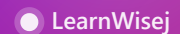


From WinForms to the Web — Lesson Book

The complete set of module readings for this course, in one document.



Contents

01	Migration Discovery and Risk Mapping	Module 1 · Lesson Guide
02	Migration Discovery and Risk Mapping	Module 1 · Lab / Exam Guide
03	Project Conversion and Wisej.NET Startup	Module 2 · Lesson Guide
04	Project Conversion and Wisej.NET Startup	Module 2 · Lab / Exam Guide
05	Forms, Navigation, Layouts, and Modal Workflow	Module 3 · Lesson Guide
06	Forms, Navigation, Layouts, and Modal Workflow	Module 3 · Lab / Exam Guide
07	Sessions, Statics, User Context, and Multi-User Safety	Module 4 · Lesson Guide
08	Sessions, Statics, User Context, and Multi-User Safety	Module 4 · Lab / Exam Guide
09	Data Access, DataGridView, Validation, and Performance	Module 5 · Lesson Guide
10	Data Access, DataGridView, Validation, and Performance	Module 5 · Lab / Exam Guide
11	Files, Reports, Office Automation, and Browser Boundaries	Module 6 · Lesson Guide
12	Files, Reports, Office Automation, and Browser Boundaries	Module 6 · Lab / Exam Guide
13	Modernization, Responsive UI, Security, Deployment, and Capstone	Module 7 · Lesson Guide
14	Modernization, Responsive UI, Security, Deployment, and Capstone	Module 7 · Lab / Exam Guide

Migration Discovery and Risk Mapping

Assess a WinForms application and produce a realistic migration backlog before changing code. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Assess a WinForms application and produce a realistic migration backlog before changing code.

Why this module matters

Desktop-to-web migration changes the runtime environment more than it changes the C# language. Students learn to separate portable business logic from desktop assumptions, identify risky dependencies, and choose a migration sequence.

Core reading

A successful WinForms-to-web migration begins before code is changed. Legacy desktop systems often contain years of business decisions in event handlers, validation code, reports, grids, and small helper classes. Rewriting from scratch can look cleaner at the start, but it usually converts known business behavior into a long rediscovery project. A Wisej.NET migration gives the team another path: keep the event-driven C# model and move the user interface to a browser-native runtime.

The assessment phase has two deliverables: an application inventory and a migration backlog. The inventory lists screens, controls, third-party dependencies, reports, resources, database dependencies, local files, registry keys, Office automation calls, shell/process calls, printers, background jobs, and global state. The backlog converts the inventory into work items with an effort category and a risk category.

Use four categories for each screen or subsystem. Direct-port items mostly use standard WinForms controls and ordinary event handlers. Port-with-adaptation items are compatible but contain desktop assumptions such as local file paths, registry access, printing, or static user state. Redesign items depend on a desktop-only component or are so crowded that a browser-first layout is better. Deferred items are rare paths, admin-only tools, or features that can remain outside the first web release.

The first migrated slice should be intentionally small but representative. It should include application startup, user/session context, navigation, one data-bound grid, one modal editor, one report or export, and one file operation. This slice exposes the hard problems early and prevents a false sense of progress caused by porting only easy forms.

Migration Assessment Template

- ✓ Area
- ✓ Questions to answer
- ✓ Decision
- ✓ Forms and controls
- ✓ How many screens? Standard controls? Custom controls? Third-party controls?

- ✓ Direct port / adapt / redesign.
- ✓ Business logic
- ✓ Is logic mixed into UI event handlers? Can it be extracted or reused as-is?
- ✓ Reuse / extract / test.
- ✓ Data access
- ✓ ADO.NET, EF, typed datasets, direct SQL in events, long-running queries?
- ✓ Port / optimize / service layer.
- ✓ State
- ✓ Statics, singleton services, current user, current company, current connection?
- ✓ Global cache / session / database.
- ✓ Desktop boundary
- ✓ Files, registry, Office COM, printers, shell commands, Windows APIs?
- ✓ Replace with web-safe pattern.
- ✓ Deployment
- ✓ IIS, Docker, Linux, Windows-only dependency, licensing, scaling?
- ✓ Target environment and constraints.



Learning objectives

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

- ✓ Inventory forms, controls, third-party components, resources, data access, reports, Office automation, registry use, file system use, and static state.
- ✓ Classify each screen as direct-port, port-with-adaptation, redesign, or defer.
- ✓ Choose a vertical slice that proves startup, login, navigation, data, modal workflow, files, and reporting early.
- ✓ Document non-negotiable acceptance criteria: parity first, modernization second, deployment third.



Key migration facts

- ✓ A WinForms-to-Wisej.NET migration should begin with an inventory and risk-based backlog, not with random namespace replacement.
- ✓ The first vertical slice should prove startup, navigation, data, modal workflow, file/report behavior, and session context.
- ✓ Migration decisions should be categorized as direct port, adapt, redesign, defer, or remove.

Migration Discovery and Risk Mapping

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Create a migration assessment workbook for the WinForms application you are migrating (LegacyOrderDesk in the course examples) and rank its screens by effort and risk.

Lab steps

- ✓ Open the WinForms application you are migrating, or any small WinForms app with a login, a list screen, and a report.
- ✓ Create an inventory of forms, controls, reports, resources, data sources, files, registry keys, and static fields.
- ✓ Tag each item as direct-port, adapt, redesign, defer, or remove.
- ✓ Choose the first vertical slice and write acceptance criteria.
- ✓ Commit the assessment workbook to the migration branch.

Project Conversion and Wisej.NET Startup

Convert or recreate the project shell so a migrated form can run in the browser. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Convert or recreate the project shell so a migrated form can run in the browser.

Why this module matters

The initial port establishes the ASP.NET/ASP.NET Core hosting model, Wisej.NET package references, default startup files, resource behavior, and the entry point that replaces the WinForms Application.Run pattern.

Core reading

The first technical milestone is a Wisej.NET project that can host the migrated application. There are two common routes. One route starts with a new Wisej.NET project and copies forms, resources, classes, and references into it. The other route modifies the existing project file to become a web application. The safer training route is to create a branch, create a new Wisej.NET project, and bring over one form at a time until the patterns are understood.

A Wisej.NET startup contains browser-facing files and a server-side entry point. Default.html loads the client script and page shell. Default.json tells Wisej.NET which server-side method or main view to start. Program.cs contains the server-side startup method and configures Wisej.NET in the ASP.NET pipeline. A converted WinForms Program.Main cannot call Application.EnableVisualStyles or Application.Run in the same way; instead it shows the first Wisej.NET form or page from the configured startup method.

The largest mechanical change is namespace replacement. Standard WinForms references such as System.Windows.Forms.Button become Wisej.Web.Button. This is intentionally familiar, but not every property or method maps perfectly. The migration workflow is to replace namespaces, build, classify compiler errors, fix obvious differences, and record anything that is not obvious. Do not randomly edit designer files without source control and backups.

Version Note

Wisej.NET migration documentation historically shows Wisej-3 examples and mixed target frameworks such as net48 plus a .NET target. For a new course project, use the current organization standard. For Wisej.NET 4, the NuGet package is Wisej-4 and current .NET projects should follow the Wisej.NET 4 guidance, including resource manager changes when applicable.

Minimal Startup Example

```
using Wisej.Web;
namespace LegacyOrderDesk.Web
{
    internal static class Program
    {
        public static void Main()
        {
            Application.MainPage = new MainPage();
        }
    }
}
```

Namespace Replacement Pattern

```
// Before
using System.Windows.Forms;

// After
using Wisej.Web;

// Typical class declaration remains familiar:
public partial class CustomerForm : Form
{
    public CustomerForm()
    {
        InitializeComponent();
    }
}
```



Learning objectives

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

- ✓ Create a new Wisej.NET project or convert the original project file using a controlled branch.
- ✓ Understand Default.html, Default.json, Program.cs, startup methods, and mainWindow startup options.
- ✓ Replace System.Windows.Forms references with Wisej.Web and resolve compiler errors systematically.
- ✓ Adapt app.config connection strings, embedded resources, designer-generated files, and package references.



Key migration facts

- ✓ Wisej.NET startup uses browser-facing files and server-side startup configuration rather than the classic WinForms executable startup model.
- ✓ Standard WinForms controls generally move from `System.Windows.Forms` to `Wisej.Web`, but properties and methods still need compiler-guided review.
- ✓ Connection strings and configuration from `app.config` must be adapted to the Wisej.NET web project configuration.

Project Conversion and Wisej.NET Startup

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Migrate the first form into a Wisej.NET app shell, run it, and document each compiler error category.

Lab steps

- ✓ Create a migration branch.
- ✓ Create a new Wisej.NET project or convert the shell according to team standard.
- ✓ Copy the first form and its dependencies.
- ✓ Replace System.Windows.Forms with Wisej.Web.
- ✓ Build, classify compiler errors, fix direct differences, and run the app.
- ✓ Document remaining unsupported or unclear differences.

Forms, Navigation, Layouts, and Modal Workflow

Preserve the WinForms user workflow while adapting desktop UI patterns to browser behavior. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Preserve the WinForms user workflow while adapting desktop UI patterns to browser behavior.

Why this module matters

A migrated Wisej.NET application can keep familiar forms, event handlers, menus, MDI concepts, and modal dialogs. The work is to remove desktop-only assumptions and use Wisej.NET designer/layout capabilities where the browser creates new constraints.

Core reading

WinForms developers expect screens to have forms, controls, events, modal dialogs, menus, toolbar buttons, and predictable focus order. Wisej.NET preserves much of this model. The browser changes only the deployment and rendering environment, not the fact that C# event handlers are executed on the server. This is the main reason migration can preserve so much existing code.

Layout conversion is a balance between parity and improvement. If a form already uses anchors and docking well, it may port cleanly. If it was designed for a fixed 1024x768 desktop resolution, make it work first, then modernize using containers, docking, responsive properties, and client profiles. Do not let early design perfection block the first working slice.

Modal workflow is a major advantage for business applications because code can display a dialog and continue after the user closes it. The migration trap is object lifetime. Desktop code often relies on forms being disposed automatically or by process termination. In the web app, closed dialogs that are not reused should be disposed intentionally. The simplest rule is: create transient dialogs in a using statement or call Dispose after ShowDialog returns.

```
private void editButton_Click(object sender, EventArgs e)
{
    var customer = GetSelectedCustomer();
    using (var dialog = new CustomerEditDialog(customer))
    {
        if (dialog.ShowDialog() == DialogResult.OK)
        {
            SaveCustomer(dialog.Customer);
            MessageBox.Show("Customer saved.");
            ReloadGrid();
        }
    }
}
```

Not every MessageBox should remain a modal MessageBox. Confirmation, validation, and dangerous operations may still need blocking dialogs. Informational “saved successfully” messages are better candidates for MessageBox or Toast because they reduce clicks and feel more modern.



Learning objectives

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

- ✓ Map WinForms Form, UserControl, menu, toolbar, tab, and MDI patterns to Wisej.NET equivalents.
- ✓ Use docking, anchoring, layout containers, document outline, tab order, and designer profiles to preserve usability.
- ✓ Use modal dialogs safely and dispose closed dialogs when they are not reused.
- ✓ Replace confirmation MessageBox patterns with MessageBox or Toast when no blocking response is needed.



Key migration facts

- ✓ Wisej.NET supports modal workflow, but transient dialogs should be disposed when closed if they are not reused.
- ✓ Informational MessageBoxes are candidates for MessageBox or Toast; blocking decision dialogs can remain modal.
- ✓ Docking, anchoring, containers, document outline, and client profile preview help preserve and improve WinForms layouts.

Forms, Navigation, Layouts, and Modal Workflow

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Port navigation and three screens, then replace one unnecessary `MessageBox` with a non-blocking notification.

Lab steps

- ✓ Port the main shell and navigation.
- ✓ Port a modal edit dialog.
- ✓ Verify dialog return values and disposal.
- ✓ Adjust docking, anchoring, and tab order.
- ✓ Replace one informational `MessageBox` with `AlertBox` or `Toast`.
- ✓ Record before/after screenshots.

Sessions, Statics, User Context, and Multi-User Safety

Make the migrated application safe for many simultaneous browser sessions. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Make the migrated application safe for many simultaneous browser sessions.

Why this module matters

Most WinForms applications were written as one process per user. A web application serves many users in one server process. Static fields, cached user objects, registry settings, open files, and global flags must be reviewed and moved to session, database, or safe shared services.

Core reading

A WinForms application usually runs one process per user. Static variables therefore often appear safe because only one user is inside that process. In a web application, many sessions can run inside the same server process. A static `CurrentUser`, `CurrentCompany`, `CurrentConnection`, or `CurrentOrder` is now shared unless refactored. This is one of the most important migration reviews.

Not all statics are wrong. Static lookup data such as country lists, currency symbols, or immutable configuration can remain global if it is thread-safe and independent of a specific user. Per-user state must move to a session, user context service, database row, or other scoped storage. The key question is: would this value be different if two users opened the app at the same time? If yes, it is not a safe global static.

Session Context Pattern

```
public sealed class UserSessionContext
{
    public int UserId { get; set; }
    public string Username { get; set; }
    public int CompanyId { get; set; }
    public string Culture { get; set; }
}

public static class SessionContext
{
    public static UserSessionContext Current
    {
        get
        {
            if (Application.Session.UserContext == null)
                Application.Session.UserContext = new UserSessionContext();
            return Application.Session.UserContext;
        }
    }
}
```

Registry access requires a similar decision. In a web app, server code accesses the server registry, not the user's desktop registry. User preferences should typically become database-backed settings, server-side profile files, or browser storage. Browser storage is useful for client-specific UI preferences, but important business settings should be server-owned.

Session timeout and cleanup must be planned. A browser tab can be abandoned, a device can sleep, and a network can drop. Long-running transactions, temporary files, report jobs, and locks must be released either when the user completes the workflow or during session cleanup.



Learning objectives

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

- ✓ Distinguish safe global statics from per-user/session state.
- ✓ Replace desktop registry settings with database rows, server-side user settings, or browser storage where appropriate.
- ✓ Understand session lifecycle, keep-alive, timeout, `Application.Session`, and cleanup hooks.
- ✓ Create a user context service and a static-state audit pattern for migrated code.



Key migration facts

- ✓ Static fields that contain per-user state are unsafe in a multi-user web application.
- ✓ Immutable global lookups may remain static if they are thread-safe and not user-specific.
- ✓ Registry access in server code targets the server registry, so user settings need database, server profile, or browser-storage replacements.

Sessions, Statics, User Context, and Multi-User Safety

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Find and refactor shared static user state in your WinForms application into a session-aware context service.

Lab steps

- ✓ Search for static fields and singleton state.
- ✓ Classify each static as immutable shared data or per-user state.
- ✓ Create UserSessionContext.
- ✓ Move current-user/current-company/current-filter values into session context.
- ✓ Test two browser sessions with different users.
- ✓ Document cleanup requirements for timeout and logout.

Data Access, DataGridView, Validation, and Performance

Port the data-heavy parts of a WinForms application without creating slow or chatty browser screens. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Port the data-heavy parts of a WinForms application without creating slow or chatty browser screens.

Why this module matters

Data-bound WinForms screens often port well, but the browser changes perceived latency, data volume, and interaction patterns. Wisej.NET supports WinForms-style data binding, powerful DataGridView capabilities, virtual rows, server events, and real-time updates; students learn when to preserve and when to redesign.

Core reading

Most data access code can be reused if it does not assume a desktop-only connection lifetime or block the UI with large operations. The browser adds network latency and scaling concerns, but Wisej.NET still lets server-side C# handle events and update controls. The goal is to reduce unnecessary data transfer and avoid loading giant datasets into controls just because the old desktop app did.

DataGridView is often the most important control in a migration. Wisej.NET DataGridView supports familiar data binding and adds web-scale features such as virtual rows, grid-hosted controls, tool buttons, custom cell editors, HTML content, painting, frozen rows, row spans, column spans, and summary rows. Use these features to preserve dense business screens while improving performance.

A ported grid should be tested with production-like row counts. A grid that works with 100 rows in a lab may not work acceptably with 500,000 rows. Use server-side filtering, lazy loading, virtual rows, or summary views. Only send what the user can reasonably inspect or act on.

Virtual Row Thinking

Do not bind every row if the user only sees a page-sized viewport.

Keep the filter/sort model server-side when the dataset is large.

Cache immutable lookup data, but do not cache user-specific row permissions globally.

Measure payload size and event frequency using browser developer tools.

Treat live updates as a business feature, not a default behavior for every grid.

Validation Pattern

```
private bool ValidateOrder(Order order)
{
    errorProvider1.Clear();
    if (order.CustomerId == 0)
    {
        errorProvider1.SetError(customerComboBox, "Select a customer.");
        return false;
    }
    if (order.Lines.Count == 0)
    {
        MessageBox.Show("Add at least one order line.");
        return false;
    }
    return true;
}
```



Learning objectives

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

- ✓ Port data binding while keeping business rules and validation on the server.
- ✓ Use DataGridView virtual rows or paging-like patterns for large datasets.
- ✓ Avoid loading unnecessary data into controls just because the desktop app did.
- ✓ Use WebSocket/server push only when the screen genuinely benefits from live updates.



Key migration facts

- ✓ DataGridView screens must be tested with realistic row counts and optimized with server-side filtering, lazy loading, or virtual rows when needed.
- ✓ Validation should remain in server/business code and display clear UI feedback.
- ✓ Live updates should be added when useful, not applied indiscriminately to every control.

Data Access, DataGridView, Validation, and Performance

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Port a customer/order grid, add validation, and replace full-table loading with an incremental or virtualized pattern.

Lab steps

- ✓ Port the customer/order grid.
- ✓ Add validation to the edit workflow.
- ✓ Test with a large row count.
- ✓ Replace full-table loading with server-side filter or virtualized access.
- ✓ Measure initial load and common interaction time.
- ✓ Record performance notes in the migration log.

Files, Reports, Office Automation, and Browser Boundaries

Replace desktop file, registry, report, and Office assumptions with browser/server-safe workflows. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Replace desktop file, registry, report, and Office assumptions with browser/server-safe workflows.

Why this module matters

The browser is not the desktop. Code that reads C:\Temp, writes HKCU, automates Excel, opens local files, or expects a local printer is now server code unless explicitly mediated. Students replace these patterns with upload/download, server storage, PDF rendering, server-safe document libraries, queues, and ClientFileSystem where appropriate.

Core reading

The browser is not the user's desktop. When migrated server code calls `File.ReadAllText`, `Directory.GetFiles`, `Registry.CurrentUser`, `Process.Start`, or `Excel.Application`, that code runs in the server environment unless a client-mediated feature is explicitly used. These areas require deliberate redesign.

File handling has several patterns. Upload moves a client file to the server for processing. Download delivers a generated server file to the user. Server storage keeps documents in a controlled location or database. ClientFileSystem-style interaction can be used when the application must interact with client files more directly and the platform permits it. Linux deployments also require attention to case-sensitive paths and forward slashes.

Reports should usually be generated server-side as PDFs and displayed with PDFViewer or offered as downloads. If the reporting engine cannot handle heavy parallel use, introduce a queue. A queue also improves user experience for long reports because the UI can show job status instead of freezing.

Office Automation must be replaced. Installing Office on a server and automating Excel or Word from IIS may appear easy, but it is not server-safe. Use libraries designed for unattended document generation and manipulation, such as EPPlus, Aspose, Syncfusion, DevExpress, Open XML SDK, or another approved library depending on license and document requirements.

```
private void exportButton_Click(object sender, EventArgs e)
{
    var rows = orderService.Search(currentFilter);
    var path = reportService.CreateOrdersWorkbook(rows);
    // In a real app, use the organization's approved download helper.
    Application.Download(path, "Orders.xlsx");
}
```



Learning objectives

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

- ✓ Classify file operations as server-only, client-mediated, upload/download, or unsupported without redesign.
- ✓ Replace registry use with database/user settings/browser storage.
- ✓ Convert report workflows to server-side generation with PDF output and PDFViewer or downloads.
- ✓ Replace Office Interop/Automation with managed, server-safe document APIs.



Key migration facts

- ✓ Migrated file code runs on the server unless upload/download or client file system mediation is used.
- ✓ Office Automation should be replaced with server-safe document libraries in web/server deployments.
- ✓ Reports should usually generate server-side documents such as PDFs and display or download them through browser-safe workflows.

Files, Reports, Office Automation, and Browser Boundaries

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Replace a desktop export-to-Excel workflow with a server-generated file and browser download; add a queued report job.

Lab steps

- ✓ Find one file import/export workflow and one report workflow.
- ✓ Replace local file assumptions with upload/download or server storage.
- ✓ Convert report output to PDF or generated document download.
- ✓ Add a simple job queue for long reports.
- ✓ Remove direct Office Automation from the migrated path.
- ✓ Test on a clean server-like environment.

Modernization, Responsive UI, Security, Deployment, and Capstone

Turn the working port into a polished, secure, deployable Wisej.NET web application. Why it matters, the core reading, and the learning objectives — read this before the lab.



Migration focus

Turn the working port into a polished, secure, deployable Wisej.NET web application.

Why this module matters

After the first successful port, teams can modernize without losing project control. Students apply theming, responsive properties, client profiles, HTML formatting carefully, security review, deployment packaging, and a final readiness checklist.

Core reading

A first successful port is not the end. It is the moment when modernization becomes safer. Now the team can improve selected areas without destabilizing the entire application. Wisej.NET theming, responsive properties, client profiles, enhanced controls, and web-native notifications let the team modernize incrementally.

Responsive behavior is a migration topic because most WinForms screens were designed for fixed resolutions. Wisej.NET client profiles let the application associate different property values with different devices and screen sizes. A control can be visible on desktop and hidden on phone, a button can display both icon and text on desktop and only an icon on smaller screens, and a screen can move controls without maintaining separate codebases.

Security review must include both old and new risks. Old risks include SQL injection, insecure storage, and over-permissive roles. New web-facing risks include exposing server operations to a browser, using AllowHtml with user-entered values, downloads of sensitive files, and session/authentication mistakes. Wisej.NET encodes text by default, but AllowHtml and HTML-enabled message content must be treated carefully.

Deployment should be planned early. Know whether the target is IIS on Windows, Linux, Docker, or a cloud platform. Confirm package versions, connection strings, file storage paths, report font dependencies, temp-folder permissions, logging, health checks, HTTPS, session timeout, and load-balancing/session-affinity requirements.

Final Readiness Checklist

- ✓ Application starts through Wisej.NET startup files and shows the correct main view.
- ✓ No per-user state remains in unsafe static fields.
- ✓ All registry, local file, Office Automation, and process-launch assumptions are reviewed.
- ✓ Large grids are tested with production-like row counts.
- ✓ Transient dialogs are disposed.

- ✓ AllowHtml is used only with trusted or sanitized content.
- ✓ Responsive behavior is tested at desktop, tablet, and phone sizes.
- ✓ Deployment configuration, secrets, logs, temp paths, and health checks are documented.



Learning objectives

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

- ✓ Apply themes and mixins without rewriting screens.
- ✓ Use ClientProfiles.json and responsive properties to adapt a fixed desktop UI to tablets and small screens.
- ✓ Apply security rules for AllowHtml, user input, authentication, and session validation.
- ✓ Prepare deployment to IIS, Linux, Docker, or cloud with configuration, logging, and load-balancing notes.



Key migration facts

- ✓ Modernization should follow a working parity baseline.
- ✓ Client profiles and responsive properties let a migrated fixed-resolution screen adapt to device/browser conditions.
- ✓ AllowHtml should be used only for trusted or sanitized content because default text encoding is safer for user input.

Modernization, Responsive UI, Security, Deployment, and Capstone

What you'll build in the hands-on lab, the lab steps, and how to verify the migrated workflow.



Hands-on lab

Finish the capstone: migrate the app, modernize one screen, run security/deployment checks, and present the migration report.

Lab steps

- ✓ Apply a theme or mixin.
- ✓ Create or update ClientProfiles.json.
- ✓ Set responsive properties for the migrated screen.
- ✓ Review AllowHtml usage and sanitize or disable where needed.
- ✓ Prepare deployment configuration and publish profile.
- ✓ Demo the capstone and submit the migration report.