BUILD A REACT DATATABLE FROM SCRATCH

A step-by-step engineering guide to state, derived data, accessibility, performance, trees, server-side processing, and headless UI

Name	Age	Qualification	Rating
Priya Patel	28	B.Tech	★★★★★
Aarav Sharma	34	M.Sc	★★★★☆
Meera Pillai	31	MBA	★★★★★
Rohan Mehta	26	B.Com	★★★☆☆
Kavya Menon	29	M.Tech	★★★★☆

`rows → filter → sort → flatten → slice → render`

The complete mental model begins with a simple pipeline.

Rajesh Pillai • Unlearning Labs

Companion ebook to github.com/rajeshpillai/react-datatable-from-scratch
Draft edition • September 2026

About this book

This book builds one React DataTable eleven times. We begin with a semantic table driven by a columns array. Each chapter adds one engineering problem: sorting, filtering, pagination, identity, selection, performance, column movement, editing, tree data, server-side processing, and finally a headless model that can drive more than one visual view.

The important part is not the final component. The important part is the reasoning that lets you predict where a feature belongs, what state it needs, and what must remain derived. If you understand that, you can rebuild the component without memorising it.

The rule for this entire book
Simplify the language, not the engineering. Every chapter starts with a working checkpoint. You change one idea, run it, test it, and only then move to the next chapter.

What you will build

- A configuration-driven DataTable where column definitions are data rather than repeated JSX.
- Accessible sorting, global search, per-column search, pagination, row selection and live status updates.
- Stable row identity that survives sorting, filtering, paging, edits and tree expansion.
- Measured performance improvements instead of speculative memoisation.
- Accessible column reordering with both drag-and-drop and button controls.
- Inline editing with a clear ownership boundary: the parent owns committed data; the table owns drafts.
- A tree table that filters and sorts hierarchically, then flattens visible branches for <tbody>.
- A server-side mode for large datasets using the same conceptual pipeline across a network boundary.
- A headless useDataTable model that can drive both a semantic table and a card list.

How to use the checkpoints

The reference repository contains steps/step-01 through steps/step-11. Each folder is the exact working code at the end of that lesson. In this book, the checkpoint box tells you what should be true before you continue. If your code differs, compare behaviour first and source code second.

Run the application

```
cd app
npm install
npm run dev

# For lesson 10, run the API in a second terminal:
node server/server.mjs
```

Reference environment
The repository is built and tested with Node 24, React 19.2, and Vite 8.2. Exact dependency versions are locked in app/package-lock.json.

The one mental model to keep

The DataTable pipeline

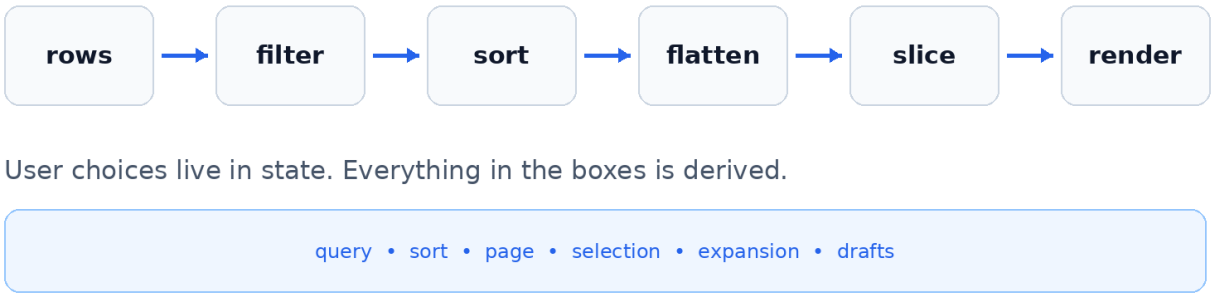

Figure 1. Every feature becomes a stage in the same data pipeline.

Start with rows. Apply the transformations that change which rows exist or what order they appear in. Only after that do you decide which rows are visible on the current page. Rendering is the final step, not the place where data logic should hide.

State vs derived data

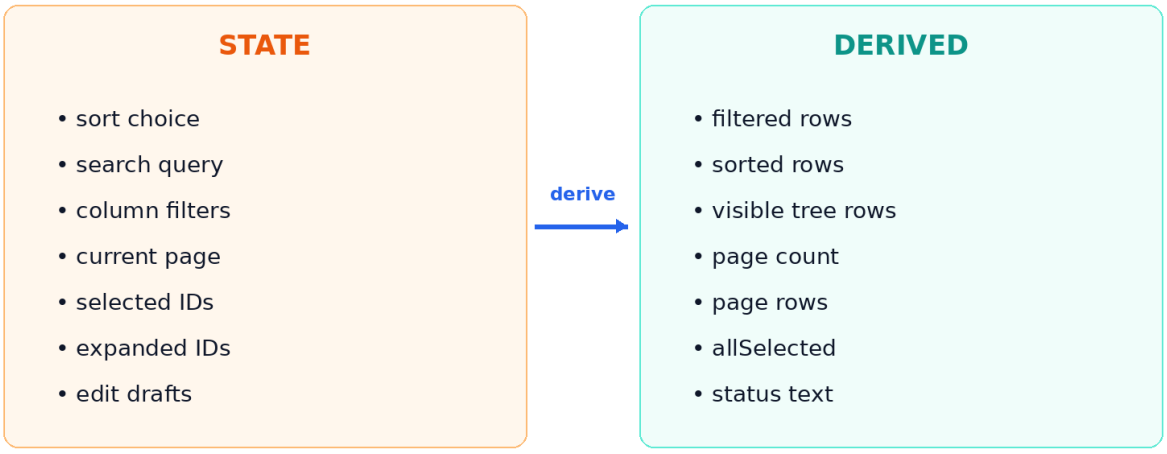

Figure 2. Store choices; derive results.

A useful test
Ask: “Did the user choose this value?” If yes, it may belong in state. Ask: “Can I calculate this value from props and state?” If yes, prefer deriving it.

Book map

Chapter	Engineering question	Pipeline change
01	Columns as data	render
02	Sorting	sort → render
03	Search and filters	filter → sort → render
04	Pagination	filter → sort → slice → render
05	Identity and selection	stable IDs around the pipeline
06	Performance	measure → memoise → scale controls
07	Column reordering	column order becomes state
08	Editing	draft overlay after row pipeline
09	Tree rows	filter → sort → flatten → slice
10	Server-side processing	pipeline crosses the network
11	Headless architecture	model separated from markup

0. Start with an empty React application

Before we build a DataTable, make the surrounding app boring. We want almost all of our attention on the table model, not on scaffolding. The repository uses Vite, but the component itself does not depend on Vite.

Create the project shape

Suggested starting files

```
src/
  App.jsx
  App.css
  index.css
  main.jsx
  components/
    datatable/
      index.jsx
      datatable.css
  data/
    users.js
```

src/main.jsx

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import App from "./App.jsx";
import "./index.css";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

src/index.css and the first .app rule

```
* { box-sizing: border-box; }
body {
  margin: 0;
  font-family: Inter, "Segoe UI", system-ui, sans-serif;
  background: #f8fafc;
  color: #0f172a;
}
.app {
  max-width: 960px;
  margin: 2rem auto;
  padding: 0 1rem;
}
```

src/data/users.js - a small dataset is enough to begin

```
const users = [
  { id: 1, name: "Aarav Sharma", age: 28, qualification: "B.Tech", completed: true,  rating: 4 },
  { id: 2, name: "Priya Patel",  age: 31, qualification: "MBA",    completed: false, rating: 5 },
  { id: 3, name: "Rohan Mehta",  age: 24, qualification: "B.Sc",   completed: true,  rating: 3 },
  { id: 4, name: "Meera Pillai", age: 29, qualification: "M.Tech", completed: true,  rating: 5 },
  { id: 5, name: "Kavya Menon",  age: 27, qualification: "M.Sc",   completed: false, rating: 4 },
];

export default users;
```

Why start with five rows?
Small data makes correctness visible. We will deliberately create a deterministic 10,000-row dataset only when we reach the performance chapter.

1. Why table libraries ask for a columns array

A hand-written table usually repeats the same knowledge three times: once in the header, once when a row renders, and again when a feature such as sorting needs to know which field belongs to that column. A columns array turns that repeated structure into data.

Define the columns

Column configuration

```
const columns = [
  { title: "Id", accessor: "id", dataType: "number", width: "70px" },
  { title: "Name", accessor: "name", dataType: "string" },
  { title: "Age", accessor: "age", dataType: "number", width: "90px" },
  { title: "Qualification", accessor: "qualification", dataType: "string" },
  { title: "Completed", accessor: "completed", dataType: "boolean", width: "130px" },
  {
    title: "Rating",
    accessor: "rating",
    dataType: "number",
    width: "180px",
    cell: (row) => (
      <div className="rating-bar" style={{ width: `${(row.rating / 5) * 100}%` }}>
        {row.rating}
      </div>
    ),
  },
];
```

Each column has a title for the header, an accessor that connects the definition to a row property, and a dataType used by later sorting/alignment logic. The optional cell function is a render escape hatch. Most columns can render the raw value; special columns can render JSX without teaching the DataTable what a rating is.

Build the first DataTable

src/components/datatable/index.jsx

```
import "./datatable.css";

function DataTable({ title = "DataTable", columns, data, noData = "No records found" }) {
  return (
    <div className="datatable">
      <table>
        <caption>{title}</caption>
        <thead>
          <tr>
            {columns.map((col) => (
              <th key={col.accessor} scope="col" style={{ width: col.width }}>
                {col.title}
              </th>
            ))}
          </tr>
        </thead>
        <tbody>
          {data.length === 0 ? (
```

```
            <tr className="no-data">
              <td colSpan={columns.length}>{noData}</td>
            </tr>
          ) : (
            data.map((row, idx) => (
              <tr key={idx}>
                {columns.map((col) => (
                  <td key={col.accessor} className={col.dataType}>
                    {col.cell ? col.cell(row) : String(row[col.accessor])}
                  </td>
                ))}
              </tr>
            ))
          )}
        </tbody>
      </table>
    </div>
  );
}

export default DataTable;
```

src/App.jsx - use the columns array from above

```
import DataTable from "./components/datatable";
import users from "./data/users";

export default function App() {
  return (
    <main className="app">
      <DataTable title="User profiles" columns={columns} data={users} />
    </main>
  );
}
```

What makes this accessible already?

The data is still a real <table>. We are not recreating table semantics with div elements.

The table has a <caption>, so assistive technology has a name for the data region.

Each column header is a <th scope="col">.

The custom rating bar still renders the numeric rating as text, so the visual bar is not the only way to read the value.

The empty state remains inside the table and spans the correct number of columns.

Deliberate imperfection
The first version uses the array index as the React row key. That is safe enough for this static lesson, but we will replace it with a stable row ID in Chapter 5. Keeping the flaw visible makes the identity lesson concrete.

Checkpoint 01
You can add, remove, or rearrange a column by changing the columns array. The DataTable component itself does not need to know the business meaning of a column.

2. Sorting: store the choice, derive the rows

Sorting creates the first important state question. Should we store “the sorted rows” in state? No. The user did not choose those rows. The user chose a column and a direction. Sorted rows are a result we can derive from data plus that choice.

Add sort state

Inside DataTable

```
import { useState } from "react";

const [sort, setSort] = useState({ col: null, dir: "asc" });

function onSort(column) {
  setSort(prev => ({
    col: column.accessor,
    dir: prev.col === column.accessor && prev.dir === "asc" ? "desc" : "asc",
  }));
}
```

Compare values by type

Comparator helper

```
function compareBy(column, dir) {
  const sign = dir === "desc" ? -1 : 1;
  return (a, b) => {
    const x = a[column.accessor];
    const y = b[column.accessor];
    let result;

    if (column.dataType === "number") {
      result = x - y;
    } else if (column.dataType === "boolean") {
      result = x === y ? 0 : x ? 1 : -1;
    } else {
      result = String(x).localeCompare(String(y));
    }

    return result * sign;
  };
}
```

Derive rows without mutating props

The first two pipeline stages: rows → sort

```
let rows = data;
const sortColumn = columns.find((c) => c.accessor === sort.col);

if (sortColumn) {
  rows = [...data].sort(compareBy(sortColumn, sort.dir));
}
```

Array.prototype.sort mutates the array. data is a prop owned by the parent, so sort a copy. The copy is not defensive ceremony; it protects the ownership boundary.

Make the header a real control

Sortable header markup

```
{columns.map((col) => {
  const sortable = col.sortable !== false;
  const isSorted = sort.col === col.accessor;
  return (
    <th
      key={col.accessor}
      scope="col"
      style={{ width: col.width }}
      aria-sort={isSorted ? (sort.dir === "asc" ? "ascending" : "descending") : undefined}
    >
      {sortable ? (
        <button type="button" className="header-sort-button" onClick={() => onSort(col)}>
          {col.title}
          <span className={isSorted ? "sort-arrow" : "sort-arrow idle"} aria-hidden="true">
            {isSorted ? (sort.dir === "asc" ? "▲" : "▼") : "◇"}
          </span>
        </button>
      ) : col.title}
    </th>
  );
})}
```

Why a button inside <th>?
A clickable <th> is not automatically a keyboard control. The button gives us focus, keyboard activation and familiar semantics for free. aria-sort belongs on the header because it describes the column's current sort state.

Checkpoint 02
Click a header once for ascending order and again for descending order. Add a row in the parent: the correct sorted view should be recomputed from the new data without synchronising a second "sortedData" state variable.

3. Search: pipeline order starts to matter

Now the table has two transformations: filtering and sorting. The order matters. If the user searches for "Priya", we do not want to sort thousands of rows that will immediately be discarded. Filter first, then sort the survivors.

Search state and global matcher

```
const [query, setQuery] = useState("");
const [colQueries, setColQueries] = useState({});

function rowMatches(row, columns, text) {
  const needle = text.toLowerCase();
  return columns.some((col) =>
    String(row[col.accessor]).toLowerCase().includes(needle)
  );
}
```

Enable global and optional per-column search

```
const searchEnabled = search ? search.enabled !== false : false;
const perColumn = searchEnabled && search.perColumn === true;

function onColQuery(accessor, value) {
  setColQueries((prev) => ({ ...prev, [accessor]: value }));
}
```

The pipeline is now rows → filter → sort

```
let rows = data;

if (query) {
  rows = rows.filter((row) => rowMatches(row, columns, query));
}

for (const col of columns) {
  const text = colQueries[col.accessor];
  if (text) {
    const needle = text.toLowerCase();
    rows = rows.filter((row) =>
      String(row[col.accessor]).toLowerCase().includes(needle)
    );
  }
}

const sortColumn = columns.find((c) => c.accessor === sort.col);
if (sortColumn) {
  rows = [...rows].sort(compareBy(sortColumn, sort.dir));
}
```

Search input and live status

```
{searchEnabled && (
  <div className="datatable-search">
    <input
      type="search"
      placeholder="Search"
      aria-label="Search all columns"
      value={query}
```

```
      onChange={(e) => setQuery(e.target.value)}
    />
  </div>
)}

<output className="datatable-status">
  Showing {rows.length} of {data.length} rows
</output>
```

A second header row for per-column search

```
{perColumn && (
  <tr className="filter-row">
    {columns.map((col) => (
      <td key={col.accessor}>
        <input
          type="search"
          aria-label={`Search ${col.title}`}
          value={colQueries[col.accessor] ?? ""}
          onChange={(e) => onColQuery(col.accessor, e.target.value)}
        />
      </td>
    ))}
  </tr>
)}
```

Per-column filtering follows the same rule. Store the text that the user typed in each column. Do not store a second filtered array. A simple object keyed by accessor is enough: { name: "pri", qualification: "tech" }.

Accessibility is not a later pass
Search changes the result set without a page navigation. A polite live region gives screen-reader users the same important feedback a sighted user gets from watching rows disappear.

Checkpoint 03
Global search and column search can be combined. The count updates as you type. Sorting still works on the filtered result, not on the original data array.

4. Pagination: the slice must come last

Pagination looks like a UI feature, but its real lesson is pipeline placement. Search and sort operate on the complete matching result. Page slicing is the last data transformation before render.

Paging choices

```
const [pageLength, setPageLength] = useState(
  (pagination && pagination.pageLength) || 10
);
const [page, setPage] = useState(1);

const paging = Boolean(pagination?.enabled);
const pageLengths = pagination?.pageLengths || [5, 10, 20];
```

Derive a valid page, then slice

```
const total = rows.length;
const pageCount = paging
  ? Math.max(1, Math.ceil(total / pageLength))
  : 1;

const currentPage = Math.min(page, pageCount);
const start = (currentPage - 1) * pageLength;

if (paging) {
  rows = rows.slice(start, start + pageLength);
}
```

Notice currentPage. Imagine that you are on page 4 and then search until only two pages remain. The stored page can temporarily still be 4. We do not need an effect whose only job is to repair it. Derive a safe current page with Math.min(page, pageCount).

First pager - deliberately simple

```
<button
  type="button"
  disabled={currentPage === 1}
  onClick={() => setPage(p => Math.max(1, p - 1))}
>
  Previous
</button>

{Array.from({ length: pageCount }, (_, i) => i + 1).map(n => (
  <button
    key={n}
    type="button"
    aria-current={n === currentPage ? "page" : undefined}
    onClick={() => setPage(n)}
  >
    {n}
  </button>
))}

<button
  type="button"
  disabled={currentPage === pageCount}
  onClick={() => setPage(p => Math.min(pageCount, p + 1))}
>
  Next
</button>
```

Changing page length deliberately returns to page 1

```
<label className="page-length">
  Rows per page{" "}
  <select
    value={pageLength}
    onChange={(e) => {
      setPageLength(Number(e.target.value));
      setPage(1);
    }}
  >
    {pageLengths.map((n) => <option key={n} value={n}>{n}</option>)}
  </select>
</label>
```

Pipeline checkpoint

rows → filter → sort → slice → render. If you slice before filtering, a search would only inspect the current page. If you slice before sorting, each page would be sorted independently instead of the result set being globally sorted.

Checkpoint 04

Go to the last page, then enter a search that reduces the result to one or two pages. The UI should show a valid page without crashing or rendering an impossible pager state.

5. Identity and selection: a row is not its position

Figure 3. Sorting changes positions, but a row ID remains the same.

The moment rows can move, an array index stops being a reliable identity. Sorting, filtering and paging all change positions. Selection makes the consequence easy to see: if selection is tied to position, the highlight can jump to a different person.

Selection is a Set of stable IDs

```
export default function DataTable({
  // ...
  keyField = "id",
  selection,
}) {
  const [selectedIds, setSelectedIds] = useState(new Set());
  const selectable = Boolean(selection?.enabled);
  const labelField = selection?.labelField || keyField;

  function onToggleRow(id) {
    setSelectedIds(current => {
      const next = new Set(current);
      next.has(id) ? next.delete(id) : next.add(id);
      return next;
    });
  }
}
```

Use the same identity for React and for selection

```
<tr key={row[keyField]} className={selectedIds.has(row[keyField]) ? "selected" : ""}>
  {selectable && (
    <td className="select-cell">
      <input
        type="checkbox"
        checked={selectedIds.has(row[keyField])}
        onChange={() => onToggleRow(row[keyField])}
        aria-label={`Select ${row[labelField]}`}
      />
    </td>
  )}
  {/* normal cells */}
</tr>
```

What does “select all” mean?

The reference implementation defines “all” as all rows matching the current filters, not merely the page you can see. That semantic choice matters. It means selection is based on the filtered result before pagination slices it.

Indeterminate selection

```
const filtered = rows; // keep the full matching result before the page slice
const allSelected = filtered.length > 0 &&
  filtered.every((row) => selectedIds.has(row[keyField]));
const someSelected = filtered.some((row) => selectedIds.has(row[keyField]));

function onToggleAll() {
  setSelectedIds((prev) => {
    const next = new Set(prev);
    if (allSelected) filtered.forEach((row) => next.delete(row[keyField]));
    else filtered.forEach((row) => next.add(row[keyField]));
    return next;
  });
}

<input
  type="checkbox"
  aria-label="Select all matching rows"
  checked={allSelected}
  ref={(el) => {
    if (el) el.indeterminate = someSelected && !allSelected;
  }}
  onChange={onToggleAll}
/>
```

Why set indeterminate with a ref?
The visual indeterminate state is a DOM property, not a normal HTML attribute. React can set it after the element exists through a ref.

Checkpoint 05
Select a row, sort by another column, search, and move between pages. The selected row remains the same logical record because every feature uses the stable ID.

6. Performance: measure first, then memoise

A DataTable is a perfect place to learn the difference between “this code can be memoised” and “this code should be memoised.” Chapter 6 deliberately creates a 10,000-row case and measures it with React Profiler before changing anything.

Measure actual render cost

```
import { Profiler, useState } from "react";
import { makeUsers } from "./data/make-users";

<Profiler
  id="datatable"
  onRender={(id, phase, actualDuration) => {
    console.log(id, phase, actualDuration.toFixed(1), "ms");
  }}
>
  <DataTable {...props} />
</Profiler>
```

A deterministic large dataset

```
const FIRST = [
  "Aarav", "Priya", "Rohan", "Ananya", "Vikram", "Sneha", "Arjun", "Kavya",
  "Aditya", "Ishita", "Karthik", "Meera", "Siddharth", "Nisha", "Rahul",
  "Divya", "Amit", "Pooja", "Manish", "Lakshmi", "Dev", "Riya", "Kabir",
  "Tara", "Om", "Zara",
];
const LAST = [
```

```
  "Sharma", "Patel", "Mehta", "Iyer", "Nair", "Kulkarni", "Reddy", "Menon",
  "Joshi", "Banerjee", "Rao", "Pillai", "Bose", "Verma", "Desai", "Krishnan",
  "Chauhan", "Hegde", "Tiwari", "Subram", "Kumar", "Gupta", "Singh", "Das",
];
const QUAL = ["B.Com", "B.Sc", "B.Tech", "Graduate", "MBA", "M.Tech", "M.Com", "M.Sc"];

export function makeUsers(count, seed = 42) {
  let s = seed >>> 0;
  const rnd = () => ((s = (s * 1664525 + 1013904223) >>> 0) / 2 ** 32);
  return Array.from({ length: count }, (_, i) => ({
    id: i + 1,
    name: `${FIRST[(rnd() * FIRST.length) | 0]} ${LAST[(rnd() * LAST.length) | 0]}`,
    age: 20 + ((rnd() * 35) | 0),
    qualification: QUAL[(rnd() * QUAL.length) | 0],
    rating: 1 + ((rnd() * 5) | 0),
    completed: rnd() < 0.5,
  }));
}
```

Deterministic data matters for a teaching benchmark. If the input changes on every run, it becomes harder to know whether a timing difference came from your code or from different data.

Memoise the expensive transformation after measuring it

```
const processedRows = useMemo(() => {
  let rows = data;

  // global filter
  // per-column filters
  // sort

  return rows;
}, [data, columns, query, colQueries, sort]);
```

The surprising bottleneck: controls can scale too

A pager that renders one button per page is harmless with 20 rows. With thousands of pages, the pager itself can become expensive. Performance work therefore includes the UI created by the data, not just the filter and sort functions.

Scale the pager instead of creating thousands of buttons

```
function pageItems(current, count) {
  if (count <= 9) return Array.from({ length: count }, (_, i) => i + 1);

  const wanted = new Set([1, count]);
  for (let n = current - 2; n <= current + 2; n += 1) {
    if (n > 1 && n < count) wanted.add(n);
  }

  const pages = [...wanted].sort((a, b) => a - b);
  const items = [];
  for (let i = 0; i < pages.length; i += 1) {
    if (i > 0 && pages[i] - pages[i - 1] > 1) items.push("gap");
    items.push(pages[i]);
  }
  return items;
}
```

Checkpoint 06
Load 10,000 rows. Record the profiler timing before and after the changes. Confirm that the pager remains a small, bounded group of controls even when pageCount is large.

7. Column reordering: move state, not DOM nodes

Dragging a header can tempt us to imperatively move DOM elements. React gives us a cleaner model: the current column order is state. A drop changes that state. React renders headers, filters and cells from the new orderedColumns array.

The only thing that “moves” is columnOrder

```
const [columnOrder, setColumnOrder] = useState(
  () => columns.map(col => col.accessor)
);

function moveColumn(accessor, toIndex) {
  setColumnOrder(current => {
    const next = current.filter(id => id !== accessor);
    next.splice(toIndex, 0, accessor);
    return next;
  });
}

const orderedColumns = columnOrder
  .map(accessor => columns.find(col => col.accessor === accessor))
  .filter(Boolean);
```

Drag is convenient; buttons keep the feature keyboard operable

```
<th
  draggable
  onDragStart={event => {
    event.dataTransfer.setData("text/plain", col.accessor);
  }}
  onDragOver={event => event.preventDefault()}
  onDrop={event => {
    event.preventDefault();
    const from = event.dataTransfer.getData("text/plain");
    moveColumn(from, index);
  }}
>
  {/* sort button */}
  <button
    type="button"
    aria-label={`Move ${col.title} left`}
    disabled={index === 0}
    onClick={() => moveColumn(col.accessor, index - 1)}
  >
    {"<"}
  </button>
  <button
    type="button"
    aria-label={`Move ${col.title} right`}
    disabled={index === orderedColumns.length - 1}
    onClick={() => moveColumn(col.accessor, index + 1)}
  >
    {">"}
  </button>
</th>
```

Use orderedColumns everywhere a column order matters: the header row, the per-column filter row and every body row. If one of those still maps the original columns array, the table will become visually misaligned.

Checkpoint 07
Move a column with drag-and-drop, then move it again using only the keyboard-accessible buttons. Headers, filters and data cells always move together.

8. Editing: who owns the data?

Inline editing introduces an ownership boundary. The parent passed data to the table, so the table should not silently replace that committed data. Instead, the table owns temporary drafts. When the user saves, the table produces updated data and calls the parent callback.

Drafts overlay committed values

```
const [edits, setEdits] = useState({});
const editable = Boolean(edit?.enabled);

function onEditCell(id, accessor, value) {
  setEdits(current => ({
    ...current,
    [id]: {
      ...current[id],
      [accessor]: value,
```

```
    },
  }));
}

function cellValue(row, col) {
  const id = row[keyField];
  return edits[id] && col.accessor in edits[id]
    ? edits[id][col.accessor]
    : row[col.accessor];
}
```

Escape removes only that draft cell

```
function onCancelCell(id, accessor) {
  setEdits((prev) => {
    const row = { ...prev[id] };
    delete row[accessor];
    const next = { ...prev, [id]: row };
    if (Object.keys(row).length === 0) delete next[id];
    return next;
  });
}
```

Commit at an explicit boundary

```
function onSubmit() {
  const nextData = data.map(row => {
    const draft = edits[row[keyField]];
    return draft ? { ...row, ...draft } : row;
  });

  onUpdateData?.(nextData);
  setEdits({});
}
```

Why draft values are overlaid after the row pipeline

Imagine editing a value in the very column being sorted. If draft values immediately participate in sorting, the row can jump to another position while the user is still typing. The reference design deliberately keeps filter and sort based on committed row data and overlays draft cell values only when rendering. After Save, the parent sends committed data back and the normal pipeline uses the new values.

Editors remain column-driven

```
function renderEditor(row, col) {
  if (col.controlType === "select") {
    return (
      <select
        className="cell-editor"
        aria-label={`${col.title} for ${row[labelField]}`}
        value={String(cellValue(row, col))}
        onChange={(e) => onEditCell(row[keyField], col.accessor, e.target.value)}
      >
        {(col.options ?? []).map((option) => (
          <option key={String(option.value)} value={String(option.value)}>
            {option.text}
          </option>
        ))}
      </select>
    );
  }

  return (
    <input
      type="text"
      className="cell-editor"
      aria-label={`${col.title} for ${row[labelField]}`}
      value={String(cellValue(row, col))}
      onChange={(e) => onEditCell(row[keyField], col.accessor, e.target.value)}
      onKeyDown={(e) => {
        if (e.key === "Escape") onCancelCell(row[keyField], col.accessor);
      }}
    />
  );
}
```

Checkpoint 08
Edit several cells. Sorting and filtering should not make the active draft jump around. Press Escape to cancel a draft cell. Save changes and confirm the parent data updates.

9. Tree tables: keep the data nested, flatten the view

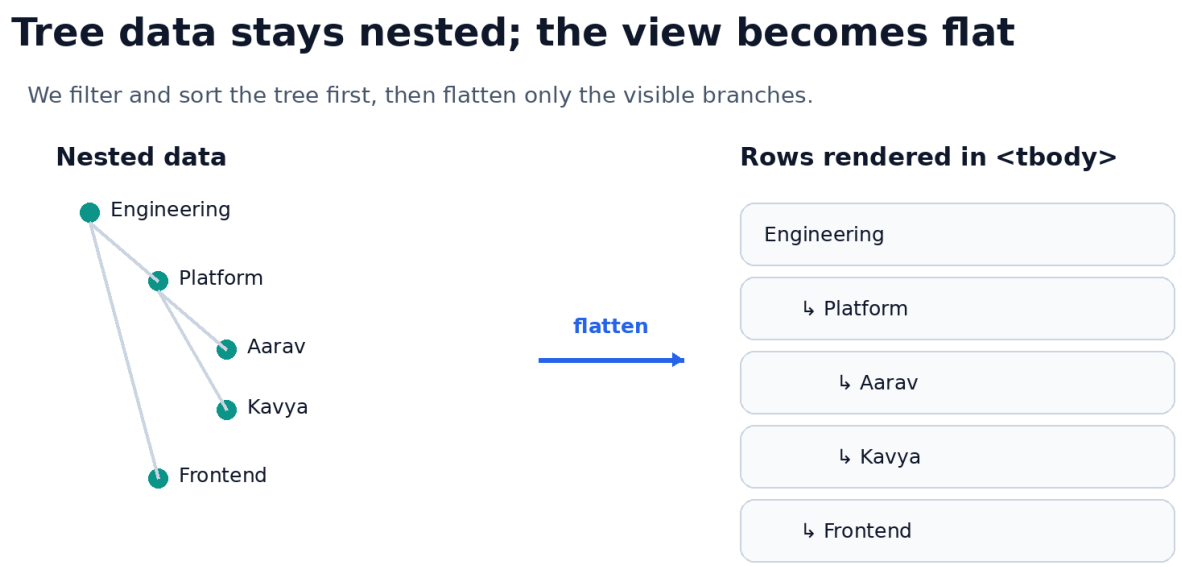

Figure 4. A semantic <tbody> is flat even when the source data is a tree.

A table body contains sibling <tr> elements. A hierarchy, however, is naturally nested. The solution is not to destroy the source tree. Keep the data nested so filtering and sorting can respect parent/child relationships, then flatten only the rows that should currently be visible.

Expansion is another choice keyed by stable identity

```
const [expandedIds, setExpandedIds] = useState(new Set());
const treeMode = Boolean(childrenKey);

function toggleExpanded(id) {
  setExpandedIds(current => {
    const next = new Set(current);
    next.has(id) ? next.delete(id) : next.add(id);
    return next;
  });
}
```

Filter a tree without losing the path to a match

Keep a parent when either it or a descendant matches

```
const filterTree = (nodes) =>
  nodes
    .map((node) => {
      const kids = node[childrenKey] ? filterTree(node[childrenKey]) : [];
      if (!rowPasses(node) && kids.length === 0) return null;
      return node[childrenKey] ? { ...node, [childrenKey]: kids } : node;
    })
    .filter(Boolean);
```

Sort siblings at every level

```
const sortTree = (nodes) => {
  const sorted = sortColumn
    ? [...nodes].sort(compareBy(sortColumn, sort.dir))
    : nodes;

  return sorted.map((node) =>
    node[childrenKey]
      ? { ...node, [childrenKey]: sortTree(node[childrenKey]) }
      : node
  );
};
```

Flatten only visible branches

```
const filtering = Boolean(query) || Object.values(colQueries).some(Boolean);
const depthOf = new Map();
const hasChildren = (row) =>
  treeMode && Array.isArray(row[childrenKey]) && row[childrenKey].length > 0;

let rows = [];
const flatten = (nodes, depth) => {
  for (const node of nodes) {
    rows.push(node);
    depthOf.set(node[keyField], depth);
    if (hasChildren(node) && (filtering || expandedIds.has(node[keyField]))) {
      flatten(node[childrenKey], depth + 1);
    }
  }
};

flatten(filtered, 0);
```

Notice the order: filter tree → sort tree → flatten visible branches → paginate the flat visible result. Slice before flattening and you no longer have a predictable notion of a “page of visible rows.”

Expansion is exposed as an accessible button state

```
<button
  type="button"
  className="tree-toggle"
  aria-expanded={expandedIds.has(row[keyField])}
  aria-label={`${expandedIds.has(row[keyField]) ? "Collapse" : "Expand"} ${row[labelField]}`}
  onClick={() => onToggleExpand(row[keyField])}
>
  <span aria-hidden="true">
    {expandedIds.has(row[keyField]) ? "▾" : "▸"}
  </span>
</button>
```

Checkpoint 09
Search for a child record. Its ancestor path remains visible. Sort the tree and confirm siblings sort within their level. Collapse a branch and confirm the descendants disappear without losing their identity or data.

10. A million rows: move the pipeline, not the idea

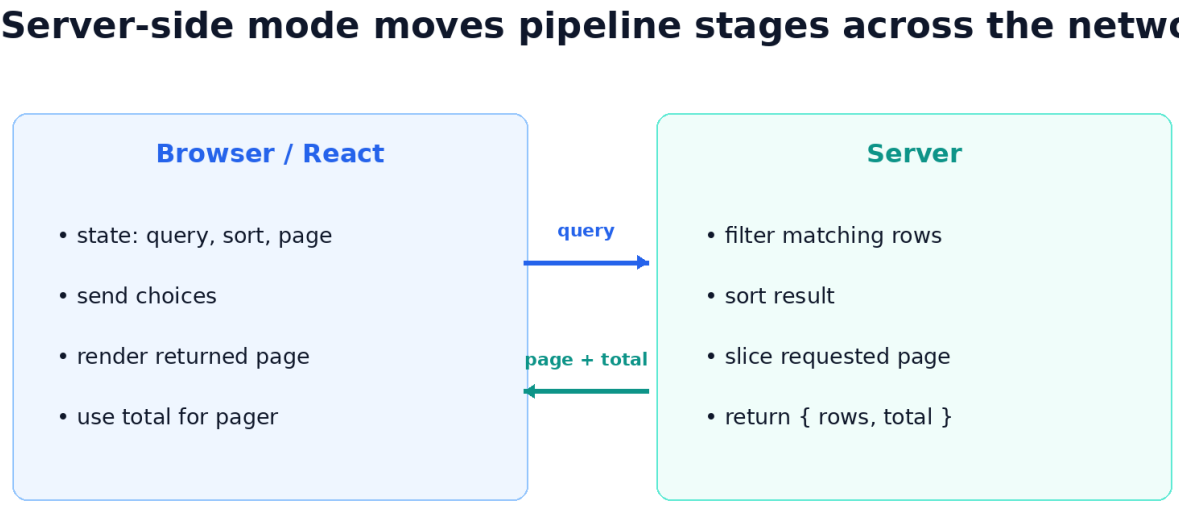

Figure 5. Server-side processing preserves the same model while moving expensive stages.

A million rows changes where work should happen, but not the conceptual pipeline. The browser still owns the user choices. In server-side mode, it reports query, sort, page and pageLength. The server applies filter, sort and slice, then returns only the requested rows plus the total match count.

Report user choices to the parent in remote mode

```
const remote = Boolean(serverSide);

useEffect(() => {
  if (!remote || !onQueryChange) return;
  onQueryChange({ query, sort, page, pageLength });
}, [remote, onQueryChange, query, sort, page, pageLength]);
```

In remote mode, the client must not filter, sort or slice again. The data prop already represents the current page. totalRecords tells the pager how many matching records exist on the server.

Skip local processing when the server already did it

```
const remote = Boolean(serverSide);

const processedRows = useMemo(() => {
  if (remote) return data;

  let rows = data;
  // local filter
  // local sort
  // local tree processing
  return rows;
}, [remote, data, /* local pipeline dependencies */]);

const total = remote ? (totalRecords ?? 0) : processedRows.length;
const visibleRows = paging && !remote
  ? processedRows.slice(start, start + pageLength)
  : processedRows;
```

Protect the UI from stale responses

The parent now owns the current remote page

```
const API = "http://localhost:4600/api/users";

const [data, setData] = useState([]);
const [total, setTotal] = useState(0);
const [loading, setLoading] = useState(false);
```

Abort an older request when a newer choice arrives

```
const [, setController] = useState(null);

const onQueryChange = useCallback((choices) => {
  setController((prev) => {
    if (prev) prev.abort();
    const next = new AbortController();

    const params = new URLSearchParams({
      q: choices.query,
      sortCol: choices.sort.col ?? "",
      sortDir: choices.sort.dir,
      page: String(choices.page),
      pageLength: String(choices.pageLength),
    });

    setLoading(true);
    fetch(`${API}?${params}`, { signal: next.signal })
      .then((r) => r.json())
      .then((body) => {
        setData(body.rows);
        setTotal(body.total);
        setLoading(false);
      })
      .catch((err) => {
        if (err.name !== "AbortError") setLoading(false);
      });

    return next;
  });
}, []);
```

Without cancellation, a slow response for an old query can arrive after a fast response for the newest query and overwrite the UI with stale data. AbortController makes “latest user choice wins” explicit.

Minimal teaching server

The server runs the same stages

```
const q = url.searchParams.get("q")?.toLowerCase() || "";
const sortCol = url.searchParams.get("sortCol") || "id";
const sortDir = url.searchParams.get("sortDir") === "desc" ? "desc" : "asc";
const page = Math.max(1, Number(url.searchParams.get("page")) || 1);
const pageLength = Math.min(100, Math.max(1, Number(url.searchParams.get("pageLength")) || 10));

let rows = q ? ALL_ROWS.filter(row => haystack(row).includes(q)) : ALL_ROWS;
rows = [...rows].sort(compareBy(sortCol, sortDir));

const total = rows.length;
const start = (page - 1) * pageLength;
const pageRows = rows.slice(start, start + pageLength);

res.end(JSON.stringify({ rows: pageRows, total }));
```

Checkpoint 10
Run the API. Type quickly in search while watching the network panel. Old requests may be cancelled. The browser receives only one page, but the pager still reflects the server’s total matching row count.

11. Headless architecture: separate the table model from the markup

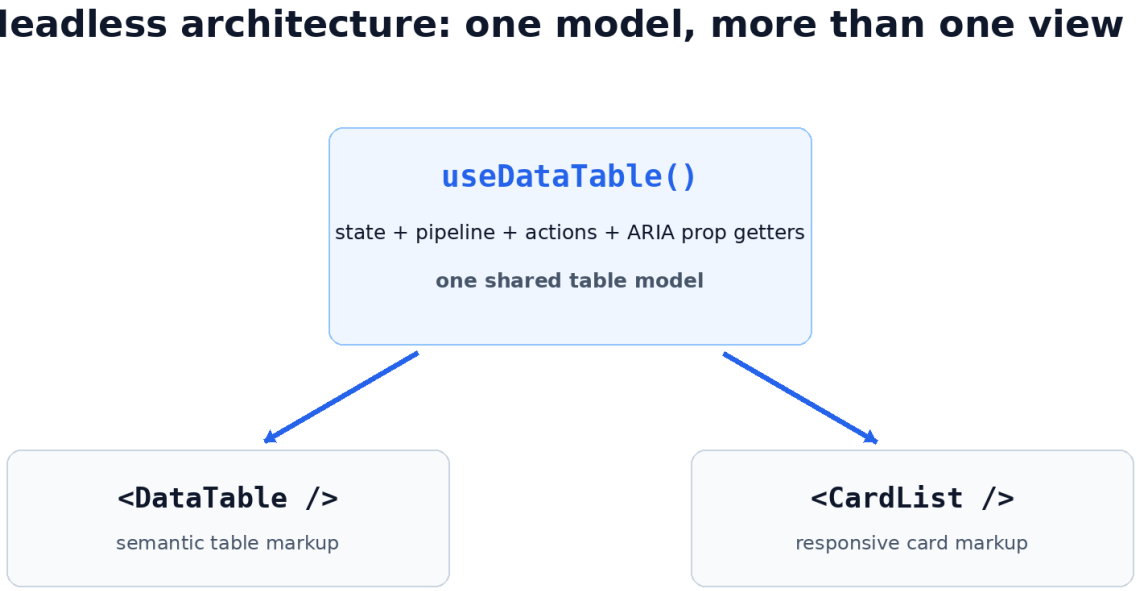

Figure 6. The DataTable and CardList can share one stateful model.

By Chapter 10 our component contains two different responsibilities: table behaviour and table markup. The last refactor makes the behaviour headless. useDataTable owns choices, derived rows, actions and accessible prop getters. DataTable becomes a view. CardList becomes another view.

Refactor without changing behaviour

Copy the working Chapter 10 component before changing anything.

Move choice state and actions into useDataTable first. Keep the JSX calling values from the hook.

Move the derived row pipeline next. Confirm the same rows render before touching markup.

Move reusable event/ARIA bundles into prop getters.

Turn the old DataTable JSX into a view that receives table={table}.

Add CardList as a second renderer of that same table instance.

What moves into useDataTable

Choice state: sort, query, column filters, selection, expansion, drafts, column order, page length and page.

The entire local or server-side row pipeline.

Actions such as moveColumn, toggle selection, toggle expansion, edit, cancel and submit.

Derived values such as total, pageCount, currentPage, selected count, depth information and edited count.

Prop getters that package behaviour and ARIA attributes so each view does not have to re-invent accessibility.

The headless model's shape

```
export function useDataTable({
  columns,
  data,
  keyField = "id",
  childrenKey,
  search,
  pagination,
  selection,
```

```
  edit,
  serverSide,
  totalRecords,
  loading = false,
  onQueryChange,
  onUpdateData,
}) {
  // choice state
  // pipeline
  // actions
  // prop getters

  return {
    keyField, labelField, childrenKey,
    searchEnabled, perColumn, paging, pageLengths, selectable, editable,
    treeMode, remote, loading,

    query, sort, page, pageLength, selectedIds, expandedIds, edits, columnOrder,

    rows, total, start, currentPage, pageCount, pageItems, orderedColumns,
    depthOf, hasChildren, cellValue, editedCount, allSelected, someSelected,

    setQuery, onSort, onColQuery, onToggleRow, onToggleAll, onToggleExpand,
    onEditCell, onCancelCell, onSubmit, moveColumn, setPage, setPageLength,

    getSearchInputProps, getColQueryProps, getSortHeaderProps,
    getSortButtonProps, getSelectAllProps, getRowCheckboxProps,
    getToggleExpandProps, getEditorProps, getPageButtonProps,
    getPageLengthProps,
  };
}
```

A thin DataTable view

DataTable no longer owns the model

```
export default function DataTable({ table: t, title = "DataTable", noData = "No records found" }) {
  return (
    <div className="datatable">
      <table aria-busy={t.loading || undefined}>
        <caption>{title}</caption>
        <thead>{/* headers use t.orderedColumns and prop getters */}</thead>
        <tbody>
          {t.rows.map(row => (
            <tr key={row[t.keyField]}>
              {/* cells use t.cellValue(row, col) */}
            </tr>
          ))}
        </tbody>
      </table>
      {/* status, save controls and pager also read from t */}
    </section>
  );
}
```

A second view with the same model

CardList consumes the same behaviour

```
function CardList({ table }) {
  const t = table;
  return (
    <div className="cardlist">
      {t.rows.map((row) => (
        <div
          key={row[t.keyField]}
          className={`card ${t.selectedIds.has(row[t.keyField]) ? "selected" : ""}`}
        >
          <div className="card-head">
            <span className="card-title">{String(row[t.labelField])}</span>
            {t.selectable && <input {...t.getRowCheckboxProps(row)} />}
          </div>
          <dl>
            {t.orderedColumns
              .filter((col) => col.accessor !== t.labelField)
              .map((col) => (
                <div key={col.accessor}>
                  <dt>{col.title}</dt>
                  <dd>{String(t.cellValue(row, col))}</dd>
                </div>
```

```
              ))}
          </dl>
        </div>
      ))}
    </div>
  );
}

export default CardList;
```

One hook instance synchronises two views

```
export default function App() {
  const [data] = useState(users);

  const table = useDataTable({
    columns,
    data,
    keyField: "id",
    search: { enabled: true },
    pagination: { enabled: true, pageLength: 6 },
    selection: { enabled: true, labelField: "name" },
  });

  return (
    <main className="app app-wide">
      <div className="two-views">
        <DataTable table={table} title="User profiles" />
        <CardList table={table} />
      </div>
    </main>
  );
}
```

Why prop getters matter
A headless abstraction can accidentally push accessibility work onto every consumer. Prop getters keep the behaviour and the accessibility contract together: the view spreads the returned props onto the correct semantic element.

Checkpoint 11
Search, paginate or select from the shared model and confirm both the table and card view reflect the same choices. The model is no longer coupled to <table> markup.

12. Read the finished component like an engineer

The final DataTable is easier to understand if you stop reading it top-to-bottom and instead trace four kinds of information: input configuration, user choices, derived model, and view bindings.

1. Input configuration

columns and data describe the dataset.

Feature objects such as search, pagination, selection and edit enable behaviour without hard-coding business fields.

keyField and childrenKey describe identity and hierarchy.

Callbacks such as onUpdateData and onQueryChange define ownership boundaries.

2. User choices

query and colQueries

sort

page and pageLength

selectedIds

expandedIds
edits
columnOrder

3. Derived model

filtered and sorted rows
flattened visible tree rows
pageCount and currentPage
paged rows
orderedColumns
allSelected and someSelected
status and edited counts

4. View bindings

Prop getters are the final bridge. They turn model state and actions into concrete input props, ARIA state and event handlers. That gives us a reusable behavioural core without making each view memorize the interaction contract.

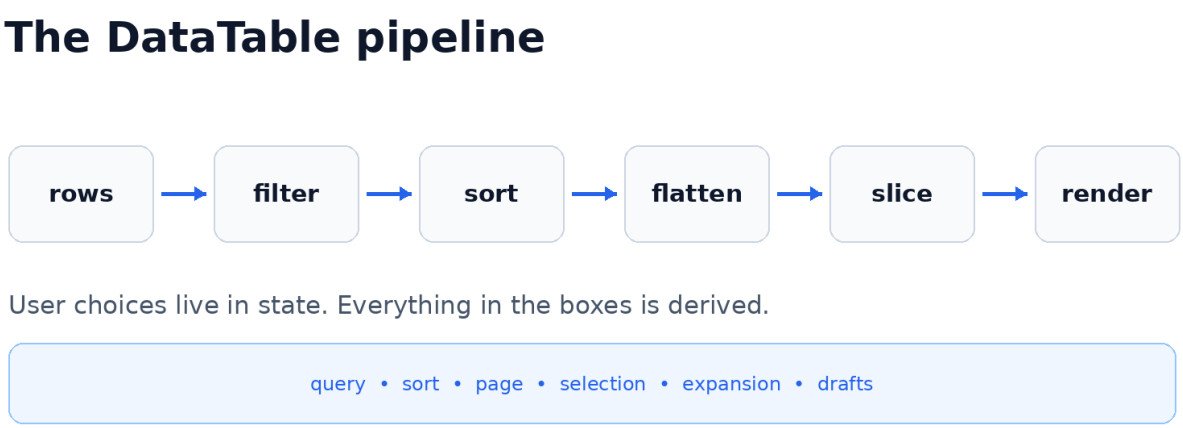

Figure 7. The same pipeline still explains the final system.

13. Accessibility checklist built into the journey

Because accessibility was introduced feature-by-feature, the final checklist reads like a set of engineering invariants rather than a separate retrofit.

Feature	Invariant
Structure	Use a real <table>, <caption>, <thead>, <tbody>, <th scope="col"> and <td> for tabular data.
Sorting	Put interaction on a real button. Put aria-sort on the active column header.
Search	Use labelled inputs and a polite status region when the result count changes.
Pagination	Use buttons, disabled states and aria-current="page" for the active page.
Selection	Give every row checkbox a useful accessible label. Expose the mixed select-all state with the indeterminate DOM property.
Column movement	Do not make drag-and-drop the only way to reorder columns. Provide keyboard-operable controls.
Editing	Use native input/select controls and preserve a

	predictable escape/cancel path.
Tree rows	Use a button for expansion and expose aria-expanded.
Loading	Expose busy state on the table or containing region when remote data is being refreshed.
Headless views	Keep accessibility behaviour in prop getters so alternate renderers inherit the contract instead of starting from zero.

14. A practical test matrix

A table feature rarely breaks in isolation. The difficult bugs happen when features cross. Test combinations deliberately.

Sort + search: Search first; sort the filtered result; clear search and retain the sort choice.

Search + page: Start on a later page; shrink the result; current page must remain valid.

Sort + selection: A selected record stays selected after moving to a different row position.

Filter + select all: Confirm whether “all” means all matching rows; preserve that semantic across pages.

Reorder + filters: Header, column filter and body cell stay aligned after a move.

Edit + sort: A draft in a sorted column should not jump until committed.

Edit + search: Choose whether drafts affect search; this book keeps filtering on committed data.

Tree + filter: A descendant match keeps its ancestor path visible.

Tree + page: Flatten visible rows before slicing so pagination describes what the user can see.

Remote + fast typing: Older requests cannot overwrite the newest query result.

Headless + two views: A choice made through the shared model is reflected by both renderers.

15. What to add before calling it a production table library

This project is deliberately teaching code. It demonstrates the architecture of a capable DataTable, but a production-grade table library has a much wider compatibility and interaction surface.

Row and column virtualisation for very large client-side render sets.

A complete keyboard grid-navigation model when the UX requires spreadsheet-like cell navigation.

Touch-friendly column resize and reorder behaviour.

Column resizing, pinning, visibility, grouping and persisted preferences.

More robust locale-aware and domain-aware sorting/filtering strategies.

Server query caching, request deduplication, retries and error UI.

Controlled/uncontrolled state APIs for applications that want to own table choices externally.

Automated accessibility testing and browser/assistive-technology regression coverage.

Virtualised tree rows and large nested datasets.

A full test suite for interactions between every feature rather than only happy-path examples.

When to use a library
After learning the internals, a production project may reasonably use TanStack Table or another mature library. Understanding this book makes that library easier to use because its concepts stop feeling arbitrary.

Appendix A. Final configuration API at a glance

Final headless configuration

```
const table = useDataTable({
  columns,
  data,
  keyField: "id",
  childrenKey: "children",        // optional tree mode
  search: {
    enabled: true,
    perColumn: true,
  },
  pagination: {
    enabled: true,
    pageLength: 10,
    pageLengths: [5, 10, 20],
  },
  selection: {
    enabled: true,
    labelField: "name",
  },
  edit: {
    enabled: true,
  },
  serverSide: false,
  totalRecords: 0,
  loading: false,
  onQueryChange,
  onUpdateData,
});
```

Column definition fields used through the book

A column remains configuration, not table-specific business logic

```
{
  title: "Qualification",      // visible header
  accessor: "qualification",   // stable column identity and row property
  dataType: "string",          // string, number or boolean comparison/alignment
  width: "160px",              // optional layout hint
  sortable: true,               // false disables sorting for this column
  cell: (row) => <strong>{row.qualification}</strong>,
  controlType: "text",         // editing: "text" or "select"
  options: [                    // only needed for select editors
    { text: "Yes", value: true },
    { text: "No", value: false },
  ],
}
```

Appendix B. Minimal CSS foundation

The reference repository contains a fuller teaching stylesheet. The following foundation is enough to reproduce the structure and keep the examples readable while you work through the logic.

Starter styling

```
.datatable { overflow-x: auto; }
.datatable table { width: 100%; border-collapse: collapse; background: white; }
.datatable caption { text-align: left; font-weight: 700; font-size: 1.15rem; padding: 0 0 .75rem; }
.datatable th, .datatable td { border-bottom: 1px solid #e2e8f0; padding: .7rem .75rem; text-align: left; }
.datatable th { background: #f8fafc; }
.datatable .numeric { text-align: right; font-variant-numeric: tabular-nums; }
.datatable .selected { background: #eff6ff; }
.header-sort-button { border: 0; background: transparent; font: inherit; font-weight: 700; cursor: pointer; }
.datatable-search { display: flex; gap: .6rem; align-items: center; margin-bottom: .75rem; }
.datatable-search input, .datatable input, .datatable select { font: inherit; padding: .38rem .5rem; }
.datatable-status { min-height: 1.2em; color: #475569; font-size: .9rem; }
.datatable-pager { display: flex; flex-wrap: wrap; gap: .35rem; align-items: center; margin-top: .8rem; }
.datatable-pager button[aria-current="page"] { font-weight: 700; text-decoration: underline; }
.visually-hidden { position: absolute !important; width: 1px; height: 1px; padding: 0; margin: -1px; overflow:
hidden; clip: rect(0,0,0,0); white-space: nowrap; border: 0; }
.rating-bar { white-space: nowrap; }
.tree-label { display: inline-flex; align-items: center; gap: .35rem; }
.card { border: 1px solid #e2e8f0; border-radius: 10px; padding: 1rem; background: white; }
```

Appendix C. Useful repository commands

Working with the reference checkpoints

```
# Start the final application
cd app
npm install
npm run dev

# Run a particular lesson snapshot from the helper script
cd steps
./run-step.sh 04

# Compare two adjacent lesson source trees
diff -r steps/step-04/src steps/step-05/src

# Lesson 10 API
cd app
node server/server.mjs
```

Closing: the table is the exercise; the pipeline is the lesson

If you remember only the JSX, you will have to re-learn the component when the requirements change. If you remember the pipeline and the ownership rules, new requirements become placement questions.

- Does the user choose it? Store the choice.
- Can it be calculated from props and choices? Derive it.
- Does it change the result set? Put it before pagination.
- Does it change only the current presentation? Keep it after the data transformations.
- Can rows move? Use stable identity.
- Does work become large? Measure before optimising.
- Does data become too large for the browser? Move pipeline stages across the network, not into a different mental model.
- Does the same behaviour need another view? Separate the headless model from markup.

The final mental model

rows → filter → sort → flatten → slice → render

state = what the user chose
derived = everything on screen, recomputed from those choices

The reference repository remains useful as an exact code oracle. But after working through these chapters, you should be able to start with an empty component and explain why every important piece exists before you type it.