dnd-kit Sorting Strategies: Complete Guide

Table of Contents

What is a Sorting Strategy? {#what-is-a-sorting-strategy}

A **sorting strategy** determines **how items are rearranged** during drag operations in a `SortableContext`. It calculates:

1. **Where to insert** the dragged item in the list
2. **How other items should shift** to make room
3. **The visual feedback** during dragging

Basic Structure

```
import { SortableContext, verticalListSortingStrategy } from "@dnd-
kit/sortable";

<SortableContext
    items={items.map(item => item.id)}      // Array of IDs
    strategy={verticalListSortingStrategy}  // How to sort
>
    {items.map(item => (
        <SortableItem key={item.id} item={item} />
    ))}
</SortableContext>
```

Key Point: The strategy is a **function** that dnd-kit uses internally to determine sorting logic.

Available Strategies {#available-strategies}

dnd-kit provides **4 built-in strategies**:

1. `verticalListSortingStrategy`

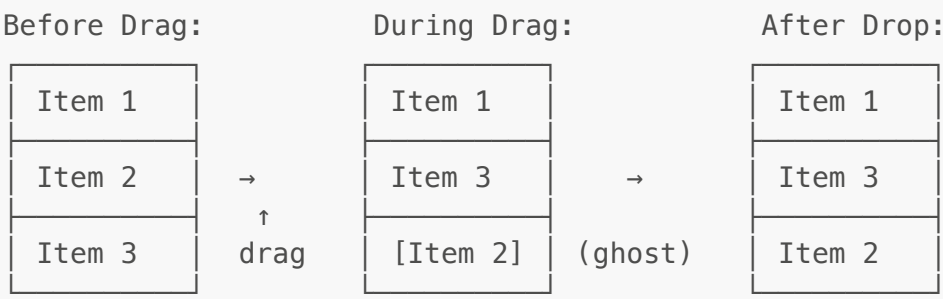
```
import { verticalListSortingStrategy } from "@dnd-kit/sortable";
```

Use for: Single-column vertical lists

How it works:

- Items are arranged **vertically** (one per row)
- When dragging, items shift **up or down** to make room
- Calculates positions based on **Y-axis** (vertical position)

Visual Example:



Characteristics:

- ✅ Simple and efficient
- ✅ Works with any item heights
- ✅ Smooth animations
- ❌ Only for vertical layouts

---

2. `horizontalListSortingStrategy`

```
import { horizontalListSortingStrategy } from "@dnd-kit/sortable";
```

Use for: Single-row horizontal lists

How it works:

- Items are arranged **horizontally** (side by side)
- When dragging, items shift **left or right** to make room
- Calculates positions based on **X-axis** (horizontal position)

Visual Example:

```
During Drag (Item 2):
┌─────┬─────┬─────┬─────┐
│  1  │  3  │ [2] │  4  │
└─────┴─────┴─────┴─────┘
            ← drag

After Drop:
┌─────┬─────┬─────┬─────┐
│  1  │  3  │  2  │  4  │
└─────┴─────┴─────┴─────┘
```

Characteristics:

- ✅ Perfect for horizontal carousels, tabs, breadcrumbs
- ✅ Works with any item widths
- ✅ Smooth left/right animations
- ❌ Only for horizontal layouts

---

3. `rectSortingStrategy`

```tsx
import { rectSortingStrategy } from "@dnd-kit/sortable";
```

Use for: Grid layouts (multi-column, multi-row)

How it works:

- Items can be arranged in **any grid pattern**
- Calculates positions based on **both X and Y axes**
- Uses **rectangular bounding boxes** to determine sorting
- More flexible than vertical/horizontal strategies

Visual Example:

```
Before Drag:
┌─────┬─────┬─────┐
│  1  │  2  │  3  │
├─────┼─────┼─────┤
│  4  │  5  │  6  │
└─────┴─────┴─────┘

During Drag (Item 2 to position 5):
┌─────┬─────┬─────┐
│  1  │  3  │  6  │
├─────┼─────┼─────┤
│  4  │ [2] │     │
└─────┴─────┴─────┘
      drag ↓
```

```
After Drop:
┌─────┬─────┬─────┐
│  1  │  3  │  6  │
├─────┼─────┼─────┤
│  4  │  2  │  5  │
└─────┴─────┴─────┘
```

Characteristics:

- ✅ Works with grids (CSS Grid, Flexbox wrap)
- ✅ Handles both vertical and horizontal movement
- ✅ Flexible item sizes
- ⚠ Slightly more complex calculations
- ⚠ May need CSS adjustments

---

4. `rectSwappingStrategy`

```
import { rectSwappingStrategy } from "@dnd-kit/sortable";
```

Use for: Grid layouts where items **swap positions** instead of shifting

How it works:

- Similar to `rectSortingStrategy` but with **swapping behavior**
- When you drag Item A over Item B, they **swap places** directly
- Other items **don't shift** to make room
- Perfect for fixed-size grids where position matters

Visual Example:

```
Before Drag:
┌─────┬─────┬─────┐
│  1  │  2  │  3  │
├─────┼─────┼─────┤
│  4  │  5  │  6  │
└─────┴─────┴─────┘

During Drag (Item 2 over Item 5):
┌─────┬─────┬─────┐
│  1  │  5  │  3  │  ← Item 5 swaps with Item 2
├─────┼─────┼─────┤
│  4  │ [2] │  6  │
└─────┴─────┴─────┘
     drag ↓

After Drop:
┌─────┬─────┬─────┐
│  1  │  5  │  3  │  ← Only these two swapped
```

```
┌─────┬─────┬─────┐
│  4  │  2  │  6  │  ← Others stay in place
└─────┴─────┴─────┘
```

Characteristics:

- ✅ Direct swapping (no cascading shifts)
- ✅ Great for chess-like grids, tile puzzles
- ✅ Predictable behavior
- ❌ Not suitable for dynamic lists (where order matters)

Strategy Comparison {#strategy-comparison}

Strategy	Layout	Item Shift	Best For
`verticalListSortingStrategy`	Single column	Up/Down	Lists, menus, todos
`horizontalListSortingStrategy`	Single row	Left/Right	Tabs, carousels, tags
`rectSortingStrategy`	Multi-column grid	All shift	Photo grids, dashboards
`rectSwappingStrategy`	Multi-column grid	Only swapped items	Fixed grids, boards

When to Use Which Strategy {#when-to-use-which-strategy}

Use `verticalListSortingStrategy` when:

✅ Items are stacked vertically
✅ One item per row
✅ Order matters (1, 2, 3, ...)
✅ Variable item heights OK

Examples:

- Todo lists
- Folder/file trees
- Navigation menus
- Settings panels
- **Scripts in this project** ✓

Use `horizontalListSortingStrategy` when:

✅ Items are arranged horizontally
✅ One item per column
✅ Order matters (left to right)
✅ Variable item widths OK

Examples:

- Tab bars
- Tag lists
- Breadcrumb navigation
- Horizontal carousels
- Toolbar items

Use `rectSortingStrategy` when:

✅ Multi-column, multi-row layout
✅ Items flow and wrap
✅ Order matters globally
✅ Items shift to fill gaps

Examples:

- Photo galleries
- Product catalogs
- Dashboard widgets
- Kanban columns (when sortable)
- Pinterest-style layouts

Use `rectSwappingStrategy` when:

✅ Multi-column, multi-row layout
✅ Direct position swapping desired
✅ Each position is meaningful
✅ No cascading shifts wanted

Examples:

- Chess/checkers boards
- Tile puzzle games
- Seat assignment grids
- Fixed dashboard layouts
- Icon grids (desktop/launcher)

How Strategies Work {#how-strategies-work}

Internal Algorithm

Each strategy is a function that receives:

```
type Strategy = (args: {
    activeNodeRect: ClientRect | null; // Dragged item's position
    activeIndex: number; // Current index of dragged item
```

```
    index: number; // Index being evaluated
    rects: ClientRect[]; // All item positions
    overIndex: number; // Index user is hovering over
}) => Transform | null;
```

And returns a `Transform`:

```
type Transform = {
    x: number; // Horizontal offset
    y: number; // Vertical offset
    scaleX: number; // Horizontal scale
    scaleY: number; // Vertical scale
};
```

Example: `verticalListSortingStrategy` Logic

```
// Simplified version
function verticalListSortingStrategy(args) {
    const { activeIndex, index, rects } = args;

    // No transform for the active (dragged) item
    if (index === activeIndex) {
        return null;
    }

    // Item should shift down if it's before the dragged item
    if (index < activeIndex) {
        const activeRect = rects[activeIndex];
        return {
            x: 0,
            y: activeRect.height, // Shift down by active item's height
            scaleX: 1,
            scaleY: 1,
        };
    }

    // Item should shift up if it's after the dragged item
    if (index > activeIndex) {
        const activeRect = rects[activeIndex];
        return {
            x: 0,
            y: -activeRect.height, // Shift up by active item's height
            scaleX: 1,
            scaleY: 1,
        };
    }

    return null;
}
```

This is why items smoothly animate to make room!

Examples in This Project {#examples-in-this-project}

1. Script Sorting (Vertical)

```
// src/app-component/ScriptsColumn/SortableScriptsContext.tsx
<SortableContext
    items={folderResponse.shellScripts.map((s) => s.id || 0)}
    strategy={verticalListSortingStrategy}  // ← Vertical list
>
    {folderResponse.shellScripts.map((script) => (
        <SortableScriptItem
            key={script.id}
            script={script}
            parentFolderId={script.parentFolderId}
        />
    ))}
</SortableContext>
```

Why `verticalListSortingStrategy`?

- Scripts are stacked vertically
- One script per row
- Order matters (execution order)

---

2. Folder Sorting (Vertical)

```
// src/app-component/ScriptsColumn/SortableSubfoldersContext.tsx
<SortableContext
    items={folderResponse.subfolders.map((s) => s.id || 0)}
    strategy={verticalListSortingStrategy}  // ← Vertical list
>
    <div className="space-y-2">
        {folderResponse.subfolders.map((folder) => (
            <CollapsableFolder key={folder.id} folder={folder} />
        ))}
    </div>
</SortableContext>
```

Why `verticalListSortingStrategy`?

- Folders are stacked vertically
- One folder per row
- Order matters (organization)

3. Nested Scripts in Folders (Vertical)

```
// Inside SortatbleCollapsableFolder.tsx
{isExpanded && folder.shellScripts.length > 0 && (
    <SortableContext
        items={folder.shellScripts.map((s) => s.id || 0)}
        strategy={verticalListSortingStrategy}  // ← Vertical list
    >
        <div className="ml-8 mt-2 space-y-2">
            {folder.shellScripts.map((script) => (
                <SortableScriptItem
                    key={script.id}
                    script={script}
                    parentFolderId={folder.id}
                />
            ))}
        </div>
    </SortableContext>
)}
```

Why `verticalListSortingStrategy`?

- Nested scripts also stack vertically
- Consistent with parent list
- Order matters within folder

Custom Strategies {#custom-strategies}

You can create **custom strategies** for unique layouts:

Example: Diagonal Grid Strategy

```
import type { SortingStrategy } from "@dnd-kit/sortable";

const diagonalGridStrategy: SortingStrategy = ({
    activeIndex,
    index,
    rects,
    overIndex,
}) => {
    // Custom logic for diagonal arrangement
    if (index === activeIndex) {
        return null;
    }

    const offset = index - overIndex;
    const activeRect = rects[activeIndex];
```

```
    return {
        x: offset * activeRect.width * 0.5,   // Diagonal X
        y: offset * activeRect.height * 0.5,  // Diagonal Y
        scaleX: 1,
        scaleY: 1,
    };
};

// Use it
<SortableContext
    items={items.map(i => i.id)}
    strategy={diagonalGridStrategy}  // ← Custom strategy
>
    {/* ... */}
</SortableContext>
```

Example: Circular Strategy

```
const circularStrategy: SortingStrategy = ({ activeIndex, index, rects })
=> {
    const totalItems = rects.length;
    const angle = (2 * Math.PI * index) / totalItems;
    const radius = 200;

    return {
        x: Math.cos(angle) * radius,
        y: Math.sin(angle) * radius,
        scaleX: 1,
        scaleY: 1,
    };
};
```

Performance Considerations

Strategy Efficiency

Strategy	Complexity	Performance
`verticalListSortingStrategy`	O(1)	⚡ Fastest
`horizontalListSortingStrategy`	O(1)	⚡ Fastest
`rectSortingStrategy`	O(n)	🔶 Moderate
`rectSwappingStrategy`	O(1)	⚡ Fast

Why `rectSortingStrategy` is slower:

- Must calculate positions for **all items** in grid

- Needs to determine **row and column** for each item
- More complex collision detection

Optimization Tips:

1. Use `verticalListSortingStrategy` for simple lists (fastest)
2. Only use `rectSortingStrategy` when you actually need grid layout
3. Limit grid size (virtual scrolling for large grids)
4. Memoize item positions when possible

Common Mistakes

❌ Wrong Strategy for Layout

```
// BAD: Using vertical strategy for horizontal list
<div style={{ display: "flex", flexDirection: "row" }}>
    <SortableContext
        items={items.map(i => i.id)}
        strategy={verticalListSortingStrategy}  // ❌ Wrong!
    >
        {/* Items are horizontal but strategy is vertical */}
    </SortableContext>
</div>
```

```
// GOOD: Match strategy to layout
<div style={{ display: "flex", flexDirection: "row" }}>
    <SortableContext
        items={items.map(i => i.id)}
        strategy={horizontalListSortingStrategy}  // ✅ Correct!
    >
        {/* Strategy matches layout */}
    </SortableContext>
</div>
```

❌ Using `rectSortingStrategy` for Simple Lists

```
// BAD: Overkill for vertical list
<SortableContext
    items={items.map(i => i.id)}
    strategy={rectSortingStrategy}  // ❌ Unnecessarily complex
>
    <div style={{ display: "flex", flexDirection: "column" }}>
        {/* Simple vertical list */}
    </div>
</SortableContext>
```

```
// GOOD: Use simplest strategy that works
<SortableContext
    items={items.map(i => i.id)}
    strategy={verticalListSortingStrategy}  // ✅ Simple and fast
>
    <div style={{ display: "flex", flexDirection: "column" }}>
        {/* Simple vertical list */}
    </div>
</SortableContext>
```

Key Takeaways

1. **Match strategy to layout**: Vertical list → `verticalListSortingStrategy`, Grid → `rectSortingStrategy`
2. **Use simplest strategy possible**: Simpler strategies are faster and more predictable
3. **Strategy determines shifting behavior**: How items move to make room for the dragged item
4. **All strategies work with `useSortable` hook**: The hook uses the strategy internally
5. **You can create custom strategies**: For unique layout requirements
6. **Strategy affects performance**: Simpler strategies (vertical/horizontal) are O(1), rect strategies are O(n)

Further Reading

- dnd-kit Sortable Documentation
- Sorting Strategies API
- Creating Custom Strategies
- Grid Layouts with dnd-kit

Summary

Sorting strategies tell dnd-kit **how to rearrange items** during drag operations:

- `verticalListSortingStrategy`: For vertical lists (most common) ✓
- `horizontalListSortingStrategy`: For horizontal lists
- `rectSortingStrategy`: For grids with shifting
- `rectSwappingStrategy`: For grids with swapping

Choose the strategy that **matches your layout** for the best performance and user experience! 🎯