

logiCells

Rich Client Views and Interaction Contracts

Architecture Guide

Native and Web Interfaces,
Reactive Views and Events

Contents

About This Guide	7
I Orientation	8
1 When to Use the logiCells View Architecture	9
1.1 When a simpler client is enough	9
1.2 What remains shared	9
1.3 What this architecture adds	9
2 Prerequisites and the Boundary with the Developer Guide	10
2.1 Read in the Developer Guide when needed	10
3 Source Contract and Capability Status	11
3.1 Authority	11
3.2 Status Vocabulary	11
3.3 Platform Boundary	11
3.4 Synchronization Boundary	12
4 Views as Interaction Contracts and Projections of Concepts	13
4.1 Interaction Contract versus Rendered View	13
4.2 Agentic Projection without a Form	13
4.3 Projection, Intention and Function	14
4.4 View Declarations	14
4.5 ClassItem and ObjectItem Views	14
4.6 Views and Data Context	15
4.7 Functional Components	15
4.8 Abstract Components	15
4.9 Views and Events	15
II View Packages and Client Structure	16
5 Interaction Contracts, VIEW Packages and Artifacts	17
5.1 Canonical source layout	17
5.2 Why the Directory Is Named contracts	18
5.3 Explicit XML to MODEL binding	18
5.4 MODEL-behind uses the ordinary MODEL pipeline	18
5.5 VIEW package ownership	19
5.6 Incremental compilation	20
5.7 Compiled VIEW artifacts are durable runtime products	20
5.8 Regression gates	20

6	Building Task Views with DataSets and Page Models	22
6.1	The Three Parts of a View	22
6.2	Declare the Page Data Cache	22
6.3	Initialize the Page State	23
6.4	Filter a Dataset Before It Opens	24
6.5	Save and Cancel Actions	24
6.6	Bind Form Controls to DataSources	25
6.7	Button Actions	25
6.8	Register MODEL-behind Classes in the Manifest	25
6.9	Exercise	26
7	Views, DataSets, and Object Cursors	27
7.1	Views as Model-Driven Projections	27
7.2	Data Cursors	27
7.3	DataSetsManager as Navigation Contract	27
7.4	Collection Paths and Action Paths	28
7.5	Event-Driven View Behavior	28
7.6	Design Rule	28
8	UXfc Intentional Functional Components	29
8.1	From User Activity to Rendered Interface	29
8.2	Intentional Components	29
8.3	Functional Components	30
8.4	Concrete Implementations	30
8.5	Abstract Components	30
8.6	Assemblies of UXfc Components	30
8.7	Non-Visual Parameters	31
8.8	Local Visual Coordination	31
8.9	Events and Abstract Actions	31
8.10	Design Rules	31
9	ObjectBehind: The Behavioral Object of a View	32
9.1	The Mental Model	32
9.2	Source Form and Compilation Ownership	32
9.3	Lifecycle	33
9.4	Finding an ObjectBehind	33
9.5	Relation with DataSets	33
9.6	Event Flow	33
9.7	What Belongs in an ObjectBehind	34
9.8	Diagnostic Checklist	34
10	functionalP for logiCells Beginners	35
10.1	Where functionalP Appears	35
10.2	A Minimal Metadata Script	35
10.3	Variables and Assignment	36
10.4	Self, Sender and Result	36
10.5	Accessing the PageControler	36
10.6	Reading a Dataset	36
10.7	Initializing Local Page State	37
10.8	Filtering Before a Dataset Opens	37
10.9	Posting the Current Object	37
10.10	Creating an Object	37

10.11	Dispatching a Visual Event	38
10.12	Calling a Visual Component Object-Behind	38
10.13	When to Move Logic Elsewhere	38
10.14	Beginner Mistakes	38
10.15	Practical Reading Path	39
III	Client State, Cache and Event Routing	40
11	Data Cache for Functional Views	41
11.1	The View Data Cache	41
11.2	Why Functional Components Should Not Own Data Access	42
11.3	DataSet, DataSource and DataCursor	42
11.4	Relation with Collections and Providers	43
11.5	Master-Detail and Context Propagation	43
11.6	Remote Providers and Local Reactivity	43
11.7	Declarative Dataset Assembly	44
11.8	Functional Components Bound to the View Cache	44
11.9	Relation with PageControler	45
11.10	Design Rules	45
12	DataSetManager Events and Dataset-Driven Views	46
12.1	Why Dataset Events Matter	46
12.2	A DataSetManager Event Is Not Automatically a Business Event	46
12.3	Event Families	47
12.4	Events Observed in Dataset-Driven ERP Views	47
12.5	Before Posting: Validation as a Dataset Event	47
12.6	After Posting: Refreshing Dependent Views	48
12.7	Filter Datasets and Reactive Searches	48
12.8	Cursor Changes and Master-Detail Views	49
12.9	Event Parameters	49
12.10	Web Reactive Mapping	49
12.11	Recommended Event Names	50
12.12	Declarative Event Assembly	51
12.13	Design Rules	51
13	PageControler and Local Visual Event Dispatching	52
13.1	Why Local Visual Dispatch Exists	52
13.2	PageControler as the Local Event Bus	52
13.3	PageModel and Component objectBehind	53
13.4	React Views and Client-Side Dispatch	53
13.5	Native Views and DispatchVisualEvent	54
13.6	Four Levels of Events	54
13.7	Declarative Event Mapping	54
13.8	Datasets, Collections and Runtime Boundaries	55
13.9	UXfc Assemblies and Local Events	55
13.10	Design Rules	55

14 Event-Driven and Asynchronous VIEW Patterns	57
14.1 Think in continuations, not in async/await	57
14.2 Controller event subscription	57
14.3 Dispatch intent, not parent implementation	58
14.4 Waiting frames and completion signals	58
14.5 Dataset lifecycle pattern	59
14.6 Event roots and propagation	59
14.7 Keep VIEW controllers thin	59
14.8 Asynchronous VIEW review checklist	59
15 Reactive Desktop Frames and Event Routing	61
15.1 From Dataset-Driven Forms to Reactive Desktops	61
15.2 What a Reactive Desktop Frame Is	62
15.3 The Runtime Participants	62
15.4 Host Pattern: Dynamic View Region	63
15.5 Context Pattern: Input Cursor and Current Object	63
15.6 Application Event Sink Pattern	64
15.7 Child Notification Router Pattern	64
15.8 Child-to-Parent Bubbling Pattern	65
15.9 Dataset-Aware Reaction Pattern	66
15.10 Refresh-and-Recompute Pattern	66
15.11 Preview Surface Pattern	67
15.12 Workflow Shell Pattern	67
15.13 Event Taxonomy for Reactive Frames	68
15.14 Recommended Event Naming	68
15.15 Declarative Assembly	69
15.16 Web Reactive Mapping	69
15.17 Local State Versus Model State	70
15.18 Avoiding Event Loops	70
15.19 Error Handling and Diagnostics	70
15.20 Design Rules	71
15.21 Why This Pattern Matters	71
IV Native and Web Rich Clients	73
16 Native Rich-Client Hosts	74
16.1 The Stable Contract	74
16.2 Cache and Asynchrony	74
16.3 Platform and Toolkit Boundary	75
16.4 When to Prefer a Custom MVP Client	75
17 Building Web Reactive Functional Components	76
17.1 From UXfc Function to Web Component	76
17.2 Reactive Component Contract	76
17.3 Reactive Props and View Data Surfaces	77
17.4 Local State is Not Model State	77
17.5 Dispatching Local Visual Events	78
17.6 When an Event Crosses the Boundary	78
17.7 Subscription and Refresh Model	78
17.8 Debouncing, Batching and Idempotence	79
17.9 Reactive Functional Components and Actions	79

17.10	Component Registration for Web Mode	79
17.11	Declarative Web Component Assembly	80
17.12	Equivalent XML View Assembly	80
17.13	Minimal Reactive Implementation Sketch	80
17.14	Error, Loading and Offline States	81
17.15	Design Rules	81
18	Modern React Syntax for logiCells Components	82
18.1	Functional Components	82
18.2	Props Destructuring	82
18.3	Destructuring Metadata Attributes	82
18.4	Local State and Hooks	83
18.5	Effects and Subscriptions	83
18.6	Event Handlers	83
18.7	Tooling Baseline	83
18.8	Boundary with the logiCells Guide	83
19	Dynamic Forms, Lists and Dashboards	85
19.1	Dynamic Forms	85
19.2	Selection Components	85
19.3	Lists, Tabular Views and Grids	85
19.4	Containers: Split, Tabbed and Reactive Frames	85
19.5	Dashboards as Assemblies	86
19.6	Navigation as Event Flow	86
19.7	Functional Component Registration	86
19.8	Abstract Versus Concrete Components	86
19.9	Design Rules	87
20	Functional Components, Concrete Implementations and Visualizations	88
20.1	Three Layers	88
20.2	Functional Component Declaration	88
20.3	Registration	88
20.4	Equivalent Implementations	89
20.5	Available Functional Families	89
20.6	Abstract Components	90
20.7	Events and Actions	90
20.8	Visualization as a Function	90
20.9	Design Rules	90
21	Web Publication from Conceptual Views	91
21.1	Views as Published Projections	91
21.2	Functional Components	91
21.3	Actions in Published Views	91
21.4	Example	91
21.5	Design Rules	91
21.6	Summary	92
V	Synchronization and External Boundaries	93
22	Local-first Conceptual Applications	94
22.1	What “Local-first” Means Here	94

22.2	Local Cache, Graph Fragments and Snapshots	94
22.3	Local Events and Remote Escalation	95
22.4	What the Runtime Synchronizer Actually Tracks	95
22.5	Remote Providers as Synchronization Boundaries	95
22.6	Offline Work and Conflict Semantics	96
22.7	Typical Handling by Operation Type	96
22.8	Design Rules	96
VI	Engineering and Diagnostics	97
23	Rich-Client Engineering Boundaries	98
23.1	Diagnose the Client Boundary First	98
23.2	Client Lifetime Is Part of Correctness	98
23.3	Platform and Toolkit Boundaries	99
23.4	Escalate to the Developer Guide	99
23.5	Runtime Evolution and View Invalidation	99
VII	Reference Patterns	100
24	Rich-Client Glossary	101
25	Conceptual View Patterns	102
25.1	Scope	102
25.2	View as Projection	102
25.3	Class and Object Views	102
25.4	Functional Components	102
25.5	UXfc Assembly Pattern	102
25.6	Functional Component Families	103

About This Guide

This guide documents the optional logiCells MVC/View architecture for rich clients. It is intended for native or web applications that benefit from local client state, DataSet/DataCursor-based caches, declarative projections, event routing and asynchronous interaction with the conceptual runtime.

The View architecture is not required to use logiCells. The same executable conceptual model can be used through CLI applications, APIs/services, MVP-style clients or other presentation technologies. Those core concerns belong to the main *Declarative Conceptual Programming with the logiCells Platform – Developer Guide*.

This guide therefore starts from a different question: *when the application needs a rich interactive client, how should the client project, cache and coordinate conceptual state without redefining the business model?*

Runtime platforms

The underlying logiCells runtime is available on Windows, Linux and macOS and can be deployed in containers. The View architecture described here is a client architecture over that runtime; platform-specific packaging is outside this guide and remains a deployment concern.

Scope boundary

The business model remains authoritative for concepts, relations, constraints, actions and permissions. View metadata describes client projections and interaction. DataSets and cursors are client-side projection/cache mechanisms; they are not replacements for model fields or conceptual relations.

Part I

Orientation

Chapter 1

When to Use the logiCells View Architecture

Use the View architecture when the client needs more than a thin invocation surface over model actions. Typical reasons include rich native or web interaction, local cache and navigation state, asynchronous operations, master/detail coordination, reusable declarative view packages, visual event routing, and controlled synchronization between local projections and the runtime.

1.1 When a simpler client is enough

Do not introduce the View stack when a CLI, API consumer, MVP-style client or another presentation technology already satisfies the requirement. The View architecture is an option, not a prerequisite for conceptual programming.

1.2 What remains shared

Whichever client architecture is chosen, business meaning should remain in the same places: concepts and relations describe the domain, constraints and H-Logic express validity, actions expose intentions, and publication boundaries expose selected capabilities.

1.3 What this architecture adds

The View stack adds a client-oriented execution layer around those capabilities:

```
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
```

Listing 1.1: Rich-client concern chain

The rest of this guide explains each step without treating it as part of the conceptual model itself.

Chapter 2

Prerequisites and the Boundary with the Developer Guide

This guide assumes that the executable conceptual model already exists. Concepts, relations, actions, permissions, transactions, service publication and general runtime diagnostics are defined in the companion *Developer Guide*; they are not redefined here.

Use this guide only when the client needs one or more of the following: a local DataSet/-DataCursor cache, asynchronous interaction, client-side event routing, reusable View packages, or a native/web rich-client projection managed by the logiCells View architecture.

2.1 Read in the Developer Guide when needed

For the conceptual model, read *The logiCells Mental Model*. For transaction and identity guarantees, read *Transactions, Consistency and Identity*. For API publication and external integration, use the Developer Guide chapters on services and publication. For general logging, debugging and troubleshooting, use the engineering chapters there.

The rule is intentional: this book documents the client projection layer. The Developer Guide remains the authority for the business model and runtime contracts that are independent of a View.

Chapter 3

Source Contract and Capability Status

This guide owns the optional client architecture only. It does not redefine the declarative MODEL grammar, H-Logic, transactions, ABI or general runtime semantics.

3.1 Authority

When a client example conflicts with a core contract, use this order:

1. current framework grammar, public interfaces and executable tests;
2. generated runtime/binding contracts for the selected distribution;
3. verified runtime-programming examples for the target subsystem;
4. the main Developer Guide for conceptual/runtime semantics;
5. this Rich Client guide for View, DataSet/DataCursor, event-routing and client-cache patterns;
6. book-owned examples and integration patterns.

A View example must not silently introduce a second MODEL dialect. XML in this guide is for View artifacts; MODEL declarations remain aligned with the canonical declarative grammar documented in the Developer Guide.

3.2 Status Vocabulary

The same vocabulary as the Developer Guide applies: **Verified**, **Available**, **Integration pattern**, **Design / in progress**, and **Planned**. A client mechanism must not be described as more mature than the runtime surface it depends on.

3.3 Platform Boundary

The underlying runtime is available on Windows, Linux and macOS and can be deployed in containers. A particular View implementation, UI toolkit, native component or synchronization transport can still have a narrower platform scope. This guide therefore separates runtime-platform availability from client-toolkit certification.

3.4 Synchronization Boundary

logiCells synchronization uses explicit registration, read/write exchanges, acknowledgements, journal positions, causal clocks and routing rules. Applications must still define how competing changes are handled: concurrent edits are not automatically guaranteed to merge without conflicts.

Whenever an application requires offline writes, concurrent mutation or automatic conflict resolution, document and test the exact synchronizer/provider contract used by that application.

Chapter 4

Views as Interaction Contracts and Projections of Concepts

A logiCells view is not primarily a screen definition. In the modern architecture, the stable artifact is an *interaction contract*: a channel-independent description of how a user can obtain information, provide missing information, invoke capabilities and react to application state. The user may be a human user or an autonomous agent.

The historical term *view* remains useful for compatibility and for rendered projections, but files under `meta/views/contracts` should be read as interaction contracts rather than as forms tied to one toolkit. A web page, a native form and an agentic dialogue may all consume the same contract differently.

This chapter explains why views should be read as conceptual interaction projections rather than as isolated screens. It also introduces the UXfc reading of contracts: an interaction contract is an assembly of intentions, data requirements, actions, events and functional components, not merely a layout of concrete widgets.

4.1 Interaction Contract versus Rendered View

The terminology is deliberately split:

```
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
```

Listing 4.1: Contract and projection terminology

A contract can therefore be compared to an evolved sequence diagram: it describes the possible exchanges between a participant and application services, while also carrying the data and behavioral metadata required to drive those exchanges. It does not require every exchange to be presented visually.

4.2 Agentic Projection without a Form

An agentic consumer may use an interaction contract without materializing a form. It can inspect the data available in the current interaction, determine which required values are still

missing, ask only for those values, invoke a declared action, and present the result in a conversational or tool-oriented form.

For example, if an operation needs a customer, an amount and a reason, an agent can reuse values already known from context and ask only for the missing amount. The contract remains the common source of interaction semantics even though no visual form is displayed.

This is an important architectural boundary: the contract describes *what interaction is possible and required*; a renderer or agent runtime decides *how that interaction is enacted*.

4.3 Projection, Intention and Function

A logiCells view should be read through three 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

```

Listing 4.2: Three questions for a view

The exact appearance, or the absence of a visual appearance, comes after these questions. An interaction contract can keep the same conceptual projection and UXfc intention while changing its concrete rendering or being consumed by an agent.

4.4 View Declarations

A view declaration associates an interaction-contract name with a target and one or more contract resources. A renderer may later interpret those resources visually, while an agent may interpret them as an interaction plan.

```

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'

```

Listing 4.3: Conceptual view declaration

The contract says how a concept can participate in an interaction. It does not say that this is the only possible representation or channel for the concept.

4.5 ClassItem and ObjectItem Views

A class-level view usually displays a collection or list of instances. It is useful for browsing, searching, selecting or creating objects.

An object-level view displays or edits one object item. It is useful for detail forms, process steps, validation screens or object-specific dashboards.

The distinction is important because it controls the context injected into the view. A class item view receives a concept-level context. An object item view receives an instance-level context.

4.6 Views and Data Context

Views are connected to data through cursors, datasets, data sources and collection paths. A view does not need to know how the database is structured. It binds to conceptual collections declared on the model.

The collection is conceptual. The view only consumes the projection.

4.7 Functional Components

Visual declarations use functional components. A functional component is a declarative description of an interaction function: button, list, grid, field, frame, responsive layout, signature area, chart, document viewer or another interaction role.

The declaration is not necessarily the concrete implementation. A component such as `lgc#button`, `lgc#gridPanel` or `lgc#reactiveDesktopFrame` identifies a function in the view model. The effective component used at runtime is selected through registration of a concrete implementation.

This separates three concerns:

```
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"
```

Listing 4.4: Functional component separation

The reusable pattern is not one particular screen appearance. The reusable pattern is the declarative functional component model and the registration mechanism that binds it to concrete components.

4.8 Abstract Components

Some components are abstract. An abstract component defines a function but is not directly instantiable. It exists to specify a contract that concrete components must implement.

For example, a generic data visualization function may define that a component consumes a data source and displays structured data. A concrete implementation may render this as a grid, a chart, a timeline or a specialized visual control.

Abstract components are useful because they let the model express stable interaction semantics while leaving the concrete rendering open.

4.9 Views and Events

Views can react to events and emit events. A frame may react to `OnShow`. A button may emit a visual action event. A page model may show or hide navigation components. A controller may listen to a process event and show a view.

This means that the contract layer is also declarative and reactive. It is not merely a static layout. It is an interaction projection with declared event behavior that may be enacted by native, web or agentic consumers.

Part II

View Packages and Client Structure

Chapter 5

Interaction Contracts, VIEW Packages, MODEL-Behind and Compiled Artifacts

Modern logiCells VIEW packages use a strict separation between interaction-contract source and executable MODEL metadata. The historical word *view* remains in file types and APIs, but `meta/views/contracts` contains channel-independent interaction contracts, not toolkit-specific screen definitions. This chapter defines the convention for new modules and migrations from legacy forms and frames.

5.1 Canonical source layout

A package that owns views should expose a view tree such as:

```
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
```

Listing 5.1: Recommended VIEW contract tree

The contract source is `*.view.xml`. It may contain visual layout information, but its architectural role is broader: it declares an interaction surface that can be projected to native, web or agentic channels. MODEL-behind classes belong in colocated `*.models.yaml` files under a YAML `classes:` root. A `*.models.yaml` file is generated only when the corresponding contract contains MODEL metadata to externalize.

The `meta/views/models` tree has a different purpose. It contains minimal MODEL redefinitions or overlays of existing business classes when view-related metadata must be separated from the business MODEL package. It is not a second copy of the contract tree.

5.2 Why the Directory Is Named `contracts`

The directory name `contracts` is intentional. These artifacts define the interaction contract between a participant and application capabilities. The participant may be a human user or an autonomous agent. A native or web renderer can materialize fields, lists and actions as controls; an agent can instead inspect the same contract, present available information, request only missing values and invoke declared capabilities without displaying a form.

A useful mental model is an evolved sequence diagram: the contract describes the possible exchanges, data requirements, events and actions between the participant and application services. Concrete rendering is a projection of that contract, not its definition.

5.3 Explicit XML to MODEL binding

A view declares its page controller and component models explicitly.

```

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>

```

Listing 5.2: A VIEW XML binding to MODEL-behind classes

Do not infer controller identity from filenames alone. The binding is explicit and should be validated. A contract with no MODEL binding does not require an artificial `*.models.yaml` companion.

5.4 MODEL-behind uses the ordinary MODEL pipeline

```

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();

```

Listing 5.3: MODEL-behind controller skeleton

These classes are compiled with the normal package/MODEL machinery. Class discovery is declared in the canonical `_manifest.yaml`. A manifest entry for a class stored in a multi-class container names that physical container with `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

```

Listing 5.4: Manifest entries for classes stored in one `.models.yaml` file

The loader resolves `fileName`, opens the `classes:` container, selects the requested class by name and passes that class through the ordinary MODEL loader. The old role of a VIEW compiler that also compiled inline MODEL metadata is deprecated. VIEW-specific compilation should assemble and validate dirty interaction artifacts and bind identities already compiled by MODEL.

5.5 VIEW package ownership

The companion VIEW package owns the view tree and participates in the application compilation plan.

```

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

```

Listing 5.5: Companion VIEW package with two schema roots

Items are relative to the declared schema roots. Do not prefix an item with `contracts.` or `models.` merely because the physical file lives below one of those directories. Keep the package local: a module should own only the contracts and view-model overlays that belong to that package. A missing `viewPaths` declaration must not turn into ownership of the complete global `meta/views` tree.

5.6 Incremental compilation

MODEL and VIEW deltas are planned separately:

```

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

```

Listing 5.6: Incremental consequences

A dirty view is recomposed deterministically from its current inputs. Reusing parsed unchanged fragments is an optimization, not an alternative source of truth.

5.7 Compiled VIEW artifacts are durable runtime products

Merged `*.view.xml` files are compiled products. They belong under `compiledPackagesBasePath`, in the same accepted generation as package `.lcmp` and `.ast` artifacts. `cachePath` contains only reconstructible caches and must not be required by the runtime to open a compiled view.

A deployed compiled-only generation should be able to start from:

```

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

```

Listing 5.7: Compiled-only VIEW runtime inputs

MODEL/VIEW sources can be omitted from a compiled-only distribution when the generation manifest is authoritative and complete. Development source-aware validation and deployed compiled-only validation are therefore distinct modes.

5.8 Regression gates

For each module with views, automate these checks:

- unchanged compile plan reports no dirty VIEW artifacts;
- one XML edit produces a bounded VIEW closure;
- one MODEL-behind binding edit recompiles the affected MODEL and dependent VIEW;

- immediate rerun is a no-op;
- full and incremental builds produce equal merged-view digests;
- `.lcmp`, `.ast` and merged view XML are from the same generation;
- a compiled-only deployment opens its views without the source tree.

Chapter 6

Building Task Views with DataSets and Page Models

A useful application is not only a model. It needs a page that exposes a coherent data cache, page behavior and visible components. This tutorial builds a richer **Task** view using the modern split: visual hierarchy in `*.view.xml` and page/controller MODEL-behind in `*.models.yaml`.

When a listing contains **functionalP**, read Chapter 10, “functionalP for logiCells Beginners” first. The script is not the model syntax; it is an executable fragment inside a declared action or event handler.

6.1 The Three Parts of a View

A logiCells view is usually organized around three complementary parts.

```
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
```

Listing 6.1: Interaction-contract source split

The **dataSets** section is the local interaction cache. The **pageModel** section binds behavior to the contract. The **masterView** section describes one visible projection, but it does not make the contract native- or web-only.

When a contract contains no MODEL/page-model/component-model declaration, no empty `*.models.yaml` file should be generated. Conversely, when several MODEL classes belong to one contract, they can share one `*.models.yaml` container and be discovered through `_manifest.yaml` using `fileName`.

6.2 Declare the Page Data Cache

The task page uses a main task dataset, a list dataset, two reference datasets and a small local memory dataset for filters.

```
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

```

Listing 6.2: Task view data cache in YAML

Do not put `kind="slot"` or `kind="attribute"` on DataSet fields. That attribute belongs to model fields in concept facets, not to view caches.

6.3 Initialize the Page State

When the page opens, it initializes the local memory dataset used by filters.

```

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;

```

Listing 6.3: Page OnShow event and action in YAML

6.4 Filter a Dataset Before It Opens

The list dataset can be sorted and filtered before it opens.

```

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;

```

Listing 6.4: BeforeDataSetOpened filter in YAML

6.5 Save and Cancel Actions

The save action posts the current object and asks the parent view to refresh. The cancel action closes the form without posting.

```

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();

```

Listing 6.5: Save and cancel actions in YAML

Use neutral event names in public documentation. Project-specific names must not leak into reusable examples.

6.6 Bind Form Controls to DataSources

The visual hierarchy belongs in `*.view.xml`. Controls bind to `DataSource` names, not directly to SQL tables or remote providers.

```

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>

```

Listing 6.6: Task edit controls in TaskEdit.view.xml

6.7 Button Actions

Button bindings are also visual metadata and therefore remain in `*.view.xml`. The business behavior stays in the MODEL-behind YAML action.

```

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>

```

Listing 6.7: Button action binding in TaskEdit.view.xml

6.8 Register MODEL-behind Classes in the Manifest

The local manifest registers each MODEL-behind class and points to the physical multi-class container.

```

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
```

Listing 6.8: TaskEdit MODEL-behind manifest

The class name selects an entry inside `TaskEdit.models.yaml`; the file name identifies the physical source; the `classId` remains the stable logical identity used by the runtime.

6.9 Exercise

Add a short `Tag` field to the task model, to `TaskEntityDs` and to the visible form. Then add `OnlyMine` to `TaskMemoryDs` and extend `OnBeforeOpenDataset`. Use the exercise as a local extension of this tutorial; general model-testing practices are covered in the companion Developer Guide.

Chapter 7

Views, DataSets, and Object Cursors

logiCells views are projections of conceptual objects. They are not isolated screen descriptions. A view binds visual components to object cursors, datasets, actions, collections, and events.

This chapter introduces the runtime navigation pattern behind forms, frames, lists, dashboards, and process screens.

7.1 Views as Model-Driven Projections

A view declaration associates a name with one or more visual resources.

```
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'
```

Listing 7.1: View declaration on a conceptual class

The view name is part of the model contract. A process, controller, or action can ask the runtime to show **BaseFrame** for a given object. The runtime resolves the view through metadata.

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

Listing 7.2: Showing a view for an object

7.2 Data Cursors

A data cursor carries the current object context for a view. It can point to the object being displayed, to a related object, or to a child context defined by a collection.

The view can then bind visual components to **EntityDataSource**. The same pattern works for simple fields, tables, grids, lists, action parameters, and nested cursors.

7.3 DataSetsManager as Navigation Contract

At runtime, a view can retrieve cursors and datasets through its datasets manager.

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

Listing 7.3: Retrieving the current object from a cursor

This is not only a UI convenience. It is a general navigation contract:

- resolve the current object from the view context;
- retrieve datasets by name;
- use master-detail cursors to propagate context;
- bind action parameters to datasets;
- route events based on the current visual and object context.

7.4 Collection Paths and Action Paths

A dataset can be backed by a named collection or by a path to a collection or action.

The second pattern is important: an action can expose a parameter structure that a view uses as a temporary interaction model. This allows forms to bind to actions without creating a permanent business object too early.

7.5 Event-Driven View Behavior

A view can define its own page model. The page model has data, events, and actions just like other conceptual objects. It can listen to visual events and application events, then update visibility, navigation, titles, or child frames.

The event handler can then retrieve the current object, update the header, show or hide navigation controls, or open the next conceptual view.

7.6 Design Rule

The practical rule is:

A view should project a conceptual context; it should not become the owner of business logic.

Business state belongs to business objects. Cross-object coordination belongs to process objects. Navigation and display adaptation belong to controllers and page models. Datasets and cursors are the bridge between these layers.

Chapter 8

UXfc Intentional Functional Components

The visual layer of logiCells is not only a collection of widgets. It is organized around functional and intentional components. A functional component declares what an area of a view is meant to support. An intentional UXfc component goes one level higher: it declares why that interaction exists and which user activity it is meant to sustain.

This distinction is important for public models. A reusable view should not be defined by the exact appearance of one screen. It should be defined by the conceptual object being projected, the user activity being supported, the functional components required by that activity, and the concrete implementations registered for the target platform.

8.1 From User Activity to Rendered Interface

An UXfc component starts from the activity of the user, not from a graphical control. The design path is:

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

Listing 8.1: UXfc design path

This path prevents the model from confusing a business or interaction intention with a particular rendering toolkit. It also allows several interfaces to be derived from the same conceptual intention.

8.2 Intentional Components

An intentional component describes a verb of use. Examples include exploration, selection, comparison, consultation, edition, validation, navigation and sequencing. These verbs are not arbitrary labels. They determine the expected data structures, the relations between elements, the interaction modes, the assembly constraints and the criteria for UX coherence.

For example, consultation is not the same as displaying a browser widget. Consultation means that the user is reading or navigating an addressable resource under a context. The implementation may be a browser frame, a document viewer, an embedded web view or a remote application view. The intention remains the same.

8.3 Functional Components

A functional component describes the interaction function required by a view. It may be a field, a selection control, a list view, a tabular view, a split view, a tabbed view, a status view, a reactive desktop frame, a document viewer or another reusable function.

The functional component remains declarative. It may carry non-visual parameters such as data source names, selection fields, navigation mode, history policy, context persistence, validation mode or event contracts. These parameters describe how the user activity is supported, not how the component is painted.

8.4 Concrete Implementations

The effective runtime component is selected through registration. The registration maps a functional component name to a concrete class, view name, controller and target platform.

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

Listing 8.2: Functional component registration principle

The same logical function can have several concrete implementations. A tabbed view may be rendered as tabs, as a stack group, or by another sequential frame. A split view may be rendered by an absolute layout frame or by a relative split frame. A list view may be rendered by a standard list, a tabular grid, or a platform-specific collection component.

8.5 Abstract Components

Some UXfc components are abstract. They define an interaction function but are not directly instantiable. They specify a contract: expected inputs, supported events, composition constraints and possible incarnations.

An abstract exploration component, for example, may define that the user explores a graph, hierarchy or linked dataset through a cursor and related data sources. A concrete implementation may render this as a tree, a graph navigator, a master-detail browser, a sequence of cards or a search-and-filter workspace.

8.6 Assemblies of UXfc Components

A view usually assembles several functional components. The assembly describes how interaction functions collaborate.

```
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
```

Listing 8.3: View assembly with UXfc functions

The assembly is not the visual appearance. It is the functional composition of the interaction. A desktop product, a mobile product and a web product may render the same assembly differently.

8.7 Non-Visual Parameters

UXfc parameters should remain functional and non-visual whenever possible. Examples include:

- initial resource or URI;
- data source and selection data source;
- selected identifier and displayed name;
- navigation mode;
- history policy;
- validation prerequisites;
- whether selection is exclusive, multiple or conditional;
- whether comparison requires homogeneous items;
- whether sequencing is linear, reversible or conditional.

Appearance parameters such as colors, skins, icon sets and exact layout should remain outside the reusable architectural pattern.

8.8 Local Visual Coordination

When UXfc components are assembled into a running page, their local events should normally be routed through the page controller before any escalation to the application model. A selection change, tab change, consultation reload or dataset notification can update the local page model without becoming a server call or a business event. The detailed runtime pattern is described in the chapter on **PageController** and local visual event dispatching. The web-specific incarnation of this pattern is described in the chapter on building web reactive functional components.

8.9 Events and Abstract Actions

Intentional and functional components expose semantic events and abstract actions. A consultation component may accept a **Browse(uri)** intent and emit **ResourceLoaded** or **NavigationFailed**. A selection component may emit **SelectionChanged**. A tabular view may emit post or submit events. A button may emit a visual action event.

The important rule is that events should describe what happened semantically, not which widget implementation produced it.

8.10 Design Rules

Start from the user intention before choosing a component. Choose the functional component before choosing the concrete implementation. Register concrete implementations per platform or package. Keep appearance separate from functional declarations. Use abstract components to describe stable interaction contracts. Use events to connect view behavior to page models, controllers, process objects and business actions.

Chapter 9

ObjectBehind: The Behavioral Object of a View

`ObjectBehind` is the runtime behavior attached to a page, frame or visual component. It is easy to confuse it with an `ObjectItem`, because both are objects and both can expose actions. Their responsibilities are different.

9.1 The Mental Model

```
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 PageControler
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
```

Listing 9.1: Runtime responsibilities around a view

The `ObjectItem` should carry business state and business capabilities. The `ObjectBehind` coordinates local view behavior: open or close a frame, react to a visual event, access a dataset, change component state, or delegate to a business action.

9.2 Source Form and Compilation Ownership

An `ObjectBehind` extracted from a `VIEW` contract is still an ordinary `MODEL` class. Its source may be stored in a colocated `*.models.yaml` container and referenced by `source`, `class` and `classId`. Discovery is performed through `_.manifest.yaml`, whose `fileName` identifies the physical multi-class container.

The `VIEW` layer owns the interaction binding, but it does not own a second `MODEL` compiler. The selected class goes through the normal `MODEL` loading and compilation pipeline. This keeps native, web and agentic projections aligned on the same executable behavioral meta-data.

9.3 Lifecycle

A typical lifecycle is:

```
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
```

Listing 9.2: ObjectBehind lifecycle

The exact construction order depends on the runtime and view type, but the ownership principle remains stable: an `ObjectBehind` belongs to the local view boundary.

9.4 Finding an ObjectBehind

View scripts commonly resolve local behaviors through the page controller:

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

Listing 9.3: Resolving a visual ObjectBehind by name

Or through a visual identifier:

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

Listing 9.4: Resolving a visual component by id

These operations are local UI coordination. They should not become hidden replacements for business actions.

9.5 Relation with DataSets

An `ObjectBehind` may read the current dataset context:

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

Listing 9.5: ObjectBehind accessing a DataSet

A good separation is:

```
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
```

Listing 9.6: Separation of responsibilities

9.6 Event Flow

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

Listing 9.7: Typical visual event flow

This pattern is developed further in Chapter 13, “PageController and Local Visual Event Dispatching” and Chapter 15, “Reactive Desktop Frames and Event Routing”.

9.7 What Belongs in an ObjectBehind

Suitable responsibilities include:

- local initialization of a page;
- selecting or refreshing a local dataset;
- translating a component event into a published action;
- coordinating child views;
- controlling local visibility, selection or navigation;
- dispatching a local visual event.

Avoid placing the following directly in the ObjectBehind:

- reusable business validation;
- domain rules shared by several views;
- long-running coordination;
- integration logic that belongs to a service or provider;
- persistence rules that must also apply outside the UI.

9.8 Diagnostic Checklist

When a visual action does not run, verify:

1. the component name or id;
2. the page controller that owns the component;
3. the ObjectBehind model attached to the view or frame;
4. the declared event and action path;
5. the dataset name used by the script;
6. whether the event is local, parent-routed or application-level.

Chapter 10

functionalP for logiCells Beginners

logiCells MODEL metadata is authored in YAML; visual VIEW sources use `*.view.xml`. Sometimes a model also needs a small executable fragment: initialize a page state, validate a dataset before it is posted, create an object, refresh a parent view or call a runtime action. In those places the metadata often uses **functionalP** scripts.

This chapter is not a complete language reference. It is the practical subset a beginner needs to read the scripts used in the first task-management tutorial and in dataset-driven views. The full language-oriented reference is in the functionalP appendix of the companion *Developer Guide*. A maintainer source map is retained in the repository but is intentionally excluded from the business-developer reading path.

10.1 Where functionalP Appears

A beginner usually meets **functionalP** in four places:

```
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.
```

Listing 10.1: Common functionalP locations

The important rule is that **functionalP** is not the model syntax. The MODEL remains YAML. The script is an executable fragment owned by a declared action or event handler.

10.2 A Minimal Metadata Script

A script is normally stored inside an action. The surrounding YAML MODEL declares the action; the script provides the executable body.

In practice, many scripts do not explicitly assign **Result**. They perform an effect such as initializing a dataset or posting an object. For beginner code, use **Result** when the action has an expected return value. This makes intent easier to debug.

10.3 Variables and Assignment

The assignment operator is `:=`. Variables are often introduced by assignment:

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

Listing 10.2: Local variables

Use local variable names that make runtime objects visible. A common convention is to prefix local variables with `l`: `lDs`, `lTask`, `lMem`, `lEvent`.

10.4 Self, Sender and Result

Most logiCells metadata scripts depend on three runtime names:

Name	Meaning in common scripts
Self	The current object behind the action or page. From Self , scripts usually reach ObjectsContainer , PageController and the current conceptual object.
Sender	The object that triggered the event. In dataset events, Sender is often the dataset being opened, posted or refreshed.
Result	The explicit return value of a script when the action contract expects one.

A common beginner mistake is to use **Sender** as if it were always a visual component. In dataset events, it is normally a dataset. In visual events, it may be a visual component or its object-behind.

10.5 Accessing the PageController

The **PageController** is the local runtime object that connects the page model, the view data cache and the visual component tree.

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

Listing 10.3: Accessing a page dataset through PageController

Use **PageController** when the script belongs to a page, a view, a component object-behind or a dataset-driven view event. Do not use it as a substitute for business modeling. It is a local coordination object for the current page.

10.6 Reading a Dataset

The **DataSetsManager** gives access to the datasets declared in the page view cache.

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

Listing 10.4: Read the edited task dataset

This code does not query a database directly. It reads the current page data surface. The provider and collection behind the dataset remain outside the visual component.

10.7 Initializing Local Page State

The first task tutorial initializes a memory dataset when the page opens. This is a good beginner pattern because it shows the difference between business data and page state.

```
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;
```

Listing 10.5: Initialize a local filter dataset

`TaskMemoryDs` is not a persistent task table. It stores local filter state for the page. If the same value must survive outside the page, model it as a concept field or persistent facet instead.

10.8 Filtering Before a Dataset Opens

Dataset events allow a view to adjust its data surface before components read it.

```
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;
```

Listing 10.6: Filter a dataset before opening

This script is a view-data rule. It should not contain core business policy. If the rule must be enforced everywhere, move it to a model constraint, action or service boundary.

10.9 Posting the Current Object

A save action usually reads the current dataset, retrieves its current object and posts it.

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

Listing 10.7: Post the current object

Posting data is an intentional runtime operation. A form field changing on screen does not necessarily mean the conceptual object has been persisted or propagated.

10.10 Creating an Object

Class-level or factory actions often create a new object through the object container.

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

Listing 10.8: Create a new Task object

For production metadata, prefer a declared creation action when one exists. Direct object creation in a script is useful for tutorials and internal orchestration, but public workflows should expose stable actions.

10.11 Dispatching a Visual Event

After saving an object, a page may need to notify a parent view that a list should refresh.

```
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);
```

Listing 10.9: Dispatch a parent visual refresh event

Use visual events for visual coordination. Do not use them as a hidden business event bus. If a business fact changed, represent it as a business action, event or process transition.

10.12 Calling a Visual Component Object-Behind

Some scripts need to call the local object-behind of a view component.

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

Listing 10.10: Close the current master view

This is a local UI operation. It belongs in a page action, not in a domain model action intended for services or background execution.

10.13 When to Move Logic Elsewhere

A `functionalP` script should stay small. Move logic elsewhere when the script becomes a hidden application inside the metadata.

If the script...	Move the logic to...
Enforces a business rule everywhere	a model constraint, validation action or process step
Coordinates several objects over time	a workflow or process model
Publishes a stable external capability	a declared action and service contract
Only updates local view state	the page model or dataset event handler
Only refreshes or routes visual components	a visual event or reactive frame pattern

10.14 Beginner Mistakes

Mistake	Correction
Treating <code>functionalP</code> as the model language	Keep MODEL metadata in YAML. Use <code>functionalP</code> only inside declared executable fragments.
Putting business policy in a view event	Put durable policy in concepts, constraints, actions or workflows.
Assuming <code>Sender</code> always has the same type	Check the event family: dataset, visual, process, service or message.
Calling a component from a service action	UI object-behind calls belong to local page actions, not service boundaries.

Mistake	Correction
Forgetting to post data	Set values on the current object, then call <code>PostData()</code> when the change must be committed.
Leaving the return value implicit	Assign <code>Result</code> when the action contract expects a return value.

10.15 Practical Reading Path

For the first project, learn `functionalP` in this order:

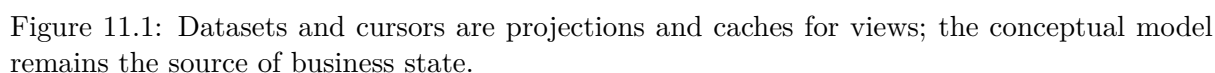
1. read variables and assignment;
2. understand `Self`, `Sender` and `Result`;
3. access `PageControler.DataSetsManager`;
4. read the current dataset object;
5. post data;
6. dispatch a local visual event;
7. create an object only after the load and view paths are clear.

Part III

Client State, Cache and Event Routing

DataSet, DataCursor and DataSource as the View Data Cache

This distinction is important because the view must remain reactive, navigable, and locally coherent even when the underlying business collection is remote, paginated, computed, filtered, or synchronized asynchronously.



The view data cache is the layer that sits between functional components and runtime collections.

Listing 11.1: View data cache chain

This gives logiCells views a strong separation of responsibilities.

41

PageController	Coordinates visual events and local interaction intentions.
PageModel	Coordinates the functional state of the page.
DataSource, DataSet, DataCursor	Expose the local navigable data surface used by the view.
Collection / Provider	Provide the runtime data source, possibly local, persistent, computed, paginated, streamed, or remote.

11.2 Why Functional Components Should Not Own Data Access

A functional component represents an interaction function: selecting, browsing, consulting, editing, validating, navigating, comparing, or displaying. It should not be responsible for deciding how the data is fetched, cached, synchronized, or persisted.

For this reason, a UXfc component should normally bind to a page-level data surface rather than directly to a provider.

```

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

```

Listing 11.2: Typical UXfc bindings to local data surfaces

The component remains functional and declarative. The page decides how the data surface is resolved.

11.3 DataSet, DataSource and DataCursor

The names intentionally echo the data-binding vocabulary used in many native client frameworks. That analogy is useful as a first orientation: a **DataSet** exposes records, a **DataSource** is a binding surface, and a cursor identifies the current navigation context. logiCells extends the pattern through **PageController**, **DataSetsManager**, collections and providers, so the analogy should not be treated as a complete runtime specification.

The three notions play complementary roles.

A **DataSet** represents a local set of records or objects exposed to the view. It may cache a collection, a filtered subset, a query result, or a computed projection.

A **DataSource** is the binding surface used by visual or functional components. It gives components a stable access point without exposing the implementation of the underlying provider.

A **DataCursor** represents the current position in a dataset or collection. It is essential for master/detail views, nested frames, dependent views, and context propagation between components.

Together, they form the local data memory of the 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?
```

Listing 11.3: Three complementary questions

11.4 Relation with Collections and Providers

Collections and providers belong to the runtime data layer. They are responsible for producing, loading, filtering, synchronizing, or persisting objects.

A provider may be local or remote. It may wrap a memory collection, a SQL query, a REST endpoint, a stream, a computed collection, a message source, or a paginated backend. The view does not have to know this directly. It binds to the local data surface exposed by the page.

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

Listing 11.4: Provider abstraction behind a view dataset

This pattern is especially important for distributed applications. The same view can be backed by a local provider during testing, a database provider in a native deployment, or a remote provider in a web deployment.

11.5 Master-Detail and Context Propagation

A common use of the view data cache is master-detail navigation.

```
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
```

Listing 11.5: Master-detail cursor propagation

When the master cursor changes, the dependent dataset can refresh locally. A visual event may be dispatched through the `PageController`, but this does not automatically mean that the business object has changed.

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

Listing 11.6: Local cursor event

The event remains local unless the page model explicitly maps it to an application action or a conceptual model operation.

11.6 Remote Providers and Local Reactivity

When a provider is remote, the distinction becomes even more important. A React component or a native visual component should not call the remote source directly for every small interaction.

Instead, the page exposes a local cache and decides when the provider must be refreshed, synchronized, or queried.

```

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

```

Listing 11.7: Remote provider with local cache

A tab selection, a local row movement, a temporary filter, or an expansion of a panel may only affect the local cache. A validation, a save command, a search request, or a process transition may cross the boundary and reach the provider or the server.

11.7 Declarative Dataset Assembly

A view can declare the datasets, cursors and datasources that form its local cache. This declaration belongs to the view/cache layer. It is not the place where model roles are classified as slots or attributes.

```

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

```

Listing 11.8: Dataset assembly in YAML view syntax

The `kind` attribute is deliberately absent here. It is used only on model field declarations inside facets, where a field is classified as a conceptual role: `kind="slot"` for a structural role, or `kind="attribute"` for a descriptive role. A `DataSet`, `DataSource` or `DataCursor` belongs to the view cache and binding layer, not to the role definition layer.

11.8 Functional Components Bound to the View Cache

Once the data surfaces exist, functional components can be assembled without knowing the provider implementation.

```

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
```

Listing 11.9: Functional components bound to local data surfaces

The tabular view consumes a datasource. The detail view consumes a cursor. The action panel can use the current object from the cursor as an action context. None of these components needs to know whether the underlying collection is local or remote.

11.9 Relation with PageController

The view data cache works together with the `PageController`.

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

Listing 11.10: Event and data mediation in a view

The `PageController` mediates local events. The dataset layer mediates local data. Together they prevent unnecessary coupling between the visual layer and the runtime provider layer.

11.10 Design Rules

A functional component should not own direct data access. It should bind to a named data surface exposed by the page.

A view should distinguish local dataset changes from collection changes. A dataset change may only update the view cache. A collection change belongs to the runtime model layer and may require provider synchronization, persistence, or process notification.

A remote provider should be hidden behind a local data surface. The view should not become a remote API client for every visual interaction.

Datasets, datasources and cursors should be named explicitly in reusable view patterns. Their names are part of the view contract, just like component names and event names.

Chapter 12

DataSetManager Events and Dataset-Driven Views

The previous chapters introduced the view data cache: `DataSet`, `DataSource` and `DataCursor` surfaces exposed by the page. In real applications, this cache is not passive. It emits and routes events when data is opened, navigated, posted, cancelled, filtered, refreshed or synchronized.

This chapter describes the event contract around the `DataSetManager`. In runtime code the object is often named `DataSetsManager`, because it manages several datasets, datasources and cursors at page level. Conceptually, it is the manager of the view data surface.

12.1 Why Dataset Events Matter

A dataset-driven view does not directly manipulate the database, a REST endpoint or a business collection. It manipulates local data surfaces. Those surfaces must notify the page when something relevant happens.

```
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
```

Listing 12.1: Dataset-driven event chain

This event chain is the reason a functional web component can remain reactive without owning the provider. A table can reload when a dataset changes. A detail view can refresh when a cursor moves. A form can validate a dataset before posting. A search box can update a filter dataset and refresh a result dataset. A page can react locally without turning every user gesture into a remote call.

12.2 A DataSetManager Event Is Not Automatically a Business Event

A dataset event first describes the state of the view data surface. It says that a dataset is about to open, has opened, is about to post, has posted, has been cancelled, has moved to another current row, has been refreshed, or has failed to load. This is a view-cache fact, not necessarily a domain fact.

A **DataSetManager** event becomes an application or business event only when the page model, page controller or component controller explicitly escalates it.

This distinction is essential. A row selection is not a business transition. A refresh is not a process event. A field edit is not necessarily a persisted change. A **BeforeDataSetPosted** event is a control point before a possible write. Only the mapped action decides whether the event validates, cancels, synchronizes or escalates the change.

12.3 Event Families

The event contract can be organized into five families.

Family	Meaning	Typical use
Resolution events	A dataset, datasource or cursor is registered or resolved by name.	Component initialization, binding diagnostics, missing datasource handling.
Lifecycle events	A dataset opens, closes, reloads, refreshes or fails to load.	Loading indicators, retry logic, dependent refresh.
Navigation events	The current row or current object changes.	Master-detail propagation, action panels, inspectors, selection-dependent views.
Edit and post events	A row is edited, posted, cancelled, inserted or deleted.	Validation, computed fields, dirty-state tracking, save buttons.
Synchronization events	A dataset exchanges changes with a provider, service or remote collection.	Offline/online coordination, conflict reporting, refresh after provider write.

The exact event names may differ between runtime generations and historical namespaces, but the functional contract remains stable: a page-level data event gives the page a chance to update local components, validate state, or call a higher-level action.

12.4 Events Observed in Dataset-Driven ERP Views

A sales ERP application provides a useful pattern. Its metadata may evolve, but its views clearly show dataset-driven behavior. The views declare datasets and datasources, bind fields and tables to datasources, and use the page **DataSetsManager** to retrieve datasets by name.

A typical form declares an entity datasource for fields and several lookup or list datasources for selectors and tables.

The view does not ask the component to locate the business collection. The component binds to the datasource. The page data layer resolves the dataset and its provider.

12.5 Before Posting: Validation as a Dataset Event

The most explicit dataset event pattern is validation before a dataset is posted.

The action receives the event sender and the mutable event parameters. It can retrieve the dataset from the page-level manager, inspect the current data, and decide whether the post is allowed.

```

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;
```

Listing 12.2: Retrieving the dataset in a before-post action

The event is not the validation rule itself. It is the hook that gives the page model or object behind the opportunity to call a validation action. The action may remain local, call an object method, or escalate to a service if the validation requires remote knowledge.

12.6 After Posting: Refreshing Dependent Views

A second common pattern is the after-post event. Once a dataset has successfully posted, the page can refresh dependent datasets, update action availability, rebuild a dashboard tile, or move the cursor to the newly created record.

```
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;
```

Listing 12.3: Refreshing a dependent list after a post

This kind of logic should remain page-scoped when it only maintains view coherence. It becomes an application event only when the post has domain significance and must notify another process, agent or service.

12.7 Filter Datasets and Reactive Searches

ERP views often use a small filter dataset to drive a larger result dataset. Search fields bind to the filter datasource. When the user changes a filter field or clicks a search action, the page reads the filter dataset and applies provider filters to the result dataset.

```
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;
```

Listing 12.4: Applying filters through the DataSetsManager

For a web reactive component, this pattern maps naturally to debouncing. The component may keep local input state while the user types. Only after a delay or explicit search request does it update the filter dataset and trigger the result refresh.

12.8 Cursor Changes and Master-Detail Views

A cursor event is a dataset event about context. It says that the current row, current object or current selection has changed. This is the basis for master-detail views.

```
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
```

Listing 12.5: Cursor-driven master-detail chain

Typical dependent components include object inspectors, action lists, relation tables, totals, document lines, contextual buttons and secondary panels. The current object carried by the cursor is the context passed to those components.

12.9 Event Parameters

Dataset events should carry enough information to let the page react without global lookup. A practical event payload contains the event class, the source dataset, the sender, the datasource or cursor name, and optional mutable parameters.

Parameter	Purpose
Sender	Runtime object that emitted or forwarded the event. Often gives access to <code>DataSetsManager</code> .
DataSet	Dataset concerned by the event. Useful when the handler is generic.
DataSetName	Stable name used to resolve the dataset from the manager.
DataSourceName	Binding surface used by visual components.
CursorName	Cursor whose current object or position changed.
CanPost	Mutable boolean used by before-post validation.
Message	Validation or diagnostic message returned to the page.
StopNotification	Propagation control flag when an event is consumed locally.

The important design rule is to pass context explicitly. A page handler should not have to infer the active dataset from unrelated visual state when the event already knows which dataset changed.

12.10 Web Reactive Mapping

In a web view, the same contract can be represented as a page-level event subscription. The component binds to a datasource, subscribes to the relevant dataset notifications, and updates its local React state when the dataset changes.

```
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 }

```

Listing 12.6: React-style subscription to dataset notifications

A component may also subscribe to cursor changes.

```

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

```

Listing 12.7: React-style cursor notification

This is a web implementation form of the same logiCells rule: the component listens to the page data surface, not to the provider directly.

12.11 Recommended Event Names

A public model can document dataset events by function rather than by implementation class. The following naming catalog is sufficient for most view designs.

Event	Meaning
BeforeDataSetOpen	The dataset is about to open or resolve its provider.
AfterDataSetOpen	The dataset is available for binding and rendering.
BeforeDataSetRefresh	A refresh is about to be executed.
AfterDataSetRefresh	A refresh has completed and dependent components may update.
DataSetLoadFailed	The dataset could not load from its provider.
BeforeDataSetPosted	A post is about to occur and may still be cancelled.
AfterDataSetPosted	The post has succeeded; dependent surfaces may refresh.
AfterDataSetCanceled	Local editing was cancelled.
CurrentObjectChanged	The cursor now points to another object or row.
FieldChanged	A field value changed in the view cache.
ProviderSyncStarted	Synchronization with a provider has started.
ProviderSyncCompleted	Synchronization with a provider has completed.
ProviderSyncFailed	Synchronization failed and the page should expose diagnostics.

These names are intentionally functional. An implementation may wrap them in a run-time namespace such as `logiCells.Visual.Data.*` or `lgcVisual.Data.*`. The model should preserve the semantic distinction: lifecycle, navigation, edit/post and synchronization.

12.12 Declarative Event Assembly

The dataset event contract can be expressed at view level. The declaration says which data event the page listens to, which action should be called, and which parameters are passed.

```

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

```

Listing 12.8: Dataset event assembly in YAML view syntax

12.13 Design Rules

Dataset-driven views are powerful, but they need clear boundaries.

- A component binds to a datasource; it should not resolve providers directly.
- A page action retrieves datasets through **DataSetsManager**, not through visual component internals.
- A dataset lifecycle event updates view coherence first; it escalates only by explicit mapping.
- A before-post event is the right place for page-level validation and cancellation.
- An after-post event is the right place for dependent refresh and local consistency updates.
- Cursor events should drive master-detail context, action availability and inspectors.
- Provider synchronization events should be separated from local row navigation events.
- The **kind** attribute is never used on **DataSet**, **DataSource** or **DataCursor**; it belongs only to model **field** declarations inside facets.

The result is a view architecture that is reactive, local-first and provider-independent. Components react to dataset notifications, page models decide what they mean, and business actions are invoked only when the view explicitly crosses the boundary from data-surface coherence to application intention.

Chapter 13

PageController and Local Visual Event Dispatching

logiCells views are reactive, but not every reaction is a business event and not every visual event should cross a runtime boundary. The view layer has its own coordination space. In that space, the `PageController`, the `PageModel` and the component `objectBehind` objects form a local dispatching architecture for visual events.

This chapter makes that architecture explicit. It complements the UXfc chapters by showing how intentional and functional components are assembled into a running page without turning each click, tab change, selection change or dataset notification into a server call or a mutation of the conceptual object container.

13.1 Why Local Visual Dispatch Exists

A view contains many interactions that are meaningful for the interface but not necessarily meaningful for the domain model. Examples include selecting a tab, expanding a panel, changing the current row of a client-side dataset, updating a status area, showing a secondary frame, reloading a consultation component, or synchronizing a local selection control.

These interactions should first be interpreted locally by the page. They may later be escalated to the application model, a process, a published action, a service call or the conceptual memory, but that escalation is a design choice. It is not automatic.

```
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
```

Listing 13.1: Local visual event principle

The important rule is simple: a visual event is not automatically a business event; it is not automatically a server call; and it is not automatically a container mutation.

13.2 PageController as the Local Event Bus

A logiCells page has a general controller, traditionally named `PageController`. It receives events addressed to the view and dispatches them to the internal components of the page. It is the local event bus of the rendered view.

The `PageController` is responsible for several local concerns:

- receiving visual events emitted by functional components;

- identifying the source and sender of an event;
- dispatching the event to the `PageModel`;
- dispatching the event to component models or `objectBehind` objects;
- stopping propagation when a local reaction consumes the event;
- escalating the event only when the view model declares that such escalation is required.

This architecture is especially important for views composed from UXfc components. The UXfc component declares the interaction intention. The functional component declares the interaction function. The `PageController` coordinates the event flow between concrete components and the local page model.

13.3 PageModel and Component objectBehind

The `PageModel` is the model of the page itself. It can be read as the highest-level local frame that contains and coordinates the functional UI components of the page.

Each component may also have its own `objectBehind`, sometimes simply called a component model. The component `objectBehind` interprets events for one functional area of the view: a tabbed view, a split view, a consultation frame, an autocomplete selector, a navigation panel, a tabular view or another functional component.

```

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

```

Listing 13.2: Local view coordination roles

This distinction prevents the object container from becoming the receiver of every visual notification. It also avoids turning the server into the default coordinator for purely local UI behavior.

13.4 React Views and Client-Side Dispatch

In a React-based view, the same principle applies. A concrete React component may emit a local event to a page controller object. The event can update local state, select a tab, refresh a local dataset, notify a sibling component or change a consultation context without immediately calling the server.

```

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

```

Listing 13.3: React-style local dispatch

This dispatch is local to the view unless the page model maps it to an action, a process event, a published service or a remote synchronization rule.

13.5 Native Views and DispatchVisualEvent

In a native view, the same logic appears through a controller method such as `DispatchVisualEvent`. A dataset manager, navigation panel, frame or visual component can build a visual event and submit it to the page controller.

```

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;
```

Listing 13.4: Native-style visual dispatch

The page controller can then notify the `PageModel` and the relevant component `objectBehind` instances. If one of these handlers consumes the event, propagation may stop. If the event is mapped to a higher-level action, the view can explicitly escalate it.

13.6 Four Levels of Events

A clear event taxonomy avoids coupling and confusion.

Level	Meaning	Typical receiver
Local visual event	A UI coordination signal: tab selected, sequence changed, panel opened, field focused, dataset cursor moved.	<code>PageController</code> , <code>PageModel</code> , component <code>objectBehind</code> .
UXfc intention event	A semantic interaction signal: <code>Browse(uri)</code> , <code>Reload</code> , <code>SelectionChanged</code> , <code>ValidationRequested</code> .	UXfc controller, functional component controller, page model.
Application event	A coordination signal between views, controllers or processes: item opened, context changed, review requested.	Page/session controller, process controller, application model.
Business or process event	A domain-level fact: order validated, process completed, message received, work item assigned.	Objects container, process object, agents, services, message handlers.

The same runtime may route all these events, but the model should keep their intent distinct.

13.7 Declarative Event Mapping

The page model can declare how local visual events are mapped to local actions, page-level reactions or higher-level operations.

The attribute `local` is used here as a documentary marker: it distinguishes view-local reactions from events that intentionally cross the boundary toward an application action or domain

process. The concrete runtime may encode this distinction through event classes, dispatch methods, controller scope, action paths or propagation flags.

13.8 DataSets, Collections and Runtime Boundaries

A practical rule follows from the logiCells MVC pattern: datasets are view-side caches, whereas collections are model-side conceptual data. A dataset notification often belongs to the visual side of the application. A collection change belongs to the conceptual model side.

```

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

```

Listing 13.5: Dataset versus collection event boundary

This distinction is central for web architectures. In a React view, a dataset may live in the browser-side view state, while the corresponding collection remains server-side or runtime-side. Dispatching a dataset event should not automatically imply a server call.

13.9 UXfc Assemblies and Local Events

UXfc assemblies become more precise when their event contracts identify the local coordination layer.

```

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

```

Listing 13.6: UXfc assembly with local event dispatching

The first events coordinate the local page. The last event intentionally crosses the boundary because it represents a domain-relevant validation request.

13.10 Design Rules

Keep visual event names in a visual namespace when their primary meaning is local UI coordination. Use domain namespaces only when the event expresses a business or process fact.

Route visual events first through the **PageController**. Let the **PageModel** and component **objectBehind** objects handle local reactions. Escalate only when the page model declares a link to an action, service, process, message or conceptual model change.

For React or web views, avoid turning every local event into a server request. For native views, avoid using the object container as the default receiver of every component notification.

Document event boundaries explicitly in reusable view patterns. The reader should be able to see which events are local to the page, which events coordinate the application, and which events alter the conceptual model.

Chapter 14

Event-Driven and Asynchronous VIEW Patterns

Modern logiCells VIEW controllers are usually asynchronous by *protocol*, not by language syntax. The common pattern is event-driven orchestration using visual events, dataset lifecycle events, post/submit continuations, waiting states and explicit refresh notifications. Controllers should remain thin and client-oriented; domain behavior belongs in reusable MODEL actions and processes.

14.1 Think in continuations, not in async/await

The preferred causal chain is:

```
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
```

Listing 14.1: Typical asynchronous view flow

Do not assume that calling `Refresh()` makes result data synchronously available to the following lines. If subsequent logic depends on loaded data, continue from the relevant lifecycle event such as `AfterDataSetOpened`.

14.2 Controller event subscription

```
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'']'

```

Listing 14.2: Controller subscribes to visual and dataset events

14.3 Dispatch intent, not parent implementation

A child view should not reach into its parent and refresh concrete widgets one by one. It should publish an intention such as `erp.Visual.View.Refresh`. The parent or an event root decides which descendants must react.

```

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.ParentPageController.CreateVisualEvent(
15             'erp.Visual.View.Refresh', Self, Self);
16           Self.ParentPageController.DispatchVisualEvent(Event, StopNotification);
17           Self.PageController.CloseForm();

```

Listing 14.3: Child completion notifies its parent

This keeps popup, child and parent views independently evolvable.

14.4 Waiting frames and completion signals

When a visible operation spans several event phases, mark the request as waiting and end the wait through a completion event according to the visual framework contract.

```

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

```

Listing 14.4: Requesting a post operation with a waiting frame

The continuation belongs to the post lifecycle event, not to the original button handler.

```

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

```

Listing 14.5: Recommended state progression

14.5 Dataset lifecycle pattern

Use `BeforeDataSetOpened` to prepare filters/context, and `AfterDataSetOpened` for work that requires the loaded dataset.

```
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
```

Listing 14.6: Dataset-driven continuation

14.6 Event roots and propagation

Frames can form event-routing boundaries. A root can receive a high-level event and notify its children. `StopNotification` is part of the protocol and should be preserved instead of replacing event propagation with direct component calls.

Use event names as stable intentions and put small contextual values on the event when one event family can serve several cases. Avoid creating a unique event name for every incidental button.

14.7 Keep VIEW controllers thin

MODEL-behind code may coordinate visual state, datasets, cursors, navigation and event dispatch. It should not become the new home for hundreds of lines of domain calculation.

Prefer:

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

Listing 14.7: Thin VIEW controller boundary

Business invariants, idempotence, cross-object orchestration and durable state transitions belong in MODEL/actions/processes so that service, agent, batch and VIEW callers reuse the same behavior.

14.8 Asynchronous VIEW review checklist

For each non-trivial view ask:

- Is the user action represented as an event or reusable action rather than a chain of widget calls?
- Does code that depends on dataset results resume from a dataset lifecycle event?
- Is a long visible operation represented by a waiting/completion protocol?
- Does a child communicate with its parent through an intention event?
- Can refresh be scoped to the affected view/subtree rather than the entire application?

- Are domain calculations outside the VIEW controller?
- Are duplicate events and retries safe or explicitly guarded?

Chapter 15

Reactive Desktop Frames and Event Routing

Advanced logiCells applications do not only display forms, lists and fields. They coordinate several views at the same time: a navigation area, a working area, preview panels, dashboards, inspectors, action menus, notification zones and workflow steps. In this kind of application, the user interface must react locally to selections, dataset changes, preview refreshes and workflow events without rebuilding the whole page or sending every gesture to a remote provider.

The `reactiveDesktopFrame` pattern answers this need. It turns a region of the desktop into a reactive host. The frame can receive events, load or refresh child views, propagate notifications to children, react to dataset changes and coordinate parent-child communication through the `PageController`. This chapter focuses on the web and published interface side. Native implementations are outside the scope of this book.

15.1 From Dataset-Driven Forms to Reactive Desktops

The previous chapter described dataset-driven views. That pattern is already powerful for forms and grids: a page declares datasets, datasources and cursors, components bind to datasources, and dataset events validate or refresh the page. However, advanced interfaces need a higher-level coordination pattern.

A dataset-driven form asks questions such as:

```
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?
```

Listing 15.1: Questions answered by dataset-driven forms

A reactive desktop asks broader questions:

```
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?
```

Listing 15.2: Questions answered by a reactive desktop frame

This is why a `reactiveDesktopFrame` must not be described as a simple layout widget. It is a local reactive boundary inside the desktop.

15.2 What a Reactive Desktop Frame Is

A `reactiveDesktopFrame` is a visual frame with runtime behavior. It is a host for dynamic content, but it is also an event participant. It has a place in the view tree, a runtime visual component, an optional object behind, and a relationship with the page controller.

```

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

```

Listing 15.3: Conceptual structure of a reactive desktop frame

The frame can therefore play several roles.

Role	Meaning
Host	It contains a child view, a set of child views, or a dynamic view tree.
Context boundary	It receives a cursor or a current object and shares that context with its children.
Event sink	It declares local or application events and maps them to actions.
Event router	It forwards a visual event to children, to a parent controller, or to a local action.
Dataset-aware coordinator	It reacts to dataset lifecycle events and updates visual state accordingly.
Preview surface	It refreshes visual output when metadata, text or model fragments change.
Workflow shell	It hosts a sequence of steps and moves between them through events.

A classical desktop container is mainly spatial. A reactive desktop frame is spatial and behavioral.

15.3 The Runtime Participants

The pattern involves several runtime participants. Keeping them distinct avoids many design errors.

Participant	Responsibility
<code>reactiveDesktopFrame</code>	Declares and hosts a reactive region in the view tree.
Runtime visual component	Concrete component instance that can be found, invalidated, refreshed, moved or notified.
Frame object behind	Conceptual controller associated with the frame. It stores local state and actions.
<code>PageController</code>	Local bus and resolver for visual components, events, datasets and page actions.
<code>ParentPageController</code>	Parent bus used when a child page or embedded view must notify its host.

DataSetsManager	Registry of page datasets, datasources and cursors; source of dataset events.
Child views	Embedded views hosted inside the frame. They may receive notifications from the frame.
Component controllers	Web or page controllers that map user gestures to functional events.

A frequent implementation pattern is to retrieve the frame object behind by name, then resolve the runtime component by id.

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

Listing 15.4: Finding a reactive frame component

The first call works at the object-behind level. The second call moves from the object behind to the runtime visual component. The distinction is important: the object behind holds conceptual state and actions; the visual component hosts children and performs visual operations.

15.4 Host Pattern: Dynamic View Region

The first pattern is the host pattern. A reactive frame defines a region in which content can be loaded, replaced or refreshed.

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

Listing 15.5: Reactive frame as dynamic region

The frame can display an object view, a list view, a preview, a dashboard, a workflow step or another composed layout. The rest of the desktop remains stable while the frame changes.

This is different from route-based web development. The whole browser route does not have to change. A local application event can ask a frame to show a new view in a named region.

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

Listing 15.6: Application event routed to a frame

The `PageController` receives or creates the event, resolves the target frame, and asks the frame to load or activate the requested child view. The component that emitted the event does not need to know how the frame is rendered.

15.5 Context Pattern: Input Cursor and Current Object

A reactive frame often receives an input cursor. The cursor defines the current object or current row that the frame and its children must use.

The frame does not own the business object. It receives a context. This distinction allows the same frame implementation to host different views depending on the current cursor. A

master list can change the current object; the reactive frame can then refresh the detail area, action area and inspector area.

```

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

```

Listing 15.7: Cursor-driven reactive region

The context pattern is particularly important for advanced dashboards. The same frame may host several child panels that all depend on the same current object: a summary, a history, an action list, a graph and a preview.

15.6 Application Event Sink Pattern

A reactive frame can declare events in its local page model. This turns the frame into an application event sink.

The event class should be stable and functional. It should describe what the application wants, not how a widget is implemented. For example, `Application.Shell.ShowView` is preferable to a low-level event such as `Button42Clicked`. The action mapped to the event decides how the frame reacts.

```

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;

```

Listing 15.8: Application event handling inside the frame

This action is page-local. It may load a view, select a tab, replace a child frame, or ask a component controller to rebuild a region. It does not have to cross a server boundary.

15.7 Child Notification Router Pattern

The second major pattern is downward notification. A reactive frame receives an event and sends a visual notification to its children.

```

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

```

Listing 15.9: Downward notification through a reactive frame

This pattern is useful when several child views must refresh after a shared condition changes. The parent frame does not have to know each child component type. It emits a semantic visual notification and lets children handle it if relevant.

```

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;

```

Listing 15.10: Notifying children from a reactive frame

The `StopNotification` parameter gives a child the possibility to consume the event. This is useful when the frame contains nested reactive frames. A child frame may handle a notification and stop propagation to avoid duplicate refreshes.

15.8 Child-to-Parent Bubbling Pattern

The complementary pattern is upward notification. A child view emits an event on the parent page controller. This is useful when an embedded view must ask the parent desktop to open, select, navigate or refresh something.

```

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

```

Listing 15.11: Child-to-parent event bubbling

A typical implementation creates a visual event on the parent page controller and attaches the child page controller to the event.

```

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;

```

Listing 15.12: Dispatching a visual event to the parent PageController

The child does not decide how the parent shell reacts. It only reports a meaningful visual interaction: an item was double-clicked. The parent can then open the object, change the active child view, load a preview, or update a workflow state.

15.9 Dataset-Aware Reaction Pattern

A reactive frame is especially useful when visual state depends on dataset state. A dataset event such as `AfterOpenDataSet`, `AfterRefresh` or `CurrentChanged` can update the frame and its children.

```

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

```

Listing 15.13: Dataset event driving frame state

This pattern is often used for dashboards and status-driven interfaces. A dataset may contain computed state: counts, flags, validation status, current phase, missing inputs or authorization results. The frame reads this state and updates the 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;

```

Listing 15.14: Dataset event handler updating visual state

This handler is not a business rule. It is a view-state rule. The dataset already exposes the computed state. The frame maps that state to interface behavior.

15.10 Refresh-and-Recompute Pattern

A frequent advanced pattern combines child notification and dataset refresh.

```

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

```

Listing 15.15: Refresh and recompute chain

This avoids a full page reload. Only the concerned child views and datasets are refreshed. The frame remains the local coordinator.

```

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;

```

Listing 15.16: Refresh and recompute action

This is one of the most important patterns for evolved interfaces. It gives the user the feeling of a live application while preserving a clear conceptual structure.

15.11 Preview Surface Pattern

A reactive frame can host a live preview. In a metadata designer, report designer, document editor or workflow designer, a change in a source component can refresh a preview frame.

```

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

```

Listing 15.17: Live preview event chain

This pattern should remain local when the preview can be rebuilt from page state. It should only call a service when the preview needs remote rendering, permissions or large computation.

```

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;

```

Listing 15.18: Preview refresh handler

For web publication, the same pattern maps to a React component that subscribes to a page event and updates local state or requests fresh generated markup from the page model.

15.12 Workflow Shell Pattern

A reactive frame can also host a workflow. The frame contains the current step, while the controller moves between steps.

```

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

```

Listing 15.19: Workflow shell with reactive frame

The frame may expose operations such as `MoveTo`, `LoadMainFrame`, `Invalidate`, `Refresh`, or `UpdateConnectionStatus`. The names may vary across implementations, but the functional pattern is stable: the frame hosts the current step and coordinates navigation through local events.

The workflow shell pattern is useful when building onboarding flows, guided configuration, document creation assistants, validation processes, or mobile-like step-by-step applications.

15.13 Event Taxonomy for Reactive Frames

Events around reactive frames can be organized into six categories.

Category	Meaning	Typical handler
Visual interaction	A user clicks, double-clicks, selects, expands or collapses a visual element.	Component controller or frame object behind.
Frame command	A parent or child asks a frame to show, move, load, refresh or invalidate a region.	Reactive frame action.
Dataset event	A dataset opens, refreshes, posts, cancels or changes current row.	Page model or frame object behind.
Child notification	A frame sends a semantic notification to children.	Child view controllers.
Parent notification	A child dispatches an event to the parent <code>PageController</code> .	Parent frame or shell controller.
Application intention	An application-level event requests a high-level change, such as showing a view or opening an object.	Page shell or application frame.

The category determines the correct escalation level. A click should often remain visual. A dataset refresh should often remain local. An application intention may cross page boundaries. A business transition should be expressed as an application or process action, not as a raw frame event.

15.14 Recommended Event Naming

Reactive desktop events should use names that communicate their level.

Example	Meaning	Level
<code>lgc.Visual.View.Refresh</code>	Refresh a visual view or its children.	Visual
<code>lgc.Visual.ViewItem.Notify.DoubleClick</code>	A view item was double-clicked and notifies its parent.	Visual
<code>lgc.Visual.Data.AfterOpenDataSet</code>	A dataset has opened and visual state may update.	Data/view

Application.Shell.ShowView	The application shell should show a named view.	Application
Application.Workflow.MoveNext	A workflow shell should move to the next step.	Application
Business.Document.Validated	A domain event occurred.	Business

Avoid encoding component ids into event class names. The target frame, source component or dataset name should be carried as parameters when needed.

15.15 Declarative Assembly

The following simplified example shows a reactive frame that hosts a dynamic detail region. It has an input cursor, receives an application event, reacts to dataset events and can notify children.

Notice that the datasets and datasources do not use `kind`. The `kind="slot|attribute"` classification belongs to model field declarations inside conceptual facets. View data surfaces such as `DataSet`, `DataSource`, `DataCursor` and `reactiveDesktopFrame` do not use it.

15.16 Web Reactive Mapping

In the web runtime, the same pattern can be expressed with component registration and local subscriptions.

```

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

```

Listing 15.20: Web reactive frame registration pattern

The React component renders the frame and children. The controller subscribes to page events, resolves datasets through the page controller, and delegates rendering changes to React state.

```

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 }

```

Listing 15.21: Reactive frame controller responsibilities

The controller does not bypass the `DataSetManager`. It uses the page data surfaces already declared by metadata. This keeps the web component aligned with the conceptual runtime.

15.17 Local State Versus Model State

A reactive desktop frame has local visual state. For example, it may store the active child view, an expanded panel, the current tab, a loading flag or a preview refresh counter. This local state must not be confused with model state.

State type	Examples
Local visual state	Active child view, collapsed panel, selected tab, loading flag, scroll position.
View data state	Current dataset row, datasource contents, cursor position, filter dataset values.
Model state	Business object fields, relations, workflow status, persisted process facts.
Application state	Current workspace, current module, authentication state, shell-level command state.

A frame event may update local visual state immediately. It should update model state only through a declared action, dataset post or service boundary. This rule is essential for traceability.

15.18 Avoiding Event Loops

Reactive interfaces are prone to accidental loops. A dataset refresh may trigger a frame refresh; the frame refresh may notify children; a child may refresh the same dataset again. The pattern must therefore define loop guards.

Practical loop guards include:

- 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.

Listing 15.22: Loop prevention rules

A good reactive desktop is not simply event-rich. It is event-disciplined.

15.19 Error Handling and Diagnostics

A reactive frame should fail visibly and locally when its dependencies are missing. Missing datasets, unresolved child views or unknown event classes should not silently do nothing.

```
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');
```

Listing 15.23: Diagnostics for reactive frame handlers

For web components, diagnostics should include the component name, functional component name, implementation class, datasource name, dataset name and event class. This makes generated interfaces much easier to debug.

15.20 Design Rules

The following rules summarize the recommended design discipline.

1. Use `reactiveDesktopFrame` for regions that host dynamic or event-driven child views.
2. Keep raw user gestures local unless they express a functional or application intention.
3. Route child-to-parent communication through the parent `PageController`, not through direct component references.
4. Use `NotifyChildren` for downward semantic notifications.
5. Use dataset events to update view state, not to encode business rules directly.
6. Escalate to application or business actions explicitly when a change has domain meaning.
7. Carry target frame, dataset and view names as parameters rather than encoding them into event class names.
8. Protect refresh chains against loops and duplicate notifications.
9. Keep `kind="slot|attribute"` restricted to conceptual model fields inside facets.
10. Treat native implementations as a separate subject; the public web pattern is metadata, `PageController`, data surfaces, event routing and reactive component state.

15.21 Why This Pattern Matters

Without a reactive desktop frame, complex interfaces tend to become tightly coupled. A button knows which tab to change. A child table knows which parent panel to refresh. A dashboard widget knows which service to call. A preview component knows too much about the editor that feeds it.

With a reactive desktop frame, the interface becomes declarative and coordinated:

```
1 components emit semantic events
2   -> PageController 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
```

Listing 15.24: Reactive desktop coordination

This is the basis for advanced logiCells interfaces. The frame is not only a place where components are displayed. It is a local orchestration boundary where visual composition, dataset state and application intention meet.

Part IV

Native and Web Rich Clients

Chapter 16

Native Rich-Client Hosts

The logiCells View architecture is not web-specific. The same client model – View projection, DataSet/DataCursor cache, page behavior, event routing and action boundaries – can be hosted by a native client when the selected runtime distribution provides the corresponding visual implementation registry.

16.1 The Stable Contract

A native client should preserve the same separation used throughout this 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
```

Listing 16.1: Native rich-client separation

The functional component describes what the interaction is for; the native implementation decides how that intent is rendered by the selected UI toolkit. Business actions remain in the conceptual model rather than moving into widget callbacks.

16.2 Cache and Asynchrony

Native clients use the same client-state rules as web rich clients. A DataSet can hold a local projection, a DataCursor tracks position/current object, and visual events can remain local until an application or business action must be invoked. Long-running or remote operations should return control to the UI and continue through the event/data lifecycle documented in the asynchronous chapters.

This is the reason to choose the View architecture for a native client: not because a native UI requires MVC, but because the application benefits from an explicit local cache, stable interaction state and asynchronous coordination with the runtime.

16.3 Platform and Toolkit Boundary

Runtime availability on Windows, Linux and macOS does not imply that every native component library or visual registry has identical coverage on every host. Treat the native UI toolkit as a deployment-specific adapter and verify the implementation set distributed for the target platform.

The portable contract in this guide is therefore the View/DataSet/event architecture. A concrete native widget class is not part of the conceptual programming model and is intentionally not used as public vocabulary here.

16.4 When to Prefer a Custom MVP Client

A native application that only needs a small number of forms or direct action calls may be simpler as an MVP/custom client over the runtime API or ABI. Choose the logiCells View architecture when its cache, component-intent, event-routing and synchronization mechanisms solve a real client problem.

Chapter 17

Building Web Reactive Functional Components

This book treats web functional components as first-class incarnations of logiCells UXfc components. Native visual components belong to another implementation guide. The purpose of this chapter is therefore limited and precise: it explains how a functional component can be implemented in a web runtime while preserving the logiCells separation between intention, data cache, local events, and remote boundaries.

A web functional component is not just a React widget. It is the web incarnation of a declared interaction function. It receives data through a page-level data surface, reacts locally through a page controller, and crosses the server or provider boundary only when the page model explicitly asks it to do so.

17.1 From UXfc Function to Web Component

A logiCells UXfc component starts as an interaction function: consultation, selection, exploration, edition, validation, navigation, tabular display, status display or another reusable function. In web mode, this function is implemented by a reactive component, typically through a registered JavaScript or TypeScript component.

```
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
```

Listing 17.1: From functional declaration to web rendering

The important point is that the web component implements a functional contract. It does not redefine the model, own the business collection, or decide by itself when a visual action becomes a domain action.

17.2 Reactive Component Contract

A web functional component should have an explicit contract. This contract is more stable than its visual implementation.

Contract element	Meaning
------------------	---------

Function name	Logical UXfc function implemented by the component, such as <code>tabularView</code> , <code>autoCompleteSelect</code> , <code>consultation</code> or <code>statusView</code> .
Inputs	Data surfaces, cursor names, resources, options, context identifiers and action descriptors received from the page.
Local state	Purely client-side state such as selected tab, open panel, temporary filter, pending text, hover state or local loading flag.
Events	Visual or UXfc events emitted to the <code>PageController</code> .
Escalation rules	Conditions under which the page maps an event to a provider refresh, service call, action, process or conceptual model change.

This contract allows the same functional component to evolve visually without changing the conceptual view declaration.

17.3 Reactive Props and View Data Surfaces

In web mode, reactivity should normally be driven by page-level data surfaces. The component receives a view model fragment or a binding handle derived from a `DataSource`, `DataSet` or `DataCursor`. It should not directly reach into a remote provider for every small interaction.

```

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

```

Listing 17.2: Reactive data flow for a web component

When the `DataSet` changes, the page state changes. The reactive runtime then re-renders the component with new props. This is the web equivalent of a local view cache notification.

17.4 Local State is Not Model State

A reactive component often keeps local state. This is normal, but the state must be classified correctly.

```

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

```

Listing 17.3: Typical local web state

These values belong to the local life of the web component or page. They are not automatically model attributes, collection changes, server updates or business facts. A local state update may trigger a visual event, but that event remains local until the `PageModel` maps it to a higher-level operation.

17.5 Dispatching Local Visual Events

A web component should communicate with the page through the page controller. This keeps the component reusable and prevents hard coupling with services, providers or business actions.

```

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 }

```

Listing 17.4: Web component local dispatch

The page controller decides whether the event is consumed locally, forwarded to the page model, sent to another functional component, or escalated to an action or provider.

17.6 When an Event Crosses the Boundary

The server boundary must be explicit. A local selection, local tab change or local filter change should not automatically become a remote call. Conversely, a search command, save command, validation request or refresh command may intentionally cross the boundary.

Event	Default scope	Possible escalation
TabSelected	Local page	None, or local dataset refresh.
CurrentRowChanged	Local page cache	Optional detail dataset refresh.
FilterTextChanged	Local component	Optional debounced lookup request.
SearchRequested	Page action	Provider query or service call.
SaveRequested	Application action	Published action, provider update or process transition.
ValidationRequested	Domain action	Business process or conceptual event.

The design rule is simple: local reactivity is the default; escalation is declared.

17.7 Subscription and Refresh Model

A reactive web component should normally subscribe to page data, not to the provider directly. The page model or data cache decides when a provider must be refreshed.

```

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

```

Listing 17.5: Recommended subscription model

This keeps provider protocols replaceable. A local mock provider, a paginated REST provider, a computed provider and a synchronized remote provider can all feed the same reactive component through the same page data surface.

17.8 Debouncing, Batching and Idempotence

Web reactivity can generate many small events. Text input, resizing, scrolling, hover, filter typing and autocomplete can emit frequent notifications. logiCells components should not treat all these notifications as business events.

```

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

```

Listing 17.6: Debounced lookup pattern

For remote interactions, commands should be designed to be idempotent when possible. A repeated refresh or lookup should not corrupt the model. A save, validation or process transition should use an explicit action contract and return a clear result event.

17.9 Reactive Functional Components and Actions

A functional component may expose visual actions such as reload, select, open, validate, cancel or submit. In web mode, these actions are usually emitted as events first. The page model then maps the event to a local reaction or to a published action.

```

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

```

Listing 17.7: Mapping web visual action to a published action

This preserves the difference between a user gesture, a UXfc intention, an application action and a business result.

17.10 Component Registration for Web Mode

Concrete web components should be registered against functional component names. The registration is a platform binding, not a change in the conceptual model.

```

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

```

Listing 17.8: Web functional component registration

The same conceptual view can later use another implementation for another web framework or another product shell, while keeping the functional declaration stable.

17.11 Declarative Web Component Assembly

A web page can declare its functional components, data surfaces and local event contracts.

```

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

```

Listing 17.9: Reactive web page assembly in YAML model syntax

The dataset fields shown in this view declaration do not classify conceptual roles. Therefore they do not carry kind. The kind="slot" or kind="attribute" attribute belongs to model field declarations inside facets.

17.12 Equivalent XML View Assembly

This example is a view/cache declaration. It is intentionally different from a model facet declaration where fields may be marked with kind="slot" or kind="attribute".

17.13 Minimal Reactive Implementation Sketch

The following sketch shows how a concrete web implementation can stay inside the logiCells boundaries.

```

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 }
```

Listing 17.10: Minimal reactive component sketch

The component receives data and a cursor from the page. It emits a local event when the current row changes. It does not update the provider directly and does not call a business service by itself.

17.14 Error, Loading and Offline States

A web component should expose functional state without hiding the provider model. Loading, stale data, offline state, synchronization delay and provider errors should be represented as page-level or datasource-level states.

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

Listing 17.11: Functional states for reactive web components

A `statusView` can subscribe to these states and render the appropriate feedback. A `tabularView` can keep showing cached rows while the provider refreshes. An `autoCompleteSelect` can display local suggestions first and then merge remote results when the provider responds.

17.15 Design Rules

Implement the UXfc contract before implementing visual appearance. Bind to `DataSource`, `DataSet` and `DataCursor` rather than directly to providers. Dispatch local events through the `PageController`. Keep local state local. Use the `PageModel` to map selected events to provider refreshes, published actions, services or processes. Make remote boundaries explicit. Treat native component implementation as a separate concern.

Chapter 18

Modern React Syntax for logiCells Components

The logiCells web chapters focus on how reactive components bind to declarative views, data sources and page events. They assume a small amount of modern JavaScript and React syntax. This chapter explains only the forms used by the platform examples; it is not a complete React course.

18.1 Functional Components

New examples should use functional components rather than class components unless compatibility with an existing component requires otherwise.

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

Listing 18.1: Functional React component

18.2 Props Destructuring

This form:

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

Listing 18.2: Destructuring component props

is JavaScript destructuring. It is equivalent to reading `props.caption` and `props.task`. logiCells reactive props may carry data structures, controllers, data-source names or view attributes; the braces are JavaScript syntax, not a logiCells metadata declaration.

18.3 Destructuring Metadata Attributes

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

Listing 18.3: Reading generated view metadata

This extracts two properties from the generated metadata object. Always handle missing optional attributes when a component can be used in several view shapes.

18.4 Local State and Hooks

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

Listing 18.4: Local React state

`searchText` is local rendering state. It is not automatically model state and it is not automatically written to a logiCells `DataSet`. Synchronize it deliberately through a `DataSource`, action or event when the value belongs to the model.

18.5 Effects and Subscriptions

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

Listing 18.5: Subscribing and cleaning up

The cleanup function prevents duplicate subscriptions when a component is unmounted or recreated.

18.6 Event Handlers

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

Listing 18.6: Dispatching a local visual event

The event name and payload belong to the logiCells view contract. The function declaration and object literal are JavaScript.

18.7 Tooling Baseline

For new standalone examples, prefer a current toolchain such as Vite, functional components and hooks. Do not present `create-react-app` or class components as the default for new development. Existing projects may still use older structures; document them as compatibility cases.

18.8 Boundary with the logiCells Guide

Use the official React documentation for JavaScript, JSX, props, state and hooks. Use this guide for the logiCells-specific concerns:

- generated metadata and `dataStructure`;
- component registration;
- `PageController` and local event dispatch;
- `DataSource/DataSet/DataCursor` binding;

- controller and ObjectBehind integration;
- packaging the web kernel and generated application metadata.

Chapter 19

Dynamic Forms, Lists and Dashboards

Dynamic interfaces in logiCells are built from declared views, datasets, cursors, functional components and event reactions. Forms, lists and dashboards are not hand-coded islands. They are runtime projections of conceptual models and assemblies of UXfc functional components.

19.1 Dynamic Forms

A dynamic form binds roles, actions and related collections to a current object item. The form receives its context from a cursor and displays roles through functional components such as edit fields, labels, selection components, check fields, buttons, panels or specialized components.

The form should be understood as an interaction surface over a concept. If the concept changes, the form can evolve by changing the declaration, not by rewriting the entire application flow.

19.2 Selection Components

Selection is a common UXfc intention. A selection component should not be reduced to a combo box. Its function is to let the user identify an element from a candidate set and propagate the selected value into the current context.

The component uses a main data source and a selection data source. It can emit a semantic change event that a page model or controller can handle.

19.3 Lists, Tabular Views and Grids

Lists and grids usually bind to class-level collections or object-level joins. A grid does not own the query. It consumes a collection projection provided by the model.

A tabular view may support richer behavior than a simple grid: editable cells, formatting, formulas, post events, submit events, checked-state events or bulk data operations. The model still declares the functional need; the registered implementation provides the concrete capabilities.

19.4 Containers: Split, Tabbed and Reactive Frames

Containers assemble other functional components. A split view organizes several subframes in parallel. A tabbed view or stack group organizes subframes sequentially. A reactive desktop frame can host child views and switch them according to navigation or process events.

This is an assembly of functions. The actual visual appearance depends on registered components, style and platform.

19.5 Dashboards as Assemblies

A dashboard is a composition of projections and functions. It may combine lists, counters, status panels, charts, process states, recent messages and actions. The important rule is that each part should remain connected to a declared concept, collection, action or service.

```
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
```

Listing 19.1: Dashboard as UXfc assembly

A dashboard should not become an undocumented procedural aggregate. Each component should have a clear data source, event behavior and functional intention.

19.6 Navigation as Event Flow

Navigation is also declarative. A button may emit `Navigation.Next`. A tabbed or stack component may move to the next frame. A header component may react to a title change event. A process controller may show a new view when a process reaches a state.

This pattern separates user interaction from process semantics. The same process can be displayed differently on another platform while preserving the underlying coordination.

19.7 Functional Component Registration

A functional component expresses what role an element plays in the interface. The runtime maps that role to a concrete component through registration. This makes the visual layer portable and extensible.

For example, a declarative `listView` function may be implemented by a standard list, a tabular view or a mobile collection. A `tabbedView` function may be implemented by tabs, a stack group or a platform-specific wizard. The model declares the function; the platform chooses the registered implementation.

19.8 Abstract Versus Concrete Components

Abstract components describe interaction contracts. They are not directly instantiated. Concrete components implement those contracts.

This separation is useful in dashboards. A dashboard may declare a data visualization or exploration function without forcing all platforms to use the same renderer. One platform may register a grid implementation; another may register a chart implementation; another may register a graph navigation component.

19.9 Design Rules

Bind components to declared datasets, not to hidden queries. Use collections and joins to prepare the data shape before the view consumes it.

Use UXfc intentions to describe why a component exists. Use functional components to describe what interaction function is needed. Use registration to select concrete implementations. Keep appearance, style and product identity outside reusable architectural examples.

Keep navigation events separate from business events. The user may click **Next**, but the process model should decide what that means.

Chapter 20

Functional Components, Concrete Implementations and Visualizations

logiCells separates the functional meaning of a visual component from its concrete implementation. This is a central design principle of the publication layer. A model declares that a view needs a function: selection, consultation, tabular display, navigation, validation, status display, document viewing, data visualization or another interaction capability. The effective component is selected later by registration.

20.1 Three Layers

The component stack has three different layers:

```
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
```

Listing 20.1: Component stack

A style, a skin, an icon set or a visual layout is not the functional component. It is an appearance choice applied after the functional and concrete layers have been resolved.

20.2 Functional Component Declaration

A functional component is declared in the view using a logical tag. The tag expresses a function, not a toolkit dependency.

This declaration says that the view needs a selection function with autocompletion. It does not say how the component is painted internally.

20.3 Registration

A concrete implementation is registered by associating a namespace, a target family, a logical component name, a view name, a runtime class and a controller.

```
1 RegisterVisualFunctionalComponentClass(
2   'www.logicells.net',
3   'visual/desktop',
4   'listView',
```

```
5 'TabularView',  
6 'rich tabular data component',  
7 TabularView,  
8 'Controllers.Desktop.ListViewControl');
```

Listing 20.2: Registration of a concrete functional component

The logical name is the stable contract used by the declaration. The view name and runtime class identify one concrete implementation.

20.4 Equivalent Implementations

A functional component can have several equivalent implementations. This is a feature, not a problem.

```
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
```

Listing 20.3: Equivalent concrete implementations

The model should rely on the function. The target platform or product package selects the registered implementation.

20.5 Available Functional Families

The current documentation gives several important families that should be represented in the guide:

- selection components, such as autocompletion over a selection data source;
- boolean or state edition components, such as check fields;
- consultation components, such as document, browser or remote view consultation;
- structural containers, such as split views and tabbed views;
- status views for logs, progress and processing state;
- tabular views for rich dataset display and edition;
- action components, such as buttons that dispatch semantic visual events;
- reactive frames that host and switch child views.

These families should be described as reusable interaction functions, not as product screens.

20.6 Abstract Components

An abstract component defines a function that cannot be instantiated directly. It is a contract. Concrete components implement the contract.

For example, an abstract exploration component may specify that the user explores a graph, hierarchy or collection through a cursor and related datasets. It may define expected inputs and events, but it does not impose a tree, a graph canvas or a list. Those are concrete incarnations.

20.7 Events and Actions

Functional components participate in event routing. A button emits a visual action event. A tabular view may emit submit and post events. A selection component emits a change event. A consultation component may emit resource loading and navigation events.

Actions should remain functional. A component should expose actions such as **ShowMessage**, **MoveTo**, **OpenResource**, **SubmitData** or **UpdateContext**. It should avoid actions that reveal an internal widget structure such as **SetEditBoxContent** unless the component is intentionally low-level.

20.8 Visualization as a Function

A visualization is not automatically a chart. It is a functional requirement to make a structure understandable. The same visualization function can be implemented by a grid, chart, timeline, tree, graph navigator or specialized canvas.

The model should describe the data source, the roles being visualized, the event contract and the constraints of interpretation. The registered implementation provides the drawing strategy.

20.9 Design Rules

Declare the intention first. Declare the functional component second. Register the concrete implementation third. Apply appearance last.

Keep component names generic and reusable. Do not encode product identity or client-specific appearance in the public model. Prefer component names such as **Selection**, **Consultation**, **TabularView**, **ProcessStepFrame** and **StatusView**.

Use the registration mechanism to adapt to desktop, web, mobile or other publication targets. Do not rewrite conceptual models when the renderer changes.

Chapter 21

Web Publication from Conceptual Views

logiCells views are conceptual projections. When a view is published on the web, the publication should preserve that conceptual structure instead of turning the view into an unrelated page.

21.1 Views as Published Projections

A view defines what part of a concept is exposed, which datasets are used, which actions are available, and which functional components render the interaction. A web publication adapter can map this view to a web page or component tree.

21.2 Functional Components

The declarative view should describe functional intent: list, form, grid, button, field, dashboard, signature, chart, navigation, conversation. The concrete web implementation is selected through registration. This makes it possible to render the same conceptual view differently on desktop, mobile or web.

21.3 Actions in Published Views

A published view should expose only actions that make sense in the current context. Some actions may be visible to users, some available through services, some reserved for processes or agents. View publication should respect the same model-level contracts.

21.4 Example

The path identifies the declarative view resource. The web runtime decides which registered concrete components render the functional elements inside the frame.

21.5 Design Rules

Do not encode platform-specific appearance in the conceptual model. Keep view structure declarative. Use functional components to express intent. Register concrete implementations per platform. Publish actions intentionally.

21.6 Summary

Web publication is not a separate application layer. It is a publication mode for conceptual views and model capabilities. When the publication target is a reactive web runtime, concrete components should follow the web functional component pattern: bind to page-level data surfaces, dispatch local events through the page controller, and cross the provider or service boundary only through explicit page-model mappings.

Part V

Synchronization and External Boundaries

Chapter 22

Local-first Conceptual Applications

logiCells View applications can keep client interaction responsive while some data is synchronized with remote providers. The important separation is between *local interaction state* and *remote synchronization state*. The client can continue to navigate cached or locally available projections without turning every interaction into a network request.

22.1 What “Local-first” Means Here

In this guide, local-first does not mean that every application owns a complete authoritative database on the client, nor does it imply CRDT-style conflict-free merging. It means that the View architecture can keep local DataSets/DataCursors, local graph fragments and local visual state, while synchronization is handled through explicit providers and synchronizer contracts.

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

Listing 22.1: Local-first application chain

The user interacts with the local projection. A network boundary is crossed only when the application maps the interaction to a provider refresh, business action, process transition or synchronization operation.

22.2 Local Cache, Graph Fragments and Snapshots

A client may keep several local resources.

Resource	Role
Local cache	Stores view-ready data, lookup lists, recently used objects and temporary results.
Local graph fragment	Holds the conceptual objects needed by the current workspace or device.
Snapshot	Captures a coherent state that can be restored or compared according to the selected runtime/provider contract.
Synchronization repository / journal	Records synchronization state, positions and pending exchanges where the selected synchronizer uses those mechanisms.
Provider state	Tracks remote cursors, pages, timestamps, tokens or continuation markers.

These resources are operational projections. They should remain attached to conceptual boundaries such as collections, actions, events, processes and identities.

22.3 Local Events and Remote Escalation

Local-first behavior begins with event classification.

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

Listing 22.2: Local-first event routing

A row selection, tab change or cursor movement can remain local. A validation, business transition or persisted modification can be mapped to an action, process event or synchronization operation.

22.4 What the Runtime Synchronizer Actually Tracks

logiCells provides a synchronization layer separate from ordinary messaging. Synchronization scenarios use explicit registration, read/write exchanges, acknowledgements, journal or clock positions, and routing rules. Distributed hypergraph applications can also use causal positions associated with named clocks and node identifiers.

This gives the application concrete concepts for answering questions such as:

- which node/site is registered for a synchronization group;
- which journal/clock position a participant has reached;
- which changes should be routed to which target;
- when a read/write exchange has been acknowledged;
- how synchronization state is separated from the transport carrying the messages.

Transport and synchronization semantics are separate. A transport such as ZeroMQ can carry synchronization messages, but it does not define the meaning of the synchronization itself.

22.5 Remote Providers as Synchronization Boundaries

Remote providers should be treated as explicit boundaries rather than as the direct data source of every visual component.

```
1 Remote provider
2   -> fetch / write / synchronize
3   -> resolve identities and positions
4   -> update local projection
5   -> emit provider/synchronization state
6   -> expose data to DataSet/DataCursor
```

Listing 22.3: Provider as synchronization boundary

This avoids coupling UI rendering directly to network latency and allows the same View to be exercised against local fixtures or remote providers.

22.6 Offline Work and Conflict Semantics

The documentation must not promise more than the selected synchronizer/provider guarantees. In particular, “local-first” does not by itself answer:

- whether writes are accepted while fully disconnected;
- how two concurrent edits of the same conceptual object are reconciled;
- whether ordering is total, causal or provider-specific;
- which side is authoritative after reconnection;
- whether a multi-object business action is atomic across sites;
- how permissions or model-version changes are revalidated after reconnect.

For an application that depends on these behaviors, document the exact provider/synchronizer contract and verify it with an end-to-end scenario. Do not substitute a generic “merge” claim for a tested conflict policy.

22.7 Typical Handling by Operation Type

Operation	Typical handling
Cursor movement, tab selection, filter expansion	Local visual event.
Search request, lazy load, page fetch	Provider request, possibly remote.
Edit draft, temporary selection, local annotation	Local cache or local graph fragment, subject to the application’s persistence policy.
Business validation, workflow transition	Conceptual action or process event; may require an authoritative runtime confirmation.
Replication / synchronization	Explicit synchronizer/provider exchange with observable state.

22.8 Design Rules

Expose local data through DataSets, DataSources and cursors. Put remote access behind providers. Keep visual events local unless they are explicitly mapped to conceptual actions or process events. Treat synchronization state as observable runtime state. For offline/concurrent-write scenarios, publish the exact guarantees and failure states instead of relying on the label “local-first” alone.

Part VI

Engineering and Diagnostics

Chapter 23

Rich-Client Engineering Boundaries

Client diagnostics should first identify which boundary failed: View loading, MODEL-behind behavior, DataSet/DataCursor state, event routing, asynchronous exchange, synchronization, or the underlying runtime. General transactions, service publication, ABI and runtime diagnostics are documented once in the Developer Guide.

23.1 Diagnose the Client Boundary First

For a View-specific defect, record:

- the View package and projection being rendered;
- the MODEL-behind declaration involved;
- the DataSet/DataCursor state and current identity;
- the triggering visual/functional event;
- whether the failure occurred before or after a provider or remote exchange;
- the lifecycle state of the page/controller when an asynchronous continuation returned.

This keeps client diagnosis separate from business-model diagnosis and avoids duplicating the general troubleshooting reference.

23.2 Client Lifetime Is Part of Correctness

A cached client has several overlapping lifetimes: page/view, DataSet/DataCursor, provider request, synchronization worker and event subscription. Do not allow a background result to update a page that has already been closed, or an event source to retain a destroyed controller.

A safe page/client shutdown normally follows this direction:

1. stop accepting new UI work for the page;
2. unsubscribe visual/provider events;
3. cancel or detach pending asynchronous requests according to their contract;
4. stop page-owned timers and background producers;
5. close page-owned cursors/providers that are not shared;
6. dispose the View/controller hierarchy.

Shared application services and synchronizers have a longer lifetime and are stopped by the application runtime, not by each View.

23.3 Platform and Toolkit Boundaries

The runtime is available on Windows, Linux and macOS and can run in containers. A specific rich-client implementation can have a narrower support matrix. A native widget toolkit, React adapter, browser feature or synchronization transport must therefore document its own platform constraints rather than inheriting a blanket claim from the runtime.

23.4 Escalate to the Developer Guide

Move from client diagnosis to core-runtime diagnosis when the same failure is reproducible without the View layer, for example through a CLI/API invocation of the same action or query. That is a strong signal that the defect belongs to the model, transaction, provider, service or runtime boundary rather than the client projection.

23.5 Runtime Evolution and View Invalidation

A changed source unit is not a reason to refresh every view. The native incremental planner tracks separate *Structural*, *Semantic*, *Behavior*, *State*, *Persistence* and *View* signatures and typed dependents. A view should be invalidated by its view signature or by a dependency it actually consumes. This is an architectural rule for future view-aware hot reload, not a claim that every current desktop/web client already performs fine-grained live replacement. The 11 October v37 evidence validates cold metadata startup on a reference fixture but leaves semantic graph delta, per-view notifications and warm security binding open.

For embedded FunctionalP Actions declared in YAML, the VS Code v1.0 source debugger requires source provenance, runtime `ScriptIdentity` and generation-aware breakpoints. The existing runtime AST debugger is a prerequisite, not proof of completed YAML-to-AST source mapping in a rich client. For exact limits and CLI workflows, use the companion Developer Guide.

Part VII

Reference Patterns

Chapter 24

Rich-Client Glossary

View A client projection of conceptual state and capabilities; not the business model itself.

MODEL-behind Declarative client-side behavior associated with a View.

ObjectBehind Runtime client object that connects a visual component or View to declared behavior.

DataSet A local tabular projection used by the client as working state or cache.

DataCursor A navigation context over one or more DataSets.

DataSource A binding surface through which visual controls consume DataSet state.

PageControler Historical public spelling of the page-controller access point used by existing View scripts. This guide uses that spelling consistently because it is part of the current public contract.

Visual event A client interaction signal routed inside the View architecture before, or instead of, becoming a business action.

View package Packaging unit that assembles View sources and their client-side declarations.

Local cache Client-side projected state that allows responsive interaction without redefining the authoritative conceptual model.

Chapter 25

Conceptual View Patterns

25.1 Scope

- object view;
- class view;
- list view;
- dashboard view;
- form view;
- web published view;

25.2 View as Projection

A view projects a concept. It does not own the concept. The view may define datasets, cursors and functional components, but the durable structure remains in the model.

25.3 Class and Object Views

A `ClassItem` view presents a collection, a search result, a dashboard or a class-level action. An `ObjectItem` view presents one conceptual object and its related roles, actions or child collections.

25.4 Functional Components

The view can request a functional component such as a list, grid, form, field, button or signature area. The concrete component is selected by registration. This separates the declarative function from the toolkit-specific implementation and from the final visual appearance.

25.5 UXfc Assembly Pattern

A view can also be read as an assembly of UXfc functions. A review view may combine consultation, selection, tabular display and validation. The view declaration names the functional components; the runtime resolves concrete implementations through registration.

25.6 Functional Component Families

Common functional families include consultation, exploration, selection, edition, validation, navigation, status display, tabular display, split layout and sequential layout. These families describe user activity and interaction function, not a fixed visual appearance.