page

Row API

All row objects have the following properties:

`id`

`string`

The resolved unique identifier for the row resolved via the `options.getRowId` option. Defaults to the row's index (or relative index if it is a subRow).

`depth`

`number`

The depth of the row (if nested or grouped) relative to the root row array.

`index`

`number`

The index of the row within its parent array (or the root data array).

`original`

`TData`

The original row object provided to the table.

🧠 If the row is a grouped row, the original row object will be the first original in the group.

parentId

`string` *(optional)*

If nested, this row's parent row id.

getValue

`(columnId: string) => TValue`

Returns the value from the row for a given `columnId`.

renderValue

`(columnId: string) => TValue`

Renders the value from the row for a given `columnId`, but will return the `renderFallbackValue` if no value is found.

getUniqueValues

`(columnId: string) => TValue[]`

Returns a unique array of values from the row for a given `columnId`.

subRows

`type subRows = Row<TData>[]`

An array of subRows for the row as returned and created by the `options.getSubRows` option.

getParentRow

`() => Row<TData> | undefined`

Returns the parent row for the row, if it exists.

getParentRows

`() => Row<TData>[]`

Returns the parent rows for the row, all the way up to a root row.

getLeafRows

`() => Row<TData>[]`

Returns the leaf rows for the row, not including any parent rows.

originalSubRows

`originalSubRows?: TData[]`

An array of the original subRows as returned by the `options.getSubRows` option.

`getAllCells`

```
() => Cell<TData>[]
```

Returns all of the Cells for the row.

Edit on GitHub

On this page

Cell APIs

Overview

These are **core** options and API properties for all cells. More options and API properties are available for other table features.

Cell API

All cell objects have the following properties:

`id`

```
id: string
```

The unique ID for the cell across the entire table.

`getValue`

```
getValue: () => any
```

Returns the value for the cell, accessed via the associated column's accessor key or accessor function.

`renderValue`

```
renderValue: () => any
```

Renders the value for a cell the same as `getValue`, but will return the `renderFallbackValue` if no value is found.

`row`

```
row: Row<TData>
```

The associated Row object for the cell.

`column`

```
column: Column<TData>
```

The associated Column object for the cell.

`getContext`

```
getContext: () => {
  table: Table<TData>;
  column: Column<TData, TValue>;
  row: Row<TData>;
  cell: Cell<TData, TValue>;
  getValue: <TTValue = TValue>() => TTValue;
  renderValue: <TTValue = TValue>() => TTValue | null;
}
```

Returns the rendering context (or props) for cell-based components like cells and aggregated cells. Use these props with your framework's `flexRender` utility to render these using the template of your choice:

```
flexRender(cell.column.columnDef.cell, cell.getContext())
```

Edit on GitHub

On this page

Additional references

- Row
- Column Filtering

Our Partners

Subscribe to Bytes

Column Filtering APIs | TanStack Table Docs

Navigation Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Introduction
- Overview
- Installation
- Migrating to V8
- FAQ
- React Table Adapter

Core Guides

- Data
- Column Defs
- Table Instance
- Row Models
- Rows
- Cells
- Header Groups
- Headers
- Columns
- Table State (React)

Feature Guides

- Column Ordering
- Column Pinning
- Column Sizing
- Column Visibility
- Column Filtering
- Global Filtering
- Fuzzy Filtering
- Column Faceting
- Global Faceting
- Grouping
- Expanding
- Pagination
- Row Pinning
- Row Selection

- Sorting
- Virtualization
- Custom Features

Core APIs

- Column Def
- Table
- Column
- Header Group
- Header
- Row
- Cell

Feature APIs

- Column Filtering
- Column Faceting
- Column Ordering
- Column Pinning
- Column Sizing
- Column Visibility
- Global Faceting
- Global Filtering
- Sorting
- Grouping
- Expanding
- Pagination
- Row Pinning
- Row Selection

Enterprise

- AG Grid

Examples

- Basic (React)
- Header Groups (React)
- Column Filters (React)
- Column Filters (Faceted) (React)
- Fuzzy Search Filters (React)
- Column Ordering (React)
- Column DnD (React)
- Column Pinning (React)
- Sticky Column Pinning (React)
- Column Sizing (React)
- Performant Column Resizing (React)
- Column Visibility (React)
- Editable Data (React)
- Expanding (React)
- Sub Components (React)
- Fully Controlled (React)
- Grouping (React)

- Pagination (React)
- Pagination Controlled (React)
- Row DnD (React)
- Row Pinning (React)
- Row Selection (React)
- Sorting (React)
- Virtualized Columns (React)
- Virtualized Columns (Experimental) (React)
- Virtualized Rows (React)
- Virtualized Rows (Experimental) (React)
- Virtualized Infinite Scrolling (React)
- Kitchen Sink (React)
- React Bootstrap (React)
- Material UI Pagination (React)
- Full Width Table (React)
- Full Width Resizable Table (React)
- Custom Features (React)
- Query Router Search Params (React)

On this page

Can-Filter

The ability for a column to be **column** filtered is determined by the following:

- The column was defined with a valid `accessorKey` / `accessorFn`.
- `column.enableColumnFilter` is not set to `false`.
- `options.enableColumnFilters` is not set to `false`.
- `options.enableFilters` is not set to `false`.

State

Filter state is stored on the table using the following shape:

```
export interface ColumnFiltersTableState {
  columnFilters: ColumnFiltersState
}

export type ColumnFiltersState = ColumnFilter[]

export interface ColumnFilter {
  id: string
  value: unknown
}
```

Filter Functions

The following filter functions are built-in to the table core:

- **includesString**
  Case-insensitive string inclusion

- **includesStringSensitive**
  Case-sensitive string inclusion

- **equalsString**
  Case-insensitive string equality
- **equalsStringSensitive**
  Case-sensitive string equality
- **arrIncludes**
  Item inclusion within an array
- **arrIncludesAll**
  All items included in an array
- **arrIncludesSome**
  Some items included in an array
- **equals**
  Object/referential equality `Object.is` / `===`
- **weakEquals**
  Weak object/referential equality `==`
- **inNumberRange**
  Number range inclusion

Every filter function receives:

- The row to filter
- The columnId to retrieve the row's value
- The filter value

and should return `true` if the row should be included, `false` if filtered out.

The type signature for filter functions:

```
export type FilterFn<TData extends AnyData> = (
  row: Row<TData>,
  columnId: string,
  filterValue: any,
  addMeta: (meta: any) => void
) => boolean
```

Optional helper types:

```
export type TransformFilterValueFn<TData extends AnyData> = (
  value: any,
  column?: Column<TData>
) => unknown

export type ColumnFilterAutoRemoveTestFn<TData extends AnyData> = (
  value: any,
  column?: Column<TData>
) => boolean

export type CustomFilterFns<TData extends AnyData> = Record<string, FilterFn<TData>>
```

---

filterFn.resolveFilterValue

This optional method on any filterFn allows the function to transform/sanitize/format the filter value before use.

filterFn.autoRemove

An optional method that takes a filter value and returns `true` if the filter value should be removed from the filter state (e.g., when a boolean filter is set to `false`).

Using Filter Functions

Filter functions can be used by passing:

- A string referencing a built-in filter function
- A custom filter function directly

to `columnDefinition.filterFn`.

Final filter function options type:

```
export type FilterFnOption<TData extends AnyData> =
  | 'auto'
  | BuiltInFilterFn
  | FilterFn<TData>
```

Filter Meta

Filtering can expose additional info about data for ranking or sorting, e.g., similar to match-sorter.

`filterFn` can optionally mark results with a **filter meta** using `addMeta()`, aiding in sorting/grouping.

Example usage with `match-sorter` utilities:

```
import { sortingFns } from '@tanstack/react-table'
import { rankItem, compareItems } from '@tanstack/match-sorter-utils'

const fuzzyFilter = (row, columnId, value, addMeta) => {
  const itemRank = rankItem(row.getValue(columnId), value)
  addMeta(itemRank)
  return itemRank.passed
}

const fuzzySort = (rowA, rowB, columnId) => {
  let dir = 0
  if (rowA.columnFiltersMeta[columnId]) {
    dir = compareItems(
      rowA.columnFiltersMeta[columnId],
      rowB.columnFiltersMeta[columnId]
    )
  }
  return dir === 0 ? sortingFns.alphanumeric(rowA, rowB, columnId) : dir
}
```

Column Def Options

filterFn

```
filterFn?: FilterFn | keyof FilterFns | keyof BuiltInFilterFns
```

Sets the filter function for the column, options:

- String referencing a built-in filter
- Custom function

enableColumnFilter

```
enableColumnFilter?: boolean
```

Enables/disables column filtering for that column.

Column API

getCanFilter

```
getCanFilter: () => boolean
```

Returns whether the column can be filtered.

getFilterIndex

```
getFilterIndex: () => number
```

Returns index (or -1) of the column filter in `state.columnFilters`.

getIsFiltered

```
getIsFiltered: () => boolean
```

Checks if the column is currently filtered.

getFilterValue

```
getFilterValue: () => unknown
```

Returns current filter value.

setFilterValue

```
setFilterValue: (updater: Updater<any>) => void
```

Sets filter value, accepting direct value or updater function.

getAutoFilterFn

```
getAutoFilterFn: (columnId: string) => FilterFn<TData> | undefined
```

Returns auto-calculated filter function based on the column data.

getFilterFn

`getFilterFn: (columnId: string) => FilterFn<TData> | undefined`

Returns the filter function for specified column.

Row API

columnFilters

`columnFilters: Record<string, boolean>`

Tracks whether each column passes filters.

columnFiltersMeta

`columnFiltersMeta: Record<string, any>`

Stores optional meta info for filtering per row.

Table Options

filterFns

`filterFns?: Record<string, FilterFn>`

Define custom filter functions accessible by key in `filterFn`.

Example:

```
declare module '@tanstack/[adapter]-table' {
  interface FilterFns {
    myCustomFilter: FilterFn<unknown>
  }
}

const column = columnHelper.data('key', {
  filterFn: 'myCustomFilter',
})

const table = useReactTable({
  columns: [column],
  filterFns: {
    myCustomFilter: (rows, columnIds, filterValue) => {
      // filter logic
    },
  },
})
```

filterFromLeafRows

`filterFromLeafRows?: boolean`

Defaults to false (filter from parent). `true` filters from leaf rows upwards.

maxLeafRowFilterDepth

```
maxLeafRowFilterDepth?: number
```

Limits filtering to a specific depth level, 0 for root only, 1 for immediate children, etc.

enableFilters

```
enableFilters?: boolean
```

Enables/disables all filtering.

manualFiltering

```
manualFiltering?: boolean
```

Disables internal filtering; useful for server-side filtering support.

onColumnFiltersChange

```
onColumnFiltersChange?: OnChangeFn<ColumnFiltersState>
```

Callback when filters change, allowing external control.

enableColumnFilters

```
enableColumnFilters?: boolean
```

Enables/disables all column filters.

getFilteredRowModel

```
getFilteredRowModel?: (table: Table<TData>) => () => RowModel<TData>
```

Provides custom filtered row model logic. For server-side filtering, can ignore; for client-side, required.

Example:

```
import { getFilteredRowModel } from '@tanstack/[adapter]-table'

getFilteredRowModel: getFilteredRowModel()
```

Table API

setColumnFilters

```
setColumnFilters: (updater: Updater<ColumnFiltersState>) => void
```

Updates filter state.

resetColumnFilters

```
resetColumnFilters: (defaultState?: boolean) => void
```

Resets filters to default or empty.

getPreFilteredRowModel

```
getPreFilteredRowModel: () => RowModel<TData>
```

Returns row model before filtering.

getFilteredRowModel

```
getFilteredRowModel: () => RowModel<TData>
```

Returns row model after filtering.

Edit on GitHub

Edit on GitHub

Related Links

- Cell
- Column Faceting

Our Partners

- React Data Grid (by ag-Grid)

Resources

- TanStack Query: Powerful async state management
- TanStack DB: Extended query capabilities

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

 Subscribe No spam. Unsubscribe at *any* time.

Column Faceting APIs

On this page

Column API

getFacetedRowModel

```
type getFacetedRowModel = () => RowModel<TData>
```

> ⚠️ Requires that you pass a valid `getFacetedRowModel` function to `options.facetedRowModel`. A default implementation is provided via the exported `getFacetedRowModel` function.
>
> Returns the row model with all other column filters applied, excluding its own filter. Useful for displaying faceted result counts.

getFacetedUniqueValues

```
getFacetedUniqueValues: () => Map<any, number>
```

> ⚠️ Requires that you pass a valid `getFacetedUniqueValues` function to `options.getFacetedUniqueValues`. A default implementation is provided via the exported `getFacetedUniqueValues` function.
>
> A function that *computes and returns* a `Map` of unique values and their occurrences derived from `column.getFacetedRowModel`. Useful for displaying faceted result values.

getFacetedMinMaxValues

```
getFacetedMinMaxValues: () => Map<any, number>
```

> ⚠️ Requires that you pass a valid `getFacetedMinMaxValues` function to `options.getFacetedMinMaxValues`. A default implementation is provided via the exported `getFacetedMinMaxValues` function.
>
> A function that *computes and returns* a min/max tuple derived from `column.getFacetedRowModel`. Useful for displaying faceted result values.

Table Options

getColumnFacetedRowModel

```
getColumnFacetedRowModel: (columnId: string) => RowModel<TData>
```

Returns the faceted row model for a given `columnId`.

Edit on GitHub

On this page

Expanding APIs | TanStack Table Docs

Overview

Expanding state is stored on the table using the following shape:

```
export type ExpandedState = true | Record<string, boolean>;

export type ExpandedTableState = {
  expanded: ExpandedState;
};
```

Row API

toggleExpanded

tsx

```
toggleExpanded: (expanded?: boolean) => void
```

Toggles the expanded state (or sets it if `expanded` is provided) for the row.

getIsExpanded

tsx

```
getIsExpanded: () => boolean
```

Returns whether the row is expanded.

getIsAllParentsExpanded

tsx

```
getIsAllParentsExpanded: () => boolean
```

Returns whether all parent rows of the row are expanded.

getCanExpand

tsx

```
getCanExpand: () => boolean
```

Returns whether the row can be expanded.

getToggleExpandedHandler

tsx

```
getToggleExpandedHandler: () => () => void
```

Returns a function that can be used to toggle the expanded state of the row. This function can be used to bind to an event handler to a button.

Table Options

manualExpanding

tsx

```
manualExpanding?: boolean
```

Enables manual row expansion. If this is set to `true`, `getExpandedRowModel` will not be used to expand rows and you would be expected to perform the expansion in your own data model. This is useful if you are doing server-side expansion.

onExpandedChange

tsx

```
onExpandedChange?: OnChangeFn<ExpandedState>
```

This function is called when the `expanded` table state changes. If a function is provided, you will be responsible for managing this state on your own. To pass the managed state back to the table, use the `tableOptions.state.expanded` option.

autoResetExpanded

tsx

```
autoResetExpanded?: boolean
```

Enable this setting to automatically reset the expanded state of the table when expanding state changes.

enableExpanding

tsx

```
enableExpanding?: boolean
```

Enable/disable expanding for all rows.

getExpandedRowModel

tsx

```
getExpandedRowModel?: (table: Table<TData>) => () => RowModel<TData>
```

This function is responsible for returning the expanded row model. If this function is not provided, the table will not expand rows. You can use the default exported `getExpandedRowModel` function to get the expanded row model or implement your own.

getIsRowExpanded

tsx

```
getIsRowExpanded?: (row: Row<TData>) => boolean
```

If provided, allows you to override the default behavior of determining whether a row is currently expanded.

getRowCanExpand

tsx

```
getRowCanExpand?: (row: Row<TData>) => boolean
```

If provided, allows you to override the default behavior of determining whether a row can be expanded.

paginateExpandedRows

tsx

```
paginateExpandedRows?: boolean
```

If `true`, expanded rows will be paginated along with the rest of the table (which means expanded rows may span multiple pages).
If `false`, expanded rows will not be considered for pagination (which means expanded rows will always render on their parents' page. This also means more rows will be rendered than the set page size).

Table API

setExpanded

tsx

```
setExpanded: (updater: Updater<ExpandedState>) => void
```

Updates the expanded state of the table via an update function or value.

toggleAllRowsExpanded

tsx

```
toggleAllRowsExpanded: (expanded?: boolean) => void
```

Toggles the expanded state for all rows. Optionally, provide a value to set the expanded state to.

resetExpanded

tsx

```
resetExpanded: (defaultState?: boolean) => void
```

Reset the expanded state of the table to the initial state. If `defaultState` is provided, the expanded state will be reset to `{}`.

getCanSomeRowsExpand

tsx

```
getCanSomeRowsExpand: () => boolean
```

Returns whether there are any rows that can be expanded.

getToggleAllRowsExpandedHandler

tsx

```
getToggleAllRowsExpandedHandler: () => (event: unknown) => void
```

Returns a handler that can be used to toggle the expanded state of all rows. This handler is meant to be used with an `<input type=checkbox>` element.

getIsSomeRowsExpanded

tsx

```
getIsSomeRowsExpanded: () => boolean
```

Returns whether there are any rows that are currently expanded.

getIsAllRowsExpanded

tsx

```
getIsAllRowsExpanded: () => boolean
```

Returns whether all rows are currently expanded.

getExpandedDepth

tsx

```
getExpandedDepth: () => number
```

Returns the maximum depth of the expanded rows.

getExpandedRowModel

tsx

```
getExpandedRowModel: () => RowModel<TData>
```

Returns the row model after expansion has been applied.

getPreExpandedRowModel

tsx

```
getPreExpandedRowModel: () => RowModel<TData>
```

Returns the row model before expansion has been applied.

Additional Resources

Edit on GitHub

On this page

- Row API
- toggleExpanded
- getIsExpanded
- getIsAllParentsExpanded
- getCanExpand
- getToggleExpandedHandler
- Table Options
- manualExpanding
- onExpandedChange
- autoResetExpanded
- enableExpanding
- getExpandedRowModel
- getIsRowExpanded
- getRowCanExpand
- paginateExpandedRows
- Table API
- setExpanded
- toggleAllRowsExpanded
- resetExpanded
- getCanSomeRowsExpand
- getToggleAllRowsExpandedHandler
- getIsSomeRowsExpanded
- getIsAllRowsExpanded
- getExpandedDepth
- getExpandedRowModel
- getPreExpandedRowModel

Partners

Our Partners

- React Data Grid - AG Grid

Additional Resources

- TanStack Query: Powerful asynchronous state management, server-state utilities, and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state".
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations that keep your UI reactive, consistent, and blazing fast 🔥.

Subscription

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

 Subscribe *No spam. Unsubscribe at any time.*

Pagination APIs

Table of Contents

State

Pagination state is stored on the table using the following shape:

```
export type PaginationState = {
  pageIndex: number
  pageSize: number
}

export type PaginationTableState = {
  pagination: PaginationState
}

export type PaginationInitialTableState = {
  pagination?: Partial<PaginationState>
}
```

Table Options

manualPagination

Enables manual pagination. If this option is set to *true*, the table will not automatically paginate rows using `getPaginationRowModel()` and instead will expect you to manually paginate the rows before passing them to the table. This is useful if you are doing server-side pagination and aggregation.

```
manualPagination?: boolean
```

---

pageCount

When manually controlling pagination, you can supply a total `pageCount` value to the table if you know it. If you do not know how many pages there are, set this to `-1`. Alternatively, provide a `rowCount` and the table will calculate the `pageCount` internally.

```
pageCount?: number
```

---

rowCount

When manually controlling pagination, supply a total `rowCount` if known. `pageCount` will be calculated internally from `rowCount` and `pageSize`.

```
rowCount?: number
```

---

autoResetPageIndex

If set to *true*, pagination will reset to the first page when page-altering state changes, e.g., `data` update, filters change, grouping changes, etc.

```
autoResetPageIndex?: boolean
```

> **Note:** Defaults to *false* if `manualPagination` is set to *true*.

---

onPaginationChange

A callback function that is called when the pagination state changes, allowing you to manage the state yourself.

```
onPaginationChange?: OnChangeFn<PaginationState>
```

---

getPaginationRowModel

Returns the row model after pagination has taken place; no further pagination processing is applied.

```
getPaginationRowModel?: (table: Table<TData>) => () => RowModel<TData>
```

Table API

setPagination

Sets or updates the `state.pagination`.

```
setPagination: (updater: Updater<PaginationState>) => void
```

---

resetPagination

Resets the pagination state to `initialState.pagination`, or resets to a blank state if `true` is passed.

```
resetPagination: (defaultState?: boolean) => void
```

---

setPageIndex

Updates the page index.

```
setPageIndex: (updater: Updater<number>) => void
```

---

resetPageIndex

Resets the page index to its initial state. If `true` is provided, resets to `0`.

```
resetPageIndex: (defaultState?: boolean) => void
```

---

setPageSize

Updates the page size.

```
setPageSize: (updater: Updater<number>) => void
```

---

resetPageSize

Resets the page size to its initial value, defaulting to `10`.

```
resetPageSize: (defaultState?: boolean) => void
```

---

getPageOptions

Returns available page options based on current page size.

```
getPageOptions: () => number[]
```

---

getCanPreviousPage

Returns whether the table can go to the previous page.

```
getCanPreviousPage: () => boolean
```

---

getCanNextPage

Returns whether the table can go to the next page.

```
getCanNextPage: () => boolean
```

---

previousPage

Decrements the page index by one, if possible.

```
previousPage: () => void
```

---

nextPage

Increments the page index by one, if possible.

```
nextPage: () => void
```

---

firstPage

Sets the page index to `0`.

```
firstPage: () => void
```

---

lastPage

Sets the page index to the last available page.

```
lastPage: () => void
```

---

getPageCount

Returns total page count. For manually controlled pagination, this comes from `options.pageCount`; otherwise, it's calculated from total rows and `pageSize`.

```
getPageCount: () => number
```

---

getPrePaginationRowModel

Returns the row model before pagination.

```
getPrePaginationRowModel: () => RowModel<TData>
```

---

getPaginationRowModel

Returns the row model after pagination.

```
getPaginationRowModel: () => RowModel<TData>
```

Additional Resources

Edit on GitHub

On this page

- Table Options
- manualPagination
- pageCount
- rowCount
- autoResetPageIndex
- onPaginationChange
- getPaginationRowModel
- Table API
- setPagination
- resetPagination
- setPageIndex
- resetPageIndex
- setPageSize
- resetPageSize
- getPageOptions
- getCanPreviousPage
- getCanNextPage
- previousPage
- nextPage
- firstPage
- lastPage
- getPageCount
- getPrePaginationRowModel
- getPaginationRowModel

Our Partners

Additional Resources

- TanStack Query: Powerful asynchronous state management, server-state utilities, and data fetching.
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic mutations.

Subscribe to Bytes

Row Pinning APIs | TanStack Table Docs

Can-Pin

The ability for a row to be **pinned** is determined by the following:

- `options.enableRowPinning` resolves to **true**
- `options.enablePinning` is not set to **false**

State

Pinning state is stored on the table using the following shape:

```
export type RowPinningPosition = false | 'top' | 'bottom'

export type RowPinningState = {
  top?: string[]
  bottom?: string[]
}

export type RowPinningRowState = {
  rowPinning: RowPinningState
}
```

Table Options

enableRowPinning

```
enableRowPinning?: boolean | ((row: Row<TData>) => boolean)
```

Enables/disables row pinning for all rows in the table.

keepPinnedRows

```
keepPinnedRows?: boolean
```

When **false**, pinned rows will not be visible if they are filtered or paginated out of the table. When **true**, pinned rows will always be visible regardless of filtering or pagination. Defaults to **true**.

onRowPinningChange

```
onRowPinningChange?: OnChangeFn<RowPinningState>
```

If provided, this function will be called with an **updaterFn** when `state.rowPinning` changes. This overrides the default internal state management, so you will also need to supply `state.rowPinning` from your own managed state.

Table API

setRowPinning

```
setRowPinning: (updater: Updater<RowPinningState>) => void
```

Sets or updates the `state.rowPinning` state.

resetRowPinning

```
resetRowPinning: (defaultState?: boolean) => void
```

Resets the **rowPinning** state to `initialState.rowPinning`, or **true** can be passed to force a default blank state reset to `{}`.

getIsSomeRowsPinned

```
getIsSomeRowsPinned: (position?: RowPinningPosition) => boolean
```

Returns whether or not any rows are pinned. Optionally specify to only check for pinned rows in either the **top** or **bottom** position.

getTopRows

```
getTopRows: () => Row<TData>[]
```

Returns all top pinned rows.

getBottomRows

```
getBottomRows: () => Row<TData>[]
```

Returns all bottom pinned rows.

getCenterRows

```
getCenterRows: () => Row<TData>[]
```

Returns all rows that are not pinned to the top or bottom.

Row API

pin

```
pin: (position: RowPinningPosition) => void
```

Pins a row to the **'top'** or **'bottom'**, or unpins the row to the center if **false** is passed.

getCanPin

```
getCanPin: () => boolean
```

Returns whether or not the row can be pinned.

getIsPinned

```
getIsPinned: () => RowPinningPosition
```

Returns the pinned position of the row. ('top', 'bottom', or false)

getPinnedIndex

```
getPinnedIndex: () => number
```

Returns the numeric pinned index of the row within a pinned row group.

Edit on GitHub

On this page

Additional Info

Table Options

- `enableRowPinning?: boolean | ((row: Row<TData>) => boolean)`
- `keepPinnedRows?: boolean`
- `onRowPinningChange?: OnChangeFn<RowPinningState>`

Row API

- `pin: (position: RowPinningPosition) => void`
- `getCanPin: () => boolean`
- `getIsPinned: () => RowPinningPosition`
- `getPinnedIndex: () => number`

State shape

```
export type RowPinningPosition = false | 'top' | 'bottom'

export type RowPinningState = {
  top?: string[]
  bottom?: string[]
}
```

```
export type RowPinningRowState = {
  rowPinning: RowPinningState
}
```

Row Selection APIs | TanStack Table Docs

Introduction

Row selection state is stored on the table using the following shape:

```
export type RowSelectionState = Record<string, boolean>;

export type RowSelectionTableState = {
  rowSelection: RowSelectionState;
}
```

By default, the row selection state uses the index of each row as the row identifiers. Row selection state can instead be tracked with a custom unique row id by passing in a custom getRowId function to the table.

Table Options

enableRowSelection

```
enableRowSelection?: boolean | ((row: Row<TData>) => boolean)
```

- Enables/disables row selection for all rows in the table OR
- A function that given a row, returns whether to enable/disable row selection for that row

enableMultiRowSelection

```
enableMultiRowSelection?: boolean | ((row: Row<TData>) => boolean)
```

- Enables/disables multiple row selection for all rows in the table OR
- A function that given a row, returns whether to enable/disable multiple row selection for that row's children/grandchildren

enableSubRowSelection

```
enableSubRowSelection?: boolean | ((row: Row<TData>) => boolean)
```

- Enables/disables automatic sub-row selection when a parent row is selected, or a function that enables/disables automatic sub-row selection for each row.
- *(Use in combination with expanding or grouping features)*

onRowSelectionChange

```
onRowSelectionChange?: OnChangeFn<RowSelectionState>
```

- If provided, this function will be called with an *updaterFn* when `state.rowSelection` changes.
- This overrides the default internal state management, so you will need to persist the state change either fully or partially outside of the table.

Table API

getToggleAllRowsSelectedHandler

`getToggleAllRowsSelectedHandler: () => (event: unknown) => void`

- Returns a handler that can be used to toggle all rows in the table.

getToggleAllPageRowsSelectedHandler

`getToggleAllPageRowsSelectedHandler: () => (event: unknown) => void`

- Returns a handler that can be used to toggle all rows on the current page.

setRowSelection

`setRowSelection: (updater: Updater<RowSelectionState>) => void`

- Sets or updates the `state.rowSelection` state.

resetRowSelection

`resetRowSelection: (defaultState?: boolean) => void`

- Resets the **rowSelection** state to the `initialState.rowSelection`, or `true` can be passed to force a default blank state reset to `{}`.

getIsAllRowsSelected

`getIsAllRowsSelected: () => boolean`

- Returns whether or not all rows in the table are selected.

getIsAllPageRowsSelected

`getIsAllPageRowsSelected: () => boolean`

- Returns whether or not all rows on the current page are selected.

getIsSomeRowsSelected

`getIsSomeRowsSelected: () => boolean`

- Returns whether or not some of the rows in the table are selected.

getIsSomePageRowsSelected

`getIsSomePageRowsSelected: () => boolean`

- Returns whether or not any rows on the current page are selected.

toggleAllRowsSelected

`toggleAllRowsSelected: (value: boolean) => void`

- Selects/deselects all rows in the table.

toggleAllPageRowsSelected

```
toggleAllPageRowsSelected: (value: boolean) => void
```

- Selects/deselects all rows on the current page.

getPreSelectedRowModel

```
getPreSelectedRowModel: () => RowModel<TData>
```

getSelectedRowModel

```
getSelectedRowModel: () => RowModel<TData>
```

getFilteredSelectedRowModel

```
getFilteredSelectedRowModel: () => RowModel<TData>
```

getGroupedSelectedRowModel

```
getGroupedSelectedRowModel: () => RowModel<TData>
```

Row API

getIsSelected

```
getIsSelected: () => boolean
```

- Returns whether or not the row is selected.

getIsSomeSelected

```
getIsSomeSelected: () => boolean
```

- Returns whether or not some of the row's sub-rows are selected.

getIsAllSubRowsSelected

```
getIsAllSubRowsSelected: () => boolean
```

- Returns whether or not all of the row's sub-rows are selected.

getCanSelect

```
getCanSelect: () => boolean
```

- Returns whether or not the row can be selected.

getCanMultiSelect

```
getCanMultiSelect: () => boolean
```

- Returns whether or not the row can be multi-selected.

getCanSelectSubRows

```
getCanSelectSubRows: () => boolean
```

- Returns whether or not the row can automatically select sub-rows when the parent row is selected.

toggleSelected

```
toggleSelected: (value?: boolean) => void
```

- Selects/deselects the row.

getToggleSelectedHandler

```
getToggleSelectedHandler: () => (event: unknown) => void
```

- Returns a handler that can be used to toggle the row.

Edit on GitHub

On this page

Column Ordering APIs

On this page

State

Column ordering state is stored on the table using the following shape:

```
export type ColumnOrderTableState = {
  columnOrder: ColumnOrderState
}

export type ColumnOrderState = string[]
```

Table Options

onColumnOrderChange

`onColumnOrderChange?: OnChangeFn<ColumnOrderState>`

If provided, this function will be called with an `updaterFn` when `state.columnOrder` changes. This overrides the default internal state management, so you will need to persist the state change either fully or partially outside of the table.

Table API

setColumnOrder

`setColumnOrder: (updater: Updater<ColumnOrderState>) => void`

Sets or updates the `state.columnOrder`.

resetColumnOrder

`resetColumnOrder: (defaultState?: boolean) => void`

Resets the **columnOrder** state to `initialState.columnOrder`, or `true` can be passed to force a default blank state reset to `[]`.

Column API

getIndex

```
getIndex: (position?: ColumnPinningPosition) => number
```

Returns the index of the column in the order of the visible columns. Optionally pass a `position` parameter to get the index of the column in a sub-section of the table.

getIsFirstColumn

```
getIsFirstColumn: (position?: ColumnPinningPosition) => boolean
```

Returns `true` if the column is the first column in the order of the visible columns. Optionally pass a `position` parameter to check if the column is the first in a sub-section of the table.

getIsLastColumn

```
getIsLastColumn: (position?: ColumnPinningPosition) => boolean
```

Returns `true` if the column is the last column in the order of the visible columns. Optionally pass a `position` parameter to check if the column is the last in a sub-section of the table.

Edit on GitHub

On this page

Column Pinning APIs

TanStack Table Docs

On this page

Can-Pin

The ability for a column to be *pinned* is determined by the following:

- `options.enablePinning` is not set to `false`
- `options.enableColumnPinning` is not set to `false`
- `columnDefinition.enablePinning` is not set to `false`

State

Pinning state is stored on the table using the following shape:

```
export type ColumnPinningPosition = false | 'left' | 'right'

export type ColumnPinningState = {
  left?: string[]
  right?: string[]
}

export type ColumnPinningTableState = {
  columnPinning: ColumnPinningState
}
```

Table Options

- **enableColumnPinning**:

  `enableColumnPinning?: boolean`

  Enables/disables column pinning for all columns in the table.

- **onColumnPinningChange**:

  `onColumnPinningChange?: OnChangeFn<ColumnPinningState>`

  If provided, this function will be called with an `updaterFn` when `state.columnPinning` changes. Overrides default internal state management; you need to manage `state.columnPinning` yourself.

Column Def Options

- **enablePinning**:

  `enablePinning?: boolean`

  Enables/disables pinning for the column.

Table API

- **setColumnPinning**

  `setColumnPinning: (updater: Updater<ColumnPinningState>) => void`

  Sets or updates the `state.columnPinning`.

- **resetColumnPinning**

  `resetColumnPinning: (defaultState?: boolean) => void`

  Resets the `columnPinning` state to `initialState.columnPinning`, or passes `true` to force reset to `{ left: [], right: [] }`.

- **getIsSomeColumnsPinned**

  `getIsSomeColumnsPinned: (position?: ColumnPinningPosition) => boolean`

Returns whether any columns are pinned. Optionally specify to check only in `'left'` or `'right'`. *Note: Does not account for column visibility.*

Header and Footer Groups

- **getLeftHeaderGroups**:

  ```
  () => HeaderGroup<TData>[]
  ```

  Returns the left pinned header groups.

- **getCenterHeaderGroups**:

  ```
  () => HeaderGroup<TData>[]
  ```

  Returns the unpinned/center header groups.

- **getRightHeaderGroups**:

  ```
  () => HeaderGroup<TData>[]
  ```

  Returns the right pinned header groups.

- Similar functions exist for Footer groups: `getLeftFooterGroups`, `getCenterFooterGroups`, `getRightFooterGroups`.

Flat Headers & Leaf Headers

- **getLeftFlatHeaders**:

  ```
  () => Header<TData>[]
  ```

  Flat array of left pinned headers including parents.

- **getCenterFlatHeaders**: Same for unpinned/center headers.
- **getRightFlatHeaders**: Same for right pinned headers.
- **getLeftLeafHeaders**: Flat array of leaf-node left pinned headers.
- **getCenterLeafHeaders**: Leaf-node unpinned/center headers.
- **getRightLeafHeaders**: Leaf-node right pinned headers.

Similarly, columns can be accessed via:

- **getLeftLeafColumns**
- **getRightLeafColumns**
- **getCenterLeafColumns**

Column API

- **getCanPin**:

  ```
  () => boolean
  ```

  Returns whether or not the column can be pinned.

- **getPinnedIndex**:

  ```
  () => number
  ```

Returns the numeric pinned index within a pinned group.

- **getIsPinned**:

  ```
  () => ColumnPinningPosition
  ```

  Returns the pinned position ('left', 'right', or `false`).

- **pin**:

  ```
  (position: ColumnPinningPosition) => void
  ```

  Pins or unpins the column to `'left'`, `'right'`, or unpins if `false` is passed.

Row API

- **getLeftVisibleCells**:

  ```
  () => Cell<TData>[]
  ```

  Returns all left pinned leaf cells in the row.

- **getRightVisibleCells**:

  ```
  () => Cell<TData>[]
  ```

  Returns all right pinned leaf cells.

- **getCenterVisibleCells**:

  ```
  () => Cell<TData>[]
  ```

  Returns all center pinned (unpinned) leaf cells.

Edit on GitHub

Column Sizing APIs | TanStack Table Docs

On this page

State

Column sizing state is stored on the table using the following shape:

```
export type ColumnSizingTableState = {
  columnSizing: ColumnSizing
  columnSizingInfo: ColumnSizingInfoState
}

export type ColumnSizing = Record<string, number>

export type ColumnSizingInfoState = {
```

```
  startOffset: null | number
  startSize: null | number
  deltaOffset: null | number
  deltaPercentage: null | number
  isResizingColumn: false | string
  columnSizingStart: [string, number][]
}
```

Column Def Options

enableResizing

Type: `enableResizing?: boolean`
Enables or disables column resizing for the column.

---

size

Type: `size?: number`
The desired size for the column.

---

minSize

Type: `minSize?: number`
The minimum allowed size for the column.

---

maxSize

Type: `maxSize?: number`
The maximum allowed size for the column.

Column API

getSize

Type: `getSize: () => number`
Returns the current size of the column.

---

getStart

Type: `getStart: (position?: ColumnPinningPosition) => number`
Returns the offset measurement along the row-axis (usually the x-axis for standard tables) for the column, measuring the size of all preceding columns.
Useful for sticky or absolute positioning of columns (e.g., `left` or `transform`).

---

getAfter

Type: `getAfter: (position?: ColumnPinningPosition) => number`
Returns the offset measurement along the row-axis (usually the x-axis for standard tables) for the column, measuring the size of all succeeding columns.
Useful for sticky or absolute positioning of columns (e.g., `right` or `transform`).

---

getCanResize

Type: `getCanResize: () => boolean`
Returns `true` if the column can be resized.

---

getIsResizing

Type: `getIsResizing: () => boolean`
Returns `true` if the column is currently being resized.

---

resetSize

Type: `resetSize: () => void`
Resets the column size to its initial size.

Header API

getSize

Type: `getSize: () => number`
Returns the size for the header, calculated by summing the size of all leaf-columns that belong to it.

---

getStart

Type: `getStart: (position?: ColumnPinningPosition) => number`
Returns the offset measurement along the row-axis (usually the x-axis for standard tables) for the header.
This is effectively a sum of the offset measurements of all preceding headers.

---

getResizeHandler

Type: `getResizeHandler: () => (event: unknown) => void`
Returns an event handler function that can be used to resize the header.
It can be used as:

- `onMouseDown` handler
- `onTouchStart` handler

The dragging and release events are automatically handled for you.

Table Options

enableColumnResizing

Type: `enableColumnResizing?: boolean`
Enables/disables column resizing for *all columns*.

---

columnResizeMode

Type: `columnResizeMode?: 'onChange' | 'onEnd'`
Determines when the `columnSizing` state is updated:

- `'onChange'` updates the state continuously while dragging.
- `'onEnd'` updates the state once the resize handle is released.

---

columnResizeDirection

Type: `columnResizeDirection?: 'ltr' | 'rtl'`
Enables or disables right-to-left support for resizing the column (defaults to `'ltr'`).

---

onColumnSizingChange

Type: `onColumnSizingChange?: OnChangeFn<ColumnSizingState>`
Optional callback invoked when the `columnSizing` state changes.
If provided, you are responsible for managing the state. You can pass this state back to the table via `state.columnSizing`.

---

onColumnSizingInfoChange

Type: `onColumnSizingInfoChange?: OnChangeFn<ColumnSizingInfoState>`
Optional callback invoked when the `columnSizingInfo` state changes.
If provided, you are responsible for managing the state. You can pass this state back to the table via `state.columnSizingInfo`.

Table API

setColumnSizing

Type: `setColumnSizing: (updater: Updater<ColumnSizingState>) => void`
Sets the column sizing state using an updater function or a value.
This triggers `onColumnSizingChange` if provided, otherwise the table manages the state automatically.

---

setColumnSizingInfo

Type: `setColumnSizingInfo: (updater: Updater<ColumnSizingInfoState>) => void`
Sets the column sizing info state similarly, triggering `onColumnSizingInfoChange` if provided.

---

resetColumnSizing

Type: `resetColumnSizing: (defaultState?: boolean) => void`
Resets column sizing to its initial state.
If `defaultState` is `true`, resets to default table state.

---

resetHeaderSizeInfo

Type: `resetHeaderSizeInfo: (defaultState?: boolean) => void`
Resets column sizing info to its initial state.
If `defaultState` is `true`, resets to default table state.

---

getTotalSize

Type: `getTotalSize: () => number`
Returns the total size of the table by summing all leaf-column sizes.

---

getLeftTotalSize

Type: `getLeftTotalSize: () => number`
If pinning, returns the total size of the left pinned columns.

---

getCenterTotalSize

Type: `getCenterTotalSize: () => number`
If pinning, returns the total size of unpinned/center columns.

---

getRightTotalSize

Type: `getRightTotalSize: () => number`
If pinning, returns the total size of right pinned columns.

Edit on GitHub

Column Visibility APIs

State

Column visibility state is stored on the table using the following shape:

```
export type VisibilityState = Record<string, boolean>;

export type VisibilityTableState = {
  columnVisibility: VisibilityState;
}
```

Column Def Options

enableHiding

Enables/disables hiding the column

```
enableHiding?: boolean
```

Description

Enables/disables hiding the column.

Column API

getCanHide

Returns whether the column can be hidden.

```
getCanHide: () => boolean
```

getIsVisible

Returns whether the column is visible.

```
getIsVisible: () => boolean
```

toggleVisibility

Toggles the column visibility.

```
toggleVisibility: (value?: boolean) => void
```

getToggleVisibilityHandler

Returns a function that can be used to toggle the column visibility. This function can be used to bind to an event handler to a checkbox.

```
getToggleVisibilityHandler: () => (event: unknown) => void
```

Table Options

onColumnVisibilityChange

If provided, this function will be called with an `updaterFn` when `state.columnVisibility` changes. This overrides the default internal state management, so you will need to persist the state change either fully or partially outside of the table.

```
onColumnVisibilityChange?: OnChangeFn<VisibilityState>
```

enableHiding

Enables/disables hiding of columns.

```
enableHiding?: boolean
```

Table API

getVisibleFlatColumns

Returns a flat array of columns that are visible, including parent columns.

```
getVisibleFlatColumns: () => Column<TData>[]
```

getVisibleLeafColumns

Returns a flat array of leaf-node columns that are visible.

```
getVisibleLeafColumns: () => Column<TData>[]
```

getLeftVisibleLeafColumns

If column pinning, returns a flat array of leaf-node columns that are visible in the left portion of the table.

```
getLeftVisibleLeafColumns: () => Column<TData>[]
```

getRightVisibleLeafColumns

If column pinning, returns a flat array of leaf-node columns that are visible in the right portion of the table.

```
getRightVisibleLeafColumns: () => Column<TData>[]
```

getCenterVisibleLeafColumns

If column pinning, returns a flat array of leaf-node columns that are visible in the unpinned/center portion of the table.

```
getCenterVisibleLeafColumns: () => Column<TData>[]
```

setColumnVisibility

Updates the column visibility state via an updater function or value.

```
setColumnVisibility: (updater: Updater<VisibilityState>) => void
```

resetColumnVisibility

Resets the column visibility state to the initial state. If `defaultState` is provided, the state will be reset accordingly.

```
resetColumnVisibility: (defaultState?: boolean) => void
```

toggleAllColumnsVisible

Toggles the visibility of all columns.

```
toggleAllColumnsVisible: (value?: boolean) => void
```

getIsAllColumnsVisible

Returns whether all columns are visible.

```
getIsAllColumnsVisible: () => boolean
```

getIsSomeColumnsVisible

Returns whether some columns are visible.

```
getIsSomeColumnsVisible: () => boolean
```

getToggleAllColumnsVisibilityHandler

Returns a handler for toggling the visibility of all columns, meant to be bound to an `<input type="checkbox">` element.

```
getToggleAllColumnsVisibilityHandler: () => (event: unknown) => void
```

Row API

getVisibleCells

Returns an array of cells that account for column visibility for the row.

```
getVisibleCells: () => Cell<TData>[]
```

Edit on GitHub

On this page

- enableHiding
- Table API
- getVisibleFlatColumns
- getVisibleLeafColumns
- getLeftVisibleLeafColumns
- getRightVisibleLeafColumns
- getCenterVisibleLeafColumns
- setColumnVisibility
- resetColumnVisibility
- toggleAllColumnsVisible
- getIsAllColumnsVisible
- getIsSomeColumnsVisible
- getToggleAllColumnsVisibilityHandler
- Row API
- getVisibleCells

Global Faceting APIs

On this page

Table API

getGlobalFacetedRowModel

```
getGlobalFacetedRowModel: () => RowModel<TData>
```

Returns the faceted row model for the global filter.

getGlobalFacetedUniqueValues

```
getGlobalFacetedUniqueValues: () => Map<any, number>
```

Returns the faceted unique values for the global filter.

getGlobalFacetedMinMaxValues

```
getGlobalFacetedMinMaxValues: () => [number, number]
```

Returns the faceted min and max values for the global filter.

Edit on GitHub

Additional on this page

- Table API
- getGlobalFacetedRowModel
- getGlobalFacetedUniqueValues
- getGlobalFacetedMinMaxValues

Related Links

- Column Visibility
- Global Filtering

Our Partners

- Introducing TanStack Query
  Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state"

- TanStack DB
  TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥

Global Filtering APIs | TanStack Table Docs

Navigation

Can-Filter

The ability for a column to be **globally** filtered is determined by the following:

- The column was defined with a valid `accessorKey` or `accessorFn`.
- If `options.getColumnCanGlobalFilter` is provided, it must return `true` for the column. Otherwise, the column is assumed to be globally filterable if the first row's value is a `string` or `number`.
- `column.enableColumnFilter` is not set to `false`.
- `options.enableColumnFilters` is not set to `false`.
- `options.enableFilters` is not set to `false`.

State

Filter state is stored on the table with the shape:

```
export interface GlobalFilterTableState {
  globalFilter: any
}
```

Filter Functions

You can use the same filter functions available for column filtering for global filtering. See the Column Filtering guide to learn more.

Using Filter Functions

Filter functions can be referenced or defined by passing the following to `options.globalFilterFn`:

- A string referencing a built-in filter function.
- A function provided directly.

The list of available filter functions `tableOptions.globalFilterFn` supports:

```
export type FilterFnOption<TData extends AnyData> =
  | 'auto'
  | BuiltInFilterFn
  | FilterFn<TData>
```

Filter Meta

Filtering can expose additional data used to augment or optimize operations like ranking, sorting, or grouping. This is done by calling the `addMeta` function provided to custom `filterFn`s.

Example: Ranking and Filtering

```
import { sortingFns } from '@tanstack/[adapter]-table'
import { rankItem, compareItems } from '@tanstack/match-sorter-utils'

const fuzzyFilter = (row, columnId, value, addMeta) => {
  const itemRank = rankItem(row.getValue(columnId), value)
  addMeta(itemRank)
  return itemRank.passed
}

const fuzzySort = (rowA, rowB, columnId) => {
  let dir = 0
  if (rowA.columnFiltersMeta[columnId]) {
    dir = compareItems(
      rowA.columnFiltersMeta[columnId],
      rowB.columnFiltersMeta[columnId]
    )
  }
  return dir === 0 ? sortingFns.alphanumeric(rowA, rowB, columnId) : dir
}
```

Column Def Options

enableGlobalFilter

```
enableGlobalFilter?: boolean
```

Enables or disables the global filter for this column.

Column API

getCanGlobalFilter

```
getCanGlobalFilter: () => boolean
```

Returns whether the column can be globally filtered. Set to `false` to exclude from global filtering.

Row API

columnFiltersMeta

```
columnFiltersMeta: Record<string, any>
```

Stores filter meta information for a row, used in ranking/sorting.

Table Options

filterFns

```
filterFns?: Record<string, FilterFn>
```

Allows defining custom filter functions by key. Example:

```
declare module '@tanstack/table-core' {
  interface FilterFns {
    myCustomFilter: FilterFn<unknown>
  }
}

const column = columnHelper.data('key', {
  filterFn: 'myCustomFilter',
})

const table = useReactTable({
  columns: [column],
  filterFns: {
    myCustomFilter: (rows, columnIds, filterValue) => {
      // return filtered rows
    },
  },
})
```

filterFromLeafRows

```
filterFromLeafRows?: boolean
```

Default: `false`. If `true`, filtering starts from leaf rows upward; otherwise, from parent rows downward.

maxLeafRowFilterDepth

```
maxLeafRowFilterDepth?: number
```

Default: `100`. Limits filtering depth:

- `0`: only root level rows
- `1`: only first child level
- etc.

enableFilters

```
enableFilters?: boolean
```

Enables/disables all filters.

manualFiltering

```
manualFiltering?: boolean
```

Disables the use of `getFilteredRowModel`, useful for toggling between client and server filtering.

getFilteredRowModel

```
getFilteredRowModel?: (table: Table<TData>) => () => RowModel<TData>
```

Custom function for generating filtered row model. Used for server-side filtering.

globalFilterFn

```
globalFilterFn?: FilterFn | keyof FilterFns | keyof BuiltInFilterFns
```

Specifies the filter function for global filtering.

onGlobalFilterChange

```
onGlobalFilterChange?: OnChangeFn<GlobalFilterState>
```

Called when `globalFilter` state changes, overriding internal state management.

enableGlobalFilter

```
enableGlobalFilter?: boolean
```

Enable/disable the global filter functionality.

getColumnCanGlobalFilter

```
getColumnCanGlobalFilter?: (column: Column<TData>) => boolean
```

Custom logic to determine if a column is eligible for global filtering.

Table API

getPreFilteredRowModel

```
getPreFilteredRowModel: () => RowModel<TData>
```

Provides the row model before filtering.

getFilteredRowModel

```
getFilteredRowModel: () => RowModel<TData>
```

Provides filtered row model.

setGlobalFilter

```
setGlobalFilter: (updater: Updater<any>) => void
```

Updates the global filter state.

resetGlobalFilter

```
resetGlobalFilter: (defaultState?: boolean) => void
```

Resets to initial state.

getGlobalAutoFilterFn

```
getGlobalAutoFilterFn: (columnId: string) => FilterFn<TData> | undefined
```

Returns the default filter function.

getGlobalFilterFn

```
getGlobalFilterFn: (columnId: string) => FilterFn<TData> | undefined
```

Returns the current global filter function assigned to a column.

External Links

Edit on GitHub

Related Sections

- Global Faceting
- Sorting

Partners

- TanStack Query: Powerful asynchronous state management.
- TanStack DB: Extends TanStack Query with collections, live queries, and optimistic updates.

Subscription

Subscribe to Bytes

Your weekly dose of JavaScript news, delivered every Monday to over 100,000 developers for free.
No spam. Unsubscribe at *any* time.

Sorting APIs | TanStack Table Docs

On this page

Sorting APIs

State

Sorting state is stored on the table using the following shape:

```ts
export type SortDirection = 'asc' | 'desc'

export type ColumnSort = {
  id: string
  desc: boolean
}

export type SortingState = ColumnSort[]

export type SortingTableState = {
  sorting: SortingState
}
```

Sorting Functions

The following sorting functions are built-in to the table core:

- **alphanumeric**
  - Sorts by mixed alphanumeric values without case-sensitivity. Slower, but more accurate if strings contain numbers needing natural sort.
- **alphanumericCaseSensitive**
  - Sorts by mixed alphanumeric values with case-sensitivity. Slower, but more precise with case distinctions.
- **text**
  - Sorts by text/string values without case-sensitivity. Faster, but less precise with numbers.
- **textCaseSensitive**
  - Sorts text/string values with case-sensitivity. Faster, but less precise with numbers.
- **datetime**
  - Sorts by time; use if values are `Date` objects.
- **basic**
  - Uses standard comparison: `a > b ? 1 : a < b ? -1 : 0`. Fastest but less precise.

Every sorting function receives two rows and a column ID and compares them to return `-1`, `0`, or `1` in ascending order.

Comparison cheat sheet:

Return	Ascending Order
`-1`	`a < b`
`0`	`a === b`
`1`	`a > b`

Type Signature

```ts
export type SortingFn<TData extends AnyData> = (
  rowA: Row<TData>,
  rowB: Row<TData>,
```

```
  columnId: string
) => number
```

Using Sorting Functions

Sorting functions can be referenced or defined by passing:

- A `string` referencing a built-in sorting function
- A `string` referencing a custom function via `tableOptions.sortingFns`
- A function directly in `columnDefinition.sortingFn`

Example of custom sorting function:

```
declare module '@tanstack/table-core' {
  interface SortingFns {
    myCustomSorting: SortingFn<unknown>
  }
}

const column = columnHelper.data('key', {
  sortingFn: 'myCustomSorting',
})

const table = useReactTable({
  columns: [column],
  sortingFns: {
    myCustomSorting: (rowA, rowB, columnId) =>
      rowA.getValue(columnId).value < rowB.getValue(columnId).value ? 1 : -1
  },
})
```

Column Def Options

sortingFn

```
sortingFn?: SortingFn | keyof SortingFns | keyof BuiltInSortingFns
```

- Implements the sorting for this column.
- Options:
  - String referencing a built-in or custom function
  - Direct function implementation

sortDescFirst

```
sortDescFirst?: boolean
```

- Defaults sorting toggle to descending first if set to `true`.

enableSorting

```
enableSorting?: boolean
```

- Enable or disable sorting on this column.

enableMultiSort

`enableMultiSort?: boolean`

- Enable or disable multi-column sorting.

invertSorting

`invertSorting?: boolean`

- Inverts the sorting order for this column, useful for ranking or inverted scales.

sortUndefined

`sortUndefined?: 'first' | 'last' | false | -1 | 1 // defaults to 1`

- Controls placement of undefined values:
  - `'first'`: on top
  - `'last'`: at bottom
  - `false`: consider tied, sort by next filter or index
  - `-1` or `1`: sort with higher/lower priority

Note: `'first'` *and* `'last'` *options are new in v8.16.0.*

Column API

getAutoSortingFn

`getAutoSortingFn: () => SortingFn<TData>`

- Returns an automatic sorting function based on column values.

getAutoSortDir

`getAutoSortDir: () => SortDirection`

- Returns inferred sort direction based on values.

getSortingFn

`getSortingFn: () => SortingFn<TData>`

- Returns the actual sorting function used.

getNextSortingOrder

`getNextSortingOrder: () => SortDirection | false`

- Provides the next ordering in cycle.

getCanSort

`getCanSort: () => boolean`

- Indicates if column can be sorted.

getCanMultiSort

```
getCanMultiSort: () => boolean
```

- Indicates if column supports multi-sort.

getSortIndex

```
getSortIndex: () => number
```

- Position of this column's sorting within current sorting stack.

getIsSorted

```
getIsSorted: () => false | SortDirection
```

- Whether column is sorted and in which direction.

getFirstSortDir

```
getFirstSortDir: () => SortDirection
```

- Default direction for sorting.

clearSorting

```
clearSorting: () => void
```

- Remove sorting from this column.

toggleSorting

```
toggleSorting: (desc?: boolean, isMulti?: boolean) => void
```

- Toggle sorting state, optionally forcing direction and multi-sort.

getToggleSortingHandler

```
getToggleSortingHandler: () => undefined | ((event: unknown) => void)
```

- Function to attach for toggling header sorting.

Table Options

sortingFns

```
sortingFns?: Record<string, SortingFn>
```

- Register custom sorting functions for reference via name.

Example:

```
declare module '@tanstack/table-core' {
  interface SortingFns {
    myCustomSorting: SortingFn<unknown>
  }
```

```
}

const table = useReactTable({
  columns: [/* ... */],
  sortingFns: {
    myCustomSorting: (rowA, rowB, columnId) =>
      rowA.getValue(columnId).value < rowB.getValue(columnId).value ? 1 : -1,
  }
})
```

manualSorting

```
manualSorting?: boolean
```

- Enable manual sorting: data expected to be pre-sorted before passing to table.

onSortingChange

```
onSortingChange?: OnChangeFn<SortingState>
```

- Callback invoked on sorting state change, allows persisting external state.

enableSorting

```
enableSorting?: boolean
```

- Enable or disable sorting for the overall table.

enableSortingRemoval

```
enableSortingRemoval?: boolean
```

- Allow removing sorting state once set.

enableMultiRemove

```
enableMultiRemove?: boolean
```

- Allow removing multiple sorting columns at once.

enableMultiSort

```
enableMultiSort?: boolean
```

- Enable multi-column sorting.

sortDescFirst

```
sortDescFirst?: boolean
```

- Default to descending first toggle.

getSortedRowModel

```
getSortedRowModel?: (table: Table<TData>) => () => RowModel<TData>
```

- Retrieves sorted row model, for client-side sorting.

maxMultiSortColCount

```
maxMultiSortColCount?: number
```

- Limits number of multi-sorted columns.

isMultiSortEvent

```
isMultiSortEvent?: (e: unknown) => boolean
```

- Custom function to detect multi-sort toggle events.

Table API

setSorting

```
setSorting: (updater: Updater<SortingState>) => void
```

- Programmatically sets sorting state.

resetSorting

```
resetSorting: (defaultState?: boolean) => void
```

- Resets sorting to initial or blank state.

getPreSortedRowModel

```
getPreSortedRowModel: () => RowModel<TData>
```

- Row model before sorting, used for server-side.

getSortedRowModel

```
getSortedRowModel: () => RowModel<TData>
```

- Row model after sorting, used for client-side.

Edit on GitHub

Grouping APIs | TanStack Table Docs

Navigation

Home
Table v8
Auto

Search and Menu

Search...

- K

Menu

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Introduction (core)
- Overview (core)
- Installation (core)
- Migrating to V8 (core)
- FAQ (core)
- React Table Adapter (react)

Core Guides

- Data (core)
- Column Defs (core)
- Table Instance (core)
- Row Models (core)
- Rows (core)
- Cells (core)
- Header Groups (core)
- Headers (core)
- Columns (core)
- Table State (react)

Feature Guides

- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Column Filtering (core)
- Global Filtering (core)
- Fuzzy Filtering (core)
- Column Faceting (core)
- Global Faceting (core)
- Grouping (core)
- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)
- Sorting (core)
- Virtualization (core)
- Custom Features (core)

Core APIs

- Column Def (core)
- Table (core)
- Column (core)
- Header Group (core)
- Header (core)
- Row (core)
- Cell (core)

Feature APIs

- Column Filtering (core)
- Column Faceting (core)
- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Global Faceting (core)
- Global Filtering (core)
- Sorting (core)
- Grouping (core)
- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)

Enterprise

- AG Grid (core)

Examples

- Basic (react)

- Header Groups (react)
- Column Filters (react)
- Column Filters (Faceted) (react)
- Fuzzy Search Filters (react)
- Column Ordering (react)
- Column Ordering (DnD) (react)
- Column Pinning (react)
- Sticky Column Pinning (react)
- Column Sizing (react)
- Performant Column Resizing (react)
- Column Visibility (react)
- Editable Data (react)
- Expanding (react)
- Sub Components (react)
- Fully Controlled (react)
- Grouping (react)
- Pagination (react)
- Pagination Controlled (react)
- Row DnD (react)
- Row Pinning (react)
- Row Selection (react)
- Sorting (react)
- Virtualized Columns (react)
- Virtualized Columns (Experimental) (react)
- Virtualized Rows (react)
- Virtualized Rows (Experimental) (react)
- Virtualized Infinite Scrolling (react)
- Kitchen Sink (react)
- React Bootstrap (react)
- Material UI Pagination (react)
- Full Width (react)
- Full Width Resizable (react)
- Custom Features (react)
- Query Router Search Params (react)

On this page

- groupingValue
- getIsGrouped
- getGroupingValue
- Table Options
- aggregationFns
- manualGrouping
- onGroupingChange
- enableGrouping
- getGroupedRowModel
- groupedColumnMode
- Table API
- setGrouping
- resetGrouping
- getPreGroupedRowModel
- getGroupedRowModel
- Cell API
- getIsAggregated
- getIsGrouped
- getIsPlaceholder

Additional Links

Edit on GitHub

AG Grid - An alternative enterprise data-grid solution

While we clearly love TanStack Table, we acknowledge that it is not a "batteries included" product packed with customer support and enterprise polish. We realize that some of our users may need this though! To help out here, we want to introduce you to **AG Grid**, an enterprise-grade data grid solution that can supercharge your applications with its extensive feature set and robust performance. While TanStack Table is also a powerful option for implementing data grids, we believe in providing our users with a diverse range of choices that best fit their specific requirements. AG Grid is one such choice, and we're excited to highlight its capabilities for you.

Why Choose AG Grid?

Here are some good reasons to consider AG Grid for your next project:

Comprehensive Feature Set

AG Grid offers an extensive set of features, making it a versatile and powerful data grid solution. With AG Grid, you get access to a wide range of functionalities that cater to the needs of complex enterprise applications. From advanced sorting, filtering, and grouping capabilities to column pinning, multi-level headers, and tree data structure support, AG Grid provides you with the tools to create dynamic and interactive data grids that meet your application's unique demands.

High Performance

When it comes to handling large datasets and achieving exceptional performance, AG Grid delivers outstanding results. It employs highly optimized rendering techniques, efficient data updates, and virtualization to ensure smooth scrolling and fast response times, even when dealing with thousands or millions of rows of data. AG Grid's performance optimizations make it an excellent choice for applications that require high-speed data manipulation and visualization.

Customization and Extensibility

AG Grid is designed to be highly customizable and extensible, allowing you to tailor the grid to your specific needs. It provides a rich set of APIs and events that enable you to integrate custom functionality seamlessly. You can define custom cell renderers, editors, filters, and aggregators to enhance the grid's behavior and appearance. AG Grid also supports a variety of themes, allowing you to match the grid's visual style to your application's design.

Support for Enterprise Needs

As an enterprise-focused solution, AG Grid caters to the requirements of complex business applications. It offers enterprise-specific features such as row grouping, column pinning, server-side row model, master/detail grids, and rich editing capabilities. AG Grid also integrates well with other enterprise frameworks and libraries, making it a reliable choice for large-scale projects.

Active Development and Community Support

AG Grid benefits from active development and a thriving community of developers. The team behind AG Grid consistently introduces new features and enhancements, ensuring that the product evolves to meet the

changing needs of the industry. The community support is robust, with forums, documentation, and examples readily available to assist you in utilizing the full potential of AG Grid.

Conclusion

While TanStack Table remains a powerful and flexible option for implementing data grids, we understand that different projects have different requirements. AG Grid offers a compelling enterprise-grade solution that may be particularly suited to your needs. Its comprehensive feature set, high performance, customization options, and focus on enterprise requirements make AG Grid an excellent choice for projects that demand a robust and scalable data grid solution.

We encourage you to explore AG Grid further by visiting their website and trying out their demo. Remember that both TanStack Table and AG Grid have their unique strengths and considerations. We believe in providing options to our users, empowering you to make informed decisions and choose the best fit for your specific use case.

Visit the AG Grid website.

Edit on GitHub

On this page

React Example: Basic

Github
StackBlitz
CodeSandbox

Code Explorer | **Code** | **Interactive Sandbox** | **Sandbox**

File Structure:

- **src**
  - index.css
  - main.tsx
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

```
import * as React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  createColumnHelper,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'

type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}

const defaultData: Person[] = [
  {
    firstName: 'tanner',
    lastName: 'linsley',
    age: 24,
    visits: 100,
    status: 'In Relationship',
    progress: 50,
  },
  {
    firstName: 'tandy',
```

```
    lastName: 'miller',
    age: 40,
    visits: 40,
    status: 'Single',
    progress: 80,
  },
  {
    firstName: 'joe',
    lastName: 'dirte',
    age: 45,
    visits: 20,
    status: 'Complicated',
    progress: 10,
  },
]

const columnHelper = createColumnHelper<Person>()

const columns = [
  columnHelper.accessor('firstName', {
    cell: info => info.getValue(),
    footer: info => info.column.id,
  }),
  columnHelper.accessor(row => row.lastName, {
    id: 'lastName',
    cell: info => <i>{info.getValue()}</i>,
    header: () => <span>Last Name</span>,
    footer: info => info.column.id,
  }),
  columnHelper.accessor('age', {
    header: () => 'Age',
    cell: info => info.renderValue(),
    footer: info => info.column.id,
  }),
  columnHelper.accessor('visits', {
    header: () => <span>Visits</span>,
    footer: info => info.column.id,
  }),
  columnHelper.accessor('status', {
    header: 'Status',
    footer: info => info.column.id,
  }),
  columnHelper.accessor('progress', {
    header: 'Profile Progress',
    footer: info => info.column.id,
  }),
]

function App() {
  const [data, _setData] = React.useState(() => [...defaultData])
  const rerender = React.useReducer(() => ({}), {})[1]

  const table = useReactTable({
    data,
    columns,
```

```
  getCoreRowModel: getCoreRowModel(),
})

return (
  <div className="p-2">
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => (
              <th key={header.id}>
                {header.isPlaceholder
                  ? null
                  : flexRender(
                      header.column.columnDef.header,
                      header.getContext()
                    )}
              </th>
            ))}
          </tr>
        ))}
      </thead>
      <tbody>
        {table.getRowModel().rows.map(row => (
          <tr key={row.id}>
            {row.getVisibleCells().map(cell => (
              <td key={cell.id}>
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </td>
            ))}
          </tr>
        ))}
      </tbody>
      <tfoot>
        {table.getFooterGroups().map(footerGroup => (
          <tr key={footerGroup.id}>
            {footerGroup.headers.map(header => (
              <th key={header.id}>
                {header.isPlaceholder
                  ? null
                  : flexRender(
                      header.column.columnDef.footer,
                      header.getContext()
                    )}
              </th>
            ))}
          </tr>
        ))}
      </tfoot>
    </table>
    <div className="h-4" />
    <button onClick={() => rerender()} className="border p-2">
      Rerender
    </button>
  </div>
```

```
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Column Groups

Github
StackBlitz
CodeSandbox

Code Explorer | Interactive Sandbox

Files

- **src**
  - `index.css`
  - `main.tsx`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Example Code (TypeScript)

```
import * as React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  createColumnHelper,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'

type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}

const defaultData: Person[] = [
  {
    firstName: 'tanner',
    lastName: 'linsley',
    age: 24,
    visits: 100,
    status: 'In Relationship',
    progress: 50,
```

```
  },
  {
    firstName: 'tandy',
    lastName: 'miller',
    age: 40,
    visits: 40,
    status: 'Single',
    progress: 80,
  },
  {
    firstName: 'joe',
    lastName: 'dirte',
    age: 45,
    visits: 20,
    status: 'Complicated',
    progress: 10,
  },
]

const columnHelper = createColumnHelper<Person>()

const columns = [
  columnHelper.group({
    id: 'hello',
    header: () => <span>Hello</span>,
    columns: [
      columnHelper.accessor('firstName', {
        cell: info => info.getValue(),
        footer: props => props.column.id,
      }),
      columnHelper.accessor(row => row.lastName, {
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      }),
    ],
  }),
  columnHelper.group({
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      columnHelper.accessor('age', {
        header: () => 'Age',
        footer: props => props.column.id,
      }),
      columnHelper.group({
        header: 'More Info',
        columns: [
          columnHelper.accessor('visits', {
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          }),
          columnHelper.accessor('status', {
            header: 'Status',
```

```
            footer: props => props.column.id,
          }),
          columnHelper.accessor('progress', {
            header: 'Profile Progress',
            footer: props => props.column.id,
          }),
        ],
      }),
    ],
  }),
]

function App() {
  const [data, _setData] = React.useState(() => [...defaultData])
  const rerender = React.useReducer(() => ({}), {})[1]

  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(),
  })

  return (
    <div className="p-2">
      <table>
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => (
            <tr key={row.id}>
              {row.getVisibleCells().map(cell => (
                <td key={cell.id}>
                  {flexRender(cell.column.columnDef.cell, cell.getContext())}
                </td>
              ))}
            </tr>
          ))}
        </tbody>
        <tfoot>
          {table.getFooterGroups().map(footerGroup => (
            <tr key={footerGroup.id}>
```

```
              {footerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.footer,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </tfoot>
      </table>
      <div className="h-4" />
      <button onClick={() => rerender()} className="border p-2">
        Rerender
      </button>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Column Resizing Performant

Github | StackBlitz | CodeSandbox

Code Explorer | Code | Interactive Sandbox

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

tsx

```
import React from 'react'
import ReactDOM from 'react-dom/client'


import './index.css'



import {
  useReactTable,
  getCoreRowModel,
  ColumnDef,
  flexRender,
  Table,
} from '@tanstack/react-table'
import { makeData } from './makeData'



type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}



const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: props => props.column.id,
    columns: [
```

```
      {
        accessorKey: 'firstName',
        cell: info => info.getValue(),
        footer: props => props.column.id,
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: props => props.column.id,
      },
      {
        accessorKey: 'visits',
        header: () => <span>Visits</span>,
        footer: props => props.column.id,
      },
      {
        accessorKey: 'status',
        header: 'Status',
        footer: props => props.column.id,
      },
      {
        accessorKey: 'progress',
        header: 'Profile Progress',
        footer: props => props.column.id,
      },
    ],
  },
]


function App() {
  const [data, _setData] = React.useState(() => makeData(200))
  const [columns] = React.useState<typeof defaultColumns>(() => [
    ...defaultColumns,
  ])



  const rerender = React.useReducer(() => ({}), {})[1]



  const table = useReactTable({
    data,
    columns,
    defaultColumn: {
```

```
    minSize: 60,
    maxSize: 800,
  },
  columnResizeMode: 'onChange',
  getCoreRowModel: getCoreRowModel(),
  debugTable: true,
  debugHeaders: true,
  debugColumns: true,
})

/**
```

- Instead of calling `column.getSize()` on every render for every header
- and especially every data cell (very expensive),
- we will calculate all column sizes at once at the root table level in a useMemo
- and pass the column sizes down as CSS variables to the <table> element.

```
     */
    const columnSizeVars = React.useMemo(() => {
  const headers = table.getFlatHeaders()
  const colSizes: { [key: string]: number } = {}
  for (let i = 0; i < headers.length; i++) {
   const header = headers[i]!
   colSizes[`--header-${header.id}-size`] = header.getSize()
   colSizes[`--col-${header.column.id}-size`] = header.column.getSize()
  }
  return colSizes
    }, [table.getState().columnSizingInfo, table.getState().columnSizing])



//demo purposes
const [enableMemo, setEnableMemo] = React.useState(true)


return (
```

This example has artificially slow cell renders to simulate complex usage

```
      Memoize Table Body:{' '}
      <input
        type="checkbox"
        checked={enableMemo}
        onChange={() => setEnableMemo(!enableMemo)}
      />
```

```
      <button onClick={() => rerender()} className="border p-2">
        Rerender

      <pre style={{ minHeight: '10rem' }}>
        {JSON.stringify(
          {
            columnSizing: table.getState().columnSizing,
          },
          null,
          2
        )}


({data.length} rows)
{/* Here in the <table> equivalent element (surrounds all table head and data
cells), we will define our CSS variables for column sizes */} <div {...{
className: 'divTable', style: { ...columnSizeVars, //Define column sizes on the
<table> element width: table.getTotalSize(), }, }} >
{table.getHeaderGroups().map(headerGroup => ( <div {...{ key: headerGroup.id,
className: 'tr', }} > {headerGroup.headers.map(header => ( <div {...{ key:
header.id, className: 'th', style: { width: calc(var(--header-${header?.id}-size)
* 1px) , }, }} > {header.isPlaceholder ? null : flexRender(
header.column.columnDef.header, header.getContext() )} <div {...{ onDoubleClick:
() => header.column.resetSize(), onMouseDown: header.getResizeHandler(),
onTouchStart: header.getResizeHandler(), className: resizer ${
header.column.getIsResizing() ? 'isResizing' : '' }, }} />
))}
))}
{/ When resizing any column we will render this special memoized version of our
table body */} {table.getState().columnSizingInfo.isResizingColumn && enableMemo
? ( ) : ( )}


) }

//un-memoized normal table body component - see memoized version below function
TableBody({ table }: { table: Table<Person> }) { return ( <div {...{ className:
'tbody', }} > {table.getRowModel().rows.map(row => ( <div {...{ key: row.id,
className: 'tr', }} > {row.getVisibleCells().map(cell => { //simulate expensive
render for (let i = 0; i < 10000; i++) { Math.random() }

        return (
          <div
            {...{
              key: cell.id,
              className: 'td',
              style: {
                width: `calc(var(--col-${cell.column.id}-size) * 1px)`,
              },
            }}
          >
            {cell.renderValue<any>()}
          </div>
        )
```

```
      })}
    </div>
  ))}
</div>

) }

//special memoized wrapper for our table body that we will use during column
resizing export const MemoizedTableBody = React.memo( TableBody, (prev, next) =>
prev.table.options.data === next.table.options.data ) as typeof TableBody

const rootElement = document.getElementById('root') if (!rootElement) throw new
Error('Failed to find the root element')
```

```
ReactDOM.createRoot(rootElement).render( <React.StrictMode> </React.StrictMode>
)
```

React Example: Column Visibility

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- `index.css`
- `main.tsx`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Example Code (TypeScript React)

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'

type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}

const defaultData: Person[] = [
  {
    firstName: 'tanner',
    lastName: 'linsley',
    age: 24,
    visits: 100,
    status: 'In Relationship',
    progress: 50,
  },
```

```
  {
    firstName: 'tandy',
    lastName: 'miller',
    age: 40,
    visits: 40,
    status: 'Single',
    progress: 80,
  },
  {
    firstName: 'joe',
    lastName: 'dirte',
    age: 45,
    visits: 20,
    status: 'Complicated',
    progress: 10,
  },
]

const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        footer: (props) => props.column.id,
      },
      {
        accessorFn: (row) => row.lastName,
        id: 'lastName',
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
```

```
            accessorKey: 'status',
            header: 'Status',
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const [data, setData] = React.useState(() => [...defaultData])
  const [columns] = React.useState<typeof defaultColumns>(() => [
    ...defaultColumns,
  ])
  const [columnVisibility, setColumnVisibility] = React.useState({})

  const rerender = React.useReducer(() => ({}), {})[1]

  const table = useReactTable({
    data,
    columns,
    state: {
      columnVisibility,
    },
    onColumnVisibilityChange: setColumnVisibility,
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
  })

  return (
    <div className="p-2">
      <div className="inline-block border border-black shadow rounded">
        <div className="px-1 border-b border-black">
          <label>
            <input
              {...{
                type: 'checkbox',
                checked: table.getIsAllColumnsVisible(),
                onChange: table.getToggleAllColumnsVisibilityHandler(),
              }}
            />{' '}
            Toggle All
          </label>
        </div>
        {table.getAllLeafColumns().map((column) => {
          return (
            <div key={column.id} className="px-1">
```

```
        <label>
          <input
            {...{
              type: 'checkbox',
              checked: column.getIsVisible(),
              onChange: column.getToggleVisibilityHandler(),
            }}
          />{' '}
          {column.id}
        </label>
      </div>
    )
  })}
</div>
<div className="h-4" />
<table>
  <thead>
    {table.getHeaderGroups().map((headerGroup) => (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map((header) => (
          <th key={header.id} colSpan={header.colSpan}>
            {header.isPlaceholder
              ? null
              : flexRender(
                  header.column.columnDef.header,
                  header.getContext()
                )}
          </th>
        ))}
      </tr>
    ))}
  </thead>
  <tbody>
    {table.getRowModel().rows.map((row) => (
      <tr key={row.id}>
        {row.getVisibleCells().map((cell) => (
          <td key={cell.id}>
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        ))}
      </tr>
    ))}
  </tbody>
  <tfoot>
    {table.getFooterGroups().map((footerGroup) => (
      <tr key={footerGroup.id}>
        {footerGroup.headers.map((header) => (
          <th key={header.id} colSpan={header.colSpan}>
            {header.isPlaceholder
              ? null
              : flexRender(
                  header.column.columnDef.footer,
                  header.getContext()
                )}
          </th>
```

```
              ))}
            </tr>
          ))}
        </tfoot>
      </table>
      <div className="h-4" />
      <button onClick={() => rerender()} className="border p-2">
        Rerender
      </button>
      <div className="h-4" />
      <pre>{JSON.stringify(table.getState().columnVisibility, null, 2)}</pre>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Editable Data

Github
StackBlitz
CodeSandbox

Code Explorer | Code

Interactive Sandbox | Sandbox

Files

- **src**
  - index.css
  - main.tsx
  - makeData.ts
- **.gitignore**
- **README.md**
- **index.html**
- **package.json**
- **tsconfig.json**
- **vite.config.js**

---

main.tsx

```
import React from 'react'
import ReactDOM from 'react-dom/client'

//
import './index.css'

//
import {
  Column,
  Table,
  ColumnDef,
  useReactTable,
  getCoreRowModel,
  getFilteredRowModel,
  getPaginationRowModel,
  flexRender,
  RowData,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

declare module '@tanstack/react-table' {
  interface TableMeta<TData extends RowData> {
    updateData: (rowIndex: number, columnId: string, value: unknown) => void
  }
```

```
}

// Give our default column cell renderer editing superpowers!
const defaultColumn: Partial<ColumnDef<Person>> = {
  cell: ({ getValue, row: { index }, column: { id }, table }) => {
    const initialValue = getValue()
    // We need to keep and update the state of the cell normally
    const [value, setValue] = React.useState(initialValue)

    // When the input is blurred, we'll call our table meta's updateData function
    const onBlur = () => {
      table.options.meta?.updateData(index, id, value)
    }

    // If the initialValue is changed external, sync it up with our state
    React.useEffect(() => {
      setValue(initialValue)
    }, [initialValue])

    return (
      <input
        value={value as string}
        onChange={e => setValue(e.target.value)}
        onBlur={onBlur}
      />
    )
  },
}

function useSkipper() {
  const shouldSkipRef = React.useRef(true)
  const shouldSkip = shouldSkipRef.current

  // Wrap a function with this to skip a pagination reset temporarily
  const skip = React.useCallback(() => {
    shouldSkipRef.current = false
  }, [])

  React.useEffect(() => {
    shouldSkipRef.current = true
  })

  return [shouldSkip, skip] as const
}

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      header: 'Name',
      footer: props => props.column.id,
      columns: [
        {
          accessorKey: 'firstName',
```

```
        footer: props => props.column.id,
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: props => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: props => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: props => props.column.id,
          },
        ],
      },
    ],
  },
], [])

const [data, setData] = React.useState(() => makeData(1000))
const refreshData = () => setData(() => makeData(1000))

const [autoResetPageIndex, skipAutoResetPageIndex] = useSkipper()

const table = useReactTable({
  data,
  columns,
  defaultColumn,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
```

```
  autoResetPageIndex,
  // Provide our updateData function to our table meta
  meta: {
    updateData: (rowIndex, columnId, value) => {
      // Skip page index reset until after next rerender
      skipAutoResetPageIndex()
      setData(old =>
        old.map((row, index) => {
          if (index === rowIndex) {
            return {
              ...old[rowIndex]!,
              [columnId]: value,
            }
          }
          return row
        })
      )
    },
  },
  debugTable: true,
})

return (
  <div className="p-2">
    <div className="h-2" />
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <div>
                      {flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                      {header.column.getCanFilter() ? (
                        <div>
                          <Filter column={header.column} table={table} />
                        </div>
                      ) : null}
                    </div>
                  )}
                </th>
              )
            })}
          </tr>
        ))}
      </thead>
      <tbody>
        {table.getRowModel().rows.map(row => {
          return (
            <tr key={row.id}>
```

```
          {row.getVisibleCells().map(cell => {
            return (
              <td key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
      {table.getPageCount()}
    </strong>
  </span>
  <span className="flex items-center gap-1">
    | Go to page:
    <input
```

```
            type="number"
            min="1"
            max={table.getPageCount()}
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={e => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={e => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map(pageSize => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>{table.getRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
    </div>
  )
}

function Filter({
  column,
  table,
}: {
  column: Column<any, any>
  table: Table<any>
}) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id)

  const columnFilterValue = column.getFilterValue()

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={e =>
          column.setFilterValue((old: [number, number]) => [
```

```
            e.target.value,
            old?.[1],
          ])
        }
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={e =>
          column.setFilterValue((old: [number, number]) => [
            old?.[0],
            e.target.value,
          ])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(columnFilterValue ?? '') as string}
      onChange={e => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Table Expanding Example | TanStack Table Docs

https://github.com/tanstack/table/tree/main/examples/react/expanding
StackBlitz Demo
CodeSandbox

```
import React, { HTMLProps } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Column,
  Table,
  ExpandedState,
  useReactTable,
  getCoreRowModel,
  getPaginationRowModel,
  getFilteredRowModel,
  getExpandedRowModel,
  ColumnDef,
  flexRender,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'firstName',
      header: ({ table }) => (
        <>
          <IndeterminateCheckbox
            {...{
              checked: table.getIsAllRowsSelected(),
              indeterminate: table.getIsSomeRowsSelected(),
              onChange: table.getToggleAllRowsSelectedHandler(),
            }}
          />{' '}
          <button
            {...{
              onClick: table.getToggleAllRowsExpandedHandler(),
            }}
          >
            {table.getIsAllRowsExpanded() ? '👇' : '👉'}
          </button>{' '}
          First Name
        </>
      ),
```

```
  cell: ({ row, getValue }) => (
    <div
      style={{
        // Since rows are flattened by default,
        // we can use the row.depth property
        // and paddingLeft to visually indicate the depth
        // of the row
        paddingLeft: `${row.depth * 2}rem`,
      }}
    >
      <div>
        <IndeterminateCheckbox
          {...{
            checked: row.getIsSelected(),
            indeterminate: row.getIsSomeSelected(),
            onChange: row.getToggleSelectedHandler(),
          }}
        />{' '}
        {row.getCanExpand() ? (
          <button
            {...{
              onClick: row.getToggleExpandedHandler(),
              style: { cursor: 'pointer' },
            }}
          >
            {row.getIsExpanded() ? '👇' : '👉'}
          </button>
        ) : (
          '🔵'
        )}{' '}
        {getValue<boolean>()}
      </div>
    </div>
  ),
  footer: (props) => props.column.id,
},
{
  accessorFn: (row) => row.lastName,
  id: 'lastName',
  cell: (info) => info.getValue(),
  header: () => <span>Last Name</span>,
  footer: (props) => props.column.id,
},
{
  accessorKey: 'age',
  header: () => 'Age',
  footer: (props) => props.column.id,
},
{
  accessorKey: 'visits',
  header: () => <span>Visits</span>,
  footer: (props) => props.column.id,
},
{
  accessorKey: 'status',
```

```
    header: 'Status',
    footer: (props) => props.column.id,
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    footer: (props) => props.column.id,
  },
], [])

const [data, setData] = React.useState(() => makeData(100, 5, 3))
const refreshData = () => setData(() => makeData(100, 5, 3))

const [expanded, setExpanded] = React.useState<ExpandedState>({})

const table = useReactTable({
  data,
  columns,
  state: {
    expanded,
  },
  onExpandedChange: setExpanded,
  getSubRows: (row) => row.subRows,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
  // filterFromLeafRows: true,
  // maxLeafRowFilterDepth: 0,
  debugTable: true,
})

return (
  <div className="p-2">
    <div className="h-2" />
    <table>
      <thead>
        {table.getHeaderGroups().map((headerGroup) => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map((header) => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <div>
                      {flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                      {header.column.getCanFilter() ? (
                        <div>
                          <Filter column={header.column} table={table} />
                        </div>
                      ) : null}
                    </div>
                  )}
```

```
            </th>
          )
        })}
      </tr>
    ))}
  </thead>
  <tbody>
    {table.getRowModel().rows.map((row) => {
      return (
        <tr key={row.id}>
          {row.getVisibleCells().map((cell) => {
            return (
              <td key={cell.id}>
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
```

```
            {table.getState().pagination.pageIndex + 1} of{' '}
            {table.getPageCount()}
          </strong>
        </span>
        <span className="flex items-center gap-1">
          | Go to page:
          <input
            type="number"
            min="1"
            max={table.getPageCount()}
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={(e) => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={(e) => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map((pageSize) => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>{table.getRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <label>Expanded State:</label>
      <pre>{JSON.stringify(expanded, null, 2)}</pre>
      <label>Row Selection State:</label>
      <pre>{JSON.stringify(table.getState().rowSelection, null, 2)}</pre>
    </div>
  )
}

function Filter({ column, table }: { column: Column<any, any>; table: Table<any> }) {
  const firstValue = table.getPreFilteredRowModel().flatRows[0]?.getValue(column.id)

  const columnFilterValue = column.getFilterValue()

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
        type="number"
```

```
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            e.target.value,
            old?.[1],
          ])
        }
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            old?.[0],
            e.target.value,
          ])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(columnFilterValue ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  )
}

function IndeterminateCheckbox({
  indeterminate,
  className = '',
  ...rest
}: { indeterminate?: boolean } & HTMLProps<HTMLInputElement>) {
  const ref = React.useRef<HTMLInputElement>(null!)

  React.useEffect(() => {
    if (typeof indeterminate === 'boolean') {
      ref.current.indeterminate = !rest.checked && indeterminate
    }
  }, [ref, indeterminate])

  return (
    <input
      type="checkbox"
      ref={ref}
      className={className + ' cursor-pointer'}
      {...rest}
    />
  )
```

```
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Sub Components

Github
StackBlitz
CodeSandbox

Code Explorer | Code
Interactive Sandbox | Sandbox

Directory Structure

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Example Code (TypeScript React)

```
import React, { Fragment } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  useReactTable,
  getCoreRowModel,
  getExpandedRowModel,
  ColumnDef,
  flexRender,
  Row,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const columns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: props => props.column.id,
    columns: [
      {
        id: 'expander',
        header: () => null,
        cell: ({ row }) => {
          return row.getCanExpand() ? (
            <button
              {...{
```

```
              onClick: row.getToggleExpandedHandler(),
              style: { cursor: 'pointer' },
            }}
          >
            {row.getIsExpanded() ? '👇' : '👉'}
          </button>
        ) : (
          '🔵'
        )
      },
    },
    {
      accessorKey: 'firstName',
      header: 'First Name',
      cell: ({ row, getValue }) => (
        <div
          style={{
            // Use row depth to visually indicate hierarchy
            paddingLeft: `${row.depth * 2}rem`,
          }}
        >
          {getValue<string>()}
        </div>
      ),
      footer: props => props.column.id,
    },
    {
      accessorFn: row => row.lastName,
      id: 'lastName',
      cell: info => info.getValue(),
      header: () => <span>Last Name</span>,
      footer: props => props.column.id,
    },
  ],
},
{
  header: 'Info',
  footer: props => props.column.id,
  columns: [
    {
      accessorKey: 'age',
      header: () => 'Age',
      footer: props => props.column.id,
    },
    {
      header: 'More Info',
      columns: [
        {
          accessorKey: 'visits',
          header: () => <span>Visits</span>,
          footer: props => props.column.id,
        },
        {
          accessorKey: 'status',
          header: 'Status',
```

```
            footer: props => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: props => props.column.id,
          },
        ],
      },
    ],
  },
]

type TableProps<TData> = {
  data: TData[]
  columns: ColumnDef<TData>[]
  renderSubComponent: (props: { row: Row<TData> }) => React.ReactElement
  getRowCanExpand: (row: Row<TData>) => boolean
}

function Table({
  data,
  columns,
  renderSubComponent,
  getRowCanExpand,
}: TableProps<Person>): JSX.Element {
  const table = useReactTable<Person>({
    data,
    columns,
    getRowCanExpand,
    getCoreRowModel: getCoreRowModel(),
    getExpandedRowModel: getExpandedRowModel(),
  })

  return (
    <div className="p-2">
      <div className="h-2" />
      <table>
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <div>
                      {flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                    </div>
                  )}
                </th>
              ))}
            </tr>
          ))}
```

```
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => (
            <Fragment key={row.id}>
              {/* Normal row */}
              <tr>
                {row.getVisibleCells().map(cell => (
                  <td key={cell.id}>
                    {flexRender(cell.column.columnDef.cell, cell.getContext())}
                  </td>
                ))}
              </tr>
              {/* Expanded row */}
              {row.getIsExpanded() && (
                <tr>
                  {/* Custom subcomponent row */}
                  <td colSpan={row.getVisibleCells().length}>
                    {renderSubComponent({ row })}
                  </td>
                </tr>
              )}
            </Fragment>
          ))}
        </tbody>
      </table>
      <div className="h-2" />
      <div>{table.getRowModel().rows.length} Rows</div>
    </div>
  )
}

const renderSubComponent = ({ row }: { row: Row<Person> }) => {
  return (
    <pre style={{ fontSize: '10px' }}>
      <code>{JSON.stringify(row.original, null, 2)}</code>
    </pre>
  )
}

function App() {
  const [data] = React.useState(() => makeData(10))

  return (
    <Table
      data={data}
      columns={columns}
      getRowCanExpand={() => true}
      renderSubComponent={renderSubComponent}
    />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')
```

```
ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Fully Controlled

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  getPaginationRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData } from './makeData'

type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}

const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: info => info.getValue(),
        footer: props => props.column.id,
```

```
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: props => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: props => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: props => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const [data] = React.useState(() => makeData(1000))
  const [columns] = React.useState<typeof defaultColumns>(() => [
    ...defaultColumns,
  ])

  const rerender = React.useReducer(() => ({}), {})[1]

  // Create the table and pass your options
  const table = useReactTable({
    data,
    columns,
```

```
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
})

// Manage your own state
const [state, setState] = React.useState(table.initialState)

// Override the state managers for the table to your own
table.setOptions(prev => ({
  ...prev,
  state,
  onStateChange: setState,
  // These are just table options, so if things
  // need to change based on your state, you can
  // derive them here

  // Just for fun, let's debug everything if the pageIndex
  // is greater than 2
  debugTable: state.pagination.pageIndex > 2,
}))

return (
  <div className="p-2">
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => (
              <th key={header.id} colSpan={header.colSpan}>
                {header.isPlaceholder
                  ? null
                  : flexRender(
                      header.column.columnDef.header,
                      header.getContext()
                    )}
              </th>
            ))}
          </tr>
        ))}
      </thead>
      <tbody>
        {table.getRowModel().rows.map(row => (
          <tr key={row.id}>
            {row.getVisibleCells().map(cell => (
              <td key={cell.id}>
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </td>
            ))}
          </tr>
        ))}
      </tbody>
      <tfoot>
        {table.getFooterGroups().map(footerGroup => (
          <tr key={footerGroup.id}>
            {footerGroup.headers.map(header => (
```

```
          <th key={header.id} colSpan={header.colSpan}>
            {header.isPlaceholder
              ? null
              : flexRender(
                  header.column.columnDef.footer,
                  header.getContext()
                )}
          </th>
        ))}
      </tr>
    ))}
  </tfoot>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of {table.getPageCount()}
    </strong>
  </span>
  <span className="flex items-center gap-1">
    | Go to page:
    <input
      type="number"
      min="1"
      max={table.getPageCount()}
```

```
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={e => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={e => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map(pageSize => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div className="h-4" />
      <button onClick={() => rerender()} className="border p-2">
        Rerender
      </button>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Grouping

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

File Structure

- `src`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  GroupingState,
  useReactTable,
  getPaginationRowModel,
  getFilteredRowModel,
  getCoreRowModel,
  getGroupedRowModel,
  getExpandedRowModel,
  ColumnDef,
  flexRender,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      header: 'Name',
      columns: [
        {
          accessorKey: 'firstName',
          header: 'First Name',
          cell: info => info.getValue(),
          // override the value used for row grouping
          getGroupingValue: row => `${row.firstName} ${row.lastName}`,
```

```
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        header: () => <span>Last Name</span>,
        cell: info => info.getValue(),
      },
    ],
  },
  {
    header: 'Info',
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        aggregatedCell: ({ getValue }) =>
          Math.round(getValue<number>() * 100) / 100,
        aggregationFn: 'median',
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            aggregationFn: 'sum',
          },
          {
            accessorKey: 'status',
            header: 'Status',
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            cell: ({ getValue }) =>
              Math.round(getValue<number>() * 100) / 100 + '%',
            aggregationFn: 'mean',
            aggregatedCell: ({ getValue }) =>
              Math.round(getValue<number>() * 100) / 100 + '%',
          },
        ],
      },
    ],
  },
], []);

const [data, setData] = React.useState(() => makeData(100000))
const refreshData = () => setData(() => makeData(100000))
const [grouping, setGrouping] = React.useState<GroupingState>([])

const table = useReactTable({
  data,
  columns,
  state: { grouping },
  onGroupingChange: setGrouping,
```

```
  getExpandedRowModel: getExpandedRowModel(),
  getGroupedRowModel: getGroupedRowModel(),
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  debugTable: true,
})

return (
  <div className="p-2">
    <div className="h-2" />
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <div>
                      {header.column.getCanGroup() ? (
                        <button
                          {...{
                            onClick: header.column.getToggleGroupingHandler(),
                            style: { cursor: 'pointer' },
                          }}
                        >
                          {header.column.getIsGrouped()
                            ? `🛑(${header.column.getGroupedIndex()}) `
                            : `👊 `}
                        </button>
                      ) : null}{' '}
                      {flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                    </div>
                  )}
                </th>
              )
            })}
          </tr>
        ))}
      </thead>
      <tbody>
        {table.getRowModel().rows.map(row => {
          return (
            <tr key={row.id}>
              {row.getVisibleCells().map(cell => {
                return (
                  <td
                    {...{
                      key: cell.id,
                      style: {
                        background: cell.getIsGrouped()
```

```
                      ? '#0aff0082'
                      : cell.getIsAggregated()
                        ? '#ffa50078'
                        : cell.getIsPlaceholder()
                          ? '#ff000042'
                          : 'white',
                    },
                  }}
                >
                  {cell.getIsGrouped() ? (
                    <React.Fragment>
                      <button
                        {...{
                          onClick: row.getToggleExpandedHandler(),
                          style: {
                            cursor: row.getCanExpand()
                              ? 'pointer'
                              : 'normal',
                          },
                        }}
                      >
                        {row.getIsExpanded() ? '👇' : '👉'}{' '}
                        {flexRender(
                          cell.column.columnDef.cell,
                          cell.getContext()
                        )}{' '}
                        ({row.subRows.length})
                      </button>
                    </React.Fragment>
                  ) : cell.getIsAggregated() ? (
                    flexRender(
                      cell.column.columnDef.aggregatedCell ??
                        cell.column.columnDef.cell,
                      cell.getContext()
                    )
                  ) : cell.getIsPlaceholder() ? null : (
                    flexRender(
                      cell.column.columnDef.cell,
                      cell.getContext()
                    )
                  )}
                </td>
              )
            })}
          </tr>
        )
      })}
    </tbody>
  </table>
  <div className="h-2" />
  <div className="flex items-center gap-2">
    <button
      className="border rounded p-1"
      onClick={() => table.setPageIndex(0)}
      disabled={!table.getCanPreviousPage()}
```

```
>
  {'<<'}
</button>
<button
  className="border rounded p-1"
  onClick={() => table.previousPage()}
  disabled={!table.getCanPreviousPage()}
>
  {'<'}
</button>
<button
  className="border rounded p-1"
  onClick={() => table.nextPage()}
  disabled={!table.getCanNextPage()}
>
  {'>'}
</button>
<button
  className="border rounded p-1"
  onClick={() => table.setPageIndex(table.getPageCount() - 1)}
  disabled={!table.getCanNextPage()}
>
  {'>>'}
</button>
<span className="flex items-center gap-1">
  <div>Page</div>
  <strong>
    {table.getState().pagination.pageIndex + 1} of {table.getPageCount()}
  </strong>
</span>
<span className="flex items-center gap-1">
  | Go to page:
  <input
    type="number"
    min="1"
    max={table.getPageCount()}
    defaultValue={table.getState().pagination.pageIndex + 1}
    onChange={e => {
      const page = e.target.value ? Number(e.target.value) - 1 : 0
      table.setPageIndex(page)
    }}
    className="border p-1 rounded w-16"
  />
</span>
<select
  value={table.getState().pagination.pageSize}
  onChange={e => {
    table.setPageSize(Number(e.target.value))
  }}
>
  {[10,20,30,40,50].map(pageSize => (
    <option key={pageSize} value={pageSize}>
      Show {pageSize}
    </option>
  ))}
```

```
        </select>
      </div>
      <div>{table.getRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>{JSON.stringify(grouping, null, 2)}</pre>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Table Pagination Example | TanStack Table Docs

Our Partners

TanStack Query

Powerful asynchronous state management, server-state utilities, and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state".

Learn More

TanStack DB

TanStack DB extends TanStack Query with collections, live queries, and optimistic mutations that keep your UI reactive, consistent, and blazing fast 🔥

Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at any time.

React Example: Pagination Controlled

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- **src**
  - fetchData.ts
  - index.css
  - main.tsx
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Code snippets:

```
// Your TypeScript code here
```

Related Links:

- Pagination
- Row DnD

Our Partners

Subscribe to Bytes

React Example: Row Dnd

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

Files:

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Example code (TypeScript)

```
import React, { CSSProperties } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  ColumnDef,
  Row,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

// needed for table body level scope DnD setup
import {
  DndContext,
  KeyboardSensor,
  MouseSensor,
  TouchSensor,
  closestCenter,
  type DragEndEvent,
  type UniqueIdentifier,
  useSensor,
  useSensors,
} from '@dnd-kit/core'
import { restrictToVerticalAxis } from '@dnd-kit/modifiers'
import {
  arrayMove,
```

```
  SortableContext,
  verticalListSortingStrategy,
} from '@dnd-kit/sortable'

// needed for row & cell level scope DnD setup
import { useSortable } from '@dnd-kit/sortable'
import { CSS } from '@dnd-kit/utilities'

// Cell Component
const RowDragHandleCell = ({ rowId }: { rowId: string }) => {
  const { attributes, listeners } = useSortable({
    id: rowId,
  })
  return (
    // Alternatively, you could set these attributes on the rows themselves
    <button {...attributes} {...listeners}>
      🟰
    </button>
  )
}

// Row Component
const DraggableRow = ({ row }: { row: Row<Person> }) => {
  const { transform, transition, setNodeRef, isDragging } = useSortable({
    id: row.original.userId,
  })

  const style: CSSProperties = {
    transform: CSS.Transform.toString(transform), //let dnd-kit do its thing
    transition: transition,
    opacity: isDragging ? 0.8 : 1,
    zIndex: isDragging ? 1 : 0,
    position: 'relative',
  }
  return (
    // connect row ref to dnd-kit, apply important styles
    <tr ref={setNodeRef} style={style}>
      {row.getVisibleCells().map((cell) => (
        <td key={cell.id} style={{ width: cell.column.getSize() }}>
          {flexRender(cell.column.columnDef.cell, cell.getContext())}
        </td>
      ))}
    </tr>
  )
}

// Table Component
function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    // Create a dedicated drag handle column. Alternatively, you could just set up dnd
    {
      id: 'drag-handle',
      header: 'Move',
      cell: ({ row }) => <RowDragHandleCell rowId={row.id} />,
      size: 60,
```

```
  },
  {
    accessorKey: 'firstName',
    cell: (info) => info.getValue(),
  },
  {
    accessorFn: (row) => row.lastName,
    id: 'lastName',
    cell: (info) => info.getValue(),
    header: () => <span>Last Name</span>,
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
  },
  {
    accessorKey: 'status',
    header: 'Status',
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
  },
], [])

const [data, setData] = React.useState(() => makeData(20))

const dataIds = React.useMemo<UniqueIdentifier[]>(() => data?.map(({ userId }) => us

const rerender = () => setData(() => makeData(20))

const table = useReactTable({
  data,
  columns,
  getCoreRowModel: getCoreRowModel(),
  getRowId: (row) => row.userId, // required because row indexes will change
  debugTable: true,
  debugHeaders: true,
  debugColumns: true,
})

// reorder rows after drag & drop
function handleDragEnd(event: DragEndEvent) {
  const { active, over } = event
  if (active && over && active.id !== over.id) {
    setData((data) => {
      const oldIndex = dataIds.indexOf(active.id)
      const newIndex = dataIds.indexOf(over.id)
      return arrayMove(data, oldIndex, newIndex) //this is just a splice util
    })
  }
```

```
  }

  const sensors = useSensors(
    useSensor(MouseSensor, {}),
    useSensor(TouchSensor, {}),
    useSensor(KeyboardSensor, {})
  )

  return (
    // NOTE: This provider creates div elements, so don't nest inside of <table> eleme
    <DndContext
      collisionDetection={closestCenter}
      modifiers={[restrictToVerticalAxis]}
      onDragEnd={handleDragEnd}
      sensors={sensors}
    >
      <div className="p-2">
        <div className="h-4" />
        <div className="flex flex-wrap gap-2">
          <button onClick={() => rerender()} className="border p-1">
            Regenerate
          </button>
        </div>
        <div className="h-4" />
        <table>
          <thead>
            {table.getHeaderGroups().map((headerGroup) => (
              <tr key={headerGroup.id}>
                {headerGroup.headers.map((header) => (
                  <th key={header.id} colSpan={header.colSpan}>
                    {header.isPlaceholder
                      ? null
                      : flexRender(header.column.columnDef.header, header.getContext()
                    </th>
                ))}
              </tr>
            ))}
          </thead>
          <tbody>
            <SortableContext items={dataIds} strategy={verticalListSortingStrategy}>
              {table.getRowModel().rows.map((row) => (
                <DraggableRow key={row.id} row={row} />
              ))}
            </SortableContext>
          </tbody>
        </table>
        <pre>{JSON.stringify(data, null, 2)}</pre>
      </div>
    </DndContext>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')
```

```
ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Filters | TanStack Table Docs

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

- `src`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Column,
  ColumnDef,
  ColumnFiltersState,
  RowData,
  flexRender,
  getCoreRowModel,
  getFilteredRowModel,
  getPaginationRowModel,
  getSortedRowModel,
  useReactTable,
} from '@tanstack/react-table'

import { makeData, Person } from './makeData'

declare module '@tanstack/react-table' {
  //allows us to define custom properties for our columns
  interface ColumnMeta<TData extends RowData, TValue> {
    filterVariant?: 'text' | 'range' | 'select'
  }
}

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const [columnFilters, setColumnFilters] = React.useState<ColumnFiltersState>(
    []
```

```
)

const columns = React.useMemo<ColumnDef<Person, any>[]>(() => [
  {
    accessorKey: 'firstName',
    cell: info => info.getValue(),
  },
  {
    accessorFn: row => row.lastName,
    id: 'lastName',
    cell: info => info.getValue(),
    header: () => <span>Last Name</span>,
  },
  {
    accessorFn: row => `${row.firstName} ${row.lastName}`,
    id: 'fullName',
    header: 'Full Name',
    cell: info => info.getValue(),
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
    meta: {
      filterVariant: 'range',
    },
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    meta: {
      filterVariant: 'range',
    },
  },
  {
    accessorKey: 'status',
    header: 'Status',
    meta: {
      filterVariant: 'select',
    },
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    meta: {
      filterVariant: 'range',
    },
  },
], [])

const [data, setData] = React.useState<Person[]>(() => makeData(5000))
const refreshData = () => setData(_old => makeData(50000)) // stress test

const table = useReactTable({
  data,
  columns,
```

```
  filterFns: {},
  state: {
    columnFilters,
  },
  onColumnFiltersChange: setColumnFilters,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(), // client side filtering
  getSortedRowModel: getSortedRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  debugTable: true,
  debugHeaders: true,
  debugColumns: false,
})

return (
  <div className="p-2">
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <>
                      <div
                        {...{
                          className: header.column.getCanSort()
                            ? 'cursor-pointer select-none'
                            : '',
                          onClick: header.column.getToggleSortingHandler(),
                        }}
                      >
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                        {{
                          asc: ' 🔼',
                          desc: ' 🔽',
                        }[header.column.getIsSorted() as string] ?? null}
                      </div>
                      {header.column.getCanFilter() ? (
                        <div>
                          <Filter column={header.column} />
                        </div>
                      ) : null}
                    </>
                  )}
                </th>
              )
            })}
          </tr>
        ))}
      </thead>
```

```
  <tbody>
    {table.getRowModel().rows.map(row => {
      return (
        <tr key={row.id}>
          {row.getVisibleCells().map(cell => {
            return (
              <td key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
      {table.getPageCount()}
    </strong>
```

```
        </span>
        <span className="flex items-center gap-1">
          | Go to page:
          <input
            type="number"
            min="1"
            max={table.getPageCount()}
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={e => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={e => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map(pageSize => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>{table.getPrePaginationRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>
        {JSON.stringify(
          { columnFilters: table.getState().columnFilters },
          null,
          2
        )}
      </pre>
    </div>
  )
}

function Filter({ column }: { column: Column<any, unknown> }) {
  const columnFilterValue = column.getFilterValue()
  const { filterVariant } = column.columnDef.meta ?? {}

  return filterVariant === 'range' ? (
    <div>
      <div className="flex space-x-2">
        {/* See faceted column filters example for min max values functionality */}
        <DebouncedInput
```

```
          type="number"
          value={(columnFilterValue as [number, number])?.[0] ?? ''}
          onChange={value =>
            column.setFilterValue((old: [number, number]) => [value, old?.[1]])
          }
          placeholder={`Min`}
          className="w-24 border shadow rounded"
        />
        <DebouncedInput
          type="number"
          value={(columnFilterValue as [number, number])?.[1] ?? ''}
          onChange={value =>
            column.setFilterValue((old: [number, number]) => [old?.[0], value])
          }
          placeholder={`Max`}
          className="w-24 border shadow rounded"
        />
      </div>
      <div className="h-1" />
    </div>
  ) : filterVariant === 'select' ? (
    <select
      onChange={e => column.setFilterValue(e.target.value)}
      value={columnFilterValue?.toString()}
    >
      {/* See faceted column filters example for dynamic select options */}
      <option value="">All</option>
      <option value="complicated">complicated</option>
      <option value="relationship">relationship</option>
      <option value="single">single</option>
    </select>
  ) : (
    <DebouncedInput
      className="w-36 border shadow rounded"
      onChange={value => column.setFilterValue(value)}
      placeholder={`Search...`}
      type="text"
      value={(columnFilterValue ?? '') as string}
    />
    // See faceted column filters example for datalist search suggestions
  )
}

// A typical debounced input react component
function DebouncedInput({
  value: initialValue,
  onChange,
  debounce = 500,
  ...props
}: {
  value: string | number
  onChange: (value: string | number) => void
  debounce?: number
} & Omit<React.InputHTMLAttributes<HTMLInputElement>, 'onChange'>) {
  const [value, setValue] = React.useState(initialValue)
```

```
  React.useEffect(() => {
    setValue(initialValue)
  }, [initialValue])

  React.useEffect(() => {
    const timeout = setTimeout(() => {
      onChange(value)
    }, debounce)

    return () => clearTimeout(timeout)
  }, [value])

  return (
    <input {...props} value={value} onChange={e => setValue(e.target.value)} />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Row Pinning

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import { makeData, Person } from './makeData'

import {
  Column,
  ColumnDef,
  ExpandedState,
  flexRender,
  getCoreRowModel,
  getExpandedRowModel,
  getFilteredRowModel,
  getPaginationRowModel,
  Row,
  RowPinningState,
  Table,
  useReactTable,
} from '@tanstack/react-table'

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  // table states
  const [rowPinning, setRowPinning] = React.useState<RowPinningState>({
    top: [],
    bottom: [],
```

```
})
const [expanded, setExpanded] = React.useState<ExpandedState>({})

// demo states
const [keepPinnedRows, setKeepPinnedRows] = React.useState(true)
const [includeLeafRows, setIncludeLeafRows] = React.useState(true)
const [includeParentRows, setIncludeParentRows] = React.useState(false)
const [copyPinnedRows, setCopyPinnedRows] = React.useState(false)

const columns = React.useMemo<ColumnDef<Person>[]>(() => [
  {
    id: 'pin',
    header: () => 'Pin',
    cell: ({ row }) =>
      row.getIsPinned() ? (
        <button
          onClick={() => row.pin(false, includeLeafRows, includeParentRows)}
        >
          ❌
        </button>
      ) : (
        <div style={{ display: 'flex', gap: '4px' }}>
          <button
            onClick={() =>
              row.pin('top', includeLeafRows, includeParentRows)
            }
          >
            ⬆️
          </button>
          <button
            onClick={() =>
              row.pin('bottom', includeLeafRows, includeParentRows)
            }
          >
            ⬇️
          </button>
        </div>
      ),
  },
  {
    accessorKey: 'firstName',
    header: ({ table }) => (
      <>
        <button
          {...{
            onClick: table.getToggleAllRowsExpandedHandler(),
          }}
        >
          {table.getIsAllRowsExpanded() ? '👇' : '👉'}
        </button>{' '}
        First Name
      </>
    ),
    cell: ({ row, getValue }) => (
      <div
```

```
        style={{
          paddingLeft: `${row.depth * 2}rem`,
        }}
      >
        <>
          {row.getCanExpand() ? (
            <button
              {...{
                onClick: row.getToggleExpandedHandler(),
                style: { cursor: 'pointer' },
              }}
            >
              {row.getIsExpanded() ? '👇' : '👉'}
            </button>
          ) : (
            '🔵'
          )}{' '}
          {getValue()}
        </>
      </div>
    ),
    footer: (props) => props.column.id,
  },
  {
    accessorFn: (row) => row.lastName,
    id: 'lastName',
    cell: (info) => info.getValue(),
    header: () => <span>Last Name</span>,
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
    size: 50,
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    size: 50,
  },
  {
    accessorKey: 'status',
    header: 'Status',
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    size: 80,
  },
], [includeLeafRows, includeParentRows]);

const [data, setData] = React.useState(() => makeData(1000, 2, 2));
const refreshData = () => setData(() => makeData(1000, 2, 2));

const table = useReactTable({
  data,
```

```
  columns,
  initialState: { pagination: { pageSize: 20, pageIndex: 0 } },
  state: {
    expanded,
    rowPinning,
  },
  onExpandedChange: setExpanded,
  onRowPinningChange: setRowPinning,
  getSubRows: (row) => row.subRows,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getExpandedRowModel: getExpandedRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  keepPinnedRows,
  debugRows: true,
});

return (
  <div className="app">
    <div className="p-2 container">
      <div className="h-2" />
      <table>
        <thead>
          {table.getHeaderGroups().map((headerGroup) => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map((header) => {
                return (
                  <th key={header.id} colSpan={header.colSpan}>
                    {header.isPlaceholder ? null : (
                      <>
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                        {header.column.getCanFilter() ? (
                          <div>
                            <Filter column={header.column} table={table} />
                          </div>
                        ) : null}
                      </>
                    )}
                  </th>
                );
              })}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getTopRows().map((row) => (
            <PinnedRow key={row.id} row={row} table={table} />
          ))}
          {(copyPinnedRows
            ? table.getRowModel().rows
            : table.getCenterRows()
          ).map((row) => {
```

```
        return (
          <tr key={row.id}>
            {row.getVisibleCells().map((cell) => {
              return (
                <td key={cell.id}>
                  {flexRender(
                    cell.column.columnDef.cell,
                    cell.getContext()
                  )}
                </td>
              );
            })}
          </tr>
        );
      })}
      {table.getBottomRows().map((row) => (
        <PinnedRow key={row.id} row={row} table={table} />
      ))}
    </tbody>
  </table>
</div>

<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
```

```
        {table.getState().pagination.pageIndex + 1} of {table.getPageCount()}
      </strong>
    </span>
    <span className="flex items-center gap-1">
      | Go to page:
      <input
        type="number"
        min="1"
        max={table.getPageCount()}
        defaultValue={table.getState().pagination.pageIndex + 1}
        onChange={(e) => {
          const page = e.target.value ? Number(e.target.value) - 1 : 0;
          table.setPageIndex(page);
        }}
        className="border p-1 rounded w-16"
      />
    </span>
    <select
      value={table.getState().pagination.pageSize}
      onChange={(e) => {
        table.setPageSize(Number(e.target.value));
      }}
    >
      {[10, 20, 30, 40, 50].map((pageSize) => (
        <option key={pageSize} value={pageSize}>
          Show {pageSize}
        </option>
      ))}
    </select>
  </div>
  <div className="h-2" />
  <hr />
  <br />
  <div className="flex flex-col gap-2 align-center vertical">
    <div>
      <input
        type="checkbox"
        checked={keepPinnedRows}
        onChange={() => setKeepPinnedRows(!keepPinnedRows)}
      />
      <label className="ml-2">
        Keep/Persist Pinned Rows across Pagination and Filtering
      </label>
    </div>
    <div>
      <input
        type="checkbox"
        checked={includeLeafRows}
        onChange={() => setIncludeLeafRows(!includeLeafRows)}
      />
      <label className="ml-2">Include Leaf Rows When Pinning Parent</label>
    </div>
    <div>
      <input
        type="checkbox"
```

```
            checked={includeParentRows}
            onChange={() => setIncludeParentRows(!includeParentRows)}
          />
          <label className="ml-2">Include Parent Rows When Pinning Child</label>
        </div>
        <div>
          <input
            type="checkbox"
            checked={copyPinnedRows}
            onChange={() => setCopyPinnedRows(!copyPinnedRows)}
          />
          <label className="ml-2">Duplicate/Keep Pinned Rows in main table</label>
        </div>
      </div>
      <div>
        <button className="border rounded p-2 mb-2" onClick={() => rerender()}>
          Force Rerender
        </button>
      </div>
      <div>
        <button
          className="border rounded p-2 mb-2"
          onClick={() => refreshData()}
        >
          Refresh Data
        </button>
      </div>
      <div>{JSON.stringify(rowPinning, null, 2)}</div>
    </div>
  );
}

function PinnedRow({ row, table }: { row: Row<any>; table: Table<any> }) {
  return (
    <tr
      style={{
        backgroundColor: 'lightblue',
        position: 'sticky',
        top:
          row.getIsPinned() === 'top'
            ? `${row.getPinnedIndex() * 26 + 48}px`
            : undefined,
        bottom:
          row.getIsPinned() === 'bottom'
            ? `${(table.getBottomRows().length - 1 - row.getPinnedIndex()) * 26}px`
            : undefined,
      }}
    >
      {row.getVisibleCells().map((cell) => {
        return (
          <td key={cell.id}>
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        );
      })}
```

```
    </tr>
  );
}

function Filter({
  column,
  table,
}: {
  column: Column<any, any>;
  table: Table<any>;
}) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id);

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
        type="number"
        value={((column.getFilterValue() as any)?.[0] ?? '') as string}
        onChange={(e) =>
          column.setFilterValue((old: any) => [e.target.value, old?.[1]])
        }
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={((column.getFilterValue() as any)?.[1] ?? '') as string}
        onChange={(e) =>
          column.setFilterValue((old: any) => [old?.[0], e.target.value])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(column.getFilterValue() ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  );
}

const rootElement = document.getElementById('root');
if (!rootElement) throw new Error('Failed to find the root element');

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

React Table Row Selection Example | TanStack Table Docs

TanStack / Table v8

Framework: *React*
Version: **Latest**

Search...

Menu:

- Home
- Frameworks
- Contributors
- GitHub
- Discord

Getting Started

- Introduction (core)
- Overview (core)
- Installation (core)
- Migrating to V8 (core)
- FAQ (core)
- React Table Adapter (react)

Core Guides

- Data (core)
- Column Defs (core)
- Table Instance (core)
- Row Models (core)
- Rows (core)
- Cells (core)
- Header Groups (core)
- Headers (core)
- Columns (core)
- Table State (react)

Feature Guides

- Column Ordering (core)
- Column Pinning (core)
- Column Sizing (core)
- Column Visibility (core)
- Column Filtering (core)
- Global Filtering (core)
- Fuzzy Filtering (core)
- Column Faceting (core)
- Global Faceting (core)
- Grouping (core)

- Expanding (core)
- Pagination (core)
- Row Pinning (core)
- Row Selection (core)
- Sorting (core)
- Virtualization (core)
- Custom Features (core)

Core APIs

- Column Def
- Table
- Column
- Header Group
- Header
- Row
- Cell

Feature APIs

- Column Filtering
- Column Faceting
- Column Ordering
- Column Pinning
- Column Sizing
- Column Visibility
- Global Faceting
- Global Filtering
- Sorting
- Grouping
- Expanding
- Pagination
- Row Pinning
- Row Selection

Enterprise

- AG Grid

Examples

- Basic (react)
- Header Groups (react)
- Column Filters (react)
- Column Filters (Faceted) (react)
- Fuzzy Search Filters (react)
- Column Ordering (react)
- Column Ordering (DnD) (react)
- Column Pinning (react)
- Sticky Column Pinning (react)
- Column Sizing (react)
- Performant Column Resizing (react)
- Column Visibility (react)
- Editable Data (react)

- Expanding (react)
- Sub Components (react)
- Fully Controlled (react)
- Grouping (react)
- Pagination (react)
- Pagination Controlled (react)
- Row DnD (react)
- Row Pinning (react)
- Row Selection (react)
- Sorting (react)
- Virtualized Columns (react)
- Virtualized Columns (Experimental) (react)
- Virtualized Rows (react)
- Virtualized Rows (Experimental) (react)
- Virtualized Infinite Scrolling (react)
- Kitchen Sink (react)
- React Bootstrap (react)
- Material UI Pagination (react)
- React Full Width (react)
- React Full Width Resizable (react)
- Custom Features (react)
- Query Router Search Params (react)

React Example: Row Selection

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox

Files:

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

```
import React, { HTMLAttributes, HTMLProps } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import { makeData, Person } from './makeData'

import {
  Column,
  ColumnDef,
  flexRender,
  getCoreRowModel,
```

```
  getFilteredRowModel,
  getPaginationRowModel,
  Table,
  useReactTable,
} from '@tanstack/react-table'

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const [rowSelection, setRowSelection] = React.useState({})
  const [globalFilter, setGlobalFilter] = React.useState('')

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      id: 'select',
      header: ({ table }) => (
        <IndeterminateCheckbox
          {...{
            checked: table.getIsAllRowsSelected(),
            indeterminate: table.getIsSomeRowsSelected(),
            onChange: table.getToggleAllRowsSelectedHandler(),
          }}
        />
      ),
      cell: ({ row }) => (
        <div className="px-1">
          <IndeterminateCheckbox
            {...{
              checked: row.getIsSelected(),
              disabled: !row.getCanSelect(),
              indeterminate: row.getIsSomeSelected(),
              onChange: row.getToggleSelectedHandler(),
            }}
          />
        </div>
      ),
    },
    {
      header: 'Name',
      footer: (props) => props.column.id,
      columns: [
        {
          accessorKey: 'firstName',
          cell: (info) => info.getValue(),
          footer: (props) => props.column.id,
        },
        {
          accessorFn: (row) => row.lastName,
          id: 'lastName',
          cell: (info) => info.getValue(),
          header: () => <span>Last Name</span>,
          footer: (props) => props.column.id,
        },
      ],
    },
```

```
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
], [])

const [data, setData] = React.useState(() => makeData(100000))
const refreshData = () => setData(() => makeData(100000))

const table = useReactTable({
  data,
  columns,
  state: {
    rowSelection,
  },
  enableRowSelection: true, // Enable row selection for all rows
  onRowSelectionChange: setRowSelection,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  debugTable: true,
})

return (
  <div className="p-2">
    <div>
      <input
        value={globalFilter ?? ''}
```

```
    onChange={(e) => setGlobalFilter(e.target.value)}
    className="p-2 font-lg shadow border border-block"
    placeholder="Search all columns..."
  />
</div>
<div className="h-2" />
<table>
  <thead>
    {table.getHeaderGroups().map((headerGroup) => (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map((header) => {
          return (
            <th key={header.id} colSpan={header.colSpan}>
              {header.isPlaceholder ? null : (
                <>
                  {flexRender(
                    header.column.columnDef.header,
                    header.getContext()
                  )}
                  {header.column.getCanFilter() ? (
                    <div>
                      <Filter column={header.column} table={table} />
                    </div>
                  ) : null}
                </>
              )}
            </th>
          )
        })}
      </tr>
    ))}
  </thead>
  <tbody>
    {table.getRowModel().rows.map((row) => {
      return (
        <tr key={row.id}>
          {row.getVisibleCells().map((cell) => {
            return (
              <td key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
  <tfoot>
    <tr>
      <td className="p-1">
        <IndeterminateCheckbox
          {...{
```

```
            checked: table.getIsAllPageRowsSelected(),
            indeterminate: table.getIsSomePageRowsSelected(),
            onChange: table.getToggleAllPageRowsSelectedHandler(),
          }}
        />
      </td>
      <td colSpan={20}>Page Rows ({table.getRowModel().rows.length})</td>
    </tr>
  </tfoot>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of {table.getPageCount()}
    </strong>
  </span>
  <span className="flex items-center gap-1">
    | Go to page:
    <input
      type="number"
      min="1"
      max={table.getPageCount()}
      defaultValue={table.getState().pagination.pageIndex + 1}
      onChange={(e) => {
        const page = e.target.value ? Number(e.target.value) - 1 : 0
```

```
            table.setPageIndex(page)
          }}
          className="border p-1 rounded w-16"
        />
      </span>
      <select
        value={table.getState().pagination.pageSize}
        onChange={(e) => {
          table.setPageSize(Number(e.target.value))
        }}
      >
        {[10, 20, 30, 40, 50].map((pageSize) => (
          <option key={pageSize} value={pageSize}>
            Show {pageSize}
          </option>
        ))}
      </select>
    </div>
    <br />
    <div>
      {Object.keys(rowSelection).length} of {table.getPreFilteredRowModel().rows.len
    </div>
    <hr />
    <br />
    <div>
      <button className="border rounded p-2 mb-2" onClick={() => rerender()}>
        Force Rerender
      </button>
    </div>
    <div>
      <button
        className="border rounded p-2 mb-2"
        onClick={() => refreshData()}
      >
        Refresh Data
      </button>
    </div>
    <div>
      <button
        className="border rounded p-2 mb-2"
        onClick={() =>
          console.info(
            'table.getSelectedRowModel().flatRows',
            table.getSelectedRowModel().flatRows
          )
        }
      >
        Log table.getSelectedRowModel().flatRows
      </button>
    </div>
    <div>
      <label>Row Selection State:</label>
      <pre>{JSON.stringify(table.getState().rowSelection, null, 2)}</pre>
    </div>
  </div>
```

```
  )
}

function Filter({ column, table }: { column: Column<any, any>; table: Table<any> }) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id)

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
        type="number"
        value={((column.getFilterValue() as any)?.[0] ?? '') as string}
        onChange={(e) =>
          column.setFilterValue((old: any) => [e.target.value, old?.[1]])
        }
        placeholder="Min"
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={((column.getFilterValue() as any)?.[1] ?? '') as string}
        onChange={(e) =>
          column.setFilterValue((old: any) => [old?.[0], e.target.value])
        }
        placeholder="Max"
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(column.getFilterValue() ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder="Search..."
      className="w-36 border shadow rounded"
    />
  )
}

function IndeterminateCheckbox({
  indeterminate,
  className = '',
  ...rest
}: { indeterminate?: boolean } & HTMLProps<HTMLInputElement>) {
  const ref = React.useRef<HTMLInputElement>(null!)

  React.useEffect(() => {
    if (typeof indeterminate === 'boolean') {
      ref.current.indeterminate = !rest.checked && indeterminate
    }
  }, [ref, indeterminate])

  return (
    <input
```

```
      type="checkbox"
      ref={ref}
      className={className + ' cursor-pointer'}
      {...rest}
    />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Sorting

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  SortingFn,
  SortingState,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

// custom sorting logic for one of our enum columns
const sortStatusFn: SortingFn<Person> = (rowA, rowB, _columnId) => {
  const statusA = rowA.original.status
  const statusB = rowB.original.status
  const statusOrder = ['single', 'complicated', 'relationship']
  return statusOrder.indexOf(statusA) - statusOrder.indexOf(statusB)
}

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const [sorting, setSorting] = React.useState<SortingState>([])

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'firstName',
      cell: (info) => info.getValue(),
      // this column will sort in ascending order by default since it is a string colu
    },
    {
      accessorFn: (row) => row.lastName,
      id: 'lastName',
      cell: (info) => info.getValue(),
      header: () => <span>Last Name</span>,
      sortUndefined: 'last', // force undefined values to the end
      sortDescFirst: false, // first sort order will be ascending
    },
    {
```

```
    accessorKey: 'age',
    header: () => 'Age',
    // this column will sort in descending order by default
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    sortUndefined: 'last',
  },
  {
    accessorKey: 'status',
    header: 'Status',
    sortingFn: sortStatusFn, // use custom sorting for enum
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    // enableSorting: false,
  },
  {
    accessorKey: 'rank',
    header: 'Rank',
    invertSorting: true, // invert sort order
  },
  {
    accessorKey: 'createdAt',
    header: 'Created At',
    // sortingFn: 'datetime'
  },
], [])

const [data, setData] = React.useState(() => makeData(1000))
const refreshData = () => setData(() => makeData(100000)) // stress test with 100k r

const table = useReactTable({
  columns,
  data,
  debugTable: true,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  onSortingChange: setSorting,
  // sortingFns: {
  //   sortStatusFn,
  // },
  state: {
    sorting,
  },
})

// access sorting state
console.log(table.getState().sorting)

return (
  <div className="p-2">
    <div className="h-2" />
```

```
<table>
  <thead>
    {table.getHeaderGroups().map(headerGroup => (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map(header => {
          return (
            <th key={header.id} colSpan={header.colSpan}>
              {header.isPlaceholder ? null : (
                <div
                  className={
                    header.column.getCanSort()
                      ? 'cursor-pointer select-none'
                      : ''
                  }
                  onClick={header.column.getToggleSortingHandler()}
                  title={
                    header.column.getCanSort()
                      ? header.column.getNextSortingOrder() === 'asc'
                        ? 'Sort ascending'
                        : header.column.getNextSortingOrder() === 'desc'
                          ? 'Sort descending'
                          : 'Clear sort'
                      : undefined
                  }
                >
                  {flexRender(
                    header.column.columnDef.header,
                    header.getContext()
                  )}
                  {{
                    asc: ' 🔼',
                    desc: ' 🔽',
                  }[header.column.getIsSorted() as string] ?? null}
                </div>
              )}
            </th>
          )
        })}
      </tr>
    ))}
  </thead>
  <tbody>
    {table
      .getRowModel()
      .rows.slice(0, 10)
      .map(row => {
        return (
          <tr key={row.id}>
            {row.getVisibleCells().map(cell => {
              return (
                <td key={cell.id}>
                  {flexRender(
                    cell.column.columnDef.cell,
                    cell.getContext()
                  )}
```

```
                    </td>
                  )
                })}
              </tr>
            )
          })}
        </tbody>
      </table>
      <div>{table.getRowModel().rows.length.toLocaleString()} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>{JSON.stringify(sorting, null, 2)}</pre>
    </div>
  )
}

const rootElement = document.getElementById('root')

if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Virtualized Columns

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

File Structure:

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import {
  Cell,
  ColumnDef,
  Header,
  HeaderGroup,
  Row,
  Table,
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  useReactTable,
} from '@tanstack/react-table'
import {
  useVirtualizer,
  VirtualItem,
  Virtualizer,
} from '@tanstack/react-virtual'
import { makeColumns, makeData, Person } from './makeData'

function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => makeColumns(1000), [])

  const [data, setData] = React.useState(() => makeData(1000, columns))
```

```
  const refreshData = React.useCallback(() => {
    setData(makeData(1000, columns))
  }, [columns])

  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(),
    getSortedRowModel: getSortedRowModel(),
    debugTable: true,
  })

  // All important CSS styles are included as inline styles for this example. This is
  return (
    <div className="app">
      {process.env.NODE_ENV === 'development' ? (
        <p>
          <strong>Notice:</strong> You are currently running React in development mode
        </p>
      ) : null}
      <div>({columns.length.toLocaleString()} columns)</div>
      <div>({data.length.toLocaleString()} rows)</div>
      <button onClick={refreshData}>Refresh Data</button>
      <TableContainer table={table} />
    </div>
  )
}

interface TableContainerProps {
  table: Table<Person>
}

function TableContainer({ table }: TableContainerProps) {
  const visibleColumns = table.getVisibleLeafColumns()

  // The virtualizers need to know the scrollable container element
  const tableContainerRef = React.useRef<HTMLDivElement>(null)

  // We are using a slightly different virtualization strategy for columns (compared t
  const columnVirtualizer = useVirtualizer<
    HTMLDivElement,
    HTMLTableCellElement
  >({
    count: visibleColumns.length,
    estimateSize: (index) => visibleColumns[index].getSize(), // estimate width of eac
    getScrollElement: () => tableContainerRef.current,
    horizontal: true,
    overscan: 3, // how many columns to render on each side off screen each way (adjus
  })

  const virtualColumns = columnVirtualizer.getVirtualItems()

  // Different virtualization strategy for columns - instead of absolute and translate
  let virtualPaddingLeft: number | undefined
```

```
  let virtualPaddingRight: number | undefined

  if (columnVirtualizer && virtualColumns?.length) {
    virtualPaddingLeft = virtualColumns[0]?.start ?? 0
    virtualPaddingRight =
      columnVirtualizer.getTotalSize() -
      (virtualColumns[virtualColumns.length - 1]?.end ?? 0)
  }

  return (
    <div
      className="container"
      ref={tableContainerRef}
      style={{
        overflow: 'auto', // our scrollable table container
        position: 'relative', // needed for sticky header
        height: '800px', // should be a fixed height
      }}
    >
      {/* Even though we're still using semantic table tags, we must use CSS grid and
      <table style={{ display: 'grid' }}>
        <TableHead
          columnVirtualizer={columnVirtualizer}
          table={table}
          virtualPaddingLeft={virtualPaddingLeft}
          virtualPaddingRight={virtualPaddingRight}
        />
        <TableBody
          columnVirtualizer={columnVirtualizer}
          table={table}
          tableContainerRef={tableContainerRef}
          virtualPaddingLeft={virtualPaddingLeft}
          virtualPaddingRight={virtualPaddingRight}
        />
      </table>
    </div>
  )
}

interface TableHeadProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  table: Table<Person>
  virtualPaddingLeft: number | undefined
  virtualPaddingRight: number | undefined
}

function TableHead({
  columnVirtualizer,
  table,
  virtualPaddingLeft,
  virtualPaddingRight,
}: TableHeadProps) {
  return (
    <thead
      style={{
```

```
        display: 'grid',
        position: 'sticky',
        top: 0,
        zIndex: 1,
      }}
    >
      {table.getHeaderGroups().map((headerGroup) => (
        <TableHeadRow
          columnVirtualizer={columnVirtualizer}
          headerGroup={headerGroup}
          key={headerGroup.id}
          virtualPaddingLeft={virtualPaddingLeft}
          virtualPaddingRight={virtualPaddingRight}
        />
      ))}
    </thead>
  )
}

interface TableHeadRowProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  headerGroup: HeaderGroup<Person>
  virtualPaddingLeft: number | undefined
  virtualPaddingRight: number | undefined
}

function TableHeadRow({
  columnVirtualizer,
  headerGroup,
  virtualPaddingLeft,
  virtualPaddingRight,
}: TableHeadRowProps) {
  const virtualColumns = columnVirtualizer.getVirtualItems()
  return (
    <tr key={headerGroup.id} style={{ display: 'flex', width: '100%' }}>
      {virtualPaddingLeft ? (
        // fake empty column to the left for virtualization scroll padding
        <th style={{ display: 'flex', width: virtualPaddingLeft }} />
      ) : null}
      {virtualColumns.map((virtualColumn) => {
        const header = headerGroup.headers[virtualColumn.index]
        return <TableHeadCell key={header.id} header={header} />
      })}
      {virtualPaddingRight ? (
        // fake empty column to the right for virtualization scroll padding
        <th style={{ display: 'flex', width: virtualPaddingRight }} />
      ) : null}
    </tr>
  )
}

interface TableHeadCellProps {
  header: Header<Person, unknown>
}
```

```
function TableHeadCell({ header }: TableHeadCellProps) {
  return (
    <th
      key={header.id}
      style={{
        display: 'flex',
        width: header.getSize(),
      }}
    >
      <div
        {...{
          className: header.column.getCanSort()
            ? 'cursor-pointer select-none'
            : '',
          onClick: header.column.getToggleSortingHandler(),
        }}
      >
        {flexRender(header.column.columnDef.header, header.getContext())}
        {{
          asc: ' 🔼',
          desc: ' 🔽',
        }[header.column.getIsSorted() as string] ?? null}
      </div>
    </th>
  )
}

interface TableBodyProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  table: Table<Person>
  tableContainerRef: React.RefObject<HTMLDivElement>
  virtualPaddingLeft: number | undefined
  virtualPaddingRight: number | undefined
}

function TableBody({
  columnVirtualizer,
  table,
  tableContainerRef,
  virtualPaddingLeft,
  virtualPaddingRight,
}: TableBodyProps) {
  const { rows } = table.getRowModel()

  // dynamic row height virtualization - alternatively you could use a simpler fixed r
  const rowVirtualizer = useVirtualizer<
    HTMLDivElement,
    HTMLTableRowElement
  >({
    count: rows.length,
    estimateSize: () => 33, // estimate row height for accurate scrollbar dragging
    getScrollElement: () => tableContainerRef.current,
    // measure dynamic row height, except in firefox because it measures table border
    measureElement:
      typeof window !== 'undefined' &&
```

```
      navigator.userAgent.indexOf('Firefox') === -1
        ? (element) => element?.getBoundingClientRect().height
        : undefined,
    overscan: 5,
  })

  const virtualRows = rowVirtualizer.getVirtualItems()

  return (
    <tbody
      style={{
        display: 'grid',
        height: `${rowVirtualizer.getTotalSize()}px`, // tells scrollbar how big the t
        position: 'relative', // needed for absolute positioning of rows
      }}
    >
      {virtualRows.map((virtualRow) => {
        const row = rows[virtualRow.index] as Row<Person>

        return (
          <TableBodyRow
            columnVirtualizer={columnVirtualizer}
            key={row.id}
            row={row}
            rowVirtualizer={rowVirtualizer}
            virtualPaddingLeft={virtualPaddingLeft}
            virtualPaddingRight={virtualPaddingRight}
            virtualRow={virtualRow}
          />
        )
      })}
    </tbody>
  )
}

interface TableBodyRowProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  row: Row<Person>
  rowVirtualizer: Virtualizer<HTMLDivElement, HTMLTableRowElement>
  virtualPaddingLeft: number | undefined
  virtualPaddingRight: number | undefined
  virtualRow: VirtualItem
}

function TableBodyRow({
  columnVirtualizer,
  row,
  rowVirtualizer,
  virtualPaddingLeft,
  virtualPaddingRight,
  virtualRow,
}: TableBodyRowProps) {
  const visibleCells = row.getVisibleCells()
  const virtualColumns = columnVirtualizer.getVirtualItems()
  return (
```

```
    <tr
      data-index={virtualRow.index} // needed for dynamic row height measurement
      ref={(node) => rowVirtualizer.measureElement(node)} // measure dynamic row heigh
      key={row.id}
      style={{
        display: 'flex',
        position: 'absolute',
        transform: `translateY(${virtualRow.start}px)`, // always as style, changes on
        width: '100%',
      }}
    >
      {virtualPaddingLeft ? (
        // fake empty column to the left for virtualization scroll padding
        <td style={{ display: 'flex', width: virtualPaddingLeft }} />
      ) : null}
      {virtualColumns.map((vc) => {
        const cell = visibleCells[vc.index]
        return <TableBodyCell key={cell.id} cell={cell} />
      })}
      {virtualPaddingRight ? (
        // fake empty column to the right for virtualization scroll padding
        <td style={{ display: 'flex', width: virtualPaddingRight }} />
      ) : null}
    </tr>
  )
}

interface TableBodyCellProps {
  cell: Cell<Person, unknown>
}

function TableBodyCell({ cell }: TableBodyCellProps) {
  return (
    <td
      key={cell.id}
      style={{
        display: 'flex',
        width: cell.column.getSize(),
      }}
    >
      {flexRender(cell.column.columnDef.cell, cell.getContext())}
    </td>
  )
}

const rootElement = document.getElementById('root')

if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Virtualized Columns Experimental Github StackBlitz CodeSandbox

Code Explorer | Interactive Sandbox

Files

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Source code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'
import {
  Cell,
  ColumnDef,
  Header,
  HeaderGroup,
  Row,
  Table,
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { useVirtualizer, Virtualizer } from '@tanstack/react-virtual'
import { makeColumns, makeData, Person } from './makeData'

// All important CSS styles are included as inline styles for this example. This is no
function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => makeColumns(1000), [])

  const [data, setData] = React.useState(() => makeData(1000, columns))

  const refreshData = React.useCallback(() => {
    setData(makeData(1000, columns))
  }, [columns])
```

```
  // refresh data every 5 seconds
  React.useEffect(() => {
    const interval = setInterval(() => {
      refreshData()
    }, 5000)
    return () => clearInterval(interval)
  }, [refreshData])

  // The table does not live in the same scope as the virtualizers
  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(),
    getSortedRowModel: getSortedRowModel(),
    debugTable: true,
  })

  return (
    <div className="app">
      {process.env.NODE_ENV === 'development' ? (
        <p>
          <strong>Notice:</strong> You are currently running React in
          development mode. Virtualized rendering performance will be slightly
          degraded until this application is built for production.
        </p>
      ) : null}
      <div>({columns.length.toLocaleString()} columns)</div>
      <div>({data.length.toLocaleString()} rows)</div>
      <button onClick={refreshData}>Refresh Data</button>
      <TableContainer table={table} />
    </div>
  )
}

interface TableContainerProps {
  table: Table<Person>
}

function TableContainer({ table }: TableContainerProps) {
  const visibleColumns = table.getVisibleLeafColumns()

  //The virtualizers need to know the scrollable container element
  const tableContainerRef = React.useRef<HTMLDivElement>(null)

  //we are using a slightly different virtualization strategy for columns (compared to
  const columnVirtualizer = useVirtualizer<
    HTMLDivElement,
    HTMLTableCellElement
  >({
    count: visibleColumns.length,
    estimateSize: (index) => visibleColumns[index].getSize(), //estimate width of each
    getScrollElement: () => tableContainerRef.current,
    horizontal: true,
    overscan: 3, //how many columns to render on each side off screen each way (adjust
    onChange: (instance) => {
```

```
      // requestAnimationFrame(() => {
      const virtualColumns = instance.getVirtualItems()
      // different virtualization strategy for columns - instead of absolute and trans
      const virtualPaddingLeft = virtualColumns[0]?.start ?? 0
      const virtualPaddingRight =
        instance.getTotalSize() -
        (virtualColumns[virtualColumns.length - 1]?.end ?? 0)

      tableContainerRef.current?.style.setProperty(
        '--virtual-padding-left',
        `${virtualPaddingLeft}px`
      )
      tableContainerRef.current?.style.setProperty(
        '--virtual-padding-right',
        `${virtualPaddingRight}px`
      )
      // })
    },
  })

  return (
    <div
      className="container"
      ref={tableContainerRef}
      style={{
        overflow: 'auto', //our scrollable table container
        position: 'relative', //needed for sticky header
        height: '800px', //should be a fixed height
      }}
    >
      {/* Even though we're still using sematic table tags, we must use CSS grid and f
      <table style={{ display: 'grid' }}>
        <TableHead table={table} columnVirtualizer={columnVirtualizer} />
        <TableBody
          columnVirtualizer={columnVirtualizer}
          table={table}
          tableContainerRef={tableContainerRef}
        />
      </table>
    </div>
  )
}

interface TableHeadProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  table: Table<Person>
}

function TableHead({ table, columnVirtualizer }: TableHeadProps) {
  return (
    <thead
      style={{
        display: 'grid',
        position: 'sticky',
        top: 0,
```

```
        zIndex: 1,
      }}
    >
      {table.getHeaderGroups().map((headerGroup) => (
        <TableHeadRow
          columnVirtualizer={columnVirtualizer}
          key={headerGroup.id}
          headerGroup={headerGroup}
        />
      ))}
    </thead>
  )
}

interface TableHeadRowProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  headerGroup: HeaderGroup<Person>
}

function TableHeadRow({ columnVirtualizer, headerGroup }: TableHeadRowProps) {
  const virtualColumnIndexes = columnVirtualizer.getVirtualIndexes()

  return (
    <tr key={headerGroup.id} style={{ display: 'flex', width: '100%' }}>
      {/* fake empty column to the left for virtualization scroll padding */}
      <th className="left-column-spacer" />
      {virtualColumnIndexes.map((virtualColumnIndex) => {
        const header = headerGroup.headers[virtualColumnIndex]
        return (
          <TableHeadCellMemo
            columnVirtualizer={columnVirtualizer}
            key={header.id}
            header={header}
          />
        )
      })}
      {/* fake empty column to the right for virtualization scroll padding */}
      <th className="right-column-spacer" />
    </tr>
  )
}

interface TableHeadCellProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  header: Header<Person, unknown>
}

function TableHeadCell({ columnVirtualizer: _columnVirtualizer, header }: TableHeadCel
  return (
    <th
      key={header.id}
      style={{
        display: 'flex',
        width: header.getSize(),
      }}
```

```
    >
      <div
        {...{
          className: header.column.getCanSort()
            ? 'cursor-pointer select-none'
            : '',
          onClick: header.column.getToggleSortingHandler(),
        }}
      >
        {flexRender(header.column.columnDef.header, header.getContext())}
        {{
          asc: ' 🔼',
          desc: ' 🔽',
        }[header.column.getIsSorted() as string] ?? null}
      </div>
    </th>
  )
}

const TableHeadCellMemo = React.memo(
  TableHeadCell,
  (_prev, next) => next.columnVirtualizer.isScrolling
) as typeof TableHeadCell

interface TableBodyProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  table: Table<Person>
  tableContainerRef: React.RefObject<HTMLDivElement>
}

function TableBody({ columnVirtualizer, table, tableContainerRef }: TableBodyProps) {
  const tableBodyRef = React.useRef<HTMLTableSectionElement>(null)
  const rowRefsMap = React.useRef<Map<number, HTMLTableRowElement>>(new Map())

  const { rows } = table.getRowModel()

  //dynamic row height virtualization - alternatively you could use a simpler fixed ro
  const rowVirtualizer = useVirtualizer<
    HTMLDivElement,
    HTMLTableRowElement
  >({
    count: rows.length,
    estimateSize: () => 33, //estimate row height for accurate scrollbar dragging
    getScrollElement: () => tableContainerRef.current,
    //measure dynamic row height, except in firefox because it measures table border h
    measureElement:
      typeof window !== 'undefined' &&
      navigator.userAgent.indexOf('Firefox') === -1
        ? (element) => element?.getBoundingClientRect().height
        : undefined,
    overscan: 5,
    onChange: (instance) => {
      // requestAnimationFrame(() => {
      tableBodyRef.current!.style.height = `${instance.getTotalSize()}px`
      instance.getVirtualItems().forEach((virtualRow) => {
```

```
        const rowRef = rowRefsMap.current.get(virtualRow.index)
        if (!rowRef) return
        rowRef.style.transform = `translateY(${virtualRow.start}px)`
      })
      // })
    },
  })

  React.useLayoutEffect(() => {
    rowVirtualizer.measure()
  }, [table.getState()])

  const virtualRowIndexes = rowVirtualizer.getVirtualIndexes()

  return (
    <tbody
      ref={tableBodyRef}
      style={{
        display: 'grid',
        position: 'relative', //needed for absolute positioning of rows
      }}
    >
      {virtualRowIndexes.map((virtualRowIndex) => {
        const row = rows[virtualRowIndex] as Row<Person>

        return (
          <TableBodyRow
            columnVirtualizer={columnVirtualizer}
            key={row.id}
            row={row}
            rowVirtualizer={rowVirtualizer}
            virtualRowIndex={virtualRowIndex}
            rowRefsMap={rowRefsMap}
          />
        )
      })}
    </tbody>
  )
}

interface TableBodyRowProps {
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
  row: Row<Person>
  rowVirtualizer: Virtualizer<HTMLDivElement, HTMLTableRowElement>
  virtualRowIndex: number
  rowRefsMap: React.MutableRefObject<Map<number, HTMLTableRowElement>>
}

function TableBodyRow({
  columnVirtualizer,
  row,
  rowVirtualizer,
  virtualRowIndex,
  rowRefsMap,
}: TableBodyRowProps) {
```

```
  const visibleCells = row.getVisibleCells()
  const virtualColumnIndexes = columnVirtualizer.getVirtualIndexes()

  return (
    <tr
      data-index={virtualRowIndex} //needed for dynamic row height measurement
      ref={(node) => {
        if (node) {
          rowVirtualizer.measureElement(node)
          rowRefsMap.current.set(virtualRowIndex, node)
        }
      }} //measure dynamic row height
      key={row.id}
      style={{
        display: 'flex',
        position: 'absolute',
        width: '100%',
      }}
    >
      {/* fake empty column to the left for virtualization scroll padding */}
      <td className="left-column-spacer" />
      {virtualColumnIndexes.map((virtualColumnIndex) => {
        const cell = visibleCells[virtualColumnIndex]
        return (
          <TableBodyCellMemo
            key={cell.id}
            cell={cell}
            columnVirtualizer={columnVirtualizer}
          />
        )
      })}
      {/* fake empty column to the right for virtualization scroll padding */}
      <td className="right-column-spacer" />
    </tr>
  )
}

// TODO: Can rows be memoized in any way without breaking column virtualization?
// const TableBodyRowMemo = React.memo(
//   TableBodyRow,
//   (_prev, next) => next.rowVirtualizer.isScrolling
// )

interface TableBodyCellProps {
  cell: Cell<Person, unknown>
  columnVirtualizer: Virtualizer<HTMLDivElement, HTMLTableCellElement>
}

function TableBodyCell({ cell, columnVirtualizer: _columnVirtualizer }: TableBodyCellP
  return (
    <td
      key={cell.id}
      style={{
        display: 'flex',
        width: cell.column.getSize(),
```

```
      }}
    >
      {flexRender(cell.column.columnDef.cell, cell.getContext())}
    </td>
  )
}

const TableBodyCellMemo = React.memo(
  TableBodyCell,
  (_prev, next) => next.columnVirtualizer.isScrolling
) as typeof TableBodyCell

const rootElement = document.getElementById('root')

if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Additional Links

- Virtualized Columns
- Virtualized Rows

Our Partners

- https://ag-grid.com/react-data-grid/?utm_source=reacttable&utm_campaign=githubreacttable

Resources

- TanStack Query: Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state". Learn More
- TanStack DB: TanStack DB extends TanStack Query with collections, live queries and optimistic mutations that keep your UI reactive, consistent and blazing fast 🔥. Learn More

Subscription

- **Subscribe to Bytes**
  Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.
  No spam. Unsubscribe at any time.

React Example: Virtualized Rows

Github | StackBlitz | CodeSandbox

Code Explorer | Code | Interactive Sandbox

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  Row,
  Table,
  useReactTable,
} from '@tanstack/react-table'
import {
  useVirtualizer,
  VirtualItem,
  Virtualizer,
} from '@tanstack/react-virtual'
import { makeData, Person } from './makeData'

//This is a dynamic row height example, which is more complicated, but allows for a mo
//See https://tanstack.com/virtual/v3/docs/examples/react/table for a simpler fixed ro
function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'id',
      header: 'ID',
      size: 60,
    },
    {
      accessorKey: 'firstName',
      cell: (info) => info.getValue(),
    },
    {
```

```
    accessorFn: (row) => row.lastName,
    id: 'lastName',
    cell: (info) => info.getValue(),
    header: () => <span>Last Name</span>,
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
    size: 50,
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    size: 50,
  },
  {
    accessorKey: 'status',
    header: 'Status',
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    size: 80,
  },
  {
    accessorKey: 'createdAt',
    header: 'Created At',
    cell: (info) => info.getValue<Date>().toLocaleString(),
    size: 250,
  },
], []);

// The virtualizer will need a reference to the scrollable container element
const tableContainerRef = React.useRef<HTMLDivElement>(null);

const [data, setData] = React.useState(() => makeData(50_000));

const refreshData = React.useCallback(() => {
  setData(makeData(50_000));
}, []);

const table = useReactTable({
  data,
  columns,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  debugTable: true,
});

// All important CSS styles are included as inline styles for this example. This is
return (
  <div className="app">
    {process.env.NODE_ENV === 'development' ? (
      <p>
        <strong>Notice:</strong> You are currently running React in
```

```
    development mode. Virtualized rendering performance will be slightly
    degraded until this application is built for production.
  </p>
) : null}
({data.length} rows)
<button onClick={refreshData}>Refresh Data</button>
<div
  className="container"
  ref={tableContainerRef}
  style={{
    overflow: 'auto', //our scrollable table container
    position: 'relative', //needed for sticky header
    height: '800px', //should be a fixed height
  }}
>
  {/* Even though we're still using semantic table tags, we must use CSS grid an
  <table style={{ display: 'grid' }}>
    <thead
      style={{
        display: 'grid',
        position: 'sticky',
        top: 0,
        zIndex: 1,
      }}
    >
      {table.getHeaderGroups().map((headerGroup) => (
        <tr
          key={headerGroup.id}
          style={{ display: 'flex', width: '100%' }}
        >
          {headerGroup.headers.map((header) => {
            return (
              <th
                key={header.id}
                style={{
                  display: 'flex',
                  width: header.getSize(),
                }}
              >
                <div
                  {...{
                    className: header.column.getCanSort()
                      ? 'cursor-pointer select-none'
                      : '',
                    onClick: header.column.getToggleSortingHandler(),
                  }}
                >
                  {flexRender(
                    header.column.columnDef.header,
                    header.getContext()
                  )}
                  {{
                    asc: ' 🔼',
                    desc: ' 🔽',
                  }[header.column.getIsSorted() as string] ?? null}
```

```
                    </div>
                  </th>
                );
              })}
            </tr>
          ))}
        </thead>
        <TableBody table={table} tableContainerRef={tableContainerRef} />
      </table>
    </div>
  </div>
);
}

interface TableBodyProps {
  table: Table<Person>;
  tableContainerRef: React.RefObject<HTMLDivElement>;
}

function TableBody({ table, tableContainerRef }: TableBodyProps) {
  const { rows } = table.getRowModel();

  // Important: Keep the row virtualizer in the lowest component possible to avoid unn
  const rowVirtualizer = useVirtualizer<HTMLDivElement, HTMLTableRowElement>({
    count: rows.length,
    estimateSize: () => 33, //estimate row height for accurate scrollbar dragging
    getScrollElement: () => tableContainerRef.current,
    //measure dynamic row height, except in firefox because it measures table border h
    measureElement:
      typeof window !== 'undefined' &&
      navigator.userAgent.indexOf('Firefox') === -1
        ? (element) => element?.getBoundingClientRect().height
        : undefined,
    overscan: 5,
  });

  return (
    <tbody
      style={{
        display: 'grid',
        height: `${rowVirtualizer.getTotalSize()}px`, //tells scrollbar how big the ta
        position: 'relative', //needed for absolute positioning of rows
      }}
    >
      {rowVirtualizer.getVirtualItems().map((virtualRow) => {
        const row = rows[virtualRow.index] as Row<Person>;
        return (
          <TableBodyRow
            key={row.id}
            row={row}
            virtualRow={virtualRow}
            rowVirtualizer={rowVirtualizer}
          />
        );
      })}
```

```
    </tbody>
  );
}

interface TableBodyRowProps {
  row: Row<Person>;
  virtualRow: VirtualItem;
  rowVirtualizer: Virtualizer<HTMLDivElement, HTMLTableRowElement>;
}

function TableBodyRow({ row, virtualRow, rowVirtualizer }: TableBodyRowProps) {
  return (
    <tr
      data-index={virtualRow.index} //needed for dynamic row height measurement
      ref={(node) => rowVirtualizer.measureElement(node)} //measure dynamic row height
      key={row.id}
      style={{
        display: 'flex',
        position: 'absolute',
        transform: `translateY(${virtualRow.start}px)`, //this should always be a `sty
        width: '100%',
      }}
    >
      {row.getVisibleCells().map((cell) => {
        return (
          <td
            key={cell.id}
            style={{
              display: 'flex',
              width: cell.column.getSize(),
            }}
          >
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        );
      })}
    </tr>
  );
}

const rootElement = document.getElementById('root');

if (!rootElement) throw new Error('Failed to find the root element');

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

React Example: Virtualized Rows Experimental

Github
StackBlitz
CodeSandbox

Code Explorer & Interactive Sandbox

Files

- **src**
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Main React Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { useVirtualizer } from '@tanstack/react-virtual'
import { makeData } from './makeData'
import type { ColumnDef, Row, Table } from '@tanstack/react-table'
import type { Virtualizer } from '@tanstack/react-virtual'
import type { Person } from './makeData'

// This is a dynamic row height example, which is more complicated, but allows for a m
// See https://tanstack.com/virtual/v3/docs/examples/react/table for a simpler fixed r
function App() {
  const columns = React.useMemo<Array<ColumnDef<Person>>>(
    () => [
      {
        accessorKey: 'id',
        header: 'ID',
```

```
      size: 60,
    },
    {
      accessorKey: 'firstName',
      cell: info => info.getValue(),
    },
    {
      accessorFn: row => row.lastName,
      id: 'lastName',
      cell: info => info.getValue(),
      header: () => <span>Last Name</span>,
    },
    {
      accessorKey: 'age',
      header: () => 'Age',
      size: 50,
    },
    {
      accessorKey: 'visits',
      header: () => <span>Visits</span>,
      size: 50,
    },
    {
      accessorKey: 'status',
      header: 'Status',
    },
    {
      accessorKey: 'progress',
      header: 'Profile Progress',
      size: 80,
    },
    {
      accessorKey: 'createdAt',
      header: 'Created At',
      cell: info => info.getValue<Date>().toLocaleString(),
      size: 250,
    },
  ],
  []
)

const [data, _setData] = React.useState(() => makeData(50_000))

const refreshData = React.useCallback(() => {
  _setData(makeData(50_000))
}, [])

// refresh data every 5 seconds
React.useEffect(() => {
  const interval = setInterval(() => {
    refreshData()
  }, 5000)
  return () => clearInterval(interval)
}, [refreshData])
```

```
const table = useReactTable({
  data,
  columns,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  debugTable: true,
})

// The virtualizer needs to know the scrollable container element
const tableContainerRef = React.useRef<HTMLDivElement>(null)

// All important CSS styles are included as inline styles for this example. This is
return (
  <div className="app">
    {process.env.NODE_ENV === 'development' ? (
      <p>
        <strong>Notice:</strong> You are currently running React in development mode
      </p>
    ) : null}
    ({data.length} rows)
    <button onClick={refreshData}>Refresh Data</button>
    <div
      className="container"
      ref={tableContainerRef}
      style={{
        overflow: 'auto',
        position: 'relative',
        height: '800px',
      }}
    >
      {/* Using CSS grid and flexbox for dynamic row heights */}
      <table style={{ display: 'grid' }}>
        <thead
          style={{
            display: 'grid',
            position: 'sticky',
            top: 0,
            zIndex: 1,
          }}
        >
          {table.getHeaderGroups().map((headerGroup) => (
            <tr
              key={headerGroup.id}
              style={{ display: 'flex', width: '100%' }}
            >
              {headerGroup.headers.map((header) => {
                return (
                  <th
                    key={header.id}
                    style={{
                      display: 'flex',
                      width: header.getSize(),
                    }}
                  >
                    <div
```

```
                        {...{
                          className: header.column.getCanSort()
                            ? 'cursor-pointer select-none'
                            : '',
                          onClick: header.column.getToggleSortingHandler(),
                        }}
                      >
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                        {{
                          asc: ' 🔼',
                          desc: ' 🔽',
                        }[header.column.getIsSorted() as string] ?? null}
                      </div>
                    </th>
                  )
                })}
              </tr>
            ))}
          </thead>
          <TableBodyWrapper
            table={table}
            tableContainerRef={tableContainerRef}
          />
        </table>
      </div>
    </div>
  )
}

interface TableBodyWrapperProps {
  table: Table<Person>
  tableContainerRef: React.RefObject<HTMLDivElement>
}

function TableBodyWrapper({ table, tableContainerRef }: TableBodyWrapperProps) {
  const rowRefsMap = React.useRef<Map<number, HTMLTableRowElement>>(new Map())

  const { rows } = table.getRowModel()

  const rowVirtualizer = useVirtualizer<HTMLDivElement, HTMLTableRowElement>({
    count: rows.length,
    estimateSize: () => 33,
    getScrollElement: () => tableContainerRef.current,
    measureElement:
      typeof window !== 'undefined' &&
      navigator.userAgent.indexOf('Firefox') === -1
        ? (element) => element?.getBoundingClientRect().height
        : undefined,
    overscan: 5,
    onChange: (instance) => {
      // Adjust position of virtual rows
      instance.getVirtualItems().forEach((virtualRow) => {
```

```
        const rowRef = rowRefsMap.current.get(virtualRow.index)
        if (!rowRef) return
        rowRef.style.transform = `translateY(${virtualRow.start}px)`
      })
    },
  })

  React.useLayoutEffect(() => {
    rowVirtualizer.measure()
  }, [table.getState()])

  return (
    <TableBody
      rowRefsMap={rowRefsMap}
      rowVirtualizer={rowVirtualizer}
      table={table}
    />
  )
}

interface TableBodyProps {
  table: Table<Person>
  rowVirtualizer: Virtualizer<HTMLDivElement, HTMLTableRowElement>
  rowRefsMap: React.MutableRefObject<Map<number, HTMLTableRowElement>>
}

function TableBody({ rowVirtualizer, table, rowRefsMap }: TableBodyProps) {
  const { rows } = table.getRowModel()
  const virtualRowIndexes = rowVirtualizer.getVirtualIndexes()

  return (
    <tbody
      style={{
        display: 'grid',
        height: `${rowVirtualizer.getTotalSize()}px`,
        position: 'relative',
      }}
    >
      {virtualRowIndexes.map((virtualRowIndex) => {
        const row = rows[virtualRowIndex]
        return (
          <TableBodyRowMemo
            key={row.id}
            row={row}
            rowRefsMap={rowRefsMap}
            rowVirtualizer={rowVirtualizer}
            virtualRowIndex={virtualRowIndex}
          />
        )
      })}
    </tbody>
  )
}

interface TableBodyRowProps {
```

```
  row: Row<Person>
  rowRefsMap: React.MutableRefObject<Map<number, HTMLTableRowElement>>
  rowVirtualizer: Virtualizer<HTMLDivElement, HTMLTableRowElement>
  virtualRowIndex: number
}

function TableBodyRow({
  row,
  rowRefsMap,
  rowVirtualizer,
  virtualRowIndex,
}: TableBodyRowProps) {
  return (
    <tr
      data-index={virtualRowIndex}
      ref={(node) => {
        if (node && typeof virtualRowIndex !== 'undefined') {
          rowVirtualizer.measureElement(node)
          rowRefsMap.current.set(virtualRowIndex, node)
        }
      }}
      key={row.id}
      style={{
        display: 'flex',
        position: 'absolute',
        width: '100%',
      }}
    >
      {row.getVisibleCells().map((cell) => {
        return (
          <td
            key={cell.id}
            style={{
              display: 'flex',
              width: cell.column.getSize(),
            }}
          >
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        )
      })}
    </tr>
  )
}

// Memoized row component to optimize rendering
const TableBodyRowMemo = React.memo(
  TableBodyRow,
  (_prev, next) => next.rowVirtualizer.isScrolling
) as typeof TableBodyRow

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
```

```
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Virtualized Infinite Scrolling

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox

Files:

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Example Code (TypeScript with React)

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

// 3 TanStack Libraries!!!
import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  getSortedRowModel,
  OnChangeFn,
  Row,
  SortingState,
  useReactTable,
} from '@tanstack/react-table'
import {
  keepPreviousData,
  QueryClient,
  QueryClientProvider,
  useInfiniteQuery,
} from '@tanstack/react-query'
import { useVirtualizer } from '@tanstack/react-virtual'

import { fetchData, Person, PersonApiResponse } from './makeData'
```

```
const fetchSize = 50

function App() {
  const tableContainerRef = React.useRef<HTMLDivElement>(null)
  const [sorting, setSorting] = React.useState<SortingState>([])

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'id',
      header: 'ID',
      size: 60,
    },
    {
      accessorKey: 'firstName',
      cell: info => info.getValue(),
    },
    {
      accessorFn: row => row.lastName,
      id: 'lastName',
      cell: info => info.getValue(),
      header: () => <span>Last Name</span>,
    },
    {
      accessorKey: 'age',
      header: () => 'Age',
      size: 50,
    },
    {
      accessorKey: 'visits',
      header: () => <span>Visits</span>,
      size: 50,
    },
    {
      accessorKey: 'status',
      header: 'Status',
    },
    {
      accessorKey: 'progress',
      header: 'Profile Progress',
      size: 80,
    },
    {
      accessorKey: 'createdAt',
      header: 'Created At',
      cell: info => info.getValue<Date>().toLocaleString(),
      size: 200,
    },
  ], []);

  const {
    data,
    fetchNextPage,
    isFetching,
    isLoading,
```

```
} = useInfiniteQuery<PersonApiResponse>({
  queryKey: ['people', sorting],
  queryFn: async ({ pageParam = 0 }) => {
    const start = (pageParam as number) * fetchSize
    const fetchedData = await fetchData(start, fetchSize, sorting)
    return fetchedData
  },
  initialPageParam: 0,
  getNextPageParam: (_lastGroup, groups) => groups.length,
  refetchOnWindowFocus: false,
  placeholderData: keepPreviousData,
})

const flatData = React.useMemo(
  () => data?.pages?.flatMap(page => page.data) ?? [],
  [data]
)

const totalDBRowCount = data?.pages?.[0]?.meta?.totalRowCount ?? 0
const totalFetched = flatData.length

const fetchMoreOnBottomReached = React.useCallback((containerRefElement?: HTMLDivEle
  if (containerRefElement) {
    const { scrollHeight, scrollTop, clientHeight } = containerRefElement
    if (
      scrollHeight - scrollTop - clientHeight < 500 &&
      !isFetching &&
      totalFetched < totalDBRowCount
    ) {
      fetchNextPage()
    }
  }
}, [fetchNextPage, isFetching, totalFetched, totalDBRowCount])

React.useEffect(() => {
  fetchMoreOnBottomReached(tableContainerRef.current)
}, [fetchMoreOnBottomReached])

const table = useReactTable({
  data: flatData,
  columns,
  state: { sorting },
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  manualSorting: true,
  debugTable: true,
})

const handleSortingChange: OnChangeFn<SortingState> = updater => {
  setSorting(updater)
  if (table.getRowModel().rows.length) {
    rowVirtualizer.scrollToIndex?.(0)
  }
}
```

```
table.setOptions(prev => ({
  ...prev,
  onSortingChange: handleSortingChange,
}))

const { rows } = table.getRowModel()

const rowVirtualizer = useVirtualizer({
  count: rows.length,
  estimateSize: () => 33,
  getScrollElement: () => tableContainerRef.current,
  measureElement:
    typeof window !== 'undefined' &&
    navigator.userAgent.indexOf('Firefox') === -1
      ? element => element?.getBoundingClientRect().height
      : undefined,
  overscan: 5,
})

if (isLoading) {
  return <>Loading...</>
}

return (
  <div className="app">
    {process.env.NODE_ENV === 'development' ? (
      <p>
        <strong>Notice:</strong> You are currently running React in
        development mode. Virtualized rendering performance will be slightly
        degraded until this application is built for production.
      </p>
    ) : null}
    ({flatData.length} of {totalDBRowCount} rows fetched)
    <div
      className="container"
      onScroll={e => fetchMoreOnBottomReached(e.currentTarget)}
      ref={tableContainerRef}
      style={{
        overflow: 'auto',
        position: 'relative',
        height: '600px',
      }}
    >
      {/* CSS grid and flexbox are used for dynamic row heights */}
      <table style={{ display: 'grid' }}>
        <thead
          style={{
            display: 'grid',
            position: 'sticky',
            top: 0,
            zIndex: 1,
          }}
        >
          {table.getHeaderGroups().map(headerGroup => (
            <tr
```

```
      key={headerGroup.id}
      style={{ display: 'flex', width: '100%' }}
    >
      {headerGroup.headers.map(header => (
        <th
          key={header.id}
          style={{
            display: 'flex',
            width: header.getSize(),
          }}
        >
          <div
            {...{
              className: header.column.getCanSort()
                ? 'cursor-pointer select-none'
                : '',
              onClick: header.column.getToggleSortingHandler(),
            }}
          >
            {flexRender(
              header.column.columnDef.header,
              header.getContext()
            )}
            {{
              asc: ' 🔼',
              desc: ' 🔽',
            }[header.column.getIsSorted() as string] ?? null}
          </div>
        </th>
      ))}
    </tr>
  ))}
</thead>
<tbody
  style={{
    display: 'grid',
    height: `${rowVirtualizer.getTotalSize()}px`,
    position: 'relative',
  }}
>
  {rowVirtualizer.getVirtualItems().map(virtualRow => {
    const row = rows[virtualRow.index] as Row<Person>
    return (
      <tr
        data-index={virtualRow.index}
        ref={node => rowVirtualizer.measureElement(node)}
        key={row.id}
        style={{
          display: 'flex',
          position: 'absolute',
          transform: `translateY(${virtualRow.start}px)`,
          width: '100%',
        }}
      >
        {row.getVisibleCells().map(cell => (
```

```
                    <td
                      key={cell.id}
                      style={{
                        display: 'flex',
                        width: cell.column.getSize(),
                      }}
                    >
                      {flexRender(cell.column.columnDef.cell, cell.getContext())}
                    </td>
                  ))}
                </tr>
              )
            })}
          </tbody>
        </table>
      </div>
      {isFetching && <div>Fetching More...</div>}
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

const queryClient = new QueryClient()

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </React.StrictMode>
)
```

React Example: Kitchen Sink

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Directory Structure

- `src`
  - `components`
  - `App.tsx`
  - `hooks.tsx`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
  - `tableModels.tsx`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Sample `tsx` code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

import './index.css'

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Additional Links

- Virtualized Infinite Scrolling
- React Bootstrap

Our Partners

Partner Logo

TanStack Start

Full-document SSR, Streaming, Server Functions, bundling, and more, powered by TanStack Router and Vite - Ready to deploy to your favorite hosting provider.
Learn More

TanStack Store

The immutable-reactive data store that powers the core of TanStack libraries and their framework adapters.
Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

Subscribe
Subscribe
No spam. Unsubscribe at *any* time.

React Example: Bootstrap

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

File Structure

- `src`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

TypeScript Code

```
import * as React from 'react'
import ReactDOM from 'react-dom/client'

import 'bootstrap/dist/css/bootstrap.min.css'

import { Table as BTable } from 'react-bootstrap'

import {
  ColumnDef,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const columns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        footer: (props) => props.column.id,
      },
      {
        accessorFn: (row) => row.lastName,
        id: 'lastName',
```

```
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const [data, setData] = React.useState(makeData(10))
  const rerender = () => setData(makeData(10))

  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(),
  })

  return (
    <div className="p-2">
      <BTable striped bordered hover responsive size="sm">
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
```

```
              {headerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => (
            <tr key={row.id}>
              {row.getVisibleCells().map(cell => (
                <td key={cell.id}>
                  {flexRender(cell.column.columnDef.cell, cell.getContext())}
                </td>
              ))}
            </tr>
          ))}
        </tbody>
        <tfoot>
          {table.getFooterGroups().map(footerGroup => (
            <tr key={footerGroup.id}>
              {footerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.footer,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </tfoot>
      </BTable>
      <div className="h-4">
        <button onClick={() => rerender()} className="border p-2">
          Rerender
        </button>
      </div>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
```

```
    <App />
  </React.StrictMode>
)
```

React Example: Filters Faceted

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- **src**
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Column,
  ColumnDef,
  ColumnFiltersState,
  RowData,
  flexRender,
  getCoreRowModel,
  getFacetedMinMaxValues,
  getFacetedRowModel,
  getFacetedUniqueValues,
  getFilteredRowModel,
  getPaginationRowModel,
  getSortedRowModel,
  useReactTable,
} from '@tanstack/react-table'

import { makeData, Person } from './makeData'

declare module '@tanstack/react-table' {
  //allows us to define custom properties for our columns
  interface ColumnMeta<TData extends RowData, TValue> {
    filterVariant?: 'text' | 'range' | 'select'
  }
}

function App() {
```

```
const rerender = React.useReducer(() => ({}), {})[1]

const [columnFilters, setColumnFilters] = React.useState<ColumnFiltersState>(
  []
)

const columns = React.useMemo<ColumnDef<Person, any>[]>(() => [
  {
    accessorKey: 'firstName',
    cell: info => info.getValue(),
  },
  {
    accessorFn: row => row.lastName,
    id: 'lastName',
    cell: info => info.getValue(),
    header: () => <span>Last Name</span>,
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
    meta: {
      filterVariant: 'range',
    },
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    meta: {
      filterVariant: 'range',
    },
  },
  {
    accessorKey: 'status',
    header: 'Status',
    meta: {
      filterVariant: 'select',
    },
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    meta: {
      filterVariant: 'range',
    },
  },
], [])

const [data, setData] = React.useState<Person[]>(() => makeData(5000))
const refreshData = () => setData(_old => makeData(100000)) //stress test

const table = useReactTable({
  data,
  columns,
  state: {
    columnFilters,
```

```
  },
  onColumnFiltersChange: setColumnFilters,
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(), //client-side filtering
  getSortedRowModel: getSortedRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  getFacetedRowModel: getFacetedRowModel(), // client-side faceting
  getFacetedUniqueValues: getFacetedUniqueValues(), // generate unique values for se
  getFacetedMinMaxValues: getFacetedMinMaxValues(), // generate min/max values for r
  debugTable: true,
  debugHeaders: true,
  debugColumns: false,
})

return (
  <div className="p-2">
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <>
                      <div
                        {...{
                          className: header.column.getCanSort()
                            ? 'cursor-pointer select-none'
                            : '',
                          onClick: header.column.getToggleSortingHandler(),
                        }}
                      >
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                        {{
                          asc: ' 🔼',
                          desc: ' 🔽',
                        }[header.column.getIsSorted() as string] ?? null}
                      </div>
                      {header.column.getCanFilter() ? (
                        <div>
                          <Filter column={header.column} />
                        </div>
                      ) : null}
                    </>
                  )}
                </th>
              )
            })}
          </tr>
        ))}
      </thead>
```

```
  <tbody>
    {table.getRowModel().rows.map(row => {
      return (
        <tr key={row.id}>
          {row.getVisibleCells().map(cell => {
            return (
              <td key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
      {table.getPageCount()}
    </strong>
```

```
        </span>
        <span className="flex items-center gap-1">
          | Go to page:
          <input
            type="number"
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={e => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={e => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map(pageSize => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>{table.getPrePaginationRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>
        {JSON.stringify(
          { columnFilters: table.getState().columnFilters },
          null,
          2
        )}
      </pre>
    </div>
  )
}

function Filter({ column }: { column: Column<any, unknown> }) {
  const { filterVariant } = column.columnDef.meta ?? {}

  const columnFilterValue = column.getFilterValue()

  const sortedUniqueValues = React.useMemo(
    () =>
      filterVariant === 'range'
        ? []
        : Array.from(column.getFacetedUniqueValues().keys())
            .sort()
```

```
          .slice(0, 5000),
  [column.getFacetedUniqueValues(), filterVariant]
)

return filterVariant === 'range' ? (
  <div>
    <div className="flex space-x-2">
      <DebouncedInput
        type="number"
        min={Number(column.getFacetedMinMaxValues()?.[0] ?? '')}
        max={Number(column.getFacetedMinMaxValues()?.[1] ?? '')}
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={value =>
          column.setFilterValue((old: [number, number]) => [value, old?.[1]])
        }
        placeholder={`Min ${
          column.getFacetedMinMaxValues()?.[0] !== undefined
            ? `(${column.getFacetedMinMaxValues()?.[0]})`
            : ''
        }`}
        className="w-24 border shadow rounded"
      />
      <DebouncedInput
        type="number"
        min={Number(column.getFacetedMinMaxValues()?.[0] ?? '')}
        max={Number(column.getFacetedMinMaxValues()?.[1] ?? '')}
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={value =>
          column.setFilterValue((old: [number, number]) => [old?.[0], value])
        }
        placeholder={`Max ${
          column.getFacetedMinMaxValues()?.[1]
            ? `(${column.getFacetedMinMaxValues()?.[1]})`
            : ''
        }`}
        className="w-24 border shadow rounded"
      />
    </div>
    <div className="h-1" />
  </div>
) : filterVariant === 'select' ? (
  <select
    onChange={e => column.setFilterValue(e.target.value)}
    value={columnFilterValue?.toString()}
  >
    <option value="">All</option>
    {sortedUniqueValues.map(value => (
      //dynamically generated select options from faceted values feature
      <option value={value} key={value}>
        {value}
      </option>
    ))}
  </select>
) : (
  <>
```

```
      {/* Autocomplete suggestions from faceted values feature */}
      <datalist id={column.id + 'list'}>
        {sortedUniqueValues.map((value: any) => (
          <option value={value} key={value} />
        ))}
      </datalist>
      <DebouncedInput
        type="text"
        value={(columnFilterValue ?? '') as string}
        onChange={value => column.setFilterValue(value)}
        placeholder={`Search... (${column.getFacetedUniqueValues().size})`}
        className="w-36 border shadow rounded"
        list={column.id + 'list'}
      />
      <div className="h-1" />
    </>
  )
}

// A typical debounced input react component
function DebouncedInput({
  value: initialValue,
  onChange,
  debounce = 500,
  ...props
}: {
  value: string | number
  onChange: (value: string | number) => void
  debounce?: number
} & Omit<React.InputHTMLAttributes<HTMLInputElement>, 'onChange'>) {
  const [value, setValue] = React.useState(initialValue)

  React.useEffect(() => {
    setValue(initialValue)
  }, [initialValue])

  React.useEffect(() => {
    const timeout = setTimeout(() => {
      onChange(value)
    }, debounce)

    return () => clearTimeout(timeout)
  }, [value])

  return (
    <input {...props} value={value} onChange={e => setValue(e.target.value)} />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
```

```
  </React.StrictMode>
)
```

React Example: Material UI Pagination

Github
StackBlitz
CodeSandbox

Code Explorer & Interactive Sandbox

Source Files

- `actions.tsx`
- `index.css`
- `main.tsx`
- `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'
import Box from '@mui/material/Box'
import Table from '@mui/material/Table'
import TableBody from '@mui/material/TableBody'
import TableCell from '@mui/material/TableCell'
import TableContainer from '@mui/material/TableContainer'
import TableHead from '@mui/material/TableHead'
import TableRow from '@mui/material/TableRow'
import TablePagination from '@mui/material/TablePagination'
import InputBase from '@mui/material/InputBase'
import Paper from '@mui/material/Paper'

import {
  Column,
  Table as ReactTable,
  useReactTable,
  getCoreRowModel,
  getFilteredRowModel,
  getPaginationRowModel,
  ColumnDef,
  flexRender,
} from '@tanstack/react-table'

import TablePaginationActions from './actions'
import { makeData, Person } from './makeData'
```

```
function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const columns = React.useMemo<ColumnDef<Person>[]>(
    () => [
      {
        header: 'Name',
        footer: (props) => props.column.id,
        columns: [
          {
            accessorKey: 'firstName',
            cell: (info) => info.getValue(),
            footer: (props) => props.column.id,
          },
          {
            accessorFn: (row) => row.lastName,
            id: 'lastName',
            cell: (info) => info.getValue(),
            header: () => <span>Last Name</span>,
            footer: (props) => props.column.id,
          },
        ],
      },
      {
        header: 'Info',
        footer: (props) => props.column.id,
        columns: [
          {
            accessorKey: 'age',
            header: () => 'Age',
            footer: (props) => props.column.id,
          },
          {
            header: 'More Info',
            columns: [
              {
                accessorKey: 'visits',
                header: () => <span>Visits</span>,
                footer: (props) => props.column.id,
              },
              {
                accessorKey: 'status',
                header: 'Status',
                footer: (props) => props.column.id,
              },
              {
                accessorKey: 'progress',
                header: 'Profile Progress',
                footer: (props) => props.column.id,
              },
            ],
          },
        ],
      },
```

```
    ],
    []
  )

  const [data, setData] = React.useState(() => makeData(100000))
  const refreshData = () => setData(() => makeData(100000))

  return (
    <>
      <LocalTable {...{ data, columns }} />
      <hr />
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
    </>
  )
}

function LocalTable({
  data,
  columns,
}: {
  data: Person[]
  columns: ColumnDef<Person>[]
}) {
  const table = useReactTable({
    data,
    columns,
    // Pipeline
    getCoreRowModel: getCoreRowModel(),
    getFilteredRowModel: getFilteredRowModel(),
    getPaginationRowModel: getPaginationRowModel(),
    //
    debugTable: true,
  })

  const { pageSize, pageIndex } = table.getState().pagination

  return (
    <Box sx={{ width: '100%' }}>
      <TableContainer component={Paper}>
        <Table sx={{ minWidth: 650 }} aria-label="simple table">
          <TableHead>
            {table.getHeaderGroups().map((headerGroup) => (
              <TableRow key={headerGroup.id}>
                {headerGroup.headers.map((header) => {
                  return (
                    <TableCell key={header.id} colSpan={header.colSpan}>
                      {header.isPlaceholder ? null : (
                        <div>
                          {flexRender(
                            header.column.columnDef.header,
```

```
                    header.getContext()
                  )}
                  {header.column.getCanFilter() ? (
                    <div>
                      <Filter column={header.column} table={table} />
                    </div>
                  ) : null}
                </div>
              )}
            </TableCell>
          )
        })}
      </TableRow>
    ))}
  </TableHead>
  <TableBody>
    {table.getRowModel().rows.map((row) => {
      return (
        <TableRow key={row.id}>
          {row.getVisibleCells().map((cell) => {
            return (
              <TableCell key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </TableCell>
            )
          })}
        </TableRow>
      )
    })}
  </TableBody>
</Table>
</TableContainer>
<TablePagination
  rowsPerPageOptions={[5, 10, 25, { label: 'All', value: data.length }]}
  component="div"
  count={table.getFilteredRowModel().rows.length}
  rowsPerPage={pageSize}
  page={pageIndex}
  slotProps={{
    select: {
      inputProps: { 'aria-label': 'rows per page' },
      native: true,
    },
  }}
  onPageChange={(_, page) => {
    table.setPageIndex(page)
  }}
  onRowsPerPageChange={(e) => {
    const size = e.target.value ? Number(e.target.value) : 10
    table.setPageSize(size)
  }}
  ActionsComponent={TablePaginationActions}
```

```
      />
      <pre>{JSON.stringify(table.getState().pagination, null, 2)}</pre>
    </Box>
  )
}

function Filter({
  column,
  table,
}: {
  column: Column<any, any>
  table: ReactTable<any>
}) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id)

  const columnFilterValue = column.getFilterValue()

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <InputBase
        type="number"
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            e.target.value,
            old?.[1],
          ])
        }
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <InputBase
        type="number"
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            old?.[0],
            e.target.value,
          ])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
        inputProps={{ 'aria-label': 'search' }}
      />
    </div>
  ) : (
    <InputBase
      value={(columnFilterValue ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
      inputProps={{ 'aria-label': 'search' }}
    />
```

```
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Full Width Table

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

File Structure

- `src`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

TypeScript React Table Full Width Example

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'

import {
  PaginationState,
  useReactTable,
  getCoreRowModel,
  getPaginationRowModel,
  ColumnDef,
  flexRender,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

function App() {
  const rerender = React.useReducer(() => ({}), {})[1]

  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      header: 'Name',
      footer: (props) => props.column.id,
      columns: [
        {
          accessorKey: 'firstName',
          cell: (info) => info.getValue(),
          footer: (props) => props.column.id,
        },
        {
```

```
        accessorFn: (row) => row.lastName,
        id: 'lastName',
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        accessorKey: 'visits',
        header: () => <span>Visits</span>,
        footer: (props) => props.column.id,
      },
      {
        accessorKey: 'status',
        header: 'Status',
        footer: (props) => props.column.id,
      },
      {
        accessorKey: 'progress',
        header: 'Profile Progress',
        footer: (props) => props.column.id,
      },
    ],
  },
], [])

const [data, setData] = React.useState(() => makeData(100000))
const refreshData = () => setData(() => makeData(100000))

const [pagination, setPagination] = React.useState<PaginationState>({
  pageIndex: 0,
  pageSize: 10,
})

const table = useReactTable({
  data,
  columns,
  state: {
    pagination,
  },
  onPaginationChange: setPagination,
  getCoreRowModel: getCoreRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  debugTable: true,
})
```

```
return (
  <>
    <div className="p-2 block max-w-full overflow-x-scroll overflow-y-hidden">
      <div className="h-2" />
      <table className="w-full ">
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => {
                return (
                  <th key={header.id} colSpan={header.colSpan}>
                    {header.isPlaceholder ? null : (
                      <div>
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                      </div>
                    )}
                  </th>
                )
              })}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => {
            return (
              <tr key={row.id}>
                {row.getVisibleCells().map(cell => {
                  return (
                    <td key={cell.id}>
                      {flexRender(
                        cell.column.columnDef.cell,
                        cell.getContext()
                      )}
                    </td>
                  )
                })}
              </tr>
            )
          })}
        </tbody>
      </table>
      <div className="h-2" />
      <div className="flex items-center gap-2">
        <button
          className="border rounded p-1"
          onClick={() => table.setPageIndex(0)}
          disabled={!table.getCanPreviousPage()}
        >
          {'<<'}
        </button>
        <button
```

```
        className="border rounded p-1"
        onClick={() => table.previousPage()}
        disabled={!table.getCanPreviousPage()}
      >
        {'<'}
      </button>
      <button
        className="border rounded p-1"
        onClick={() => table.nextPage()}
        disabled={!table.getCanNextPage()}
      >
        {'>'}
      </button>
      <button
        className="border rounded p-1"
        onClick={() => table.setPageIndex(table.getPageCount() - 1)}
        disabled={!table.getCanNextPage()}
      >
        {'>>'}
      </button>
      <span className="flex items-center gap-1">
        <div>Page</div>
        <strong>
          {table.getState().pagination.pageIndex + 1} of {table.getPageCount()}
        </strong>
      </span>
      <span className="flex items-center gap-1">
        | Go to page:
        <input
          type="number"
          min="1"
          max={table.getPageCount()}
          defaultValue={table.getState().pagination.pageIndex + 1}
          onChange={e => {
            const page = e.target.value ? Number(e.target.value) - 1 : 0
            table.setPageIndex(page)
          }}
          className="border p-1 rounded w-16"
        />
      </span>
      <select
        value={table.getState().pagination.pageSize}
        onChange={e => {
          table.setPageSize(Number(e.target.value))
        }}
      >
        {[10, 20, 30, 40, 50].map(pageSize => (
          <option key={pageSize} value={pageSize}>
            Show {pageSize}
          </option>
        ))}
      </select>
    </div>
    <div>{table.getRowModel().rows.length} Rows</div>
  </div>
```

```
      <hr />
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>{JSON.stringify(pagination, null, 2)}</pre>
    </>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Full Width Resizable Table

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

Files:

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import './index.css'

import {
  getCoreRowModel,
  ColumnDef,
  flexRender,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const columns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: info => info.getValue(),
        footer: props => props.column.id,
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      },
    ],
  },
  {
```

```
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: props => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: props => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: props => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const data = React.useMemo(() => makeData(20), [])

  const table = useReactTable({
    data,
    columns,
    enableColumnResizing: true,
    columnResizeMode: 'onChange',
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
  })

  return (
    <div className="p-2 block max-w-full overflow-x-scroll overflow-y-hidden">
      <div className="h-2" />
      <table className="w-full ">
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => {
                return (
```

```
                  <th
                    key={header.id}
                    colSpan={header.colSpan}
                    style={{ position: 'relative', width: header.getSize() }}
                  >
                    {header.isPlaceholder
                      ? null
                      : flexRender(header.column.columnDef.header, header.getContext()
                    {header.column.getCanResize() && (
                      <div
                        onMouseDown={header.getResizeHandler()}
                        onTouchStart={header.getResizeHandler()}
                        className={`resizer ${
                          header.column.getIsResizing() ? 'isResizing' : ''
                        }`}
                      ></div>
                    )}
                  </th>
                )
              })}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => {
            return (
              <tr key={row.id}>
                {row.getVisibleCells().map(cell => {
                  return (
                    <td key={cell.id} style={{ width: cell.column.getSize() }}>
                      {flexRender(cell.column.columnDef.cell, cell.getContext())}
                    </td>
                  )
                })}
              </tr>
            )
          })}
        </tbody>
      </table>
      <div className="h-4" />
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'
```

```
import './index.css'

import {
  getCoreRowModel,
  ColumnDef,
  flexRender,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const columns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: info => info.getValue(),
        footer: props => props.column.id,
      },
      {
        accessorFn: row => row.lastName,
        id: 'lastName',
        cell: info => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: props => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: props => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: props => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: props => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: props => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: props => props.column.id,
```

```
          },
        ],
      },
    ],
  },
]

function App() {
  const data = React.useMemo(() => makeData(20), [])

  const table = useReactTable({
    data,
    columns,
    enableColumnResizing: true,
    columnResizeMode: 'onChange',
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
  })

  return (
    <div className="p-2 block max-w-full overflow-x-scroll overflow-y-hidden">
      <div className="h-2" />
      <table className="w-full ">
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => {
                return (
                  <th
                    key={header.id}
                    colSpan={header.colSpan}
                    style={{ position: 'relative', width: header.getSize() }}
                  >
                    {header.isPlaceholder
                      ? null
                      : flexRender(header.column.columnDef.header, header.getContext()
                    {header.column.getCanResize() && (
                      <div
                        onMouseDown={header.getResizeHandler()}
                        onTouchStart={header.getResizeHandler()}
                        className={`resizer ${
                          header.column.getIsResizing() ? 'isResizing' : ''
                        }`}
                      ></div>
                    )}
                  </th>
                )
              })}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => {
```

```
            return (
              <tr key={row.id}>
                {row.getVisibleCells().map(cell => {
                  return (
                    <td key={cell.id} style={{ width: cell.column.getSize() }}>
                      {flexRender(cell.column.columnDef.cell, cell.getContext())}
                    </td>
                  )
                })}
              </tr>
            )
          })}
        </tbody>
      </table>
      <div className="h-4" />
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Custom Features

Github | StackBlitz | CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

React Application

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  useReactTable,
  makeStateUpdater,
  getSortedRowModel,
  getPaginationRowModel,
  getFilteredRowModel,
  getCoreRowModel,
  flexRender,
  TableFeature,
  Table,
  RowData,
  OnChangeFn,
  ColumnDef,
  Column,
  Updater,
  functionalUpdate,
} from '@tanstack/react-table'

import { makeData, Person } from './makeData'

// Define custom state type for the new feature
export type DensityState = 'sm' | 'md' | 'lg'
interface DensityTableState {
```

```
  density: DensityState
}

interface DensityOptions {
  enableDensity?: boolean
  onDensityChange?: OnChangeFn<DensityState>
}

interface DensityInstance {
  setDensity: (updater: Updater<DensityState>) => void
  toggleDensity: (value?: DensityState) => void
}

declare module '@tanstack/react-table' {
  interface TableState extends DensityTableState {}
  interface TableOptionsResolved<TData extends RowData>
    extends DensityOptions {}
  interface Table<TData extends RowData> extends DensityInstance {}
}

// Define the custom feature
export const DensityFeature: TableFeature<any> = {
  getInitialState: (state): DensityTableState => ({
    density: 'md',
    ...state,
  }),
  getDefaultOptions: <TData extends RowData>(
    table: Table<TData>
  ): DensityOptions => ({
    enableDensity: true,
    onDensityChange: makeStateUpdater('density', table),
  }),
  createTable: <TData extends RowData>(table: Table<TData>): void => {
    table.setDensity = (updater) => {
      const safeUpdater: Updater<DensityState> = (old) => {
        return functionalUpdate(updater, old)
      }
      return table.options.onDensityChange?.(safeUpdater)
    }
    table.toggleDensity = (value) => {
      table.setDensity((old) => {
        if (value) return value
        return old === 'lg' ? 'md' : old === 'md' ? 'sm' : 'lg'
      })
    }
  },
}

// Main App component
function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'firstName',
      cell: (info) => info.getValue(),
      footer: (props) => props.column.id,
```

```
  },
  {
    accessorFn: (row) => row.lastName,
    id: 'lastName',
    cell: (info) => info.getValue(),
    header: () => <span>Last Name</span>,
    footer: (props) => props.column.id,
  },
  {
    accessorKey: 'age',
    header: () => 'Age',
    footer: (props) => props.column.id,
  },
  {
    accessorKey: 'visits',
    header: () => <span>Visits</span>,
    footer: (props) => props.column.id,
  },
  {
    accessorKey: 'status',
    header: 'Status',
    footer: (props) => props.column.id,
  },
  {
    accessorKey: 'progress',
    header: 'Profile Progress',
    footer: (props) => props.column.id,
  },
], []);

const [data] = React.useState(() => makeData(1000))
const [density, setDensity] = React.useState<DensityState>('md')

const table = useReactTable({
  _features: [DensityFeature],
  columns,
  data,
  debugTable: true,
  getCoreRowModel: getCoreRowModel(),
  getSortedRowModel: getSortedRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  state: { density },
  onDensityChange: setDensity,
})

return (
  <div className="p-2">
    <div className="h-2" />
    <button
      onClick={() => table.toggleDensity()}
      className="border rounded p-1 bg-blue-500 text-white mb-2 w-64"
    >
      Toggle Density
    </button>
```

```
<table>
  <thead>
    {table.getHeaderGroups().map((headerGroup) => (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map((header) => (
          <th
            key={header.id}
            colSpan={header.colSpan}
            style={{
              padding:
                density === 'sm'
                  ? '4px'
                  : density === 'md'
                  ? '8px'
                  : '16px',
              transition: 'padding 0.2s',
            }}
          >
            <div
              {...{
                className: header.column.getCanSort()
                  ? 'cursor-pointer select-none'
                  : '',
                onClick: header.column.getToggleSortingHandler(),
              }}
            >
              {flexRender(
                header.column.columnDef.header,
                header.getContext()
              )}
              {{
                asc: ' 🔼',
                desc: ' 🔽',
              }[header.column.getIsSorted() as string] ?? null}
            </div>
            {header.column.getCanFilter() ? (
              <div>
                <Filter column={header.column} table={table} />
              </div>
            ) : null}
          </th>
        ))}
      </tr>
    ))}
  </thead>
  <tbody>
    {table.getRowModel().rows.map((row) => (
      <tr key={row.id}>
        {row.getVisibleCells().map((cell) => (
          <td
            key={cell.id}
            style={{
              padding:
                density === 'sm'
                  ? '4px'
```

```
                  : density === 'md'
                  ? '8px'
                  : '16px',
              transition: 'padding 0.2s',
            }}
          >
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        ))}
      </tr>
    ))}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.firstPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.lastPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
      {table.getPageCount().toLocaleString()}
    </strong>
  </span>
  <span className="flex items-center gap-1">
    | Go to page:
    <input
      type="number"
      defaultValue={table.getState().pagination.pageIndex + 1}
```

```
            onChange={(e) => {
              const page = e.target.value
                ? Number(e.target.value) - 1
                : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={(e) => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map((pageSize) => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>
        Showing {table.getRowModel().rows.length.toLocaleString()} of{' '}
        {table.getRowCount().toLocaleString()} Rows
      </div>
      <pre>{JSON.stringify(table.getState().pagination, null, 2)}</pre>
    </div>
  )
}

function Filter({
  column,
  table,
}: {
  column: Column<any, any>
  table: Table<any>
}) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id)

  const columnFilterValue = column.getFilterValue()

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            e.target.value,
            old?.[1],
          ])
        }
```

```
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            old?.[0],
            e.target.value,
          ])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(columnFilterValue ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  useReactTable,
  makeStateUpdater,
  getSortedRowModel,
  getPaginationRowModel,
  getFilteredRowModel,
  getCoreRowModel,
  flexRender,
  TableFeature,
  Table,
  RowData,
  OnChangeFn,
  ColumnDef,
  Column,
  Updater,
```

```
  functionalUpdate,
} from '@tanstack/react-table'

import { makeData, Person } from './makeData'

// Define custom state type for the new feature
export type DensityState = 'sm' | 'md' | 'lg'
interface DensityTableState {
  density: DensityState
}

interface DensityOptions {
  enableDensity?: boolean
  onDensityChange?: OnChangeFn<DensityState>
}

interface DensityInstance {
  setDensity: (updater: Updater<DensityState>) => void
  toggleDensity: (value?: DensityState) => void
}

declare module '@tanstack/react-table' {
  interface TableState extends DensityTableState {}
  interface TableOptionsResolved<TData extends RowData>
    extends DensityOptions {}
  interface Table<TData extends RowData> extends DensityInstance {}
}

// Define the custom feature
export const DensityFeature: TableFeature<any> = {
  getInitialState: (state): DensityTableState => ({
    density: 'md',
    ...state,
  }),
  getDefaultOptions: <TData extends RowData>(
    table: Table<TData>
  ): DensityOptions => ({
    enableDensity: true,
    onDensityChange: makeStateUpdater('density', table),
  }),
  createTable: <TData extends RowData>(table: Table<TData>): void => {
    table.setDensity = (updater) => {
      const safeUpdater: Updater<DensityState> = (old) => {
        return functionalUpdate(updater, old)
      }
      return table.options.onDensityChange?.(safeUpdater)
    }
    table.toggleDensity = (value) => {
      table.setDensity((old) => {
        if (value) return value
        return old === 'lg' ? 'md' : old === 'md' ? 'sm' : 'lg'
      })
    }
  },
}
```

```
// Main App component
function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(() => [
    {
      accessorKey: 'firstName',
      cell: (info) => info.getValue(),
      footer: (props) => props.column.id,
    },
    {
      accessorFn: (row) => row.lastName,
      id: 'lastName',
      cell: (info) => info.getValue(),
      header: () => <span>Last Name</span>,
      footer: (props) => props.column.id,
    },
    {
      accessorKey: 'age',
      header: () => 'Age',
      footer: (props) => props.column.id,
    },
    {
      accessorKey: 'visits',
      header: () => <span>Visits</span>,
      footer: (props) => props.column.id,
    },
    {
      accessorKey: 'status',
      header: 'Status',
      footer: (props) => props.column.id,
    },
    {
      accessorKey: 'progress',
      header: 'Profile Progress',
      footer: (props) => props.column.id,
    },
  ], []);

  const [data] = React.useState(() => makeData(1000))
  const [density, setDensity] = React.useState<DensityState>('md')

  const table = useReactTable({
    _features: [DensityFeature],
    columns,
    data,
    debugTable: true,
    getCoreRowModel: getCoreRowModel(),
    getSortedRowModel: getSortedRowModel(),
    getFilteredRowModel: getFilteredRowModel(),
    getPaginationRowModel: getPaginationRowModel(),
    state: { density },
    onDensityChange: setDensity,
  })

  return (
```

```
<div className="p-2">
  <div className="h-2" />
  <button
    onClick={() => table.toggleDensity()}
    className="border rounded p-1 bg-blue-500 text-white mb-2 w-64"
  >
    Toggle Density
  </button>
  <table>
    <thead>
      {table.getHeaderGroups().map((headerGroup) => (
        <tr key={headerGroup.id}>
          {headerGroup.headers.map((header) => (
            <th
              key={header.id}
              colSpan={header.colSpan}
              style={{
                padding:
                  density === 'sm'
                    ? '4px'
                    : density === 'md'
                    ? '8px'
                    : '16px',
                transition: 'padding 0.2s',
              }}
            >
              <div
                {...{
                  className: header.column.getCanSort()
                    ? 'cursor-pointer select-none'
                    : '',
                  onClick: header.column.getToggleSortingHandler(),
                }}
              >
                {flexRender(
                  header.column.columnDef.header,
                  header.getContext()
                )}
                {{
                  asc: ' 🔼',
                  desc: ' 🔽',
                }[header.column.getIsSorted() as string] ?? null}
              </div>
              {header.column.getCanFilter() ? (
                <div>
                  <Filter column={header.column} table={table} />
                </div>
              ) : null}
            </th>
          ))}
        </tr>
      ))}
    </thead>
    <tbody>
      {table.getRowModel().rows.map((row) => (
```

```
      <tr key={row.id}>
        {row.getVisibleCells().map((cell) => (
          <td
            key={cell.id}
            style={{
              padding:
                density === 'sm'
                  ? '4px'
                  : density === 'md'
                  ? '8px'
                  : '16px',
              transition: 'padding 0.2s',
            }}
          >
            {flexRender(cell.column.columnDef.cell, cell.getContext())}
          </td>
        ))}
      </tr>
    ))}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.firstPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.lastPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
```

```
            {table.getPageCount().toLocaleString()}
          </strong>
        </span>
        <span className="flex items-center gap-1">
          | Go to page:
          <input
            type="number"
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={(e) => {
              const page = e.target.value
                ? Number(e.target.value) - 1
                : 0
              table.setPageIndex(page)
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={(e) => {
            table.setPageSize(Number(e.target.value))
          }}
        >
          {[10, 20, 30, 40, 50].map((pageSize) => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>
        Showing {table.getRowModel().rows.length.toLocaleString()} of{' '}
        {table.getRowCount().toLocaleString()} Rows
      </div>
      <pre>{JSON.stringify(table.getState().pagination, null, 2)}</pre>
    </div>
  )
}

function Filter({
  column,
  table,
}: {
  column: Column<any, any>
  table: Table<any>
}) {
  const firstValue = table
    .getPreFilteredRowModel()
    .flatRows[0]?.getValue(column.id)

  const columnFilterValue = column.getFilterValue()

  return typeof firstValue === 'number' ? (
    <div className="flex space-x-2">
      <input
```

```
        type="number"
        value={(columnFilterValue as [number, number])?.[0] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            e.target.value,
            old?.[1],
          ])
        }
        placeholder={`Min`}
        className="w-24 border shadow rounded"
      />
      <input
        type="number"
        value={(columnFilterValue as [number, number])?.[1] ?? ''}
        onChange={(e) =>
          column.setFilterValue((old: [number, number]) => [
            old?.[0],
            e.target.value,
          ])
        }
        placeholder={`Max`}
        className="w-24 border shadow rounded"
      />
    </div>
  ) : (
    <input
      type="text"
      value={(columnFilterValue ?? '') as string}
      onChange={(e) => column.setFilterValue(e.target.value)}
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Query Router Search Params

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Directory Structure

- `src`
  - `api`
  - `components`
  - `hooks`
  - `routes`
  - `utils`
  - `App.tsx`
  - `index.css`
  - `main.tsx`
  - `routeTree.gen.ts`

Files

- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Example Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

import './index.css'

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Our Partners

- TanStack Query: Powerful asynchronous state management, server-state utilities and data fetching. Fetch, cache, update, and wrangle all forms of async data in your TS/JS, React, Vue, Solid, Svelte & Angular applications all without touching any "global state"
  Learn More

- TanStack DB: TanStack DB extends TanStack Query with collections, live queries, and optimistic mutations that keep your UI reactive, consistent, and blazing fast 🔥
  Learn More

Subscribe to Bytes

Your weekly dose of JavaScript news. Delivered every Monday to over 100,000 devs, for free.

No spam. Unsubscribe at *any* time.

React Example: Filters Fuzzy

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Project Files

- `src/`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

Main React Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Column,
  ColumnDef,
  ColumnFiltersState,
  FilterFn,
  SortingFn,
  flexRender,
  getCoreRowModel,
  getFilteredRowModel,
  getPaginationRowModel,
  getSortedRowModel,
  sortingFns,
  useReactTable,
} from '@tanstack/react-table'

// A TanStack fork of Kent C. Dodds' match-sorter library that provides ranking inform
import {
  RankingInfo,
  rankItem,
  compareItems,
```

```
} from '@tanstack/match-sorter-utils'

import { makeData, Person } from './makeData'

declare module '@tanstack/react-table' {
  // add fuzzy filter to the filterFns
  interface FilterFns {
    fuzzy: FilterFn<unknown>;
  }
  interface FilterMeta {
    itemRank: RankingInfo;
  }
}

// Define a custom fuzzy filter function that will apply ranking info to rows (using m
const fuzzyFilter: FilterFn<any> = (row, columnId, value, addMeta) => {
  // Rank the item
  const itemRank = rankItem(row.getValue(columnId), value)

  // Store the itemRank info
  addMeta({
    itemRank,
  })

  // Return if the item should be filtered in/out
  return itemRank.passed;
}

// Define a custom fuzzy sort function that will sort by rank if the row has ranking i
const fuzzySort: SortingFn<any> = (rowA, rowB, columnId) => {
  let dir = 0;

  // Only sort by rank if the column has ranking information
  if (rowA.columnFiltersMeta[columnId]) {
    dir = compareItems(
      rowA.columnFiltersMeta[columnId]?.itemRank!,
      rowB.columnFiltersMeta[columnId]?.itemRank!
    );
  }

  // Provide an alphanumeric fallback for when the item ranks are equal
  return dir === 0 ? sortingFns.alphanumeric(rowA, rowB, columnId) : dir;
}

function App() {
  const rerender = React.useReducer(() => ({}), {})[1];

  const [columnFilters, setColumnFilters] = React.useState<ColumnFiltersState>([]);
  const [globalFilter, setGlobalFilter] = React.useState('');

  const columns = React.useMemo<ColumnDef<Person, any>[]>(
    () => [
      {
        accessorKey: 'id',
        filterFn: 'equalsString', // exact match required
```

```
    },
    {
      accessorKey: 'firstName',
      cell: info => info.getValue(),
      filterFn: 'includesStringSensitive', // case sensitive
    },
    {
      accessorFn: row => row.lastName,
      id: 'lastName',
      cell: info => info.getValue(),
      header: () => <span>Last Name</span>,
      filterFn: 'includesString', // case insensitive
    },
    {
      accessorFn: row => `${row.firstName} ${row.lastName}`,
      id: 'fullName',
      header: 'Full Name',
      cell: info => info.getValue(),
      filterFn: 'fuzzy', // custom fuzzy filter
      // filterFn: fuzzyFilter,
      sortingFn: fuzzySort,
    },
  ],
  []
);

const [data, setData] = React.useState<Person[]>(() => makeData(5000));
const refreshData = () => setData(_old => makeData(50000));

const table = useReactTable({
  data,
  columns,
  filterFns: {
    fuzzy: fuzzyFilter,
  },
  state: {
    columnFilters,
    globalFilter,
  },
  onColumnFiltersChange: setColumnFilters,
  onGlobalFilterChange: setGlobalFilter,
  globalFilterFn: 'fuzzy',
  getCoreRowModel: getCoreRowModel(),
  getFilteredRowModel: getFilteredRowModel(),
  getSortedRowModel: getSortedRowModel(),
  getPaginationRowModel: getPaginationRowModel(),
  debugTable: true,
  debugHeaders: true,
  debugColumns: false,
});

// apply the fuzzy sort if the fullName column is being filtered
React.useEffect(() => {
  if (table.getState().columnFilters[0]?.id === 'fullName') {
    if (table.getState().sorting[0]?.id !== 'fullName') {
```

```
      table.setSorting([{ id: 'fullName', desc: false }]);
    }
  }
}, [table.getState().columnFilters[0]?.id]);

return (
  <div className="p-2">
    <div>
      <DebouncedInput
        value={globalFilter ?? ''}
        onChange={value => setGlobalFilter(String(value))}
        className="p-2 font-lg shadow border border-block"
        placeholder="Search all columns..."
      />
    </div>
    <div className="h-2" />
    <table>
      <thead>
        {table.getHeaderGroups().map(headerGroup => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map(header => {
              return (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder ? null : (
                    <>
                      <div
                        {...{
                          className: header.column.getCanSort()
                            ? 'cursor-pointer select-none'
                            : '',
                          onClick: header.column.getToggleSortingHandler(),
                        }}
                      >
                        {flexRender(
                          header.column.columnDef.header,
                          header.getContext()
                        )}
                        {{
                          asc: ' 🔼',
                          desc: ' 🔽',
                        }[header.column.getIsSorted() as string] ?? null}
                      </div>
                      {header.column.getCanFilter() ? (
                        <div>
                          <Filter column={header.column} />
                        </div>
                      ) : null}
                    </>
                  )}
                </th>
              )
            })}
          </tr>
        ))}
      </thead>
```

```
  <tbody>
    {table.getRowModel().rows.map(row => {
      return (
        <tr key={row.id}>
          {row.getVisibleCells().map(cell => {
            return (
              <td key={cell.id}>
                {flexRender(
                  cell.column.columnDef.cell,
                  cell.getContext()
                )}
              </td>
            )
          })}
        </tr>
      )
    })}
  </tbody>
</table>
<div className="h-2" />
<div className="flex items-center gap-2">
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(0)}
    disabled={!table.getCanPreviousPage()}
  >
    {'<<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.previousPage()}
    disabled={!table.getCanPreviousPage()}
  >
    {'<'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.nextPage()}
    disabled={!table.getCanNextPage()}
  >
    {'>'}
  </button>
  <button
    className="border rounded p-1"
    onClick={() => table.setPageIndex(table.getPageCount() - 1)}
    disabled={!table.getCanNextPage()}
  >
    {'>>'}
  </button>
  <span className="flex items-center gap-1">
    <div>Page</div>
    <strong>
      {table.getState().pagination.pageIndex + 1} of{' '}
      {table.getPageCount()}
    </strong>
```

```
        </span>
        <span className="flex items-center gap-1">
          | Go to page:
          <input
            type="number"
            defaultValue={table.getState().pagination.pageIndex + 1}
            onChange={e => {
              const page = e.target.value ? Number(e.target.value) - 1 : 0;
              table.setPageIndex(page);
            }}
            className="border p-1 rounded w-16"
          />
        </span>
        <select
          value={table.getState().pagination.pageSize}
          onChange={e => {
            table.setPageSize(Number(e.target.value));
          }}
        >
          {[10, 20, 30, 40, 50].map(pageSize => (
            <option key={pageSize} value={pageSize}>
              Show {pageSize}
            </option>
          ))}
        </select>
      </div>
      <div>{table.getPrePaginationRowModel().rows.length} Rows</div>
      <div>
        <button onClick={() => rerender()}>Force Rerender</button>
      </div>
      <div>
        <button onClick={() => refreshData()}>Refresh Data</button>
      </div>
      <pre>
        {JSON.stringify(
          {
            columnFilters: table.getState().columnFilters,
            globalFilter: table.getState().globalFilter,
          },
          null,
          2
        )}
      </pre>
    </div>
  );
}

function Filter({ column }: { column: Column<any, unknown> }) {
  const columnFilterValue = column.getFilterValue();

  return (
    <DebouncedInput
      type="text"
      value={(columnFilterValue ?? '') as string}
      onChange={value => column.setFilterValue(value)}
```

```
      placeholder={`Search...`}
      className="w-36 border shadow rounded"
    />
  );
}

// A typical debounced input react component
function DebouncedInput({
  value: initialValue,
  onChange,
  debounce = 500,
  ...props
}: {
  value: string | number;
  onChange: (value: string | number) => void;
  debounce?: number;
} & Omit<React.InputHTMLAttributes<HTMLInputElement>, 'onChange'>) {
  const [value, setValue] = React.useState(initialValue);

  React.useEffect(() => {
    setValue(initialValue);
  }, [initialValue]);

  React.useEffect(() => {
    const timeout = setTimeout(() => {
      onChange(value);
    }, debounce);

    return () => clearTimeout(timeout);
  }, [value]);

  return (
    <input {...props} value={value} onChange={e => setValue(e.target.value)} />
  );
}

const rootElement = document.getElementById('root');
if (!rootElement) throw new Error('Failed to find the root element');

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

React Example: Column Ordering

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

Files:

- `src`
  - `index.css`
  - `main.tsx`
  - `makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { faker } from '@faker-js/faker'

import './index.css'

import {
  ColumnDef,
  ColumnOrderState,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        footer: (props) => props.column.id,
      },
      {
        accessorFn: (row) => row.lastName,
        id: 'lastName',
```

```
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const [data, setData] = React.useState(() => makeData(20))
  const [columns] = React.useState(() => [...defaultColumns])

  const [columnVisibility, setColumnVisibility] = React.useState({})
  const [columnOrder, setColumnOrder] = React.useState<ColumnOrderState>([])

  const rerender = () => setData(() => makeData(20))

  const table = useReactTable({
    data,
    columns,
    state: {
      columnVisibility,
      columnOrder,
    },
```

```
  onColumnVisibilityChange: setColumnVisibility,
  onColumnOrderChange: setColumnOrder,
  getCoreRowModel: getCoreRowModel(),
  debugTable: true,
  debugHeaders: true,
  debugColumns: true,
})

const randomizeColumns = () => {
  table.setColumnOrder(
    faker.helpers.shuffle(table.getAllLeafColumns().map(d => d.id))
  )
}

return (
  <div className="p-2">
    <div className="inline-block border border-black shadow rounded">
      <div className="px-1 border-b border-black">
        <label>
          <input
            {...{
              type: 'checkbox',
              checked: table.getIsAllColumnsVisible(),
              onChange: table.getToggleAllColumnsVisibilityHandler(),
            }}
          />{' '}
          Toggle All
        </label>
      </div>
      {table.getAllLeafColumns().map(column => {
        return (
          <div key={column.id} className="px-1">
            <label>
              <input
                {...{
                  type: 'checkbox',
                  checked: column.getIsVisible(),
                  onChange: column.getToggleVisibilityHandler(),
                }}
              />{' '}
              {column.id}
            </label>
          </div>
        )
      })}
    </div>
    <div className="h-4" />
    <div className="flex flex-wrap gap-2">
      <button onClick={() => rerender()} className="border p-1">
        Regenerate
      </button>
      <button onClick={() => randomizeColumns()} className="border p-1">
        Shuffle Columns
      </button>
    </div>
```

```
      <div className="h-4" />
      <table>
        <thead>
          {table.getHeaderGroups().map(headerGroup => (
            <tr key={headerGroup.id}>
              {headerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </thead>
        <tbody>
          {table.getRowModel().rows.map(row => (
            <tr key={row.id}>
              {row.getVisibleCells().map(cell => (
                <td key={cell.id}>
                  {flexRender(cell.column.columnDef.cell, cell.getContext())}
                </td>
              ))}
            </tr>
          ))}
        </tbody>
        <tfoot>
          {table.getFooterGroups().map(footerGroup => (
            <tr key={footerGroup.id}>
              {footerGroup.headers.map(header => (
                <th key={header.id} colSpan={header.colSpan}>
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.footer,
                        header.getContext()
                      )}
                </th>
              ))}
            </tr>
          ))}
        </tfoot>
      </table>
      <pre>{JSON.stringify(table.getState().columnOrder, null, 2)}</pre>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
```

```
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Column Dnd

Github | StackBlitz | CodeSandbox

Code Explorer | Interactive Sandbox

Files

- src
  - index.css
  - main.tsx
  - makeData.ts
- .gitignore
- README.md
- index.html
- package.json
- tsconfig.json
- vite.config.js

Implementation

```
import React, { CSSProperties } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Cell,
  ColumnDef,
  Header,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

// needed for table body level scope DnD setup
import {
  DndContext,
  KeyboardSensor,
  MouseSensor,
  TouchSensor,
  closestCenter,
  type DragEndEvent,
  useSensor,
  useSensors,
} from '@dnd-kit/core'
import { restrictToHorizontalAxis } from '@dnd-kit/modifiers'
import {
  arrayMove,
```

```
  SortableContext,
  horizontalListSortingStrategy,
} from '@dnd-kit/sortable'

// needed for row & cell level scope DnD setup
import { useSortable } from '@dnd-kit/sortable'
import { CSS } from '@dnd-kit/utilities'

const DraggableTableHeader = ({
  header,
}: {
  header: Header<Person, unknown>
}) => {
  const { attributes, isDragging, listeners, setNodeRef, transform } =
    useSortable({
      id: header.column.id,
    })

  const style: CSSProperties = {
    opacity: isDragging ? 0.8 : 1,
    position: 'relative',
    transform: CSS.Translate.toString(transform), // translate instead of transform to
    transition: 'width transform 0.2s ease-in-out',
    whiteSpace: 'nowrap',
    width: header.column.getSize(),
    zIndex: isDragging ? 1 : 0,
  }

  return (
    <th colSpan={header.colSpan} ref={setNodeRef} style={style}>
      {header.isPlaceholder
        ? null
        : flexRender(header.column.columnDef.header, header.getContext())}
      <button {...attributes} {...listeners}>🟰</button>
    </th>
  )
}

const DragAlongCell = ({ cell }: { cell: Cell<Person, unknown> }) => {
  const { isDragging, setNodeRef, transform } = useSortable({
    id: cell.column.id,
  })

  const style: CSSProperties = {
    opacity: isDragging ? 0.8 : 1,
    position: 'relative',
    transform: CSS.Translate.toString(transform), // translate instead of transform to
    transition: 'width transform 0.2s ease-in-out',
    width: cell.column.getSize(),
    zIndex: isDragging ? 1 : 0,
  }

  return (
    <td style={style} ref={setNodeRef}>
      {flexRender(cell.column.columnDef.cell, cell.getContext())}
```

```
    </td>
  )
}

function App() {
  const columns = React.useMemo<ColumnDef<Person>[]>(
    () => [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        id: 'firstName',
        size: 150,
      },
      {
        accessorFn: (row) => row.lastName,
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        id: 'lastName',
        size: 150,
      },
      {
        accessorKey: 'age',
        header: () => 'Age',
        id: 'age',
        size: 120,
      },
      {
        accessorKey: 'visits',
        header: () => <span>Visits</span>,
        id: 'visits',
        size: 120,
      },
      {
        accessorKey: 'status',
        header: 'Status',
        id: 'status',
        size: 150,
      },
      {
        accessorKey: 'progress',
        header: 'Profile Progress',
        id: 'progress',
        size: 180,
      },
    ],
    []
  )

  const [data, setData] = React.useState(() => makeData(20))
  const [columnOrder, setColumnOrder] = React.useState<string[]>(
    () => columns.map((c) => c.id!)
  )

  const rerender = () => setData(() => makeData(20))
```

```
const table = useReactTable({
  data,
  columns,
  getCoreRowModel: getCoreRowModel(),
  state: {
    columnOrder,
  },
  onColumnOrderChange: setColumnOrder,
  debugTable: true,
  debugHeaders: true,
  debugColumns: true,
})

// reorder columns after drag & drop
function handleDragEnd(event: DragEndEvent) {
  const { active, over } = event
  if (active && over && active.id !== over.id) {
    setColumnOrder((columnOrder) => {
      const oldIndex = columnOrder.indexOf(active.id as string)
      const newIndex = columnOrder.indexOf(over.id as string)
      return arrayMove(columnOrder, oldIndex, newIndex) // this is just a splice uti
    })
  }
}

const sensors = useSensors(
  useSensor(MouseSensor, {}),
  useSensor(TouchSensor, {}),
  useSensor(KeyboardSensor, {})
)

return (
  // NOTE: This provider creates div elements, so don't nest inside of <table> eleme
  <DndContext
    collisionDetection={closestCenter}
    modifiers={[restrictToHorizontalAxis]}
    onDragEnd={handleDragEnd}
    sensors={sensors}
  >
    <div className="p-2">
      <div className="h-4" />
      <div className="flex flex-wrap gap-2">
        <button onClick={() => rerender()} className="border p-1">
          Regenerate
        </button>
      </div>
      <div className="h-4" />
      <table>
        <thead>
          {table.getHeaderGroups().map((headerGroup) => (
            <tr key={headerGroup.id}>
              <SortableContext items={columnOrder} strategy={horizontalListSortingSt
                {headerGroup.headers.map((header) => (
                  <DraggableTableHeader key={header.id} header={header} />
                ))}
```

```
                </SortableContext>
              </tr>
            ))}
          </thead>
          <tbody>
            {table.getRowModel().rows.map((row) => (
              <tr key={row.id}>
                {row.getVisibleCells().map((cell) => (
                  <SortableContext key={cell.id} items={columnOrder} strategy={horizon
                    <DragAlongCell key={cell.id} cell={cell} />
                  </SortableContext>
                ))}
              </tr>
            ))}
          </tbody>
        </table>
        <pre>{JSON.stringify(data, null, 2)}</pre>
      </div>
    </DndContext>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Column Pinning

Github
StackBlitz
CodeSandbox

Code Explorer | Code | Interactive Sandbox | Sandbox

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { faker } from '@faker-js/faker'

import './index.css'

import {
  ColumnDef,
  ColumnOrderState,
  flexRender,
  getCoreRowModel,
  useReactTable,
  VisibilityState,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'

const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        footer: (props) => props.column.id,
      },
      {
        accessorFn: (row) => row.lastName,
        id: 'lastName',
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
```

```
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
];

function App() {
  const [data, setData] = React.useState(() => makeData(5000));
  const [columns] = React.useState(() => [...defaultColumns]);

  const [columnVisibility, setColumnVisibility] = React.useState<VisibilityState>({});
  const [columnOrder, setColumnOrder] = React.useState<ColumnOrderState>([]);
  const [columnPinning, setColumnPinning] = React.useState({});

  const [isSplit, setIsSplit] = React.useState(false);
  const rerender = () => setData(() => makeData(5000));

  const table = useReactTable({
    data,
    columns,
    state: {
      columnVisibility,
      columnOrder,
      columnPinning,
    },
    onColumnVisibilityChange: setColumnVisibility,
    onColumnOrderChange: setColumnOrder,
    onColumnPinningChange: setColumnPinning,
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
  });

  const randomizeColumns = () => {
```

```
  table.setColumnOrder(
    faker.helpers.shuffle(table.getAllLeafColumns().map((d) => d.id))
  );
};

return (
  <div className="p-2">
    <div className="inline-block border border-black shadow rounded">
      <div className="px-1 border-b border-black">
        <label>
          <input
            {...{
              type: 'checkbox',
              checked: table.getIsAllColumnsVisible(),
              onChange: table.getToggleAllColumnsVisibilityHandler(),
            }}
          />{' '}
          Toggle All
        </label>
      </div>
      {table.getAllLeafColumns().map((column) => {
        return (
          <div key={column.id} className="px-1">
            <label>
              <input
                {...{
                  type: 'checkbox',
                  checked: column.getIsVisible(),
                  onChange: column.getToggleVisibilityHandler(),
                }}
              />{' '}
              {column.id}
            </label>
          </div>
        );
      })}
    </div>
    <div className="h-4" />
    <div className="flex flex-wrap gap-2">
      <button onClick={rerender} className="border p-1">
        Regenerate
      </button>
      <button onClick={randomizeColumns} className="border p-1">
        Shuffle Columns
      </button>
    </div>
    <div className="h-4" />
    <div>
      <label>
        <input
          type="checkbox"
          checked={isSplit}
          onChange={(e) => setIsSplit(e.target.checked)}
        />{' '}
        Split Mode
```

```
  </label>
</div>
<div className={`flex ${isSplit ? 'gap-4' : ''}`}>
  {isSplit ? (
    <table className="border-2 border-black">
      <thead>
        {table.getLeftHeaderGroups().map((headerGroup) => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map((header) => (
              <th key={header.id} colSpan={header.colSpan}>
                <div className="whitespace-nowrap">
                  {header.isPlaceholder
                    ? null
                    : flexRender(header.column.columnDef.header, header.getConte
                </div>
                {!header.isPlaceholder && header.column.getCanPin() && (
                  <div className="flex gap-1 justify-center">
                    {header.column.getIsPinned() !== 'left' && (
                      <button
                        className="border rounded px-2"
                        onClick={() => {
                          header.column.pin('left');
                        }}
                      >
                        {'<='}
                      </button>
                    )}
                    {header.column.getIsPinned() && (
                      <button
                        className="border rounded px-2"
                        onClick={() => {
                          header.column.pin(false);
                        }}
                      >
                        X
                      </button>
                    )}
                    {header.column.getIsPinned() !== 'right' && (
                      <button
                        className="border rounded px-2"
                        onClick={() => {
                          header.column.pin('right');
                        }}
                      >
                        {'=>'}
                      </button>
                    )}
                  </div>
                )}
              </th>
            ))}
          </tr>
        ))}
      </thead>
      <tbody>
```

```
      {table
        .getRowModel()
        .rows.slice(0, 20)
        .map((row) => (
          <tr key={row.id}>
            {row.getLeftVisibleCells().map((cell) => (
              <td key={cell.id}>
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </td>
            ))}
          </tr>
        ))}
    </tbody>
  </table>
) : null}
<table className="border-2 border-black">
  <thead>
    {(isSplit
      ? table.getCenterHeaderGroups()
      : table.getHeaderGroups()
    ).map((headerGroup) => (
      <tr key={headerGroup.id}>
        {headerGroup.headers.map((header) => (
          <th key={header.id} colSpan={header.colSpan}>
            <div className="whitespace-nowrap">
              {header.isPlaceholder
                ? null
                : flexRender(header.column.columnDef.header, header.getContext
            </div>
            {!header.isPlaceholder && header.column.getCanPin() && (
              <div className="flex gap-1 justify-center">
                {header.column.getIsPinned() !== 'left' && (
                  <button
                    className="border rounded px-2"
                    onClick={() => {
                      header.column.pin('left');
                    }}
                  >
                    {'<='}
                  </button>
                )}
                {header.column.getIsPinned() && (
                  <button
                    className="border rounded px-2"
                    onClick={() => {
                      header.column.pin(false);
                    }}
                  >
                    X
                  </button>
                )}
                {header.column.getIsPinned() !== 'right' && (
                  <button
                    className="border rounded px-2"
                    onClick={() => {
```

```
                      header.column.pin('right');
                    }}
                  >
                    {'=>'}
                  </button>
                )}
              </div>
            )}
          </th>
        ))}
      </tr>
    ))}
  </thead>
  <tbody>
    {table
      .getRowModel()
      .rows.slice(0, 20)
      .map((row) => (
        <tr key={row.id}>
          {(isSplit
            ? row.getCenterVisibleCells()
            : row.getVisibleCells()
          ).map((cell) => (
            <td key={cell.id}>
              {flexRender(cell.column.columnDef.cell, cell.getContext())}
            </td>
          ))}
        </tr>
      ))}
  </tbody>
</table>
{isSplit ? (
  <table className="border-2 border-black">
    <thead>
      {table.getRightHeaderGroups().map((headerGroup) => (
        <tr key={headerGroup.id}>
          {headerGroup.headers.map((header) => (
            <th key={header.id} colSpan={header.colSpan}>
              <div className="whitespace-nowrap">
                {header.isPlaceholder
                  ? null
                  : flexRender(header.column.columnDef.header, header.getConte
              </div>
              {!header.isPlaceholder && header.column.getCanPin() && (
                <div className="flex gap-1 justify-center">
                  {header.column.getIsPinned() !== 'left' && (
                    <button
                      className="border rounded px-2"
                      onClick={() => {
                        header.column.pin('left');
                      }}
                    >
                      {'<='}
                    </button>
                  )}
```

```
                        {header.column.getIsPinned() && (
                          <button
                            className="border rounded px-2"
                            onClick={() => {
                              header.column.pin(false);
                            }}
                          >
                            X
                          </button>
                        )}
                        {header.column.getIsPinned() !== 'right' && (
                          <button
                            className="border rounded px-2"
                            onClick={() => {
                              header.column.pin('right');
                            }}
                          >
                            {'=>'}
                          </button>
                        )}
                      </div>
                    )}
                  </th>
                ))}
              </tr>
            ))}
          </thead>
          <tbody>
            {table
              .getRowModel()
              .rows.slice(0, 20)
              .map((row) => (
                <tr key={row.id}>
                  {row.getRightVisibleCells().map((cell) => (
                    <td key={cell.id}>
                      {flexRender(cell.column.columnDef.cell, cell.getContext())}
                    </td>
                  ))}
                </tr>
              ))}
          </tbody>
        </table>
      ) : null}
    </div>
    <pre>{JSON.stringify(table.getState().columnPinning, null, 2)}</pre>
  </div>
);
}

const rootElement = document.getElementById('root');
if (!rootElement) throw new Error('Failed to find the root element');

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
```

```
  </React.StrictMode>
);
```

React Example: Column Pinning Sticky

Github | StackBlitz | CodeSandbox

Code Explorer | Code

Interactive Sandbox | Sandbox

Files:

- `src/index.css`
- `src/main.tsx`
- `src/makeData.ts`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

##tsx

```
import React, { CSSProperties } from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  Column,
  ColumnDef,
  flexRender,
  getCoreRowModel,
  useReactTable,
} from '@tanstack/react-table'
import { makeData, Person } from './makeData'
import { faker } from '@faker-js/faker'

// These are the important styles to make sticky column pinning work!
// Apply styles like this using your CSS strategy of choice with this kind of logic to
// View the index.css file for more needed styles such as border-collapse: separate
const getCommonPinningStyles = (column: Column<Person>): CSSProperties => {
  const isPinned = column.getIsPinned()
  const isLastLeftPinnedColumn =
    isPinned === 'left' && column.getIsLastColumn('left')
  const isFirstRightPinnedColumn =
    isPinned === 'right' && column.getIsFirstColumn('right')
```

```
  return {
    boxShadow: isLastLeftPinnedColumn
      ? '-4px 0 4px -4px gray inset'
      : isFirstRightPinnedColumn
        ? '4px 0 4px -4px gray inset'
        : undefined,
    left: isPinned === 'left' ? `${column.getStart('left')}px` : undefined,
    right: isPinned === 'right' ? `${column.getAfter('right')}px` : undefined,
    opacity: isPinned ? 0.95 : 1,
    position: isPinned ? 'sticky' : 'relative',
    width: column.getSize(),
    zIndex: isPinned ? 1 : 0,
  }
}

const defaultColumns: ColumnDef<Person>[] = [
  {
    accessorKey: 'firstName',
    id: 'firstName',
    header: 'First Name',
    cell: info => info.getValue(),
    footer: props => props.column.id,
    size: 180,
  },
  {
    accessorFn: row => row.lastName,
    id: 'lastName',
    cell: info => info.getValue(),
    header: () => <span>Last Name</span>,
    footer: props => props.column.id,
    size: 180,
  },
  {
    accessorKey: 'age',
    id: 'age',
    header: 'Age',
    footer: props => props.column.id,
    size: 180,
  },
  {
    accessorKey: 'visits',
    id: 'visits',
    header: 'Visits',
    footer: props => props.column.id,
    size: 180,
  },
  {
    accessorKey: 'status',
    id: 'status',
    header: 'Status',
    footer: props => props.column.id,
    size: 180,
  },
  {
    accessorKey: 'progress',
```

```
    id: 'progress',
    header: 'Profile Progress',
    footer: props => props.column.id,
    size: 180,
  },
]

function App() {
  const [data, setData] = React.useState(() => makeData(30))
  const [columns] = React.useState(() => [...defaultColumns])

  const rerender = () => setData(() => makeData(30))

  const table = useReactTable({
    data,
    columns,
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
    columnResizeMode: 'onChange',
  })

  const randomizeColumns = () => {
    table.setColumnOrder(
      faker.helpers.shuffle(table.getAllLeafColumns().map(d => d.id))
    )
  }

  return (
    <div className="p-2">
      <div className="inline-block border border-black shadow rounded">
        <div className="px-1 border-b border-black">
          <label>
            <input
              {...{
                type: 'checkbox',
                checked: table.getIsAllColumnsVisible(),
                onChange: table.getToggleAllColumnsVisibilityHandler(),
              }}
            />{' '}
            Toggle All
          </label>
        </div>
        {table.getAllLeafColumns().map(column => {
          return (
            <div key={column.id} className="px-1">
              <label>
                <input
                  {...{
                    type: 'checkbox',
                    checked: column.getIsVisible(),
                    onChange: column.getToggleVisibilityHandler(),
                  }}
                />{' '}
```

```
          {column.id}
        </label>
      </div>
    )
  })}
</div>
<div className="h-4" />
<div className="flex flex-wrap gap-2">
  <button onClick={() => rerender()} className="border p-1">
    Regenerate
  </button>
  <button onClick={() => randomizeColumns()} className="border p-1">
    Shuffle Columns
  </button>
</div>
<div className="h-4" />
<div className="table-container">
  <table
    style={{
      width: table.getTotalSize(),
    }}
  >
    <thead>
      {table.getHeaderGroups().map(headerGroup => (
        <tr key={headerGroup.id}>
          {headerGroup.headers.map(header => {
            const { column } = header

            return (
              <th
                key={header.id}
                colSpan={header.colSpan}
                //IMPORTANT: This is where the magic happens!
                style={{ ...getCommonPinningStyles(column) }}
              >
                <div className="whitespace-nowrap">
                  {header.isPlaceholder
                    ? null
                    : flexRender(
                        header.column.columnDef.header,
                        header.getContext()
                      )}{' '}
                  {/* Demo getIndex behavior */}
                  {column.getIndex(column.getIsPinned() || 'center')}
                </div>
                {!header.isPlaceholder && header.column.getCanPin() && (
                  <div className="flex gap-1 justify-center">
                    {header.column.getIsPinned() !== 'left' ? (
                      <button
                        className="border rounded px-2"
                        onClick={() => {
                          header.column.pin('left')
                        }}
                      >
                        {'<='}
```

```
                  </button>
                ) : null}
                {header.column.getIsPinned() ? (
                  <button
                    className="border rounded px-2"
                    onClick={() => {
                      header.column.pin(false)
                    }}
                  >
                    X
                  </button>
                ) : null}
                {header.column.getIsPinned() !== 'right' ? (
                  <button
                    className="border rounded px-2"
                    onClick={() => {
                      header.column.pin('right')
                    }}
                  >
                    {'=>'}
                  </button>
                ) : null}
              </div>
            )}
            <div
              {...{
                onDoubleClick: () => header.column.resetSize(),
                onMouseDown: header.getResizeHandler(),
                onTouchStart: header.getResizeHandler(),
                className: `resizer ${
                  header.column.getIsResizing() ? 'isResizing' : ''
                }`,
              }}
            />
          </th>
        )
      })}
    </tr>
  ))}
</thead>
<tbody>
  {table.getRowModel().rows.map(row => (
    <tr key={row.id}>
      {row.getVisibleCells().map(cell => {
        const { column } = cell
        return (
          <td
            key={cell.id}
            //IMPORTANT: This is where the magic happens!
            style={{ ...getCommonPinningStyles(column) }}
          >
            {flexRender(
              cell.column.columnDef.cell,
              cell.getContext()
            )}
```

```
                    </td>
                  )
                })}
              </tr>
            ))}
          </tbody>
        </table>
      </div>
      <pre>{JSON.stringify(table.getState().columnPinning, null, 2)}</pre>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

React Example: Column Sizing

Github | StackBlitz | CodeSandbox

Code Explorer | Code | Interactive Sandbox

Files

- `src`
  - `index.css`
  - `main.tsx`
- `.gitignore`
- `README.md`
- `index.html`
- `package.json`
- `tsconfig.json`
- `vite.config.js`

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import './index.css'

import {
  useReactTable,
  ColumnResizeMode,
  getCoreRowModel,
  ColumnDef,
  flexRender,
  ColumnResizeDirection,
} from '@tanstack/react-table'

type Person = {
  firstName: string
  lastName: string
  age: number
  visits: number
  status: string
  progress: number
}

const defaultData: Person[] = [
  {
    firstName: 'tanner',
    lastName: 'linsley',
    age: 24,
    visits: 100,
    status: 'In Relationship',
    progress: 50,
  },
  {
    firstName: 'tandy',
```

```
    lastName: 'miller',
    age: 40,
    visits: 40,
    status: 'Single',
    progress: 80,
  },
  {
    firstName: 'joe',
    lastName: 'dirte',
    age: 45,
    visits: 20,
    status: 'Complicated',
    progress: 10,
  },
]

const defaultColumns: ColumnDef<Person>[] = [
  {
    header: 'Name',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'firstName',
        cell: (info) => info.getValue(),
        footer: (props) => props.column.id,
      },
      {
        accessorFn: (row) => row.lastName,
        id: 'lastName',
        cell: (info) => info.getValue(),
        header: () => <span>Last Name</span>,
        footer: (props) => props.column.id,
      },
    ],
  },
  {
    header: 'Info',
    footer: (props) => props.column.id,
    columns: [
      {
        accessorKey: 'age',
        header: () => 'Age',
        footer: (props) => props.column.id,
      },
      {
        header: 'More Info',
        columns: [
          {
            accessorKey: 'visits',
            header: () => <span>Visits</span>,
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'status',
            header: 'Status',
```

```
            footer: (props) => props.column.id,
          },
          {
            accessorKey: 'progress',
            header: 'Profile Progress',
            footer: (props) => props.column.id,
          },
        ],
      },
    ],
  },
]

function App() {
  const [data] = React.useState(() => [...defaultData])
  const [columns] = React.useState<typeof defaultColumns>(() => [
    ...defaultColumns,
  ])

  const [columnResizeMode, setColumnResizeMode] =
    React.useState<ColumnResizeMode>('onChange')

  const [columnResizeDirection, setColumnResizeDirection] =
    React.useState<ColumnResizeDirection>('ltr')

  const rerender = React.useReducer(() => ({}), {})[1]

  const table = useReactTable({
    data,
    columns,
    columnResizeMode,
    columnResizeDirection,
    getCoreRowModel: getCoreRowModel(),
    debugTable: true,
    debugHeaders: true,
    debugColumns: true,
  })

  return (
    <div className="p-2">
      <select
        value={columnResizeMode}
        onChange={(e) => setColumnResizeMode(e.target.value as ColumnResizeMode)}
        className="border p-2 border-black rounded"
      >
        <option value="onEnd">Resize: "onEnd"</option>
        <option value="onChange">Resize: "onChange"</option>
      </select>
      <select
        value={columnResizeDirection}
        onChange={(e) =>
          setColumnResizeDirection(e.target.value as ColumnResizeDirection)
        }
        className="border p-2 border-black rounded"
      >
```

```
  <option value="ltr">Resize Direction: "ltr"</option>
  <option value="rtl">Resize Direction: "rtl"</option>
</select>
<div style={{ direction: table.options.columnResizeDirection }}>
  <div className="h-4" />
  <div className="text-xl">&lt;table/&gt;</div>
  <div className="overflow-x-auto">
    <table
      {...{
        style: {
          width: table.getCenterTotalSize(),
        },
      }}
    >
      <thead>
        {table.getHeaderGroups().map((headerGroup) => (
          <tr key={headerGroup.id}>
            {headerGroup.headers.map((header) => (
              <th
                {...{
                  key: header.id,
                  colSpan: header.colSpan,
                  style: {
                    width: header.getSize(),
                  },
                }}
              >
                {header.isPlaceholder
                  ? null
                  : flexRender(
                      header.column.columnDef.header,
                      header.getContext()
                    )}
                <div
                  {...{
                    onDoubleClick: () => header.column.resetSize(),
                    onMouseDown: header.getResizeHandler(),
                    onTouchStart: header.getResizeHandler(),
                    className: `resizer ${
                      table.options.columnResizeDirection
                    } ${
                      header.column.getIsResizing() ? 'isResizing' : ''
                    }`,
                    style: {
                      transform:
                        columnResizeMode === 'onEnd' &&
                        header.column.getIsResizing()
                          ? `translateX(${
                              (table.options.columnResizeDirection ===
                              'rtl'
                                ? -1
                                : 1) *
                              (table.getState().columnSizingInfo
                                .deltaOffset ?? 0)
                            }px)`
```

```
                        : '',
                    },
                  }}
                />
              </th>
            ))}
          </tr>
        ))}
      </thead>
      <tbody>
        {table.getRowModel().rows.map((row) => (
          <tr key={row.id}>
            {row.getVisibleCells().map((cell) => (
              <td
                {...{
                  key: cell.id,
                  style: {
                    width: cell.column.getSize(),
                  },
                }}
              >
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  </div>
  <div className="h-4" />
  <div className="text-xl">&lt;div/&gt; (relative)</div>
  <div className="overflow-x-auto">
    <div
      {...{
        className: 'divTable',
        style: {
          width: table.getTotalSize(),
        },
      }}
    >
      <div className="thead">
        {table.getHeaderGroups().map((headerGroup) => (
          <div
            {...{
              key: headerGroup.id,
              className: 'tr',
            }}
          >
            {headerGroup.headers.map((header) => (
              <div
                {...{
                  key: header.id,
                  className: 'th',
                  style: {
                    width: header.getSize(),
```

```
            },
          }}
        >
          {header.isPlaceholder
            ? null
            : flexRender(header.column.columnDef.header, header.getContext
          <div
            {...{
              onDoubleClick: () => header.column.resetSize(),
              onMouseDown: header.getResizeHandler(),
              onTouchStart: header.getResizeHandler(),
              className: `resizer ${
                table.options.columnResizeDirection
              } ${
                header.column.getIsResizing() ? 'isResizing' : ''
              }`,
              style: {
                transform:
                  columnResizeMode === 'onEnd' &&
                  header.column.getIsResizing()
                    ? `translateX(${
                        (table.options.columnResizeDirection ===
                        'rtl'
                          ? -1
                          : 1) *
                        (table.getState().columnSizingInfo
                          .deltaOffset ?? 0)
                      }px)`
                    : '',
              },
            }}
          />
        </div>
      ))}
    </div>
  ))}
</div>
<div {...{ className: 'tbody' }}>
  {table.getRowModel().rows.map((row) => (
    <div
      {...{
        key: row.id,
        className: 'tr',
      }}
    >
      {row.getVisibleCells().map((cell) => (
        <div
          {...{
            key: cell.id,
            className: 'td',
            style: {
              width: cell.column.getSize(),
            },
          }}
        >
```

```
              {flexRender(cell.column.columnDef.cell, cell.getContext())}
            </div>
          ))}
        </div>
      ))}
    </div>
  </div>
</div>
<div className="h-4" />
<div className="text-xl">&lt;div/&gt; (absolute positioning)</div>
<div className="overflow-x-auto">
  <div
    {...{
      className: 'divTable',
      style: {
        width: table.getTotalSize(),
      },
    }}
  >
    <div className="thead">
      {table.getHeaderGroups().map((headerGroup) => (
        <div
          {...{
            key: headerGroup.id,
            className: 'tr',
            style: {
              position: 'relative',
            },
          }}
        >
          {headerGroup.headers.map((header) => (
            <div
              {...{
                key: header.id,
                className: 'th',
                style: {
                  position: 'absolute',
                  left: header.getStart(),
                  width: header.getSize(),
                },
              }}
            >
              {header.isPlaceholder
                ? null
                : flexRender(header.column.columnDef.header, header.getContext
              <div
                {...{
                  onDoubleClick: () => header.column.resetSize(),
                  onMouseDown: header.getResizeHandler(),
                  onTouchStart: header.getResizeHandler(),
                  className: `resizer ${
                    table.options.columnResizeDirection
                  } ${
                    header.column.getIsResizing() ? 'isResizing' : ''
                  }`,
```

```
                    style: {
                      transform:
                        columnResizeMode === 'onEnd' &&
                        header.column.getIsResizing()
                          ? `translateX(${
                              (table.options.columnResizeDirection ===
                              'rtl'
                                ? -1
                                : 1) *
                              (table.getState().columnSizingInfo
                                .deltaOffset ?? 0)
                            }px)`
                          : '',
                    },
                  }}
                />
              </div>
            ))}
          </div>
        ))}
      </div>
      <div {...{ className: 'tbody' }}>
        {table.getRowModel().rows.map((row) => (
          <div
            {...{
              key: row.id,
              className: 'tr',
              style: {
                position: 'relative',
              },
            }}
          >
            {row.getVisibleCells().map((cell) => (
              <div
                {...{
                  key: cell.id,
                  className: 'td',
                  style: {
                    position: 'absolute',
                    left: cell.column.getStart(),
                    width: cell.column.getSize(),
                  },
                }}
              >
                {flexRender(cell.column.columnDef.cell, cell.getContext())}
              </div>
            ))}
          </div>
        ))}
      </div>
    </div>
  </div>
</div>
<div className="h-4" />
<button onClick={() => rerender()} className="border p-2">
```

```
        Rerender
      </button>
      <pre>
        {JSON.stringify(
          {
            columnSizing: table.getState().columnSizing,
            columnSizingInfo: table.getState().columnSizingInfo,
          },
          null,
          2
        )}
      </pre>
    </div>
  )
}

const rootElement = document.getElementById('root')
if (!rootElement) throw new Error('Failed to find the root element')

ReactDOM.createRoot(rootElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```