

logiCells

Vues pour client riche
et contrats d'interaction

Guide d'architecture

Interfaces natives et Web,
vues réactives et événements



Table des matières

À propos de ce guide	1
I Repères et orientation	2
1 Quand utiliser l’architecture View de logiCells	3
1.1 Quand un client plus simple suffit	3
1.2 Ce qui reste commun	3
1.3 Ce qu’apporte cette architecture	3
2 Prérequis et articulation avec le Guide du développeur	4
2.1 Quand consulter le Guide du développeur	4
3 Contrat de référence et état des capacités	5
3.1 Sources faisant autorité	5
3.2 Vocabulaire des statuts	5
3.3 Frontière entre plateformes	5
3.4 Frontière de synchronisation	6
4 Les vues comme contrats d’interaction et projections de concepts	7
4.1 Contrat d’interaction et vue affichée	7
4.2 Projection agentique sans formulaire	8
4.3 Projection, intention et fonction	8
4.4 Déclarations de vues	8
4.5 Vues ClassItem et ObjectItem	8
4.6 Vues et contexte de données	9
4.7 Composants fonctionnels	9
4.8 Composants abstraits	9
4.9 Vues et événements	10
II Packages View et structure du client	11
5 Contrats d’interaction, packages VIEW et artefacts	12
5.1 Organisation canonique des sources	12
5.2 Pourquoi le dossier s’appelle contracts	13
5.3 Liaison explicite de XML vers MODEL	13
5.4 MODEL-behind suit le pipeline MODEL ordinaire	13
5.5 Responsabilité du package VIEW	14
5.6 Compilation incrémentale	15
5.7 Les artefacts VIEW compilés sont des produits d’exécution durables	15
5.8 Contrôles de non-régression	16

6	Créer des vues de tâches avec DataSet et modèles de page	17
6.1	Les trois parties d'une vue	17
6.2	Déclarer le cache de données de page	17
6.3	Initialiser l'état de la page	18
6.4	Filtrer un DataSet avant son ouverture	19
6.5	Actions Enregistrer et Annuler	19
6.6	Lier les contrôles de formulaire aux DataSource	20
6.7	Actions des boutons	20
6.8	Enregistrer les classes MODEL-behind dans le manifeste	20
6.9	Exercice	21
7	Vues, DataSet et curseurs d'objets	22
7.1	Vues comme projections pilotées par le modèle	22
7.2	Curseurs de données	22
7.3	DataSetsManager comme contrat de navigation	22
7.4	Chemins de collections et d'actions	23
7.5	Comportement événementiel des vues	23
7.6	Règle de conception	23
8	Composants fonctionnels intentionnels UXfc	24
8.1	De l'activité utilisateur à l'interface affichée	24
8.2	Composants intentionnels	24
8.3	Composants fonctionnels	25
8.4	Implémentations concrètes	25
8.5	Composants abstraits	25
8.6	Assemblages de composants UXfc	25
8.7	Paramètres non visuels	26
8.8	Coordination visuelle locale	26
8.9	Événements et actions abstraites	26
8.10	Règles de conception	26
9	ObjectBehind : l'objet comportemental d'une View	27
9.1	Modèle mental	27
9.2	Forme source et responsabilité de compilation	27
9.3	Cycle de vie	28
9.4	Trouver un ObjectBehind	28
9.5	Relation avec les DataSet	28
9.6	Flux d'événements	29
9.7	Ce qui relève d'ObjectBehind	29
9.8	Liste de contrôle du diagnostic	29
10	functionalP pour débutants sur logiCells	30
10.1	Où apparaît functionalP	30
10.2	Script minimal dans les métadonnées	30
10.3	Variables et affectation	31
10.4	Self, Sender et Result	31
10.5	Accéder à PageControler	31
10.6	Lire un DataSet	31
10.7	Initialiser un état local de page	32
10.8	Filtrer avant l'ouverture d'un DataSet	32
10.9	Publier l'objet courant	32
10.10	Créer un objet	32

10.11	Émettre un événement visuel	33
10.12	Appeler l'ObjectBehind d'un composant visuel	33
10.13	Quand déplacer la logique ailleurs	33
10.14	Erreurs fréquentes des débutants	34
10.15	Parcours de lecture pratique	34
III	État client, cache et routage des événements	35
11	Cache de données des vues fonctionnelles	36
11.1	Cache de données de la vue	36
11.2	Pourquoi les composants fonctionnels ne doivent pas gérer l'accès aux données .	37
11.3	DataSet, DataSource et DataCursor	37
11.4	Relation avec les collections et fournisseurs	38
11.5	Maître-détail et propagation du contexte	38
11.6	Fournisseurs distants et réactivité locale	39
11.7	Assemblage déclaratif des DataSet	39
11.8	Composants fonctionnels liés au cache de la vue	39
11.9	Relation avec PageControler	40
11.10	Règles de conception	40
12	Événements DataSetManager et vues pilotées par les DataSet	41
12.1	Pourquoi les événements des DataSet sont importants	41
12.2	Un événement DataSetManager n'est pas automatiquement un événement métier	41
12.3	Familles d'événements	42
12.4	Événements observés dans des vues ERP pilotées par DataSet	42
12.5	Avant publication : validation par un événement de DataSet	43
12.6	Après publication : actualiser les vues dépendantes	43
12.7	DataSet de filtres et recherches réactives	43
12.8	Modifications de curseur et vues maître-détail	44
12.9	Paramètres des événements	44
12.10	Correspondance avec le web réactif	45
12.11	Noms d'événements recommandés	45
12.12	Assemblage événementiel déclaratif	46
12.13	Règles de conception	46
13	PageControler et émission locale d'événements visuels	48
13.1	Pourquoi émettre les événements visuels localement	48
13.2	PageControler comme bus d'événements local	48
13.3	PageModel et objectBehind des composants	49
13.4	Vues React et émission côté client	49
13.5	Vues natives et DispatchVisualEvent	50
13.6	Quatre niveaux d'événements	50
13.7	Correspondance déclarative des événements	50
13.8	DataSet, collections et frontières d'exécution	51
13.9	Assemblages UXfc et événements locaux	51
13.10	Règles de conception	51
14	Modèles VIEW événementiels et asynchrones	53
14.1	Penser en continuations, pas en async/await	53
14.2	Abonnement aux événements du contrôleur	53
14.3	Émettre une intention, pas l'implémentation du parent	54

14.4	Cadres d'attente et signaux de fin	54
14.5	Modèle de cycle de vie des DataSet	55
14.6	Racines d'événements et propagation	55
14.7	Garder les contrôleurs VIEW légers	55
14.8	Liste de vérification d'une VIEW asynchrone	55
15	Cadres de bureau réactifs et routage des événements	57
15.1	Des formulaires pilotés par DataSet aux bureaux réactifs	57
15.2	Définition d'un cadre de bureau réactif	58
15.3	Participants à l'exécution	58
15.4	Modèle d'hébergement : région de vue dynamique	59
15.5	Modèle de contexte : curseur d'entrée et objet courant	59
15.6	Modèle de récepteur d'événements applicatifs	60
15.7	Modèle de routeur des notifications vers les enfants	60
15.8	Modèle de remontée des événements enfant-vers-parent	61
15.9	Modèle de réaction aux DataSet	62
15.10	Modèle d'actualisation et de recalcul	62
15.11	Modèle de surface d'aperçu	63
15.12	Modèle d'enveloppe de workflow	64
15.13	Taxonomie des événements des cadres réactifs	64
15.14	Nommage recommandé des événements	64
15.15	Assemblage déclaratif	65
15.16	Correspondance avec le web réactif	65
15.17	État local et état du modèle	66
15.18	Éviter les boucles d'événements	66
15.19	Gestion des erreurs et diagnostics	67
15.20	Règles de conception	67
15.21	Pourquoi ce modèle est important	68
IV	Clients riches natifs et web	69
16	Hôtes natifs de clients riches	70
16.1	Un contrat stable	70
16.2	Cache et asynchronisme	70
16.3	Frontière entre plateformes et bibliothèques d'interface	71
16.4	Quand préférer un client MVP personnalisé	71
17	Créer des composants fonctionnels web réactifs	72
17.1	De la fonction UXfc au composant web	72
17.2	Contrat d'un composant réactif	72
17.3	Props réactives et surfaces de données de la vue	73
17.4	L'état local n'est pas l'état du modèle	73
17.5	Émettre des événements visuels locaux	74
17.6	Quand un événement franchit la frontière	74
17.7	Modèle d'abonnement et d'actualisation	74
17.8	Anti-rebond, regroupement et idempotence	75
17.9	Composants fonctionnels réactifs et actions	75
17.10	Enregistrement des composants en mode web	75
17.11	Assemblage déclaratif des composants web	76
17.12	Assemblage View XML équivalent	76
17.13	Esquisse d'implémentation réactive minimale	77

17.14 États d'erreur, de chargement et hors ligne	77
17.15 Règles de conception	77
18 Syntaxe React moderne pour les composants logiCells	78
18.1 Composants fonctionnels	78
18.2 Déstructuration des props	78
18.3 Déstructuration des attributs de métadonnées	78
18.4 État local et hooks	79
18.5 Effets et abonnements	79
18.6 Gestionnaires d'événements	79
18.7 Outillage de référence	79
18.8 Frontière avec le guide logiCells	79
19 Formulaire, listes et tableaux de bord dynamiques	81
19.1 Formulaire dynamique	81
19.2 Composants de sélection	81
19.3 Listes, vues tabulaires et grilles	81
19.4 Conteneurs : cadres fractionnés, à onglets et réactifs	82
19.5 Tableaux de bord comme assemblages	82
19.6 La navigation comme flux d'événements	82
19.7 Enregistrement des composants fonctionnels	82
19.8 Composants abstraits et concrets	83
19.9 Règles de conception	83
20 Composants fonctionnels, implémentations concrètes et visualisations	84
20.1 Trois couches	84
20.2 Déclaration du composant fonctionnel	84
20.3 Enregistrement	84
20.4 Implémentations équivalentes	85
20.5 Familles fonctionnelles disponibles	85
20.6 Composants abstraits	86
20.7 Événements et actions	86
20.8 La visualisation comme fonction	86
20.9 Règles de conception	86
21 Publication web à partir de vues conceptuelles	87
21.1 Les vues comme projections publiées	87
21.2 Composants fonctionnels	87
21.3 Actions dans les vues publiées	87
21.4 Exemple	87
21.5 Règles de conception	87
21.6 Synthèse	88
V Synchronisation et frontières externes	89
22 Applications conceptuelles « local-first »	90
22.1 Ce que signifie ici « local-first »	90
22.2 Cache local, fragments de graphe et instantanés	90
22.3 Événements locaux et remontée vers le distant	91
22.4 Ce que suit réellement le synchroniseur d'exécution	91
22.5 Fournisseurs distants comme frontières de synchronisation	91
22.6 Travail hors ligne et sémantique des conflits	92

22.7	Traitement courant selon le type d'opération	92
22.8	Règles de conception	92
VI	Ingénierie et diagnostics	93
23	Frontières techniques des clients riches	94
23.1	Diagnostiquer d'abord la frontière cliente	94
23.2	La durée de vie du client fait partie de la correction	94
23.3	Frontières entre plateformes et bibliothèques d'interface	95
23.4	Se reporter au Guide du développeur	95
23.5	Évolution de l'exécution et invalidation des vues	95
VII	Modèles et références	96
24	Glossaire du client riche	97
25	Modèles de vues conceptuelles	98
25.1	Périmètre	98
25.2	La View comme projection	98
25.3	Vues de classe et d'objet	98
25.4	Composants fonctionnels	98
25.5	Modèle d'assemblage UXfc	98
25.6	Familles de composants fonctionnels	99

À propos de ce guide

Ce guide présente l'architecture MVC/View facultative de logiCells pour les clients riches. Elle s'adresse aux applications natives ou web qui tirent parti d'un état client local, de caches fondés sur les DataSet/DataCursor, de projections déclaratives, du routage d'événements et d'interactions asynchrones avec l'environnement d'exécution conceptuel.

L'architecture View n'est pas indispensable pour utiliser logiCells. Un même modèle conceptuel exécutable peut être exploité par des applications en ligne de commande (CLI), des API et services, des clients de type MVP ou d'autres technologies de présentation. Ces fondements relèvent du guide principal, *Programmation conceptuelle déclarative avec la plateforme logiCells – Guide du développeur*.

Ce guide part donc d'une autre question : *lorsqu'une application nécessite un client interactif riche, comment celui-ci peut-il projeter, mettre en cache et coordonner l'état conceptuel sans redéfinir le modèle métier ?*

Plateformes d'exécution

L'environnement d'exécution logiCells sous-jacent est disponible sous Windows, Linux et macOS et peut être déployé dans des conteneurs. L'architecture View décrite ici est une architecture cliente reposant sur cet environnement. L'emballage propre aux plateformes sort du cadre de ce guide et relève du déploiement.

Périmètre

Le modèle métier demeure la référence pour les concepts, les relations, les contraintes, les actions et les autorisations. Les métadonnées View décrivent les projections clientes et les interactions. Les DataSet et les curseurs sont des mécanismes de projection et de cache côté client ; ils ne remplacent ni les champs du modèle ni les relations conceptuelles.

Première partie

Repères et orientation

Chapitre 1

Quand utiliser l'architecture View de logiCells

Utilisez l'architecture View lorsque le client doit faire davantage que simplement invoquer des actions du modèle. Les besoins courants comprennent une interaction native ou web riche, un cache local et un état de navigation, des opérations asynchrones, la coordination maître-détail, des packages View déclaratifs réutilisables, le routage d'événements visuels et une synchronisation maîtrisée entre les projections locales et l'environnement d'exécution.

1.1 Quand un client plus simple suffit

N'introduisez pas la pile View si une CLI, un consommateur d'API, un client de type MVP ou une autre technologie de présentation répond déjà au besoin. L'architecture View est une possibilité, non un préalable à la programmation conceptuelle.

1.2 Ce qui reste commun

Quelle que soit l'architecture cliente choisie, la sémantique métier doit rester au même endroit : les concepts et les relations décrivent le domaine, les contraintes et H-Logic expriment les règles de validité, les actions exposent les intentions et les frontières de publication rendent accessibles certaines capacités.

1.3 Ce qu'apporte cette architecture

La pile View ajoute autour de ces capacités une couche d'exécution orientée client :

```
1 conceptual model and actions
2 -> view projection
3 -> local DataSet / DataCursor state
4 -> component binding
5 -> local visual events
6 -> asynchronous action or synchronization boundary
7 -> refresh / merge / user-visible state
```

Exemple 1.1 – Chaîne des responsabilités d'un client riche

La suite du guide détaille chaque étape sans la confondre avec le modèle conceptuel lui-même.

Chapitre 2

Prérequis et articulation avec le Guide du développeur

Ce guide suppose que le modèle conceptuel exécutable existe déjà. Les concepts, relations, actions, autorisations, transactions, publications de services et diagnostics généraux de l'environnement d'exécution sont définis dans le *Guide du développeur* complémentaire ; ils ne sont pas redéfinis ici.

Utilisez le présent guide uniquement lorsque le client a besoin d'au moins l'un des éléments suivants : un cache DataSet/DataCursor local, des interactions asynchrones, le routage d'événements côté client, des packages View réutilisables ou une projection native/web de client riche gérée par l'architecture View de logiCells.

2.1 Quand consulter le Guide du développeur

Pour le modèle conceptuel, consultez *Le modèle mental de logiCells*. Pour les garanties transactionnelles et d'identité, voyez *Transactions, cohérence et identité*. Pour la publication d'API et l'intégration externe, consultez les chapitres consacrés aux services et à la publication. Pour la journalisation, le débogage et la résolution générale des incidents, référez-vous aux chapitres d'ingénierie de cet ouvrage.

Cette séparation est volontaire : le présent livre documente la couche de projection cliente. Le Guide du développeur reste la référence pour les contrats métier et d'exécution indépendants d'une View.

Chapitre 3

Contrat de référence et état des capacités

Ce guide ne couvre que l'architecture cliente facultative. Il ne redéfinit ni la grammaire MODEL déclarative, ni H-Logic, ni les transactions, ni l'ABI, ni la sémantique générale de l'environnement d'exécution.

3.1 Sources faisant autorité

Lorsqu'un exemple client entre en conflit avec un contrat fondamental, respectez l'ordre de priorité suivant :

1. la grammaire actuelle du framework, ses interfaces publiques et ses tests exécutables ;
2. les contrats d'exécution et de liaison générés pour la distribution choisie ;
3. les exemples vérifiés de programmation de l'environnement d'exécution pour le sous-système visé ;
4. le Guide du développeur principal pour la sémantique conceptuelle et celle de l'exécution ;
5. le présent guide du client riche pour les schémas View, DataSet/DataCursor, de routage d'événements et de cache client ;
6. les exemples et modèles d'intégration propres à cet ouvrage.

Un exemple View ne doit pas introduire discrètement un second dialecte MODEL. Le XML de ce guide concerne les artefacts View ; les déclarations MODEL restent conformes à la grammaire déclarative canonique décrite dans le Guide du développeur.

3.2 Vocabulaire des statuts

Le même vocabulaire que celui du Guide du développeur s'applique : **Vérifié**, **Disponible**, **Modèle d'intégration**, **Conception / en cours** et **Prévu**. Un mécanisme client ne doit pas être présenté comme plus mature que la surface d'exécution dont il dépend.

3.3 Frontière entre plateformes

L'environnement d'exécution sous-jacent est disponible sous Windows, Linux et macOS et peut être déployé dans des conteneurs. Une implémentation View, une bibliothèque d'interface, un composant natif ou un transport de synchronisation particulier peut néanmoins prendre en charge un périmètre de plateformes plus restreint. Ce guide distingue donc la disponibilité de l'environnement d'exécution de la certification des bibliothèques d'interface clientes.

3.4 Frontière de synchronisation

La documentation d'ingénierie de l'environnement d'exécution fournie confirme l'existence d'une couche de synchronisation avec enregistrement explicite, échanges de lecture/écriture, accusés de réception, positions dans le journal, horloges causales et routage fondé sur des règles. Elle ne permet *pas* d'affirmer de manière générale que toutes les modifications concurrentes sont fusionnées automatiquement ou sans conflit.

Quand une application nécessite des écritures hors ligne, des mutations concurrentes ou une résolution automatique des conflits, documentez et testez le contrat précis du synchroniseur ou du fournisseur utilisé.

Chapitre 4

Les vues comme contrats d'interaction et projections de concepts

Une vue logiCells n'est pas principalement une définition d'écran. Dans l'architecture moderne, l'artefact stable est un *contrat d'interaction* : une description indépendante du canal de la manière dont un utilisateur peut obtenir des informations, fournir les informations manquantes, invoquer des capacités et réagir à l'état de l'application. Cet utilisateur peut être une personne ou un agent autonome.

Le terme historique *view* reste utile pour la compatibilité et les projections affichées, mais les fichiers sous `meta/views/contracts` doivent être interprétés comme des contrats d'interaction, et non comme des formulaires liés à une bibliothèque d'interface. Une page web, un formulaire natif et un dialogue agentique peuvent exploiter différemment le même contrat.

Ce chapitre explique pourquoi les vues doivent être comprises comme des projections conceptuelles d'interaction plutôt que comme des écrans isolés. Il introduit également la lecture UXfc des contrats : un contrat d'interaction est un assemblage d'intentions, d'exigences de données, d'actions, d'événements et de composants fonctionnels, et non un simple agencement de widgets concrets.

4.1 Contrat d'interaction et vue affichée

La terminologie est volontairement distincte :

```
1 Interaction contract
2   channel-independent interaction semantics
3   data made available to the interaction
4   information that may need to be collected
5   actions, events and navigation capabilities
6   functional component intentions
7
8 Rendered or enacted projection
9   Web view      -> HTML/component rendering
10  Native view   -> VCL/FMX or another native toolkit
11  Agentic view  -> dialogue, tool invocation and selective prompting
```

Exemple 4.1 – Terminologie des contrats et projections

Un contrat peut donc être comparé à un diagramme de séquence évolué : il décrit les échanges possibles entre un participant et les services applicatifs tout en portant les métadonnées de données et de comportement nécessaires à leur pilotage. Tous les échanges n'ont pas besoin d'être présentés visuellement.

4.2 Projection agentique sans formulaire

Un agent peut exploiter un contrat d'interaction sans matérialiser de formulaire. Il peut inspecter les données disponibles dans l'interaction courante, déterminer quelles valeurs obligatoires manquent encore, ne demander que celles-ci, invoquer une action déclarée et présenter le résultat sous une forme conversationnelle ou orientée outils.

Par exemple, si une opération exige un client, un montant et un motif, l'agent peut réutiliser les valeurs déjà connues du contexte et ne demander que le montant manquant. Le contrat demeure la source commune de sémantique de l'interaction, même lorsqu'aucun formulaire visuel n'est affiché.

Il s'agit d'une frontière d'architecture essentielle : le contrat décrit *quelle interaction est possible et nécessaire* ; un moteur de rendu ou un environnement d'exécution d'agent décide *comment elle est réalisée*.

4.3 Projection, intention et fonction

Une vue logiCells doit être envisagée au travers de trois questions :

```
1 What concept is being projected?
2   -> object, class, relation, process or service
3
4 What participant intention is being supported?
5   -> exploration, selection, consultation, edition, validation, navigation, completion
6
7 What interaction functions are exposed?
8   -> fields, lists, choices, status, actions, prompts, navigation and component intents
```

Exemple 4.2 – Trois questions pour une vue

L'apparence exacte, ou l'absence d'apparence visuelle, intervient après ces questions. Un contrat d'interaction peut conserver la même projection conceptuelle et la même intention UXfc tout en changeant de rendu concret ou en étant consommé par un agent.

4.4 Déclarations de vues

Une déclaration de vue associe le nom d'un contrat d'interaction à une cible et à une ou plusieurs ressources contractuelles. Un moteur de rendu peut ensuite les interpréter visuellement, tandis qu'un agent peut les lire comme un plan d'interaction.

```
1 views:
2   - WorkItemList:
3     caption: '#WorkItemList'
4     enabled: true
5     viewTarget: ClassItem
6     items:
7       - Fields:
8         path: /WorkItem/List.view.xml
9         enabled: true
10        caption: '#Fields'
```

Exemple 4.3 – Déclaration de vue conceptuelle

Le contrat indique comment un concept peut participer à une interaction. Il ne prétend pas définir sa seule représentation ou son seul canal possible.

4.5 Vues ClassItem et ObjectItem

Une vue de niveau classe affiche généralement une collection ou une liste d'instances. Elle sert à parcourir, rechercher, sélectionner ou créer des objets.

Une vue de niveau objet affiche ou modifie un objet donné. Elle convient aux formulaires de détail, aux étapes de processus, aux écrans de validation et aux tableaux de bord propres à un objet.

La distinction est importante, car elle contrôle le contexte injecté dans la vue. Une vue `ClassItem` reçoit un contexte au niveau du concept ; une vue `ObjectItem` reçoit un contexte au niveau d'une instance.

4.6 Vues et contexte de données

Les vues se relient aux données par des curseurs, `DataSet`, sources de données et chemins de collections. Une vue n'a pas besoin de connaître la structure de la base de données. Elle se lie aux collections conceptuelles déclarées dans le modèle.

La collection est conceptuelle. La vue ne fait que consommer sa projection.

4.7 Composants fonctionnels

Les déclarations visuelles utilisent des composants fonctionnels. Un composant fonctionnel est la description déclarative d'une fonction d'interaction : bouton, liste, grille, champ, cadre, disposition adaptative, zone de signature, graphique, lecteur de documents ou autre rôle interactif.

La déclaration n'est pas nécessairement l'implémentation concrète. Un composant tel que `lgc#button`, `lgc#gridPanel` ou `lgc#reactiveDesktopFrame` désigne une fonction dans le modèle View. Le composant effectivement utilisé à l'exécution est choisi par l'enregistrement d'une implémentation concrète.

Trois préoccupations sont ainsi séparées :

```
1 Declarative function:
2   "this part of the view is a grid"
3
4 Registered implementation:
5   "on this platform, render the grid with this concrete component"
6
7 Appearance and style:
8   "use this layout, skin, size, caption or style lookup"
```

Exemple 4.4 – Séparation des composants fonctionnels

Le modèle réutilisable n'est pas l'apparence d'un écran particulier, mais le modèle déclaratif de composants fonctionnels et le mécanisme d'enregistrement qui les relie à des composants concrets.

4.8 Composants abstraits

Certains composants sont abstraits. Un composant abstrait définit une fonction sans être directement instanciable. Il sert à préciser un contrat que les composants concrets doivent réaliser.

Une fonction générique de visualisation de données peut, par exemple, indiquer qu'un composant consomme une source de données et affiche des données structurées. Une implémentation concrète peut le rendre comme une grille, un graphique, une frise chronologique ou un contrôle visuel spécialisé.

Les composants abstraits permettent au modèle d'exprimer une sémantique d'interaction stable tout en laissant le rendu concret ouvert.

4.9 Vues et événements

Les vues peuvent réagir aux événements et en émettre. Un cadre peut réagir à `OnShow`. Un bouton peut émettre un événement d'action visuelle. Un modèle de page peut afficher ou masquer des composants de navigation. Un contrôleur peut écouter un événement de processus et afficher une vue.

La couche contractuelle est donc aussi déclarative et réactive. Ce n'est pas une simple disposition statique, mais une projection d'interaction dotée d'un comportement événementiel déclaré, pouvant être réalisée par des clients natifs, web ou agentiques.

Deuxième partie

Packages View et structure du client

Chapitre 5

Contrats d'interaction, packages VIEW, MODEL-behind et artefacts compilés

Les packages VIEW modernes de logiCells séparent strictement les sources des contrats d'interaction des métadonnées MODEL exécutables. Le terme historique *view* reste présent dans les types de fichiers et les API, mais `meta/views/contracts` contient des contrats d'interaction indépendants du canal, et non des définitions d'écrans propres à une bibliothèque d'interface. Ce chapitre fixe la convention applicable aux nouveaux modules et aux migrations d'anciens formulaires et cadres.

5.1 Organisation canonique des sources

Un package propriétaire de vues doit présenter une arborescence comparable à celle-ci :

```
1 Modules/erp-orders/  
2   meta/  
3     models/  
4       Order.model.yaml  
5       _manifest.yaml  
6     views/  
7       contracts/  
8         Order/  
9           OrderList.view.xml  
10          OrderList.models.yaml  
11          OrderEdit.view.xml  
12          _manifest.yaml  
13       models/  
14         Order/  
15           Order.model.yaml  
16           _manifest.yaml  
17 packages/  
18   erp-orders.package.yaml  
19   erp-orders.view.package.yaml
```

Exemple 5.1 – Arborescence recommandée des contrats VIEW

La source du contrat est `*.view.xml`. Elle peut contenir des informations de disposition visuelle, mais son rôle architectural est plus large : elle déclare une surface d'interaction projetable vers des canaux natifs, web ou agentiques. Les classes MODEL-behind se trouvent dans des fichiers `*.models.yaml` voisins, sous une racine YAML `classes:`. Un fichier `*.models.yaml` n'est généré que si le contrat correspondant contient des métadonnées MODEL à externaliser.

L'arborescence `meta/views/models` répond à un autre besoin. Elle contient des redéfinitions MODEL minimales ou des surcharges de classes métier existantes lorsque les métadonnées liées

aux vues doivent être séparées du package MODEL métier. Ce n'est pas une seconde copie de l'arborescence des contrats.

5.2 Pourquoi le dossier s'appelle contracts

Le nom `contracts` est volontaire : ces artefacts définissent le contrat d'interaction entre un participant et les capacités de l'application. Le participant peut être une personne ou un agent autonome. Un moteur natif ou web peut matérialiser champs, listes et actions en contrôles ; un agent peut plutôt inspecter le même contrat, présenter l'information disponible, ne demander que les valeurs manquantes et invoquer les capacités déclarées sans afficher de formulaire.

Le bon modèle mental est un diagramme de séquence évolué : le contrat décrit les échanges possibles, les exigences de données, les événements et les actions entre le participant et les services applicatifs. Le rendu concret est une projection du contrat, non sa définition.

5.3 Liaison explicite de XML vers MODEL

Une vue déclare explicitement son contrôleur de page et les modèles de ses composants.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <lgc:frames xmlns:lgc="www.logicells.net">
3   <lgc:dataModel>
4     <dataCursor name="MainCursor">
5       <dataSet name="Orders" collectionName="OpenOrders">
6         <fields>
7           <field name="Id" fieldType="String" fieldSize="32"/>
8           <field name="Number" fieldType="String" fieldSize="64"/>
9           <field name="Status" fieldType="String" fieldSize="32"/>
10        </fields>
11        <dataSource name="OrdersSource"/>
12        <dataCursor name="OrdersCursor"/>
13      </dataSet>
14    </dataCursor>
15  </lgc:dataModel>
16
17  <lgc:pageModel
18    source="OrderList.models.yaml"
19    class="Page"
20    classId="view#erp.orders.OrderList.Page"/>
21
22  <lgc:masterView name="Master" align="client" responsive="true">
23    <model
24      source="OrderList.models.yaml"
25      class="Master"
26      classId="view#erp.orders.OrderList.Master"/>
27  </lgc:masterView>
28 </lgc:frames>
```

Exemple 5.2 – Liaison XML VIEW vers les classes MODEL-behind

Ne déduisez pas l'identité du contrôleur à partir des seuls noms de fichiers. La liaison est explicite et doit être validée. Un contrat dépourvu de liaison MODEL n'exige pas la création artificielle d'un compagnon `*.models.yaml`.

5.4 MODEL-behind suit le pipeline MODEL ordinaire

```
1 classes:
2   - class:
3     name: Page
4     classId: view#erp.orders.OrderList.Page
5     currentGeneratorName: Uri
6     type: Entity
```

```

7   concepts:
8     - Entity:
9       constraints:
10        - implies: lgc#desktop_object_behind
11   events:
12     - idv.Visual.View.Refresh:
13       actions:
14         - action:
15           path: './Action[@name='OnRefreshView']'
16   actions:
17     - action:
18       name: OnRefreshView
19       class: ObjectItemScriptableObjectElementAction
20       type: ObjectItem
21       published: true
22       scripts:
23         - script:
24           clientScript: true
25           serverScript: false
26           language: functionalP
27           classes: Desktop
28           cdata: |
29             OrdersCursor := Self.PageController.DataSetsManager.CursorByName('OrdersCursor');
30             OrdersCursor.DataSet.Refresh();

```

Exemple 5.3 – Squelette du contrôleur MODEL-behind

Ces classes sont compilées par les mécanismes ordinaires de package/MODEL. La découverte des classes est déclarée dans le manifeste canonique `_.manifest.yaml`. Pour une classe stockée dans un conteneur multi-classes, l'entrée du manifeste nomme ce conteneur physique à l'aide de `fileName` :

```

1 package:
2   uri: view#meta.views.Order
3   classes:
4     - Page:
5       classId: view#meta.views.Order.OrderList.Page
6       fileName: OrderList.models.yaml
7       type: Entity
8     - Master:
9       classId: view#meta.views.Order.OrderList.Master
10      fileName: OrderList.models.yaml
11      type: Entity

```

Exemple 5.4 – Entrées du manifeste pour les classes d'un fichier .models.yaml

Le chargeur résout `fileName`, ouvre le conteneur `classes`, sélectionne la classe demandée par nom et la transmet au chargeur MODEL ordinaire. L'ancien rôle d'un compilateur VIEW compilant également les métadonnées MODEL intégrées est obsolète. La compilation propre à VIEW doit assembler et valider les artefacts d'interaction modifiés, puis lier des identités déjà compilées par MODEL.

5.5 Responsabilité du package VIEW

Le package VIEW compagnon est propriétaire de l'arborescence des vues et participe au plan de compilation de l'application.

```

1 package:
2   dependencies:
3     - erp-orders
4   files:
5     schemaDirectories:
6       schemaPath:
7         value: '[Modules]\\erp-orders\\meta\\views\\models\\'
8       schemaPath:
9         value: '[Modules]\\erp-orders\\meta\\views\\contracts\\'
10  items:

```

```

11 - Order:
12   caption: Order
13   enabled: true

```

Exemple 5.5 – Package VIEW compagnon avec deux racines de schéma

Les éléments sont relatifs aux racines de schéma déclarées. Ne préfixez pas un élément par `contracts.` ou `models.` simplement parce que son fichier physique se trouve sous l'un de ces dossiers. Gardez la responsabilité au niveau local du package : un module ne doit posséder que ses propres contrats et surcharges de modèles de vues. L'absence d'une déclaration `viewPaths` ne doit pas conférer la propriété de toute l'arborescence globale `meta/views`.

5.6 Compilation incrémentale

Les deltas MODEL et VIEW sont planifiés séparément :

```

1 VIEW XML changed, MODEL unchanged
2 -> dirty VIEW artifact only
3
4 VIEW MODEL changed, consumed binding unchanged
5 -> MODEL metadata delta only
6
7 VIEW MODEL binding changed
8 -> MODEL delta + dependent VIEW artifact
9
10 shared VIEW fragment changed
11 -> only consuming VIEW artifacts
12
13 immediate rerun
14 -> zero VIEW artifact compilation work

```

Exemple 5.6 – Conséquences incrémentales

Une vue modifiée est recomposée de façon déterministe à partir de ses entrées courantes. Réutiliser des fragments analysés et inchangés est une optimisation, pas une source de vérité alternative.

5.7 Les artefacts VIEW compilés sont des produits d'exécution durables

Les fichiers fusionnés `*.view.xml` sont des produits compilés. Ils se trouvent sous `compiledPackagesBasePath`, dans la même génération acceptée que les artefacts de package `.lcmp` et `.ast`. Le dossier `cachePath` ne contient que des caches restructurables et ne doit pas être nécessaire à l'environnement d'exécution pour ouvrir une vue compilée.

Une génération déployée uniquement sous forme compilée doit pouvoir démarrer à partir de :

```

1 application bootstrap/configuration
2 *.lcmp
3 *.ast when required
4 merged views/*.view.xml
5 compiled generation + artifact manifest
6 required semantic runtime artifacts

```

Exemple 5.7 – Entrées d'exécution d'une VIEW compilée uniquement

Les sources MODEL/VIEW peuvent être omises d'une distribution compilée uniquement lorsque son manifeste de génération fait autorité et est complet. La validation de développement tenant compte des sources et la validation de déploiement compilé uniquement constituent donc deux modes distincts.

5.8 Contrôles de non-régression

Pour chaque module comportant des vues, automatisez les vérifications suivantes :

- un plan de compilation inchangé ne signale aucun artefact VIEW à recomposer ;
- une modification XML produit une fermeture VIEW limitée ;
- une modification de liaison MODEL-behind recompile le MODEL concerné et la VIEW dépendante ;
- une relance immédiate n'entraîne aucune opération ;
- les compilations intégrales et incrémentales produisent des empreintes identiques pour les vues fusionnées ;
- les fichiers `.lcmp`, `.ast` et le XML de vue fusionné proviennent de la même génération ;
- un déploiement compilé uniquement ouvre ses vues sans arborescence de sources.

Chapitre 6

Créer des vues de tâches avec DataSet et modèles de page

Une application utile n'est pas seulement un modèle. Elle a besoin d'une page qui expose un cache de données cohérent, un comportement de page et des composants visibles. Ce tutoriel construit une vue `Task` plus riche selon la séparation moderne : hiérarchie visuelle dans `*.view.xml` et MODEL-behind de page/contrôleur dans `*.models.yaml`.

Lorsqu'un extrait contient du `functionalP`, consultez d'abord chapitre 10 «functionalP pour débutants sur logiCells». Le script n'est pas la syntaxe du modèle : c'est un fragment exécutable dans une action ou un gestionnaire d'événements déclaré.

6.1 Les trois parties d'une vue

Une vue logiCells s'organise généralement autour de trois parties complémentaires.

```
1 TaskEdit.view.xml      # contract structure, data context and component bindings
2 TaskEdit.models.yaml   # MODEL-behind behavior, events and actions
3 _.manifest.yaml        # discovery of classes stored in TaskEdit.models.yaml
```

Exemple 6.1 – Répartition des sources d'un contrat d'interaction

La section `dataSets` est le cache d'interaction local. La section `pageModel` relie le comportement au contrat. La section `masterView` décrit une projection visible, sans pour autant rendre le contrat exclusivement natif ou web.

Lorsqu'un contrat ne contient aucune déclaration MODEL, de modèle de page ou de composant, il ne faut pas générer de fichier `*.models.yaml` vide. Inversement, plusieurs classes MODEL appartenant au même contrat peuvent partager un conteneur `*.models.yaml`, découvert via `_.manifest.yaml` grâce à `fileName`.

6.2 Déclarer le cache de données de page

La page des tâches utilise un DataSet principal de tâche, un DataSet de liste, deux DataSet de référence et un petit DataSet de mémoire locale pour les filtres.

```
1 dataSets:
2   dataCursor:
3     name: TaskPageCursor
4     dataSets:
5       - name: TaskEntityDs
6         collectionPath: /ClassItem[@name='TaskManagement.Task']/Collection[@name='Entity']
7         fields:
8           - Id: String
9           - Title:
10             fieldType: String
```

```

11     fieldSize: 200
12   - Description:
13     fieldType: String
14     fieldSize: 2000
15   - StatusId: String
16   - AssigneeId: String
17   - DueDate: DateTime
18   - Priority: Integer
19   dataSource: TaskEntityDS
20   dataCursor: TaskEntityCursor
21 - name: TaskListDs
22   collectionPath: /ClassItem[@name='TaskManagement.Task']/Collection[@name='Entity']
23   fields:
24     - Id: String
25     - Title: String
26     - StatusName: String
27     - AssigneeName: String
28     - DueDate: DateTime
29     - Priority: Integer
30   dataSource: TaskListDS
31   dataCursor: TaskListCursor
32 - name: StatusesDs
33   collectionPath: /ClassItem[@name='TaskManagement.Status']/Collection[@name='Entity']
34   fields:
35     - Id: String
36     - Name: String
37   dataSource: StatusesDS
38 - name: UsersDs
39   collectionPath: /ClassItem[@name='Accounts.User']/Collection[@name='Entity']
40   fields:
41     - Id: String
42     - DisplayName: String
43   dataSource: UsersDS
44 - name: TaskMemoryDs
45   collectionName: Entity
46   fields:
47     - SearchText:
48       fieldType: String
49       fieldSize: 100
50     - OnlyOpen: Boolean
51   dataSource: TaskMemoryDS

```

Exemple 6.2 – Cache de données d'une vue de tâches en YAML

Ne placez pas `kind="slot"` ni `kind="attribute"` sur les champs DataSet. Cet attribut appartient aux champs de modèle dans les facettes conceptuelles, et non aux caches de vues.

6.3 Initialiser l'état de la page

À son ouverture, la page initialise le DataSet de mémoire locale utilisé par les filtres.

```

1 pageModel:
2   class:
3     data:
4       concepts:
5         - TaskPage:
6           constraints:
7             - implies:
8               relatedTo: lgc#desktop_object_behind
9       events:
10      - event:
11        class: logiCells.Frame.OnShow
12        action: OnShowPage
13      actions:
14      - action:
15        name: OnShowPage
16        class: ObjectItemScriptableObjectElementAction
17        type: ObjectItem
18        scripts:
19      - clientScript: true

```

```

20     language: functionalP
21     code: |
22         lMem := Self.PageController.DataSetsManager.DataSetByName('TaskMemoryDs');
23         if lMem.IsEmpty then begin
24             lMem.Append();
25             lMem.OnlyOpen := True;
26             lMem.PostData();
27         end;

```

Exemple 6.3 – Événement OnShow et action de page en YAML

6.4 Filtrer un DataSet avant son ouverture

Le DataSet de liste peut être trié et filtré avant l'ouverture.

```

1  events:
2  - event:
3      class: logiCells.Visual.Data.BeforeDataSetOpened
4      action: OnBeforeOpenDataset
5  actions:
6  - action:
7      name: OnBeforeOpenDataset
8      class: ObjectItemScriptableObjectElementAction
9      type: ObjectItem
10     scripts:
11     - clientScript: true
12       language: functionalP
13       code: |
14           if Sender.Name='TaskListDs' then begin
15               Sender.SetOrdering('Priority desc, DueDate asc');
16               lMem := Self.PageController.DataSetsManager.DataSetByName('TaskMemoryDs');
17               if lMem.OnlyOpen then
18                   Sender.AddStringFilterItem('StatusName', '<>', 'Done');
19               if lMem.SearchText<>' ' then
20                   Sender.AddStringFilterItem('Title', 'contains', lMem.SearchText);
21           end;

```

Exemple 6.4 – Filtre BeforeDataSetOpened en YAML

6.5 Actions Enregistrer et Annuler

L'action Enregistrer publie l'objet courant et demande à la vue parente de s'actualiser. L'action Annuler ferme le formulaire sans publication.

```

1  actions:
2  - action:
3      name: SaveTask
4      class: ObjectItemScriptableObjectElementAction
5      type: ObjectItem
6      scripts:
7      - clientScript: true
8        language: functionalP
9        code: |
10            lDs := Self.PageController.DataSetsManager.DataSetByName('TaskEntityDs');
11            lTask := lDs.ObjectItem;
12            lTask.PostData();
13            Ev := Self.PageController.ParentPageController.CreateVisualEvent('Lgc.Visual.View.Refresh', nil,
14                nil, Self.ObjectsContainer.MetadataContainer, false);
15            Self.PageController.ParentPageController.DispatchVisualEvent(Ev, Stop);
16            Self.PageController.GetVisualComponentObjectBehindByName('Master').CloseForm();
17  - action:
18      name: CancelEdit
19      class: ObjectItemScriptableObjectElementAction
20      type: ObjectItem
21      scripts:
22      - clientScript: true
23        language: functionalP

```

```

23     code: |
24         Self.PageController.GetVisualComponentObjectBehindByName('Master').CloseForm();

```

Exemple 6.5 – Actions enregistrer et annuler en YAML

Utilisez des noms d'événements neutres dans la documentation publique. Les noms propres à un projet ne doivent pas se retrouver dans des exemples réutilisables.

6.6 Lier les contrôles de formulaire aux DataSource

La hiérarchie visuelle appartient à *.view.xml. Les contrôles se lient aux noms de DataSource, pas directement aux tables SQL ni aux fournisseurs distants.

```

1 <ResponsiveLayoutFrame name="TaskForm" columns="12">
2   <Row>
3     <Col span="7">
4       <Edit name="TitleEdit"
5         nameField="Title"
6         dataSource="TaskEntityDS"
7         caption="#Title" />
8     </Col>
9     <Col span="4">
10      <AutoCompleteSelect name="StatusSelect"
11        nameField="StatusId"
12        dataSource="TaskEntityDS"
13        selectIdField="Id"
14        selectNameField="Name"
15        selectDataSource="StatusesDS"
16        caption="#Status" />
17    </Col>
18  </Row>
19 </ResponsiveLayoutFrame>

```

Exemple 6.6 – Contrôles d'édition des tâches dans TaskEdit.view.xml

6.7 Actions des boutons

Les liaisons des boutons sont aussi des métadonnées visuelles et restent donc dans *.view.xml. Le comportement métier demeure dans l'action YAML MODEL-behind.

```

1 <Button name="SaveBtn" caption="#Save">
2   <Model>
3     <Class>
4       <Events>
5         <Event class="logiCells.Visual.Button.Action"
6           actionPath="../../../../Action[@name='SaveTask']" />
7       </Events>
8     </Class>
9   </Model>
10 </Button>

```

Exemple 6.7 – Liaison de l'action d'un bouton dans TaskEdit.view.xml

6.8 Enregistrer les classes MODEL-behind dans le manifeste

Le manifeste local enregistre chaque classe MODEL-behind et référence le conteneur physique multi-classes.

```

1 package:
2   uri: view#meta.views.TaskManagement.Task
3   classes:
4     - Page:
5       classId: view#meta.views.TaskManagement.Task.TaskEdit.Page

```

```
6     fileName: TaskEdit.models.yaml
7     type: Entity
```

Exemple 6.8 – Manifeste MODEL-behind de TaskEdit

Le nom de classe sélectionne une entrée dans `TaskEdit.models.yaml`; le nom de fichier identifie la source physique; le `classId` reste l'identité logique stable utilisée à l'exécution.

6.9 Exercice

Ajoutez un champ court `Tag` au modèle de tâche, à `TaskEntityDs` et au formulaire visible. Ajoutez ensuite `OnlyMine` à `TaskMemoryDs` et étendez `OnBeforeOpenDataset`. Traitez cet exercice comme une extension locale du tutoriel; les pratiques générales de test des modèles sont décrites dans le Guide du développeur complémentaire.

Chapitre 7

Vues, DataSet et curseurs d'objets

Les vues logiCells sont des projections d'objets conceptuels et non des descriptions d'écrans isolées. Une vue relie les composants visuels aux curseurs d'objets, DataSet, actions, collections et événements.

Ce chapitre introduit le modèle de navigation à l'exécution qui sous-tend les formulaires, cadres, listes, tableaux de bord et écrans de processus.

7.1 Vues comme projections pilotées par le modèle

Une déclaration de vue associe un nom à une ou plusieurs ressources visuelles.

```
1 views:
2   - BaseFrame:
3     caption: '#BaseFrame'
4     enabled: true
5     viewTarget: ObjectItem
6     items:
7       - Fields:
8         path: /WorkItem/Fields.view.xml
9         enabled: true
10        caption: '#Fields'
```

Exemple 7.1 – Déclaration de vue sur une classe conceptuelle

Le nom de la vue fait partie du contrat du modèle. Un processus, un contrôleur ou une action peut demander à l'environnement d'exécution d'afficher **BaseFrame** pour un objet donné. La vue est résolue à partir des métadonnées.

```
1 WorkItem.ShowObjectItemView(Self.PageControler, 'BaseFrame', '', '', false, true, true);
```

Exemple 7.2 – Affichage d'une vue pour un objet

7.2 Curseurs de données

Un curseur de données porte le contexte de l'objet courant pour une vue. Il peut pointer vers l'objet affiché, un objet lié ou un contexte enfant défini par une collection.

La vue peut ensuite lier des composants visuels à **EntityDataSource**. Le même modèle s'applique aux champs simples, tableaux, grilles, listes, paramètres d'action et curseurs imbriqués.

7.3 DataSetsManager comme contrat de navigation

À l'exécution, une vue peut obtenir curseurs et DataSet auprès de son gestionnaire de jeux de données.

```
1 MainSourceCursor := Self.PageController.DataSetsManager.CursorByName('MainSourceCursor');  
2 CurrentObject := MainSourceCursor.ObjectItem;
```

Exemple 7.3 – Obtention de l’objet courant depuis un curseur

Il ne s’agit pas seulement d’une commodité pour l’interface : c’est un contrat général de navigation permettant de :

- résoudre l’objet courant à partir du contexte de la vue ;
- récupérer les DataSet par nom ;
- propager le contexte par les curseurs maître-détail ;
- lier les paramètres d’action aux DataSet ;
- router les événements selon le contexte visuel et objet courant.

7.4 Chemins de collections et d’actions

Un DataSet peut s’appuyer sur une collection nommée ou sur un chemin menant à une collection ou à une action.

Le second modèle est important : une action peut exposer une structure de paramètres qu’une vue utilise comme modèle d’interaction temporaire. Il devient possible de lier des formulaires à des actions sans créer prématurément un objet métier permanent.

7.5 Comportement événementiel des vues

Une vue peut définir son propre modèle de page. Celui-ci possède des données, événements et actions comme les autres objets conceptuels. Il peut écouter des événements visuels et applicatifs, puis mettre à jour la visibilité, la navigation, les titres ou les cadres enfants.

Le gestionnaire d’événements peut ainsi récupérer l’objet courant, mettre à jour l’en-tête, afficher ou masquer les commandes de navigation ou ouvrir la vue conceptuelle suivante.

7.6 Règle de conception

La règle pratique est la suivante :

Une vue doit projeter un contexte conceptuel ; elle ne doit pas devenir propriétaire de la logique métier.

L’état métier appartient aux objets métier. La coordination entre objets appartient aux objets de processus. La navigation et l’adaptation de l’affichage appartiennent aux contrôleurs et aux modèles de page. Les DataSet et les curseurs constituent le pont entre ces couches.

Chapitre 8

Composants fonctionnels intentionnels UXfc

La couche visuelle de logiCells n'est pas une simple collection de widgets. Elle s'organise autour de composants fonctionnels et intentionnels. Un composant fonctionnel déclare ce qu'une zone de vue doit permettre. Un composant intentionnel UXfc va plus loin : il déclare pourquoi cette interaction existe et quelle activité de l'utilisateur elle doit soutenir.

Cette distinction compte pour les modèles publics. Une vue réutilisable ne doit pas être définie par l'apparence exacte d'un écran. Elle doit l'être par l'objet conceptuel projeté, l'activité utilisateur prise en charge, les composants fonctionnels requis et les implémentations concrètes enregistrées pour la plateforme cible.

8.1 De l'activité utilisateur à l'interface affichée

Un composant UXfc part de l'activité de l'utilisateur, non d'un contrôle graphique. Le chemin de conception est :

```
1 user activity
2   -> usage intention
3   -> functional constraints
4   -> intentional UXfc component
5   -> functional incarnation
6   -> registered concrete component
7   -> rendered interface
```

Exemple 8.1 – Chemin de conception UXfc

Ce chemin évite de confondre une intention métier ou d'interaction avec une bibliothèque d'affichage particulière. Il permet aussi de dériver plusieurs interfaces d'une même intention conceptuelle.

8.2 Composants intentionnels

Un composant intentionnel décrit un verbe d'usage : exploration, sélection, comparaison, consultation, édition, validation, navigation ou séquençement. Ces verbes ne sont pas de simples étiquettes. Ils déterminent les structures de données attendues, les relations entre éléments, les modes d'interaction, les contraintes d'assemblage et les critères de cohérence UX.

Par exemple, consulter ne revient pas à afficher un widget de navigateur. Consulter signifie que l'utilisateur lit ou parcourt une ressource adressable dans un contexte. L'implémentation peut être un cadre de navigateur, un visualiseur de documents, une vue web intégrée ou une vue d'application distante. L'intention reste identique.

8.3 Composants fonctionnels

Un composant fonctionnel décrit la fonction d'interaction demandée par la vue. Il peut s'agir d'un champ, d'un contrôle de sélection, d'une vue de liste, d'une vue tabulaire, d'une vue fractionnée, d'une vue à onglets, d'une vue d'état, d'un cadre de bureau réactif, d'un visualiseur de documents ou d'une autre fonction réutilisable.

Le composant fonctionnel reste déclaratif. Il peut recevoir des paramètres non visuels : noms de sources de données, champs de sélection, mode de navigation, politique d'historique, persistance du contexte, mode de validation ou contrats d'événements. Ces paramètres décrivent comment l'activité de l'utilisateur est prise en charge, et non la façon dont le composant est dessiné.

8.4 Implémentations concrètes

Le composant effectif à l'exécution est choisi par enregistrement. Celui-ci associe un nom de composant fonctionnel à une classe concrète, un nom de vue, un contrôleur et une plateforme cible.

```
1 logical function: listView
2 specific view:    TabularView
3 runtime class:    TabularView
4 controller:       Controllers.Desktop.ListViewControl
5 platform:         visual/desktop
```

Exemple 8.2 – Principe d'enregistrement d'un composant fonctionnel

Une même fonction logique peut disposer de plusieurs implémentations concrètes. Une vue à onglets peut se présenter sous forme d'onglets, de groupe empilé ou d'un autre cadre séquentiel. Une vue fractionnée peut utiliser un cadre à disposition absolue ou relative. Une vue de liste peut être rendue comme une liste standard, une grille tabulaire ou un composant de collection propre à une plateforme.

8.5 Composants abstraits

Certains composants UXfc sont abstraits. Ils définissent une fonction d'interaction sans pouvoir être instanciés directement. Ils spécifient un contrat : entrées attendues, événements pris en charge, contraintes de composition et incarnations possibles.

Un composant abstrait d'exploration peut, par exemple, stipuler que l'utilisateur explore un graphe, une hiérarchie ou un DataSet lié au moyen d'un curseur et de sources de données associées. Une implémentation concrète peut le présenter sous forme d'arbre, de navigateur de graphe, de navigateur maître-détail, de série de cartes ou d'espace de recherche et de filtrage.

8.6 Assemblages de composants UXfc

Une vue assemble généralement plusieurs composants fonctionnels. L'assemblage décrit comment les fonctions d'interaction collaborent.

```
1 view: WorkItemReview
2 context:
3   object: WorkItem
4   cursor: MainSourceCursor
5 components:
6   - consultation: show the current reference document
7   - selection: choose a related contact or phase
8   - tabularView: display related work items
9   - validation: confirm or reject the current state
10 events:
11   - SelectionChanged
```

```
12 - ResourceLoaded
13 - ValidationRequested
```

Exemple 8.3 – Assemblage de vue avec des fonctions UXfc

L'assemblage ne représente pas l'apparence visuelle, mais la composition fonctionnelle de l'interaction. Un produit desktop, mobile ou web peut présenter le même assemblage différemment.

8.7 Paramètres non visuels

Dans la mesure du possible, les paramètres UXfc doivent rester fonctionnels et non visuels. Exemples :

- ressource ou URI initiale ;
- source de données et source de données de sélection ;
- identifiant sélectionné et nom affiché ;
- mode de navigation ;
- politique d'historique ;
- prérequis de validation ;
- sélection exclusive, multiple ou conditionnelle ;
- exigence d'éléments homogènes pour la comparaison ;
- séquençement linéaire, réversible ou conditionnel.

Les paramètres d'apparence, tels que couleurs, thèmes visuels, jeux d'icônes et disposition exacte, doivent rester en dehors du modèle d'architecture réutilisable.

8.8 Coordination visuelle locale

Lorsque des composants UXfc sont assemblés dans une page en cours d'exécution, leurs événements locaux doivent normalement transiter par le contrôleur de page avant toute remontée au modèle applicatif. Un changement de sélection ou d'onglet, un rechargement de consultation ou une notification de DataSet peuvent modifier le modèle de page local sans devenir un appel serveur ou un événement métier. Le modèle détaillé d'exécution est décrit dans le chapitre sur `PageController` et l'émission d'événements visuels locaux. Son incarnation propre au web est décrite dans le chapitre sur la création de composants web fonctionnels réactifs.

8.9 Événements et actions abstraites

Les composants intentionnels et fonctionnels exposent des événements sémantiques et des actions abstraites. Un composant de consultation peut accepter l'intention `Browse(uri)` et émettre `ResourceLoaded` ou `NavigationFailed`. Un composant de sélection peut émettre `SelectionChanged`. Une vue tabulaire peut émettre des événements de publication ou de soumission. Un bouton peut émettre un événement d'action visuelle.

La règle essentielle est que les événements doivent décrire ce qui s'est produit sur le plan sémantique, et non l'implémentation du widget qui les a générés.

8.10 Règles de conception

Partez de l'intention de l'utilisateur avant de choisir un composant. Choisissez le composant fonctionnel avant son implémentation concrète. Enregistrez les implémentations concrètes par plateforme ou package. Séparez l'apparence des déclarations fonctionnelles. Utilisez des composants abstraits pour décrire les contrats d'interaction stables. Utilisez des événements pour relier le comportement des vues aux modèles de page, contrôleurs, objets de processus et actions métier.

Chapitre 9

ObjectBehind : l'objet comportemental d'une View

`ObjectBehind` correspond au comportement d'exécution attaché à une page, un cadre ou un composant visuel. Il est facile de le confondre avec un `ObjectItem`, car les deux sont des objets et peuvent exposer des actions. Pourtant, leurs responsabilités diffèrent.

9.1 Modèle mental

```
1 ObjectItem
2   business or conceptual object displayed by the page
3
4 ObjectBehind
5   local behavioral object of a page, frame or component
6
7 PageController
8   local coordinator and event bus for the page
9
10 DataSetsManager
11   registry of DataSet, DataSource and DataCursor surfaces
12
13 Visual component
14   rendered implementation bound to data and events
```

Exemple 9.1 – Responsabilités à l'exécution autour d'une vue

L'`ObjectItem` doit porter l'état et les capacités métier. L'`ObjectBehind` coordonne le comportement local de la vue : ouvrir ou fermer un cadre, réagir à un événement visuel, accéder à un `DataSet`, modifier l'état d'un composant ou déléguer une action métier.

9.2 Forme source et responsabilité de compilation

Un `ObjectBehind` extrait d'un contrat `VIEW` reste une classe `MODEL` ordinaire. Sa source peut être stockée dans un conteneur `*.models.yaml` voisin, référencé par `source`, `class` et `classId`. La découverte passe par `_.manifest.yaml`, dont le champ `fileName` désigne le conteneur physique multi-classes.

La couche `VIEW` est responsable de la liaison d'interaction, mais ne possède pas de second compilateur `MODEL`. La classe choisie emprunte le pipeline habituel de chargement et de compilation `MODEL`. Les projections natives, web et agentiques restent ainsi alignées sur les mêmes métadonnées comportementales exécutables.

9.3 Cycle de vie

Un cycle de vie courant est le suivant :

```
1 view metadata is loaded
2 -> PageController is created
3 -> page/frame ObjectBehind is instantiated
4 -> DataSetsManager is assembled
5 -> components are created and bound
6 -> local events call ObjectBehind actions
7 -> ObjectBehind coordinates data and visual state
8 -> view closes and local objects are released
```

Exemple 9.2 – Cycle de vie d’ObjectBehind

L’ordre précis de construction dépend de l’environnement d’exécution et du type de vue, mais le principe de responsabilité reste stable : un `ObjectBehind` appartient à la frontière de la vue locale.

9.4 Trouver un ObjectBehind

Les scripts View résolvent couramment les comportements locaux grâce au contrôleur de page :

```
1 lMaster := Self.PageController
2   .GetVisualComponentObjectBehindByName('Master');
3 lMaster.CloseForm();
```

Exemple 9.3 – Résolution d’un ObjectBehind visuel par son nom

Ou grâce à un identifiant visuel :

```
1 lComponent := Self.PageController.FindVisualComponentById(Self.Id);
```

Exemple 9.4 – Résolution d’un composant visuel par identifiant

Ces opérations relèvent de la coordination locale de l’interface. Elles ne doivent pas devenir des substituts cachés aux actions métier.

9.5 Relation avec les DataSet

Un `ObjectBehind` peut lire le contexte courant d’un `DataSet` :

```
1 lDs := Self.PageController.DataSetsManager.DataSetByName('TaskEntityDs');
2 lTask := lDs.ObjectItem;
```

Exemple 9.5 – Accès à un DataSet depuis ObjectBehind

La bonne séparation est la suivante :

```
1 ObjectBehind
2   decides how the view reacts
3
4 DataSet / DataSource / DataCursor
5   provide the local data and navigation context
6
7 ObjectItem action
8   enforces business behavior and persistence rules
```

Exemple 9.6 – Séparation des responsabilités

9.6 Flux d'événements

```
1 button click
2 -> local visual event
3 -> PageControler dispatch
4 -> ObjectBehind action
5 -> dataset update or business action
6 -> dependent component refresh
```

Exemple 9.7 – Flux typique d'un événement visuel

Ce modèle est développé dans les chapitres chapitre 13 «PageControler et émission locale d'événements visuels» et chapitre 15 «Cadres de bureau réactifs et routage des événements».

9.7 Ce qui relève d'ObjectBehind

Les responsabilités adaptées comprennent :

- l'initialisation locale d'une page ;
- la sélection ou l'actualisation d'un DataSet local ;
- la conversion d'un événement de composant en action publiée ;
- la coordination de vues enfants ;
- le contrôle local de la visibilité, de la sélection ou de la navigation ;
- l'émission d'un événement visuel local.

Évitez de placer directement dans ObjectBehind :

- la validation métier réutilisable ;
- des règles de domaine partagées entre plusieurs vues ;
- une coordination de longue durée ;
- la logique d'intégration relevant d'un service ou d'un fournisseur ;
- des règles de persistance devant aussi s'appliquer hors de l'interface.

9.8 Liste de contrôle du diagnostic

Lorsqu'une action visuelle ne s'exécute pas, vérifiez :

1. le nom ou l'identifiant du composant ;
2. le contrôleur de page auquel il appartient ;
3. le modèle ObjectBehind attaché à la vue ou au cadre ;
4. l'événement et le chemin d'action déclarés ;
5. le nom du DataSet utilisé par le script ;
6. si l'événement est local, routé par le parent ou de niveau applicatif.

Chapitre 10

functionalP pour débutants sur logiCells

Les métadonnées MODEL de logiCells sont écrites en YAML ; les sources visuelles VIEW utilisent `*.view.xml`. Il arrive qu'un modèle nécessite aussi un petit fragment exécutable : initialiser un état de page, valider un DataSet avant publication, créer un objet, actualiser une vue parente ou appeler une action d'exécution. Dans ces situations, les métadonnées utilisent souvent des scripts **functionalP**.

Ce chapitre n'est pas une référence complète du langage. Il présente le sous-ensemble pratique nécessaire aux débutants pour lire les scripts du premier tutoriel de gestion des tâches et des vues pilotées par les DataSet. La référence complète du langage figure dans l'annexe **functionalP** du *Guide du développeur* complémentaire. Une cartographie des sources pour les mainteneurs est conservée dans le dépôt, mais volontairement exclue du parcours de lecture des développeurs métier.

10.1 Où apparaît functionalP

Un débutant rencontre généralement **functionalP** à quatre endroits :

```
1 action script
2   Implements a declared logiCells action.
3
4 pageModel event handler
5   Reacts to page-level events such as OnShow.
6
7 dataset event handler
8   Reacts to dataset lifecycle events such as BeforeDataSetOpened
9   or BeforeDataSetPosted.
10
11 service or integration glue
12   Calls a published capability or prepares a request/response boundary.
```

Exemple 10.1 – Emplacements courants de functionalP

La règle essentielle est que **functionalP** n'est pas la syntaxe du modèle. MODEL reste en YAML. Le script est un fragment exécutable appartenant à une action ou un gestionnaire d'événements déclaré.

10.2 Script minimal dans les métadonnées

Un script est normalement placé dans une action. Le MODEL YAML environnant déclare l'action ; le script en fournit le corps exécutable.

En pratique, de nombreux scripts n'affectent pas explicitement **Result**. Ils produisent un effet, comme initialiser un **DataSet** ou publier un objet. Pour du code destiné aux débutants, utilisez **Result** lorsque l'action attend une valeur de retour : l'intention sera plus facile à déboguer.

10.3 Variables et affectation

L'opérateur d'affectation est **:=**. Les variables sont souvent introduites par affectation :

```
1 lTitle := 'New task';
2 lPriority := 1;
3 Result := lTitle;
```

Exemple 10.2 – Variables locales

Choisissez des noms de variables locales qui rendent les objets d'exécution reconnaissables. Une convention courante consiste à préfixer les variables locales par **l** : **lDs**, **lTask**, **lMem**, **lEvent**.

10.4 Self, Sender et Result

La plupart des scripts de métadonnées logiCells utilisent trois noms d'exécution :

Nom	Signification dans les scripts courants
Self	L'objet courant derrière l'action ou la page. Depuis Self , les scripts accèdent généralement à ObjectsContainer , PageController et à l'objet conceptuel courant.
Sender	L'objet ayant déclenché l'événement. Dans les événements de DataSet , Sender est souvent le DataSet en cours d'ouverture, de publication ou d'actualisation.
Result	La valeur de retour explicite d'un script lorsque le contrat de l'action en attend une.

Une erreur fréquente chez les débutants consiste à supposer que **Sender** désigne toujours un composant visuel. Dans les événements de **DataSet**, il s'agit normalement d'un **DataSet**. Dans les événements visuels, ce peut être un composant visuel ou son objet-behind.

10.5 Accéder à PageController

PageController est l'objet d'exécution local qui relie le modèle de page, le cache de données de la vue et l'arborescence des composants visuels.

```
1 lDs := Self.PageController.DataSetsManager.DataSetByName('TaskEntityDs');
```

Exemple 10.3 – Accès à un DataSet de page via PageController

Utilisez **PageController** lorsque le script appartient à une page, une vue, un **ObjectBehind** de composant ou un événement de vue piloté par **DataSet**. Ne l'employez pas à la place de la modélisation métier : c'est un objet de coordination locale pour la page courante.

10.6 Lire un DataSet

Le **DataSetsManager** donne accès aux **DataSet** déclarés dans le cache de la page.

```
1 lDs := Self.PageController.DataSetsManager.DataSetByName('TaskEntityDs');
2 lTask := lDs.ObjectItem;
```

Exemple 10.4 – Lecture du DataSet de la tâche éditée

Ce code n'interroge pas directement une base de données : il lit la surface de données de la page courante. Le fournisseur et la collection sous-jacents restent hors du composant visuel.

10.7 Initialiser un état local de page

Le premier tutoriel sur les tâches initialise un DataSet mémoire à l'ouverture de la page. C'est un bon modèle pour débutants, car il montre la différence entre données métier et état de page.

```
1 lMem := Self.PageController.DataSetsManager.DataSetByName('TaskMemoryDs');
2 if lMem.IsEmpty then begin
3   lMem.Append();
4   lMem.OnlyOpen := True;
5   lMem.PostData();
6 end;
```

Exemple 10.5 – Initialisation d'un DataSet de filtres local

TaskMemoryDs n'est pas une table persistante de tâches. Il stocke l'état des filtres locaux de la page. Si une même valeur doit survivre à la page, modélisez-la plutôt comme un champ de concept ou une facette persistante.

10.8 Filtrer avant l'ouverture d'un DataSet

Les événements de DataSet permettent à une vue d'adapter sa surface de données avant lecture par les composants.

```
1 if Sender.Name = 'TaskListDs' then begin
2   Sender.SetOrdering('Priority desc, DueDate asc');
3   lMem := Self.PageController.DataSetsManager.DataSetByName('TaskMemoryDs');
4   if lMem.OnlyOpen then
5     Sender.AddStringFilterItem('StatusName', '<>', 'Done');
6   if lMem.SearchText <> '' then
7     Sender.AddStringFilterItem('Title', 'contains', lMem.SearchText);
8 end;
```

Exemple 10.6 – Filtrage d'un DataSet avant son ouverture

Ce script constitue une règle de données de vue. Il ne doit pas contenir de politique métier fondamentale. Si la règle doit être imposée partout, déplacez-la vers une contrainte, une action ou une frontière de service du modèle.

10.9 Publier l'objet courant

Une action d'enregistrement lit généralement le DataSet courant, récupère son objet et le publie.

```
1 lDs := Self.PageController.DataSetsManager.DataSetByName('TaskEntityDs');
2 lTask := lDs.ObjectItem;
3 lTask.PostData();
```

Exemple 10.7 – Publication de l'objet courant

La publication des données est une opération d'exécution intentionnelle. La modification d'un champ à l'écran ne signifie pas nécessairement que l'objet conceptuel a été persisté ou propagé.

10.10 Créer un objet

Les actions de niveau classe ou les fabriques créent souvent un nouvel objet à partir du conteneur d'objets.

```

1 lClassTask := Self.ObjectsContainer.ClassItemByName('TaskManagement.Task');
2 lTask := Self.ObjectsContainer.CreateObjectItem(lClassTask, 1, '');
3 lTask.SetValue('Title', 'New task');
4 lTask.PostData();

```

Exemple 10.8 – Création d'un nouvel objet Task

Pour des métadonnées de production, préférez une action de création déclarée lorsqu'elle existe. La création directe dans un script est utile pour les tutoriels et l'orchestration interne, mais les workflows publics doivent exposer des actions stables.

10.11 Émettre un événement visuel

Après l'enregistrement d'un objet, une page peut devoir signaler à sa vue parente qu'une liste doit être actualisée.

```

1 lEvent := Self.PageController.ParentPageController.CreateVisualEvent(
2   'lgc.Visual.View.Refresh', nil, nil,
3   Self.ObjectsContainer.MetadataContainer, false);
4 Self.PageController.ParentPageController.DispatchVisualEvent(lEvent, StopNotification);

```

Exemple 10.9 – Émission d'un événement visuel d'actualisation vers le parent

Utilisez les événements visuels pour la coordination de l'affichage. N'en faites pas un bus caché d'événements métier. Si un fait métier a changé, représentez-le comme une action métier, un événement ou une transition de processus.

10.12 Appeler l'ObjectBehind d'un composant visuel

Certains scripts doivent appeler l'objet-behind local d'un composant de vue.

```

1 Self.PageController.GetVisualComponentObjectBehindByName('Master').CloseForm();

```

Exemple 10.10 – Fermeture de la vue maîtresse courante

Il s'agit d'une opération locale d'interface. Elle relève d'une action de page, et non d'une action de domaine destinée aux services ou à l'exécution en arrière-plan.

10.13 Quand déplacer la logique ailleurs

Un script `functionalP` doit rester court. Déplacez la logique lorsque le script devient une application cachée dans les métadonnées.

Si le script...	Déplacez la logique vers...
Impose une règle métier partout	une contrainte du modèle, une action de validation ou une étape de processus
Coordonne plusieurs objets dans le temps	un modèle de workflow ou de processus
Publie une capacité externe stable	une action déclarée et un contrat de service
Ne met à jour que l'état local de la vue	le modèle de page ou le gestionnaire d'événements de DataSet
Ne fait qu'actualiser ou router des composants visuels	un événement visuel ou un modèle de cadre réactif

10.14 Erreurs fréquentes des débutants

Erreur	Correction
Considérer <code>functionalP</code> comme le langage du modèle	Conservez les métadonnées MODEL en YAML. Réservez <code>functionalP</code> aux fragments exécutables déclarés.
Placer les règles métier dans un événement de vue	Inscrivez les règles pérennes dans les concepts, contraintes, actions ou workflows.
Supposer que <code>Sender</code> a toujours le même type	Vérifiez la famille de l'événement : DataSet, visuel, processus, service ou message.
Appeler un composant depuis une action de service	Les appels à l'ObjectBehind d'interface relèvent des actions de page locales, non des frontières de services.
Oublier de publier les données	Définissez les valeurs sur l'objet courant, puis appelez <code>PostData()</code> si la modification doit être validée.
Laisser la valeur de retour implicite	Affectez <code>Result</code> lorsque le contrat de l'action attend une valeur de retour.

10.15 Parcours de lecture pratique

Pour un premier projet, apprenez `functionalP` dans cet ordre :

1. lire les variables et l'affectation ;
2. comprendre `Self`, `Sender` et `Result` ;
3. accéder à `PageControler.DataSetsManager` ;
4. lire l'objet courant d'un DataSet ;
5. publier les données ;
6. émettre un événement visuel local ;
7. créer un objet seulement après avoir compris les chemins de chargement et de vue.

Troisième partie

État client, cache et routage des événements

Chapitre 11

DataSet, DataCursor et DataSource comme cache de données de la vue

Une vue logiCells n'est pas qu'une composition visuelle : c'est aussi un espace local de coordination des données. Les composants UX fonctionnels ne se lient normalement pas directement à une table de base de données, un service distant ou une collection métier brute. Ils se lient à une surface de données locale exposée par la page via **DataSource**, **DataSet** ou **DataCursor**.

Cette distinction importe, car la vue doit rester réactive, navigable et cohérente localement même lorsque la collection métier sous-jacente est distante, paginée, calculée, filtrée ou synchronisée de façon asynchrone.

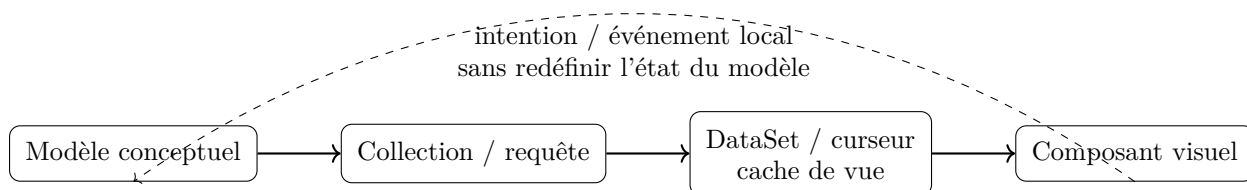


FIGURE 11.1 – DataSet et curseurs servent de projections et de caches aux vues ; le modèle conceptuel reste la source de l'état métier.

11.1 Cache de données de la vue

Le cache de données de la vue est la couche située entre les composants fonctionnels et les collections d'exécution.

```
1 UXfc functional component
2   -> PageControler / PageModel
3   -> DataSource / DataSet / DataCursor
4   -> Collection / Provider
5   -> local memory, SQL, service, stream,
6       remote provider, or computed source
```

Exemple 11.1 – Chaîne du cache de données de la vue

Le composant fonctionnel voit une surface de données stable au niveau de la vue. Il n'a pas besoin de savoir si les données proviennent d'une collection en mémoire, d'une collection persistante, d'un fournisseur distant, d'un appel de service ou d'un fournisseur calculé.

Les vues logiCells bénéficient ainsi d'une séparation nette des responsabilités.

Couche	Responsabilité
PageController	Coordonne les événements visuels et les intentions d'interaction locales.
PageModel	Coordonne l'état fonctionnel de la page.
DataSource, DataSet, DataCursor	Exposent la surface de données navigable locale utilisée par la vue.
Collection/fournisseur	Fournit la source de données d'exécution, éventuellement locale, persistante, calculée, paginée, diffusée en flux ou distante.

11.2 Pourquoi les composants fonctionnels ne doivent pas gérer l'accès aux données

Un composant fonctionnel représente une fonction d'interaction : sélection, parcours, consultation, édition, validation, navigation, comparaison ou affichage. Il ne doit pas décider comment les données sont récupérées, mises en cache, synchronisées ou persistées.

Un composant UXfc doit donc normalement se lier à une surface de données de niveau page, et non directement à un fournisseur.

```

1  tabularView
2    binds to a DataSource or DataSet
3
4  detailView
5    binds to the current object exposed by a DataCursor
6
7  splitView
8    binds one region to a master cursor
9    and another region to a dependent DataSet
10
11 autoCompleteSelect
12   binds to a lookup DataSource,
13   possibly backed by a remote or filtered provider

```

Exemple 11.2 – Liaisons UXfc typiques aux surfaces de données locales

Le composant demeure fonctionnel et déclaratif. La page décide de la résolution de la surface de données.

11.3 DataSet, DataSource et DataCursor

Ces noms reprennent volontairement un vocabulaire de liaison de données familier aux développeurs d'applications dont l'implémentation suit le style de l'environnement d'exécution natif. L'analogie fournit un premier repère utile : un DataSet expose des enregistrements, une DataSource est une surface de liaison et un curseur identifie le contexte de navigation courant. logiCells étend ce modèle grâce à **PageController**, **DataSetsManager**, aux collections et aux fournisseurs ; l'analogie ne constitue donc pas une spécification complète de l'exécution.

Les trois notions jouent des rôles complémentaires.

Un **DataSet** représente un ensemble local d'enregistrements ou d'objets exposé à la vue. Il peut mettre en cache une collection, un sous-ensemble filtré, un résultat de requête ou une projection calculée.

Une **DataSource** est la surface de liaison utilisée par les composants visuels ou fonctionnels. Elle leur fournit un point d'accès stable sans exposer l'implémentation du fournisseur sous-jacent.

Un **DataCursor** représente la position courante dans un DataSet ou une collection. Il est essentiel aux vues maître-détail, cadres imbriqués, vues dépendantes et à la propagation du contexte entre composants.

Ensemble, ils constituent la mémoire locale de données de la page.

```
1 DataSet
2   answers: what data is visible here?
3
4 DataSource
5   answers: what can this component bind to?
6
7 DataCursor
8   answers: what is the current object or current position?
```

Exemple 11.3 – Trois questions complémentaires

11.4 Relation avec les collections et fournisseurs

Les collections et les fournisseurs appartiennent à la couche de données de l'environnement d'exécution. Ils produisent, chargent, filtrent, synchronisent ou persistent les objets.

Un fournisseur peut être local ou distant. Il peut encapsuler une collection en mémoire, une requête SQL, un point de terminaison REST, un flux, une collection calculée, une source de messages ou un service paginé. La vue n'a pas besoin d'en connaître directement la nature : elle se lie à la surface de données locale exposée par la page.

```
1 Functional component
2   -> DataSource
3   -> DataSet
4   -> Collection
5   -> Provider
6   -> actual source
```

Exemple 11.4 – Abstraction du fournisseur derrière un DataSet de vue

Ce modèle est particulièrement important pour les applications distribuées. Une même vue peut utiliser un fournisseur local pendant les tests, un fournisseur de base de données dans un déploiement natif ou un fournisseur distant dans un déploiement web.

11.5 Maître-détail et propagation du contexte

La navigation maître-détail est une utilisation courante du cache de données de la vue.

```
1 Master DataCursor
2   current object: WorkItem
3
4 Detail DataSet
5   collection path: WorkItem.Attachments
6
7 Detail DataSource
8   used by: attachment tabularView
```

Exemple 11.5 – Propagation d'un curseur maître-détail

Lorsque le curseur maître change, le DataSet dépendant peut s'actualiser localement. Un événement visuel peut être émis via `PageController`, mais cela ne signifie pas automatiquement que l'objet métier a changé.

```
1 master cursor changed
2   -> PageController.DispatchVisualEvent(...)
3   -> PageModel updates dependent DataSet
4   -> detail component refreshes
```

Exemple 11.6 – Événement de curseur local

L'événement reste local, sauf si le modèle de page l'associe explicitement à une action applicative ou à une opération du modèle conceptuel.

11.6 Fournisseurs distants et réactivité locale

Avec un fournisseur distant, la distinction devient encore plus importante. Un composant React ou un composant visuel natif ne doit pas appeler la source distante pour chaque petite interaction. La page expose plutôt un cache local et détermine quand actualiser, synchroniser ou interroger le fournisseur.

```
1 React component
2   -> DataSource in page state
3   -> local DataSet cache
4   -> provider adapter
5   -> remote collection service
```

Exemple 11.7 – Fournisseur distant avec cache local

La sélection d'un onglet, le déplacement local dans les lignes, un filtre temporaire ou le déploiement d'un panneau peuvent ne modifier que le cache local. Une validation, un enregistrement, une recherche ou une transition de processus peuvent franchir la frontière vers le fournisseur ou le serveur.

11.7 Assemblage déclaratif des DataSet

Une vue peut déclarer les DataSet, curseurs et sources de données qui composent son cache local. Cette déclaration relève de la couche vue/cache ; ce n'est pas l'endroit où les rôles du modèle sont classés comme emplacements ou attributs.

```
1 views:
2   - WorkItemReview:
3     pageModel: WorkItemReviewPageModel
4     dataSets:
5       - MainCursor:
6         dataSet:
7           name: WorkItems
8           collectionName: WorkItems
9           provider: WorkItemProvider
10          fields:
11            - Id:
12              type: String
13            - ClassId:
14              type: String
15            - Title:
16              type: String
17            - Status:
18              type: String
19          dataSource: WorkItemDataSource
```

Exemple 11.8 – Assemblage des DataSet dans une vue YAML

L'attribut `kind` est volontairement absent ici. Il n'est utilisé que sur les déclarations de `field` du modèle dans les facettes, pour classer un champ comme rôle conceptuel : `kind="slot"` pour un rôle structurel ou `kind="attribute"` pour un rôle descriptif. `DataSet`, `DataSource` et `DataCursor` relèvent de la couche cache/liaison de la vue et non de celle de la définition des rôles.

11.8 Composants fonctionnels liés au cache de la vue

Une fois les surfaces de données disponibles, les composants fonctionnels peuvent être assemblés sans connaître l'implémentation du fournisseur.

```
1 components:
2   - tabularView:
3     name: WorkItemList
4     dataSource: WorkItemDataSource
```

```

5     events:
6         - lgcVisual.Data.CurrentChanged local
7
8     - detailView:
9         name: WorkItemSummary
10        cursor: MainCursor
11        events:
12            - lgcVisual.Detail.FieldChanged local
13
14    - actionPanel:
15        name: ReviewActions
16        cursor: MainCursor
17        actions:
18            - RequestValidation
19            - ReloadFromProvider

```

Exemple 11.9 – Composants fonctionnels liés aux surfaces de données locales

La vue tabulaire consomme une `DataSource`. La vue de détail consomme un curseur. Le panneau d’actions peut utiliser l’objet courant du curseur comme contexte d’action. Aucun de ces composants n’a besoin de savoir si la collection sous-jacente est locale ou distante.

11.9 Relation avec PageController

Le cache de données de la vue travaille de concert avec `PageController`.

```

1 visual event
2     -> PageController
3     -> PageModel
4     -> DataCursor / DataSet update
5     -> DataSource notification
6     -> functional component refresh

```

Exemple 11.10 – Médiation des événements et données dans une vue

`PageController` sert d’intermédiaire pour les événements locaux. La couche `DataSet` assure la médiation des données locales. Ensemble, ils empêchent un couplage inutile entre couche visuelle et fournisseurs d’exécution.

11.10 Règles de conception

Un composant fonctionnel ne doit pas gérer directement l’accès aux données. Il doit se lier à une surface de données nommée exposée par la page.

Une vue doit distinguer les modifications de `DataSet` des modifications de collection. Un changement de `DataSet` peut ne mettre à jour que le cache de la vue. Un changement de collection relève de la couche de modèle d’exécution et peut nécessiter synchronisation, persistance ou notification de processus.

Un fournisseur distant doit être masqué derrière une surface de données locale. La vue ne doit pas devenir un client d’API distante pour chaque interaction visuelle.

Nommez explicitement les `DataSet`, `DataSource` et curseurs dans les modèles de vues réutilisables. Leurs noms font partie du contrat View, au même titre que ceux des composants et événements.

Chapitre 12

Événements DataSetManager et vues pilotées par les DataSet

Les chapitres précédents ont présenté le cache de données de la vue : les surfaces `DataSet`, `DataSource` et `DataCursor` exposées par la page. Dans les applications réelles, ce cache n'est pas passif. Il émet et achemine des événements lorsque des données sont ouvertes, parcourues, publiées, annulées, filtrées, actualisées ou synchronisées.

Ce chapitre décrit le contrat d'événements autour de `DataSetManager`. Dans le code d'exécution, l'objet est souvent nommé `DataSetsManager`, car il gère plusieurs `DataSet`, `DataSource` et curseurs au niveau page. Conceptuellement, il est le gestionnaire de la surface de données de la vue.

12.1 Pourquoi les événements des DataSet sont importants

Une vue pilotée par `DataSet` ne manipule pas directement une base de données, un point de terminaison REST ou une collection métier. Elle manipule des surfaces de données locales qui doivent informer la page de tout changement pertinent.

```
1 User or component interaction
2   -> visual component
3   -> DataSource / DataSet / DataCursor
4   -> DataSetManager event
5   -> PageController / PageModel
6   -> component controller or objectBehind
7   -> optional application action
8   -> optional provider or service boundary
```

Exemple 12.1 – Chaîne d'événements pilotée par `DataSet`

Cette chaîne événementielle explique pourquoi un composant web fonctionnel peut rester réactif sans être propriétaire du fournisseur. Un tableau peut se recharger lorsqu'un `DataSet` change. Une vue de détail peut s'actualiser au déplacement d'un curseur. Un formulaire peut valider un `DataSet` avant publication. Une zone de recherche peut mettre à jour un `DataSet` de filtres et actualiser un `DataSet` de résultats. La page réagit localement sans transformer chaque geste de l'utilisateur en appel distant.

12.2 Un événement DataSetManager n'est pas automatiquement un événement métier

Un événement de `DataSet` décrit d'abord l'état de la surface de données de la vue. Il indique qu'un `DataSet` va s'ouvrir, s'est ouvert, va être publié, a été publié, a été annulé, a changé de

ligne courante, s'est actualisé ou n'a pas pu être chargé. C'est un fait relatif au cache de la vue, pas nécessairement au domaine métier.

Un événement **DataSetManager** ne devient un événement applicatif ou métier que lorsque le modèle de page, son contrôleur ou le contrôleur d'un composant le fait explicitement remonter.

La distinction est essentielle. La sélection d'une ligne n'est pas une transition métier. Une actualisation n'est pas un événement de processus. La modification d'un champ n'est pas nécessairement une modification persistée. Un événement **BeforeDataSetPosted** constitue un point de contrôle avant une éventuelle écriture. Seule l'action associée décide si l'événement valide, annule, synchronise ou fait remonter le changement.

12.3 Familles d'événements

Le contrat d'événements peut se répartir en cinq familles.

Famille	Signification	Usage habituel
Événements de résolution	Un DataSet, une DataSource ou un curseur est enregistré ou résolu par nom.	Initialisation des composants, diagnostic de liaison, gestion des sources manquantes.
Événements de cycle de vie	Un DataSet s'ouvre, se ferme, se recharge, s'actualise ou ne peut être chargé.	Indicateurs de chargement, nouvelles tentatives, actualisation des dépendants.
Événements de navigation	La ligne ou l'objet courant change.	Propagation maître-détail, panneaux d'actions, inspecteurs et vues dépendant de la sélection.
Événements d'édition/-publication	Une ligne est modifiée, publiée, annulée, insérée ou supprimée.	Validation, champs calculés, suivi des modifications non enregistrées, boutons Enregistrer.
Événements de synchronisation	Un DataSet échange des modifications avec un fournisseur, un service ou une collection distante.	Coordination en/hors ligne, remontée des conflits, actualisation après écriture du fournisseur.

Les noms exacts peuvent varier selon les générations de l'environnement d'exécution et les anciens espaces de noms. Le contrat fonctionnel demeure : un événement de données au niveau page donne à celle-ci l'occasion de mettre à jour les composants locaux, de valider l'état ou d'appeler une action de niveau supérieur.

12.4 Événements observés dans des vues ERP pilotées par DataSet

Une application ERP commerciale fournit un modèle utile. Ses métadonnées peuvent évoluer, mais ses vues montrent clairement un comportement piloté par DataSet. Elles déclarent les DataSet et DataSource, lient champs et tableaux aux sources de données et utilisent le **DataSetsManager** de la page pour récupérer les DataSet par nom.

Un formulaire type déclare une DataSource d'entité pour les champs et plusieurs DataSource de référence ou de liste pour les sélecteurs et tableaux.

La vue ne demande pas au composant de trouver la collection métier. Le composant se lie à la DataSource. La couche de données de page résout le DataSet et son fournisseur.

12.5 Avant publication : validation par un événement de DataSet

Le modèle d'événement le plus explicite est la validation avant la publication d'un DataSet.

L'action reçoit l'émetteur de l'événement et ses paramètres modifiables. Elle peut obtenir le DataSet depuis le gestionnaire de page, inspecter les données courantes et décider si la publication est autorisée.

```
1 procedure VerificationData(Sender, var CanPost, var Message);
2 var
3   EntityDataSet: Object;
4 begin
5   EntityDataSet := Sender.DataSetsManager.DataSetByName('Entity');
6
7   if EntityDataSet.Amount = 0 then
8   begin
9     CanPost := false;
10    Message := 'An amount must be provided.';
11  end;
12 end;
```

Exemple 12.2 – Obtention du DataSet dans une action avant publication

L'événement n'est pas la règle de validation en elle-même : c'est le point d'extension permettant au modèle de page ou à l'ObjectBehind d'appeler une action de validation. L'action peut rester locale, appeler une méthode d'objet ou remonter à un service si la validation nécessite des informations distantes.

12.6 Après publication : actualiser les vues dépendantes

Un second modèle courant est l'événement après publication. Une fois le DataSet publié avec succès, la page peut actualiser les DataSet dépendants, mettre à jour la disponibilité des actions, reconstruire une tuile de tableau de bord ou déplacer le curseur vers l'enregistrement créé.

```
1 procedure AfterDataSetPosted(Sender);
2 var
3   Items: Object;
4 begin
5   Items := Sender.DataSetsManager.DataSetByName('DataSetListItems');
6   if Assigned(Items) then
7     Items.Refresh();
8 end;
```

Exemple 12.3 – Actualisation d'une liste dépendante après publication

Cette logique doit rester limitée à la page lorsqu'elle ne sert qu'à maintenir la cohérence de la vue. Elle devient événement applicatif uniquement si la publication a une portée métier et doit informer un autre processus, agent ou service.

12.7 DataSet de filtres et recherches réactives

Les vues ERP utilisent souvent un petit DataSet de filtres pour piloter un DataSet de résultats plus important. Les champs de recherche se lient à la DataSource de filtres. Lorsque l'utilisateur modifie un filtre ou lance une recherche, la page lit le DataSet de filtres et applique les filtres du fournisseur au DataSet de résultats.

```
1 procedure ApplyFilter(DataSetsManager);
2 var
3   FilterDataSet: Object;
4   ListDataSet: Object;
5 begin
6   FilterDataSet := DataSetsManager.DataSetByName('DataSetFilter');
7   ListDataSet := DataSetsManager.DataSetByName('DataSetListSalesDocuments');
```

```

8
9   ListDataSet.Provider.ClearFilters();
10
11   if FilterDataSet.SearchName <> '' then
12       ListDataSet.Provider.AddStringFilterItem(
13           'Name', 'like', FilterDataSet.SearchName);
14
15   ListDataSet.Refresh();
16 end;

```

Exemple 12.4 – Application des filtres via DataSetsManager

Pour un composant web réactif, ce modèle se traduit naturellement par une temporisation anti-rebond (*debouncing*). Le composant peut conserver l'état local de la saisie pendant la frappe. Ce n'est qu'après un délai ou une demande explicite qu'il met à jour le DataSet de filtres et déclenche l'actualisation des résultats.

12.8 Modifications de curseur et vues maître-détail

Un événement de curseur est un événement de DataSet portant sur le contexte. Il indique que la ligne, l'objet ou la sélection courante a changé. C'est le fondement des vues maître-détail.

```

1 Master table selection changed
2 -> MainSourceCursor current object changed
3 -> DataSetManager emits cursor/data change event
4 -> PageModel updates dependent DataSet
5 -> detail datasource refreshes
6 -> detail component rerenders

```

Exemple 12.5 – Chaîne maître-détail pilotée par curseur

Parmi les composants dépendants figurent les inspecteurs d'objets, listes d'actions, tableaux de relations, totaux, lignes de documents, boutons contextuels et panneaux secondaires. L'objet courant porté par le curseur est le contexte transmis à ces composants.

12.9 Paramètres des événements

Les événements de DataSet doivent fournir assez d'informations pour que la page réagisse sans recherche globale. Une charge utile pratique comprend la classe de l'événement, le DataSet source, l'émetteur, le nom de DataSource ou de curseur et d'éventuels paramètres modifiables.

Paramètre	Rôle
Sender	Objet d'exécution qui a émis ou transmis l'événement. Donne souvent accès à DataSetsManager .
DataSet	DataSet concerné par l'événement, utile pour un gestionnaire générique.
DataSetName	Nom stable servant à retrouver le DataSet dans le gestionnaire.
DataSourceName	Surface de liaison utilisée par les composants visuels.
CursorName	Curseur dont l'objet ou la position courante a changé.
CanPost	Booléen modifiable utilisé par la validation avant publication.
Message	Message de validation ou de diagnostic renvoyé à la page.
StopNotification	Indicateur de contrôle de propagation lorsqu'un événement est consommé localement.

La règle de conception majeure consiste à transmettre explicitement le contexte. Un gestionnaire de page ne devrait pas avoir à déduire le DataSet actif depuis un état visuel sans rapport alors que l'événement sait déjà quel DataSet a changé.

12.10 Correspondance avec le web réactif

Dans une vue web, le même contrat peut être représenté par un abonnement événementiel au niveau page. Le composant se lie à une DataSource, s'abonne aux notifications pertinentes du DataSet et met à jour son état React local lorsque celui-ci change.

```
1 componentDidMount() {
2   const pc = this.props.pc;
3   const dsName = this.props.dataStructure.dataSet;
4
5   this.subscription = pc.dataSetManager.subscribe(
6     dsName,
7     'AfterOpen',
8     () => this.refreshFromDataSet(dsName)
9   );
10 }
11
12 componentWillUnmount() {
13   if (this.subscription) {
14     this.subscription.unsubscribe();
15   }
16 }
```

Exemple 12.6 – Abonnement de style React aux notifications de DataSet

Un composant peut également s'abonner aux changements de curseur.

```
1 pc.dataSetManager.subscribeCursor(
2   'MainSourceCursor',
3   'CurrentObjectChanged',
4   evt => this.setState({ currentObject: evt.currentObject })
5 );
```

Exemple 12.7 – Notification de curseur de style React

Il s'agit de l'implémentation web de la même règle logiCells : le composant écoute la surface de données de la page, pas directement le fournisseur.

12.11 Noms d'événements recommandés

Un modèle public peut documenter les événements des DataSet selon leur fonction et non selon leur classe d'implémentation. Le catalogue suivant suffit à la plupart des conceptions de vues.

Événement	Signification
BeforeDataSetOpen	Le DataSet va s'ouvrir ou résoudre son fournisseur.
AfterDataSetOpen	Le DataSet est disponible pour la liaison et le rendu.
BeforeDataSetRefresh	Une actualisation va être exécutée.
AfterDataSetRefresh	L'actualisation est terminée; les composants dépendants peuvent se mettre à jour.
DataSetLoadFailed	Le fournisseur n'a pas permis de charger le DataSet.

BeforeDataSetPosted	Une publication va se produire mais peut encore être annulée.
AfterDataSetPosted	La publication a réussi ; les surfaces dépendantes peuvent s'actualiser.
AfterDataSetCanceled	L'édition locale a été annulée.
CurrentObjectChanged	Le curseur pointe désormais vers un autre objet ou une autre ligne.
FieldChanged	La valeur d'un champ a changé dans le cache de la vue.
ProviderSyncStarted	La synchronisation avec un fournisseur a démarré.
ProviderSyncCompleted	La synchronisation avec un fournisseur est terminée.
ProviderSyncFailed	La synchronisation a échoué ; la page doit exposer les diagnostics.

Ces noms sont volontairement fonctionnels. Une implémentation peut les placer dans un espace de noms d'exécution comme `logiCells.Visual.Data.*` ou `lgcVisual.Data.*`. Le modèle doit préserver la distinction sémantique entre cycle de vie, navigation, édition/publication et synchronisation.

12.12 Assemblage événementiel déclaratif

Le contrat d'événements des `DataSet` peut s'exprimer au niveau de la vue. La déclaration indique quel événement de données la page écoute, quelle action appeler et quels paramètres transmettre.

```

1 views:
2   - SalesDocumentForm:
3     pageModel: SalesDocumentPageModel
4     dataSets:
5       - Entity:
6         collectionName: Entity
7         dataSource: EntityDataSource
8       - DataSetLines:
9         collectionName: DocumentLines
10        dataSource: LineDataSource
11    events:
12      - BeforeDataSetPosted:
13        class: logiCells.Visual.Data.BeforeDataSetPosted
14        action: ValidateBeforePost
15        params:
16          - Sender: Sender
17          - CanPost: CanPost
18          - Message: Message
19      - AfterDataSetPosted:
20        class: logiCells.Visual.Data.AfterDataSetPosted
21        action: RefreshAfterPost
22        params:
23          - Sender: Sender

```

Exemple 12.8 – Assemblage des événements `DataSet` dans une vue YAML

12.13 Règles de conception

Les vues pilotées par `DataSet` sont puissantes, mais demandent des frontières claires.

- Un composant se lie à une `DataSource` ; il ne doit pas résoudre directement les fournisseurs.

- Une action de page récupère les `DataSet` via `DataSetsManager`, pas via les détails internes des composants visuels.
- Un événement de cycle de vie de `DataSet` met d’abord à jour la cohérence de la vue ; il ne remonte que par association explicite.
- Un événement avant publication est le bon endroit pour la validation et l’annulation au niveau page.
- Un événement après publication est le bon endroit pour l’actualisation des dépendants et le maintien de la cohérence locale.
- Les événements de curseur doivent piloter le contexte maître-détail, la disponibilité des actions et les inspecteurs.
- Les événements de synchronisation avec un fournisseur doivent être séparés des événements de navigation locale entre lignes.
- L’attribut `kind` ne s’utilise jamais sur `DataSet`, `DataSource` ou `DataCursor` ; il appartient exclusivement aux déclarations de `field` du modèle dans les facettes.

On obtient ainsi une architecture de vues réactive, local-first et indépendante des fournisseurs. Les composants réagissent aux notifications des `DataSet`, les modèles de page en interprètent le sens et les actions métier ne sont invoquées que lorsque la vue franchit explicitement la frontière entre cohérence de la surface de données et intention applicative.

Chapitre 13

PageController et émission locale d'événements visuels

Les vues logiCells sont réactives, mais toute réaction n'est pas un événement métier et tout événement visuel n'a pas vocation à franchir une frontière d'exécution. La couche View possède son propre espace de coordination. Dans cet espace, **PageController**, **PageModel** et les objets **objectBehind** des composants forment une architecture locale d'émission et de routage des événements visuels.

Ce chapitre explicite cette architecture. Il complète les chapitres UXfc en montrant comment les composants intentionnels et fonctionnels sont assemblés dans une page en cours d'exécution, sans transformer chaque clic, changement d'onglet ou de sélection, ou notification de DataSet en appel serveur ou en mutation du conteneur d'objets conceptuels.

13.1 Pourquoi émettre les événements visuels localement

Une vue contient de nombreuses interactions utiles à l'interface mais pas nécessairement au modèle de domaine. Par exemple : sélectionner un onglet, déployer un panneau, changer la ligne courante d'un DataSet client, mettre à jour une zone d'état, afficher un cadre secondaire, recharger un composant de consultation ou synchroniser un contrôle de sélection local.

Ces interactions doivent d'abord être interprétées localement par la page. Elles peuvent ensuite remonter vers le modèle applicatif, un processus, une action publiée, un appel de service ou la mémoire conceptuelle, mais cette remontée est un choix de conception. Elle n'est pas automatique.

```
1 visual component
2   -> local visual event
3   -> PageController
4   -> PageModel or component objectBehind
5   -> local view reaction
6   -> optional escalation to action, process, service or container
```

Exemple 13.1 – Principe d'événement visuel local

La règle essentielle est simple : un événement visuel n'est pas automatiquement un événement métier, un appel serveur ni une mutation du conteneur.

13.2 PageController comme bus d'événements local

Une page logiCells possède un contrôleur général, traditionnellement nommé **PageController**. Il reçoit les événements adressés à la vue et les distribue aux composants internes de la page. C'est le bus local des événements de la vue affichée.

PageController gère plusieurs responsabilités locales :

- recevoir les événements visuels émis par les composants fonctionnels ;
- identifier la source et l'émetteur d'un événement ;
- distribuer l'événement à `PageModel` ;
- transmettre l'événement aux modèles des composants ou aux objets `objectBehind` ;
- arrêter la propagation lorsqu'une réaction locale consomme l'événement ;
- faire remonter l'événement uniquement si le modèle de vue déclare cette remontée nécessaire.

Cette architecture est particulièrement importante pour les vues composées de composants UXfc. Le composant UXfc déclare l'intention d'interaction. Le composant fonctionnel en déclare la fonction. `PageController` coordonne le flux d'événements entre les composants concrets et le modèle de page local.

13.3 PageModel et objectBehind des composants

`PageModel` est le modèle de la page elle-même. Il peut être considéré comme le cadre local de plus haut niveau qui contient et coordonne les composants fonctionnels de l'interface.

Chaque composant peut aussi posséder son propre `objectBehind`, parfois simplement appelé modèle de composant. Celui-ci interprète les événements d'une zone fonctionnelle de la vue : vue à onglets, vue fractionnée, cadre de consultation, sélecteur à autocomplétion, panneau de navigation, vue tabulaire ou autre composant fonctionnel.

```

1 PageController
2   receives and routes visual events
3
4 PageModel
5   coordinates the page-level reaction
6
7 component objectBehind
8   handles the local behavior of one functional component
9
10 ObjectsContainer
11   receives only events that are explicitly escalated to the model/runtime layer

```

Exemple 13.2 – Rôles de coordination de la vue locale

Cette séparation évite que le conteneur d'objets reçoive chaque notification visuelle. Elle empêche également de faire du serveur le coordinateur par défaut de comportements d'interface purement locaux.

13.4 Vues React et émission côté client

Le même principe s'applique dans une vue React. Un composant React concret peut émettre un événement local vers un objet contrôleur de page. Cet événement peut actualiser l'état local, sélectionner un onglet, actualiser un `DataSet` local, notifier un composant frère ou modifier un contexte de consultation sans appeler immédiatement le serveur.

```

1 function onTabSelected(tabId) {
2   pageController.dispatchEvent({
3     eventClass: 'lgcVisual.TabbedView.TabSelected',
4     source: 'ReviewTabs',
5     sender: 'TabHeader',
6     params: { tabId: tabId }
7   });
8 }

```

Exemple 13.3 – Émission locale de style React

Cette émission reste locale à la vue, sauf si le modèle de page l'associe à une action, un événement de processus, un service publié ou une règle de synchronisation distante.

13.5 Vues natives et DispatchVisualEvent

Dans une vue native, la même logique passe par une méthode de contrôleur comme `DispatchVisualEvent`. Un gestionnaire de `DataSet`, un panneau de navigation, un cadre ou un composant visuel peut construire un événement visuel et le soumettre au contrôleur de page.

```
1 procedure DataSetsManager.AfterDataSetCanceled(DataSet: DataSet);
2 var
3   StopNotification: Boolean;
4 begin
5   if Assigned(PageController) then
6     PageController.DispatchVisualEvent(
7       GlobalVisualEventElement.Create(
8         'lgcVisual.Data.AfterDataSetCanceled',
9         DataSet,
10        DataSet),
11        StopNotification);
12 end;
```

Exemple 13.4 – Émission d'événements visuels en mode natif

Le contrôleur peut ensuite notifier `PageModel` et les instances `objectBehind` concernées. Si l'un des gestionnaires consomme l'événement, la propagation peut s'arrêter. Si l'événement est associé à une action de niveau supérieur, la vue peut le faire remonter explicitement.

13.6 Quatre niveaux d'événements

Une classification claire des événements évite couplage et confusion.

Niveau	Signification	Récepteur habituel
Événement visuel local	Signal de coordination de l'interface : onglet sélectionné, séquence modifiée, panneau ouvert, champ ayant le focus, curseur de <code>DataSet</code> déplacé.	<code>PageController</code> , <code>PageModel</code> , <code>objectBehind</code> du composant.
Événement d'intention UXfc	Signal d'interaction sémantique : <code>Browse(uri)</code> , <code>Reload</code> , <code>SelectionChanged</code> , <code>ValidationRequested</code> .	Contrôleur UXfc, contrôleur de composant fonctionnel, modèle de page.
Événement applicatif	Signal de coordination entre vues, contrôleurs ou processus : élément ouvert, contexte modifié, révision demandée.	Contrôleur de page/session, contrôleur de processus, modèle applicatif.
Événement métier ou de processus	Fait du domaine : commande validée, processus terminé, message reçu, tâche attribuée.	Conteneur d'objets, objet de processus, agents, services, gestionnaires de messages.

Le même environnement d'exécution peut acheminer tous ces événements, mais le modèle doit en distinguer les intentions.

13.7 Correspondance déclarative des événements

Le modèle de page peut déclarer comment les événements visuels locaux sont associés à des actions locales, des réactions de page ou des opérations de niveau supérieur.

L'attribut `local` sert ici de marqueur documentaire : il distingue les réactions internes à la vue de celles qui franchissent volontairement la frontière vers une action applicative ou un

processus du domaine. L'environnement d'exécution concret peut coder cette distinction grâce aux classes d'événements, méthodes d'émission, portées de contrôleurs, chemins d'actions ou indicateurs de propagation.

13.8 DataSet, collections et frontières d'exécution

Une règle pratique découle du modèle MVC de logiCells : les DataSet sont des caches côté vue, tandis que les collections sont des données conceptuelles côté modèle. Une notification de DataSet relève souvent de la partie visuelle de l'application. Un changement de collection relève du côté modèle conceptuel.

```
1 DataSet event
2   -> local view cache changed
3   -> PageController.DispatchVisualEvent(...)
4   -> PageModel/component reaction
5
6 Collection event
7   -> conceptual model changed
8   -> ObjectsContainer/EventManager/process reaction
```

Exemple 13.5 – Frontière entre événements de DataSet et de collection

Cette distinction est centrale pour l'architecture web. Dans une vue React, un DataSet peut résider dans l'état de vue du navigateur, tandis que la collection correspondante reste du côté serveur ou exécution. Émettre un événement de DataSet ne doit pas impliquer automatiquement un appel serveur.

13.9 Assemblages UXfc et événements locaux

Les assemblages UXfc gagnent en précision lorsque leurs contrats d'événements identifient la couche de coordination locale.

```
1 view: WorkItemReview
2 pageController: WorkItemReviewPageController
3 pageModel: WorkItemReviewPageModel
4 components:
5   - tabbedView: ReviewTabs
6     events:
7       - lgcVisual.TabbedView.TabSelected local
8   - consultation: ReferenceDocument
9     events:
10      - lgcVisual.Consultation.ResourceLoaded local
11      - lgcVisual.Consultation.NavigationFailed local
12   - validation: ReviewDecision
13     events:
14      - lbs.WorkItem.ValidationRequested escalated
```

Exemple 13.6 – Assemblage UXfc avec émission d'événements locaux

Les premiers événements coordonnent la page locale. Le dernier franchit volontairement la frontière, puisqu'il représente une demande de validation pertinente pour le domaine.

13.10 Règles de conception

Utilisez un espace de noms visuel pour les événements dont le sens principal est la coordination locale de l'interface. Réservez les espaces de noms de domaine aux événements exprimant un fait métier ou de processus.

Acheminez d'abord les événements visuels via **PageController**. Laissez **PageModel** et les objets **objectBehind** des composants gérer les réactions locales. Ne faites remonter un événement

que si le modèle de page déclare un lien vers une action, un service, un processus, un message ou une modification du modèle conceptuel.

Pour les vues React ou web, évitez de transformer chaque événement local en requête serveur. Pour les vues natives, évitez d'employer le conteneur d'objets comme récepteur par défaut de chaque notification de composant.

Documentez explicitement les frontières d'événements dans les modèles de vues réutilisables. Le lecteur doit pouvoir distinguer les événements locaux à la page, ceux qui coordonnent l'application et ceux qui modifient le modèle conceptuel.

Chapitre 14

Modèles VIEW événementiels et asynchrones

Les contrôleurs VIEW modernes de logiCells sont généralement asynchrones par leur *protocole*, et non par la syntaxe du langage. Le modèle courant consiste en une orchestration événementielle utilisant événements visuels, événements du cycle de vie des DataSet, continuations de publication/soumission, états d'attente et notifications explicites d'actualisation. Les contrôleurs doivent rester légers et centrés sur le client ; le comportement métier appartient aux actions et processus MODEL réutilisables.

14.1 Penser en continuations, pas en async/await

La chaîne causale à privilégier est :

```
1 user action
2   -> dispatch visual event
3   -> dataset / post / submit request
4   -> lifecycle event
5   -> continuation action
6   -> commit or domain action completion
7   -> completion notification
8   -> local refresh
```

Exemple 14.1 – Flux typique d'une vue asynchrone

Ne supposez pas qu'un appel à `Refresh()` rende les données de résultat disponibles de façon synchrone aux lignes suivantes. Si la suite de la logique dépend des données chargées, poursuivez à partir de l'événement de cycle de vie pertinent, par exemple `AfterDataSetOpened`.

14.2 Abonnement aux événements du contrôleur

```
1 classes:
2   - class:
3     name: Page
4     classId: view#erp.orders.OrderList.Page
5     type: Entity
6     events:
7       - logiCells.Frame.OnShow:
8         actions:
9           - action:
10             path: './Action[@name=''OnShowForm'']'
11       - logiCells.Visual.Data.AfterDataSetOpened:
12         actions:
13           - action:
14             path: './Action[@name=''OnAfterOrdersOpened'']'
15     - erp.Visual.View.Refresh:
```

```

16         actions:
17             - action:
18                 path: './Action[@name=''OnRefreshView'']'

```

Exemple 14.2 – Le contrôleur s’abonne aux événements visuels et de DataSet

14.3 Émettre une intention, pas l’implémentation du parent

Une vue enfant ne doit pas accéder directement à son parent pour actualiser un par un les widgets concrets. Elle doit publier une intention telle que `erp.Visual.View.Refresh`. Le parent, ou une racine d’événements, décide quels descendants doivent réagir.

```

1  actions:
2      - action:
3          name: NotifyCompleted
4          class: ObjectItemScriptableObjectElementAction
5          type: ObjectItem
6          scripts:
7              - script:
8                  clientScript: true
9                  serverScript: false
10                 language: functionalP
11                 classes: Desktop
12                 cdata: |
13                     StopNotification := false;
14                     Event := Self.ParentPageControler.CreateVisualEvent(
15                         'erp.Visual.View.Refresh', Self, Self);
16                     Self.ParentPageControler.DispatchVisualEvent(Event, StopNotification);
17                     Self.PageControler.CloseForm();

```

Exemple 14.3 – La fin du traitement d’un enfant informe son parent

Ainsi, les vues surgissantes, enfants et parentes peuvent évoluer indépendamment.

14.4 Cadres d’attente et signaux de fin

Quand une opération visible s’étend sur plusieurs phases événementielles, signalez que la requête est en attente et mettez fin à cette attente par un événement de complétion conforme au contrat du framework visuel.

```

1  cdata: |
2      StopNotification := false;
3      Event := Self.PageControler.CreateVisualEvent(
4          'logiCells.Visual.TabularView.PostData', Self, Self);
5      Event.WaitingFrame := true;
6      Self.PageControler.DispatchVisualEvent(Event, StopNotification);

```

Exemple 14.4 – Demande de publication avec cadre d’attente

La continuation appartient à l’événement du cycle de vie de publication, pas au gestionnaire du bouton initial.

```

1  IDLE
2      -> user command
3  WAITING
4      -> post / submit / refresh accepted
5  PROCESSING
6      -> lifecycle continuation
7  COMMITTING
8      -> completion notification
9  READY
10     -> targeted refresh
11  DISPLAYED

```

Exemple 14.5 – Progression d’état recommandée

14.5 Modèle de cycle de vie des DataSet

Utilisez `BeforeDataSetOpened` pour préparer les filtres et le contexte, et `AfterDataSetOpened` pour les traitements qui exigent un DataSet chargé.

```
1 BeforeDataSetOpened
2   -> clear filters
3   -> add context/date filters
4   -> allow/open dataset
5
6 AfterDataSetOpened
7   -> inspect result rows
8   -> update presentation state
9   -> dispatch follow-up intent if needed
```

Exemple 14.6 – Continuation pilotée par un DataSet

14.6 Racines d'événements et propagation

Les cadres peuvent constituer des frontières de routage d'événements. Une racine peut recevoir un événement de haut niveau puis avertir ses enfants. `StopNotification` fait partie du protocole et doit être préservé au lieu de remplacer la propagation par des appels directs aux composants.

Employez des noms d'événements comme intentions stables et ajoutez de petites valeurs contextuelles à l'événement lorsqu'une même famille couvre plusieurs situations. Évitez de créer un nom unique pour chaque bouton accessoire.

14.7 Garder les contrôleurs VIEW légers

Le code MODEL-behind peut coordonner l'état visuel, les DataSet, les curseurs, la navigation et l'émission d'événements. Il ne doit pas devenir le nouvel emplacement de centaines de lignes de calcul métier.

Préférez :

```
1 VIEW controller
2   -> gather UI context
3   -> invoke reusable MODEL action / process
4   -> receive result or completion event
5   -> refresh the relevant projection
```

Exemple 14.7 – Frontière d'un contrôleur VIEW léger

Les invariants métier, l'idempotence, l'orchestration entre objets et les transitions d'état persistantes appartiennent à MODEL, aux actions ou aux processus, afin que les appelants services, agents, batchs et VIEW réutilisent le même comportement.

14.8 Liste de vérification d'une VIEW asynchrone

Pour toute vue non triviale, vérifiez :

- L'action de l'utilisateur est-elle représentée par un événement ou une action réutilisable, plutôt que par une chaîne d'appels de widgets ?
- Le code dépendant du résultat d'un DataSet reprend-il à partir d'un événement de son cycle de vie ?
- Une opération longue et visible suit-elle un protocole d'attente et de complétion ?
- Un enfant communique-t-il avec son parent par un événement exprimant une intention ?
- L'actualisation peut-elle être limitée à la vue ou au sous-arbre concerné, plutôt qu'à l'application entière ?

- Les calculs métier sont-ils placés hors du contrôleur VIEW ?
- Les événements en double et les tentatives répétées sont-ils sûrs ou explicitement protégés ?

Chapitre 15

Cadres de bureau réactifs et routage des événements

Les applications logiCells avancées n'affichent pas uniquement des formulaires, listes et champs. Elles coordonnent plusieurs vues simultanément : zone de navigation, espace de travail, panneaux d'aperçu, tableaux de bord, inspecteurs, menus d'actions, zones de notification et étapes de workflow. Dans ce type d'application, l'interface doit réagir localement aux sélections, modifications de DataSet, actualisations d'aperçu et événements de workflow, sans reconstruire la page entière ni envoyer chaque geste à un fournisseur distant.

Le modèle `reactiveDesktopFrame` répond à ce besoin. Il transforme une région du bureau en hôte réactif. Le cadre peut recevoir des événements, charger ou actualiser des vues enfants, propager des notifications aux enfants, réagir aux changements de DataSet et coordonner les échanges parent-enfant via `PageController`. Ce chapitre se concentre sur les interfaces web et publiées. Les implémentations natives sont hors du périmètre de cet ouvrage.

15.1 Des formulaires pilotés par DataSet aux bureaux réactifs

Le chapitre précédent décrivait les vues pilotées par DataSet. Ce modèle est déjà puissant pour les formulaires et grilles : une page déclare DataSet, DataSource et curseurs, les composants se lient aux DataSource et les événements des DataSet valident ou actualisent la page. Cependant, les interfaces avancées nécessitent une coordination de niveau supérieur.

Un formulaire piloté par DataSet répond notamment aux questions suivantes :

```
1 Which dataset feeds this field?  
2 Which datasource feeds this table?  
3 Can this row be posted?  
4 Which detail dataset must refresh when the cursor changes?
```

Exemple 15.1 – Questions traitées par les formulaires pilotés par DataSet

Un bureau réactif pose des questions plus larges :

```
1 Which child view must be shown in this region?  
2 Which child views must be notified when the current object changes?  
3 Which frame owns the event?  
4 Should the event be handled locally, forwarded to a parent frame,  
5 or escalated as an application action?  
6 Which visual state must change after a dataset refresh?  
7 Can a preview be refreshed without rebuilding the page?
```

Exemple 15.2 – Questions traitées par un cadre de bureau réactif

C'est pourquoi un `reactiveDesktopFrame` ne doit pas être décrit comme un simple widget de disposition. Il constitue une frontière réactive locale à l'intérieur du bureau.

15.2 Définition d'un cadre de bureau réactif

Un `reactiveDesktopFrame` est un cadre visuel doté d'un comportement d'exécution. Il héberge du contenu dynamique, mais participe également aux événements. Il occupe une place dans l'arborescence des vues, possède un composant visuel d'exécution, éventuellement un `ObjectBehind`, et entretient une relation avec le contrôleur de page.

```
1 reactiveDesktopFrame
2   -> visual frame region
3   -> runtime visual component
4   -> optional objectBehind / frame controller
5   -> optional input cursor
6   -> child view container
7   -> local event handlers
8   -> child notification router
9   -> parent event bridge
```

Exemple 15.3 – Structure conceptuelle d'un cadre de bureau réactif

Le cadre peut donc remplir plusieurs rôles.

Rôle	Signification
Hôte	Contient une vue enfant, un ensemble de vues enfants ou une arborescence dynamique de vues.
Frontière de contexte	Reçoit un curseur ou un objet courant et partage ce contexte avec ses enfants.
Récepteur d'événements	Déclare des événements locaux ou applicatifs et les associe à des actions.
Routeur d'événements	Transmet un événement visuel aux enfants, à un contrôleur parent ou à une action locale.
Coordinateur sensible aux DataSet	Réagit aux événements du cycle de vie des DataSet et adapte l'état visuel.
Surface d'aperçu	Actualise le rendu visuel lorsque les métadonnées, le texte ou des fragments du modèle changent.
Enveloppe de workflow	Héberge une séquence d'étapes et passe de l'une à l'autre par événements.

Un conteneur de bureau classique est avant tout spatial. Un cadre de bureau réactif est à la fois spatial et comportemental.

15.3 Participants à l'exécution

Ce modèle implique plusieurs participants à l'exécution. Les distinguer évite de nombreuses erreurs de conception.

Participant	Responsabilité
<code>reactiveDesktopFrame</code>	Déclare et héberge une région réactive dans l'arborescence de vues.
Composant visuel d'exécution	Instance concrète de composant pouvant être trouvée, invalidée, actualisée, déplacée ou notifiée.
<code>ObjectBehind</code> de cadre	Contrôleur conceptuel associé au cadre ; contient l'état et les actions locales.
<code>PageControler</code>	Bus local et résolveur pour les composants visuels, événements, DataSet et actions de page.

ParentPageController	Bus parent utilisé lorsqu'une page enfant ou une vue intégrée doit notifier son hôte.
DataSetsManager	Registre des DataSet, DataSource et curseurs de la page ; source des événements de DataSet.
Vues enfants	Vues intégrées hébergées dans le cadre ; elles peuvent recevoir ses notifications.
Contrôleurs de composants	Contrôleurs web ou de page associant les gestes utilisateurs aux événements fonctionnels.

Un modèle d'implémentation courant consiste à récupérer l'ObjectBehind du cadre par nom, puis à résoudre le composant d'exécution par identifiant.

```
1 FrameOB := Self.PageController.GetVisualComponentObjectBehindByName('MainFrame');
2 FrameComponent := Self.PageController.FindVisualComponentById(FrameOB.Id);
```

Exemple 15.4 – Recherche d'un composant de cadre réactif

Le premier appel travaille au niveau de l'ObjectBehind. Le second passe de cet objet au composant visuel d'exécution. La distinction est importante : l'ObjectBehind porte l'état et les actions conceptuels ; le composant visuel héberge les enfants et exécute les opérations visuelles.

15.4 Modèle d'hébergement : région de vue dynamique

Le premier modèle concerne l'hébergement. Un cadre réactif définit une région dont le contenu peut être chargé, remplacé ou actualisé.

```
1 Desktop
2   Header
3   Navigation
4   MainReactiveFrame
5     current child view
6   Footer
```

Exemple 15.5 – Cadre réactif comme région dynamique

Le cadre peut afficher une vue d'objet, une vue de liste, un aperçu, un tableau de bord, une étape de workflow ou une autre disposition composée. Le reste du bureau demeure stable tandis que le cadre change.

Ce modèle diffère du développement web fondé sur les routes. La route complète du navigateur n'a pas besoin de changer. Un événement applicatif local peut demander à un cadre d'afficher une nouvelle vue dans une région nommée.

```
1 Application.Shell.ShowView
2   parameters:
3     TargetFrame = MainReactiveFrame
4     ViewName = CustomerSummary
5     Context = current customer
```

Exemple 15.6 – Événement applicatif routé vers un cadre

PageController reçoit ou crée l'événement, résout le cadre cible et lui demande de charger ou d'activer la vue enfant demandée. Le composant émetteur n'a pas besoin de savoir comment le cadre est affiché.

15.5 Modèle de contexte : curseur d'entrée et objet courant

Un cadre réactif reçoit souvent un curseur d'entrée. Celui-ci définit l'objet ou la ligne courante que le cadre et ses enfants doivent utiliser.

Le cadre n'est pas propriétaire de l'objet métier : il reçoit un contexte. La même implémentation peut ainsi héberger différentes vues selon le curseur courant. Une liste maîtresse peut changer l'objet courant ; le cadre réactif actualise ensuite les zones de détail, d'actions et d'inspection.

```

1 Master list selection
2   -> MainSourceCursor changes
3   -> DataSetManager emits cursor or dataset event
4   -> reactiveDesktopFrame receives or observes the event
5   -> child views are refreshed or notified

```

Exemple 15.7 – Région réactive pilotée par curseur

Le modèle de contexte est particulièrement important pour les tableaux de bord avancés. Un même cadre peut héberger plusieurs panneaux enfants dépendant tous du même objet courant : résumé, historique, liste d'actions, graphe et aperçu.

15.6 Modèle de récepteur d'événements applicatifs

Un cadre réactif peut déclarer des événements dans son modèle de page local. Il devient alors un récepteur d'événements applicatifs.

La classe d'événement doit être stable et fonctionnelle. Elle doit décrire ce que veut l'application, et non comment un widget est implémenté. Ainsi, `Application.Shell.ShowView` est préférable à un événement de bas niveau tel que `Button42Clicked`. L'action associée détermine la réaction du cadre.

```

1 procedure ShowView(Source, Sender, ViewName);
2 var
3   FrameOB: Object;
4   FrameComponent: Object;
5 begin
6   FrameOB := Self.PageController.GetVisualComponentObjectBehindByName('MainFrame');
7   FrameComponent := Self.PageController.FindVisualComponentById(FrameOB.Id);
8
9   FrameComponent.MoveTo(ViewName);
10 end;

```

Exemple 15.8 – Traitement d'un événement applicatif dans le cadre

Cette action est locale à la page. Elle peut charger une vue, sélectionner un onglet, remplacer un cadre enfant ou demander au contrôleur de composant de reconstruire une région. Elle n'a pas à franchir la frontière serveur.

15.7 Modèle de routeur des notifications vers les enfants

Le deuxième grand modèle est la notification descendante. Un cadre réactif reçoit un événement et envoie une notification visuelle à ses enfants.

```

1 Parent action
2   -> find reactive frame component
3   -> create visual event
4   -> NotifyChildren(event, stopNotification)
5   -> each child decides whether it handles the event

```

Exemple 15.9 – Notification descendante via un cadre réactif

Ce modèle convient lorsque plusieurs vues enfants doivent être actualisées après la modification d'une condition partagée. Le cadre parent n'a pas besoin de connaître chaque type de composant enfant. Il émet une notification visuelle sémantique et laisse les enfants concernés la traiter.

```

1 procedure RefreshChildViews;
2 var
3   FrameOB: Object;

```

```

4   FrameComponent: Object;
5   Event: Object;
6   StopNotification: Boolean;
7 begin
8   FrameOB := Self.PageController.GetVisualComponentObjectBehindByName('TableContainer');
9   FrameComponent := Self.PageController.FindVisualComponentById(FrameOB.Id);
10
11  StopNotification := false;
12  Event := Self.PageController.CreateVisualEvent(
13    'lgc.Visual.View.Refresh',
14    nil,
15    nil,
16    Self.ObjectsContainer.MetadataContainer,
17    false
18  );
19
20  FrameComponent.NotifyChildren(Event, StopNotification);
21 end;

```

Exemple 15.10 – Notification des enfants depuis un cadre réactif

Le paramètre `StopNotification` permet à un enfant de consommer l'événement. C'est utile lorsque le cadre contient des cadres réactifs imbriqués. Un cadre enfant peut traiter la notification et arrêter la propagation pour éviter les actualisations en double.

15.8 Modèle de remontée des événements enfant-vers-parent

Le modèle complémentaire est la notification ascendante. Une vue enfant émet un événement sur le contrôleur de page parent. C'est utile lorsqu'une vue intégrée demande au bureau parent d'ouvrir, de sélectionner, de parcourir ou d'actualiser un élément.

```

1 Child item double-click
2 -> child PageController creates visual event on ParentPageController
3 -> event carries child PageController as context
4 -> parent dispatches event
5 -> parent reactive frame decides what to do

```

Exemple 15.11 – Remontée des événements enfant-vers-parent

Une implémentation courante crée un événement visuel sur le contrôleur de page parent et y joint le contrôleur de page enfant.

```

1 procedure OnDbClick(Source, Sender);
2 var
3   MasterOB: Object;
4   MasterComponent: Object;
5   Event: Object;
6 begin
7   MasterOB := Self.PageController.GetVisualComponentObjectBehindByName('Master');
8   MasterComponent := Self.PageController.FindVisualComponentById(MasterOB.Id);
9
10  Event := Self.PageController.ParentPageController.CreateVisualEvent(
11    'lgc.Visual.ViewItem.Notify.DblClick',
12    MasterComponent.ParentFrame,
13    MasterComponent.ParentFrame,
14    Self.ObjectsContainer.MetadataContainer,
15    false
16  );
17
18  Event.PageController := Self.PageController;
19  Self.PageController.ParentPageController.DispatchVisualEvent(Event);
20 end;

```

Exemple 15.12 – Émission d'un événement visuel vers le PageController parent

L'enfant ne décide pas comment l'enveloppe parente doit réagir. Il signale seulement une interaction visuelle significative : un élément a été double-cliqué. Le parent peut alors ouvrir

l'objet, changer de vue enfant active, charger un aperçu ou mettre à jour l'état du workflow.

15.9 Modèle de réaction aux DataSet

Un cadre réactif est particulièrement utile lorsque l'état visuel dépend de l'état d'un DataSet. Un événement tel que `AfterOpenDataSet`, `AfterRefresh` ou `CurrentChanged` peut mettre à jour le cadre et ses enfants.

```
1 DataSet opened or refreshed
2 -> OnAfterOpenDataSet receives Sender
3 -> handler checks Sender.Name
4 -> reads fields from the dataset
5 -> finds visual components
6 -> changes visibility, caption, state or enabled flags
7 -> optionally notifies children
```

Exemple 15.13 – Événement DataSet pilotant l'état du cadre

Ce modèle est souvent utilisé pour les tableaux de bord et les interfaces pilotées par états. Un DataSet peut contenir un état calculé : compteurs, indicateurs, statut de validation, phase courante, entrées manquantes ou résultats d'autorisation. Le cadre lit cet état et adapte l'interface.

```
1 procedure OnAfterOpenDataSet(Source, Sender);
2 var
3   StateDataSet: Object;
4   GreenState: Object;
5   RedState: Object;
6 begin
7   if Sender.Name = 'CurrentStateDS' then
8     begin
9       GreenState := Self.PageController.FindVisualComponentByName('StateGreen');
10      RedState := Self.PageController.FindVisualComponentByName('StateRed');
11
12      StateDataSet := Self.PageController.DataSetsManager.DataSetByName('CurrentStateDS');
13      StateDataSet.First();
14
15      if not StateDataSet.GetEof() then
16        begin
17          GreenState.Visible := StateDataSet.FieldByName('IsValid').AsBoolean;
18          RedState.Visible := not StateDataSet.FieldByName('IsValid').AsBoolean;
19        end;
20      end;
21 end;
```

Exemple 15.14 – Gestionnaire DataSet mettant à jour l'état visuel

Ce gestionnaire n'est pas une règle métier, mais une règle d'état de vue. Le DataSet expose déjà l'état calculé. Le cadre associe cet état à un comportement d'interface.

15.10 Modèle d'actualisation et de recalcul

Un modèle avancé courant combine notification des enfants et actualisation des DataSet.

```
1 User changes a child item
2 -> parent receives a visual event
3 -> parent notifies children with Refresh
4 -> parent refreshes a computed dataset
5 -> parent calls OnAfterOpenDataSet
6 -> status panels and action buttons update
```

Exemple 15.15 – Chaîne d'actualisation et de recalcul

Cela évite un rechargement complet de la page. Seules les vues enfants et les DataSet concernés sont actualisés. Le cadre reste le coordinateur local.

```

1 procedure RefreshWorkspace;
2 var
3   FrameOB: Object;
4   FrameComponent: Object;
5   RefreshEvent: Object;
6   StopNotification: Boolean;
7   StateDataSet: Object;
8 begin
9   FrameOB := Self.PageController.GetVisualComponentObjectBehindByName('MainReactiveFrame');
10  FrameComponent := Self.PageController.FindVisualComponentById(FrameOB.Id);
11
12  StopNotification := false;
13  RefreshEvent := Self.PageController.CreateVisualEvent(
14    'lgc.Visual.View.Refresh', nil, nil,
15    Self.ObjectsContainer.MetaDataContainer, false);
16
17  FrameComponent.NotifyChildren(RefreshEvent, StopNotification);
18
19  StateDataSet := Self.PageController.DataSetsManager.FindDataSetByName('CurrentStateDS');
20  if Assigned(StateDataSet) then
21  begin
22    StateDataSet.Refresh();
23    Self.OnAfterOpenDataSet(nil, StateDataSet);
24  end;
25 end;

```

Exemple 15.16 – Action d’actualisation et de recalcul

C’est l’un des modèles les plus importants pour les interfaces évoluées. Il donne à l’utilisateur l’impression d’une application en temps réel tout en préservant une structure conceptuelle claire.

15.11 Modèle de surface d’aperçu

Un cadre réactif peut héberger un aperçu en direct. Dans un concepteur de métadonnées, de rapports, un éditeur de documents ou un concepteur de workflows, la modification d’un composant source peut actualiser un cadre d’aperçu.

```

1 Editor content changed
2 -> lgc.Visual.TextEditor.Refresh
3 -> Preview reactive frame receives the event
4 -> Preview.Refresh()
5 -> child preview rerenders

```

Exemple 15.17 – Chaîne d’événements d’aperçu en direct

Ce modèle doit rester local lorsque l’aperçu peut être reconstruit à partir de l’état de page. Il ne doit appeler un service que si le rendu nécessite un traitement distant, des autorisations ou un calcul important.

```

1 procedure RefreshPreview(Source, Sender);
2 var
3   PreviewOB: Object;
4   PreviewComponent: Object;
5 begin
6   PreviewOB := Self.PageController.GetVisualComponentObjectBehindByName('Preview');
7   PreviewComponent := Self.PageController.FindVisualComponentById(PreviewOB.Id);
8   PreviewComponent.Refresh();
9 end;

```

Exemple 15.18 – Gestionnaire d’actualisation de l’aperçu

Pour la publication web, le même modèle correspond à un composant React s’abonnant à un événement de page, puis mettant à jour son état local ou demandant au modèle de page un nouveau balisage généré.

15.12 Modèle d'enveloppe de workflow

Un cadre réactif peut aussi héberger un workflow. Le cadre contient l'étape courante tandis que le contrôleur passe d'une étape à l'autre.

```
1 Workflow event
2   -> StepValidated
3   -> reactive frame moves to next step
4   -> child step view opens
5   -> step dataset opens
6   -> OnAfterOpenDataSet updates footer actions
```

Exemple 15.19 – Enveloppe de workflow avec cadre réactif

Le cadre peut exposer des opérations comme `MoveTo`, `LoadMainFrame`, `Invalidate`, `Refresh` ou `UpdateConnectionStatus`. Les noms varient selon les implémentations, mais le modèle fonctionnel est stable : le cadre héberge l'étape courante et coordonne la navigation au moyen d'événements locaux.

Ce modèle est utile pour les parcours d'intégration, les configurations guidées, les assistants de création de documents, les processus de validation et les applications séquentielles de type mobile.

15.13 Taxonomie des événements des cadres réactifs

Les événements autour des cadres réactifs peuvent être classés en six catégories.

Catégorie	Signification	Gestionnaire habituel
Interaction visuelle	L'utilisateur clique, double-clique, sélectionne, déploie ou replie un élément visuel.	Contrôleur de composant ou <code>ObjectBehind</code> du cadre.
Commande de cadre	Un parent ou un enfant demande à un cadre d'afficher, déplacer, charger, actualiser ou invalider une région.	Action du cadre réactif.
Événement de DataSet	Un <code>DataSet</code> s'ouvre, s'actualise, publie, annule ou change de ligne courante.	Modèle de page ou <code>ObjectBehind</code> du cadre.
Notification d'enfant	Un cadre envoie une notification sémantique à ses enfants.	Contrôleurs des vues enfants.
Notification au parent	Un enfant émet un événement vers le <code>PageController</code> parent.	Cadre parent ou contrôleur de l'enveloppe.
Intention applicative	Un événement applicatif demande un changement de haut niveau, tel que l'affichage d'une vue ou l'ouverture d'un objet.	Enveloppe de page ou cadre applicatif.

La catégorie détermine le niveau de remontée approprié. Un clic doit souvent rester visuel. L'actualisation d'un `DataSet` doit souvent rester locale. Une intention applicative peut franchir les frontières de page. Une transition métier doit être exprimée comme une action applicative ou de processus, non comme un événement brut de cadre.

15.14 Nommage recommandé des événements

Les noms des événements du bureau réactif doivent rendre leur niveau compréhensible.

Exemple	Signification	Niveau
<code>lgc.Visual.View.Refresh</code>	Actualiser une vue visuelle ou ses enfants.	Visuel
<code>lgc.Visual.ViewItem.Notify.DblClick</code>	Un élément de vue a été double-cliqué et notifie son parent.	Visuel
<code>lgc.Visual.Data.AfterOpenDataSet</code>	Un DataSet s'est ouvert et l'état visuel peut être adapté.	Données/vue
<code>Application.Shell.ShowView</code>	L'enveloppe applicative doit afficher une vue nommée.	Applicatif
<code>Application.Workflow.MoveNext</code>	L'enveloppe de workflow doit passer à l'étape suivante.	Applicatif
<code>Business.Document.Validated</code>	Un événement du domaine s'est produit.	Métier

Évitez d'intégrer les identifiants de composants dans les noms de classes d'événements. Le cadre cible, le composant source ou le nom du DataSet doivent être transmis comme paramètres si nécessaire.

15.15 Assemblage déclaratif

L'exemple simplifié suivant montre un cadre réactif hébergeant une région de détail dynamique. Il possède un curseur d'entrée, reçoit un événement applicatif, réagit aux événements de DataSet et peut notifier ses enfants.

Remarquez que les DataSet et DataSource n'utilisent pas `kind`. La classification `kind="slot|attribute"` concerne les déclarations de `field` du modèle dans les facettes conceptuelles. Les surfaces de données de vue telles que DataSet, DataSource, DataCursor et `reactiveDesktopFrame` ne l'emploient pas.

15.16 Correspondance avec le web réactif

Dans l'environnement d'exécution web, le même modèle peut s'exprimer à l'aide d'un enregistrement de composant et d'abonnements locaux.

```

1 componentManager.register(
2   'ReactiveDesktopFrame',
3   'default',
4   {
5     component: ReactiveDesktopFrame,
6     controller: ReactiveDesktopFrameController,
7     dataStructure: ReactiveDesktopFrameStruct
8   }
9 );
```

Exemple 15.20 – Modèle d'enregistrement d'un cadre réactif web

Le composant React affiche le cadre et ses enfants. Le contrôleur s'abonne aux événements de page, résout les DataSet via le contrôleur de page et délègue les modifications de rendu à l'état React.

```

1 class ReactiveDesktopFrameController {
2   constructor(pageController, componentState) {
3     this.pc = pageController;
4     this.state = componentState;
5   }
6 }
```

```

7  componentDidMount() {
8      this.pc.subscribe('Application.Shell.ShowView', this, 'showView');
9      this.pc.subscribe('lgc.Visual.View.Refresh', this, 'refresh');
10     this.pc.subscribe('lgc.Visual.Data.AfterOpenDataSet', this, 'afterOpenDataSet');
11 }
12
13 showView(event) {
14     this.state.setCurrentView(event.params.viewName);
15 }
16
17 refresh(event) {
18     this.state.refreshChildren(event);
19 }
20
21 afterOpenDataSet(event) {
22     if (event.sender.name === 'CurrentStateDS') {
23         const ds = this.pc.dataSetManager.DataSourceByName('CurrentStateSource')
24             .getDataSet();
25         this.state.updateStatusFromDataSet(ds);
26     }
27 }
28 }

```

Exemple 15.21 – Responsabilités du contrôleur de cadre réactif

Le contrôleur ne contourne pas **DataSetManager**. Il utilise les surfaces de données de page déjà déclarées par les métadonnées. Le composant web reste ainsi aligné avec l’environnement conceptuel d’exécution.

15.17 État local et état du modèle

Un cadre de bureau réactif possède un état visuel local. Il peut, par exemple, mémoriser la vue enfant active, un panneau déployé, l’onglet courant, un indicateur de chargement ou un compteur d’actualisations d’aperçu. Cet état ne doit pas être confondu avec l’état du modèle.

Type d’état	Exemples
État visuel local	Vue enfant active, panneau replié, onglet sélectionné, chargement, position de défilement.
État des données de vue	Ligne courante du DataSet, contenu de la DataSource, position du curseur, valeurs du DataSet de filtres.
État du modèle	Champs d’objets métier, relations, état du workflow, faits de processus persistés.
État applicatif	Espace de travail courant, module courant, état d’authentification, état des commandes de l’enveloppe.

Un événement de cadre peut mettre à jour immédiatement l’état visuel local. Il ne doit modifier l’état du modèle qu’au travers d’une action déclarée, d’une publication de DataSet ou d’une frontière de service. Cette règle est essentielle à la traçabilité.

15.18 Éviter les boucles d’événements

Les interfaces réactives sont sujettes aux boucles accidentelles. L’actualisation d’un DataSet peut déclencher l’actualisation d’un cadre, laquelle notifie les enfants, dont l’un actualise de nouveau le même DataSet. Le modèle doit donc définir des garde-fous.

Parmi les protections pratiques contre les boucles figurent :

- 1 - Check Sender.Name before reacting to a dataset event.
- 2 - Do not refresh a dataset if the event already comes from that dataset.

```

3 - Use StopNotification when a child frame consumes an event.
4 - Separate RefreshRequested from RefreshCompleted.
5 - Debounce high-frequency events such as text changes.
6 - Avoid posting from an AfterOpen handler.
7 - Keep application events idempotent when possible.

```

Exemple 15.22 – Règles de prévention des boucles

Un bon bureau réactif ne se contente pas d'avoir beaucoup d'événements : il impose une discipline de gestion des événements.

15.19 Gestion des erreurs et diagnostics

Un cadre réactif doit signaler localement et visiblement l'absence de ses dépendances. Les DataSet manquants, les vues enfants non résolues ou les classes d'événements inconnues ne doivent pas entraîner un silence inexpliqué.

```

1 if not Assigned(FrameOB) then
2   Log('Reactive frame object behind not found: MainReactiveFrame');
3
4 if not Assigned(FrameComponent) then
5   Log('Reactive frame component not found for object behind id');
6
7 if not Assigned(StateDataSet) then
8   Log('Dataset not found: CurrentStateDS');

```

Exemple 15.23 – Diagnostics des gestionnaires de cadres réactifs

Pour les composants web, les diagnostics doivent inclure le nom du composant, son nom fonctionnel, sa classe d'implémentation, le nom de la DataSource, le nom du DataSet et la classe d'événement. Cela facilite considérablement le débogage des interfaces générées.

15.20 Règles de conception

Les règles suivantes résument la discipline recommandée.

1. Utilisez `reactiveDesktopFrame` pour les régions hébergeant des vues enfants dynamiques ou pilotées par événements.
2. Conservez les gestes utilisateurs bruts au niveau local, sauf s'ils expriment une intention fonctionnelle ou applicative.
3. Acheminez les communications enfant-vers-parent via le `PageControler` parent, et non par des références directes aux composants.
4. Utilisez `NotifyChildren` pour les notifications sémantiques descendantes.
5. Utilisez les événements de DataSet pour adapter l'état de la vue, et non pour coder directement des règles métier.
6. Faites remonter explicitement vers les actions applicatives ou métier les changements ayant une portée de domaine.
7. Transmettez les noms de cadre cible, de DataSet et de vue comme paramètres au lieu de les intégrer dans les noms de classes d'événements.
8. Protégez les chaînes d'actualisation contre les boucles et les notifications en double.
9. Réservez `kind="slot|attribute"` aux champs du modèle conceptuel déclarés dans les facettes.
10. Traitez les implémentations natives comme un sujet distinct ; le modèle web public repose sur les métadonnées, `PageControler`, les surfaces de données, le routage des événements et l'état réactif des composants.

15.21 Pourquoi ce modèle est important

Sans cadre de bureau réactif, les interfaces complexes tendent à être fortement couplées. Un bouton connaît l'onglet à changer. Un tableau enfant sait quel panneau parent actualiser. Un widget de tableau de bord sait quel service appeler. Un composant d'aperçu en sait trop sur l'éditeur qui l'alimente.

Avec un cadre de bureau réactif, l'interface devient déclarative et coordonnée :

```
1 components emit semantic events
2   -> PageControler routes them
3   -> reactiveDesktopFrame receives or forwards them
4   -> DataSetManager supplies data state
5   -> child views update locally
6   -> application actions are called only when needed
```

Exemple 15.24 – Coordination d'un bureau réactif

C'est le fondement des interfaces logiCells avancées. Le cadre n'est pas simplement un endroit où afficher des composants. Il constitue une frontière locale d'orchestration où se rencontrent composition visuelle, état des DataSet et intention applicative.

Quatrième partie

Clients riches natifs et web

Chapitre 16

Hôtes natifs de clients riches

L'architecture View de logiCells n'est pas propre au web. Le même modèle client – projection View, cache DataSet/DataCursor, comportement de page, routage d'événements et frontières d'actions – peut être hébergé par un client natif lorsque la distribution de l'environnement d'exécution choisie fournit le registre d'implémentations visuelles correspondant.

16.1 Un contrat stable

Un client natif doit conserver la même séparation que celle présentée dans tout ce guide :

```
1 conceptual model / actions
2   |
3   v
4 View intent + MODEL-behind
5   |
6   v
7 DataSet / DataCursor client state
8   |
9   v
10 functional component intent (UXfc)
11  |
12  v
13 registered native visual implementation
```

Exemple 16.1 – Séparation des responsabilités d'un client riche natif

Le composant fonctionnel décrit la finalité de l'interaction ; l'implémentation native détermine comment cette intention est présentée par la bibliothèque d'interface choisie. Les actions métier restent dans le modèle conceptuel au lieu d'être déplacées dans les gestionnaires d'événements des widgets.

16.2 Cache et asynchronisme

Les clients natifs suivent les mêmes règles d'état client que les clients web riches. Un DataSet peut contenir une projection locale, un DataCursor suit la position et l'objet courant, et les événements visuels peuvent rester locaux jusqu'à ce qu'une action applicative ou métier doive être invoquée. Les opérations longues ou distantes doivent rendre la main à l'interface et se poursuivre selon le cycle de vie des événements et des données décrit dans les chapitres sur l'asynchronisme.

C'est pourquoi on peut choisir l'architecture View pour un client natif : non parce qu'une interface native impose MVC, mais parce que l'application bénéficie d'un cache local explicite, d'un état d'interaction stable et d'une coordination asynchrone avec l'environnement d'exécution.

16.3 Frontière entre plateformes et bibliothèques d'interface

La disponibilité de l'environnement d'exécution sous Windows, Linux et macOS ne signifie pas que toutes les bibliothèques de composants natifs ou tous les registres visuels offrent exactement les mêmes fonctionnalités sur chaque hôte. Traitez la bibliothèque d'interface native comme un adaptateur propre au déploiement et vérifiez l'ensemble d'implémentations livré pour la plateforme cible.

Le contrat portable de ce guide est donc l'architecture View/DataSet/événements. Une classe précise de widget natif ne fait pas partie du modèle de programmation conceptuelle et n'est volontairement pas utilisée ici comme vocabulaire public.

16.4 Quand préférer un client MVP personnalisé

Une application native qui n'a besoin que de quelques formulaires ou d'appels d'actions directs peut être plus simple à réaliser comme client MVP/personnalisé utilisant l'API ou l'ABI d'exécution. Choisissez l'architecture View de logiCells lorsque ses mécanismes de cache, d'intention des composants, de routage d'événements et de synchronisation répondent à un besoin client réel.

Chapitre 17

Créer des composants fonctionnels web réactifs

Cet ouvrage considère les composants fonctionnels web comme des incarnations à part entière des composants UXfc de logiCells. Les composants visuels natifs relèvent d'un autre guide d'implémentation. Le présent chapitre a donc un objectif précis et limité : expliquer comment implémenter un composant fonctionnel dans un environnement d'exécution web en conservant la séparation logiCells entre intention, cache de données, événements locaux et frontières distantes.

Un composant fonctionnel web n'est pas simplement un widget React. C'est l'incarnation web d'une fonction d'interaction déclarée. Il reçoit des données par une surface au niveau page, réagit localement via un contrôleur de page et ne franchit la frontière du serveur ou du fournisseur que si le modèle de page le lui demande explicitement.

17.1 De la fonction UXfc au composant web

Un composant UXfc logiCells commence comme une fonction d'interaction : consultation, sélection, exploration, édition, validation, navigation, affichage tabulaire, affichage d'état ou autre fonction réutilisable. En mode web, cette fonction est réalisée par un composant réactif, généralement un composant JavaScript ou TypeScript enregistré.

```
1 UXfc functional component
2   -> registered web component
3   -> reactive props and local state
4   -> PageController dispatch
5   -> DataSource / DataSet / DataCursor binding
6   -> optional provider or service boundary
```

Exemple 17.1 – De la déclaration fonctionnelle au rendu web

Le point essentiel est que le composant web réalise un contrat fonctionnel. Il ne redéfinit pas le modèle, n'est pas propriétaire de la collection métier et ne décide pas seul à quel moment une action visuelle devient une action du domaine.

17.2 Contrat d'un composant réactif

Un composant fonctionnel web doit posséder un contrat explicite, plus stable que son implémentation visuelle.

Élément du contrat	Signification
Nom de fonction	Fonction UXfc logique réalisée, telle que <code>tabularView</code> , <code>autoCompleteSelect</code> , <code>consultation</code> ou <code>statusView</code> .

Entrées	Surfaces de données, noms de curseurs, ressources, options, identifiants de contexte et descripteurs d'actions reçus de la page.
État local	État strictement client : onglet sélectionné, panneau ouvert, filtre temporaire, texte en attente, état de survol ou indicateur local de chargement.
Événements	Événements visuels ou UXfc émis vers PageController .
Règles de remontée	Conditions selon lesquelles la page associe un événement à une actualisation de fournisseur, un appel de service, une action, un processus ou une modification du modèle conceptuel.

Ce contrat permet au même composant fonctionnel d'évoluer visuellement sans modifier la déclaration de vue conceptuelle.

17.3 Props réactives et surfaces de données de la vue

En mode web, la réactivité doit normalement être pilotée par les surfaces de données de niveau page. Le composant reçoit un fragment de modèle de vue ou un point de liaison dérivé d'une *DataSource*, d'un *DataSet* ou d'un *DataCursor*. Il ne doit pas solliciter directement un fournisseur distant à chaque petite interaction.

```

1 Remote or local provider
2   -> collection adapter
3   -> DataSet cache
4   -> DataSource binding
5   -> reactive component props
6   -> component render

```

Exemple 17.2 – Flux de données réactif d'un composant web

Quand le *DataSet* change, l'état de la page change. L'environnement réactif affiche alors à nouveau le composant avec de nouvelles props. C'est l'équivalent web d'une notification du cache de vue local.

17.4 L'état local n'est pas l'état du modèle

Un composant réactif conserve souvent un état local. C'est normal, mais il faut le classer correctement.

```

1 selectedTab
2 expandedPanel
3 currentClientFilter
4 pendingAutocompleteText
5 isDropDownOpen
6 isLocalLoading
7 lastNavigationError

```

Exemple 17.3 – État web local courant

Ces valeurs appartiennent au cycle de vie local du composant web ou de la page. Elles ne sont pas automatiquement des attributs du modèle, des modifications de collection, des mises à jour serveur ou des faits métier. Une mise à jour locale peut déclencher un événement visuel, mais celui-ci reste local tant que *PageModel* ne l'associe pas à une opération de niveau supérieur.

17.5 Émettre des événements visuels locaux

Un composant web doit communiquer avec la page par son contrôleur. Cela le rend réutilisable et évite un couplage fort avec les services, fournisseurs ou actions métier.

```
1 function onTabSelected(tabId) {  
2   pageController.dispatchEvent({  
3     eventClass: 'lgcVisual.TabbedView.TabSelected',  
4     source: 'CustomerWorkspaceTabs',  
5     sender: 'TabHeader',  
6     local: true,  
7     params: { tabId: tabId }  
8   });  
9 }
```

Exemple 17.4 – Émission locale par un composant web

Le contrôleur de page décide si l'événement est consommé localement, transmis au modèle de page, envoyé à un autre composant fonctionnel ou remonté vers une action ou un fournisseur.

17.6 Quand un événement franchit la frontière

La frontière serveur doit être explicite. Une sélection, un changement d'onglet ou de filtre local ne doit pas devenir automatiquement un appel distant. À l'inverse, une commande de recherche, d'enregistrement, de validation ou d'actualisation peut volontairement franchir cette frontière.

Événement	Portée par défaut	Remontée possible
TabSelected	Page locale	Aucune, ou actualisation du DataSet local.
CurrentRowChanged	Cache de page local	Actualisation facultative du DataSet de détail.
FilterTextChanged	Composant local	Recherche distante facultative avec anti-rebond.
SearchRequested	Action de page	Requête du fournisseur ou appel de service.
SaveRequested	Action applicative	Action publiée, mise à jour du fournisseur ou transition de processus.
ValidationRequested	Action de domaine	Processus métier ou événement conceptuel.

La règle de conception est simple : la réactivité locale est le comportement par défaut ; toute remontée doit être déclarée.

17.7 Modèle d'abonnement et d'actualisation

Un composant web réactif doit normalement s'abonner aux données de la page, pas directement au fournisseur. Le modèle de page ou le cache de données décide quand le fournisseur doit être actualisé.

```
1 component subscribes to:  
2   - DataSource records  
3   - DataCursor current object  
4   - local PageModel flags  
5   - local event stream  
6
```

```

7 component does not directly own:
8   - provider connection
9   - remote pagination protocol
10  - persistence transaction
11  - business validation process

```

Exemple 17.5 – Modèle d’abonnement recommandé

Les protocoles de fournisseurs restent ainsi remplaçables. Un fournisseur de simulation local, un fournisseur REST paginé, un fournisseur calculé et un fournisseur distant synchronisé peuvent tous alimenter le même composant réactif via la même surface de données de la page.

17.8 Anti-rebond, regroupement et idempotence

La réactivité web peut produire une multitude de petits événements. Saisie de texte, redimensionnement, défilement, survol, frappe dans un filtre et autocomplétion peuvent générer des notifications fréquentes. Les composants logiCells ne doivent pas les considérer toutes comme des événements métier.

```

1 text changed locally
2   -> update local component state
3   -> dispatch lgcVisual.Lookup.TextChanged local
4   -> PageModel debounces
5   -> LookupRequested event if needed
6   -> provider query
7   -> lookup DataSet refresh
8   -> component re-render

```

Exemple 17.6 – Modèle de recherche avec anti-rebond

Pour les interactions distantes, concevez les commandes de manière idempotente lorsque c’est possible. Une actualisation ou une recherche répétée ne doit pas corrompre le modèle. L’enregistrement, la validation et les transitions de processus doivent suivre un contrat d’action explicite et renvoyer un événement de résultat clair.

17.9 Composants fonctionnels réactifs et actions

Un composant fonctionnel peut exposer des actions visuelles comme recharger, sélectionner, ouvrir, valider, annuler ou soumettre. En mode web, ces actions sont généralement d’abord émises comme événements. Le modèle de page associe ensuite l’événement à une réaction locale ou à une action publiée.

```

1 button clicked
2   -> lgcVisual.ActionButton.Clicked local event
3   -> PageModel checks context and permissions
4   -> PageModel maps to RequestValidation
5   -> published action call
6   -> result event
7   -> local DataSet and statusView refresh

```

Exemple 17.7 – Association d’une action visuelle web à une action publiée

La distinction entre geste utilisateur, intention UXfc, action applicative et résultat métier est ainsi préservée.

17.10 Enregistrement des composants en mode web

Les composants web concrets doivent être enregistrés sous les noms de composants fonctionnels. L’enregistrement constitue une liaison de plateforme, pas une modification du modèle conceptuel.

```

1 functionalComponent: tabularView
2 platform: web/reactive
3 implementation: ReactTabularView
4 controller: WebTabularViewController
5 bindings:
6   dataSource: required
7   cursor: optional
8   actions: optional

```

Exemple 17.8 – Enregistrement de composants fonctionnels web

La même vue conceptuelle pourra ensuite utiliser une autre implémentation pour un framework web ou une enveloppe produit différente, en conservant la déclaration fonctionnelle.

17.11 Assemblage déclaratif des composants web

Une page web peut déclarer ses composants fonctionnels, surfaces de données et contrats d'événements locaux.

```

1 views:
2   - CustomerWorkspaceWeb:
3     target: web
4     pageController: CustomerWorkspacePageController
5     pageModel: CustomerWorkspacePageModel
6     dataSets:
7       - MainCursor:
8         dataSet:
9           name: Customers
10          provider: CustomerProvider
11          dataSource: CustomerDataSource
12          fields:
13            - Id:
14              type: String
15            - DisplayName:
16              type: String
17            - Status:
18              type: String
19      components:
20        - tabularView:
21          name: CustomerList
22          implementation: ReactTabularView
23          dataSource: CustomerDataSource
24          cursor: MainCursor
25          events:
26            - lgcVisual.Data.CurrentChanged local
27        - detailView:
28          name: CustomerSummary
29          implementation: ReactDetailView
30          cursor: MainCursor
31        - statusView:
32          name: CustomerStatus
33          implementation: ReactStatusView

```

Exemple 17.9 – Assemblage d'une page web réactive en syntaxe MODEL YAML

Les champs de DataSet présentés dans cette déclaration de vue ne classent pas des rôles conceptuels. Ils ne portent donc pas l'attribut `kind`. Les attributs `kind="slot"` et `kind="attribute"` appartiennent aux déclarations de champs du modèle dans les facettes.

17.12 Assemblage View XML équivalent

Cet exemple est une déclaration de vue/cache. Il diffère volontairement d'une déclaration de facette de modèle, où les champs peuvent être marqués `kind="slot"` ou `kind="attribute"`.

17.13 Esquisse d'implémentation réactive minimale

L'esquisse suivante montre comment une implémentation web concrète peut respecter les frontières logiCells.

```
1 function ReactTabularView(props) {
2   const rows = props.dataSource.records;
3   const currentId = props.cursor.currentId;
4
5   function selectRow(row) {
6     props.pageController.dispatchEvent({
7       eventClass: 'lgcVisual.Data.CurrentChanged',
8       source: props.name,
9       local: true,
10      params: { id: row.Id }
11    });
12  }
13
14  return renderTable(rows, currentId, selectRow);
15 }
```

Exemple 17.10 – Esquisse minimale de composant réactif

Le composant reçoit les données et un curseur de la page. Il émet un événement local lorsque la ligne courante change. Il ne met pas directement le fournisseur à jour et n'appelle pas lui-même de service métier.

17.14 États d'erreur, de chargement et hors ligne

Un composant web doit exposer un état fonctionnel sans masquer le modèle du fournisseur. Chargement, données obsolètes, mode hors ligne, délai de synchronisation et erreurs du fournisseur doivent être représentés comme états de page ou de DataSource.

```
1 DataSource state:
2   ready
3   loading
4   stale
5   offline
6   synchronizationPending
7   providerError
```

Exemple 17.11 – États fonctionnels des composants web réactifs

Un `statusView` peut s'abonner à ces états et afficher le retour approprié. Un `tabularView` peut continuer d'afficher les lignes en cache pendant l'actualisation du fournisseur. Un `autocompleteSelect` peut afficher d'abord les suggestions locales puis fusionner les résultats distants à la réponse du fournisseur.

17.15 Règles de conception

Implémentez le contrat UXfc avant son apparence visuelle. Liez les composants à `DataSource`, `DataSet` et `DataCursor` plutôt qu'aux fournisseurs directs. Émettez les événements locaux via `PageController`. Gardez l'état local au niveau local. Utilisez `PageModel` pour associer certains événements à des actualisations de fournisseurs, actions publiées, services ou processus. Rendez explicites les frontières distantes. Traitez l'implémentation des composants natifs comme un sujet distinct.

Chapitre 18

Syntaxe React moderne pour les composants logiCells

Les chapitres web de logiCells expliquent comment les composants réactifs se lient aux vues déclaratives, aux sources de données et aux événements de page. Ils supposent une connaissance minimale de la syntaxe moderne de JavaScript et React. Ce chapitre présente uniquement les formes employées dans les exemples de la plateforme ; il ne constitue pas un cours complet sur React.

18.1 Composants fonctionnels

Les nouveaux exemples doivent utiliser des composants fonctionnels plutôt que des composants de classe, sauf lorsqu'une compatibilité avec un composant existant impose une autre approche.

```
1 function TaskHeader(props) {  
2   return <h1>{props.caption}</h1>;  
3 }
```

Exemple 18.1 – Composant fonctionnel React

18.2 Déstructuration des props

Cette forme :

```
1 function TaskHeader({ caption, task }) {  
2   return <h1>{caption}: {task.Title}</h1>;  
3 }
```

Exemple 18.2 – Déstructuration des propriétés d'un composant

correspond à la déstructuration JavaScript. Elle équivaut à lire `props.caption` et `props.task`. Les propriétés réactives de logiCells peuvent véhiculer des structures de données, des contrôleurs, des noms de sources de données ou des attributs de vue ; les accolades relèvent de la syntaxe JavaScript, pas d'une déclaration de métadonnées logiCells.

18.3 Déstructuration des attributs de métadonnées

```
1 const { dataSource, caption } = dataStructure.attributes;
```

Exemple 18.3 – Lecture des métadonnées de vue générées

Cette instruction extrait deux propriétés de l'objet de métadonnées générées. Gérez toujours l'absence éventuelle d'attributs facultatifs lorsqu'un composant peut être utilisé dans plusieurs configurations de vues.

18.4 État local et hooks

```
1 const [searchText, setSearchText] = useState("");
```

Exemple 18.4 – État React local

`searchText` est un état local d'affichage. Il n'est pas automatiquement un état du modèle et n'est pas automatiquement écrit dans un `DataSet` `logiCells`. Synchronisez-le explicitement par une `DataSource`, une action ou un événement lorsque cette valeur appartient au modèle.

18.5 Effets et abonnements

```
1 useEffect(() => {  
2   const unsubscribe = controller.subscribe(refreshFromDataSet);  
3   return () => unsubscribe();  
4 }, [controller]);
```

Exemple 18.5 – Abonnement et nettoyage

La fonction de nettoyage évite les abonnements en double lorsqu'un composant est démonté ou recréé.

18.6 Gestionnaires d'événements

```
1 function handleOpen() {  
2   pageController.dispatchEvent({  
3     name: "Visual.Task.Open",  
4     objectId: task.Id  
5   });  
6 }
```

Exemple 18.6 – Émission d'un événement visuel local

Le nom et la charge utile de l'événement relèvent du contrat `View` de `logiCells`. La déclaration de fonction et l'objet littéral relèvent de JavaScript.

18.7 Outillage de référence

Pour les nouveaux exemples autonomes, privilégiez une chaîne d'outils actuelle telle que `Vite`, les composants fonctionnels et les hooks. Ne présentez pas `create-react-app` ou les composants de classe comme le choix par défaut pour un nouveau développement. Les projets existants peuvent encore utiliser des structures anciennes ; documentez-les comme des cas de compatibilité.

18.8 Frontière avec le guide `logiCells`

Utilisez la documentation officielle de React pour JavaScript, JSX, les props, l'état et les hooks. Utilisez ce guide pour les aspects propres à `logiCells` :

- les métadonnées générées et `dataStructure` ;
- l'enregistrement des composants ;
- `PageController` et l'émission d'événements locaux ;

- les liaisons DataSource/DataSet/DataCursor ;
- l'intégration des contrôleurs et d'ObjectBehind ;
- l'empaquetage du noyau web et des métadonnées applicatives générées.

Chapitre 19

Formulaires, listes et tableaux de bord dynamiques

Les interfaces dynamiques de logiCells reposent sur des vues, DataSet, curseurs, composants fonctionnels et réactions aux événements déclarés. Les formulaires, listes et tableaux de bord ne sont pas des blocs de code isolés : ce sont des projections de modèles conceptuels à l'exécution et des assemblages de composants fonctionnels UXfc.

19.1 Formulaires dynamiques

Un formulaire dynamique relie les rôles, actions et collections associées à un élément objet courant. Il reçoit son contexte d'un curseur et affiche les rôles au moyen de composants fonctionnels : champs de saisie, libellés, composants de sélection, cases à cocher, boutons, panneaux ou composants spécialisés.

Le formulaire doit être compris comme une surface d'interaction avec un concept. Si celui-ci évolue, le formulaire peut être adapté en modifiant la déclaration, sans réécrire tout le flux applicatif.

19.2 Composants de sélection

La sélection est une intention UXfc fréquente. Un composant de sélection ne se réduit pas à une liste déroulante. Sa fonction est de permettre à l'utilisateur d'identifier un élément dans un ensemble de candidats et de propager la valeur retenue dans le contexte courant.

Le composant utilise une source de données principale et une source de données de sélection. Il peut émettre un événement de changement sémantique traité par le modèle de page ou le contrôleur.

19.3 Listes, vues tabulaires et grilles

Les listes et grilles se lient généralement à des collections de niveau classe ou à des jointures de niveau objet. Une grille n'est pas propriétaire de la requête : elle consomme une projection de collection fournie par le modèle.

Une vue tabulaire peut offrir davantage qu'une grille simple : cellules modifiables, mise en forme, formules, événements de publication, de soumission ou de changement d'état coché, et opérations de masse. Le modèle déclare toujours le besoin fonctionnel ; l'implémentation enregistrée fournit les capacités concrètes.

19.4 Conteneurs : cadres fractionnés, à onglets et réactifs

Les conteneurs assemblent d'autres composants fonctionnels. Une vue fractionnée organise plusieurs sous-cadres en parallèle. Une vue à onglets ou un groupe empilé les organise séquentiellement. Un cadre de bureau réactif peut accueillir des vues enfants et les remplacer en fonction des événements de navigation ou de processus.

Il s'agit d'un assemblage de fonctions. L'apparence effective dépend des composants enregistrés, du style et de la plateforme.

19.5 Tableaux de bord comme assemblages

Un tableau de bord est une composition de projections et de fonctions. Il peut associer listes, compteurs, panneaux d'état, graphiques, états de processus, messages récents et actions. La règle essentielle est que chaque partie reste reliée à un concept, une collection, une action ou un service déclaré.

```
1 dashboard: OperationalOverview
2 context:
3   class: WorkItem
4 components:
5   - exploration:
6     dataSource: WorkItemsDataSource
7   - comparison:
8     dataSource: SelectedItemsDataSource
9   - status:
10    events: ProcessEvents
11   - validation:
12    action: ValidateWorkItem
```

Exemple 19.1 – Tableau de bord comme assemblage UXfc

Un tableau de bord ne doit pas devenir un agrégat procédural non documenté. Chaque composant doit posséder une source de données, un comportement événementiel et une intention fonctionnelle explicites.

19.6 La navigation comme flux d'événements

La navigation est elle aussi déclarative. Un bouton peut émettre `Navigation.Next`. Un composant à onglets ou empilé peut passer au cadre suivant. Un composant d'en-tête peut réagir à un événement de modification de titre. Un contrôleur de processus peut afficher une nouvelle vue lorsque le processus atteint un état donné.

Ce modèle sépare l'interaction de l'utilisateur de la sémantique du processus. Le même processus peut être présenté différemment sur une autre plateforme tout en conservant sa coordination sous-jacente.

19.7 Enregistrement des composants fonctionnels

Un composant fonctionnel exprime le rôle d'un élément dans l'interface. L'environnement d'exécution associe ce rôle à un composant concret grâce à l'enregistrement. Cette approche rend la couche visuelle portable et extensible.

Par exemple, la fonction déclarative `listView` peut être réalisée par une liste standard, une vue tabulaire ou une collection mobile. La fonction `tabbedView` peut être réalisée par des onglets, un groupe empilé ou un assistant propre à une plateforme. Le modèle déclare la fonction ; la plateforme choisit l'implémentation enregistrée.

19.8 Composants abstraits et concrets

Les composants abstraits décrivent des contrats d'interaction et ne sont pas instanciés directement. Les composants concrets réalisent ces contrats.

Cette séparation est utile pour les tableaux de bord. Un tableau de bord peut déclarer une fonction de visualisation ou d'exploration des données sans obliger toutes les plateformes à utiliser le même moteur d'affichage. L'une peut enregistrer une grille, une autre un graphique et une troisième un composant de navigation dans un graphe.

19.9 Règles de conception

Liez les composants à des DataSet déclarés, et non à des requêtes cachées. Utilisez les collections et les jointures pour préparer la structure des données avant leur consommation par la vue.

Utilisez les intentions UXfc pour décrire pourquoi un composant existe. Utilisez les composants fonctionnels pour décrire la fonction d'interaction nécessaire. Recourez à l'enregistrement pour sélectionner les implémentations concrètes. Gardez l'apparence, le style et l'identité du produit en dehors des exemples d'architecture réutilisables.

Séparez les événements de navigation des événements métier. L'utilisateur peut cliquer sur **Next**, mais c'est au modèle de processus de décider ce que cela signifie.

Chapitre 20

Composants fonctionnels, implémentations concrètes et visualisations

logiCells sépare la signification fonctionnelle d'un composant visuel de son implémentation concrète. C'est un principe central de la conception de la couche de publication. Un modèle déclare qu'une vue a besoin d'une fonction : sélection, consultation, affichage tabulaire, navigation, validation, affichage d'état, lecture de documents, visualisation de données ou autre capacité d'interaction. Le composant effectif est choisi ensuite par enregistrement.

20.1 Trois couches

La pile de composants comprend trois couches distinctes :

```
1 intentional UXfc component
2   -> why the interaction exists
3
4 functional component
5   -> what interaction function the view requires
6
7 concrete implementation
8   -> which registered class renders it on a platform
```

Exemple 20.1 – Pile de composants

Un style, un thème visuel, un ensemble d'icônes ou une disposition graphique ne constitue pas le composant fonctionnel. C'est un choix d'apparence appliqué une fois les couches fonctionnelle et concrète résolues.

20.2 Déclaration du composant fonctionnel

Un composant fonctionnel est déclaré dans la vue au moyen d'une balise logique. Celle-ci exprime une fonction, et non une dépendance envers une bibliothèque d'interface.

Cette déclaration signifie que la vue a besoin d'une fonction de sélection avec autocomplétion. Elle ne précise pas comment le composant est dessiné intérieurement.

20.3 Enregistrement

Une implémentation concrète est enregistrée en associant un espace de noms, une famille cible, un nom logique de composant, un nom de vue, une classe d'exécution et un contrôleur.

```

1 RegisterVisualFunctionalComponentClass(
2   'www.logicells.net',
3   'visual/desktop',
4   'listView',
5   'TabularView',
6   'rich tabular data component',
7   TabularView,
8   'Controllers.Desktop.ListViewControl');

```

Exemple 20.2 – Enregistrement d’un composant fonctionnel concret

Le nom logique est le contrat stable utilisé par la déclaration. Le nom de vue et la classe d’exécution identifient une implémentation concrète donnée.

20.4 Implémentations équivalentes

Un composant fonctionnel peut posséder plusieurs implémentations équivalentes. C’est un avantage, non un problème.

```

1 functional component: splittedView
2   concrete implementations:
3     absoluteLayoutFrame
4     relativeSplittedView
5
6 functional component: tabbedView
7   concrete implementations:
8     tabFrame
9     stackGroup
10    classicTabbedFrame
11
12 functional component: listView
13   concrete implementations:
14     standard list
15     tabular view
16     platform-specific list renderer

```

Exemple 20.3 – Implémentations concrètes équivalentes

Le modèle doit s’appuyer sur la fonction. La plateforme ou le package du produit cible choisit l’implémentation enregistrée.

20.5 Familles fonctionnelles disponibles

La documentation actuelle présente plusieurs familles importantes à faire figurer dans ce guide :

- les composants de sélection, notamment l’autocomplétion sur une source de données de sélection ;
- les composants d’édition booléenne ou d’état, notamment les cases à cocher ;
- les composants de consultation, comme les vues de documents, de navigateur ou d’applications distantes ;
- les conteneurs structurels, tels que les vues fractionnées et à onglets ;
- les vues d’état pour les journaux, l’avancement et les traitements ;
- les vues tabulaires pour l’affichage et l’édition riches de DataSet ;
- les composants d’action, comme les boutons émettant des événements visuels sémantiques ;
- les cadres réactifs accueillant des vues enfants et permettant de passer de l’une à l’autre.

Ces familles doivent être décrites comme des fonctions d’interaction réutilisables, et non comme des écrans de produits.

20.6 Composants abstraits

Un composant abstrait définit une fonction qui ne peut pas être instanciée directement. Il constitue un contrat, réalisé par des composants concrets.

Ainsi, un composant abstrait d'exploration peut préciser que l'utilisateur parcourt un graphe, une hiérarchie ou une collection à l'aide d'un curseur et de DataSet associés. Il peut définir les entrées et événements attendus sans imposer un arbre, un canevas de graphe ou une liste : ce sont des incarnations concrètes.

20.7 Événements et actions

Les composants fonctionnels participent au routage des événements. Un bouton émet un événement d'action visuelle. Une vue tabulaire peut émettre des événements de soumission et de publication. Un composant de sélection émet un événement de changement. Un composant de consultation peut émettre des événements de chargement de ressources et de navigation.

Les actions doivent rester fonctionnelles. Un composant doit exposer des actions telles que `ShowMessage`, `MoveTo`, `OpenResource`, `SubmitData` ou `UpdateContext`. Il doit éviter les actions qui divulguent la structure interne d'un widget, comme `SetEditTextContent`, sauf lorsqu'il s'agit volontairement d'un composant de bas niveau.

20.8 La visualisation comme fonction

Une visualisation n'est pas nécessairement un graphique. C'est un besoin fonctionnel consistant à rendre une structure compréhensible. La même fonction peut être réalisée par une grille, un graphique, une frise chronologique, un arbre, un navigateur de graphe ou un canevas spécialisé.

Le modèle doit décrire la source de données, les rôles visualisés, le contrat d'événements et les contraintes d'interprétation. L'implémentation enregistrée fournit la stratégie de rendu.

20.9 Règles de conception

Déclarez d'abord l'intention, ensuite le composant fonctionnel, puis enregistrez l'implémentation concrète. Appliquez l'apparence en dernier.

Conservez des noms de composants génériques et réutilisables. N'encodez pas l'identité du produit ni l'apparence propre à un client dans le modèle public. Préférez des noms tels que `Selection`, `Consultation`, `TabularView`, `ProcessStepFrame` et `StatusView`.

Utilisez le mécanisme d'enregistrement pour vous adapter aux cibles desktop, web, mobile ou autres. Ne réécrivez pas les modèles conceptuels lorsque le moteur de rendu change.

Chapitre 21

Publication web à partir de vues conceptuelles

Les vues logiCells sont des projections conceptuelles. Lorsqu'une vue est publiée sur le web, la publication doit préserver cette structure conceptuelle au lieu de transformer la vue en une page sans lien avec elle.

21.1 Les vues comme projections publiées

Une vue définit la partie d'un concept exposée, les DataSet utilisés, les actions disponibles et les composants fonctionnels qui présentent l'interaction. Un adaptateur de publication web peut associer cette vue à une page web ou à une arborescence de composants.

21.2 Composants fonctionnels

La vue déclarative doit décrire l'intention fonctionnelle : liste, formulaire, grille, bouton, champ, tableau de bord, signature, graphique, navigation ou conversation. L'implémentation web concrète est choisie par enregistrement. Cela permet de présenter différemment une même vue conceptuelle sur ordinateur, mobile ou web.

21.3 Actions dans les vues publiées

Une vue publiée ne doit exposer que les actions pertinentes dans son contexte. Certaines sont visibles aux utilisateurs, d'autres accessibles via des services, d'autres encore réservées à des processus ou agents. La publication des vues doit respecter les mêmes contrats au niveau du modèle.

21.4 Exemple

Le chemin désigne la ressource de vue déclarative. L'environnement d'exécution web décide quels composants concrets enregistrés affichent les éléments fonctionnels dans le cadre.

21.5 Règles de conception

N'inscrivez pas l'apparence propre à une plateforme dans le modèle conceptuel. Conservez une structure de vue déclarative. Utilisez les composants fonctionnels pour exprimer l'intention. Enregistrez des implémentations concrètes par plateforme. Publiez les actions de manière intentionnelle.

21.6 Synthèse

La publication web n'est pas une couche applicative indépendante : c'est un mode de publication des vues conceptuelles et des capacités du modèle. Lorsque la cible est un environnement d'exécution web réactif, les composants concrets doivent suivre le modèle des composants web fonctionnels : se lier aux surfaces de données de la page, émettre des événements locaux via le contrôleur de page et ne franchir la frontière du fournisseur ou du service que par des correspondances explicites dans le modèle de page.

Cinquième partie

Synchronisation et frontières externes

Chapitre 22

Applications conceptuelles « local-first »

Les applications View de logiCells peuvent rester réactives côté client pendant que certaines données sont synchronisées avec des fournisseurs distants. Il faut surtout distinguer l'*état d'interaction local* de l'*état de synchronisation distant*. Le client peut continuer de parcourir des projections en cache ou disponibles localement sans transformer chaque interaction en requête réseau.

22.1 Ce que signifie ici « local-first »

Dans ce guide, « local-first » ne signifie pas que chaque application possède une base de données cliente complète faisant autorité, ni qu'une fusion sans conflit de type CRDT est assurée. Cela signifie que l'architecture View peut conserver des DataSet/DataCursor locaux, des fragments locaux de graphe et un état visuel local, tandis que la synchronisation repose sur des fournisseurs explicites et des contrats de synchroniseur.

```
1 Functional component
2   -> local DataSource / DataSet / DataCursor
3   -> local collection or graph fragment
4   -> synchronization provider
5   -> remote runtime/source
```

Exemple 22.1 – Chaîne d'une application local-first

L'utilisateur interagit avec la projection locale. La frontière réseau n'est franchie que si l'application associe l'interaction à une actualisation du fournisseur, une action métier, une transition de processus ou une opération de synchronisation.

22.2 Cache local, fragments de graphe et instantanés

Un client peut conserver plusieurs ressources locales.

Ressource	Rôle
Cache local	Stocke les données prêtes à être affichées, les listes de référence, les objets récemment utilisés et les résultats temporaires.
Fragment de graphe local	Contient les objets conceptuels nécessaires à l'espace de travail ou à l'appareil courant.
Instantané	Capture un état cohérent pouvant être restauré ou comparé suivant le contrat d'exécution ou de fournisseur choisi.

Dépôt/journal de synchronisation	Enregistre l'état de synchronisation, les positions et les échanges en attente lorsque le synchroniseur utilise ces mécanismes.
État du fournisseur	Suit les curseurs, pages, horodatages, jetons ou marqueurs de continuation distants.

Ces ressources sont des projections opérationnelles. Elles doivent rester associées aux frontières conceptuelles : collections, actions, événements, processus et identités.

22.3 Événements locaux et remontée vers le distant

Le comportement local-first commence par la classification des événements.

```

1 visual event
2   -> PageControler
3   -> PageModel
4   -> local DataSet or DataCursor update
5   -> optional provider refresh
6   -> optional action or process event
7   -> optional synchronization exchange

```

Exemple 22.2 – Routage d'événements local-first

La sélection d'une ligne, le changement d'onglet ou le déplacement d'un curseur peuvent rester locaux. Une validation, une transition métier ou une modification persistante peuvent être associées à une action, un événement de processus ou une opération de synchronisation.

22.4 Ce que suit réellement le synchroniseur d'exécution

La documentation d'ingénierie de l'environnement d'exécution fournie décrit une couche de synchronisation distincte de la simple messagerie. Son contrat actuel comprend l'enregistrement explicite, des échanges de requêtes et réponses en lecture/écriture, des accusés de réception, des positions de journal ou d'horloge et un routage fondé sur des règles. La surface d'hypergraphe distribuée expose aussi des opérations de position causale autour d'horloges nommées et d'identifiants de nœuds.

L'application dispose ainsi de notions concrètes pour répondre notamment aux questions suivantes :

- quel nœud ou site est enregistré dans un groupe de synchronisation ;
- quelle position de journal ou d'horloge un participant a atteint ;
- quels changements doivent être acheminés vers quelles cibles ;
- à quel moment un échange lecture/écriture a été acquitté ;
- comment l'état de synchronisation est séparé du transport des messages.

La sémantique du transport et celle de la synchronisation sont distinctes. Un transport tel que ZeroMQ peut acheminer les messages de synchronisation, mais n'en définit pas la signification.

22.5 Fournisseurs distants comme frontières de synchronisation

Les fournisseurs distants doivent être considérés comme des frontières explicites, non comme la source de données directe de chaque composant visuel.

```

1 Remote provider
2   -> fetch / write / synchronize
3   -> resolve identities and positions
4   -> update local projection
5   -> emit provider/synchronization state

```

Exemple 22.3 – Le fournisseur comme frontière de synchronisation

Cela évite de lier directement le rendu de l'interface à la latence réseau et permet de tester la même View avec des jeux de données locaux ou des fournisseurs distants.

22.6 Travail hors ligne et sémantique des conflits

La documentation ne doit pas promettre plus que ce que garantissent le synchroniseur et le fournisseur choisis. En particulier, l'expression « local-first » ne répond pas, à elle seule, aux questions suivantes :

- les écritures sont-elles acceptées lorsqu'il n'y a aucune connexion ?
- comment deux modifications concurrentes du même objet conceptuel sont-elles réconciliées ?
- l'ordre des événements est-il total, causal ou propre au fournisseur ?
- quelle partie fait autorité après la reconnexion ?
- une action métier multi-objets est-elle atomique entre plusieurs sites ?
- comment les autorisations ou changements de version du modèle sont-ils revérifiés après la reconnexion ?

Pour une application qui dépend de ces comportements, documentez le contrat exact du fournisseur/synchroniseur et vérifiez-le dans un scénario de bout en bout. Ne remplacez pas une politique de résolution des conflits testée par une promesse générale de « fusion ».

22.7 Traitement courant selon le type d'opération

Opération	Traitement habituel
Déplacement du curseur, sélection d'onglet, déploiement de filtre	Événement visuel local.
Recherche, chargement différé, récupération d'une page	Requête du fournisseur, éventuellement distante.
Brouillon, sélection temporaire, annotation locale	Cache ou fragment de graphe local, selon la politique de persistance de l'application.
Validation métier, transition de workflow	Action conceptuelle ou événement de processus ; peut nécessiter la confirmation de l'environnement d'exécution faisant autorité.
Réplication/synchronisation	Échange explicite entre synchroniseur et fournisseur, avec état observable.

22.8 Règles de conception

Exposez les données locales par les DataSet, DataSource et curseurs. Placez les accès distants derrière des fournisseurs. Gardez les événements visuels locaux, sauf lorsqu'ils sont explicitement associés à des actions conceptuelles ou à des événements de processus. Traitez l'état de synchronisation comme un état d'exécution observable. Pour les scénarios hors ligne ou avec écritures concurrentes, publiez les garanties et les états d'échec exacts plutôt que de vous reposer uniquement sur l'étiquette « local-first ».

Sixième partie

Ingénierie et diagnostics

Chapitre 23

Frontières techniques des clients riches

Le diagnostic côté client doit d'abord déterminer quelle frontière est en cause : chargement de la View, comportement MODEL-behind, état DataSet/DataCursor, routage d'événements, échange asynchrone, synchronisation ou environnement d'exécution sous-jacent. Les transactions générales, la publication de services, l'ABI et les diagnostics d'exécution sont documentés une seule fois dans le Guide du développeur.

23.1 Diagnostiquer d'abord la frontière cliente

Pour un défaut propre à une View, consignez :

- le package View et la projection affichée ;
- la déclaration MODEL-behind concernée ;
- l'état DataSet/DataCursor et l'identité courante ;
- l'événement visuel ou fonctionnel déclencheur ;
- si l'échec s'est produit avant ou après un échange avec un fournisseur ou un système distant ;
- l'état du cycle de vie de la page/du contrôleur au retour d'une continuation asynchrone.

Cette méthode sépare le diagnostic client du diagnostic du modèle métier et évite de dupliquer la documentation générale de débogage.

23.2 La durée de vie du client fait partie de la correction

Un client disposant d'un cache combine plusieurs durées de vie : page/vue, DataSet/DataCursor, requête du fournisseur, agent de synchronisation et abonnement aux événements. Ne permettez ni à un résultat d'arrière-plan de mettre à jour une page déjà fermée, ni à une source d'événements de retenir un contrôleur détruit.

L'arrêt sûr d'une page ou d'un client suit généralement cet ordre :

1. cesser d'accepter de nouvelles tâches d'interface pour la page ;
2. se désabonner des événements visuels et des fournisseurs ;
3. annuler ou détacher les requêtes asynchrones en attente selon leur contrat ;
4. arrêter les minuteries et producteurs d'arrière-plan appartenant à la page ;
5. fermer les curseurs et fournisseurs non partagés appartenant à la page ;
6. libérer la hiérarchie View/contrôleur.

Les services applicatifs partagés et les synchroniseurs ont une durée de vie plus longue et sont arrêtés par l'environnement d'exécution de l'application, non par chaque View.

23.3 Frontières entre plateformes et bibliothèques d’interface

L’environnement d’exécution est disponible sous Windows, Linux et macOS et peut fonctionner dans des conteneurs. Une implémentation particulière du client riche peut présenter une matrice de prise en charge plus étroite. Une bibliothèque de widgets natifs, un adaptateur React, une fonctionnalité de navigateur ou un transport de synchronisation doit donc documenter ses propres contraintes de plateforme au lieu d’hériter automatiquement des garanties générales de l’environnement d’exécution.

23.4 Se reporter au Guide du développeur

Passez du diagnostic client à celui du noyau d’exécution lorsque la même erreur est reproductible sans la couche View, par exemple en invoquant la même action ou requête par CLI/API. Cela indique fortement que le défaut se situe au niveau du modèle, de la transaction, du fournisseur, du service ou de l’exécution, plutôt que dans la projection cliente.

23.5 Évolution de l’exécution et invalidation des vues

La modification d’une unité source ne justifie pas l’actualisation de toutes les vues. Le planificateur incrémental natif suit séparément les signatures *Structural*, *Semantic*, *Behavior*, *State*, *Persistence* et *View*, ainsi que des dépendants typés. Une vue doit être invalidée par sa propre signature View ou par une dépendance réellement consommée. C’est une règle d’architecture pour le futur rechargement à chaud conscient des vues, et non l’affirmation que tous les clients actuels, desktop ou web, assurent déjà un remplacement à chaud fin. Les éléments probants du 11 octobre (v37) valident le démarrage à froid des métadonnées sur un jeu de référence, mais laissent ouverts le delta du graphe sémantique, les notifications par vue et la liaison de sécurité à chaud.

Pour les actions FunctionalP intégrées déclarées en YAML, le débogueur de sources VS Code v1.0 exige une provenance du code source, une `ScriptIdentity` à l’exécution et des points d’arrêt tenant compte de la génération. Le débogueur AST d’exécution existant est un prérequis, et non la preuve que la correspondance source YAML vers AST fonctionne entièrement dans un client riche. Pour les limites exactes et les procédures CLI, consultez le Guide du développeur associé.

Septième partie

Modèles et références

Chapitre 24

Glossaire du client riche

View Projection côté client de l'état et des capacités conceptuels ; ce n'est pas le modèle métier lui-même.

MODEL-behind Comportement client déclaratif associé à une View.

ObjectBehind Objet client d'exécution reliant un composant visuel ou une View au comportement déclaré.

DataSet Projection tabulaire locale utilisée par le client comme état de travail ou cache.

DataCursor Contexte de navigation au-dessus d'un ou de plusieurs DataSet.

DataSource Surface de liaison permettant aux contrôles visuels de consommer l'état d'un DataSet.

PageControler Orthographe historique publique du point d'accès au contrôleur de page, employée par les scripts View existants. Ce guide la conserve systématiquement, car elle fait partie du contrat public actuel.

Événement visuel Signal d'interaction client acheminé dans l'architecture View avant de devenir une action métier, ou à sa place.

Package View Unité d'emballage regroupant les sources View et leurs déclarations côté client.

Cache local État projeté côté client permettant une interaction réactive sans redéfinir le modèle conceptuel de référence.

Chapitre 25

Modèles de vues conceptuelles

25.1 Périmètre

- vue d’objet ;
- vue de classe ;
- vue en liste ;
- vue de tableau de bord ;
- vue de formulaire ;
- vue publiée sur le web.

25.2 La View comme projection

Une vue projette un concept ; elle n’en est pas propriétaire. Elle peut définir des DataSet, des curseurs et des composants fonctionnels, mais la structure pérenne demeure dans le modèle.

25.3 Vues de classe et d’objet

Une vue `ClassItem` présente une collection, un résultat de recherche, un tableau de bord ou une action au niveau d’une classe. Une vue `ObjectItem` présente un objet conceptuel et ses rôles associés, ses actions ou ses collections filles.

25.4 Composants fonctionnels

Une vue peut demander un composant fonctionnel tel qu’une liste, une grille, un formulaire, un champ, un bouton ou une zone de signature. Le composant concret est choisi par enregistrement. Cela sépare la fonction déclarative de son implémentation propre à la bibliothèque d’interface et de son apparence visuelle finale.

25.5 Modèle d’assemblage UXfc

Une vue peut aussi se comprendre comme un assemblage de fonctions UXfc. Une vue de révision peut combiner consultation, sélection, affichage tabulaire et validation. La déclaration de vue nomme les composants fonctionnels ; l’environnement d’exécution résout leurs implémentations concrètes grâce à l’enregistrement.

25.6 Familles de composants fonctionnels

Les familles fonctionnelles courantes comprennent la consultation, l'exploration, la sélection, l'édition, la validation, la navigation, l'affichage d'état, l'affichage tabulaire, la disposition fractionnée et la disposition séquentielle. Elles décrivent l'activité de l'utilisateur et la fonction d'interaction, sans imposer d'apparence visuelle.