**OmniStudio Best Practices**

**Think of simplicity and reusability.**

Configure and build with standard, out-of-the-box (OOTB) Salesforce features whenever possible, and reuse components as often as possible. Consider what you already have to meet use case requirements. Remember to understand the scale of the function you’re building—will it be complex but rarely used, or does it execute thousands of times an hour? Solutions must scale into production quantities and work well in a development environment.

For example, don’t build a different DataRaptor or Integration Procedure for every OmniScript or FlexCard. When you build a DataRaptor or Integration Procedure, consider keeping it as generic as possible, so you might reuse it later.

* Consider using a Set Value ahead of the DataRaptor to change parameters and thus allow you to use the DataRaptor again.
* You don’t always need to create a DataRaptor to match the way an OmniScript has already been built. What if you included the data structure in the OmniScript to match what the DataRaptor is expecting?

**Consider This**

Can you manipulate the data before invoking the DataRaptor or Integration Procedure, so you can reuse an existing component instead of building a new one to complete the same task?

It all depends on the use case and how the DataRaptor or Integration Procedure is invoked. While you don't want to create more components than necessary, it’s better to have more efficiently executing DataRaptors or Integration Procedures than fewer, poorly executed ones.

# DataRaptor Best Practices

* Create targeted DataRaptors to only extract or load the data needed for one operation.
* Use relationship notation (\_\_r) or queries whenever possible to pull data from other SObjects.
  + Using relationship notation improves the performance of DataRaptors retrieving data from a parent of the primary sObject. Use this notation in the Extract JSON Paths for the parent sObject's fields instead of adding a second sObject in the DataRaptor's Extract tab.
  + Relationship notation in DataRaptors is based on relationship queries in Salesforce.
  + DataRaptors support only child-to-parent relationship queries.
* Ensure all filtering and sorting (Order By) operations are on indexed fields. The ID and Name fields are always indexed.
* Use caching to store frequently accessed, infrequently updated data. This saves round trips to the database and improves performance.

# Integration Procedure Best Practices

* Use Integration Procedures for all data calls to Salesforce.
* Use the Send/Response Transformations in the Response Action for Integration Procedures to trim the data and only return what’s needed.
* Similar to trimming Response Action data returned, consider trimming the data sent to the Integration Procedure. Resist sending the entire OmniScript Data JSON.
* Use multiple Response Actions with different Execution Conditional Formulas to allow an Integration Procedure to exit early under appropriate conditions.
* Just as recommended with DataRaptors, use caching to store frequently accessed, infrequently updated data.
* bullet

To run operations asynchronously, call Integration Procedures using these settings:

* + When calling an OmniScript or Integration Procedure that doesn’t need a response and completion time isn’t critical, select Remote Options and enter useFuture: as a key and true as its value.
  + Use the Invoke Mode option, Fire and Forget instead of useFuture when calling an OmniScript that must invoke the Integration Procedure immediately.
  + The Invoke Mode option, Non-Blocking runs the Integration Procedure immediately while continuing the user interaction of the calling OmniScript. A response is returned when the Integration Procedure is complete.

# FlexCard Best Practices

**Styling of FlexCards**

Avoid manual styling of individual FlexCards to avoid creating too many unnecessary FlexCards. This impacts performance.

* Use conditional styling and FlexCard states.
* Use global Cascading Style Sheets (CSS) styling.
* Arrange data in a way the end user understands it rather than based on an object. For example, My Policies or recent transactions might come from multiple objects but make sense together for the end user.
* Always think of ways to reuse layouts elsewhere; they should be self-sufficient components unless the business use case doesn’t allow for this.
* Use session variables to store specific variables and set default values. They can be set with values from a data source or with hard-coded values.
* Avoid too many levels of nested layouts, as it decreases performance overall.
* When creating a complex application, break the app into components by creating a FlexCard for each component. Then, place these into a layout that serves as a container. As a result, the application quickly loads by referencing only the single ‘container’ layout.
  + Use the event framework to communicate between different FlexCards. With events you can:
    - Refresh data sources.
    - Update data sources.
    - Display or hide FlexCards.
    - Set records.
    - Set loading status.

**It’s also important to use meaningful names for FlexCards.**

Title case and separate words with a space, like this:

{Prefix {Object} Desc}

or use lowercase and separate words with an underscore like this:

{prefix}\_{obj}\_{description}

**Check out these examples of FlexCard names.**

* teamGetAccount
* team\_get\_account

**FlexCard Data Source Tips**

* Use Integration Procedures over all other data sources when handling complex tasks since it allows for the most flexibility in retrieving and handling data.
* To use your Apex Class in Apex Remote or Dual data sources, it must implement the Vlocity Open Interface.
* When exporting layouts and FlexCards with a data source, copy over the result of the data source into the sample data source so there’s sample data available.
* If field-level security (FLS) is a concern, use a DataRaptor as a data source instead of SOQL, as it enforces FLS.
* Use the interval option to keep your data refreshed but avoid aggressive timings; anything below 1000 milliseconds (ms) should be used only in special scenarios.
* Use the timeout option to avoid keeping the UI waiting for data to be fetched after a certain period of time.
* When debugging the layout or FlexCard for performance, check the data source in the FlexCard Designer first and sample the time it takes to fetch the data. This is usually the biggest bottleneck.
* When using Apex Remote, turn on Async mode to increase the governor limits by calling the Apex class asynchronously.
* Use the Order By and Reverse Order options in the data source to order records on the client side.
* Use the Result JSON Path option to select a JSON node from the result. The layout or FlexCard only uses that node and discards the rest.
* Set header variables for a REST data source. However, when sending credentials it's better to use the Named Credentials option to avoid setting up sensitive information on the data source itself.
* To view historic information when using Streaming API set Get All Messages to True; this option returns the last 24 hours of messages for the channel.

**OmniScript Best Practices**

**BUSINESS PROCESS AND LOGIC**

**Ownership**

Use one owner for each OmniScript. An owner owns the logic and requirements.

**Reusability**

* Build a skeleton of the entire OmniScript to help identify reusable components.
* When processes are repeatable across multiple OmniScripts, create a reusable OmniScript and add it to the appropriate parent OmniScripts.

**OmniScript Designer**

Use the OmniScript Designer to build quick mockups of processes, starting with Steps and Integration Procedure placeholders for data in and out.

**Document**

Use the Internal Notes property to document the purpose and change the history of an element. This is especially useful for Formula or Action Elements.

**Naming or Renaming**

Do not use special characters in the type and subtype of OmniScripts and Integration Procedures and the name of DataRaptors. Newer versions of OmniStudio prevent this at object creation. However, older OmniStudio for Vlocity packages may still allow this naming pattern. Using special characters prohibits the activation of the Metadata API, requiring the renaming of the objects.

OmniScript and FlexCard author fields should not contain special characters, as it may prevent LWC deployment.

To ensure DataRaptors and Apex classes remain valid after script creation, don’t rename OmniScript elements. If the element name requires an update, apply the name changes to the DataRaptor or Apex class.

**ContextId**

Avoid assigning a ContextId within the OmniScript during an ongoing process. A reserved key, OmniScript's Context ID assigns a Record ID from the URL. Use Set Values to set other important non-context IDs.

**USER INTERFACE (UI)**

OmniScripts use Lightning Web Components (LWCs) to define the styling for both individual elements and the OmniScript itself.

**Extend LWCs**

Create custom LWCs to extend an OmniScript element's component to apply styling changes.

Extending an OmniScript element's LWC adds custom behavior and styling to an OmniScript Element while maintaining its core functionality.

LWCs in OmniScript provide the HyperText Markup Language (HTML), JavaScript, and CSS for an OmniScript element. For example, a Text element has an extendable component named OmniScriptText.

Since it’s best practice to design solutions with the minimum extension necessary, always look for a standard LWC or OmniScript component that meets the needs of the use case before deciding to write your own LWC. This also allows previewing without activation, which speeds up the development process.

**Global Branding**

Apply global style changes to OmniScripts by using the Newport Design System. The Newport Design System enables you to download, edit, and override the existing Newport CSS styling to apply your changes.

**USER EXPERIENCE (UX) DESIGN PRINCIPLES**

From the script developer’s standpoint, use standard naming conventions for ease of maintenance and optimize the script for best performance.

From the user’s standpoint, strive to maximize the clarity of the User Interface (UI) by choosing the right elements and providing descriptive, unambiguous text and an understandable logical structure. Remember, the end user is much more likely to finish a task if they know they’re doing it correctly.

|  |  |
| --- | --- |
| Prefill elements. | Minimize the amount of data your users must enter by using the context to prefill any available key information. Launch an OmniScript from a detail page, and use DataRaptor to prefill any field from the record into an OmniScript. |
| Remove extraneous information. | Whenever possible, remove optional information or tangential to your use case. |
| Be concise. | Break complex processes into shorter steps and minimize the number of elements. |
| Sequence. | The order of input fields should make sense for your use case. Use Block elements under steps only if required for your use case. |
| Guide the user. | Provide informative, actionable help text and feedback in the UI. |

**Design Collaboration**

We all want to create beautiful websites! When working with new OmniStudio customers, consider illustrating the full capability of OmniStudio with the user experience (UX) designer as soon as possible. Using the features and functions already available in OmniStudio reduces the need for time-consuming customization and is easier to maintain. If there’s already a design requiring multiple complex customizations in the works and OmniScripts could achieve the tasks more efficiently, provide the UX designer with some options to save time and improve maintainability. Collaborate and explain when particular designs might deviate from standard components and provide examples of ways to achieve the same results with a different approach, especially when there’s limited time to build. Work with the UX designer to weigh the benefits and time requirements of more custom design outcomes.

**PERFORMANCE FACTORS**

**Server-Side Performance**

| **Best Practices** | **Description** | **Impact** |
| --- | --- | --- |
| Reduce conditional views, and merge fields, and formulas. | Where possible, reduce these and implement logic from the server side. | Design |
| Trim JSON responses. | Use Send and Response transformations to trim API responses, which speeds up the application of the response across the OmniScript Data JSON. | Integration |
| Load from CMS. | Make sure the OmniOut application and JSON definitions are loaded from CMS. This improves the performance to retrieve the definitions and display the flow. | OmniOut |
| Remove spaces. | Remove spaces from all UI element names to improve OmniScript load time. | Load Time |
| Reduce elements. | Reduce the number of OmniScript elements. Stay below 200 elements by creating child OmniScripts. | Design |
| Run logic. | Run logic on the server where possible, including conditional logic in Integration Procedures and formulas in DataRaptors. | Integration |
| Enable time tracking. | Test performance by enabling time tracking using the OmniStudio Tracking Service. If time tracking isn’t used in production, disable the feature before deploying it to production. |  |

# Reusable OmniScripts

Build a variety of smaller scripts and then piece them together into one or more parent scripts.

**Here are some important things to know about reusable OmniScripts:**

* Parent OmniScripts may contain one or more embedded reusable OmniScripts.
* It’s possible to share reusable OmniScripts across multiple parents.
* Embedded OmniScripts behave just like other OmniScript elements.
* A reusable OmniScript can’t contain another reusable OmniScript, since only one level down is allowed—parent and child.
* No element in a reusable OmniScript can have the same name as an element in the parent script, including remote response data.
* Reusable OmniScripts adopt the script configuration of the parent script. The reusable OmniScript JSON is embedded inside the parent JSON and at compilation time there is no functional difference between the two.

**Follow these steps to embed an OmniScript in another OmniScript.**

* In the OmniScript being reused, click Setup, and check the Reusable checkbox.
* Activate the reusable OmniScript.
* In the OmniScript that hosts the reusable OmniScript, go to the Build panel and expand the OmniScripts section.
* Locate the reusable OmniScript and drag it into the canvas.
* Preview the OmniScript to test its behavior.

Embed active reusable OmniScripts in other OmniScripts. For example, for an OmniScript that updates a case, you may want to add the reusable script component at the end of an OmniScript form used for troubleshooting.

**Follow these steps to embed a reusable OmniScript in another form.**

* Expand the OmniScripts section of the Available Components.
* Drag the reusable OmniScript onto the Structure pane.

**Element Naming**

**Follow these naming conventions when creating an OmniStudio component.**

|  |  |
| --- | --- |
| Element names | Choose short, descriptive names. Avoid using spaces. |
| Capitalization | Use UpperCamel case.  No spaces, and avoid special characters. |
| Use standard names. | FirstName, LastName, MiddleName, Street, City, State, PostalCode, Email, Dob, Age, etc. |
| IDs | Specify the object. For example, AccountId, PolicyAssetId, Contact Id, or CaseId. |
| Use standard abbreviations. | wk (week)  wkly (weekly)  mos (months)  mnly (monthly)  yr (year)  yrly (yearly)  hrs (hours)  hh (household)  bene (beneficiary)  rel (relationship) |
| Same object, different purpose | Relate the element to the purpose.  For example, if an insurance script uses contact data for three different parties—the insured, claimants, and beneficiaries—assign element names that disambiguate them.   * InsuredStreet * ClaimantStreet * BeneficiaryStreet |
| Actions | Use VerbObjectDetails, and standardize the verbs you use.   * For REST actions, use Get and Post as verbs. * For DataRaptor actions, use Create, Read, and Update as verbs.   Other recommended verbs for action element names:  **Remote actions:** Invoke  **DocuSign envelope:** DocuSend  **DocuSign signature:** DocuSign  **Email:** Send  **Calculation:** Calc  **Set Values, Set Errors:**Set  **PDF:**PDFGen |

**DataRaptor vs. Integration Procedure**

**Let’s compare and contrast DataRaptors and Integration Procedures.**

DataRaptors read or write Salesforce sObject data or perform single-step data transformations.

Integration Procedures interact with many types of data and process it in multiple steps.

For some use cases, a single DataRaptor is sufficient. Integration Procedures usually call one or more DataRaptors and are more flexible and powerful.

Use these guidelines to determine which to build.

**Use a single DataRaptor when:**

1. You need to read data from sObjects **OR** write data to sObjects, but not both. DataRaptors go to the server. In a given DataRaptor that works with data, you can’t read and write data in the same call.
2. The sObjects you need to read from or write to have a defined relationship. For example, Accounts and Contacts have a relationship because a Contact can have an AccountId value.
3. You just need some data transformation. In this case, you only need to work with JSON or XML data. No sObjects are involved.
4. You can perform any needed filtering, calculation, or reformatting of data values using one or a series of formulas.
5. You can make any needed changes to the data structure by mapping input JSON nodes to output JSON nodes.

**Embed DataRaptor operations in an Integration Procedure when:**

1. You need to both read from **AND** write to one or more sObjects, which means you need to call at least two DataRaptors. If you add multiple DataRaptors, wrap them in an Integration Procedure to allow for only one roundtrip server call.
2. The sObjects you need to read from or write to have no defined relationship.
3. Transforming your data can’t be done using formulas alone. For example, different conditions determine whether some filtering or calculations are performed at all.
4. JSON node mappings aren’t straightforward or require a series of steps.
5. You need to read from or write to multiple data source types, including sObjects, CSV files, external objects, and Apex classes.
6. You need to perform actions, including calling a REST API, sending an email, merging lists, or handling errors.