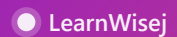


Thèmes, ressources, localisation et modernisation de l'UI

Modernisez les applications Wisej.NET grâce aux thèmes, à CSSStyle, aux ressources, aux icônes, au changement de thème à l'exécution, à la localisation et à des règles de cohérence visuelle.



Objectif de la leçon

Moderniser les applications Wisej.NET grâce aux thèmes, à CSSStyle, aux ressources, aux icônes, au changement de thème à l'exécution, à la localisation et à des règles de cohérence visuelle — en séparant l'*apparence* (thème/CSS) du *comportement* (C#), pour qu'une refonte ou une nouvelle langue n'oblige jamais à modifier chaque écran.

Tout ce module repose sur une discipline : **l'apparence et la logique sont deux métiers différents et vivent à des endroits différents**. Votre C# décide de *ce que fait l'application* ; un thème décide de *son apparence* ; les ressources contiennent les *chaînes et images* qu'elle affiche. Tant que ces trois éléments restent séparés, vous pouvez restyler le produit, ajouter un mode sombre ou livrer une version italienne sans toucher à un seul gestionnaire d'événements. Dès qu'ils se mélangent — une couleur hexadécimale saisie dans un contrôle ici, un libellé codé en dur là — chaque changement devient une chasse à travers tout le code. Cette leçon explique comment les garder délibérément séparés.

1. Séparer l'apparence du comportement

Dans Wisej.NET, l'*apparence* d'un écran est une donnée, pas du code. Un **thème** est un document JSON qui définit les couleurs, les polices, les espacements, les bordures et l'apparence de chaque type de contrôle — boutons, grilles, panneaux, onglets. Votre C# ne mentionne jamais `#0d6efd` ni `font-size: 14px` ; il crée simplement un `Button`, et le thème actif décide de la façon dont ce bouton est dessiné. Changez de thème et toute l'application change d'habillage, car les contrôles n'ont jamais porté leurs propres couleurs.

Couche	Ce dont elle est responsable	Où elle vit
Thème (JSON)	L'apparence globale — la palette de couleurs, les polices, l'échelle d'espacement et l'apparence par défaut de chaque type de contrôle.	Un fichier <code>.theme</code> / JSON de thème, ou Theme Builder.
CSSStyle	Une surcharge visuelle ciblée appliquée à un contrôle ou à une petite famille de contrôles.	La propriété <code>CssStyle</code> du contrôle (C# ou Designer).
Comportement (C#)	Ce que <i>fait</i> l'application — validation, workflows, données, navigation.	Vos gestionnaires d'événements et vos services.
Ressources	Les chaînes et images affichées — libellés, messages, icônes, logos.	Fichiers de ressources / le dossier Resources.

Centraliser couleurs et espacements dans un thème plutôt que de styler chaque contrôle apporte un effet de levier. Supposons que votre couleur de marque passe du bleu au bleu-vert. Avec un thème, vous modifiez une valeur et tous les boutons, liens et surbrillances se mettent à jour ensemble. Avec un style par contrôle, ce même changement devient un rechercher-remplacer dans des dizaines de fichiers — et vous en oublierez *forcément* un. La centralisation impose aussi une cohérence à laquelle vous n'avez pas à penser : comme chaque contrôle lit ses espacements dans la même échelle, les écrans s'alignent sans que personne ne mesure les pixels à l'œil.



La règle en clair

Si une valeur visuelle décrit *toute votre application* (le bleu de marque, la marge standard), elle appartient au thème. Si elle décrit *un contrôle inhabituel*, elle appartient au CSSStyle de ce contrôle. Elle ne devrait presque jamais apparaître comme littéral en C#.

2. Les thèmes et Theme Builder

Wisej.NET est livré avec un ensemble de thèmes intégrés et fournit **Theme Builder**, un éditeur visuel pour les créer ou les ajuster. Vous partez d'un thème de base — par exemple un thème clair par défaut ou une variante sombre comme **BootstrapDark-4** — vous ajustez la palette, la typographie et l'apparence des contrôles, puis vous enregistrez le résultat comme thème de votre application. Comme Theme Builder modifie le même JSON que celui lu à l'exécution, il n'y a pas d'étape d'« export » séparée où se tromper : ce que vous concevez est ce que l'application charge.

Quelques concepts méritent d'être nommés. Un thème définit des **couleurs nommées** (une palette), si bien que les apparences font référence à `"highlight"` ou `"error"` plutôt qu'à des valeurs hexadécimales brutes — une seule modification de la palette se répercute partout. Les **appearances** sont les définitions de style par type de contrôle. Et les **mixins** permettent de superposer un petit ensemble de surcharges à un thème de base au lieu de le dupliquer entièrement — l'outil idéal quand vous voulez « le thème standard, mais avec notre couleur de marque et notre logo ».

3. Changer de thème à l'exécution

Comme le thème n'est que l'apparence active, vous pouvez le changer pendant l'exécution de l'application — pour une bascule de mode sombre par utilisateur, un mode d'accessibilité à fort contraste ou un habillage propre à un tenant. Vous définissez le thème actif de l'application, et Wisej.NET redessine les écrans ouverts avec les nouvelles apparences. Vous ne reconstruisez pas les contrôles et ne rechargez pas les données ; seul le rendu change.

Une bascule de mode sombre qui change le thème actif à l'exécution

```
private void btnDarkMode_Click(object sender, EventArgs e)
{
    // Pick the theme by name; the runtime re-skins open screens.
    // Behavior does not change – only the active look does.
    var next = _darkMode ? "Default-4" : "BootstrapDark-4";
    Application.Theme = next;           // switch the whole app's look
    _darkMode = !_darkMode;

    // Persist the choice so it survives the next login.
    _userPrefs.Save("theme", next);
}
```

Choisir le thème au démarrage, avant que le premier écran s'affiche

```
protected override void OnSessionStart(EventArgs e)
{
    // Read the saved preference and set the look once, up front,
    // so the user never sees a flash of the wrong theme.
    var saved = _userPrefs.Load("theme") ?? "Default-4";
    Application.Theme = saved;
    base.OnSessionStart(e);
}
```

Remarquez ce que le gestionnaire ne fait *pas* : aucun littéral de couleur, aucun restylage contrôle par contrôle, aucun rechargement de données. L'apparence est une donnée nommée ; en changer tient en une ligne. (La propriété ou l'API exacte pour définir le thème actif peut varier selon la version de Wisej.NET — le mécanisme reste le même : affecter par son nom le thème actif de l'application, et le runtime redessine l'interface.)

4. CSSStyle — l'exception ciblée

Parfois, un contrôle doit réellement se distinguer de ses voisins d'une façon que le thème ne devrait pas encoder globalement — un bandeau d'avertissement avec un fond particulier, une carte avec une ombre douce. C'est le rôle de la propriété **CssStyle** du contrôle : elle attache du CSS à ce widget précis sans polluer le thème ni vous obliger à fouiller vous-même le DOM du navigateur.

```
pnlBanner.CssStyle =  
    "background: var(--wisej-warning, #fff3cd);" + // prefer a theme var over a raw hex  
    "border-radius: 8px;" +  
    "padding: 12px 16px;"  
  
lblBanner.Text = "Your session expires in 5 minutes.";
```

Utilisez CSSStyle avec parcimonie et délibérément. Chaque chaîne CSSStyle est un petit morceau d'apparence qui vit *en dehors* du thème ; elle ne suivra donc pas automatiquement une refonte. L'habitude à prendre : passez d'abord par le thème, et ne descendez à CSSStyle que lorsqu'un contrôle est une véritable exception. Si vous vous surprenez à coller le même CSSStyle sur de nombreux contrôles, c'est le signe qu'il appartient au thème.

Thème, CSSStyle ou propriété en ligne — quand utiliser chacun

Outil	Portée	À utiliser quand...
Thème (JSON / Theme Builder)	Toute l'application, tous les contrôles d'un type.	La valeur fait partie de l'identité du produit — palette, polices, échelle d'espacement, apparence par défaut des contrôles. Premier choix pour tout ce qui est partagé.
Propriété CSSStyle	Un contrôle, ou une petite famille délibérée.	Une véritable exception visuelle que le thème ne doit pas porter — un bandeau particulier, une carte mise en avant.
Propriété en ligne (BackColor, Font...)	Un contrôle, une propriété.	Rarement — un ajustement réellement local et piloté par les données (par exemple une ligne qui devient rouge parce qu'une valeur est en retard). Jamais pour des couleurs de marque.

5. Ressources — images, icônes et chaînes

L'autre moitié de « séparer l'apparence de la logique » consiste à traiter le *contenu* affiché par l'application — ses libellés, messages, logos et icônes — comme des **ressources** plutôt que comme des littéraux dispersés dans le code. Les images et icônes vont dans le dossier Ressources et sont référencées par leur nom, pour qu'un designer puisse changer le logo sans recompiler vos gestionnaires. Les chaînes visibles par l'utilisateur vont dans des fichiers de ressources indexés par nom, ce qui rend précisément la localisation possible : la même clé se résout en un texte différent selon la culture.

Charger un message localisé et une icône depuis les ressources au lieu de les coder en dur

```
// String comes from a resource keyed by name – not a literal.  
// At runtime it resolves to the entry for the current culture.  
lblWelcome.Text = Resources.Strings.Welcome;           // "Welcome" / "Benvenuto" / "Willkommen"  
  
// Image comes from a named resource, not an ad-hoc file path.  
btnSave.ImageSource = "resource.wx/App.Resources.SaveIcon.png";  
  
// A whole Form/Page can be marked Localizable so the Designer  
// stores its captions per culture automatically.
```

La forme générale est toujours la même : une **clé** stable dans le code, résolue en contenu réel à l'exécution selon la culture active. Toute l'astuce tient dans cette indirection — le code référence des clés, et le sens d'une clé peut changer selon la langue, la marque ou la saison sans que le code le sache.

6. Localisation — culture, chaînes et mise en forme

La localisation ne se limite pas à la traduction. Deux choses varient selon la **culture** : les *chaînes* affichées, et la *mise en forme* des dates, nombres et montants. Wisej.NET localise le texte visible en résolvant les clés de ressources pour la culture courante (les Forms et Pages peuvent être marqués **Localizable** pour que le Designer gère leurs libellés par langue). La mise en forme suit la culture du thread : une date s'affiche **7/3/2026** pour un utilisateur et **03/07/2026** pour un autre — à partir du *même* C#, parce que vous avez formaté avec la culture plutôt qu'avec un motif fixe.

Définir la culture de l'utilisateur, puis laisser la mise en forme la suivre

```
// Set the culture for this user's session (from a preference or a picker).  
var culture = new CultureInfo(_userPrefs.Load("culture") ?? "en-US");  
System.Threading.Thread.CurrentThread.CurrentUICulture = culture; // which strings  
System.Threading.Thread.CurrentThread.CurrentCulture = culture; // how dates/numbers  
format  
  
// Now the SAME code formats correctly for every user:  
lblTotal.Text = amount.ToString("C");           // $1,234.50 vs 1.234,50 €  
lblDue.Text = dueDate.ToShortDateString(); // culture-aware, no hard-coded pattern
```

Un avertissement pratique : le texte traduit change de longueur. Les libellés allemands sont souvent plus longs que les anglais ; certaines écritures sont plus hautes. Une mise en page parfaite en anglais peut être coupée ou mal renvoyée à la ligne dans une autre langue ; **testez donc la localisation dans de vraies mises en page d'écran**, pas seulement dans un tableur de chaînes. C'est pourquoi ce module traite la localisation comme une question d'interface, pas seulement de texte.



Réflexe de production

Le thème d'abord, les exceptions ensuite. Gardez chaque chaîne et chaque image visibles comme ressources, définissez la culture une fois par session et formatez avec la culture plutôt qu'avec un motif fixe — puis vérifiez le résultat dans l'écran réel pour chaque langue livrée.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Appliquer des thèmes intégrés et changer de thème à l'exécution ou au démarrage sans modifier le comportement.
- ✓ Utiliser Theme Builder et CSSStyle sans disperser des styles ponctuels partout.
- ✓ Centraliser couleurs, espacements, chaînes et images pour qu'une refonte ne modifie qu'un seul endroit, pas chaque écran.
- ✓ Localiser les Forms, les Pages, les ressources système et la mise en forme des données quand c'est pertinent.
- ✓ Créer une checklist de modernisation de l'interface pour les écrans d'entreprise.

7. Une interface moderne, en bref

Moderniser ne se limite pas aux thèmes. Un écran d'entreprise moderne utilise une **échelle d'espacement** cohérente, une **hiérarchie typographique** claire, un **contraste** accessible, des **icônes** reconnaissables et des **modèles de statut et d'erreur** prévisibles, pour que les utilisateurs apprennent un seul langage visuel et le retrouvent partout. Si tout cela appartient au thème, c'est pour la même raison que précédemment : une cohérence obtenue gratuitement est une cohérence qu'on ne peut pas casser par accident, et une refonte devient une modification de données plutôt qu'un projet.

Erreurs courantes

Erreur	Pourquoi c'est un problème	À faire plutôt
Couleurs et espacements codés en dur, dispersés dans les écrans.	Un changement de marque devient un rechercher-remplacer dans des dizaines de fichiers — et vous en oublierez un.	Définir la palette et les espacements dans le thème ; référencer des valeurs nommées, pas des valeurs hexadécimales.
Coller le même CSSStyle sur de nombreux contrôles.	Le style vit en dehors du thème ; il ne suit pas une refonte et se désynchronise.	Promouvoir le style répété dans l'apparence du thème pour ce type de contrôle.
Chaînes visibles écrites comme littéraux en C#.	Elles ne peuvent pas être traduites, et une correction de texte impose de recompiler.	Garder les chaînes dans des ressources indexées par nom ; les résoudre selon la culture.
Images intégrées au cas par cas depuis des chemins de fichiers.	Les ressources graphiques se dispersent, cassent au déploiement et ne peuvent pas être remplacées sans recompilation.	Les ajouter à Ressources et les référencer par nom.
Ignorer la culture — motifs de date et de nombre fixes.	Dates, décimales et montants s'affichent mal pour les autres régions.	Définir la culture de la session et formater avec elle (<code>ToString("C")</code> , date courte).
Tester les traductions uniquement sous forme de liste de chaînes.	Un texte plus long est coupé ou renvoyé à la ligne ; les mises en page cassent dans l'écran réel.	Tester chaque langue dans la mise en page réelle avant de livrer.

Vocabulaire clé

- **Thème** — le document JSON qui définit l'apparence de toute l'application : palette, polices, espacements et apparences par contrôle.
- **Theme Builder** — l'éditeur visuel pour créer et ajuster des thèmes.
- **Appearance** — la définition de style d'un thème pour un type de contrôle.
- **Mixin** — une petite couche de surcharges superposée à un thème de base au lieu de le dupliquer.
- **CSSStyle** — la propriété d'un contrôle pour une surcharge CSS ciblée sur ce widget précis.
- **Ressource** — une chaîne ou une image nommée que l'application référence au lieu d'un littéral, ce qui permet remplacement et traduction.
- **Propriété Localizable** — un Form/une Page (ou un contrôle) marqué pour que le Designer stocke ses libellés par culture.
- **Culture** — le réglage qui détermine à la fois les chaînes affichées et la mise en forme des dates, nombres et montants.
- **Changement de thème à l'exécution** — changer le thème actif pendant l'exécution, en redessinant les écrans ouverts sans les reconstruire.

Piège courant à éviter



Piège courant

Disperser des couleurs codées en dur, du CSS ponctuel et des chaînes littérales dans de nombreux écrans. Cela fonctionne en démo et pourrit en production — un restylage ou une nouvelle langue impose alors de modifier chaque fichier au lieu d'un thème et d'un jeu de ressources. C'est exactement ce défaut que vous éliminerez dans le lab de ce module.

Checklist de revue du Designer et du code

- ✓ L'écran reste utilisable dans le Designer de Visual Studio / Wisej.NET.
- ✓ Les contrôles sont nommés assez clairement pour qu'un collègue suive le code des événements.
- ✓ Les couleurs et espacements viennent du thème, pas de littéraux en C# ni de CSSStyle répétés.
- ✓ Les chaînes et images visibles sont des ressources, pas des valeurs codées en dur.
- ✓ La culture est définie par session et la mise en forme la suit.
- ✓ La logique métier n'est pas prisonnière des gestionnaires d'événements visuels.
- ✓ Les chemins d'échec sont visibles, journalisés et expliqués sans divulguer de détails internes.
- ✓ Le livrable peut être relu sans exécuter l'ensemble du cours.



Auto-vérification avant le lab

Quels styles relèvent du thème et lesquels sont de véritables exceptions propres à un écran ? Pouvez-vous changer l'apparence de toute l'application en une ligne sans toucher à un gestionnaire ? Chaque chaîne et chaque image visibles sont-elles des ressources ? Qu'est-ce qui change avec un texte allemand ou italien, et la mise en page y résiste-t-elle ? Une future refonte peut-elle éviter de modifier chaque écran ? Si la réponse est oui partout, vous êtes prêt.

Code d'accompagnement

`ThemingLocalizationPatterns.cs` réunit les modèles de ce module dans un seul fichier compilable. Téléchargez-le et lisez-le en parallèle du lab.

↓ [Télécharger ThemingLocalizationPatterns.cs](#)