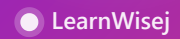


# Charts, Dashboards, Content, Media, and Documents

Lesson Guide · Lab / Exam Guide



## Lesson Guide

ChartJS datasets, labels and options; gauges and ProgressBar; PdfViewer, HtmlPanel, Upload and GoogleMaps with the browser/server boundary in mind; dashboards refreshed as one model. What this means for a Wisej.NET developer — and how to read this module.



### Production focus

ChartJS datasets, labels and options; gauges and ProgressBar; PdfViewer, HtmlPanel, Upload and GoogleMaps with the browser/server boundary in mind; dashboards refreshed as one model.

## 1. The production problem

Dashboards are not just charts. A dashboard is a control composition that turns data into decisions, and the Operations Console dashboard tab will be the screen a service manager opens first every morning. Wisej.NET gives you native content and media controls plus extensions for charts, maps, document viewing and gauges; the intermediate skill is knowing what belongs in the Wisej.NET control model, what is better delegated to a specialised JavaScript library wrapped as an extension or Widget, and how to keep the whole screen refreshing as one unit instead of a pile of independently reloading visuals.

## 2. Key concepts

| Concept                  | Why it matters                                                                                                                                                                                      |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| dashboard                | A composition of controls whose only job is to answer a decision question; if a visual does not help someone decide something, it is clutter.                                                       |
| ChartJS extension        | A NuGet package that exposes Chart.js as a server-side component: the server owns the chart state, the browser renders the visual, and the chart joins the same event model as every other control. |
| labels and DataSets      | The two halves of chart data; labels name the points along the axis and each DataSet carries one series, so they are always set together.                                                           |
| dashboard view model     | A display-shaped object such as <code>DashboardModel</code> that holds months, counts and a completion value, separate from any database entity and from the code that draws it.                    |
| browser boundary         | The client is the browser and the server is where your Wisej.NET code runs; files, PDFs, HTML and map tiles all cross that line in one direction or the other.                                      |
| Upload versus local path | Upload moves a file from the browser to the server; it is not the same as reading a desktop path, because the server can never browse the user's disk.                                              |
| model-based refresh      | One service call returns a complete dashboard model and one method updates every control from it, in a batch, with a visible timestamp.                                                             |

## 3. Lesson narrative

### Start from the question, then choose the control

Every control on the dashboard tab must match the question the user is asking. A chart shows a pattern: use a line chart for a trend over time, a bar chart to compare categories, and a scatter or bubble chart for a relationship between two values. Reserve pie and doughnut charts for a small part-to-whole relationship; with a large category set the slices become unreadable and the chart stops answering anything.

Do not use a chart because it looks good. When exact record-level information matters, a `DataGridView` is the better control, and the Orders grid from Module 5 already exists for that. When the user needs a single status value, such as the completion percentage of this month's tickets, a `ProgressBar` or a gauge-like indicator is clearer than any chart. When the question is geographic, a map answers it. The decision rules are short: charts for patterns, grids for records, gauges and progress for single values, maps for geography.

### ChartJS: options, labels and DataSets on the server

ChartJS is the common choice for line, bar, horizontal bar, radar, polar area, pie, doughnut, bubble and scatter charts. You add the extension to `OperationsConsole` through NuGet and then drop it on the tab like any other component. The chart is configured through its options, its `Labels` and its `DataSets`; the important distinction is that the server owns the component state while the browser renders the visual, which is why a chart click can raise a normal server-side event and why a chart can be rebuilt from C# without any JavaScript.

Chart data should be separated from chart rendering. Create a view model such as `DashboardModel` with the months, the opened and closed counts and the completion percentage, and convert it to labels and `DataSets` in exactly one place. Do not spread chart construction across unrelated click handlers. Labels are part of the data, not optional decoration: when they are missing the chart may not display its points as expected, so the lab sets labels and `DataSets` together in the same method. Configure axes, tooltips, legends, minimum and maximum values and colours intentionally rather than accepting whatever the defaults produce.

```
private void RefreshDashboard(DashboardModel model)
{
    this.chartTickets.Labels = model.Months.ToArray();
    this.chartTickets.DataSets.Clear();
    var opened = this.chartTickets.DataSets.Add("Opened");
    opened.Data = model.Opened.Cast<object>().ToArray();
    var closed = this.chartTickets.DataSets.Add("Closed");
    closed.Data = model.Closed.Cast<object>().ToArray();

    this.progressCompletion.Value = model.CompletionPercent;
    this.pdfPreview.PdfSource = model.PreviewDocumentUrl;
    this.lblLastRefreshed.Text = $"Last refreshed {DateTime.Now:HH:mm:ss}";
}
```

## Content and media controls at the browser boundary

The content controls solve content problems. `PdfViewer` displays a PDF document in the browser using the supported viewer options. `HtmlPanel` embeds a full HTML document, while `WebBrowser` and `IFramePanel` embed external browser content. `Upload` moves files from the browser to the server. `PictureBox` displays image formats, `ImageList` supplies shared images to lists and trees, and `Canvas`, `Shape` and `Line` cover custom drawing and simple visual elements. Use each of them with a clear browser-boundary model in your head: the client is the browser, the server is where the Wisej.NET code runs, and every one of these controls exists to move content across that line safely.

Two boundary mistakes appear in almost every first dashboard. The first is treating `Upload` as if the server could browse the user's local disk; the server only ever sees the bytes the browser chose to send, so the workflow is select, upload, then process on the server. The second is embedding arbitrary HTML in an `HtmlPanel` without sanitising or constraining it, which turns a preview into an injection surface. These viewers are content boundaries, not a way to avoid designing the application.

`GoogleMaps` is a good example of a specialised extension. It is controlled as a server component, supports map types, markers, options, shapes, routes and events such as map clicks or marker drags, and it requires a Google Maps API key configured on the component. That is the core Wisej.NET integration promise: an external browser capability wrapped in a server-side component that participates in ordinary application events. In the lab a map or content placeholder is enough; the point is to reserve the slot and document the boundary.

## One refresh, one model, no clutter

Dashboards need performance rules. Avoid refreshing every visual control independently, because five controls with five timers produce five round trips and five moments where the screen disagrees with itself.

Build a refresh service that returns a complete dashboard model, then update the controls in a batch from one method such as `RefreshDashboard`. Do not load a huge chart data set when an aggregate will answer the question; the service should return twelve monthly totals, not the raw ticket table.

If the dashboard updates in real time, update the minimum number of controls and show a timestamp so the user knows how fresh the numbers are. Use loading states, and disable the actions that cannot be used while a refresh is running. The habit that ties the module together is a short checklist you can apply to any dashboard screen:

- Start from the question the user needs answered, and keep chart data in a view model separate from rendering code.
- Set chart labels and DataSets together; use PdfViewer, HtmlPanel, WebBrowser and IFramePanel for content boundaries; use Upload and Download patterns for browser file transfer.
- Refresh the dashboard as a model, not as unrelated controls, and stamp the result with the refresh time.



### Common pitfalls

Pie charts for large category sets; chart data updated without labels; Upload treated as a way to read the user's local disk; arbitrary HTML embedded without sanitising or constraining it; and dashboards that are pretty but do not answer a decision question.



### 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:

- ✓ Select a chart, gauge, progress, document, media, or content control based on the user's information need.
- ✓ Use ChartJS datasets, labels, options, and chart types for business dashboards.
- ✓ Use PdfViewer, HtmlPanel, PictureBox, Upload, ProgressBar, Canvas, and GoogleMaps with browser/server boundaries in mind.
- ✓ Design dashboards that update from server data without becoming visual clutter.

## 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 dashboard tab: a ChartJS line or bar chart whose labels and DataSets come from a dashboard view model, a ProgressBar or gauge-like indicator for one status value, a PdfViewer or HtmlPanel content preview, an Upload workflow, a map or content placeholder, and one refresh method that updates every dashboard control from a single model.

## Starting point

Open the `OperationsConsole` solution you have carried through Modules 1 to 5 and select the Dashboard page of the TabControl shell; it is still the placeholder from Module 3. Keep this module's work in its own namespace or folder, for example `OperationsConsole.Dashboard`, so the capstone can reuse it, and add the ChartJS extension package through NuGet before you open the designer.

## Suggested approach

- ✓ Write the question the dashboard answers ("how is ticket throughput trending, and how close are we to this month's target?"), then define a `DashboardModel` record with months, opened and closed counts, a completion percentage and a preview document reference, and a `DashboardService` that returns it as aggregates.
- ✓ Drop a ChartJS control named `chartTickets` on the tab, choose line or bar for the trend, and set its axes, legend and colours deliberately; fill `Labels` and `DataSets` together from the model, never from a click handler.
- ✓ Add a `ProgressBar` or gauge-like indicator named `progressCompletion` for the single completion value, and a `PdfViewer` or `HtmlPanel` named `pdfPreview` or `htmlPreview` for the content preview, constraining any HTML you embed.
- ✓ Add an `Upload` control with allowed file types and a size limit, handle its uploaded event on the server, store the file through a service, then update the preview from the result; reserve a map or content placeholder panel and note the GoogleMaps API key requirement in a comment.
- ✓ Write one `RefreshDashboard(DashboardModel)` method that updates the chart, the indicator, the preview and a `lblLastRefreshed` stamp, called from a thin `btnRefresh_Click` that shows a loading state, disables the refresh button and handles the failure path.

## Required deliverables

- ✓ Dashboard tab with a ChartJS line or bar chart
- ✓ Chart labels and DataSets built from a dashboard view model
- ✓ ProgressBar or gauge-like indicator for one status value
- ✓ PdfViewer or HtmlPanel content preview plus an Upload workflow
- ✓ One refresh method that updates every dashboard control

## Student review questions

- ✓ Which control on your dashboard tab is doing the most important work, and what question would go unanswered if you removed it?
- ✓ What would break first if the ticket data set became ten times larger, and why does returning aggregates from the service protect the chart?
- ✓ Where does the uploaded file exist at each step of the Upload workflow, and why can the server never open a path the user typed in?



### 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.