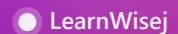


Lists, Trees, Repeaters, and Hierarchical Data

Lesson Guide · Lab / Exam Guide



Lesson Guide

ComboBox, ListBox, ListView, TreeView, DataRepeater and PropertyGrid; ListView virtual mode, TreeView lazy loading, stable node identity and selection-driven detail loading. What this means for a Wisej.NET developer — and how to read this module.



Production focus

ComboBox, ListBox, ListView, TreeView, DataRepeater and PropertyGrid; ListView virtual mode, TreeView lazy loading, stable node identity and selection-driven detail loading.

1. The production problem

List and tree controls are often treated as simple selection widgets, but in a line-of-business application they are the user's map: folders, customers, orders, procedures, roles, tasks, documents, product catalogs and the modules of the application itself. The Operations Console document explorer you build in this module has to stay fast when the category tree is deep and the document list is long, and every click has to map to a real business record rather than to a piece of display text.

2. Key concepts

Concept	Why it matters
data shape	The first design question: is the data flat, grouped, hierarchical, repeated as cards, or a set of properties? The answer selects the control.
ListView views	One control that shows the same collection as LargeIcon, SmallIcon, Details or Tile, the natural choice for file-like and catalog-like screens.
virtual mode	A ListView mode in which the application sets <code>VirtualListSize</code> and answers <code>RetrieveVirtualItem</code> , so items exist only for the rows the control asks for.
lazy loading	Loading only the top levels of a tree at startup and fetching children when a node expands, so startup stays cheap and the UI responsive.
virtual scrolling and prefetch	Rendering only the visible rows of a large list or tree in the browser, with <code>PrefetchItems</code> pre-rendering a few extra rows for smooth scrolling.
stable identity	A unique business ID stored in the node's or item's <code>Tag</code> or <code>Name</code> , or in a view-model map, never derived from displayed text.
selection as an event	A selection change resolves an ID and asks a service for data; no query ever runs inside the code that builds nodes or items.

3. Lesson narrative

Lists and trees are the user's map

Each list control in Wisej.NET answers a different question. `ComboBox` selects one value in a compact space; its `DropDownStyle` decides whether the user can type a free value or must pick from the list. It is right for small and medium reference lists and wrong for thousands of items: a drop-down with a scrollbar is not search. `ListBox` and `CheckedListBox` are for the case where the list itself stays visible and selecting, or ticking, is the task. `ListView` is for collections that benefit from a presentation style: icons, tiles or detail rows. `TreeView` is for hierarchy, expansion and parent/child context.

`DataRepeater` belongs where a repeated visual template communicates better than a grid: a task board, notification cards, product cards, search result cards. Every repeated `ItemTemplate` is a set of server controls with browser widgets behind them, so a large result set needs paging, filtering or a virtualization strategy before it needs a prettier card. `PropertyGrid` is the generic inspector: point `SelectedObject` at an object and the user can read or edit its properties. That suits administrative screens, developer tools and object inspectors, and is less appropriate for polished end-user data entry, where the generic structure exposes implementation details.

The decision rules fit on a card: `ComboBox` for compact selection, `ListView` when presentation style matters, `TreeView` when hierarchy matters, `DataRepeater` when each record needs a card, `PropertyGrid` for generic inspection, virtual mode or lazy loading before a list or tree becomes slow, and stable IDs stored separately from displayed text. In the Operations Console the categories are a `TreeView`, the documents under a category a `ListView` in Details view, and the selected document opens in a detail `UserControl`.

ListView views and virtual mode

A `ListView` changes presentation through its `View` property, so a document catalog can switch between tiles with large icons and a Details view with columns without rebuilding anything. For a handful of items, adding `ListViewItem` objects directly is fine. For data-backed screens, bind the control or give it one clearly named loading method, so there is exactly one place where documents become items.

For large collections, turn on virtual mode. The `Items` collection is then no longer the primary store: the application tells the control how many rows exist through `VirtualListSize` and handles `RetrieveVirtualItem`, creating a `ListViewItem` only when the control asks for a given `ItemIndex`. The handler must be cheap, so it reads from a server-side page or cache that the service filled when the category changed, never from the database row by row. Image key and stable ID are set on the item at the same moment.

```
private void categoryTree_AfterExpand(object sender, TreeViewEventArgs e)
{
    if (e.Node.Nodes.Count != 1 || e.Node.Nodes[0].Name != "placeholder") return;
    e.Node.Nodes.Clear();
    foreach (var child in _documents.GetChildren((string)e.Node.Tag))
        e.Node.Nodes.Add(new TreeNode(child.Title) { Name = child.Id, Tag = child.Id,
ImageKey = "folder" });
}

private void documentList_RetrieveVirtualItem(object sender, RetrieveVirtualItemEventArgs e)
{
    var doc = _page[e.ItemIndex]; // server-side page filled by DocumentService
    e.Item = new ListViewItem(doc.Title) { Name = doc.Id, Tag = doc.Id, ImageKey = doc.Kind
};
}
```

TreeView: lazy loading, check boxes, images and large trees

A `TreeView` shows `TreeNode` objects in a hierarchy. Loading a deep tree completely at startup is the most common performance mistake in this family, so the lab uses lazy loading: build only the top-level categories, give each a single placeholder child so the expand glyph appears, and in `AfterExpand` replace the placeholder with real children fetched from the service. In the snippet above, the placeholder's `Name` is the marker that tells the handler whether the node has already been loaded.

Two properties change what the tree means to the user. `CheckBoxes` shows a check box next to every node and turns the tree into a multi-select surface for batch operations such as archiving several categories. Node images, assigned through an `ImageList` and each node's `ImageKey`, let the user tell folders, documents, warnings and terminal nodes apart at a glance; the `ListView` reuses the same image keys so the two controls speak the same visual language.

When a tree is large even after lazy loading, virtual scrolling reduces browser rendering cost by rendering only the visible nodes, and `PrefetchItems` pre-renders a few additional rows outside the visible area so scrolling stays smooth. Neither replaces lazy loading; they are the second and third strategies, in that order.

Hierarchical data also needs clear identity. The same label appears under many branches: a "Documents" folder exists under every customer, and two categories can share a title. Store a stable ID in the node's `Tag` or `Name`, or in a view-model dictionary keyed by node, so the selected node always maps to one unique business object. Display text is for people; identity is for code.

Selection is an event, not a query

Selection should be an event, not a database query buried inside rendering code. When the user selects a node in the tree or an item in the list, the handler for `AfterSelect` or `SelectedIndexChanged` does three things: resolve the stable ID from `Tag`, ask a `DocumentService` or a view model for the data, and hand the result to the detail `UserControl`. The detail control exposes one method that accepts a document model and knows nothing about the tree, the list or the database.

That split keeps the screen predictable and testable: controls display view state, services retrieve business data, the page coordinates the workflow. It also gives the user honest feedback: the page can show a loading state, catch a failed service call and show a friendly message without leaving half-built items in the list. Before you call the explorer done, run the large-data checklist:

- No unbounded data loads; virtual mode or lazy loading covers every large path.
- A server-side cache or page sits between the controls and the database.
- Stable IDs are separate from display text on every node and item.
- Performance is tested with realistic data, and you know what breaks first at ten times the volume.



Common pitfalls

Loading an entire deep tree at startup. Storing business identity only in node text. Creating thousands of rich repeated controls in a `DataRepeater` instead of paging or virtualizing. Using a `ComboBox` as a poor replacement for search. Binding selection events directly to database code without a service boundary.



Control-library habit

Choose the simplest native control that expresses the user's task before writing custom code. Native controls already give you design-time support, theming, events and server state; reach for an extender or a `UserControl` next, and build a JavaScript widget only when no native control fits.



Learning objectives

By the end of this module you'll be able to:

- ✓ Use `ComboBox`, `ListBox`, `CheckedListBox`, `ListView`, `TreeView`, `DataRepeater`, and `PropertyGrid` for the right interaction pattern.
- ✓ Implement `ListView` virtual mode with `RetrieveVirtualItem` and `VirtualListSize`.
- ✓ Implement `TreeView` loading, check boxes, icons, and large-tree strategies with virtual scrolling and prefetching.
- ✓ Connect selection in a tree or list to a detail panel without coupling data retrieval to UI rendering.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Build the Operations Console document explorer: a `TreeView` of categories or folders with a stable ID attached to every node, a `ListView` of documents tied to the selected node, lazy loading or virtual mode on at least one list or tree path, icons or image keys for the different item types, and a detail `UserControl` that loads through a service or view-model layer whenever the selection changes.

Starting point

Open the `OperationsConsole` solution from Module 3 and go to the Lists and Trees tab of the shell: it still holds the Module 1 placeholder page. Keep this module's work in its own folder or namespace (for example `Explorer`) so the capstone can reuse it, and use a small in-memory `DocumentService` with a few hundred generated documents so the large-data behaviour is visible without a database.

Suggested approach

- ✓ Write down the data shape first: categories are hierarchical, documents under a category are a flat collection, and one document has properties. Choose `TreeView`, `ListView` and a detail `UserControl` and record why a `ComboBox`, `DataRepeater` or `PropertyGrid` was not chosen.
- ✓ Build the category tree lazily: add only the top-level categories, give each a placeholder child, store the category ID in `Tag` or `Name`, and replace the placeholder with real children in `AfterExpand` through the service.
- ✓ Fill the document `ListView` from the selected node: either bind the page returned by the service or enable virtual mode, set `VirtualListSize` and create items in `RetrieveVirtualItem` from a server-side page, never from a query per row.
- ✓ Add an `ImageList` shared by both controls and assign an `ImageKey` per item type (folder, contract, invoice, drawing, warning), so the tree and the list speak the same visual language.
- ✓ Create `DocumentDetailControl` with one public method that accepts a document model; in the selection handlers resolve the stable ID, call the service inside a try/catch, show a loading state and a friendly message on failure, and pass the result to the detail control.

Required deliverables

- ✓ `TreeView` of categories with a stable ID stored on every node
- ✓ `ListView` of documents tied to the selected tree node
- ✓ Lazy loading or virtual mode on at least one list or tree path
- ✓ Detail `UserControl` loaded through a service or view-model layer on selection
- ✓ Icons or image keys for the different item types

Student review questions

- ✓ Two categories under different customers are both called "Contracts". What exactly does your selection handler use to tell them apart, and where does that value live?
- ✓ If the document count under one category grew from 300 to 300,000, which part of your explorer would break first, and which property or event would you change to fix it?
- ✓ Which lines of your selection handler could be moved into a unit test without a browser, and which lines belong to the detail UserControl instead?



Control-library habit

Choose the simplest native control that expresses the user's task before writing custom code. Native controls already give you design-time support, theming, events and server state; reach for an extender or a UserControl next, and build a JavaScript widget only when no native control fits.